EXPERT INSIGHT

Mastering TypeScript

Build enterprise-ready, modular web applications using TypeScript 4 and modern frameworks

Fourth Edition

Nathan Rozentals

Packt

Mastering TypeScript

Fourth Edition

Build enterprise-ready, modular web applications using TypeScript 4 and modern frameworks

Nathan Rozentals

Packt

BIRMINGHAM – MUMBAI

Mastering TypeScript

Fourth Edition

Producers: Aarthi Kumaraswamy, Caitlin Meadows
Acquisition Editor – Peer Reviews: Divya Mudaliar
Project Editor: Janice Gonsalves
Content Development Editor: Alex Patterson
Copy Editor: Safis Editing
Technical Editor: Aniket Shetty
Proofreader: Safis Editing
Indexer: Rekha Nair
Presentation Designer: Ganesh Bhadwalkar

First published: April 2015
Second edition: February 2017
Third edition: February 2019
Fourth edition: April 2021

Production reference: 1210421

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-80056-473-2

www.packt.com

Contributors

About the author

Nathan Rozentals has been writing commercial software for over 30 years in C, C++, Java, and C#.

He picked up TypeScript a week after its initial release in October 2012, and was one of the first people to start blogging about TypeScript, discussing early frameworks such as Backbone, Marionette, ExtJS, and AngularJS.

Nathan's TypeScript solutions now control user interfaces in IoT devices, run as stand-alone applications for point-of-sale solutions, provide complex application configuration for websites, and are used for mission-critical server APIs.

When he is not programming, he is thinking about programming. To stop thinking about programming, he goes windsurfing or plays soccer, or just watches the professionals play soccer. They are so much better at it than he is.

I would like to thank my partner, Kathy, for her unwavering support and love over the years. To Ayron and Daya, I am so proud of you both. To Matt, just keep being yourself.

To everyone at Vix, thanks for making work such a gratifying experience. Continue to challenge yourselves, continue to question, and continue to learn. You are all so highly talented, and are fantastic people to work with.

About the reviewer

Dan Vanderkam is the author of *Effective TypeScript* (O'Reilly 2019, now available in Polish!) and a principal software engineer at Sidewalk Labs. He previously worked on open source genome visualizations at Mt. Sinai's Icahn School of Medicine and on search features used by billions of users at Google (try "population of france" or "sunset nyc"). He has a long history of working on open source projects, including the popular dygraphs library and source-map-explorer, a tool for visualizing JavaScript code size. He is also a co-founder of the NYC TypeScript meetup and blogs at `effectivetypescript.com`. When he's not writing type-safe JavaScript, Dan enjoys playing bridge and rock climbing near his upstate New York home.

Table of Contents

Preface

The TypeScript language and compiler has been a huge success story since its release in late 2012. It quickly carved out a solid footprint in the JavaScript development community, and continues to go from strength to strength. The language has broken into top ten lists of both popular programming languages, and languages most sought after by companies looking to employ programmers.

Many large-scale JavaScript projects, realizing that their JavaScript code base is becoming unwieldy, have made the decision to switch their code base from JavaScript to TypeScript. In 2014, the Microsoft and Google teams announced that Angular 2.0 would be developed using TypeScript, thereby merging Google's AtScript language and Microsoft's TypeScript languages into one. Angular is now one of the top three most popular Single-Page Application frameworks, and has progressed to version 11.

This large-scale industry adoption of TypeScript shows the value of the language, the flexibility of the compiler, and the productivity gains that can be realized with its rich development toolset. On top of this industry support, the ECMAScript standards are being ratified and published on a yearly basis, therefore introducing new language elements quite rapidly. TypeScript provides a way to use features of these standards in our applications today.

Writing TypeScript applications has been made even more appealing with the large collection of declaration files that have been built by the TypeScript community. These declaration files seamlessly integrate a large range of existing JavaScript frameworks into the TypeScript development environment, bringing with them increased productivity, early error detection, and advanced IntelliSense features. Many JavaScript libraries are now automatically bundling TypeScript types into their releases, meaning that there is first-class support from the library developers, who recognize that a large portion of their user base is using TypeScript.

In the end, TypeScript generates JavaScript. This means that wherever we can use JavaScript, we can use TypeScript.

Who this book is for

Whether you are a developer wanting to learn TypeScript, or an experienced JavaScript or TypeScript developer wanting to take your skills to the next level, this book is for you.

It is a guide for beginners, and also provides practical insights and techniques for experienced JavaScript and TypeScript programmers. No prior knowledge of JavaScript is required, although some prior programming experience is assumed.

If you are keen to learn TypeScript, this book will give you all the necessary knowledge and skills to tackle any TypeScript project. It will also give you an understanding of which application frameworks are out there, and which one to choose for your next project.

What this book covers

Chapter 1, *Up and Running Quickly*, shows how to quickly set up a TypeScript development environment, and generate JavaScript. It also includes an introduction to the TypeScript language and its general syntax, and shows how to debug code and use 3rd party JavaScript libraries.

Chapter 2, *Exploring the Type System*, discusses all of the types used in the TypeScript language, and commonly used language features. It starts with primitive types and type aliases, moves on to object spread and tuples, and finally discusses the use of types in function signatures, callbacks, and overrides.

Chapter 3, *Interfaces, Classes, Inheritance, and Modules*, builds on the work from the previous chapter, and introduces the object-oriented concepts and capabilities of interfaces, classes, and inheritance. It then shows how to work with modules, which is how we arrange our code, but also how we consume 3rd party libraries.

Chapter 4, *Generics and Advanced Type Inference*, discusses the more advanced language feature of generics, before working through the concepts of advanced type inference using conditional types, type chaining, and type distribution.

Chapter 5, *Asynchronous Language Features*, walks the reader through asynchronous programming concepts, starting with callbacks, then on to Promises, and finally async await.

Chapter 6, *Decorators*, shows the reader how to build and use decorators, which are used within some of the popular Single-Page Application frameworks.

Chapter 7, *Integration with JavaScript*, takes a look at declaration files, how to find them, how to use them, and how to write them. It also discusses the various compiler options that assist with integrating JavaScript and TypeScript files within the same project.

Chapter 8, *Strict Compiler Options*, works through each of the strict set of compiler options available with the TypeScript compiler, and explains what errors these options will automatically pick up within our code.

Chapter 9, *Using Observables to Transform Data*, discusses the RxJS library, and how it uses the Observable pattern to handle streams of events. It discusses how to register for an event stream, how to process data as it flows through the stream, and how to transform this data and create new streams, using combinations of operators.

Chapter 10, *Test-Driven Development*, explores the Jest unit testing framework, and how to write tests to cover a multitude of test cases. It also discusses asynchronous testing techniques, tests that update the DOM, and how to run end-to-end tests against a running website.

Chapter 11, *Angular*, discusses the Angular SPA framework, including shared modules, Angular services and form-based input. It builds a website using Angular that combines all of these techniques, along with the Angular Material set of components.

Chapter 12, *React*, walks through setting up and building a React website using TypeScript and the JSX syntax. It discusses props, event handling, state, and how to use React forms, along with integration of the Material UI library for React.

Chapter 13, *Vue*, sets up and builds a Vue web-site using TypeScript and class syntax. It walks through Vue components, child components, props, and state, as well as computed properties and form handling, integrating with the Bootstrap Material UI library.

Chapter 14, *Node and Express*, builds an Express web server with a few lines of code running in Node. It then expands on this to build a functional multi-page Express application, including page redirects, form processing, and session variables.

Chapter 15, *An AWS Serverless API*, shows how to use Amazon Web Services libraries to build and deploy a full REST-based API using Serverless technologies, backed by a DynamoDB database.

Chapter 16, *Micro Front-Ends*, discusses the concepts around micro front-ends, and how multiple front-ends can communicate with each other. It then goes on to combine the web sites built within the book using Angular, React, and Vue into a single application using these micro front-end techniques, integrating the REST API built using AWS.

To get the most out of this book

- You will need the TypeScript compiler and an editor of some sort. The TypeScript compiler is available on Windows, MacOS, and Linux as a Node plugin. *Chapter 1, Up and Running Quickly*, describes the setup of a development environment
- This book has been designed in a "type as you read" format. The code snippets are small, so feel free to type out the code yourself, as the text progresses, and learn by building up your code base and knowledge as you go.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-TypeScript-Fourth-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800564732_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "We can configure webpack using the `webpack.config.js` file."

A block of code is set as follows:

```
class MyClass {
    add(x: number, y: number) {
      return x + y;
   }
}
```

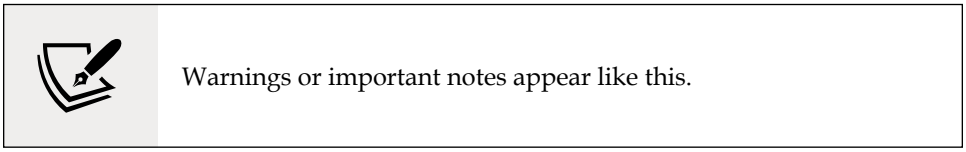
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
const modulus: Observable<number> = emitter.pipe(
    map((value: number) => {
        console.log(`received : ${value}`);
        return value % 2;
    }));
```

Any command-line input or output is written as follows:

```
npm install @types/express
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **Run** from the **Debug** panel."



Warnings or important notes appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1

Up and Running Quickly

So, what exactly is TypeScript? And how can we use it? Well, it really is as simple as "TypeScript generates JavaScript", to quote Anders Hejlsberg, the designer of the TypeScript language. Wherever we see a use for JavaScript in the modern world, we can generate this JavaScript using TypeScript.

JavaScript is being used everywhere – the more you look, the more JavaScript you find running in the most unlikely of places. Just about every website that you visit is using JavaScript to make the site more responsive, more readable, or more attractive to use. The power and flexibility of JavaScript also means that more and more tools are moving on-line. Where we once needed to download and install a program to write a document, or draw a diagram, we can now do this all from within the confines of our humble web browser.

The popularity and simplicity of JavaScript has inspired a whole host of runtime environments that can now run on a server, through Node.js (or simply Node), in the cloud through serverless technologies, or even on embedded devices with purpose-built microcontroller chips. Apache Cordova is a fully fledged web server that runs as a native mobile phone application, meaning that JavaScript, HTML, and CSS can be used to create mobile applications. Even desktop applications can be built using JavaScript, such as the popular editors Visual Studio Code and Atom. These desktop projects use the Electron framework and, as such, are fully fledged desktop applications that can run on Windows, macOS, or Linux.

Building JavaScript applications, therefore, means that what you build could be run on just about any operating system, use any architectural framework, or be used in any embedded device.

It is in the building of these JavaScript applications where TypeScript comes into its own. JavaScript is an interpreted language, and as such has benefits, but also has its drawbacks. Interpreted languages do not have a compilation step, and therefore cannot check that all code written has no minor mistakes in spelling or syntax before it is actually run. TypeScript is a strongly typed, object-oriented language that uses a compiler to generate JavaScript. The compiler will identify errors within the code base even before it is run in an interpreter.

This chapter is an introduction to the TypeScript language, and will focus on getting up and running in order to generate JavaScript as quickly as possible. We will be comparing some JavaScript code to the equivalent TypeScript code in this chapter, in order to explain why writing JavaScript using TypeScript is so beneficial. Your JavaScript code will become more robust, be less prone to errors, easier to read, easier to maintain, and easier to refactor.

We will cover the following topics in this chapter:

- Setting up a simple TypeScript IDE
- Generating different versions of JavaScript
- Configuring a TypeScript project
- Type syntax and basic types
- Function signatures
- Using third-party libraries
- An introduction to declaration files

Let's dive right in and set up a TypeScript development environment.

A simple TypeScript IDE

TypeScript generates JavaScript through what is called a compilation step. This means that once you have written some TypeScript code, you will need to compile, or, more correctly, transpile this code, which will generate JavaScript. In order to do this compilation step, you will need a Node environment, and the TypeScript compiler itself. In this section of the chapter, we will explore the setup of a simple TypeScript IDE.

Node is a JavaScript runtime environment. Similar to how a web browser can interpret and run JavaScript code, Node can do the same thing. It is a command-line driven environment, which means that all you need is a standard command prompt in order to run JavaScript.

Installing Node is as simple as downloading the Node installer package from the Node website (`https://nodejs.org`) for your operating system of choice, and running it. As it is an open source package, so you can also download the source code, and compile it from scratch on your system if you are so inclined. We will not run through the installation process here, and will simply assume that the Node packages have been correctly installed on your operating system. To verify that Node is installed, open up a command prompt and type:

```
node --version
```

If Node has been installed correctly, you should see something similar to the following output on the console:

```
v15.12.0
```

Here, we can see that Node has output its version number to the console, which, as of the time of writing, is version `v15.12.0`.

Using npm

Node includes a command-line program called the **Node Package Manager** (**npm**), in order to download, and make available a whole host of runtime libraries, of which TypeScript is one. These packages can either be installed globally, and therefore usable throughout your system, or can be installed locally. Locally installed Node packages will be installed within a sub-directory named `node_modules`, and globally installed Node packages will be installed in the system-wide `node_modules` directory. To check that npm has been installed correctly, type the following in the command prompt:

```
npm --version
```

The output of this command should be similar to the following:

```
7.6.3
```

Here, we can see that the npm version on our system is at version `7.6.3`.

To install the TypeScript compiler globally, type the following in your command prompt:

```
npm install -g typescript
```

Here, we are installing the `typescript` Node package globally, using the `-g` option. Although the Node package is named `typescript`, it makes a Node executable available named `tsc` that will invoke the TypeScript compiler. To verify that TypeScript has been installed correctly, type the following into the command prompt:

```
tsc --version
```

Here, we are invoking the TypeScript command-line compiler (`tsc`), and using the `--version` argument to report the current version of the compiler. The output should be similar to the following:

```
Version 4.2.3
```

Here, we can see that the version of TypeScript installed on our system, and used throughout this book, is version `4.2.3`.

Hello TypeScript

Let's dive right in and generate some JavaScript from a TypeScript file. In order to do this, we will need a text editor, and none fits the bill better than Microsoft's Visual Studio Code, or simply VS Code. VS Code was built using TypeScript, and was built for editing TypeScript. It has the ability to use third-party plugins, and can therefore be used as an IDE for a variety of programming languages, including Java, C, and C++. It is free to use, and uses the Electron framework, meaning that it can be run on Windows, Linux or macOS. Installation is as simple as downloading the installer from `https://code.visualstudio.com/download` and running it.

Assuming that you have successfully installed VS Code, we can set up a new project directory from a command prompt, and launch VS Code as follows:

```
mkdir ch01
cd ch01
touch hello_typescript.ts
code .
```

Here, we have created a new directory named `ch01`, changed into that directory, and created a new file named `hello_typescript.ts`. We have then launched VS Code with the command `code .`. The dot in this command refers to the current directory. This will launch VS Code as follows:

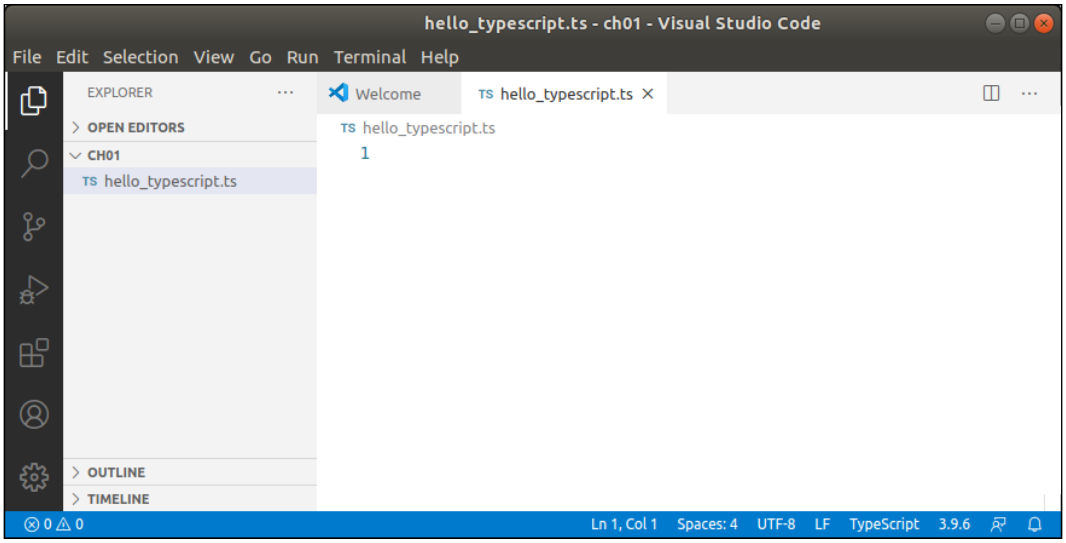


Figure 1.1: The Visual Studio Code editing window

The layout of VS Code is very similar to other text editors, with an Explorer window on the left, and a Source Code view on the right. Let's now write some TypeScript in the `hello_typescript.ts` file, as follows:

```
console.log(`hello TypeScript`);
```

This single line of code is the equivalent to the "Hello World" example that you will find in any other textbook on any other language. We are calling the `log` function on a globally available object named `console`, and giving it a string value of `` `hello TypeScript` ``. We can now generate a JavaScript file from this TypeScript file by executing the `tsc` command from the console as follows:

```
tsc hello_typescript.ts
```

Here, we are invoking the TypeScript compiler with a single command-line argument, which is the name of the TypeScript file we wish to generate JavaScript from. This will now generate a new JavaScript file named `hello_typescript.js`. We can execute this JavaScript using Node with the following command:

```
node hello_typescript.js
```

Here, we have invoked the Node runtime environment, and asked it to execute the JavaScript it finds in the file `hello_typescript.js`. The output of this command will be:

```
hello TypeScript
```

And that's it. It's as simple as creating a TypeScript file, generating JavaScript from it using the TypeScript compiler (`tsc`), and executing the resulting JavaScript using Node.

Now that we have an IDE set up, and can compile JavaScript from TypeScript, let's take a look at how we can generate different versions of JavaScript from the same TypeScript source files.

Template strings and JavaScript versions

In order to introduce how the TypeScript compiler can make our JavaScript development easier, let's take a quick look at something called template strings, or template literals. In the previous code snippet, you may have noticed that we used the backtick (`) as a string delimiter for the text `` `hello TypeScript` ``. If we take a look at the generated JavaScript file (`hello_typescript.js`), we will see that the TypeScript compiler has modified this line of code to use double quotes, as follows:

```
console.log("hello TypeScript");
```

The use of the backtick (`) to delineate strings gives us the ability to inject values directly into the string, as follows:

```
var version = `es6`;
console.log(`hello ${version} TypeScript`);
```

Here, we have declared a local variable named `version`, which holds a string with the value of `` `es6` ``. We have then modified the string that we are printing on the console to inject the value of the `version` string into the middle of the output by using the `${ ... variable name ... }` syntax. If we compile this file, and take a look at the generated JavaScript, we will see that this now becomes:

```
var version = "es6";
console.log("hello " + version + " TypeScript");
```

Here, we can see that TypeScript has interpreted our template string, and generated the equivalent JavaScript, which uses double quotes (") to surround each string, and the plus sign (+) to concatenate strings.

The use of template strings is actually valid JavaScript, but there is a significant catch to using them. Template strings have only been introduced into later versions of the JavaScript standard, in particular ES2015, also known as the 6th Edition, or just ES6. This means that you need an ES6-capable JavaScript runtime in order to use template strings. TypeScript, however, will generate JavaScript based on the version of the JavaScript runtime you are targeting.

If you are a little confused about JavaScript versions, don't worry, it is still a hotly debated topic. To simplify things, just bear in mind that any internet browser out in the wild will support at least ES3, or, to give it its full name, ECMA-262, 3rd Edition, which was published in 1999, 22 years ago.

As JavaScript is so widely used, its language features are published as a global standard, which is called the ECMAScript Standard. This standard has, at least historically, taken an extremely long time to update. After ES3 was published in 1999, it took a further 10 years to publish the next version, ES5, and a further 6 years to publish ES6. Since the 6th Edition was published in 2015, however, new standards have been published on a yearly basis.

Just bear in mind, however, that even though a new standard has been published, this does not mean that all internet browsers, or, more correctly, all JavaScript runtimes will support the new standard immediately.

Most modern internet browsers now support the ES5 standard. Unfortunately, we will have to wait for a number of years before we can say that most modern internet browsers support the ES6 standard. This means that we must be mindful of our target audience, and target runtime, if we attempt to use any ES6 features, such as template strings, within code that will be running within a browser. Older browsers, which could be found on older mobile phones, for example, will support at least the ES3 standard.

As mentioned earlier, TypeScript will generate JavaScript based on the target version of JavaScript that you specify. We can therefore write TypeScript code using any of the newest JavaScript standards, and TypeScript will correctly generate the equivalent JavaScript. Let's explore this concept a little further.

TypeScript project configuration

TypeScript uses a configuration file, named `tsconfig.json`, that holds a number of compilation options. We can use the `tsc` command-line compiler to generate a `tsconfig.json` file by using the `--init` command, as follows:

```
tsc --init
```

This `--init` option will automatically generate a `tsconfig.json` file within the current directory, which is used, amongst other things, to specify the target JavaScript version. Let's take a quick look at this file, as follows:

```
{
    "compilerOptions": {
        "target": "ES3",
        "module": "commonjs",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    }
}
```

Here, we can see that the `tsconfig.json` file uses standard JSON syntax to specify an object that has one main property, named `compilerOptions`, and within this main property, a number of sub-properties named `target`, `module`, `strict`, and so on. The `target` property is used to specify the JavaScript version that we would like to generate for, and its value can be one of a number of options. If we double-click on the `ES3` value, hit the *Delete* key, and then hit *Ctrl+Space*, VS Code will show us the valid values that can be used for this property as follows:

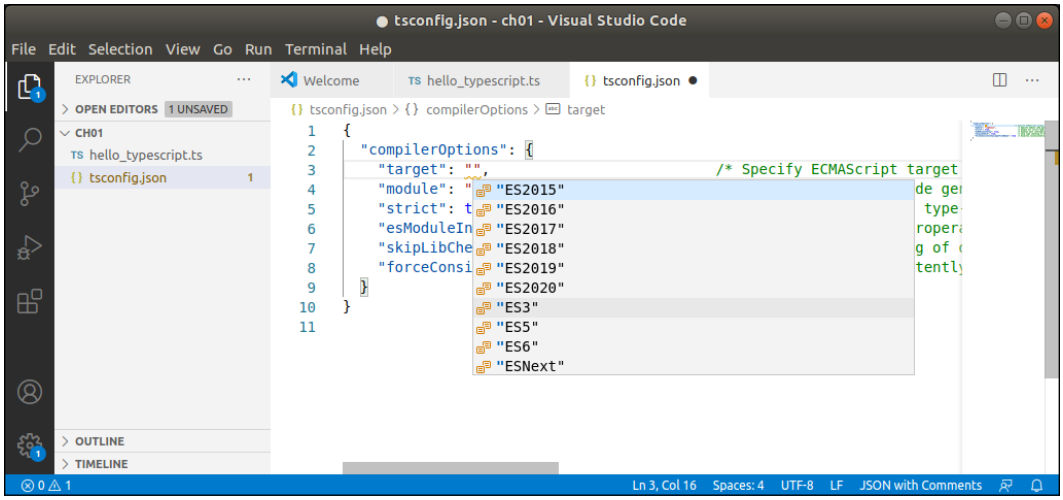


Figure 1.2: Visual Studio Code showing the available options for the JavaScript target version

Here, we can see that VS Code is using its IntelliSense, or code completion logic, to suggest what the valid values for the `target` property are. Let's modify this value to `ES6`, and see what the generated JavaScript looks like.

With a `tsconfig.json` file in place, we do not need to specify the name of the TypeScript file that we would like to compile; we can simply type `tsc` on the command line, as follows:

```
tsc
```

Here, the TypeScript compiler is reading the `tsconfig.json` file, and applying the compilation values that it finds there, to any `.ts` file that it finds in either the current directory, or any sub-directories. This means that in order to compile a TypeScript project that may contain many files in many sub-directories, we just need to type `tsc` in the root directory of the project.

Now that we have changed the target version of JavaScript that we wish to generate for, which is now `ES6`, let's take a look at the output of the compiler, in the file `hello_typescript.js`, as follows:

```
"use strict";
var version = `es6`;
console.log(`hello ${version} typescript`);
```

Ignoring the `"use strict"` line at the top of this file, we can see that the generated JavaScript has not changed from our original TypeScript file. This shows that the compiler is correctly generating ES6-compatible JavaScript, even though we have not modified our original TypeScript file.

Note that we will explore the various options in the `tsconfig.json` file, and how they affect our code in a later chapter.

Watching files for changes

TypeScript also has a handy option that will watch an entire directory tree, and if a file changes, it will automatically recompile the entire project. Let's run the TypeScript compiler in this mode as follows:

```
tsc -w
```

Here, we have added the `-w` command-line option to our `tsc` command. The output will now become:

```
[3:13:43 PM] Starting compilation in watch mode...
[3:13:43 PM] Found 0 errors. Watching for file changes.
```

Here, we can see that the TypeScript compiler is now in watch mode, and is watching files in our project directory for any changes.

Now that we have a simple, bare-bones development environment set up, let's start to explore some of the main features of TypeScript and start to see how it makes writing JavaScript better.

TypeScript basics

JavaScript is not strongly typed. It is a language that is very dynamic, as it allows for objects to change their types, properties, and behavior on the fly. TypeScript, however, is strongly typed, and as such will enforce rules that govern how we use variables, functions, and objects.

In this section of the chapter, we will work through the basics of the TypeScript language. We will start with the concepts of strong typing, or static typing as it is also known. We will then move on to exploring some of the basic types that the language uses, and how the language can detect what type a variable is, based on how and where it is used within our code. We will then tackle the basics of function signatures, show how to debug our code with VS Code, and finally see how to import and use third-party JavaScript libraries.

Strong typing

Strong typing, or static typing as it is also known, means that when we create a variable, or define a parameter in a function, we specify what type it is. Once this type has been specified, we cannot change it. Loose typing is the exact opposite of this concept, and JavaScript is a loosely typed language.

As an example of loose typing in JavaScript, let's create a JavaScript file named `test_javascript.js`, and add the following code:

```
var test = "a string";
console.log("test = " + test);

test = 1;
console.log("test = " + test);

test = function (a, b) {
    return a + b;
}
console.log("test = " + test);
```

On the first line of this code snippet, we have declared a local variable named `test`, and assigned a string value of `"a string"` to it. We then use our handy `console.log` function to print its value to the console. We then assign a numeric value of `1` to the `test` variable, and again print its value to the console. Finally, we assign a function that takes two parameters, named `a` and `b`, to the `test` variable, and print its value to the console. If we run this code, by typing `node test_javascript.js`, we will get the following results:

```
test = a string
test = 1
test = function (a, b) {
    return a + b;
}
```

Here, we can clearly see the changes that we are making to the `test` variable. It changes from a string value to a numeric value, and then finally becomes a function.

Unfortunately, changing the type of a variable at runtime can be a very dangerous thing to do. If your code is expecting a variable to be a number, and tries to perform calculations on it, the results of these calculations may be unexpected if the variable held a string. In a similar way, if a variable is a function that does something, and someone inadvertently changes that variable into a string, a whole block of functionality has suddenly become lost.

TypeScript is a strongly typed language. This means that if you declare a variable to be of type string, then you need to treat it as a string throughout your code base. So how do we declare that a variable should be of a particular type? Well, TypeScript introduces a simple notation using the colon (`:`) symbol to indicate what type a variable should be, as follows:

```
var myString: string = `this is a string`;
```

Here, we can see that we have declared a variable named `myString`, and then followed it by a colon and the `string` keyword, before we assign a value. This technique is called type annotation, and it sets the type of the `myString` variable to be of type string. If we were to now try and assign a number to it, as follows:

```
myString = 1;
```

TypeScript will generate a compilation error, as follows:

```
hello_typescript.ts:6:1 - error TS2322: Type '1' is not assignable to
type 'string'.
6 myString = 1;
  ~~~~~~~~
```

Here, we have a `TS2322` error, which indicates that the type `'1'` (which is of type number) is not assignable to a variable that has been specified as being of type string. Note that the output of the compiler will let us know which line is generating this error, as well as where in the line the error is occurring.

Basic types

TypeScript provides a number of basic types that can be used for type annotation. As an example of these types, let's focus on four common types, namely string, number, boolean, and array. We will cover the rest of these basic types, such as any, null, never, and others, in a later chapter, but for now, let's use these four basic types as follows:

```
var myBoolean : boolean = true;
var myNumber : number = 1234;
var myStringArray : string[] = [`first`, `second`, `third`];
```

Here, we have declared a variable named `myBoolean`, which is of type `boolean`, and is set to the value `true`. We have then declared a variable named `myNumber`, which is of type `number`, and is set to the value `1234`. Finally, we have declared a variable named `myStringArray`, which is of type `string` array, using the `[]` array syntax. All of the following statements are now invalid:

```
myBoolean = myNumber;
myStringArray = myNumber;
myNumber = myStringArray[0];
```

Here, we are attempting to assign the value of `myNumber` to the `myBoolean` variable, and then we are attempting to assign the value of `myNumber` to the `myStringArray` variable. Finally, we are attempting to assign the value of the first element of `myStringArray` to the `myNumber` variable. The following errors will be produced:

```
error TS2322: Type 'number' is not assignable to type 'boolean'.
error TS2322: Type 'number' is not assignable to type 'string[]'.
error TS2322: Type 'string' is not assignable to type 'number'.
```

This output is clearly telling us that we are not allowed to assign values from one type to another. Let's fix this code as follows:

```
myBoolean = myNumber === 456;
myStringArray = [myNumber.toString(), `5678`];
myNumber = myStringArray.length;

console.log(`myBoolean = ${myBoolean}`);
```

```
console.log(`myStringArray = ${myStringArray}`);
console.log(`myNumber = ${myNumber}`);
```

Here, we are assigning the value of the statement `myNumber === 456` to the `myBoolean` variable. The type of the equals expression is `boolean`, as it will result in either `true` or `false`. So in this case, the right-hand side of the assignment is of the same type as the left-hand side of the assignment, and is therefore valid TypeScript.

In the same manner, we are then converting the `myNumber` variable into a string using the `.toString()` function, and creating an array out of it, with the value `"5678"` by enclosing both in two square brackets `[` and `]`. Finally, we are assigning the `length` property of the `myStringArray` array to the `myNumber` variable. Again, this is allowed, since the `length` property of an array is of type number. The output of this code is as follows:

```
myBoolean = false
myStringArray = 1234,5678
myNumber = 2
```

Here, we can see that the `console.log` statements are producing the expected results. The `myNumber` variable is not equal to the value `456`, hence the `myBoolean` variable is set to `false`. The value of the `myStringArray` variable is now `1234` and `5678`, and the value of the `myNumber` variable is the length of the `myStringArray` array, which is `2`.

Inferred typing

TypeScript also uses a technique called inferred typing, or type inference, to determine the type of a variable. This means that even if we do not explicitly specify the type of a variable, the compiler will determine its type based on when it was first assigned. Again, once the variable has a type, normal type comparisons will be used. As an example of this, consider the following code:

```
var inferredString = "this is a string";
var inferredNumber = 1;

inferredNumber = inferredString;
```

Here, we have created two variables named `inferredString` and `inferredNumber`, and assigned a string value to the first variable, and a numeric value to the second. We are then attempting to assign the value of `inferredString` to the variable `inferredNumber`. This code will generate the following error:

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

What this error is telling us is that TypeScript has inferred that the type of the variable `inferredString` is of type string, even though we have not used the explicit `: string` type syntax. In the same manner, TypeScript has inferred that the type of the variable `inferredNumber` is of type number, hence the error.

VS Code will show the type of a variable if you hover your mouse over it when editing. This feature is pretty handy when dealing with inferred types.

Duck typing

TypeScript also uses a method called duck typing for more complex variable types. Duck typing suggests that "if it looks like a duck, and quacks like a duck, then it probably is a duck." In other words, two variables are considered to have the same type if they have the same properties and methods. Let's test this theory out with the following code:

```
var nameIdObject = { name: "myName", id: 1, print() { } };
nameIdObject = { id: 2, name: "anotherName", print() { } };
```

Here, we have defined a variable called `nameIdObject`, which is a standard JavaScript object, and has a `name` property, an `id` property, and a `print` function. We then reassign the value of this variable to another object that also has a `name` property, an `id` property, and a `print` function. The compiler will use duck typing in this instance to figure out if this assignment is valid. In other words, if an object has the same set of properties and functions, then they are considered to be of the same type.

To further illustrate this technique, let's see how the compiler reacts if we attempt to break this rule, as follows:

```
nameIdObject = { id: 3, name: "thirdName" };
```

Here, we are attempting to assign an object to `nameIdObject` that does not have all of the required properties. TypeScript will generate the following error:

```
error TS2741: Property 'print' is missing in type '{ id: number; name:
string; }' but required in type '{ name: string; id: number; print():
void; }'.
```

Here, we can see that the error message clearly states that the `print` property is missing, as TypeScript is using duck typing to ensure type safety.

Note that these errors will occur if the name of a property, the type of a property, or the number of properties are not exactly the same when assigning a value to an object.

Let's take a look at a slightly different example of duck typing, as follows:

```
var obj1 = { id: 1, print() { } };
var obj2 = { id: 2, print() { }, select() { } }

obj1 = obj2;
obj2 = obj1;
```

Here, we have two objects, named `obj1` and `obj2`, which are identical except that `obj2` has an extra `select` function. We then assign the value of `obj2` to `obj1`. This will not generate an error, as the type of `obj2` has all of the properties of the type of `obj1`. This means that the duck typing method is checking whether `obj2` has at least the properties of `obj1`, which it does.

The last line of this code snippet, where we assign the value of `obj1` to `obj2`, will generate an error as follows:

```
Property 'select' is missing in type '{ id: number; print(): void; }'
but required in type '{ id: number; print(): void; select(): void; }'
```

Here, we can see that the compiler has correctly identified that the type of `obj1` does not have at least all of the properties of `obj2`, as it is missing the `select` function.

Remember that the duck typing examples used here are also using inferred typing, so the type of an object is inferred from when it is first assigned.

Function signatures and void

Thus far, we have had a quick tour of the basics of the TypeScript language, how it introduces type annotations for variables, and how it uses inferred typing and duck typing to check for valid assignments. One of the best features of using type annotations in TypeScript is that they can also be used to strongly type function signatures.

To explore this feature a little further, let's write a JavaScript function to perform a calculation, as follows:

```
function calculate(a, b, c) {
    return (a * b) + c;
}

console.log("calculate() = " + calculate(2, 3, 1));
```

Here, we have defined a JavaScript function named `calculate` that has three parameters, `a`, `b`, and `c`. Within this function, we are multiplying `a` and `b`, and then adding the value of `c`. The output of this code is what we expect, as follows:

```
calculate() = 7
```

The result is correct, as 2 * 3 = 6, and 6 + 1 = 7. Now, let's see what happens if we incorrectly call the function with strings, instead of numbers, as follows:

```
console.log("calculate() = " + calculate("2", "3", "1"));
```

The output of this code sample is:

```
calculate() = 61
```

The result of `61` is very different from the expected result of `7`. So what is going on here?

If we take a closer look at the code in the `calculate` function, we can figure out what JavaScript does when it tries to mix variables of different types. The product of two numbers, that is, `( a * b )`, will generate a numeric value. So even though our values of `a` and `b` are strings, JavaScript will attempt to convert these to numbers before multiplying them together. So we end up with 3 * 2 = 6. Unfortunately, the `+` operator can be used on both numbers and strings, and if one of the arguments is a string, JavaScript will convert both to strings prior to addition. This results in the string `"6"` being appended to the string `"1"`, resulting in the string `"61"`.

This code snippet is an example of how JavaScript can modify the type of a variable depending on how it is actually used. This means that in order to work effectively with JavaScript, we need to be aware of this sort of type conversion, and understand when and where it could take place. Obviously, these sorts of automatic type conversions can cause unwanted behavior in our code.

Let's now write the equivalent function in TypeScript, but this time specify that all arguments must be of type number, as follows:

```
function calculate(a: number, b: number, c: number): number {
    return (a * b) + c;
}

console.log(`calculate() = ${calculate(3, 2, 1)}`);
```

Here, we have specified that the arguments `a`, `b`, and `c` in our TypeScript version of the `calculate` function must all be of type number. In addition to this, we can see that we have specified the return type of the entire function to be of type `number`, by placing a type annotation at the end of the function definition, that is, `: number { … }`.

This also ensures that the caller of this function knows that the function will return a value of type number.

With our function definition now specifying a type for each argument, attempting to call this function with strings will fail, as follows:

```
console.log(`calculate() = ${calculate("3", "2", "1")}`);
```

Here, TypeScript will generate the following error:

```
error TS2345: Argument of type '"3"' is not assignable to parameter of
type 'number'.
```

This error message clearly tells us that we cannot use a string as an argument where a numeric argument is expected. We also cannot use the value that is returned by the function incorrectly as follows:

```
var returnedValue: string = calculate(3, 2, 1);
```

This statement will generate the following error:

```
error TS2322: Type 'number' is not assignable to type 'string'.
```

So the TypeScript compiler is also telling us that the return value of the `calculate` function is of type number, and so we cannot assign it to a variable that is of type string.

So, what about functions that do not return a value? This is where the TypeScript keyword `void` comes in handy. Consider the following code:

```
function printString(a: string) : void {
    console.log(a);
}

var returnedValue : string = printString("this is a string");
```

Here, we have defined a function named `printString` that takes a single parameter named `a`, which is of type string. We have also specified that this function will not return a value, by using the return type of `void`. The last line of this code snippet will therefore generate the following error:

```
error TS2322: Type 'void' is not assignable to type 'string'.
```

This error message is telling us that the `printString` function does not return anything, and therefore returns a type of void. If we attempt to assign something of a void type to a string, we will generate this type of error.

VS Code IntelliSense

The ability to add type annotations to variables and functions means that the TypeScript compiler knows how each portion of our code should be used. Once the code has been parsed by the compiler, it makes these annotations available through the TypeScript Language Service. VS Code, and indeed other editors, can access this Language Service in real time, and build up code completion hints, or IntelliSense.

Let's take a quick look at how the VS Code editor gives us type information. If we want to call the `calculate` function that we created earlier, all we need to do is to start typing a part of the word `calculator`, and VS Code will show us what it knows about this function, as follows:

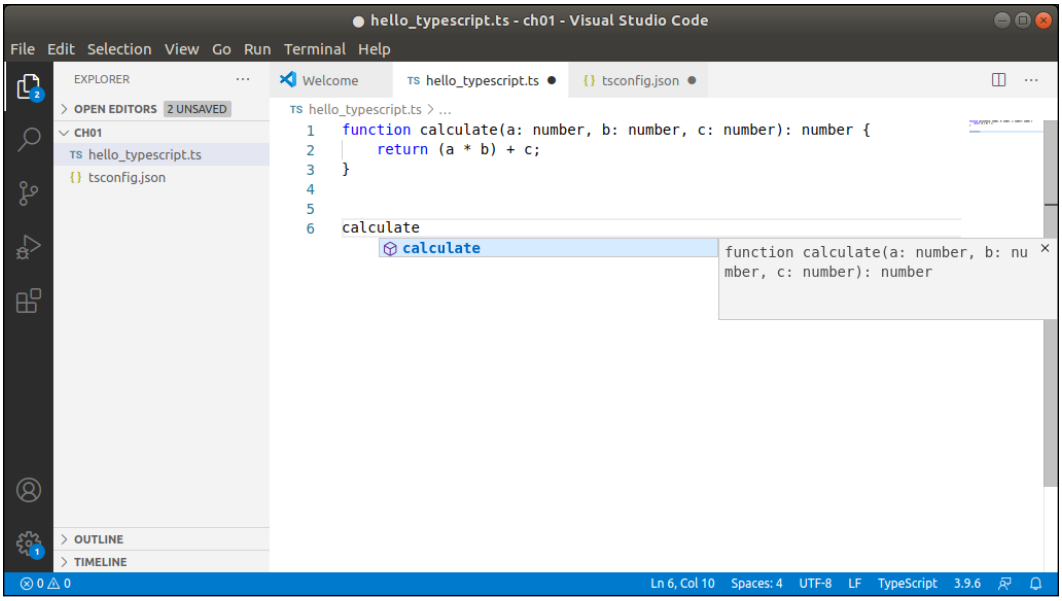


Figure 1.3: VS Code IntelliSense showing the definition of a function

Here, we can see that VS Code is telling us that the `calculate` function has three arguments, which are all of type number, and that it returns a number.

This sort of information is incredibly useful to a modern programmer, and makes writing and understanding code so much easier, and less prone to errors. We can also document the functions we are writing using JSDoc-style comments, and this JSDoc documentation will be included in the IntelliSense information, as shown in the following screenshot:

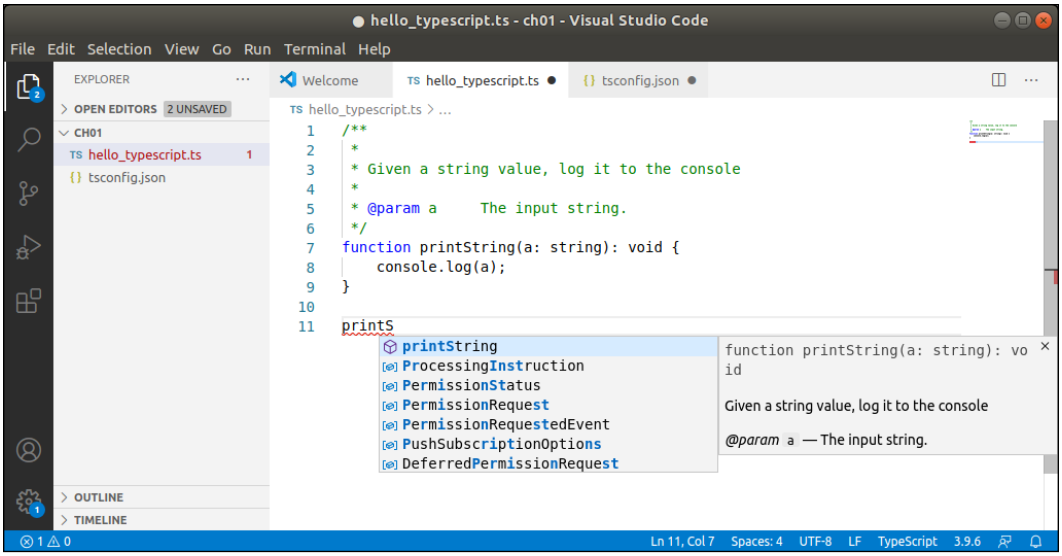


Figure 1.4: VS Code IntelliSense showing documentation from JSDoc-style comments

Here, we have updated the `printString` function, and included some JSDoc-style comments in the comment immediately preceding the function definition itself. In these comments, we are able to provide a description of the function, and we are also able to provide a description for each of the function parameters. VS Code will now include this documentation whenever we invoke the IntelliSense options in our IDE.

VS Code debugging

Using an IDE such as VS Code allows us to debug our code pretty easily. In order to do this, we will need to modify the TypeScript compilation options, found in `tsconfig.json`, and add the `sourceMap` property as follows:

```
{
    "compilerOptions": {
        "target": "ES3",
        "module": "commonjs",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true,
        "sourceMap": true
    }
}
```

Here, we have added the `sourceMap` property, and set it to `true`. This compiler option will now generate a `.map` file as well as a `.js` file for each of our TypeScript files. Remember that Node is a JavaScript runtime, and will only execute JavaScript code. The VS Code editor, therefore, needs to know how to map the executing JavaScript code back to our TypeScript source. This is what the `.map` file is used for.

With a `.map` file in place, we create a breakpoint in our source file by clicking on the margin to the left of the line number. This will put a small red dot in the margin, indicating that it is a breakpoint. If we simply hit *F5* now, VS Code will generate a small popup asking about the project we are attempting to debug. As this is a Node project, we need to select the **Node.js** option, and VS Code will start running our `hello_typescript` program and stop on the first breakpoint it encounters, as shown in the following screenshot:

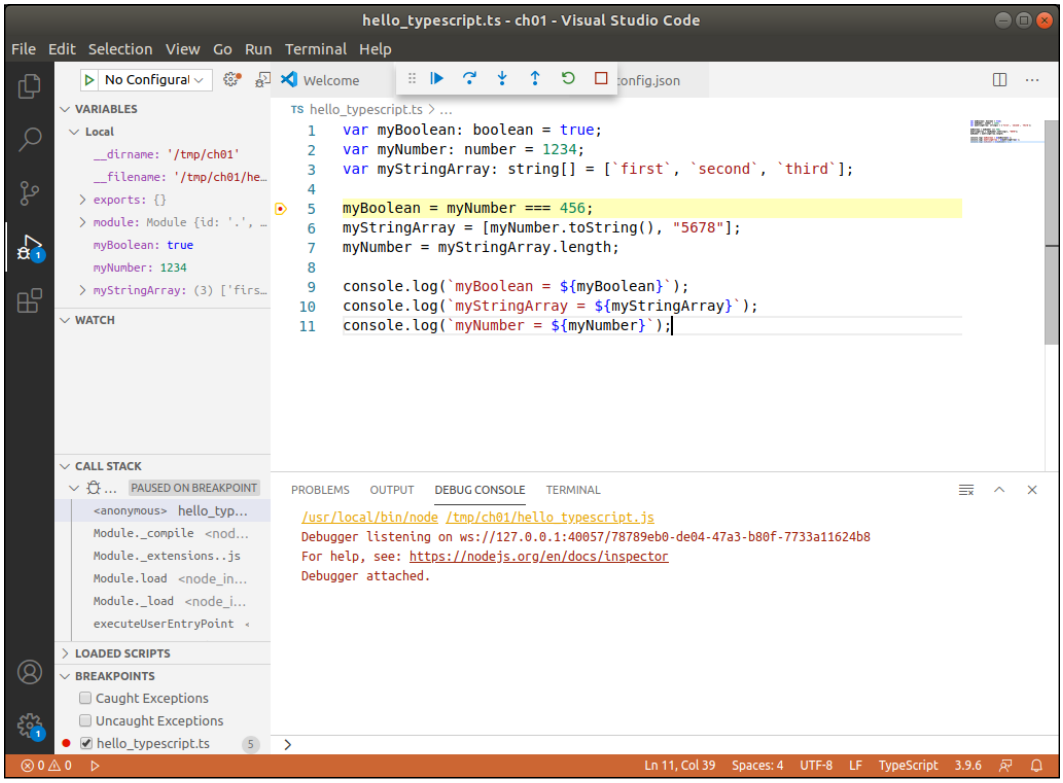


Figure 1.5: VS Code active debugging, showing breakpoints and the DEBUG CONSOLE view

Here, we can see that VS Code has started executing our program, and stopped at the first breakpoint. On the left of the screen, we can see that we have access to the **VARIABLES** panel, as well as a **WATCH** panel and a **CALL STACK** panel. On the bottom of the code editing window, we can see the **DEBUG CONSOLE** window, which will show some debugging information, as well as any `console.log` output.

Within this debugging environment, we can use the standard run (*F5*), step into (*F10*), and step over (*F11*) keys to navigate the code, or use the debug buttons provided. If we allow the code to run through to the end, we will see an option in the left-hand side **DEBUG** panel to create a `launch.json` file. If we click this, and then select the **Node.js** option, VS Code will automatically create a `launch.json` file in the `.vscode` subdirectory of our project. Let's take a look at this `launch.json` file, as follows:

```
{
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "skipFiles": [
                "<node_internals>/**"
            ],
            "program":
                 "${workspaceFolder}/ch01/hello_typescript.ts",
            "outFiles": [
                "${workspaceFolder}/**/*.js"
            ]
        }
    ]
}
```

Here, we have a simple JSON format file that has a `configurations` property, which holds an array of configuration options. The one and only configuration in this file specifies a few values for the `type`, `request`, `name`, `skipFiles`, `program`, and `outFiles` properties. Note that since we chose the **Node.js** option when generating this file, the `type` property is set to `node`. The `name` property can be modified to a human-readable text value, so something like `"Launch hello_typescript"`. Note the `program` property, which has automatically been set to point to the source of our file.

This `launch.json` file can be modified to allow for Chrome debugging, which is used to debug any code that runs within a browser. Note that IntelliSense has been provided for this file, so to find out more about what this file can do, edit it, and hit *Ctrl+Space* to view the available options. We will not discuss these launch options here, as the point of this chapter is to get up and running as quickly as possible, but at least we have an idea on how to get a debugger going, and where to look to configure it.

Introducing third-party libraries

One of the benefits of JavaScript development is the ability to download and install a whole host of third-party libraries that have been built to accomplish a huge variety of things in JavaScript. The easiest way to download and install them is to use the npm program that is packaged with Node. In order to do this, we start by initializing the npm environment within our project subdirectory with the following command:

```
npm init
```

The `init` parameter will cause npm to ask us a few simple questions, including the package name, version, and entry point to our program. We can simply accept all of the defaults at this stage by just hitting *Enter*, as we can modify them easily at a later stage. The `npm init` command, when complete, will generate a `package.json` file in the current directory. We will need this `package.json` file in order to install third-party JavaScript packages.

As an example of using a third-party JavaScript package, let's use a package named `inquirer`, which allows us to ask a user a series of questions from the command prompt, and then stores these answers for later use. This will be handy if we are building command-line interfaces in Node. We can install the `inquirer` package using npm as follows:

```
npm install inquirer
```

After a few seconds, npm should have downloaded and installed this library into the directory named `node_modules`. If the package itself has dependencies on other packages, then npm will download and install those as well. Let's now take a look at what happened to the `package.json` file that we created earlier:

```
{
    "name": "ch01",
    "version": "1.0.0",
    "description": "",
    "main": "hello_typescript.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "inquirer": "^7.1.0"
    }
}
```

Here, we can see that npm has included a property named `dependencies` in the `package.json` file, which lists a single entry named `inquirer`. The `inquirer` entry's value is `"^7.1.0"`, which is the version number of the package at the time of installation. This version number is using the major version, minor version and patch version numbering scheme (a little more on this later). This project, therefore, has a single runtime dependency, and will not run correctly without the `inquirer` package being installed.

The `package.json` file can also be used to recreate the `node_modules` directory, and must be committed to source control for your project. This means that any developer getting a copy of the source code can simply type `npm install` in the project directory, and npm will download copies of the dependent packages.

Once a `package.json` file has been created, or if a `package.json` file has been modified, npm will either create or update a file named `package-lock.json`. This file contains the exact versions of each npm package that was installed for the project. If a `package-lock.json` file exists in the project, npm will use the specific versions found in this file to recreate the `node_modules` directory. In this way, running `npm install` will download the same version of a package, or any dependent packages into the `node_modules` directory, thereby recreating the same environment for each developer. For this reason, the `package-lock.json` file must also be committed to source control.

Note that in large projects, the `node_modules` directory can very easily grow to several gigabytes of data, and therefore should not be committed to source control. Use the `package.json` and the `package-lock.json` files instead.

Note the caret (`^`) at the start of the package version number in the `package.json` file. This is used to indicate what can be done if new versions of the package are found during an `npm install` step. If we use the caret (`^`), this means that npm will upgrade the package if a new minor version or patch version is found. So `"^7.1.0"` will upgrade if a minor version number `"7.2.0"` is found, or if a new patch version `"7.1.1"` is found.

If we use the tilde (`~`) character at the start of the version number, as in `"~7.1.0"`, then only patch versions will be used if a new version is found. So `"~7.1.0"` will upgrade if a new patch version `"7.1.1"` is found, but will not upgrade if a new minor version is found, as in `"7.2.0"`.

If we do not use a preceding character, and simply leave the version at `"7.1.0"`, then npm will not attempt to upgrade packages, and will leave the version as stated.

Declaration files

Keep in mind that third-party libraries, such as the `inquirer` library that we just installed, are published as JavaScript libraries. This means that the TypeScript compiler does not have access to any type annotations that it would have if the library was written in TypeScript. So how does TypeScript enforce strict type checking on external JavaScript libraries?

The solution to this problem is, in fact, surprisingly simple. TypeScript uses files known as declaration files as a sort of header file, similar to languages such as C++, in order to superimpose strong typing on existing JavaScript libraries. These declaration files, which have a `.d.ts` extension, hold information that describes the available functions and variables that a library exposes, along with their associated type annotations.

As an example of this, let's install the declaration files for the `inquirer` npm package as follows:

```
npm install --save-dev @types/inquirer
```

There are two interesting things about these command-line options. Firstly, we have used the `--save-dev` option with `npm install`. This option specifies that the library we are installing is only used during development, and should not be packaged, or used, in a production version.

Secondly, we have prefixed the library named `inquirer` with an `@types/` prefix. This directs npm to download the TypeScript declaration files that are associated with the `inquirer` project. Once it is complete, we will notice that npm has downloaded a declaration file named `index.d.ts` into the `node_modules/@types/inquirer` directory. We will discuss declaration files at length in a later chapter, but for now just remember that a declaration file gives us type annotations for published JavaScript libraries.

Note that it is becoming increasingly popular for JavaScript libraries to include a declaration file within their standard distribution, so for these libraries, we do not need to install a separate `@types` npm package.

As an example of what a declaration file would contain, let's take a look at the second to last line of the `@types/inquirer/index.d.ts` file, as follows:

```
declare var inquirer: inquirer.Inquirer;
```

Here, we see that the `declare` keyword is being used to prefix a variable named `inquirer`. This `declare` keyword tells the TypeScript compiler that there is an object named `inquirer` that is provided by this library. We do not need to know the implementation of this library at this point; we just need to know that this library exists, what name it has, and what we can do with it. This is the purpose of declaration files.

With the declaration files installed for our `inquirer` library, we can go ahead and write some TypeScript code to use this library as follows:

```
import * as inquirer from 'inquirer';

inquirer.prompt(
    [
        {
            name: "first_name",
            message: "what is your name ?"
        }
    ]
).then(answers => {
    console.log(`you answered : ${answers.first_name}`);
});
```

Here, we start with an `import` statement. This tells the Node runtime that it must load a file named `'inquirer'` before running our code. The Node runtime will search the `node_modules` directory, and if found, will load the `'inquirer.js'` file as a dependency. The `import` statement uses the asterisk (`*`) symbol to specify that we want to import all publicly available functions and objects in the `inquirer` library, and the `as` keyword to attach these functions and objects to a namespace named `inquirer`. This means that in order to access any functions or objects in the `inquirer` library, we need to prefix them with the name `inquirer`, as seen in the second line of the code snippet.

We are calling the `prompt` function on the `inquirer` namespace, and passing in an array of objects that have both a `name` property, and a `message` property. The `name` property will be used to reference the answers that the user gives us, and the `message` property contains the text for the question.

The `prompt` function returns what is known as a **Promise**. A Promise is a special way of writing JavaScript code that is asynchronous in nature. We will cover asynchronous programming and Promises in depth in *Chapter 5*, *Asynchronous Language Features*, but for now, all we need to know is that when a Promise completes, it will execute the `.then` function. Our `then` function takes a single parameter named `answers`, and it is through this `answers` object that we can reference the answer provided to our `"what is your name?"` question. The `answers` object has a property that matches the `name` property from our array of questions. We are then echoing the answer to the question to the console. Running this code produces the following result:

```
? what is your name ? nathan
you answered : nathan
```

Here, we can see that the `inquirer` library has taken care of emitting our question to the console, and is awaiting user input. Once we type an answer to the question, the Promise completes, our `then` function is invoked, and we can see the results of the `console.log` output.

Summary

This has been a fast-paced introduction to what TypeScript is, and what it can do for JavaScript development. We started by setting up a simple development environment, got the TypeScript compiler installed, and started to use it to generate JavaScript. We then took a look at the basics of the type system, and explored how types can elevate the development experience. Finally, we used a third-party JavaScript library with the help of the declaration files.

In the next chapter, we will take a closer look at the type syntax in a lot more detail, get to understand where and how to use the available types, and how the type syntax can be used in different ways when writing functions.

2

Exploring the Type System

Thus far, we have covered the basic principles of using types in TypeScript. We already know how to define a type for a variable or a function argument, and some of the rules that the compiler uses in order to ensure strongly typed code. We have discussed some of the basic, or primitive, types (number, string, and boolean), as well as the array type. However, we have only just skimmed the surface of the TypeScript type system.

In this chapter, we will explore all of the primitive types that are available, as well as some language elements that we can use to better describe how to use these types. We will discuss when and where to use these types and language elements, and even when not to use some types.

This chapter is broken up into five main sections. In the first section, we will explore the `any` type, the `let` and `const` keywords, and union types. The concept of union types feeds directly into a discussion on type guards, and type aliases. We will round this first section off with a discussion on enums, including string enums and const enums.

The second section of this chapter will work through the remainder of the primitive types that are available for use, including undefined, null, object, and unknown. There are also language features that help us when we have to work with values that could be undefined or null, including optional chaining, nullish coalescing, and definite assignment. We will also discuss the `never` type, and how it can be used to identify logic errors.

In the third section of this chapter, we will discuss the object spread syntax, which is used to combine properties of one object with the properties of another. We will see how this spread syntax can also be used with arrays as a handy syntax for combining arrays and array values.

The fourth section of this chapter is all about tuples, what they are, and how they can be used.

The fifth and final section of this chapter deals with the use of types within functions and function signatures. We will see how we can use optional parameters in a function, default parameters, and rest syntax. We will also show how we can define a function signature as a function parameter, in order to ensure that any function provided as a callback function has the correct parameters and parameter types.

There is a lot of ground to cover in this chapter. It will, however, give us a great understanding of the TypeScript type system. We will explore what types are available for use, how and where these types can be used, and what language features are available to help with our use of types.

Let's explore the TypeScript type system.

any, let, unions, and enums

We have already seen how TypeScript introduced a simple type annotation syntax in order to promote strong typing, which helps us ensure that we are using variables as they are intended to be used within our code. We also know that TypeScript generates JavaScript, and as such must be able to mimic what JavaScript can do. Unfortunately, matching what JavaScript can do may also include relaxing the strict typing rules, and allowing a string value to be assigned to a numeric value, for example. In this section of the chapter, we will introduce the `any` type, which completely removes the strict type rules associated to a variable, and allows the fluid and unconstrained use of variables, like in JavaScript. We will also strongly recommend not using the `any` type, as much as possible.

We will also explore the `let` and `const` keywords, which are language elements introduced in later versions of the ECMAScript standard, and take a look at how they can be used in TypeScript. We will then take a look at union types, type guards, and type aliases, which allow us to clearly define how we would like our code to manage groups of types. Finally, we will discuss enums, which are a mechanism to replace magic strings, or magic numbers, with human-readable values.

The any type

We have already seen how TypeScript uses the type annotation syntax to define what type a particular variable or function parameter should be. Once we have set a type for a variable, the compiler will ensure that this type is maintained throughout our code base. Unfortunately, this means that we cannot re-create JavaScript where the JavaScript does not match these strict type rules. Consider the following JavaScript code:

```
var item1 = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Here, we create a variable named `item1` and assign to it an object value that has an `id` and `name` property. We then reassign this variable to an object that only has an `id` property. This is valid JavaScript code, and therefore, if we are using TypeScript to generate JavaScript, we will need to be able to mimic this functionality.

TypeScript introduces the `any` type for such occasions. Specifying that an object has a type of `any` will, in essence, remove the TypeScript strict type checking. The following TypeScript code shows how to use the `any` type to mimic our original JavaScript code, as follows:

```
var item1: any = { id: 1, name: "item1" }
item1 = { id: 2 };
```

Here, we have specified that the type of the `item1` variable is `any`. The `any` type then allows a variable to follow JavaScript's loosely defined typing rules so that anything can be assigned to anything. Without the type specifier of `any`, the second line of this code snippet would normally generate an error.

While the `any` type is a necessary feature of the TypeScript language, and is used for backward compatibility with JavaScript, its usage should be limited as much as possible. As we have seen with untyped JavaScript, excessive use of the `any` type will quickly lead to coding errors that will be difficult to find. Rather than using the `any` type, try to figure out the correct type of the object that you are using, and then use this type instead.

We will discuss the concept of interfaces in the next chapter, which are a way of defining custom types. Using interfaces allows us to cover almost every possible combination of types, meaning that using the `any` type, in most cases, is unnecessary.

We use an acronym within our programming teams, which is: **Simply Find an Interface for the Any Type** (**S.F.I.A.T**), pronounced sveat, or sweat. While this may sound rather odd, it simply brings home the point that the `any` type can and should be defined as an interface, so simply find it.

In short, avoid the `any` type at any cost.

Explicit casting

As with any strongly typed language, there comes a time when we need to explicitly specify the type of an object. This is generally used when working with object-oriented concepts, such as classes, abstract classes, and interfaces, but this technique can be used on primitive TypeScript types as well.

Let's rewrite our previous example using explicit casting, as follows:

```
var item1 = <any>{ id: 1, name: "item1" }
item1 = { id: 2 };
```

Here, we have replaced the `: any` type specifier on the left-hand side of the assignment operator with an explicit cast of `<any>` on the right-hand side of the assignment operator. This explicit casting technique uses the angled bracket syntax, that is, `<` and `>`, surrounding the name of the type. In essence, this syntax is equivalent to our earlier example, and specifies that the type of the `item1` variable is `any`.

Another variant of this casting technique is to use the `as` keyword, as follows:

```
var item1 = { id: 1, name: "item1" } as any;
item1 = { id: 2 };
```

Here, we have defined the variable named `item1`, similar to our earlier definitions, but have appended the keyword `as` and then named the type that this variable should be treated as, which in this case is `any`. This example, and the previous example, are equivalent in outcome, as the `item1` variable is of type `any` no matter which version of the explicit casting syntax we use.

Hopefully, this will be one of the last times that we use the type of any. Remember that using the type of any removes all strict type checking, and is therefore the antithesis of TypeScript. There may be very specific edge cases where we will still need to use the `any` type, but these should be few and far between.

The let keyword

The fluid nature of JavaScript variables can sometimes cause errors when we inadvertently define variables with the same name, but in a different scope within a code block. Consider the following TypeScript code:

```
var index: number = 0;

if (index == 0) {
    var index: number = 2;
    console.log(`index = ${index}`);
}

console.log(`index = ${index}`);
```

Here, we define a variable named `index` of type number using the `var` keyword, and assign it a value of `0`. We then test if this value is equal to `0`, and if it is, we enter a code block. The first statement in this code block defines a variable named `index`, of type number, and assigns the value `2` to it. We then print the value of the `index` variable both inside this code block and outside of it. The output of this code is as follows:

```
index = 2
index = 2
```

What this is showing us is that even though we thought we created a new variable within the `if` code block named `index`, this variable re-declaration actually points to the original `index` variable and does not create a new one. So, setting the value of the `index` variable within the code block will modify the value of the `index` variable outside of the code block as well. This is not what was intended.

The ES6 JavaScript specification introduces the `let` keyword to prevent this from happening. Let's refactor the preceding code using the `let` keyword, as follows:

```
let index: number = 0;

if (index == 0) {
    let index: number = 2;
    console.log(`index = ${index}`);
}

console.log(`index = ${index}`);
```

Here, we are defining the `index` variable by using the `let` keyword instead of the `var` keyword, both in the original declaration and within the `if` code block. No other change to the code is necessary. The output of this code is as follows:

```
index = 2
index = 0
```

Here, we can see that modifying the variable named `index` inside of our `if` code block does not affect the variable named `index` that is defined outside of the code block. They are seen as two separate variables.

It is best practice to use the `let` keyword to define variables, and not to use the `var` keyword at all. By using `let`, we are being more explicit about the intended use of these variables, which will help the compiler to pick up any mistakes in our code where these rules are broken.

Const values

When working with variables, it sometimes helps to indicate that the variable's value cannot be modified after is has been created with a specific value. TypeScript uses the `const` keyword, which was introduced in ES6, in order to accomplish this. Consider the following code:

```
const constValue = "this should not be changed";
constValue = "updated";
```

Here, we have defined a variable named `constValue` and indicated that its value cannot be changed using the `const` keyword. Attempting to compile this code will result in the following error:

```
error TS2588: Cannot assign to 'constValue' because it is a constant.
```

This error is being generated because of the second line in this code snippet. We are attempting to modify the value of the `constValue` variable, which is not allowed.

It is best practice to identify constant variables within our code and explicitly mark them as `const`. The use of `const` and `let` clearly indicates to the reader of the code the intent of the variable. A variable marked as `const` cannot be changed, and a variable declared with `let` is a block-scoped temporary variable.

Union types

TypeScript allows us to express a type as a combination of two or more other types. These types are known as union types, and they use the pipe symbol (|) to list all of the types that will make up this new type. Consider the following code:

```
function printObject(obj: string | number) {
    console.log(`obj = ${obj}`);
}

printObject(1);
printObject("string value");
```

Here, we have defined a function named `printObject` that has a single parameter named `obj`. Note how we have specified that the `obj` parameter can be either of type string or of type number by listing them as a union type with a pipe separator. The last two lines of the code call this function with a number, and then with a string. The output of this code is as follows:

```
obj = 1
obj = string value
```

Here, we can see that the `printObject` function will work with either a string or a number.

Type guards

When working with union types, the compiler will still apply its strong typing rules to ensure type safety. As an example of this, consider the following code:

```
function addWithUnion(
    arg1: string | number,
    arg2: string | number
) {
    return arg1 + arg2;
}
```

Here, we have defined a function named `addWithUnion` that accepts two parameters and returns their sum. The `arg1` and `arg2` parameters are union types, and can therefore hold either a string or a number. Unfortunately, this code will generate the following error:

```
error TS2365: Operator '+' cannot be applied to types 'string | number'
and 'string | number'
```

What the compiler is telling us here is that it cannot tell what type it should use when it attempts to add `arg1` to `arg2`. Is it supposed to add a string to a number, or a string to a string? As we discussed in *Chapter 1*, *Up and Running Quickly*, the effects of adding a string and a number in JavaScript can lead to unwanted results.

This is where type guards come in. A type guard is an expression that performs a check on our type, and then guarantees that type within its scope. Let's re-write our previous function with a type guard as follows:

```
function addWithTypeGuard(
    arg1: string | number,
    arg2: string | number
) {
    if (typeof arg1 === "string") {
        // arg 1 is treated as a string
        console.log(`arg1 is of type string`);
        return arg1 + arg2;
    }
```

```
    if (typeof arg1 === "number" && typeof arg2 === "number") {
        // both are numbers
        console.log(`arg1 and arg2 are numbers`);
        return arg1 + arg2;
    }
    console.log(`default return treat both as strings`)
    return arg1.toString() + arg2.toString();
}
```

Here, we have added two `if` statements within the body of our code. The first `if` statement uses the JavaScript `typeof` keyword to test what type the `arg1` argument is. The `typeof` operator will return a string depending on what the value of the argument is at runtime. This can be one of the following possible values: `"number"`, `"string"`, `"boolean"`, `"object"`, or `"undefined"`. If the type of the `arg1` argument is a string, then the first code block will execute. Within this code block, the compiler knows that `arg1` is of type string, and will therefore treat `arg1` to be of type string within the code block. Our type guard, therefore, is the code block after the check for the type of string.

Our second `if` statement has two `typeof` checks and is checking whether both the `arg1` and `arg2` arguments are of type number. If they are both numbers, then both `arg1` and `arg2` are treated as type number within the code block. This type guard, therefore, will treat both the `arg1` and `arg2` arguments as type number within this code block.

Let's test this function as follows:

```
console.log(` "1", "2" = ${addWithTypeGuard("1", "2")}`);
console.log(`  1 ,  2  = ${addWithTypeGuard(1, 2)}`);
console.log(`  1 , "2" = ${addWithTypeGuard(1, "2")}`);
```

Here, we call the `addWithTypeGuard` function three times: once with both arguments of type string, once with both arguments of type number, and the third time with a number and a string. The output of this code is as follows:

```
arg1 is of type string
 "1", "2" = 12
arg1 and arg2 are numbers
  1 ,  2  = 3
default return treat both as strings
  1 , "2" = 12
```

Here, we can see that our first call to the `addWithTypeGuard` function is using two arguments that are strings. The code identifies the first argument as being of type string and therefore enters the first `if` statement block.

The concatenation of the string `"1"` with the string `"2"` results in the string `"12"`. The second call to the `addWithTypeGuard` function uses two numbers as arguments, and our code therefore identifies both arguments as numbers, and as such adds the value `1` and the value `2`, resulting in `3`. The third call to the `addWithTypeGuard` function uses a number as the first argument and a string as the second. The code therefore falls through to our default code, and treats both arguments as strings.

Type aliases

TypeScript introduces the concept of a type alias, where we can create a named type that can be used as a substitute for a type union. Type aliases can be used wherever normal types are used and are denoted by using the `type` keyword, as follows:

```
type StringOrNumber = string | number;

function addWithTypeAlias(
    arg1: StringOrNumber,
    arg2: StringOrNumber
    ) {
    return arg1.toString() + arg2.toString();
}
```

Here, we have defined a type alias named `StringOrNumber` by using the `type` keyword and assigning a type union of string or number to it. We then use this `StringOrNumber` type in our function definition for the `addWithTypeAlias` function. Note that both the `arg1` and `arg2` arguments are of type `StringOrNumber`, which will allow us to call this function with either strings or numbers.

Type aliases are a handy way of specifying a type union and giving it a name, and are particularly useful when the type union is used over and over again.

Enums

Enums are a special type whose concept is similar to other languages such as C#, C++, or Java, and provides the solution to the problem of special numbers, or special strings. Enums are used to define a human-readable name for a specific number or string. Consider the following code:

```
enum DoorState {
    Open,
    Closed
}
```

```
function checkDoorState(state: DoorState) {
    console.log(`enum value is : ${state}`);
    switch (state) {
        case DoorState.Open:
            console.log(`Door is open`);
            break;
        case DoorState.Closed:
            console.log(`Door is closed`);
            break;
    }
}
```

Here, we start by using the `enum` keyword to define an enum named `DoorState`. This enum has two possible values, either `Open` or `Closed`. We then have a function named `checkDoorState` that has a single parameter named `state`, of type `DoorState`. This means that the correct way to call this function is with one of the values that the `DoorState` enum provides us. This function starts by logging the actual value of the `state` parameter to the console, and then executes a `switch` statement. This `switch` statement simply logs a message to the console depending on the value of the state parameter that was passed in.

We can now run this code as follows:

```
checkDoorState(DoorState.Open);
checkDoorState(DoorState.Closed);
```

Here, we are calling the `checkDoorState` function, once for each possible value within the `DoorState` enum. The output of this code is as follows:

```
enum value is : 0
Door is open
enum value is : 1
Door is closed
```

Here, we can clearly see that the compiler has generated a numerical value for each of our defined enum values. The numerical value for the enum value `DoorState.Open` is `0`, and likewise, the numerical value of `DoorState.Closed` has been set to `1`. This all occurs under the hood.

Using enums helps us to provide a clear set of values for a variable or function parameter. They also provide a tried and tested way of eliminating so called magic numbers by defining a limited number of possible values.

One last note on enums is that we can set the numerical value of an enum value to whatever we like, as shown in the following code:

```
enum DoorStateSpecificValues {
    Open = 3,
    Closed = 7,
    Unspecified = 256
}
```

Here, we have defined an enum named `DoorStateSpecificValues` that has three possible values, `Open`, `Closed`, and `Unspecified`. We have also overridden the default values for this enum such that the `Open` value will be `3`, the `Closed` value will be `7`, and the `Unspecified` value will be `256`.

String enums

A further variant of the enum type is what is known as a **string enum**, where the numerical values are replaced with strings, as follows:

```
enum DoorStateString {
    OPEN = "Open",
    CLOSED = "Closed"
}

console.log(`OPEN = ${DoorStateString.OPEN}`);
```

Here, we have an enum named `DoorStateString` and have replaced the numerical values with string values for each of the defined enum values. We then log a message to the console with the value of the `DoorStateString.OPEN` enum. The output of this code is as follows:

```
OPEN = Open
```

As expected, the compiler is resolving the enum value of `DoorStateString.OPEN` to the `"Open"` string.

Const enums

The final variant of the enum family is called the **const enum**, which adds the `const` keyword before the enum definition, as follows:

```
const enum DoorStateConst {
    Open = 10,
    Closed = 20
}

console.log(`const Closed = ${DoorStateConst.Open}`);
```

Here, we have defined a `const` enum named `DoorStateConst`, which has provided two possible values. We then log the value of the `DoorStateConst.Open` enum value to the console.

const enums have been introduced for performance reasons. To see what happens under the hood, we will need to view the JavaScript that this code produces. Firstly, let's take a look at the JavaScript implementation of the `DoorState` enum that we were discussing earlier. As the `DoorState` enum has not been marked as `const`, its JavaScript implementation is as follows:

```
var DoorState;
(function (DoorState) {
    DoorState[DoorState["Open"] = 0] = "Open";
    DoorState[DoorState["Closed"] = 1] = "Closed";
})(DoorState || (DoorState = {}));
```

Here, we have some pretty complex-looking JavaScript. We will not discuss this implementation here, but instead we'll take a look at what this structure becomes when we examine it in a debugger, such as the one in VSCode, as shown in the following screenshot:

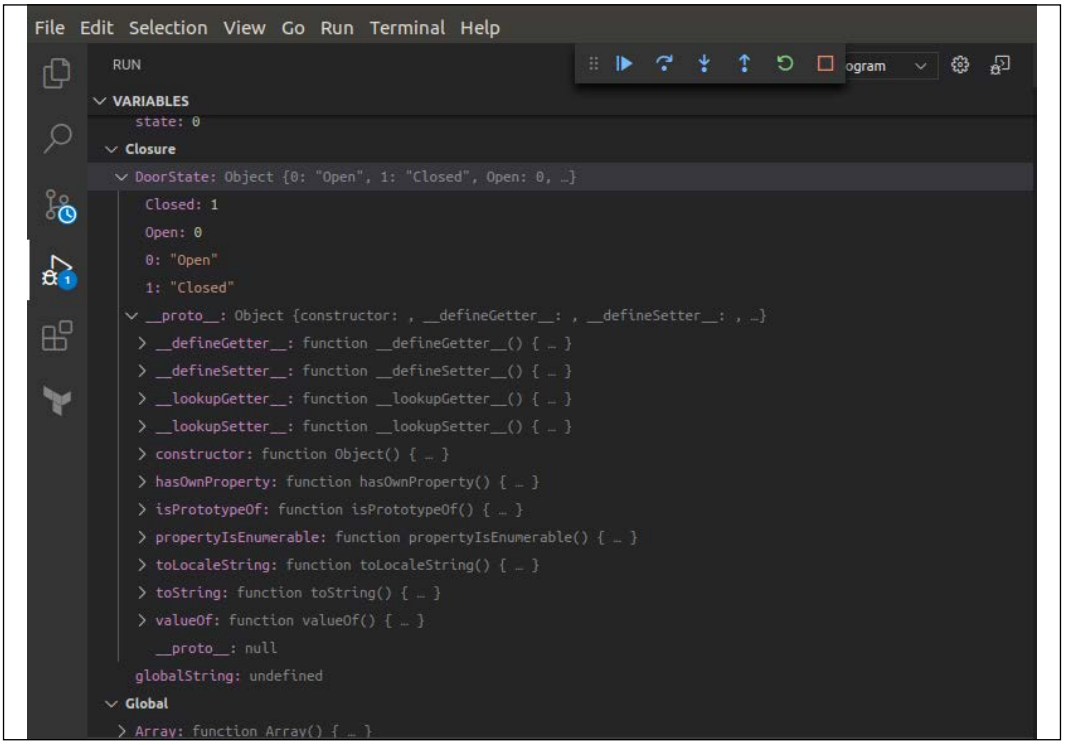


Figure 2.1: VSCode debugging window showing the internal structure of an enum

Here, we are viewing the object named `DoorState` within the VSCode debugger. We can see the `DoorState` object has four properties, named `Closed`, `Open`, `0`, and `1`. It also has a number of functions that have been attached to the object prototype, including a `constructor` and the `hasOwnProperty` and `toString` functions, to name a few. The purpose of this exercise was to show that when we create an enum, the compiler will generate a fully fledged JavaScript object, complete with properties and functions for the enum's implementation.

Let's now look at the generated JavaScript for a `const` enum:

```
console.log("const Closed = " + 10 /* Open */);
```

Here, we find that there is no actual implementation of the enum itself at all. The compiler has simply substituted the JavaScript code of `10 /* Open */` wherever we have used the `const` enum value of `DoorStateConst.Open`. This reduces the size of code that is generated, as the JavaScript runtime does not need to work with a full-blown JavaScript object in order to check a value.

More primitive types

In the last chapter, we discussed a few of the basic, or primitive, types that are available in TypeScript. We covered numbers, strings, and booleans, which are part of the group of primitive types, and we also covered arrays. While these represent some of the most basic and widely used types in the language, there are quite a few more of these primitive types, including undefined, null, unknown, and never. Related to these primitive types, we also have some language features, such as conditional expressions and optional chaining, that provide a convenient short-hand method of writing otherwise rather long-winded code. We will explore the remainder of the primitive types, as well as these convenient language features, in this part of the chapter.

Undefined

There are a range of circumstances where the value of something in JavaScript is undefined. Let's take a look at an example of this, as follows:

```
let array = ["123", "456", "789"];

delete array[0];

for (let i = 0; i < array.length; i++) {
    console.log(`array[${i}] = ${array[i]}`);
}
```

Here, we start by declaring a variable that holds an array of strings named `array`. We then delete the first element of this array. Finally, we use a simple `for` loop to loop through the elements of this array and print the value of the array element to the console. The output of this code is as follows:

```
array[0] = undefined
array[1] = 456
array[2] = 789
```

As we can see, the array still has three elements, but the first element has been set to `undefined`, which is the result of deleting this array element.

In TypeScript, we can use the `undefined` type to explicitly state that a variable could be undefined, as follows:

```
for (let i = 0; i < array.length; i++) {
    checkAndPrintElement(array[i]);
}

function checkAndPrintElement(arrElement: string | undefined) {
    if (arrElement === undefined)
        console.log(`invalid array element`);
    else
        console.log(`valid array element : ${arrElement}`);
}
```

Here, we are looping through our array and calling a function named `checkAndPrintElement`. This function has a single parameter named `arrayElement`, and is defined as allowing it to be of type string or undefined. Within the function itself, we are checking if the array element is, in fact, `undefined`, and are logging a warning message to the console. If the parameter is not `undefined`, we simply log its value to the console. The output of this code is as follows:

```
invalid array element
valid array element : 456
valid array element : 789
```

Here, we can see the two different messages being logged to the console.

The undefined type, therefore, allows us to explicitly state when we expect a variable to be undefined. We are essentially telling the compiler that we are aware that a variable may not yet have been defined a value, and we will write our code accordingly.

Null

Along with `undefined`, JavaScript also allows values to be set to `null`. Setting a value to `null` is intended to indicate that the variable is known, but has no value, as opposed to undefined, where the variable has not been defined in the current scope. Consider the following code:

```
function printValues(a: number | null) {
    console.log(`a = ${a}`);
}

printValues(1);
printValues(null);
```

Here, we have defined a function named `printValues`, which has a single parameter named `a`, which can be of type number or of type null. The function simply logs the value to the console. We then call this function with the values of `1` and `null`. The output of this code is as follows:

```
a = 1
a = null
```

Here, we can see that the console logs match the input values that we called the `printValues` function with. Again, `null` is used to indicate that a variable has no value, as opposed to the variable not being defined in the current scope.

The use of null and undefined has been debated for many years, with some arguing that null is not really necessary and that undefined could be used instead. There are others that argue the exact opposite, stating that null should be used in particular cases. Just remember that TypeScript will warn us if it detects that a value could be null, or possibly undefined, which can help to detect unwanted issues with our code.

Conditional expressions

One of the features of newer JavaScript versions that we are able to use in the TypeScript language is a simple, streamlined version of the if then else statement, which uses a question mark (`?`) symbol to define the `if` statement and a colon (`:`) to define the `then` and `else` path. These are called **conditional expressions**. The format of a conditional expression is as follows:

```
(conditional) ? ( true statement )  :  ( false statement );
```

As an example of this syntax, consider the following code:

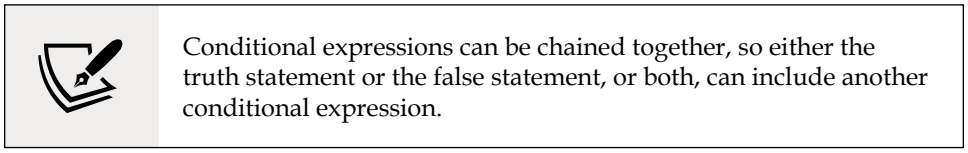
```
const value : number = 10;
const message : string = value > 10 ?
    "value is larger than 10" : "value is 10 or less";
console.log(message);
```

Here, we start by declaring a variable named `value`, of type number, that is set to the value of `10`. We then create a variable named `message`, which is of type string, and uses the conditional expression syntax to check whether the value of the `value` variable is greater than `10`. The output of this code is as follows:

```
value is 10 or less
```

Here, we can see that the `message` variable has been set to the string value of `"value is 10 or less"`, because the `value > 10` conditional check returned `false`.

Conditional expressions are a very handy syntax to use in place of the long-winded syntax we would normally have to use in order to code a simple if then else statement.



Conditional expressions can be chained together, so either the truth statement or the false statement, or both, can include another conditional expression.

Optional chaining

When using object properties in JavaScript, and in particular nested properties, it is important to ensure that a nested property exists before attempting to access it. Consider the following JavaScript code:

```
var objectA = {
    nestedProperty: {
        name: "nestedPropertyName"
    }
}

function printNestedObject(obj) {
    console.log("obj.nestedProperty.name = "
        + obj.nestedProperty.name);
}

printNestedObject(objectA);
```

Here, we have an object named `objectA` that has a nested structure. It has a single property named `nestedProperty`, which holds a child object with a single property called `name`. We then have a function called `printNestedObject` that has a single parameter named `obj`, which will log the value of the `obj.nestedProperty.name` property to the console. We then invoke the `printNestedObject` function and pass in `objectA` as the single argument. The output of this code is as follows:

```
obj.nestedProperty.name = nestedPropertyName
```

As expected, the function works correctly. Let's now see what happens if we pass in an object that does not have the nested structure that we were expecting, as follows:

```
console.log("calling printNestedObject");
printNestedObject({});
console.log("completed");
```

The output of this code is as follows:

```
calling printNestedObject
TypeError: Cannot read property 'name' of undefined
    at printNestedObject (javascript_samples.js:28:67)
    at Object.<anonymous> (javascript_samples.js:32:1)
```

Here, our code has logged the first message to the console, and has then caused a JavaScript runtime error. Note that the final call to log the `"completed"` message to the console has not even executed, as the entire program crashed while attempting to read the `'name'` property on an object that is `undefined`.

This is obviously a situation to avoid, and it can actually happen quite often. This sort of nested object structure is most often seen when working with JSON data. It is best practice to check that the properties that you are expecting to find are actually there, before attempting to access them. This results in code that's similar to the following:

```
function printNestedObject(obj: any) {
    if (obj != undefined
        && obj.nestedProperty != undefined
        && obj.nestedProperty.name) {
        console.log(`name = ${obj.nestedProperty.name}`)
    } else {
        console.log(`name not found or undefined`);
    }
}
```

Here, we have modified our `printNestedObject` function, which now starts with a long `if` statement. This `if` statement first checks whether the `obj` parameter is defined. If it is, it then checks if the `obj.nestedProperty` property is defined, and finally if the `obj.nestedProperty.name` property is defined. If none of these return `undefined`, the code prints the value to the console. Otherwise, it logs a message to state that it was unable to find the whole nested property.

This type of code is fairly common when working with nested structures, and must be put in place to protect our code from causing runtime errors.

The TypeScript team, however, have been hard at work in driving a proposal in order to include a feature named optional chaining into the ECMAScript standard, which has now been adopted in the ES2020 version of JavaScript. This feature is best described through looking at the following code:

```
function printNestedOptionalChain(obj: any) {
    if (obj?.nestedProperty?.name) {
        console.log(`name = ${obj.nestedProperty.name}`)
    } else {
        console.log(`name not found or undefined`);
    }
}
```

Here, we have a function named `printNestedOptionalChain` that has exactly the same functionality as our previous `printNestedObject` function. The only difference is that the previous `if` statement, which consisted of three lines, is now reduced to one line. Note how we are using the `?.` syntax in order to access each nested property. This has the effect that if any one of the nested properties returns `null` or `undefined`, the entire statement will return `undefined`.

Let's test this theory by calling this function as follows:

```
printNestedOptionalChain(undefined);
printNestedOptionalChain({
    aProperty: "another property"
});
printNestedOptionalChain({
    nestedProperty: {
        name: null
    }
});
printNestedOptionalChain({
```

```
    nestedProperty: {
        name: "nestedPropertyName"
    }
});
```

Here, we have called our `printNestedOptionalChain` function four times. The first call sets the entire `obj` argument to `undefined`. The second call has provided a valid `obj` argument, but it does not have the `nestedProperty` property that the code is looking for. The third call has the `nestedProperty.name` property, but it is set to `null`. Finally, we call the function with a valid object that has the nested structure that we are looking for. The output of this code is as follows:

```
name not found or undefined
name not found or undefined
name not found or undefined
name = nestedPropertyName
```

Here, we can see that the optional chaining syntax will return `undefined` if any of the properties within the property chain is either `null` or `undefined`.

Optional chaining has been a much-anticipated feature, and the syntax is a welcome sight for developers who are used to writing long-winded if statements to ensure that code is robust and will not fail unexpectedly.

Nullish coalescing

As we have just seen, it is a good idea to check that a particular variable is not either `null` or `undefined` before using it, as this can lead to errors. TypeScript allows us to use a feature of the 2020 JavaScript standard called nullish coalescing, which is a handy shorthand that will provide a default value if a variable is either `null` or `undefined`. Consider the following code:

```
function nullishCheck(a: number | undefined | null) {
    console.log(`a : ${a ?? `undefined or null`}`);
}

nullishCheck(1);
nullishCheck(null);
nullishCheck(undefined);
```

Here, we have a single function named `nullishCheck` that accepts a single parameter named `a` that can be either a number, undefined, or null. This function then logs the value of the `a` variable to the console, but uses a double question mark (`??`), which is the nullish coalescing operator. This syntax provides an alternative value, which is provided on the right hand side of the operator, to use if the variable on the left hand side is either `null` or `undefined`. We then call this function three times, with the values `1`, `null`, and `undefined`. The output of this code is as follows:

```
a : 1
a : undefined or null
a : undefined or null
```

Here, we can see that the first call to the `nullishCheck` function provides the value `1`, and this value is printed to the console untouched. The second call to the `nullishCheck` function provides `null` as the only argument, and therefore the function will substitute the string `undefined or null` in place of the value of `a`. The third call uses `undefined`, and as we can see, the nullish check will fail over to `undefined or null` in this case as well.

We can also use a function on the right-hand side of the nullish coalescing operator, or indeed a conditional statement as well, as long as the type of the value returned is correct.

Null or undefined operands

TypeScript will also apply its checks for `null` or `undefined` when we use basic operands, such as add (`+`), multiply (`*`), divide (`/`), or subtract (`-`). This can best be seen using a simple example, as follows:

```
function testNullOperands(a: number, b: number | null | undefined) {
    let addResult = a + b;
}
```

Here, we have a function named `testNullOperands` that accepts two parameters. The first, named `a`, is of type number. The second parameter, named `b`, can be of type number, null, or undefined. The function creates a variable named `addResult`, which should hold the result of adding `a` to `b`. This code will, however, generate the following error:

```
error TS2533: Object is possibly 'null' or 'undefined'
```

This error occurs because we are trying to add two values, and one of them may not be a numeric value. As we have defined the parameter `b` in this function to be of type number, null, or undefined, the compiler is picking up that we cannot add null to a number, nor can we add undefined to a number, hence the error.

A simple fix to this function may be to use the nullish coalescing operator as follows:

```
function testNullOperands(a: number, b: number | null | undefined) {
    let addResult = a + (b ?? 0);
}
```

Here, we are using the nullish coalescing operator to substitute the value of `0` for the value of `b` if `b` is either null or undefined.

Definite assignment

Variables in JavaScript are defined by using the `var` keyword. Unfortunately, the JavaScript runtime is very lenient on where these definitions occur, and will allow a variable to be used before it has been defined. Consider the following JavaScript code:

```
console.log("aValue = " + aValue);
var aValue = 1;
console.log("aValue = " + aValue);
```

Here, we start by logging the value of a variable named `aValue` to the console. Note, however, that we only declare the `aValue` variable on the second line of this code snippet. The output of this code will be as follows:

```
aValue = undefined
aValue = 1
```

As we can see from this output, the value of the `aValue` variable before it had been declared is `undefined`. This can obviously lead to unwanted behavior, and any good JavaScript programmer will check that a variable is not `undefined` before attempting to use it. If we attempt the same thing in TypeScript, as follows:

```
console.log(`lValue = ${lValue}`);
var lValue = 2;
```

The compiler will generate the following error:

```
error TS2454: Variable 'lValue' is used before being assigned
```

Here, the compiler is letting us know that we have possibly made a logic error by using the value of a variable before we have declared the variable itself.

Let's consider another, more tricky case of where this could happen, where even the compiler can get things wrong, as follows:

```
var globalString: string;

setGlobalString("this string is set");

console.log(`globalString = ${globalString}`);

function setGlobalString(value: string) {
    globalString = value;
}
```

Here, we start by declaring a variable named `globalString`, of type string. We then call a function named `setGlobalString` that will set the value of the `globalString` variable to the string provided. Then, we log the value of the `globalString` variable to the console. Finally, we have the definition of the `setGlobalString` function that just sets the value of the `globalString` variable to the parameter named `value`. This looks like fairly simple, understandable code, but it will generate the following error:

```
error TS2454: Variable 'globalString' is used before being assigned
```

According to the compiler, we are attempting to use the value of the `globalString` variable before it has been given a value. Unfortunately, the compiler does not quite understand that by invoking the `setGlobalString` function, the `globalString` variable will actually have been assigned a value before we attempt to log it to the console.

To cater for this scenario, as the code that we have written will work correctly, we can use the definite assignment assertion syntax, which is to append an exclamation mark (`!`) after the variable name that the compiler is complaining about. There are actually two places to do this.

Firstly, we can modify the code on the line where we use this variable for the first time, as follows:

```
console.log(`globalString = ${globalString!}`);
```

Here, we have placed an exclamation mark after the use of the `globalString` variable, which has now become `globalString!`. This will tell the compiler that we are overriding its type checking rules, and are willing to let it use the `globalString` variable, even though it thinks it has not been assigned.

The second place that we can use the definite assignment assertion syntax is in the definition of the variable itself, as follows:

```
var globalString!: string;
```

Here, we have used the definite assignment assertion operator on the definition of the variable itself. This will also remove the compilation error.

While we do have the ability to break standard TypeScript rules by using definite assignment operators, the most important question is why? Why do we need to structure our code in this way? Why are we using a global variable in the first place? Why are we using the value of a variable where if we change our logic, it could end up being undefined? It certainly would be better to refactor our code so that we avoid these scenarios.

The only place that the author has found where it makes sense to use definite assignment is when writing unit tests. In a unit test scenario, we may be testing the boundaries of a specific code path, and are purposefully bending the rules of TypeScript in order to write a particular test. All other cases of using definite assignment should really warrant a review of the code to see if it can be structured in a different way.

Object

TypeScript introduces the `object` type to cover types that are not primitive types. This includes any type that is not number, boolean, string, null, symbol, or undefined. Consider the following code:

```
let structuredObject: object = {
    name: "myObject",
    properties: {
        id: 1,
        type: "AnObject"
    }
}

function printObjectType(a: object) {
    console.log(`a: ${JSON.stringify(a)}`);
}
```

Here, we have a variable named `structuredObject` that is a standard JavaScript object, with a `name` property, and a nested property named `properties`. The `properties` property has an `id` property and a `type` property. This is a typical nested structure that we find used within JavaScript, or a structure returned from an API call that returns JSON. Note that we have explicitly typed this `structuredObject` variable to be of type `object`.

We then define a function named `printObjectType` that accepts a single parameter, named `a`, which is of type `object`. The function simply logs the value of the `a` parameter to the console. Note, however, that we are using the `JSON.stringify` function in order to format the `a` parameter into a human-readable string. We can then call this function as follows:

```
printObjectType(structuredObject);
printObjectType("this is a string");
```

Here, we call the `printObjectType` function with the `structuredObject` variable, and then attempt to call the `printObjectType` function with a simple string. This code will produce an error, as follows:

```
error TS2345: Argument of type '"this is a string"' is not assignable
to parameter of type 'object'.
```

Here, we can see that because we defined the `printObjectType` function to only accept a parameter of type `object`, we cannot use any other type to call this function. This is due to the fact that `object` is a primitive type, similar to string, number, boolean, null, or undefined, and as such we need to conform to standard TypeScript typing rules.

Unknown

TypeScript introduces a special type into its list of basic types, which is the type unknown. The unknown type can be seen as a type-safe alternative to the type any. A variable marked as `unknown` can hold any type of value, similar to a variable of type `any`. The difference between the two, however, is that a variable of type `unknown` cannot be assigned to a known type without explicit casting.

Let's explore these differences with some code as follows:

```
let a: any = "test";
let aNumber: number = 2;
aNumber = a;
```

Here, we have defined a variable named `a` that is of type `any`, and set its value to the string `"test"`. We then define a variable named `aNumber`, of type number, and set its value to `2`.

We then assign the value of `a`, which is the string `"test"`, to the variable `aNumber`. This is allowed, since we have defined the type of the variable `a` to be of type `any`. Even though we have assigned a string to the `a` variable, TypeScript assumes that we know what we are doing, and therefore will allow us to assign a string to a number.

Let's rewrite this code but use the `unknown` type instead of the `any` type, as follows:

```
let u: unknown = "an unknown";
u = 1;
let aNumber2: number;
aNumber2 = u;
```

Here, we have defined a variable named `u` of type `unknown`, and set its value to the string `"an unknown"`. We then assign the numeric value of `1` to the variable `u`. This shows that the `unknown` type mimics the behavior of the `any` type in that it has relaxed the normal strict type checking rules, and therefore this assignment is allowed.

We then define a variable named `aNumber2` of type number and attempt to assign the value of the `u` variable to it. This will cause the following error:

```
error TS2322: Type 'unknown' is not assignable to type 'number'
```

This is a very interesting error, and highlights the differences between the `any` type and the `unknown` type. While the `any` type in effect relaxes all type checking, the `unknown` type is a primitive type and follows the same rules that are applied to any of the primitive types, such as string, number, or boolean.

This means that we must cast an `unknown` type to another primitive type before assignment. We can fix the preceding error as follows:

```
aNumber2 = <number>u;
```

Here, we have used explicit casting to cast the value of `u` from type `unknown` to type `number`. Because we have explicitly specified that we are converting an `unknown` type to a `number` type, the compiler will allow this.

Using the `unknown` type forces us to make a conscious decision when using these values. In essence, we are letting the compiler know that we know what type this value should be when we actually want to use it. This is why it is seen as a type-safe version of `any`, as we need to use explicit casting to convert an unknown type into a known type before using it.

Never

The final primitive type in the TypeScript collection is a type of `never`. This type is used to indicate instances where something should never occur. Even though this may sound confusing, we can often write code where this occurs. Consider the following code:

```
function alwaysThrows() {
    throw new Error("this will always throw");
    return -1;
}
```

Here, we have a function named `alwaysThrows`, which will, according to its logic, always throw an error. Remember that once a function throws an error, it will immediately return, and no other code in the function will execute. This means that the second line of this function, which returns a value of `-1`, will never execute.

This is where the `never` type can be used to guard against possible logic errors in our code. Let's change the function definition to return a type of `never`, as follows:

```
function alwaysThrows(): never {
    throw new Error("this will always throw");
    return -1;
}
```

With the addition of the return type of `never` for this function, the compiler will now generate the following error:

```
error TS2322: Type '-1' is not assignable to type 'never'
```

This error message is clearly telling us that the function, which returns a type of `never`, is attempting to return the value of `-1`. The compiler, therefore, has identified a flaw in our logic.

Never and switch

A more advanced use of the `never` type can be used to trap logic errors within `switch` statements. Consider the following code:

```
enum AnEnum {
    FIRST,
    SECOND
}
```

```
function getEnumValue(enumValue: AnEnum): string {
    switch (enumValue) {
        case AnEnum.FIRST: return "First Case";
    }
    let returnValue: never = enumValue;
    return returnValue;
}
```

Here, we start with a definition of an enum named `AnEnum`, which has two values, `FIRST` and `SECOND`. We then define a function named `getEnumValue`, which has a single parameter named `enumValue` of type `AnEnum` and returns a string. The logic within this function is pretty simple and is designed to return a string based on the `enumValue` passed in.

Note, however, that the `switch` statement only has a `case` statement for the `FIRST` value of the enum, but does not have a `case` statement for the `SECOND` value of the enum. This code, therefore, will not work correctly if we call the function with `AnEnum.SECOND`.

This is where the last two lines of this function come in handy. The error message that is generated for this code is as follows:

```
error TS2322: Type 'AnEnum.SECOND' is not assignable to type 'never'
```

Let's take a closer look at this code. After our `switch` statement, we define a variable named `returnValue`, which is of type `never`. The trick in this code is that we assign the value of the incoming parameter, `enumValue`, which is of type `AnEnum`, to the `returnValue` variable, which is of type `never`. This statement is generating the error.

The TypeScript compiler, then, is examining our code, and determining that there is a `case` statement missing for the `AnEnum.SECOND` value. In this case, the logic falls through the `switch` statement, and then attempts to assign the `AnEnum.SECOND` value to a variable of type `never`, hence the error.

This code can be easily fixed, as follows:

```
function getEnumValue(enumValue: AnEnum): string {
    switch (enumValue) {
        case AnEnum.FIRST: return "First Case";
        case AnEnum.SECOND: return "Second Case";
    }
    let returnValue: never = enumValue;
    return returnValue;
}
```

Here, we have simply added the missing `case` statement to handle the `AnEnum.SECOND` value. With this in place, the error is resolved. While this may be fairly easy to spot in a simple example like this, this sort of error is commonplace when working with large code bases. Over time, developers often add values to an enum to get their unit tests to work, but can easily miss these missing `case` statements. Using the `never` type here safeguards our code so that we can pick up these errors earlier.

Object spread

When working with basic JavaScript objects, we often need to copy the properties of one object to another, or do some mixing and matching of properties from various objects. In TypeScript, we can use an ES7 technique known as object spread to accomplish this. Consider the following code:

```
let firstObj: object = { id: 1, name: "firstObj" };
let secondObj: object = { ...firstObj };

console.log(`secondObj : ${JSON.stringify(secondObj)}`);
```

Here, we have defined a variable named `firstObj` that is of type `object` and has an `id` property and a `name` property. We then define a variable named `secondObj` and use the object spread syntax of three dots (`...`) to assign a value to it. The value we are assigning is an object that is made up of the `firstObj` variable, that is `{ ...firstObj }`. The output of this code is as follows:

```
secondObj : {"id":1,"name":"firstObj"}
```

Here, we can see that the `id` and `name` properties and values have been copied into the new `secondObj` variable.

We can also use this technique to combine multiple objects together. Consider the following code:

```
let nameObj: object = { name: "nameObj name" };
let idObj: object = { id: 1 };

let obj3 = { ...nameObj, ...idObj };

console.log(`obj3 = ${JSON.stringify(obj3)}`);
```

Here, we have define a variable named `nameObj` that has a single property called `name`. We then have a variable named `idObj` that has a single property named `id`. Note how we are using the spread syntax to create a variable named `obj3` that is the result of combining the properties of `nameObj` and the properties of the `idObj` variables. The output of this code is as follows:

```
obj3 = {"name":"nameObj name","id":1}
```

This output shows us that the properties of both objects have been merged into the `obj3` variable, using the object spread syntax.

Spread precedence

When using object spread, properties will be copied incrementally. In other words, if two objects have a property with the same name, then the object that was specified last will take precedence. As an example of this, consider the following:

```
let objPrec1: object = { id: 1, name: "obj1 name" };
let objPrec2: object = { id: 1001, desc: "obj2 description" };

let objPrec3 = { ...objPrec1, ...objPrec2 };
console.log(`objPrec3 : ${JSON.stringify(objPrec3, null, 4)}`);
```

Here, we have defined two variables named `objPrec1` and `objPrec2`. Both of these objects have an `id` property; however, `objPrec1` has a `name` property, and `objPrec2` has a `desc` property. We then create a variable named `objPrec3` that is a combination of these two objects. Finally, we print the value of the `objPrec3` object to the console. The output of this code is as follows:

```
objPrec3 : {
    "id": 1001,
    "name": "obj1 name",
    "desc": "obj2 description"
}
```

Here, we can see that the spread operator has combined the properties of both original objects into the `objPrec3` variable. This new object has all three properties, `id`, `name`, and `desc`. Note that the `id` property was common between both original objects, and that the value of `1001` has taken precedence in this case, as it has been taken from the object that was specified last.

Spread with arrays

Interestingly, the spread syntax can also be used with arrays. Consider the following code:

```
let firstArray = [1, 2, 3];
let secondArray = [3, 4, 5];
let thirdArray = [...firstArray, ...secondArray];

console.log(`third array = ${thirdArray}`);
```

Here, we have defined two arrays, named `firstArray` and `secondArray`. We then use the spread syntax to combine these two arrays into another variable named `thirdArray`. We then print the value of the `thirdArray` variable to the console. The output of this code is as follows:

```
third array = 1,2,3,3,4,5
```

Here, we can see that the contents of the two arrays have been combined into the `thirdArray` variable. Interestingly, the new array contains the value `3` twice, as it was present in both arrays. Note that this syntax can be used on arrays of any type.

The spread syntax can also appear in any order. Consider the following code:

```
let objArray1 = [
    { id: 1, name: "first element" },
]

let objArray2 = [
    { id: 2, name: "second element" }
]

let objArray3 = [
    ...objArray1,
    { id: 3, name: "third element" },
    ...objArray2
]

console.log(`objArray3 = ${JSON.stringify(objArray3, null, 4)}`);
```

Here, we have defined two arrays named `objArray1` and `objArray2`, each with a single array element, that has both an `id` property and a `name` property. We then create a third variable named `objArray3`, which uses object spread to create a third array. Note that we are building the `objArray3` array out of the `objArray1` array, then adding an element, and then including the contents of the `ojbArray2` array. The output of this code is as follows:

```
objArray3 = [
    {
        "id": 1,
        "name": "first element"
    },
    {
        "id": 3,
        "name": "third element"
    },
    {
        "id": 2,
        "name": "second element"
    }
]
```

Here, we can see that the `objArray3` variable contains all of the elements of both the `objArray1` and `objArray2` arrays, as well as the element with `id : 3` , and `name : "third element"` that we injected into the middle of the array using spread syntax.

Tuples

Tuples are a method of defining a type that has a finite number of unnamed properties, with each property having an associated type. When using a tuple, all of the properties must be provided. This can best be explained in an example, as follows:

```
let tuple1: [string, boolean];
tuple1 = ["test", true];

tuple1 = ["test"];
```

Here, we have defined a variable named `tuple1`, whose type is defined as an array of types. The first type is a string, and the second type is a boolean. We then assign a value to the `tuple1` variable that contains an array with two values, the first of type string and the second of type boolean.

Note that the last line of this code attempts to assign a value to the `tuple1` variable that does not have all of the properties that are required. This last line will generate an error as follows:

```
error TS2741: Property '1' is missing in type '[string]' but required
in type '[string, boolean]'
```

What this error is telling us is that the number and types defined in a tuple must be provided when we assign anything to a tuple.

Tuple destructuring

As tuples use the array syntax, they can be destructured or disassembled in two ways. The first way of destructuring a tuple uses the simple array syntax, as follows:

```
console.log(`tuple1[0] : ${tuple1[0]}`);
console.log(`tuple1[1] : ${tuple1[1]}`);
```

Here, we are logging the values of the `tuple1` variable to the console by referencing its index within the array, that is, `tuple1[0]` and `tuple1[1]`. The output of this code is as follows:

```
tuple1[0] : test
tuple1[1] : true
```

Here, we can see that we can access each of the values in the tuple by using the array destructuring syntax. Note that the compiler knows that there are only two elements of this array, and if we attempt to access the third value within this tuple, that is, `tuple1[2]`, the compiler will generate an error.

Another way of destructuring a tuple is to use the array syntax to create an array of named elements and then assign the value of the tuple to this variable, as follows:

```
let [tupleString, tupleBoolean] = tuple1;
console.log(`tupleString = ${tupleString}`);
console.log(`tupleBoolean = ${tupleBoolean}`);
```

Here, we have used the array syntax to create a tuple out of two variable names, that is, `tupleString` and `tupleBoolean`. We then assign the value of our original tuple, that is, `tuple1`, to this array of named variables. We can then use these named variables instead of needing to access them using the standard array syntax, that is, `tuple1[0]`. The output of this code is as follows:

```
tupleString = test
tupleBoolean = true
```

Here, we can see that the tuple has been correctly destructured into our two named variables, `tupleString` and `tupleBoolean`.

Using the named variable syntax to destructure tuples is a better way of constructing your code, as you can name the variable according to how it will be used. We will see some practical examples of using tuples in *Chapter 9*, *Using Observables to Transform Data*, where we use the RxJS library to initiate multiple API calls to retrieve JSON data.

Optional tuple elements

Note that tuple elements can be marked optional by using the question mark (`?`) after the type, as follows:

```
let tupleOptional: [string, boolean?];
tupleOptional = ["test", true];
tupleOptional = ["test"];

console.log(`tupleOptional[0] : ${tupleOptional[0]}`);
console.log(`tupleOptional[1] : ${tupleOptional[1]}`);
```

Here, we have defined a tuple named `tupleOptional` that consists of two elements, a string, and an optional boolean value. We then assign the value of `["test", true]` to this tuple, and then we assign just the value `["test"]` to this tuple. As the second element has been marked as optional, we do not need to specify it. We then log the values of the tuple elements to the console, using array syntax. The output of this code is as follows:

```
tupleOptional[0] : test
tupleOptional[1] : undefined
```

Here, we can see that the tuple value at index `0` has been set to the value of `"test"`, but that the tuple value at index `1` is undefined as it was not specified in our last assignment statement.

Tuples and spread syntax

We are also able to use spread syntax to define a tuple that can have a variable number of elements. Consider the following code:

```
let tupleRest: [number, ...string[]];
tupleRest = [1];
tupleRest = [1, "string1"];
tupleRest = [1, "string1", "string2"];
```

Here, we are using spread syntax to indicate that the variable named `tupleRest` has a number element, followed by a variable number of string elements. We then assign values to this tuple, starting with a single numerical value, and then a numerical value and a variable number of string values. All of these assignments are valid.

Object destructuring

In a similar way to tuples, standard objects can be also be destructured. Consider the following example:

```
let complexObject = {
    aNum: 1,
    bStr: "name",
    cBool: true
}

let { aNum, bStr, cBool } = complexObject;
```

Here, we have defined an object named `complexObject` that has three properties, `aNum`, `bStr`, and `cBool`. Each of these properties has been assigned a value. We then destructure this object into three separate variables, named `aNum`, `bStr`, and `cBool`, in a similar manner to how we destructured tuples. We can now use these variables as follows:

```
console.log(`aNum : ${aNum}`);
console.log(`bStr : ${bStr}`);
console.log(`cBool : ${cBool}`);
```

Here, we are using the `aNum`, `bStr`, and `cBool` variables that we created when destructuring the `complexObject` object. The output of this code is as follows:

```
aNum : 1
objId : 1
objName : name
```

As we can see from this output, we are able to destructure simple objects into a series of variables, which allows us to access the value of these properties through our standard variable naming syntax.

Note that we are also able to rename the variable names during the destructuring step as follows:

```
let { aNum: objId, bStr: objName, cBool: isValid }
    = complexObject;
console.log(`objId : ${objId}`);
console.log(`objName : ${objName}`);
console.log(`isValid : ${isValid}`);
```

Here, we are destructuring the `complexObject` into a series of variables. Note the use of the colon (`:`) in this example. We are using the colon to rename the `aNum` property into the `objId` variable, using the syntax `aNum: objId`. Similarly, the `bStr` property is renamed to a variable named `objName`, and the `cBool` property is renamed to a variable named `isValid`. The colon (`:`) symbol as used here is not specifying a type as it normally would, but instead is used to rename the variable name used in destructuring.

Functions

In this section of the chapter, we will take a look at functions and their definitions, and how the TypeScript language can be used to introduce further type safety whenever functions are used. Functions can use many of the concepts that we have already discussed, including optional parameters and spread syntax. We will also discuss how we can define a function signature in such a manner that if a function defines another function as a parameter, we can make sure that the function we pass in has the correct parameters. Finally, we will take a look at how to define function overrides.

Optional parameters

Similar to how we have seen tuples using optional elements, we can specify that a function can have optional elements in the same way, using the question mark (`?`). Consider the following code:

```
function concatValues(a: string, b?: string) {
    console.log(`a + b = ${a + b}`);
}

concatValues("first", "second");
concatValues("third");
```

Here, we have defined a function named `concatValues` that has two parameters, `a` and `b`, both of type string. The second argument, `b`, however, has been marked as optional using the question mark after the argument name, that is, `b?: string`. We then call this function with two parameters, and then with only a single parameter. The output of this code is as follows:

```
a + b = firstsecond
a + b = thirdundefined
```

Here, we can see that the first call to the `concatValues` function concatenates the strings `"first"` and `"second"`, logging the value of `"firstsecond"` to the console. The second call to the `concatValues` function only provided a value for the first argument, as the second argument was marked as optional.

This second call to the `concatValues` function produces the output `"thirdundefined"`, as we have not specified a value for the second argument. This means that the argument `b` was not been specified and is thus `undefined`.

Note that any optional parameters must be listed last in the parameter list of the function definition. You can have as many optional parameters as you like, as long as non-optional parameters precede the optional parameters.

Default parameters

A variant of the optional parameter syntax allows us to specify a default value for a parameter, if it has not been supplied. Consider the following code:

```
function concatWithDefault(a: string, b: string = "default") {
    console.log(`a + b = ${a + b}`);
}

concatWithDefault("first", "second");
concatWithDefault("third");
```

Here, we have defined a function named `concatWithDefault` that has two parameters, `a` and `b`, both of type string. Note, however, the definition of the parameter named `b`. We are assigning the value of `"default"` to this parameter within the function definition. This assignment will automatically make this parameter optional, and we do not use the question mark syntax to define this parameter as optional. Note, too, that the use of the explicit type for the parameter `b`, as in `:string`, is also optional, as the compiler will infer the type from the default value, which in this case is type string.

We then call this function with two arguments, and then with just a single argument. The output of this code is as follows:

```
a + b = firstsecond
a + b = thirddefault
```

Here, we can see that when we supply two arguments to the `concatWithDefault` function, the function will concatenate the arguments as expected. When we only supply a single argument, the second argument will default to the value `"default"`.

Rest parameters

Interestingly, the parameters specified in a JavaScript function are all optional. Even if a JavaScript function specifies parameters in its function definition, we do not need to supply them when calling the function. In a quirky twist of the language, even if we do not specify any parameters in a function definition, we can still access the values that were provided when the function was invoked. Consider the following JavaScript code:

```
function testArguments() {
    for (var i = 0; i < arguments.length; i++) {
        console.log("argument[" + i + "] = " + arguments[i]);
    }
}

testArguments(1, 2);
testArguments("first", "second", "third");
```

Here, we have defined a JavaScript function named `testArguments` that does not specify any parameters. We then create a `for` loop to loop through the values of an array named `arguments`. If an array element is found, we log the value of the array element to the console. All JavaScript functions automatically have access to a special variable, named `arguments`, that can be used to retrieve all of the arguments that were used when the function is invoked.

We then invoke the `testArguments` function twice, once with the arguments `1` and `2`, and the second time with the arguments `"first"`, `"second"`, and `"third"`.

The output of this code is as follows:

```
argument[0] = 1
argument[1] = 2
argument[0] = first
argument[1] = second
argument[2] = third
```

Here, we can see a log of the arguments that were used to invoke the `testArguments` function. The first time we invoked the function, we used the arguments of `1` and `2`. The second time we invoked this function, we used the arguments of `"first"`, `"second"`, and `"third"`.

In order to express the equivalent function definition in TypeScript, we will need to use **rest syntax**, as follows:

```
function testArguments(...args: string[] | number[]) {
    for (let i in args) {
        console.log(`args[${i}] = ${args[i]}`);
    }
}

testArguments("1");
testArguments(10, 20);
```

Here, we have defined a function named `testArguments` using rest syntax, that is, the three dots (`...`), to specify that the function can be called with any number of parameters. We are also using a type union here to indicate that the variable parameters can be of type string or of type number.

We then invoke the `testArguments` function with one argument, which is the string `"1"`, and then invoke it with two numbers, namely `10` and `20`. The output of this code is as follows:

```
args[0] = 1
args[0] = 10
args[1] = 20
```

Here, we can see that the `testArguments` function can be called with multiple arguments, and because the function definition allows these parameters to be either of type string or of type number, we are able to mimic the functionality of the earlier JavaScript function.

Function callbacks

One of the most powerful features of JavaScript, and in fact the technology that NodeJS was built on, is the concept of callback functions. A callback function is a function that is passed in as an argument to another function, and is then generally invoked within the original function. In other words, we are calling a function and telling it to go and do what it needs to do, and when it is finished, to call the function that we have supplied.

Just as we can pass a value into a function, we can also pass a function into a function as one of its arguments.

This is best illustrated by taking a look at some JavaScript code, as follows:

```
var myCallback = function (text) {
    console.log("myCallback called with " + text);
}

function withCallbackArg(message, callbackFn) {
    console.log("withCallback called, message : " + message);
    callbackFn(message + " from withCallback");
}

withCallbackArg("initial text", myCallback);
```

Here, we start with a function named `myCallback` that accepts a single parameter named `text`. It simply logs the value of the `text` argument to the console. We then define a function named `withCallbackArg`, which has two parameters, named `message` and `callbackFn`. This function logs a message to the console using the `message` argument, and then invokes the function passed in as the `callbackFn` parameter. When invoking the function passed in, it invokes it with some text indicating that it was called within the `withCallback` function.

Finally, we invoke the `withCallbackArg` function with two arguments. The first argument is the text string of `"initial text"`, and the second argument is the `myCallback` function itself. The output of this code is as follows:

```
withCallback called, message : initial text
myCallback called with initial text from withCallback
```

As we can see from this output, the `withCallbackArg` function is being invoked and logging the `"withCallback called, message : initial text"` message to the console. It is then invoking the function that we passed into it as a callback function, which is the `myCallback` function.

Unfortunately, JavaScript cannot tell until it executes this code whether the second argument passed into the `withCallbackArg` function is actually a function. Let's test this theory by passing in a string for the `callbackFn` parameter, instead of an actual function, as follows:

```
withCallbackArg("text", "this is not a function");
```

Here, we are invoking the `withCallbackArg` function with two string values, instead of a string value and a function signature, as the function is expecting. The output of this code is as follows:

```
withCallback called, message : text

TypeError: callbackFn is not a function
    at withCallbackArg (javascript_samples.js:75:5)
    at Object.<anonymous> (javascript_samples.js:80:1)
    at Module._compile (internal/modules/cjs/loader.js:1133:30)
```

Here, we can see that we have caused a JavaScript runtime exception to occur, because the second argument that we passed into the `withCallbackArg` function was not a function, it was just a string.

JavaScript programmers, therefore, need to be careful when working with callbacks. The most useful technique for avoiding this sort of runtime error is to check if the argument passed in is actually a function using `typeof`, similarly to how we used `typeof` when creating type guards. This leads to a lot of defensive code being written to ensure that when a function is expecting a function to be passed in as a callback, it really is a function, before attempting to invoke it.

Function signatures as parameters

TypeScript uses its strong typing rules to ensure that if we define a function that needs a callback function, we can ensure that this function is provided correctly. In order to specify that a function parameter must be a function signature, TypeScript introduces the fat arrow syntax, or `() =>`, to indicate a function signature. Let's rewrite our previous JavaScript code using this syntax as follows:

```
function myCallback(text: string): void {
    console.log(`myCallback called with ${text}`);
}

function withCallbackArg(
    message: string,
    callbackFn: (text: string) => void
) {
    console.log(`withCallback called, message : ${message}`);
    callbackFn(`${message} from withCallback"`);
}
```

Here, we have defined a strongly typed function named `myCallback` that has a single parameter named `text`, which is of type string, and returns void. We have then defined a strongly typed function named `withCallbackArg` that also has two parameters. The first parameter is named `message` and is of type string, and the second parameter, named `callbackFn`, is using the fat arrow syntax, as follows:

```
callbackFn: (text: string) => void
```

This syntax defines the `callbackFn` parameter as being a function that accepts a single parameter of type string, and returns void.

We can then use this `withCallbackArg` function as follows:

```
withCallbackArg("initial text", myCallback);
withCallbackArg("text", "this is not a function");
```

Here, we have invoked the `withCallbackArg` function twice: once legitimately, by providing a string and a function as arguments, and once in error, by providing two strings as arguments. This code will produce the following error:

```
error TS2345: Argument of type '"this is not a function"' is not
assignable to parameter of type '(text: string) => void'
```

Here, we can clearly see that the compiler will not allow us to invoke the `withCallbackArg` function if we do not provide the second argument as a function with a signature that matches our function definition.

This is a very powerful feature of TypeScript. With its strong typing rules, it is preventing us from providing callback functions that do not conform to the correct function signature. Again, this helps to catch errors at the time of compilation, and not further down the line when the code needs to be actually run and tested.

Function overrides

TypeScript provides an alternative to union types when defining a function and allows a function signature to provide different parameter types. Consider the following code:

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any, b: any) {
    return a + b;
}

add("first", "second");
add(1, 2);
```

Here, we have defined a function definition named `add` that accepts two parameters, named `a` and `b`, which are both of type string, and returns a string. We have then defined another function with the same name, `add`, that accepts two parameters named `a` and `b` that are of type number, which returns a number. Note that neither of these function definitions has an actual function implementation.

Finally, we define a function, again with the name of `add`, that accepts two parameters named `a` and `b` but that are of type `any`. This function definition also provides a function implementation, which simply returns the addition of the `a` and `b` arguments.

This technique is used to provide what are known as function overrides. We can call this function with two arguments of type string, or two arguments of type number, as follows:

```
add("first", "second");
add(1, 2);
add(true, false);
```

Here, we have invoked the `add` function with three types of arguments. Firstly, we invoke it with two arguments of type string. We then invoke it with two arguments of type number. Finally, we invoke the `add` function with two arguments of type boolean. This last line of code will generate the following error:

```
 error TS2769: No overload matches this call.
  Overload 1 of 2, '(a: string, b: string): string', gave the following
error.
    Argument of type 'true' is not assignable to parameter of type
'string'.
  Overload 2 of 2, '(a: number, b: number): number', gave the following
error.
    Argument of type 'true' is not assignable to parameter of type
'number'.
```

Here, we can see that the only valid function signatures are where the arguments `a` and `b` are both of type string, or where the arguments `a` and `b` are both of type number. Even though our final function definition uses the type of any, this function definition is not made available and is simply used for the function implementation. We therefore cannot invoke this function with two boolean arguments, as the error shows.

Literals

TypeScript also allows us to use what are known as **literals**, which are almost a hybrid of enums and type aliases. A literal will limit the allowed values to a set of values specified. A literal can be made of string, number, or boolean values. Consider the following code:

```
type AllowedStringValues = "one" | "two" | "three";
type AllowedNumericValues = 1 | 20 | 65535;

function withLiteral(input:
    AllowedStringValues | AllowedNumericValues) {
    console.log(`called with : ${input}`);
}
```

Here, we have defined a literal named `AllowedStringValues`, as well as a literal named `AllowedNumericValues`. The syntax used for literals is very similar to the syntax of a type alias, where we use the `type` keyword followed by a set of allowed values. Unlike type aliases, however, we are not specifying a set of different types. We are specifying a set of allowed values, which is similar in concept to an enum.

We then have a function named `withLiteral` that accepts a single parameter of type `AllowedStringValues`, or of type `AllowedNumericValues`. This function simply logs the value of the `input` argument to the console. We can now use this function as follows:

```
withLiteral("one")
withLiteral("two");
withLiteral("three");
withLiteral(65535);

withLiteral("four");
withLiteral(2);
```

Here, we are invoking the `withLiteral` function with six values, namely `"one"`, `"two"`, `"three"`, `65535`, `"four"`, and `2`. Our literals, however, will only allow the values of `"one"`, `"two"`, `"three"`, `1`, `20`, and `65535`. As such, the last two lines of this code will generate the following errors:

```
error TS2345: Argument of type '"four"' is not assignable to parameter
of type '1 | 20 | "one" | "two" | "three" | 65535'.
 error TS2345: Argument of type '2' is not assignable to parameter of
type '1 | 20 | "one" | "two" | "three" | 65535'.
```

These error messages are generated because our literals do not allow the value `"four"` or the value `2` to be used.

Literals provide us with another tool that we can use when we need to define a function that accepts a standard string, number, or boolean, but where we need to limit the values provided to a defined set of values.

This concludes our exploration of the use of functions and function definitions with regard to the strong typing that TypeScript provides. We have discussed optional parameters, default parameters, rest syntax, function signatures, and function overrides. We also explored literals and how they can be used to limit the values allowed for function arguments.

Summary

In this chapter, we have taken a deep dive into the remainder of the primitive types that TypeScript makes available, such as any, null, undefined, object, and never. We covered a diverse range of language constructs, including the `let` keyword, optional chaining, nullish coalescing, object rest and spread, and tuples. We finished the chapter with a discussion on function definitions, and showed how we can use a variety of language constructs to accurately specify function parameters. In the next chapter, we will explore object-oriented programming techniques, including the use of classes, interfaces, inheritance, and modules.

3

Interfaces, Classes, Inheritance, and Modules

We have already seen a variety of language enhancements that TypeScript brings to modern JavaScript development. This includes the primitive types, like `string`, `number`, `boolean`, `undefined`, and `never`, as well as features brought in from multiple ECMAScript standards, like `let`, `const`, and optional chaining. TypeScript, therefore, allows us to use these enhanced language features from future JavaScript standards in our code right now, and it takes care of generating the correct JavaScript based on our runtime target.

The ECMAScript standard published in 2015, known as ES6, introduced the concept of classes and inheritance. JavaScript programmers, however, have been able to create classes and use object-oriented programming techniques for many years, by using what is known as the **closure design pattern** and the **prototype design pattern**. The creation of JavaScript objects through closures, however, has been more by convention than being baked directly into the language.

TypeScript has supported the object-oriented concepts of interfaces, classes, and inheritance since its earliest versions. Remember that it generates JavaScript, and as such, is capable of converting the classes that you construct today into older style JavaScript closures, if we are targeting older versions of the JavaScript runtime. This means that no matter what JavaScript runtime you are targeting, the TypeScript compiler will take care of generating the correct JavaScript.

In this chapter, we will discuss all things object-oriented, as follows:

- Interfaces
- Classes
- Inheritance
- Modules

Interfaces

In *Chapter 1*, *Up and Running Quickly*, we discussed how TypeScript uses duck typing to assess if two objects are compatible. These typing rules govern whether an object can be assigned to another and compares the properties of one object to the other.

Interfaces provide us with a mechanism to define what properties an object must implement and is, therefore, a way for us to define a custom type. By defining an interface, we are describing the properties and functions that an object is expected to have in order to be used by our code.

To illustrate these concepts, consider the following code:

```
interface IIdName {
    id: number;
    name: string;
}
```

Here, we have used the `interface` keyword to define a TypeScript interface named `IIdName`. This interface describes an object that has an `id` property of type number, and a `name` property of type string. Once we have defined an interface, we can use it in the same way as a primitive type, as follows:

```
let idObject: IIdNameObject = {
    id: 2
}
```

Here, we have created a variable named `idObject`, and specified that its type is the interface `IIdName`. We have, however, only specified a single property named `id`, and have not specified the `name` property, which therefore means that our object does not match the properties we are expecting according to the interface definition. This code will generate the following error:

```
error TS2741: Property 'name' is missing in type '{ id: number; }' but
required in type 'IIdName'
```

Here, we can see that the compiler is treating our interface definition just like it would treat a standard type. We are stating that the object `idObject` implements the `IIdNameObject` interface, but have not provided the `name` property that is required, hence the error.

To remove this error, we need to define both properties on our object, as required by the interface, as follows:

```
let idObject: IIdName = {
    id: 2,
    name : "this is a name"
}
```

Here, our object `idObject` provides both the `id` and the `name` properties correctly.

TypeScript will treat interfaces in the same way as it treats primitive types, and it will use duck typing to ensure that when we say that an object matches an interface, then it really does match the interface, with no property left behind.

Optional properties

Interface definitions may also include optional properties, in a similar way that functions may specify optional parameters, using the question mark (`?`) syntax. Consider the following interface:

```
interface IOptional {
    id: number;
    name?: string;
}
```

Here, we have an interface named `IOptional`, which has defined an `id` property of type `number`, and an optional property named `name` of type string. We can now define objects as follows:

```
let optionalId: IOptional = {
    id: 1
}

let optionalIdName: IOptional = {
    id: 2,
    name: "optional name"
}
```

Here, we have two objects named `optionalId` and `optionalIdName` that both implement the `IOptional` interface. As the `name` property has been marked optional, both of these objects correctly implement the interface.

Interfaces are compiled away

Interfaces do not generate any JavaScript code. This means that interfaces are a construct only used in the TypeScript compilation step and language services, and are there to ensure type safety. The JavaScript that is generated for our code thus far is as follows:

```
var idObject = {
    id: 2,
    name: "this is a name"
};
var optionalId = {
    id: 1
};
var optionalIdName = {
    id: 2,
    name: "optional name"
};
```

Here, we can see that the interface definitions that we have been using have all been removed during the compilation step, and all we are left with in this generated JavaScript are the objects themselves.

Interface naming

The TypeScript team have a coding standard for contributions to the TypeScript code base that prohibits the use of the letter `I` for prefixing interfaces. In the code samples that we have seen thus far, we have used the interface names `IIdName` and `IOptional`, which, according to the TypeScript teams standard, should just have been named `IdName` and `Optional`.

The reason for this standard boils down to two things. Firstly, an interface defines the shape of an object, and is therefore really just a type, in the same manner as a string is a type or a number is a type. When comparing types, the TypeScript compiler will use duck typing to check whether the shape of one object is compatible with another, so therefore there is no distinction between an interface and a class.

Secondly, the TypeScript language does not generate any code related to interfaces, so we cannot interrogate an interface at runtime, and ask questions such as "what properties does this interface define?". Other languages, such as C#, use a technique called **reflection**, which allows code to query an interface, and gather information on the available properties, and their types at runtime.

So an interface is just a name for a type, in the same way that a type alias is a name for a type. We do not, however, prefix type aliases with the letter `A`, or classes with the letter `C`, so why should we prefix interface names with the letter `I` ?

There are, however, some reasons why using a prefix with the letter `I` does make sense. If you or your team are migrating from languages like C# or Java, then this naming convention will seem familiar, and natural. It also helps when reading code outside of an IDE such as VSCode, which has access to the TypeScript language service, and can give you hints as to when an interface is used. Reading code in a code review tool, for example, does not have this contextual help, and can therefore lead to confusion around whether a type is an interface or something else.

For the purpose of this chapter, we will continue to distinguish between interfaces and other types by using an `I` prefix for interfaces, where it helps within the code samples. This serves to make it explicit when we are using an interface as opposed to a class or any other type. The rule that the TypeScript team has adopted is a rule for their particular development team, and you should adopt it if it makes sense within your code base, or within your team itself. Once you have decided on a particular naming convention, then stick with it, as there are pros and cons to either naming standard.

Weak types

When we define an interface where all of its properties are optional, this is considered to be a weak type. In other words, we have defined an interface, but none of the properties of the interface are mandatory. Let's see what happens if we create a weak type, and then try to bend the standard type rules, as follows:

```
interface IWeakType {
    id?: number,
    name?: string
}

let weakTypeNoOverlap: IWeakType = {
    description: "a description"
}
```

Here, we have defined an interface name `IWeakType`, which has an `id` property of type number, and a `name` property of type string. Both of these properties have been marked as optional. We then create a variable named `weakTypeNoOverlap`, which implements the `IWeakType` interface, and has a single property named `description`. This code generates the following error:

```
error TS2322: Type '{ description: string; }' is not assignable to type
'IWeakType'.
  Object literal may only specify known properties, and 'description'
does not exist in type 'IWeakType'
```

Here, we can see that the compiler is detecting that the `weakTypeNoOverlap` object has not implemented the `IWeakType` interface, as it does not have either an `id` or a `name` property. We can see, therefore, that TypeScript will strongly type even weak types.

Note that while we are able to create weak types in TypeScript, they are really just an edge case in terms of usage. We should really be thinking differently about our interface definitions, if all properties become optional, and we are introducing a weak type.

The in operator

JavaScript allows us to interrogate an object and see if it has a property using the `in` operator. Let's explore this operator with the following interfaces:

```
interface IIdName {
    id: number;
    name: string;
}

interface IDescrValue {
    descr: string;
    value: number;
}
```

Here, we have two interfaces. The first is named `IIdName`, and contains an `id` property of type number, and a `name` property of type string. The second interface is named `IDescrValue`, and contains a `descr` property of type string, and a `value` property of type number. Note that these interfaces describe completely different objects, and have no overlapping properties whatsoever. We can now write a function that will distinguish between these two interfaces using the `in` operator as follows:

```
function printNameOrValue(
    obj: IIdName | IDescrValue): void {
    if ('id' in obj) {
        console.log(`obj.name : ${obj.name}`);
    }
    if ('descr' in obj) {
        console.log(`obj.value : ${obj.value}`);
    }
}
```

Here, we have a function named `printNameOrValue` that has a single parameter named `obj`, which can either be of the type `IIdName`, or of the type `IDescrValue`. Note how we are then using an `if` statement to create a type guard that checks for the property `id` using the `in` operator. In other words, if the object has a property named `id`, then enter the `if` block. Within this `if` block, the object will be treated as implementing the `IIdName` interface, and this code block is acting as a type guard. We then log the value of the `obj.name` property, which is also part of the `IIdName` interface, to the console.

The second `if` statement in this function is again using the `in` operator, but this time using the `descr` property to detect if the object implements the `IDescrValue` interface. If it does implement this interface, we then log the value of the `obj.value` property, which is part of the `IDescrValue` interface, to the console.

We can now use this function as follows:

```
printNameOrValue({
    id: 1,
    name: "nameValue"
});

printNameOrValue({
    descr: "description",
    value: 2
});
```

Here, we are calling the `printNameOrValue` function with two different objects. The structure of the first object conforms to the `IIdName` interface, and the structure of the second objects conforms to the `IDescrValue` interface. The compiler will let us know if we call this function with anything that does not match either of these interfaces. The output of this code is as follows:

```
obj.name : nameValue
obj.value : 2
```

Here, we can see the function `printNameOrValue` is detecting the properties of each of these objects using the `in` operator, and entering the type guard code block depending on what properties it finds. Note that if an object implements both interfaces, or has properties that match both of these interfaces, then both type guards will be executed, as we have two `if` statements, and not an `if else` statement.

keyof

TypeScript allows us to iterate through the properties of a type and extract the names of its properties through the `keyof` keyword, which we can use as a string literal type. Let's explore this concept and how it applies to interfaces, as follows:

```
interface IPerson {
    id: number;
    name: string;
}

type PersonPropertyName = keyof IPerson;
```

Here, we have defined an interface named `IPerson` that defines two properties named `id` and `name`. We are then creating a string literal type for the valid properties of this interface named `PersonPropertyName`. Note that here, we are simply using the `keyof` keyword to generate a string literal type for the properties found in the `IPerson` interface. This is equivalent to the following string literal:

```
type PersonPropertyLiteral = "id" | "name";
```

We can now use this type as follows:

```
function getProperty(key: PersonPropertyName, value: IPerson) {
    console.log(`${key} = ${value[key]}`);
}

getProperty("id",
    { id: 1, name: "firstName" }
);
getProperty("name",
    { id: 2, name: "secondName" }
);
getProperty("telephone",
    { id: 3, name: "thirdName" }
);
```

Here, we have defined a function named `getProperty` that has two parameters, named `key` of type `PersonPropertyName`, and `value` of type `IPerson`. This function is using the string literal `PersonPropertyName` that was constructed using the `keyof` keyword. The function itself simply logs the name of the property key provided, and the value of the corresponding property, to the console. We are then invoking the function three times.

The first call to the `getProperty` function will output the value of the `"id"` property, and the second call to the `getProperty` function will output the value of the `"name"` property. The third call, however, will generate the following error:

```
error TS2345: Argument of type '"telephone"' is not assignable to
parameter of type '"id" | "name"'.'
```

Here, we can see that the compiler is correctly identifying that we are not allowed to call this function with the first argument of `"telephone"`, as this argument is not part of our string literal type.

Using the `keyof` keyword will generate a string literal that automatically includes all of the properties of an interface. This technique is obviously preferable to having to maintain string literals manually.

In this first section of the chapter, we have explored interfaces, and seen that they behave just like other types in TypeScript. We can use them to define custom types, and bring structure to nested objects. We know that they are a language construct only, as the TypeScript compiler will "compile away" any of our interfaces. The use of interfaces, however, becomes very powerful when used with other object-oriented programming concepts, such as classes, which we will discuss next.

Classes

A class is the definition of an object, what data it holds, and what operations it can perform. Classes and interfaces form the cornerstone of object-oriented programming. Let's take a look at a simple class definition, as follows:

```
class SimpleClass {
    id: number;
    print(): void {
        console.log(`SimpleClass.print() called.`);
    }
}
```

Here, we have defined a class, using the `class` keyword, which is named `SimpleClass`, and has an `id` property of type number, and a `print` function, which just logs a message to the console. Notice anything wrong with this code? Well, the compiler will generate an error message as follows:

```
error TS2564: Property 'id' has no initializer and is not definitely
assigned in the constructor
```

What this error is indicating is that if we create an instance of this class, then the newly created class will not have the `id` property initialized, and it will therefore be `undefined`. If our code is expecting the `id` property to have a value, then we might be surprised when it returns `undefined`, and the compiler is warning us of this potential error. There are two ways to fix this compiler error. We could set it to a default value such as `0`, or we could simply make the `id` property a type union, as follows:

```
id: number | undefined;
```

Here, we specified that the `id` property could be a `number`, or it could be `undefined`. We are therefore making a conscious decision within our code that we are prepared for this value to be `undefined`.

In order to use our `SimpleClass` class definition, we will need to create an instance of this class as follows:

```
let mySimpleClass = new SimpleClass();
mySimpleClass.print();
```

Here, we have defined a variable named `mySimpleClass`, and set it to a new instance of the `SimpleClass` class by using the `new` keyword. Once the instance of this class has been created, we call the `print` function. The output of this code is as follows:

```
SimpleClass.print() called.
```

The this keyword

As we have seen, a class definition specifies both the properties of a class, and the functions that it implements. Within the class, if we need to access a property of the class instance, we need to use the `this` keyword, as follows:

```
class SimpleClass {
    id: number | undefined;
    print(): void {
        console.log(`SimpleClass.id = ${this.id}`);
```

```
    }
}
```

Here, we have modified the `print` function to print the value of the `id` property to the console, within the template string, `${this.id}`. Note how we need to prefix the `id` property with the keyword `this` in order to access it within a class. We can now test this code as follows:

```
let mySimpleClass = new SimpleClass();
mySimpleClass.id = 2020;
mySimpleClass.print();
```

Here, we have modified our earlier code sample, and assigned the value of `2020` to the `id` property of the variable `mySimpleClass`. We then call the `print` function of this class, which will give us the following output:

```
SimpleClass.id = 2020
```

Here, the `print` function is accessing the value of the internal class property named `id`, and correctly printing the value of `2020` to the console.

Implementing interfaces

There is a very strong relationship between classes and interfaces, particularly in object-oriented design patterns. An interface describes a custom type and can include both properties and functions. A class is the definition of an object, also including its properties and functions. This allows us to use interfaces to describe some common behavior within a set of classes, and write code that will work with this set of classes. As an example of this, consider the following class definitions:

```
class ClassA {
    print(): void {
        console.log(`ClassA.print() called.`)
    };
}

class ClassB {
    print(): void {
        console.log(`ClassB.print() called.`)
    };
}
```

Here, we have class definitions for two classes, named `ClassA` and `ClassB`. Both of these classes have a `print` function. Now suppose that we want to write some code that did not really care what type of class we used, all it cared about was whether the class implemented a `print` function. We can easily create an interface describing the behavior we need, as follows:

```
interface IPrint {
    print(): void;
}

function printClass(a: IPrint) {
    a.print();
}
```

Here we have defined an interface named `IPrint` that contains a single function named `print` and returns `void`. We then have a function named `printClass` that takes a single parameter named `a`, and is of type `IPrint`. We are using the interface `IPrint` in this function definition to describe the attributes of an object that can be passed in as a parameter. We can now update our class definitions, and mark them as being usable by the `printClass` function, as follows:

```
class ClassA implements IPrint {
    print(): void {
        console.log(`ClassA.print() called.`)
    };
}

class ClassB implements IPrint {
    print(): void {
        console.log(`ClassB.print() called.`)
    };
}
```

Here, we are using the TypeScript keyword `implements` to state that both `ClassA` and `ClassB` implement the `IPrint` interface, and are therefore usable by our `printClass` function. We can now use this function as follows:

```
let classA = new ClassA();
let classB = new ClassB();

printClass(classA);
printClass(classB);
```

Here, we have created an instance of each of our classes, and assigned them to the variables `classA` and `classB`. We then pass these variables into our `printClass` function as arguments. The output of this code is as follows:

```
ClassA.print() called.
ClassB.print() called.
```

Interfaces, therefore, can be seen as a type of contract that classes must implement, if they are expected to provide certain properties and certain behaviors.

Note that TypeScript's duck typing rules will ensure that a particular class has the correct shape when used, even if it does not implement the interface as described. As an example of this, consider the following code:

```
class ClassC {
    print(): void {
        console.log(`ClassC.print() called.`)
    };
}

let classC = new ClassC();
printClass(classC);
```

Here, we have defined a class named `ClassC` that has the `print` function that is required by the `IPrint` interface, but it does not explicitly state that it implements the `IPrint` interface. Note how we are missing the code `implements IPrint`. We are still able to use the instance of this class, named `classC`, however, in a call to the `printClass` function, on the last line of this snippet. This is a further example of TypeScript's duck typing rules, making sure that the shape of the type used is correct for a call to the `printClass` function. Defining interfaces, and using them in our code, however, ensures that when changes are made to class definitions, or interface definitions, we are able to trap any possible errors early.

Class constructors

Class constructors can accept arguments during their initial construction. This allows us to combine the creation of a class and the setting of its parameters into a single line of code. Consider the following class definition:

```
class ClassWithConstructor {
    id: number;
    constructor(_id: number) {
        this.id = _id;
    }
}
```

Here, we have defined a class named `ClassWithConstructor` that has a single property named `id` of type number. We then have a function definition for a function named `constructor`, with a single parameter named `_id` of type number. Within this constructor function, we are setting the value of the internal `id` property to the value of the `_id` parameter that was passed in.

Note the definition of the `id` property in this class. Previously, we needed to define a property as being of the type `number | undefined`. Here, however, because we are initializing the property via the constructor, it will always be defined when an instance of the class is initialized.

We can construct an instance of this class as follows:

```
let classWithConstructor = new ClassWithConstructor(10);

console.log(`classWithConstructor =
    ${JSON.stringify(classWithConstructor)}`);
```

Here, we have created an instance of the `ClassWithConstructor` class, and passed in the argument of `10` to the constructor. We then simply log the value of this class instance to the console. The output of this code is as follows:

```
classWithConstructor = {"id":10}
```

Another aspect to note about class constructors, is that TypeScript understands the scoping rules of class properties, and the scoping rules of parameters. Our earlier class definition can be rewritten slightly as follows:

```
class ClassWithConstructor {
    id: number;
    constructor(id: number) {
        this.id = id;
    }
}
```

Here, we have the same class definition as used earlier, with a subtle difference. Note how the constructor function now accepts a parameter named `id`, instead of `_id`, as we used earlier. This is possible, as the scope of the `id` parameter, as passed into the constructor function, is different to scope of the class instance property named `id`. Using `this.id` to access the class instance property clearly distinguishes it from the `id` function parameter.

Class modifiers

TypeScript introduces the `public` and `private` access modifiers to indicate whether a class variable or function can be accessed from outside the class itself. Additionally, we can also use the `protected` access modifier, which we will discuss a little later.

A public property can be accessed by any calling code, as follows:

```
class ClassWithPublicProperty {
    public id: number | undefined;
}

let publicAccess = new ClassWithPublicProperty();
publicAccess.id = 10;
```

Here, we have defined a class named `ClassWithPublicProperty`, which has a single property named `id` that has been marked as `public`. This means that we are able to set the value of the `id` property once we have created an instance of this class, which in this instance has been set to the value `10`.

Let's now explore how marking a property `private` will affect access to this property, as follows:

```
class ClassWithPrivateProperty {
    private id: number;
    constructor(id: number) {
        this.id = id;
    }
}

let privateAccess = new ClassWithPrivateProperty(10);
privateAccess.id = 20;
```

Here, we have defined a class named `ClassWithPrivateProperty` that has a single property named `id`, which has been marked as `private`. This class also has a constructor function, which will set the value of the internal `id` property to the value of the `id` argument that was passed in. We then create an instance of this class, and attempt to assign the value of `20` to the private `id` property. This code will generate the following error:

```
error TS2341: Property 'id' is private and only accessible within class
'ClassWithPrivateProperty'
```

Here, we can see that because we have marked the `id` property as private, we cannot access it outside the class itself. Note, however, that within the constructor function, which is inside the class itself, we are able to set the value of the private `id` property.

Class functions and properties are `public` by default. The use of class access modifiers is a tool that we can use when writing TypeScript code and helps to protect variables from accidental assignment. These access modifiers, however, will not appear in the JavaScript that is generated from our code. The compiler will, in fact, remove any of these constraints when generating JavaScript.

JavaScript private fields

An experimental proposal to the ECMAScript standard introduces the concept of a private field, by using the hash (`#`) symbol before a property name. This means that if we are targeting a runtime that supports it, such as Node v12, we can write a JavaScript class as follows:

```
class ClassES6Private {
    #id: number;
    constructor(id: number) {
        this.#id = id;
    }
}

let es6PrivateClass = new ClassES6Private(10);
es6PrivateClass.#id = 20;
```

Here, we have used the hash symbol as a prefix to the `id` property in our class definition, which has become `#id`. The rest of the class is identical to the previous class definition that used the `private` keyword, and will work in the same way. Note that on the last line of this code snippet, we are attempting to assign the value of `20` to the private `#id` property of our class instance. This will generate the following error:

```
error TS18013: Property '#id' is not accessible outside class
'ClassES6Private' because it has a private identifier.
```

There are pros and cons to using TypeScript's access modifiers versus ES6 private fields. Thus far, we have only worked with the `public` and `private` modifiers, but TypeScript enhances these modifiers with both `readonly` and `protected`, which we will cover a bit later.

Just remember that the access modifiers used in TypeScript are compiled away in the resulting JavaScript. JavaScript private fields, however, will still translate into the generated JavaScript.

Constructor parameter properties

TypeScript also introduces a shorthand version for access modifiers that can be applied to parameters in a constructor function. As an example of this, consider the following code:

```
class ClassWithCtorMods {
    constructor(public id: number, private name: string) {
    }
}

let myClassMod = new ClassWithCtorMods(1, "test");
console.log(`myClassMod.id = ${myClassMod.id}`);
console.log(`myClassMod.name = ${myClassMod.name}`);
```

Here, we have defined a class named `ClassWithCtorMods` that has a single constructor function. This constructor function has two parameters. The first is named `id` and is of type number, and the second is named `name`, and is of type string. Note, however, that we have marked the `id` property as `public` within the constructor function definition, and we have marked the `name` property as `private`. This shorthand automatically creates an internal `id` property, and a `name` property on the class itself, which can be used as standard properties.

We then create an instance of this class, and assign it to a variable named `myClassMod`. We are then logging the value of both the `id` property and the `name` property to the console. This code, however, will generate the following error:

```
error TS2341: Property 'name' is private and only accessible within
class 'ClassWithCtorMods'
```

This error is telling us that the automatically created property named `name` is marked as `private` and is, therefore, not accessible outside of the class itself.

This shorthand syntax is only available for use within the constructor function itself, and not in any other functions of a class.

Readonly

In addition to the public and private access modifiers, we can also mark a class property as `readonly`. This is similar to the concept of the `const` keyword, and means that once a value has been assigned to a `readonly` property, it is not allowed to be modified, as follows:

```
class ClassWithReadonly {
    readonly name: string;
    constructor(_name: string) {
        this.name = _name;
    }
    setNameValue(_name: string) {
        this.name = _name;
    }
}
```

Here, we have a class named `ClassWithReadonly` that has a single property named `name`, of type string, that has been marked as `readonly`. We then have a `constructor` function, that accepts a single parameter named `_name`. Note how we are assigning the value of the incoming `_name` parameter to the internal `readonly name` property of our class. `readonly` properties are only allowed to set within the constructor function.

We then have a function named `setNameValue` that is attempting to set the value of the internal `name` property, but is not a constructor function. This code will generate the following error:

```
error TS2540: Cannot assign to 'name' because it is a read-only
property.
```

This error occurs when we attempt to assign a value to the `name` property within the `setNameValue` member function. As this error shows, we can only set a value for a `readonly` property within the class constructor function.

Note that `readonly` can also be used within interface definitions, and that it is also excluded from the generated JavaScript.

Get and set

ECMAScript 5 introduced the concept of property accessors, or get and set functions. A property accessor is simply a function that is called when a user of our class gets the value of a property, or sets its value.

By using a function instead of a simple property, we can detect when someone modifies or accesses a property, which we can use to trigger other logic. Consider the following example:

```
class ClassWithAccessors {
    private _id: number = 0;
    get id(): number {
        console.log(`get id property`);
        return this._id;
    }
    set id(value: number) {
        console.log(`set id property`);
        this._id = value;
    }
}
```

Here, we have defined a class named `ClassWithAccessors` that has a single private property named `_id`, of type number. We then define a function named `id`, that logs a message to the console, and then returns the value of the `_id` property. Note that this function has been marked as a `get` function, using the `get` keyword. We then define a `set` function, also with the name `id`, that will set the value of the internal `_id` property.

Note that the `get` function and the `set` function are both named `id`. We can now use this class as follows:

```
let classWithAccessors = new ClassWithAccessors();
classWithAccessors.id = 10;
console.log(`classWithAccessors.id = ${classWithAccessors.id}`);
```

Here, we have created a variable named `classWithAccessors`, which holds an instance of the `ClassWithAccessors` class. This class instance exposes what looks like a normal property named `id`. We set the value of this property to `10`, and then print the value of the property to the console.

The `get` and `set` functions, therefore, are exposing what looks like a property to the outside world, named `id`, but internally within the class, they are actually functions and not properties. The output of this code is as follows:

```
set id property
get id property
classWithAccessors.id = 10
```

Here, we can indeed see that the `get` and `set` functions were invoked, even though, to users of this class, it looks like a single `id` property.

Static functions

A class can mark a function with the `static` keyword, meaning that there will only be a single instance of this function available throughout the code base. When using a static function, we do not need to create an instance of the class in order to invoke this function, as follows:

```
class StaticFunction {
    static printTwo() {
        console.log(`2`)
    }
}

StaticFunction.printTwo();
```

Here, we have the definition of a class named `StaticFunction`. This class has a single function name `printTwo` that has been marked as `static`. This function simply prints the value `2` to the console.

Note, however, the way that we use this function. We do not need to create an instance of this class using the new keyword. We simply call this function by its fully qualified name, that is, `<className>.<functionName>`, which in this case is `StaticFunction.printTwo()`.

Static properties

In a similar manner to static functions, classes can also have static properties. If a class property has been marked as `static`, then there will only be a single instance of this property throughout the code base. Consider the following code:

```
class StaticProperty {
    static count = 0;
    updateCount() {
        StaticProperty.count++;
    }
}
```

Here, we have defined a class named `StaticProperty` that has a single property named `count` that has been marked as `static`. We have then defined a function named `updateCount` that will increase the value of the static `count` property by one each time it is called. Consider how this class is used, as follows:

```
let firstInstance = new StaticProperty();
let secondInstance = new StaticProperty();
```

```
firstInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);

secondInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
```

Here, we have created two variables named `firstInstance` and `secondInstance` that hold instances of the `StaticProperty` class. We then call the `updateCount` function on the `firstInstance` variable, and log the value of the `count` property of the `StaticProperty` class to the console.

We then use the `secondInstance` variable to again call the `updateCount` function, this time on another instance of the `StaticProperty` class. The output of this code is as follows:

```
StaticProperty.count = 1
StaticProperty.count = 2
```

Here, we can see that both calls to the `updateCount` function, whether it be from the `firstInstance` variable, or the `secondInstance` variable, have updated the value of the static property `count`. This shows that if a class property has been marked as static, then there will only be a single instance of this property throughout the code base.

Namespaces

When working within large projects, and particularly when working with large numbers of external libraries, there may come a time when two classes or interfaces share the same name. TypeScript uses what are known as namespaces to cater for these situations, as follows:

```
namespace FirstNameSpace {
    export class NameSpaceClass {}
    class NotExported {}
}
```

Here, we have used the `namespace` keyword to create a namespace named `FirstNameSpace`. This namespace covers all class or interface definitions that fall within its code block. Within the `FirstNameSpace` code block, we are defining two classes, named `NameSpaceClass` and `NotExported`. Note how we have used the `export` keyword for the `NameSpaceClass`.

The `export` keyword will make this class available outside of the namespace itself. As an example of this, consider the following code:

```
let nameSpaceClass = new FirstNameSpace.NameSpaceClass();
let notExported = new FirstNameSpace.NotExported();
```

Here, we have created two variables, named `nameSpaceClass` and `notExported`. We then create instances of the classes held within the `FirstNameSpace` namespace. Note how we need to prefix the class name that we are trying to instantiate with the name of the namespace itself, that is, `FirstNameSpace.NameSpaceClass`. This code, however, will generate the following error:

```
error TS2339: Property 'NotExported' does not exist on type 'typeof
FirstNameSpace'
```

This error is clearly telling us that the `NotExported` class is not made available for use outside of the namespace, as it has not been marked with the `export` keyword.

By making us refer to a class that is within a namespace by its fully qualified name, the compiler will ensure that we have unique class names across our code base.

In this section of the chapter, we have covered the usage of classes, how they are constructed, and how they can be used. We took a look at constructor functions, constructor parameter properties, and the use of getter and setter functions. We then explored static functions and static properties. In the next section of this chapter, we will explore inheritance.

Inheritance

Inheritance is another paradigm that is one of the cornerstones of object-oriented programming. Inheritance allows us to create an object based on another object, thereby inheriting all of its characteristics, including properties and functions. In TypeScript, inheritance can be used by classes and interfaces. In this section of the chapter, we will take a closer look at inheritance, and what it means for both classes and interfaces.

When discussing inheritance, it is important to clearly distinguish between the class that forms the basis of the inheritance structure, and the class that is doing all of the inheriting. We will use the term "base class," or "base interface," to denote the class or interface that forms the base of the inheritance structure, and the term "derived class," or "derived interface," to denote the class or interface that is doing the inheriting.

TypeScript uses the keyword `extends` to implement inheritance.

Interface inheritance

One interface can form the base interface for one or many other interfaces. As an example of this, consider the following code:

```
interface IBase {
    id: number;
}

interface IDerivedFromBase extends IBase {
    name: string;
}

class IdNameClass implements IDerivedFromBase {
    name: string = "nameString";
}
```

Here, we have defined an interface named `IBase` that has a single property, named `id` of type number. We then define a second interface named `IDerivedFromBase` that is using the `extends` keyword to inherit from `IBase`. We then define a class named `IdNameClass` that implements the `IDerivedFromBase` interface, and has a single property named `name` of type string. Note that this code will generate the following error:

```
error TS2420: Class 'IdNameClass' incorrectly implements interface
'IDerivedFromBase'.
  Property 'id' is missing in type 'IdNameClass' but required in type
'IDerivedFromBase'.
```

Here, we can see that the compiler is telling us that the `IDerivedFromBase` interface has both a `name` and an `id` property, as it inherits the `id` property from the `IBase` interface. As we have not defined the `id` property in our class definition, the class is not correctly implementing the interface, hence the error. We can fix this class definition as follows:

```
class IdNameClass implements IDerivedFromBase {
    id: number = 0;
    name: string = "nameString";
}
```

Here, we have added the `id` property on the class definition for `IdNameClass`, and are therefore correctly implementing the `IDerivedFromBase` interface.

When using interface inheritance, we can also narrow a type on a derived interface. Consider the following two interfaces:

```
interface IBaseStringOrNumber {
    id: string | number;
}

interface IDerivedFromBaseNumber
    extends IBaseStringOrNumber {
    id: number;
}
```

Here, we have an interface named `IBaseStringOrNumber`, which has a single property named `id` that is of type string or number. We then define a second interface that derives from it named `IDerivedFromBaseNumber`, which has a single property named `id` that is of type number. So in essence, we have narrowed the type of the `id` property from string or number to just number in our derived interface.

Interfaces also support multiple inheritance. Consider the following code:

```
interface IMultiple extends
    IDerivedFromBase,
    IDerivedFromBaseNumber
{
    description: string;
}

let multipleObject: IMultiple = {
    id: 1,
    name: "myName",
    description: "myDescription"
};
```

Here, we have defined an interface named `IMultiple`, which derives from both the `IDerivedFromBase` and the `IDerivedFromBaseNumber` interfaces, using a comma separated list. It also defines a property named `description` of type string. We then define an object named `multipleObject` that implements this interface. According to our inheritance rules, this `IMultiple` interface therefore has three properties – an `id` property of type number from the `IDerivedFromBaseNumber` interface, a `name` property from the `IDerivedFromBase` interface, and also the `description` property. This `multipleObject` object, therefore, must provide a property for each of the properties found in the interface.

Class inheritance

Classes can also use the `extends` keyword to create an inheritance structure. Using our existing definitions for the `IBase` and `IDerivedFromBase` interfaces, the following code shows an example of class inheritance:

```
class BaseClass implements IBase {
    id: number = 0;
}

class DerivedFromBaseClass
    extends BaseClass
    implements IDerivedFromBase
{
    name: string = "nameString";
}
```

Here, we have defined a class named `BaseClass` that implements the `IBase` interface, and therefore must define an `id` property of type number. We then have a class definition for a class named `DerivedFromBaseClass`, which inherits from `BaseClass` (using the `extends` keyword), and also implements the `IDerivedFromBase` interface. As `BaseClass` already has an `id` property, the only other property that `DerivedFromBaseClass` needs to implement is the `name` property.

JavaScript does not support the concept of multiple inheritance when it comes to classes. This means that a class can only inherit from one other class. Interfaces, however, do support multiple inheritance, as we have seen earlier.

A class, however, can implement multiple interfaces, as follows:

```
interface IFirstInterface {
    id: number;
}

interface ISecondInterface {
    name: string;
}

class MultipleInterfaces implements
    IFirstInterface,
    ISecondInterface
{
    id: number = 0;
    name: string = "nameString";
}
```

Here, we have two interfaces that are independent of each other. `IFirstInterface` has a single property named `id` of type number, and `ISecondInterface` has a single property named `name` of type string. We then define a class named `MultipleInterfaces` that implements both interfaces, by simply naming them in a comma separated list.

As the `MultipleInterfaces` class implements both independent interfaces, it must define both an `id` property and a `name` property.

The super function

When using inheritance, it is quite common for a base class and a derived class to implement the same method. This is seen most often with class constructors. If a derived class has a constructor, then this constructor must call the base class constructor using the `super` keyword, or TypeScript will generate an error, as follows:

```
class BaseClassWithCtor {
    private id: number;
    constructor(id: number) {
        this.id = id;
    }
}

class DerivedClassWithCtor extends BaseClassWithCtor {
    private name: string;
    constructor(id: number, name: string) {
        super(id);
        this.name = name;
    }
}
```

Here, we have defined a class named `BaseClassWithCtor` that has a single property named `id` of type number, and also defines a constructor function that initializes this `id` property.

We then define a class named `DerivedClassWithCtor` that inherits from the `BaseClassWithCtor` class. Note how we are calling the constructor for the base class, `BaseClassWithCtor`, within the constructor of the `DerivedClassWithCtor` class, by using the `super` function. This `super` call is also passing the value that it is receiving in the `id` parameter through to the base class constructor.

Note that even if a base class does not define a constructor function, if the derived class does define a constructor function, then the derived class must call the `super` function with no arguments.

Function overriding

When a derived class defines a method that has the same name as a base class method, this technique is known as **function overriding**. The derived class can determine whether or not to call the implementation of the function in the base class. Let's take a look at this technique, as follows:

```
class BaseClassWithFn {
    print(text: string) {
        console.log(`BaseClassWithFn.print() : ${text}`)
    }
}

class DerivedClassFnOverride extends
    BaseClassWithFn {
    print(text: string) {
        console.log(`DerivedClassFnOverride.print(${text})`);
    }
}
```

Here, we have defined a class named `BaseClassWithFn` that has a single `print` function, with a single parameter named `text` of type string. We then define a class named `DerivedClassFnOverride` that inherits from the `BaseClassWithFn` class, and also has a single `print` function with the same parameters and types. The derived class's implementation of the `print` function overrides the base class implementation of the `print` function, as can be seen by the behavior of the following code:

```
let derivedClassFnOverride = new DerivedClassFnOverride();
derivedClassFnOverride.print("test");
```

Here, we have created a new instance of the `DerivedClassFnOverride` class, and called its `print` function with the argument `"test"`. The output of this code is as follows:

```
DerivedClassFnOverride.print(test)
```

As we can see from this output, the derived class's implementation of the `print` function is being called, and is overriding the base class implementation.

We can, however, still call the base class's function implementation using the `super` keyword, as follows:

```
class DerivedClassFnCallthrough extends
    BaseClassWithFn
{
    print(text: string) {
        super.print(`from DerivedClassFncallthrough : ${text}`);
    }
}

let derivedCallthrough = new DerivedClassFnCallthrough();
derivedCallthrough.print("text");
```

Here, we have defined a class named `DerivedClassFnCallthrough` that inherits from the `BaseClassWithFn` class, and also has a function override for the `print` function. The implementation of this `print` function, however, calls through to the base class implementation by using the `super` keyword, that is, `super.print`.

To test that this works, we have created an instance of the `DerivedClassFnCallthrough` class named `derivedCallThrough` and called the `print` function. The output of this code is as follows:

```
BaseClassWithFn.print() : from DerivedClassFncallthrough : text
```

Here, we can clearly see that the base class's `print` function has been invoked by the derived class's `print` function, through the use of the `super` keyword.

Protected

Classes can mark both properties and functions with the `protected` keyword. If a property is marked as protected, then it is not accessible outside of the class itself, similar to the behavior of the `private` keyword. It is, however, accessible to derived classes, which is different to private variables that are not accessible to derived classes, as can be seen in the following example:

```
class BaseClassProtected {
    protected id: number;
    private name: string = "";
    constructor(id: number) {
        this.id = id;
    }
```

```
}

class AccessProtected extends BaseClassProtected {
    constructor(id: number) {
        super(id);
        console.log(`base.id = ${this.id}`);
        console.log(`base.name = ${this.name}`)
    }
}
```

Here, we have defined a class named `BaseClassProtected` that has a protected property named `id` of type number, and a private property named `name` of type string. We are setting the value of the protected `id` property within the constructor. We then define a class named `AccessProtected` that derives from the `BaseClassProtected` class. This class has a constructor function that is passing the `id` argument down into the base class constructor. This code will produce the following error:

```
error TS2341: Property 'name' is private and only accessible within
class 'BaseClassProtected'
```

Here, we can see that within the constructor function of the `AccessProtected` class, we are attempting to access the protected `id` property of the base class, and then we are attempting to access the private `name` property of the base class. Access to the protected property is allowed, but access to the private `name` property is not.

Let's also test this theory outside of the classes themselves, as follows:

```
let accessProtected = new AccessProtected(1);
accessProtected.id = 1;
accessProtected.name = "test";
```

Here, we have a variable named `accessProtected` that holds an instance of the `AccessProtected` class. We then attempt to assign values to the `id` property and `name` property of this class. This code will generate the following errors:

```
error TS2445: Property 'id' is protected and only accessible within
class 'BaseClassProtected' and its subclasses
error TS2341: Property 'name' is private and only accessible within
class 'BaseClassProtected'
```

These error messages are clearly telling us that the `id` property is protected, and therefore only accessible within the `BaseClassProtected` class, and any class that is derived from it. The `name` property, however, is private, and is therefore only accessible within the `BaseClassProtected` class itself. Keep in mind that these access modifiers are TypeScript constructs, and again, these will be "compiled away" in the generated JavaScript.

Abstract classes

An abstract class is a class that cannot be instantiated. In other words, it is a class that is designed to be derived from. The purpose of abstract classes is generally to provide a set of basic properties or functions that are shared across a group of similar classes. Abstract classes are marked with the `abstract` keyword.

Let's take a look at the use of an abstract class, as follows:

```
abstract class EmployeeBase {
    public id: number;
    public name: string;
    constructor(id: number, name: string) {
        this.id = id;
        this.name = name;
    }
}
class OfficeWorker extends EmployeeBase {
}

class OfficeManager extends OfficeWorker {
    public employees: OfficeWorker[] = [];
}
```

Here, we have defined an abstract class named `EmployeeBase` that has an `id` property of type number, and a `name` property of type string. The `EmployeeBase` class has a constructor function that initializes these properties. We have then defined a class named `OfficeWorker` that derives from the abstract `EmployeeBase` class. Finally, we have a class named `OfficeManager` that derives from the `OfficeWorker` class, and has an internal property named `employees` that is an array of type `OfficeWorker`. We can construct instances of these classes as follows:

```
let joeBlogg = new OfficeWorker(1, "Joe");
let jillBlogg = new OfficeWorker(2, "Jill")
let jackManager = new OfficeManager(3, "Jack");
```

Here, we have created two variables named `joeBlogg` and `jillBlogg` that are of type `OfficeWorker`. We have also created a variable name `jackManager` that is of type `OfficeWorker`. Note how we need to provide two arguments when constructing these objects. This is because both the `OfficeWorker` class and the `OfficeManager` class inherit from the abstract base class, `EmployeeBase`. The abstract base class `EmployeeBase` requires both an `id` and a `name` argument in its constructor.

Abstract classes are designed to be derived from. They provide a convenient method of sharing common properties and functions between groups of objects.

Abstract class methods

An abstract class method is similar to an abstract class, in that it is designed to be overridden. In other words, declaring a class method as `abstract` means that a derived class must provide an implementation of this method. For this reason, abstract class methods are not allowed to provide a function implementation. As an example of this, let's update our `EmployeeBase` class as follows:

```
abstract class EmployeeBase {
    public id: number;
    public name: string;
    abstract doWork(): void;
    constructor(id: number, name: string) {
        this.id = id;
        this.name = name;
    }
}
```

Here, we have added a class method named `doWork` that has been marked as `abstract`. This means that we will need to provide an implementation of this method in any derived class, as follows:

```
class OfficeWorker extends EmployeeBase {
    doWork() {
        console.log(`${this.name} : doing work`);
    }
}
```

Here we have updated the `OfficeWorker` class, which derives from the `EmployeeBase` class, and provided an implementation of the `doWork` abstract method. This function simply logs a message to the console.

Let's now update our `OfficeManager` class, as follows:

```
class OfficeManager extends OfficeWorker {
    public employees: OfficeWorker[] = [];
    manageEmployees() {
        super.doWork();
```

```
        for (let employee of this.employees) {
            employee.doWork();
        }
    }
}
```

Here, we have added a method named `manageEmployees` to the `OfficeManager` class. This method calls the `doWork` method, which is defined on the `OfficeWorker` base class, and therefore we need to use the `super` keyword to access it. It then loops through each of the elements in the `employees` array, and calls the `doWork` method on each of these classes. We can use these updated classes as follows:

```
jackManager.employees.push(joeBlogg);
jackManager.employees.push(jillBlogg);

jackManager.manageEmployees();
```

Here, we have added the `joeBlogg` and `jillBlogg` instances of the `OfficeWorker` class to the `employees` array on the `jackManager` instance of the `OfficeManager` class. We then call the `manageEmployees` function on the `jackManager` instance. This code will produce the following output:

```
Jack : doing work
Joe : doing work
Jill : doing work
```

Here, we can see the output of each of the `doWork` abstract method calls. As the `OfficeManager` class inherits from the `OfficeWorker` class, the `doWork` method is available on this class instance as well. The `OfficeManager` class simply calls its own instance of the `doWork` method, and then iterates through each element in its `employees` array, calling the `doWork` method on each instance.

instanceof

JavaScript provides the `instanceof` operator to test whether the given function name appears in the prototype of an object. In TypeScript terms, the use of this keyword allows us to detect whether an object is an instance of a class, or whether it has been derived from a particular class. This is best illustrated with an example, as follows:

```
class A { }
class BfromA extends A { }
class CfromA extends A { }
class DfromC extends CfromA { }
```

Here, we have defined four classes in a class hierarchy, starting with a simple class named `A`. We then define two classes that derive from class `A`, named `BfromA` and `CfromA`. We then define a final class that derives from `CfromA`, named `DfromA`, which means that it is also derived from `A` indirectly. Let's test how this hierarchy is interpreted by the `instanceof` operator, as follows:

```
console.log(`A instance of A :
    ${new A() instanceof A}`);

console.log(`BfromA instance of A :
    ${new BfromA() instanceof A}`);

console.log(`BfromA instance of BfromA :
    ${new BfromA() instanceof BfromA}`);
```

Here, we start with the simplest case, testing whether a new instance of the `A` class will be interpreted by the `instanceof` operator correctly. We then test if an instance of the `BfromA` class will also be seen as an instance of the `A` class, and finally if an instance of the `BfromA` class will be seen as an instance of the `BfromA` class. The output of this code is as follows:

```
A instance of A :
    true
BfromA instance of A :
    true
BfromA instance of BfromA :
    true
```

Here, we can see that the `instanceof` operator is returning `true` for all of these cases. This matches the class hierarchy that we have defined. Let's now test for a negative case, as follows:

```
console.log(`CfromA instance of BfromA :
    ${new CfromA() instanceof BfromA}`);
```

Here, we are checking whether the class `CfromA` is an instance of the class `BfromA`. Both classes are derived from `A`, but `CfromA` is not related to the class `BfromA`, and will produce the following output:

```
CfromA instance of BfromA :
    false
```

Here, we can see that the `CfromA` class is not derived from the `BfromA` class. Let's now check the results of inheriting from a class that inherits from another class, as follows:

```
console.log(`DfromC instance of CfromA :
    ${new DfromC() instanceof CfromA}`);

console.log(`DfromC instance of A :
    ${new DfromC() instanceof A}`);
```

Here, we are checking if the `DfromC` class can be seen as an instance of the `CfromA` class, and then we are checking if the `DfromC` class can be seen as an instance of the `A` class. The output of this code is as follows:

```
DfromC instance of CfromA :
    true
DfromC instance of A :
    true
```

Here, we can see that the `DfromC` class inherits from the `CfromA` class. As the `CfromA` class inherits from the `A` class, then the `DfromC` class also inherits from the `A` class.

Interfaces extending classes

On a final note with regard to interface and class definitions, note that an interface can derive from a class definition, as can be seen in the following example:

```
class BaseInterfaceClass {
    id: number = 0;
    print() {
        console.log(`this.id = ${this.id}`);
    }
}

interface IBaseInterfaceClassExt
    extends BaseInterfaceClass {
    setId(id: number): void;
}
```

Here, we have a class definition for a class named `BaseInterfaceClass` that has a single property named `id`, and a function named `print`. We then define an interface named `IBaseInterfaceClassExt` that derives from the `BaseInterfaceClass` using the `extends` keyword.

This interface then defines a function named `setId` that takes a single parameter named `id` of type number. So we have derived an interface directly from a class definition, and added a function definition. Note that we can then use this interface in the same way that we use any other type of interface, as follows:

```
class ImplementsExt extends BaseInterfaceClass
    implements IBaseInterfaceClassExt {
    setId(id: number): void {
        this.id = id;
    }
}
```

Here, we have a class named `ImplementsExt` that derives from the base class named `BaseInterfaceClass`, and also implements the `IBaseInterfaceClassExt` interface. This class, therefore, must provide an implementation of the `setId` function that was defined by the `IBaseInterfaceClassExt` interface.

In this section of the chapter, we have discussed inheritance as it applies to interfaces and classes. We have seen how to use inheritance with the `extends` keyword, and discussed how to call class methods in base classes with the `super` keyword. We also covered abstract classes, and abstract class methods. In the last section of this chapter, we will take a look at modularization.

Modules

Modularization is a popular technique used in programming languages that allows programs to be built from a series of smaller libraries, or modules. This technique is also applied to object-oriented code bases, where each class is typically housed in its own file. When referencing classes that exist in another source file, we need a mechanism for the TypeScript compiler, as well as the JavaScript runtime, to be able to locate the class that we are referencing. This is where modules are used.

TypeScript has adopted the module syntax that is part of ES2015. This means that we can use this syntax when working with modules, and the compiler will generate the relevant JavaScript to support modules based on the target JavaScript version we have selected. There is no change in syntax, and there is no change in our code in order to support earlier versions of the JavaScript runtime.

In this section of the chapter, we will walk through the module syntax, and discuss how to write our code to support modules.

Exporting modules

There are two things that we need in order to write and use modules. Firstly, a module needs to expose something to the outside world in order for it to be consumed. This is called exporting a symbol from a module, and uses the TypeScript keyword `export`, similar to what we saw with namespaces. As an example of this, let's create a source directory named `modules`, and within this `modules` directory, a file name `Module1.ts` containing the following code:

```
export class Module1 {
    print(): void {
        localPrint(`Module1.print() called`);
    }
}

function localPrint(text: string) {
    console.log(`localPrint: ${text}`);
}
```

Here, we have defined a class named `Module1` that is using the `export` keyword to mark it as consumable outside of this file. This class has a single function named `print`, which is calling another function named `localPrint` to print a message to the console. We then have the definition of the `localPrint` function, which takes a single parameter named `text`, of type string, and logs a message to the console.

What is important about the `localPrint` function is that it has not been marked with the `export` keyword. This means that the `localPrint` function is not available outside of the `Module1.ts` module, and has a private scope. Another point to note with this file is that even if the `Module1` class is used in another file, it will still have access to its privately scoped `localPrint` function. We will demonstrate this in the next section.

Importing modules

In order to consume a symbol that has been exported, any source file that needs this module must import it using the `import` keyword. Let's test this `import` syntax by creating a `modules_main.ts` class in our project's base directory as follows:

```
import { Module1 } from "./modules/Module1";

let mod1 = new Module1();
mod1.print();
```

Here, we start by using the `import` keyword to import the definition of the `Module1` class from the `modules/Module1` file. Note the syntax that we are using. Following the `import` statement is a name in braces `{ Module1 }`, which is the name that we will use for the imported class. Note too, that we do not specify a `.ts` or a `.js` extension when importing modules. The module loader will take care of locating the correct file on disk.

Once the class has been imported, we can use the class definition of `Module1` as normal. We create an instance of this class named `mod1`, and then call the `print` function. The output of this code is as follows:

```
localPrint: Module1.print() called
```

Here, we can see that the print function on the `Module1` class is indeed calling the `localPrint` function that was defined within the `Module1.ts` file, which was privately scoped. Even though we only imported the `Module1` class itself, this class still has access to any privately scoped functions defined in its source file.

Module renaming

When importing a symbol from a module, we can also choose the name that we want to use when referencing the exported symbols within it, as follows:

```
import { Module1 as MyMod1 } from "./modules/Module1";

let myRenamedMod = new MyMod1();
myRenamedMod.print();
```

Here, we have another `import` statement that is importing the class from the `modules.Module1` file. Note, however, that we have used the `as` keyword when specifying the module name, that is, `{ Module1 as MyMod1 }`. This means that we have renamed the class that has been exported from `Module1` to `MyMod1` within this file.

We then use this renamed class name to create an instance of the `MyMod1` class (which really is the `Module1` class), named `myRenamedMod`. We then call the `print` function on this class instance. The output of this code is the same, as follows:

```
localPrint: Module1.print() called
```

Here, we can see that even though we have renamed the class itself, the behavior of this class has not changed.

While module renaming is allowed in the module syntax, it's standard practice not to do it at all. The act of reading code, and trying to understand it, means that we start to build a mental model of where each class or interface is defined. Renaming a module within a specific file just makes the act of connecting a class name to a particular file name all that more difficult. Module renaming should only be used in exceptional circumstances, and for a very good reason.

Multiple exports

When working with external libraries, that is, libraries that have been published for general consumption, it is common practice for a published library to export all classes, functions, and interfaces from a single module file. This means that we do not need to know how this library is structured, or how complex the class hierarchy is internally; we simply import the entire library from one file. As an example of this technique, let's create a file named `modules/MultipleExports.ts`, as follows:

```
export class MultipleClass1 {}

export class MultipleClass2 {}
```

Here, we are exporting two classes from this module file, which are named `MultipleClass1` and `MultipleClass2`. We can then import both of these classes in our `modules_main.ts` file as follows:

```
import { MultipleClass1, MultipleClass2 }
    from "./modules/MultipleExports";

let mc1 = new MultipleClass1();
let mc2 = new MultipleClass2();
```

Here, we are using the standard `import` syntax as we have seen earlier, but have included both of the `MultipleClass1` and `MultipleClass2` classes as named classes in our `import` statement, in a comma separated list. As we have named both of these classes on our `import`, we can create instances of these classes as normal.

Module namespaces

There is, however, another syntax that we can use to import multiple symbols from a module. This syntax will import all available exports from a module, without naming each of them individually, by attaching them to a namespace, as follows:

```
import * as MultipleExports from "./modules/MultipleExports";

let meMc1 = new MultipleExports.MultipleClass1();
let meMc2 = new MultipleExports.MultipleClass2();
```

Here, we are not importing individual classes from the `ModuleExports.ts` file by naming each of them one by one. We are instead importing everything that has been exported by using the `* as` syntax. Note that this technique will attach a namespace to this module, and all references to classes or interfaces within this module must use the namespace name. This can be seen in the last two lines of the code snippet. We are creating an instance of the `MultipleClass1` class and the `MultipleClass2` class, but in so doing, must reference them both using the namespace, that is, `MultipleExports.MultipleClass1` and `MultipleExports.MultipleClass2`.

Default exports

A module file can also mark an exported item as the default export for the file using the `default` keyword. While this is generally not used for class definitions, it is sometimes used for function definitions. Let's create a new file named `modules/DefaultExport.ts`, as follows:

```
export default function DefaultAdd(
    a: number, b: number) {
    return a + b;
}
export class ModuleNonDefaultExport { }
```

Here, we have exported a function named `DefaultAdd` that will return the sum of two numbers. This function has also been marked with the `default` keyword. Note that we are also exporting a class named `ModuleNonDefaultExport`. We can now import this default function as follows:

```
import DefaultAdd from "./modules/DefaultExport";

let modDefault = DefaultAdd(1, 2);
```

Here, we are using a very simple syntax for importing the `DefaultAdd` function. We do not need to use the curly braces, that is, `{` and `}`, but are instead targeting the default `export` function by name. We are then invoking the `DefaultAdd` function to add two numbers, and are assigning the return value to a variable named `modDefault`.

The `DefaultExport.ts` module, however, also exports a class named `ModuleNonDefaultExport`. We are still able to use this class, but it must be imported using the standard `import` syntax, as follows:

```
import DefaultAdd, { ModuleNonDefaultExport }
    from "./modules/DefaultExport";

let modNonDefault = new ModuleNonDefaultExport();
```

Here, we have updated our `import` statement to include the `ModuleNonDefaultExport` class using the standard `import` syntax that surrounds the class name with curly braces. We are then able to use the class as usual within our code.

Summary

We have covered a lot of ground in this chapter. We started with a discussion on interfaces, and how they allow us to define custom types that can be used to better describe objects with their designated properties and sub-properties. We then moved on to discuss classes, and how to use the various TypeScript language features that are specially designed with classes in mind. This discussion included class constructors, class modifiers, `readonly` properties, `get` and `set` functions, and static properties and methods. We then discussed the ins and outs of class and interface inheritance. We took a look at the `super` keyword, and how it can be used within class constructors as well as class methods in a class hierarchy. We then discussed abstract classes, abstract methods, the protected access modifier, and `instanceof`. In the final section of this chapter, we discussed modules, and how breaking up a large project into multiple smaller files can aid with code structure. We showed the various ways in which modules can be exported and then imported when needed.

In the next chapter, we will take a look at generics, and some advance type inference rules that TypeScript uses to manipulate types for various uses.

4

Generics and Advanced Type Inference

Thus far, we have been exploring the type system within TypeScript, and how it relates to interfaces, classes, and primitive types. We have also explored how to use various language features to mix and match these types, including type aliases and type guards. All of the techniques we have used, however, eventually boil down to writing code that will work with a single particular type. This is how we achieve type safety within TypeScript.

But what if we would like to write some code that will work with any sort of type, or any sort of interface or class definition? Perhaps a function that needs to find an element in a list, where the list could be made of strings, or numbers, or any other type. This is where generics come into play. Generics provide a mechanism to write code that does not need to specify a specific type. It is left up to the caller of these generic functions or classes to specify the type that the generic will be working with. There are, of course, some constraints that come into play when working with generics. How do we limit the types that generic code can work with, down to a small subset of classes or interfaces? Are we able to specify multiple types within generic code, and can we specify a relationship between these two generic types?

This chapter is broken up into two sections. The first section will discuss the generic type syntax that TypeScript uses, and the various ways that we can work with types when writing generic code.

We already know that TypeScript uses inferred typing in certain cases, in order to determine what the type of an object is, if we do not specify it explicitly. TypeScript also allows us to use advanced type inference when working with generic code.

In other words, when given a generic type, we can compute or construct another completely different type based on the properties and structure of the original type. This technique allows us to map one type to another, which we will discuss in the second section of this chapter.

We will cover the following topics in this chapter:

- Generic syntax, including:
    - Multiple generic types
    - Constraining and using the type `T`
    - Generic constraints and interfaces
    - Creating new objects within generics
- Advanced type inference, including:
    - Mapped types
    - Conditional types and conditional type chaining
    - Distributed conditional types
    - Conditional type inference
    - Type inference from function signatures and arrays
    - Standard conditional types

Let's get started with generics.

Generics

Generics, or, more specifically, generic syntax is a way of writing code that will work with a wide range of objects and primitives. As an example, suppose that we wanted to write a function that iterates over a given array of objects, and returns a concatenation of their values. So, given a list of numbers, say `[1,2,3]`, it should return the string `"1,2,3"`. Or, given a list of strings, say `["first", "second", "third"]`, it should return the string `"first, second, third"`.

Using generics allows us to write type-safe code that can force each element of the array to be of the same type, and as such would not allow a mixed list of values to be sent through to our function, say `[1,"second", true]`.

In this section of the chapter, we will introduce the generic code syntax, and explore the rules around what we can do with generic types.

Generic syntax

TypeScript uses an angled bracket syntax, and a type symbol, or type substitute name, to indicate that we are using generic syntax. In other words, to specify that the type named `T` is being used within generic syntax, we will write `<T>` to indicate that this code is substituting a normal type name with the symbol `T`. Then, when we use a generic type, TypeScript will essentially substitute the type named `T` with an actual type.

This is best explained through some example code, as follows:

```
function printGeneric<T>(value: T) {
    console.log(`typeof T is : ${typeof value}`);
    console.log(`value is : ${value}`)
}
```

Here, we have a function named `printGeneric` that is using generic syntax and specifying that its type substitute name is named `T` by appending `<T>` to the function name. This function takes a single parameter named `value`, of the type `T`.

So, what we have done here is replace the type within the function definition, which would normally be `value: string`, or `value: number`, for example, with the generic syntax of `value: T`.

This `printGeneric` function will log the result of the `typeof` operator for the value that was sent in, as well as its actual value. We can now use this function as follows:

```
printGeneric(1);
printGeneric("test");
printGeneric(true);
printGeneric(() => { });
printGeneric({ id: 1 });
```

Here, we are calling the `printGeneric` function with a wide range of values. The first call is with a numeric value of `1`, the second with a string value of `"test"`, and the third with a boolean value of `true`. We then call the `printGeneric` function with an actual function, and then with an object with a single property named `id`. The output of this code is as follows:

```
typeof T is : number
value is : 1
typeof T is : string
value is : test
typeof T is : boolean
```

```
value is : true
typeof T is : function
value is : function () { }
typeof T is : object
value is : [object Object]
```

As we can see from this output, the `printGeneric` function is indeed working with pretty much every type that we can throw at it. The `typeof` operator is identifying what the type of `T` is, either a string, number, boolean, function, or object, and the value that is logged to the console is correct.

Note that there is an interesting subtlety about how we have called this function. This is best explained by showing how we can also call this function as follows:

```
printGeneric<string>("test");
```

Here, we are using type casting notation, that is, the angled brackets `<type>`, to explicitly specify what type we are calling this function with. Note that previously, we did not explicitly set the type using this long form notation, but simply called the function with an argument, that is, `printGeneric(1)`. In this instance, TypeScript is inferring the type `T` to be a number.

Note, too, that if we explicitly set the type to be used using this long form notation, then this will override any usage of the type `T`, and our normal type rules will apply for any usage of the type `T`. Consider the following example:

```
printGeneric<string>(1);
```

Here, we are explicitly specifying that the function will be called with a type `<string>`, but our single argument is actually of type number. This code will generate the following error:

```
error TS2345: Argument of type '1' is not assignable to parameter of
type 'string'
```

This error is telling us that we are attempting to call a generic function with the wrong type as an argument, as the type of `T` was explicitly set.

If we do not explicitly specify what type the generic function should use, by omitting the `<type>` specifier, the compiler will infer the type to be used from the type of each argument.

Multiple generic types

We can also specify more than one type to be used in a generic function, as follows:

```
function usingTwoTypes<A, B> ( first: A, second: B) {
}
```

Here we have a function named `usingTwoTypes` that has a type name for both the first and second parameters in this function, that is, `A` and `B`. This function can specify any type for either `A` or `B`, as follows:

```
usingTwoTypes<number, string> ( 1, "test");
usingTwoTypes(1, "test");
usingTwoTypes<boolean, boolean>(true, false);
usingTwoTypes("first", "second");
```

Here, we are freely mixing the syntax we are using to call the `usingTwoTypes` function. The first call is using explicit type syntax, and therefore `A` is of type number, and `B` is of type string. The second call is inferring the type of `A` as a `number`, and the type of `B` as a `string`. In the third call, we are explicitly setting both types as a boolean, and in the fourth and last call, both types are inferred as strings.

Constraining the type of T

In most instances, we will want to limit the type of `T` in order to only allow a specific set of types to be used within our generic code. This is best explained through an example, as follows:

```
class Concatenator<T extends Array<string> | Array<number>> {
    public concatenateArray(items: T): string {
        let returnString = "";
        for (let i = 0; i < items.length; i++) {
            returnString += i > 0 ? "," : "";
            returnString += items[i].toString();
        }
        return returnString;
    }
}
```

Here, we have defined a class named `Concatenator` that is using generic syntax, and is also constraining the type of `T` to be either an array of strings or an array of numbers, via the `extends` keyword. This means that wherever `T` is used within our code, `T` can only be interpreted as either a string array or a number array. This class has a single function named `concatenateArray` that has a single parameter named `items`, of type `T`, and returns a `string`.

Within this function, we are simply looping through each element in the argument array named `items` and appending a string representation of the item to the variable `returnString`. We can now use this class as follows:

```
let concator = new Concatenator();

let concatResult = concator.concatenateArray([
    "first", "second", "third"
]);
console.log(`concatResult = ${concatResult}`);

concatResult = concator.concatenateArray([
    1000, 2000, 3000
]);
console.log(`concatResult = ${concatResult}`);
```

Here, we start by creating a new instance of the `Concatenator` class, and assigning it to the variable named `concator`. We then call the `concatenateArray` function with an array of strings, and assign the result to the variable named `concatResult`, which is of type string. We then print this value to the console.

We then call the `concatenateArray` function with an array of numbers, and again print the returned string value to the console. The results of this code are as follows:

```
concatResult = first,second,third
concatResult = 1000,2000,3000
```

Here, we can see that our `concatArray` function is returning a string representation of the array that we passed in as an argument.

If, however, we attempt to call this function with an array of booleans, as follows:

```
concatResult = concator.concatenateArray([
    true, false, true
]);
```

We will generate a number of errors, as follows:

```
error TS2322: Type 'true' is not assignable to type 'string | number'
error TS2322: Type 'false' is not assignable to type 'string | number'
error TS2322: Type 'true' is not assignable to type 'string | number'
```

Here, we can see that because we constrained the type of `T` in our `Concatenator` class to only allow arrays of strings or arrays of numbers, the compiler will not allow us to call this function with an array of booleans.

Using the type T

We have already seen how we can constrain the type of `T` in our generic code in order to limit the number of types that can be used. Another limit of generic code is that it can only reference functions or properties of objects that are common to any type of `T`. As an example of this limitation, consider the following code:

```
interface IPrintId {
    id: number;
    print(): void;
}

interface IPrintName {
    name: string;
    print(): void;
}
```

Here, we have two interfaces, named `IPrintId`, and `IPrintName`. Both interfaces have a function named `print` that returns `void`. The `IPrintId` interface, however, has a property named `id` of type number, and the `IPrintName` interface has a property named `name` of type string. These two properties are unique to each of these interfaces. Now let's consider a generic function that is designed to work with these two interfaces, as follows:

```
function useT<T extends IPrintId | IPrintName>(item: T)
    : void {
    item.print();
    item.id = 1;  // error : id is not common
    item.name = "test"; // error : name is not common
}
```

Here, we have defined a function named `useT` that accepts a type named `T` that can be either an instance of the `IPrintId` interface, or an instance of the `IPrintName` interface. The function has a single parameter named `item` of type `T`.

Within this function, we are calling the `print` method of the `item` parameter, and then we are attempting to assign the value of `1` to the `id` property, and a value of `"test"` to the `name` property of the `item` parameter.

This code will generate the following errors:

```
error TS2339: Property 'id' does not exist on type 'T'
error TS2339: Property 'name' does not exist on type 'T'
```

This error clearly indicates that the `id` and `name` property does not exist on the type `T`. In other words, we are only able to call properties or functions on type `T` where they are common to all types of `T`. As the `id` property is unique to the `IPrintId` interface, and the `name` property is unique to the `IPrintName` interface, we are not allowed to reference these properties when we reference `T`. The only property that these two interfaces have in common is the `print` function, and therefore only the `print` function can be used in this case.

TypeScript will ensure that we are only able to reference properties and functions on a type of `T`, where these properties and functions are common across all types that are allowed for `T`.

Generic constraints

A generic type can be constructed out of another generic type. This technique essentially uses one type to apply a constraint on another type. Let's take a look at an example, as follows:

```
function printProperty<T, K extends keyof T>
    (object: T, key: K) {
    let propertyValue = object[key];
    console.log(`object[${key}] = ${propertyValue}`);
}
```

Here, we have a function named `printProperty` that has two generic types, named `T` and `K`. The type `K` is constrained to be a value computed from the `keyof` operator on type `T`. Remember that the `keyof` operator will return a string literal type that is made up of the properties of an object, so `K` will be constrained to the property names of the type `T`.

The `printProperty` function has two parameters, named `object` of type `T`, and `key` of type `K`. The function assigns the value of the object's property named in the `key` parameter to a variable named `propertyValue`, using the syntax `object[key]`. It then logs this value to the console.

Let's test this function as follows:

```
let obj1 = {
    id: 1,
    name: "myName",
    print() { console.log(`${this.id}`) }
}

printProperty(obj1, "id");
printProperty(obj1, "name");
printProperty(obj1, "surname");
```

Here, we have constructed an object named `obj1`, which has an `id` property of type number, a `name` property of type string, and a `print` function. We are then calling the `printProperty` function three times, once with the `key` argument set to `"id"`, another with it set to `"name"`, and the third time with the `key` argument set to `"surname"`. Note that the last line of this code snippet will produce the following error:

```
error TS2345: Argument of type '"surname"' is not assignable to
parameter of type '"id" | "name" | "print"'
```

Here, we can see that the compiler has generated a string literal type based on the properties of the `obj1` object, and we are only allowed to call this function with a valid property name as the second argument. As the `obj1` object does not have a `surname` property, this last line of the code snippet is generating an error. The output of this code is as follows:

```
object[id] = 1
object[name] = myName
```

Here we can see that our `printProperty` function is indeed printing the value of the corresponding property. Let's now see what happens when we use this function to print the `print` property, as follows:

```
printProperty(obj1, "print");
```

Here, we are calling the `printProperty` function, and have used the `"print"` property as our second argument. The output of this code is as follows:

```
object[print] = function () { console.log("" + this.id); }
```

Here, we can see that we are printing out the definition of the `print` function, which corresponds to the value of `obj1["print"]`. Note that this code does not invoke the function; it simply logs the value of the property, which happens to be a function.

Generic interfaces

In the same manner that functions and classes can use generics, we are also able to create interfaces that use generic syntax. Let's take a look at an example, as follows:

```
interface IPrint {
    print(): void;
}

interface ILogInterface<T extends IPrint> {
    logToConsole(iPrintObj: T): void;
}

class LogClass<T extends IPrint>
    implements ILogInterface<T>
{
    logToConsole(iPrintObj: T): void {
        iPrintObj.print();
    }
}
```

Here, we have defined an interface named `IPrint` that has a single function named `print` that returns `void`. We then define an interface named `IlogInterface` that is using generic syntax to define a type `T` that extends the `IPrint` interface. This interface has a single function named `logToConsole`, which has a single parameter named `iPrintObj`, of type `T`. We then have a class definition for a class named `LogClass`, which is also using generic syntax to define a type `T` that extends from the `IPrint` interface. This class implements the `ILogInterface` interface, and as such, must define a `logToConsole` function.

Note that the `ILogInterface` interface requires the type of `T` to implement the `IPrint` interface, and uses the generic syntax to define this, that is, `<T extends IPrint>`. When we define the `LogClass` class, the type `T` must match the interface type definition exactly, that is, it must also be `<T extends IPrint>`.

We can now use this class definition as follows:

```
let printObject: IPrint = {
    print() { console.log(`printObject.print() called`) }
}
```

```
let logClass = new LogClass();
logClass.logToConsole(printObject);
```

Here, we have an object named `printObject` that implements the `IPrint` interface, as it has a `print` function. We then create an instance of the `LogClass` class, and call the `logToConsole` function with the `printObject` variable as the only argument. The output of this code is as follows:

```
printObject.print() called
```

Here, we can see that the `logToConsole` function of the `LogClass` instance is calling the `print` function of the `printObject` variable.

Creating new objects within generics

From time to time, generic classes may need to create an object of the type that was passed in as the generic type `T`. Consider the following code:

```
class ClassA { }

class ClassB { }

function createClassInstance<T>
    (arg1: T): T {
    return new arg1(); // error : see below
}

let classAInstance = createClassInstance(ClassA);
```

Here, we have defined two classes, named `ClassA` and `ClassB`. We then define a function named `createClassInstance` that is using generic syntax to define the type `T`. This function has a single parameter named `arg1` of type `T`. This function returns a type `T`, and is intended to create a new instance of the class that was passed in as the single `arg1` parameter.

The last line of this code snippet show how we intend to use the `createClassInstance` function. We are calling this function with the class definition of `ClassA` as the only argument.

Unfortunately, this code will generate the following error:

```
error TS2351: This expression is not constructable.
  Type 'unknown' has no construct signatures
return new arg1();
```

Here, we can see that the compiler will not allow us to construct a new instance of the type `T` in this way. This is because the type of `T` is really of type unknown to the function at this stage.

According to the TypeScript documentation, in order for a generic class to be able to construct an object of type `T`, we need to refer to type `T` by its constructor function. Our `createClassInstance` function therefore needs to be rewritten as follows:

```
function createClassInstance<T>
    (arg1: { new(): T }): T {
    return new arg1();
}
```

Here, we have modified the `arg1` parameter, and are constructing an anonymous type that defines a `new` function, and returns the type `T`, that is, `arg1: { new() : T }`. In other words, the `arg` parameter is a type that overloads the `new` function, and returns an instance of `T`. Our code will now compile and work as expected.

This concludes our initial discussion on generics. We have discussed how to use generic syntax, how to constrain the type of `T`, and what is and what is not possible within generic code. Generic syntax, however, allows us to express types as a combination of other types, using mapped types and conditional types. We will discuss these language features in the next section of this chapter.

Advanced type inference

The TypeScript language has given us a large toolbox with which to define custom types, inherit types from each other, and use generic syntax to work with any number of different types. By combining these features, we can start to describe some seriously advanced type definitions, including types based on other types, or types based on some or all of the properties of another type. We can also completely modify a type by adding and removing properties as we see fit.

In this section of the chapter, we will explore more advance type inference, including conditional types, inferred types, and mapped types, or, as the author describes it, "type mathematics." Be warned that the syntax used with advance types can quickly become rather complicated to read, but if we apply some simple rules, it is easily understandable.

Remember that although types help us to describe our code, and also help to harden our code, they do not affect the generated JavaScript. Simply describing a type is a theoretical exercise, and much of the "type mathematics" in this section of the chapter will still only do exactly that – specify a type. We will still need to put these types to use in order to realize their benefit within our code base.

Mapped types

We already know that we can use a type alias to define a special named type, as discussed in *Chapter 2*, *Exploring the Type System*. Type aliases, however, can become even more powerful when combined with generic syntax, allowing us to create types based on other types. Add in the `keyof` keyword, and we can create new types based on the properties of another type. This is best illustrated with an example, as follows:

```
interface IAbRequired {
    a: number;
    b: string;
}

let ab: IAbRequired = {
    a: 1,
    b: "test"
}

type WeakInterface<T> = {
    [K in keyof T]?: T[K];
}

let allOptional: WeakInterface<IAbRequired> = {}
```

Here, we have defined an interface named `IAbRequired` that has two properties, named `a` of type number, and `b` of type string. We then create an instance of an object named `ab`, which is of type `IAbRequired`, and as such, must define both an `a` and `b` property, as both properties are required.

We then create a type alias name `WeakInterface`, which uses generic syntax to allow it to be used with any type named `T`. This type alias also specifies a second type, `K`, that is using the `keyof` keyword on the type `T`. The effect of the `keyof` keyword is that the `WeakInterface` type will contain a property for each property that the type `T` defines. Note, however, that the definition of the properties, that is `[K in keyof T]?`, is also using the optional property operator `?`, and the type of each property has been defined as `T[K]`. In other words, return the type of the original property of type `T`, named `K`, but make it optional.

What this means is that we are defining a type named `WeakInterface`, which accepts a type named `T`, and we are transforming each property that is defined for `T` into an optional property.

The last line of this code snippet defines a variable named `allOptional`, which is of the type `WeakInterface<IAbRequired>`. This, therefore, makes all of the properties that were named on the `IAbRequired` interface optional, and our object can be constructed with no properties.

Note too that even though we are making each property in the type `IAbRequired` optional, we cannot define properties that are not available on this original type.

Partial, Readonly, Record, and Pick

Using mapped types that transform properties are seen as so fundamental that their definitions have been included in the standard TypeScript type definitions. The `WeakType` type alias that we created earlier is actually called `Partial`, which can be seen from the type definition in `lib.es5.d.ts`, as follows:

```
/**
 * Make all properties in T optional
 */
type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

Here, we can see the type definition for a type named `Partial`, which will transform each property in the type named `T` into an optional property. There is also a mapped type named `Required`, which will do the opposite of `Partial`, and mark each property as required.

Similarly, we can use the `Readonly` mapped type to mark each property as `readonly`, as follows:

```
/**
 * Make all properties in T readonly
 */
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

Here, we can see the definition of the `Readonly` mapped type that will mark each property found in the type named `T` as `readonly`. We can use this predefined type as follows:

```
let readonlyVar: Readonly<IAbRequired> =
{
    a: 1,
    b: "test"
}

readonlyVar.a = 1;
```

Here, we have used the `Readonly` mapped type with our previous interface named `IAbRequired` to create a variable named `readonlyVar`. As the `IAbRequired` interface needs both an `a` and a `b` property, we have provided values for both of these properties. Note, however, the last line of this code snippet, which is attempting to assign the value of `1` to the `a` property of our `readonlyVar` variable. This line of code will generate the following error:

```
error TS2540: Cannot assign to 'a' because it is a read-only property.
```

Here, we can see that by using the `Readonly` mapped type, all properties of our original `IAbInterface` interface have been marked as `readonly`, and we therefore cannot modify their values. Note that these mapped types are shallow, and will only apply to top-level properties.

Outside of the standard mapped types of `Partial` and `Readonly`, there are two other interesting mapped types, named `Pick` and `Record`. The `Pick` mapped type is used to construct a type based on a subset of properties of another type, as follows:

```
interface IAbc {
    a: number;
    b: string;
    c: boolean
}

type PickAb = Pick<IAbc, "a" | "b">;

let pickAbObject: PickAb = {
    a: 1,
    b: "test"
}
```

Here, we have an interface named `IAbc` that defines three properties, named `a` of type number, `b` of type string, and `c` of type boolean. We then construct a new type named `PickAb` using the `Pick` mapped type, and provide two generic types. The first generic type is our interface name `IAbc`, and the second generic type is a string literal, which is constrained to match a property name of the original type. In essence, the `Pick` mapped type will pick a set of properties from the original type that are to be applied to the new type. In this example, therefore, only the `a` and `b` properties from the `IAbc` interface will be applied to the new type.

We can see an example of the resultant type in the last line of this code snippet. Here, we are creating a variable named `pickAbObject` that is of the type `PickAb`. As the `PickAb` mapped type only uses the `"a"` and `"b"` properties in its string literal, we only need to specify an `a` and `b` property for this variable.

The final mapped type that we will explore is the `Record` mapped type, which is used to construct a type on the fly. It is almost the opposite of the `Pick` mapped type, and uses a provided list of properties as a string literal to define what properties the type must have. Consider the following example:

```
type RecordedCd = Record<"c" | "d", number>;

let recordedCdVar: RecordedCd = {
    c: 1,
    d: 1
};
```

Here, we have defined a type named `RecordedCd` that is using the `Record` mapped type, and has provided two generic arguments. The first generic argument is a string literal with the values of `"c"` and `"d"`, and the second generic argument is the type number. The `Record` mapped type will then create a new type with the properties of `c` and `d`, both of type number.

We can see an example of using this new type named `RecordedCd` in the last line of the code snippet. As the variable named `recordedCdVar` is of type `RecordedCd`, it must provide a property named `c`, and a property named `d`, both of type number.

Conditional types

In *Chapter 2*, *Exploring the Type System*, we introduced the concept of conditional expressions, with the following format:

```
(conditional) ? ( true statement )  :  ( false statement );
```

Here, we have a condition, followed by a question mark (`?`), and then either a `true` expression or a `false` expression, separated by a colon (`:`). We can use this syntax with types as well, to form what is known as a **conditional type**, as follows:

```
type NumberOrString<T> = T extends number ? number : string;
```

Here, we have defined a type named `NumberOrString` that is using the generic syntax to define a type named `T`. The type of `T` is set to the result of the conditional type statement to the right of the assignment operator (`=`).

This conditional type statement is checking whether the type of `T` extends the type number. If it does, it will return the number type, and if not, it will return the string type.

We can use this conditional type within a function as follows:

```
function logNumberOrString<T>(input: NumberOrString<T>) {
    console.log(`logNumberOrString : ${input}`);
}
```

Here, we have a function named `logNumberOrString` that is using generic syntax to define a type named `T`. This function has a single parameter that is of type `NumberOrString`, which is our conditional type. The function simply logs the value of the input parameter to the console. We can now use this function as follows:

```
logNumberOrString<number>(1);
logNumberOrString<string>("test");
logNumberOrString<boolean>(true);
```

Here, we are calling the `logNumberOrString` function three times. The first call is using the number type for the type of `T`, and the second call is using the string type for the type of `T`. The third call is using the `boolean` type for the type of `T`, and this line will generate the following error:

```
error TS2345: Argument of type 'true' is not assignable to parameter of
type 'string'.
```

Let's break down what is causing this error. Remember that our conditional type checks whether the type we used for `T` extends number. If it does, then the conditional type returns the type number. If it does not extend number, then the conditional type will return the type string. As boolean does not extend number, the returned type in this case will be the type string. We therefore need to use a string in this case, as follows:

```
logNumberOrString<boolean>("boolean does not extend number");
```

Here, we have modified our function argument to be of type string, which then matches the result of the conditional type, `NumberOrString`. When given a boolean type, it will return the type string.

Conditional type chaining

In a similar way to how conditional statements can be chained together, conditional types can also be chained together to form a logic tree that will return a specific type. Let's take a look at a slightly more complex example that uses this technique, as follows:

```
interface IA {
    a: number;
}

interface IAb {
    a: number;
    b: string;
}

interface IAbc {
    a: number;
    b: string;
    c: boolean;
}
```

Here, we have three interfaces, named `IA`, `IAb`, and `IAbc`. The `IA` interface has a single property named `a`, the `IAb` interface has two properties named `a` and `b`, and the `IAbc` interface has three properties named `a`, `b`, and `c`. Let's now create a conditional type that uses conditional type chaining as follows:

```
type abc_ab_a<T> =
    T extends IAbc ? [number, string, boolean] :
    T extends IAb ? [number, string] :
    T extends IA ? [number] :
    never;
```

Here, we have created a conditional type named `abc_ab_a`, which uses generic syntax for the type named `T`. If `T` extends the `IAbc` interface, then the conditional type will return a tuple of the type number, string, and boolean. If `T` does not extend the `IAbc` interface, then a new condition is checked. If `T` extends the `IAb` interface, the conditional type will return a tuple of type number and string. The final condition checks whether the type `T` extends the `IA` interface. If it does, it will return a tuple of type number, and if not, it will return `never`.

We can now write a function that uses this conditional type, as follows:

```
function getTupleStringAbc<T>
    (tupleValue: abc_ab_a<T>): string
{
    let [...tupleDestructured] = tupleValue;
```

```
    let returnString = "|";
    for (let value of tupleDestructured) {
        returnString += `${value}|`;
    }
    return returnString;
}
```

Here, we have a function named `getTupleStringAbc` that uses generic type syntax to define a type named `T`. The function accepts a single parameter named `tupleValue` that is the result of our conditional type named `abc_ab_a<T>`. This function destructures the `tupleValue` parameter into an array, and then generates a string by looping through all elements of the array. We can now call this function as follows:

```
let keyA = getTupleStringAbc<IA>([1]);
console.log(`keyA = ${keyA}`);

let keyAb = getTupleStringAbc<IAb>([1, "test"]);
console.log(`keyAb = ${keyAb}`);

let keyAbc = getTupleStringAbc<IAbc>([1, "test", true]);
console.log(`keyAbc = ${keyAbc}`);
```

Here, we are creating three variables, named `keyA`, `keyAb`, and `keyAbc`, by calling the `getTupleStringAbc` function with three different types, namely, `IA`, `IAb`, and `IAbc`. Note that the first call to the `getTupleStringAbc` function has a tuple with a single numeric value. This matches the output of our conditional type, as the type of `T` will be `IA`, and the conditional statement `T extends IA ? [number]` will return `true`. The results of the conditional type evaluation will drive the structure of the tuple that is needed in the function call. The results of this code are as follows:

```
keyA = |1|
keyAb = |1|test|
keyAbc = |1|test|true|
```

Here, we can see that the `getTupleStringAbc` function will return a string value that is the concatenation of the values passed in as the tuple argument.

Distributed conditional types

When defining conditional types, instead of returning only a single type as part of our conditional statements, we can also return a number of types, or distributed conditional types. As an example of this, consider the following code:

```
type dateOrNumberOrString<T> =
    T extends Date ? Date :
```

```
    T extends number ? Date | number :
    T extends string ? Date | number | string :
    never;

function compareValues
    <T extends string | number | Date | boolean>
(
    input: T,
    compareTo: dateOrNumberOrString<T>
) {
    // do comparison
}
```

Here, we have a conditional type named `dateOrNumberOrString` that is using generic syntax to define a type named `T`. If the type of `T` is a `Date`, then the conditional type will return a Date type. If the type of `T` is a number, then the conditional type will return a type of Date or number. If the type of `T` is a `string`, then the conditional type will return a Date type or a number or a string. If the type of `T` is neither a date nor a number or string, the conditional type will return never.

We then define a function named `compareValues` that also uses generic syntax to define a type named `T`. This function has two parameters, named `input`, of type `T`, and `compareTo`, which uses our conditional type named `dateOrNumberOrString`. This introduces some interesting logic when using this function as follows:

- If the `input` parameter is of type Date, then the `compareTo` parameter may only be of type Date.
- If the `input` parameter is of type number, then the `compareTo` parameter may be either a Date or a number.
- If the `input` parameter is of type string, then the `compareTo` parameter may be either a Date or a number or a string.
- If the `input` parameter is not of type Date or number or string, then do not allow this function to be called.

All of this type inference is handled purely by the distributed conditional type. As an example of using this function, each of the following function calls are valid:

```
compareValues(new Date(), new Date());
compareValues(1, new Date());
compareValues(1, 2)
compareValues("test", new Date());
```

```
compareValues("test", 1);
compareValues("test", "test");
```

Here, we can see the various combinations that can be used with our `compareValues` function. Based on the type of the first argument, the second argument can be one of a range of different types.

There are a number of situations where distributed conditional types can be used to either limit or expand what types are allowed in these cases. They can also be useful, as we have seen, when defining what types can be compared against each other, or what types can be used in a computational algorithm.

Conditional type inference

There is a further, and more esoteric version of the conditional type syntax, where we are able to infer a new type as part of a conditional type statement. The simplest form of these inferred types can best be explained by an example, as follows:

```
type inferFromPropertyType<T> =
    T extends { id: infer U } ? U : never;
```

Here, we have defined a type named `inferFromPropertyType` that is using the generic syntax to define a type named `T`. We are then using a conditional type to check whether the type of `T` extends an object that has a property named `id`. If the type of `T` is an object that has a property named `id`, then we will return the type of the `id` property itself. This is done by introducing a new type name, which in this case is `U`, and using the `infer` keyword. In other words, we are inferring a new generic type named `U` that is the type of the `id` property of the object `T`. If the object `T` does not have an `id` property, then we simply return `never`. Let's now take a look at how we would use this inferred conditional type, as follows:

```
function testInferFromPropertyType<T>
(
    arg: inferFromPropertyType<T>
) { }

testInferFromPropertyType<{ id: string }>("test");
testInferFromPropertyType<{ id: number }>(1);
```

Here, we have defined a function named `testInferFromPropertyType` that is using the generic syntax to define a type named `T`. This function has a single parameter named `arg` that is using our conditional type, `inferFromPropertyType`. Note that in order to use this function, we must specify the type of `T` when we call the function, as seen in the final two lines of the code snippet.

In the first call to our `testInferFromPropertyType` function, we are specifying that the type of `T` is an object that has a property name `id`, which is of type string. Our inferred type, therefore, takes its type from the type of the `id` property. As the type of the `id` property is of type string, the argument named `arg` must be of type string.

The second call to the `testInferFromPropertyType` function specifies that the type of `T` is an object that has a property named `id` of type number. The inferred type `U`, therefore, is of type number.

Remember that a conditional type is a computed type based on the original type that is given as an input. This means that in order to use a conditional type, we need to supply an input type, and the conditional type will be computed for us, based on the input type.

Type inference from function signatures

In the same way that we can define inferred types based on object properties, we can also infer types based on function signatures. These inferred types can be inferred from either the function arguments, or from the function return type. Let's take a look at an example of this, as follows:

```
type inferredFromFnParam<T> =
    T extends (a: infer U) => void ? U : never;
```

Here, we have a conditional type named `inferredFromFnParam`, which will infer the type of `U` from the argument named `a` of a function signature that has a single parameter, and returns `void`. If the function signature does not match what is specified by the `extends` clause, that is, it does not take a single parameter, and does not return `void`, then the inferred type will be `never`. We can use this inferred type as follows:

```
function testInferredFromFnParam<T>(
    arg: inferredFromFnParam<T>
) { }

testInferredFromFnParam<(a: number) => void>(1);
testInferredFromFnParam<(a: string) => void>("test");
```

Here, we have a function name `testInferredFromFnParam` that accepts a single argument of the type that is the result of our conditional inferred type named `inferredFromFnParam`. We then call this function twice, providing two function signatures with which to compute the conditional inferred type.

The first call to the `testInferredFromFnParam` function specifies a function signature that accepts a single argument of type number, and returns `void`. Our inferred type, therefore, is the type of the argument named `a`, which in this case is of type number. Therefore, the `testInferredFromFnParam` function takes a single parameter of type number.

The second call to the `testInferredFromFnParam` function specifies a function signature that accepts a single argument named `a` of type string, and returns `void`. Our inferred conditional type will therefore resolve to the type of the argument `a`, which in this case is of type string.

In a similar manner, we can also infer a type from the return type of a function, as seen in the following example:

```
type inferredFromFnReturnType<T> =
    T extends (a: string) => infer U ? U : never;

function testInferredFromReturnType<T>(
    arg: inferredFromFnReturnType<T>
) { }

testInferredFromReturnType<(a: string) => number>(1);
testInferredFromReturnType<(a: string) => boolean>(false);
```

Here, we have an inferred conditional type named `inferredFromFnReturnType`, that will infer the type `U` to be the return type of a function signature that takes a single argument named `a` of type string. If the function signature does not match the `extends` clause, then the inferred type will be `never`.

We have then defined a function named `testInferredFromReturnType` that defines a type `T`, and derives the type of its only parameter, named `arg`, to be the type returned from our inferred conditional type named `inferredFromFnReturnType`.

We then have two examples of calling this function, which use a function signature as the type of `T`. The first call uses a function signature that returns a type number, and therefore the `arg` argument must be of type number. The second call uses a function signature that returns a type string, and therefore the `arg` argument must be of type string.

Type inference from arrays

There is one other syntax that can be used for an inferred type, which is used when inferring a type from an array. This is best described with an example, as follows:

```
type inferredTypeFromArray<T> =
    T extends (infer U)[] ? U : never;

function testInferredFromArray<T>
    (args: inferredTypeFromArray<T>)
{ }

testInferredFromArray<string[]>("test");
testInferredFromArray<number[]>(1);
```

Here, we have an inferred conditional type named `inferredTypeFromArray` that is checking whether the type of `T` extends an array, that is, `T extends []`. Note that this `extends` clause injects an inferred type named `U` within the extends clause itself by wrapping the inferred type name in braces, that is `(infer U)`. What this means is that the type of `U` will be inferred from the type of the array itself.

The second line of this code snippet defines a function named `testInferredFromArray` that uses our inferred conditional type to narrow the type of `T` to the result of the conditional type. We then call this function with a string array as the type of `T`. This means that the inferred conditional type will resolve to string, and the type of the `args` parameter is therefore a string.

Finally, we call this function with a `number` array as the type of `T`, and, therefore, the `args` argument must be of type number as well.

Standard conditional types

There are some handy conditional type combinations that have been included as part of the standard TypeScript library, in much the same way as we saw with the mapped types of `Partial` and `Readonly`. Let's explore three of these conditional types, named `Exclude`, `Extract`, and `NonNullable`, as follows:

```
type ExcludeStringAndNumber = Exclude<
    string | number | boolean,
    string | number>;

let boolValue: ExcludeStringAndNumber = true;
```

Here, we have defined a type named `ExcludeStringAndNumber`, which is using the standard conditional type named `Exclude`. The `Exclude` conditional type takes two generic parameters. It will exclude those types given in the second generic parameter from the types given in the first generic parameter. In this example, we have specified that we wish to exclude the types of number and string from the list of types `number | string | boolean`. Logically, this only leaves boolean as a valid type, as seen on the last line of this code snippet. The variable `boolValue`, of type `ExcludeStringAndNumber`, only allows assignment of a boolean value.

In a similar manner, the `Extract` standard conditional type will extract a set of types from another set of types, as can be seen in the following example:

```
type StringOrNumber = Extract<
    string | boolean | never,
    string | number>;

let stringValue: StringOrNumber = "test";
```

Here, we have defined a type named `StringOrNumber` that is using the standard conditional type named `Extract`, which also takes two types as its generic parameters. The `Extract` conditionals type will return all matching types given in the second generic parameter from the list given in the first parameter. In our preceding example, we are extracting either a string or a number type from the list of `string | boolean | never`. Logically, the only matching type in this case is string, as seen by the usage of this type on the last line of the code snippet, where the `stringValue` variable, of type `StringOrNumber`, can only be assigned a string value.

Another standard conditional type will exclude null and undefined from a type union. This conditional type is named `NonNullable`, as follows:

```
type NotNullOrUndef = NonNullable<number | undefined | null>;
let numValue: NotNullOrUndef = 1;
```

Here, we have defined a type named `NotNullOrUndef`, that is using the conditional type named `NonNullable` to extract the types from a given type union that are not `null` or `undefined`. Removing null and undefined from the given type union, which was `number | undefined | null`, only leaves type number. Our `NotNullOrUndef` type, therefore, will resolve to a type of number, as can be seen in the usage of this type on the last line of the code snippet.

Summary

In this chapter, we have explored the concepts of generics, including how TypeScript defines a specific syntax for generics, and how we can constrain generic types in certain circumstances. We then discussed how we can use generics with interfaces, and how to create new objects within generic code. The second section of this chapter explored advance type inference, starting with mapped types, and then moving on to conditional types. We discussed distributed conditional types, conditional type inference, and finally took a look at standard conditional types that are available to use with a standard TypeScript installation.

In the next chapter, we will explore asynchronous language features, and how we can use specific TypeScript language constructs to help with the asynchronous nature of JavaScript programming.

5

Asynchronous Language Features

The JavaScript runtime, whether it is running in the browser, or whether it is running on a server through Node, is single threaded. This means that one, and only one, piece of code will be running at a particular time. This code runs in what is known as the main thread. JavaScript has also been built around an asynchronous approach, meaning that the main thread will not pause when requested to load a resource of some sort. It will, instead, place this request onto an internal queue, which will eventually be processed at a later point in time. While the single-threadedness of JavaScript may take a while to get your head around, it does take away the need for in-memory locking mechanisms, as are used in other languages to handle multiple threads of execution. This makes the JavaScript runtime a little easier to understand, and to work with.

The traditional JavaScript mechanism for dealing with asynchronous requests is through the callback mechanism. This means that we provide a function, known as a callback function, to an asynchronous request, and this function will be executed once the asynchronous request has been processed.

There have been a number of techniques introduced into the JavaScript language to help with writing asynchronous code. Each of these techniques has built upon the traditional callback mechanism that is typically used in JavaScript. One of the most popular techniques is known as Promises, which provides a simplified syntax for writing asynchronous code. The Promise mechanism also allows us to chain multiple asynchronous calls one after another, and this technique is known as fluent syntax. Another technique is known as async and await, where we mark certain functions as asynchronous, and can then use the `await` keyword to pause the execution flow of our code until the asynchronous function returns.

In this chapter, we will explore the following concepts:

- Traditional callbacks, what they are, and how they are used
- Creating Promises
- Returning values from Promises
- Async and await syntax
- Trapping errors in async await code

Callbacks

Let's start by examining callbacks, which is the standard mechanism for registering a function to execute after an asynchronous event has occurred. Consider the following example:

```
function delayedResponseWithCallback(callback: () => void) {
    function executeAfterTimeout() {
        console.log(`5. executeAfterTimeout()`);
        callback();
    }
    console.log(`2. calling setTimeout`)
    setTimeout(executeAfterTimeout, 1000);
    console.log(`3. after calling setTimeout`)
}
```

Here, we have a function named `delayedResponseWithCallback` that has a single parameter named `callback`, which is a function with no arguments that returns void. Within this function, we define another function named `executeAfterTimeout`, which will log a message to the console, and then execute the callback function that was passed in as the parameter named `callback`. Note that each console log in this snippet starts with a number, which shows the order of execution of the statements. We will see logs a little later with the 1, 4, and 6 ordering.

We then log a message to the console to indicate that we are about to call the `setTimeout` function. After this, we call the `setTimeout` JavaScript function, passing in our `executeAfterTimeout` function as an argument, as well as the amount of time to delay this call. This will add a delay of one second, or 1,000 ms, after which our `executeAfterTimeout` function will be triggered. We then log another message to the console to indicate that the call to the `setTimeout` function has completed.

Note that within this code, we actually have two callback functions. The first is the parameter named `callback`, and the second is the function named `executeAfterTimeout`. We are passing the `executeAfterTimeout` function as a callback function into the `setTimeout` function.

We can now use this callback function as follows:

```
function callDelayedAndWait() {
    function afterWait() {
        console.log(`6. afterWait()`);
    }
    console.log(`1. calling delayedResponseWithCallback`);
    delayedResponseWithCallback(afterWait);
    console.log(`4. after calling delayedResponseWithCallback`)
}

callDelayedAndWait();
```

Here, we have defined a function named `callDelayedAndWait`. Within this function, we have defined another callback function named `afterWait`, which simply logs a message to the console.

We then log a message to the console to indicate that we are about to call the `delayedResponseWithCallback` function. After logging the message, we invoke the `delayedResponseWithCallback` function, and pass in the `afterWait` function as the only argument. Finally, we log a message to the console, indicating that we have called the `delayedResponseWithCallback` function.

The last line of this code snippet actually calls the `callDelayedAndWait` function to start our little program. The output of this code is as follows:

```
1. calling delayedResponseWithCallback
2. calling setTimeout
3. after calling setTimeout
4. after calling delayedResponseWithCallback
... ( 1 second pause ) ...
5. executeAfterTimeout()
6. afterWait()
```

Here, we can see that the first message logged to the console was logged just before we called the `delayedResponseWithCallback` function. The second and third messages were logged from within the `delayedResponseWithCallback` function, just before and just after we called the `setTimeout` function. The fourth message logged to the console was after the call to the `delayedResponseWithCallback` function returned.

If you execute this code, you will notice that the first four messages are logged to the console immediately. Then there is a one second pause, before the fifth and sixth messages are logged to the console.

If we examine this sequence of events a little more closely, we will see that although we are using two callback functions, namely `afterWait` and `executeAfterTimeout`, there is only one asynchronous call in this example. This asynchronous call is the call to the `setTimeout` function, which causes a one second delay before invoking the callback function provided. This means that the `executeAfterTimeout` function is being placed on a queue, and the JavaScript runtime is invoking this callback function at a later point in time. Once it has been called, the `executeAfterTimeout` function then invokes the `afterWait` function, which was passed in as a callback function.

This sequence of events clearly shows the nature of asynchronous processing in the JavaScript runtime. The runtime will process and execute each line of code that it encounters. If it finds an asynchronous code block, it will place this code block on a queue for later processing. Once this code block is in the queue, the runtime will continue processing our code base, and repeat the process if it encounters any other asynchronous calls. The numbering of the console logs in this example illustrates the execution order of the code.

Promises

The asynchronous nature of JavaScript does take some time to get used to. Any time we need to wait for a resource, or wait for user input, we need to implement a callback mechanism to handle this correctly. Unfortunately, as a code base grows, we find that we need to rely on callbacks more and more. This can easily lead to what is known as callback hell, where we have so many callbacks that are nested in other callbacks that the code becomes increasingly difficult to read and maintain.

As an example of this, let's consider some code that must read three files one after the other, and print their contents, as follows:

```
import * as fs from "fs";

fs.readFile("./test1.txt", (err, data) => {
    if (err) {
        console.log(`an error occurred : ${err}`);
    } else {
        console.log(`test1.txt contents : ${data}`);
        fs.readFile("./test2.txt", (err, data) => {
            if (err) {
                console.log(`an error occurred : ${err}`);
            } else {
                console.log(`test2.txt contents : ${data}`);
                fs.readFile("./test3.txt", (err, data) => {
                    if (err) {
```

```
                        console.log(`an error occurred : ${err}`);
                    } else {
                        console.log(`test3.txt contents
                            : ${data}`);
                    }
                })
            }
        })
    }
});
```

Here, we are importing the Node filesystem library named `fs`, using our standard module syntax. We then call the `readFile` function of the `fs` library in order to read a file from disk. This `readFile` function takes two parameters. The first parameter is the filename itself, which is `"./test1.txt"`, and the second parameter is a callback function that should be invoked once the file has been read from disk. This callback function has two parameters, named `err` and `data`. Within this callback function, we are checking whether the `readFile` function returned an error, by checking the `err` argument. If it did not return an error, we can then proceed to log the contents of the file to the console, and read the second file named `"./test2.txt"`. This second call to the `readFile` function must again provide a callback function, and within this callback function check whether an error occurred. The pattern is repeated for the third file.

As can be seen from this code snippet, by doing things one after the other using callbacks, our code becomes a series of nested functions within nested functions, and each nested function is repeating the same set of steps. Each nested function, for example, must check the error condition within the callback before proceeding.

The nested nature of asynchronous calls each needing to provide a callback function of the same type, and that does the same sort of thing, is what contributes to callback hell.

To make asynchronous code a lot simpler, and to eliminate callback hell, many different JavaScript libraries implemented similar design patterns to make the syntax of callbacks easier to work with. Eventually, these design patterns converged into a proposal for the JavaScript language, named Promises. Let's take a look at the same code, but using Promises as follows:

```
fs.promises.readFile("./test1.txt")
    .then((value) => {
        console.log(`ps test1.txt read : ${value}`);
        return fs.promises.readFile("./test2.txt");
    }).then((value) => {
        console.log(`ps test2.txt read : ${value}`);
```

```
        return fs.promises.readFile("./test3.txt");
    }).then((value) => {
        console.log(`ps test3.txt read : ${value}`);
    })
    .catch((error) => {
        console.log(`an error occurred : ${error}`);
    });
```

Here, we are using the `promises` namespace of the `fs` library to call a `readFile` function, which is a Promise-based version of the same `readFile` function call we used earlier. This Promise-based version of the `readFile` function takes a single argument, which is the name of the file. We can then call a `then` function, which is made available to all functions that return Promises, such as the `readFile` function. This `then` function will be called when the asynchronous function returns, and it provides a single parameter that will hold the results of the asynchronous call. The `readFile` function will, therefore, make the contents of the file read from disk available through this parameter, which is named `value` in our first `then` function. Within this `then` function, which we know will be called after the file is read from disk, we log a message to the console, which contains the contents of the file. We then return the results of another call to the `readFile` function with the name of the next file to read. We repeat this pattern for the third file as well.

This technique of returning a Promise within a `then` function allows us to chain a series of `then` functions one after the other. This technique is known as fluent syntax, and it simplifies our code immensely. There are no nested functions within nested functions; we just have a series of `then` functions following one another.

Note that after all of the `then` functions, we can also add a `catch` function to the call chain. This single `catch` function will handle any errors that may occur if we encounter an error reading any one of the three files. Having a single `catch` function further simplifies our code, and we do not have to repeatedly check for errors after each file is read. Also, if an error occurs when reading the first file, the `catch` block will be executed immediately, and no other `then` function will be executed.

This Promise-based syntax is a far cleaner and simpler way to write a series of asynchronous calls that need to be executed one after another. It also ensures that any error will terminate execution of the Promise call chain, and our code execution path will fall through to our `catch` clause.

A Promise-based asynchronous call is also referred to as being thenable, meaning that we can attach a `then` function to the original function call.

Promise syntax

The code examples that we have seen up until now have used Promises that are provided by a third-party library, such as the Promise-based versions of filesystem access available through the Node `fs.promise` namespace. So how do we write our own Promises?

A Promise is an instance of a new Promise class whose constructor requires a function signature that accepts two callback functions, generally named `resolve` and `reject`. Consider the following function definition:

```
function fnDelayedPromise(
    resolve: () => void,
    reject: () => void) {
    function afterTimeout() {
        resolve();
    }
    setTimeout(afterTimeout, 1000);
}
```

Here, we have a function named `fnDelayedPromise` that accepts two functions as parameters, named `resolve` and `reject`, both returning `void`. Within the body of this function, we define a callback function named `afterTimeout`, which will invoke the `resolve` callback that was passed in as the first argument. It then calls the `setTimeout` function, which will cause a one second delay before executing the `afterTimeout` function.

We can now use this function to construct a Promise object, as follows:

```
function delayedResponsePromise(): Promise<void> {
    return new Promise<void>(fnDelayedPromise);
}
```

Here, we have defined a function named `delayedResponsePromise` that returns a Promise object, and is using generic syntax to indicate that the type returned from the Promise object is of type void. Within this function, we are simply creating a new instance of a Promise object, and passing in the function named `fnDelayedPromise` as the single constructor argument.

Note that in general practice, these two function definitions are combined into a single code block. The purpose of the previous two snippets has been to highlight two important concepts when creating Promises. Firstly, to use a Promise, you must return a new Promise object. Secondly, a Promise object is constructed with a function that takes two callback arguments, generally named `resolve` and `reject`.

Let's take a look at how these two steps are combined in general practice, as follows:

```
function delayedPromise(): Promise<void> {
    // return new Promise object
    return new Promise<void>
        ( // start constructor
            (
                resolve: () => void, // resolve function
                reject: () => void   // reject function
            ) => {
                // start of function definition
                function afterTimeout() {
                    resolve();
                }
                setTimeout(afterTimeout, 1000);
                // end of function definition
            }
        ); // end constructor
}
```

Here, we have a function named `delayedPromise` that is returning a Promise of type void. Within this function, we simply construct and then return a new Promise object. The constructor function for the Promise object accepts a single argument, which is a function that has two arguments, named `resolve` and `reject`. The `resolve` and `reject` arguments are also functions that return `void`. Within the function definition, we set up an `afterTimeout` callback function, and then call the `setTimeout` function to pause for one second before executing the `afterTimeout` function. When the `afterTimeout` function is invoked, it will invoke the `resolve` callback function.

Note that this code snippet includes comments that show where the constructor for the new Promise object starts and finishes, as well as where the actual function definition starts and finishes.

We can now use this `delayedPromise` function as follows:

```
delayedPromise().then(() => {
    console.log(`delayed promise returned`);
});
```

Here, we are calling the `delayedPromise` function and attaching a `then` function, which will be executed once the Promise returns.

Promise errors

A Promise object is constructed with a function that has two callback functions, generally named `resolve` and `reject`. Thus far, we have only used the `resolve` callback to indicate that the Promise processed successfully. We can use the `reject` callback in the case where we want to report an error, as follows:

```
function errorPromise(): Promise<void> {
    return new Promise<void>(
        (   // constructor
            resolve: () => void,
            reject: () => void
        ) => {
            // function definition
            console.log(`2. calling reject()`);
            reject();
        }
    )
}

console.log(`1. calling errorPromise()`);
errorPromise().then(() => { })
    .catch(() => { console.log(`3. caught an error`) });
```

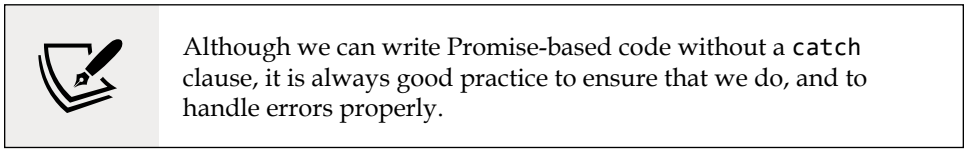
Here, we have defined a function named `errorPromise` that returns a Promise object of type void. Within this function, we construct and immediately return a new Promise object. Again, we construct this Promise object with two callback functions named `resolve` and `reject`. Within the function definition, we log a message to the console, and call the `reject` callback function.

The last two lines of this code snippet log a message to the console, and then call the `errorPromise` function, with our standard `then` and `catch` functions. Within the `catch` function, we are logging an error to the console. The output of this code is as follows:

```
1. calling errorPromise()
2. calling reject()
3. caught an error
```

Here, we can clearly see the sequence of events that our code is executing from the logged output. Our first console log occurs just before we call the `errorPromise` function. The second log output occurs within our Promise itself, just before we call the `reject` callback on the Promise. The third log output message occurs within our `catch` block.

What we can see with this sample is that calling the `reject` callback function within a Promise will trigger the `catch` function.



Returning values from Promises

Most Promises will, in fact, return some sort of information when called, and we use the generic syntax when we construct a Promise, in order to indicate what type it will return. Thus far, we have been using `Promise<void>` to indicate that the Promise will return a type `void`, or in other words, will not return any data. We can use `Promise<string>` to indicate that the Promise will return a string, as follows:

```
function promiseReturningString(throwError: boolean)
    : Promise<string>
{
    return new Promise<string>(
        (
            resolve: (outputValue: string) => void,
            reject: (errorCode: number) => void
        ) => {
            if (throwError) {
                reject(101);
            }
            resolve(`resolve with message`);
        }
    )
}
```

Here, we have a function named `promiseReturningString` that has a single input argument named `throwError`, which is of type boolean. This function returns a Promise that is using generic syntax to indicate that it will return a Promise of type string. The body of the function constructs and returns a new Promise object. Within the body of the Promise, we are using the `throwError` input parameter to either call the `reject` or `resolve` callback function.

Note, however, the format of the `resolve` callback function. It is returning a type of `void`, but has a single input parameter named `outputValue` of type string. This is the method of returning values from Promises.

Remember that within our Promise code, when we invoke the `resolve` callback function, we provide a single argument of the type that was specified as the Promise generic type. This can be seen when we invoke the `resolve` function, which has now become `resolve(`resolve with message`)`. So, if we define a `Promise<string>`, then the `resolve` function signature must have a single parameter of type string.

Note that it is only the `resolve` callback signature that must conform to this rule. The `reject` callback signature can return something other than a string. In this code sample, it is returning a number.

Let's take a look at how we can use this Promise, as follows:

```
console.log(`1. calling promiseReturningString`)

promiseReturningString(false)
    .then((returnValue: string) => {
        console.log(`2. returnedValue : ${returnValue}`);
    }).catch((errorCode: number) => {
        console.log(`this is not called`);
    });
```

Here, we start by logging a message to the console, and then call the `promiseReturningString` function with the argument `false`, indicating that we do not want the Promise to be rejected. Note that our `then` function now has a single parameter named `returnValue`, which is of type string. This function signature matches the `resolve` function signature in our Promise, in that it has a single parameter of type string. When this function is invoked, we log a message to the console indicating what the value of the `returnValue` argument was.

So, what we can see by this example is that in order to return values from a Promise, we modify the `resolve` callback function to have a single parameter of the type specified by the Promise itself. When we are consuming this Promise, we modify the `then` function definition to also have a single parameter of the type specified by the Promise. The output of this code is as follows:

```
1. calling promiseReturningString
2. returnedValue : resolve with message
```

Here, we can see the two messages that we logged to the console. The second message is the interesting one. This message is constructed with the output of the Promise, which is made available in the `returnValue` parameter of the `then` function. Remember that our Promise invokes the `resolve` function as follows:

```
resolve(`resolve with message`);
```

Which, in turn, matches our `then` function arguments, which were:

```
.then((returnValue: string) => {
        console.log(`2. returnedValue : ${returnValue}`);
    })
```

Here, we can see that the `then` function is invoked with the argument `` `resolve with message` ``, and therefore the output of this `then` function is:

```
2. returnedValue : resolve with message
```

Here, we have successfully read and used the return value of a Promise.

Note that we can also force an error from this Promise by switching the `throwError` argument from `false` to `true`, as follows:

```
promiseReturningString(true)
    .then((returnValue: string) => {
        console.log(`this is not called`);
    })
    .catch((errorCode: number) => {
        console.log(`2. caught : ${errorCode}`);
    });
```

Here, we are forcing the Promise to call the `reject` callback, instead of the `resolve` callback. In this instance, the `then` function is not called, but the `catch` function is. Note too that we have specified that the `catch` function signature has a single parameter of type number, named `errorCode`. In the same manner as we return values from resolved Promises, we can return values from rejected Promises. These returned values do not need to be of the same type.

Promise return types

There are a few things to note about the syntax that we use when writing Promises, or a few Promise rules, as follows:

- A Promise is an object that requires a function to be passed in as part of its constructor. As we have seen, this function can be a named function, or an anonymous function, but it is more common to see the anonymous function syntax used.
- The function that is passed into a Promise has a very specific signature. It requires two parameters, generally named `resolve` and `reject`, that are themselves callback functions. The `resolve` function will be invoked as a callback if the Promise itself succeeds, and the `reject` function will be invoked as a callback if the Promise fails.

- There can be one, and only one, parameter for the `resolve` callback function. This parameter is of the type that we used in the Promise generic syntax.
- There can be one, and only one, parameter for the `reject` callback function. This parameter does not need to be the same type as the type used in the `resolve` callback function.
- These `resolve` and `reject` callback functions must return void.

Note that even with these strict but simple rules of Promise syntax, they are still incredibly powerful, and can be used for any sort of asynchronous function. Let's take a look at an example of this. Suppose we wanted to create a Promise that connects to a database, and returns some data. We might start with a few interfaces as follows:

```
interface IConnection {
    server: string;
    port: number;
}

interface IError {
    code: number;
    message: string;
}

interface IDataRow {
    id: number;
    name: string;
    surname: string;
}
```

Here, we have defined three interfaces. The first is named `IConnection`, and represents some information related to creating a database connection, such as the name of the server and the port it is listening on. The second interface, named `IError`, represents information that will be returned in the case of an error. The third interface, named `IDataRow`, represents a single row of data being returned.

We can now write a Promise using these types as follows:

```
function complexPromise
(
    connection: IConnection,
    accessKey: string
)
    : Promise<IDataRow[]>
{
    return new Promise<IDataRow[]>(
```

```
        (
            resolve: (results: IDataRow[]) => void,
            reject: (results: IError) => void
        ) => {
            // check the connection properties
            // connect to the database
            // retrieve data, or
            // reject with an error
        }

    );
}
```

Here, we have defined a function named `complexPromise` that has two parameters, named `connection`, of type `IConnection`, and `accessKey`, which is of type string. This means that when we call this function, we will need to provide values for each of the properties of the `IConnection` interface, as well as providing the `accessKey`. This Promise is using generic syntax to indicate that it will return a value of type `IDataRow[]`.

The internal workings of this Promise are not really important to show here, but the Promise would typically check that the connection argument was filled in correctly, attempt to connect to the database, retrieve some data, and raise an error if anything went wrong.

What is important to notice, however, is that even though we are using the rather simple and strict Promise structure, we are able to handle any sort of data structure, either being input to the function returning a Promise, or being output from the Promise. Let's see how we would use this Promise, as follows:

```
complexPromise(
    {
        server: "test",
        port: 4200
    },
    "abcd"
).then((rows: IDataRow[]) => {
    // do something with rows
})
.catch((error: IError) => {
    // do something with error
});
```

Here, we are calling the `complexPromise` function, and providing an object that matches the `IConnection` interface, as well as a string that will be used as an `accessKey`. We have then attached our `then` statement as well as our `catch` statement.

The `then` function would typically do something with the `rows` property that was returned, and the `catch` function would typically interrogate the `error` property, and do something with it.

This concludes our exploration of Promises, what they are, and how they can be written. As we have seen, Promises follow a simple but strict syntax, and will only ever return a successful result, or a failure condition. This structure allows us to use Promises in a uniform manner, and we can use fluent syntax to chain multiple Promises together to execute them in sequence, if we so desire.

Using Promises gives us a tool to write code that is easier to follow, and has a cleaner syntax. There is also, however, another benefit to using Promises, in that we can actually pause execution of our code until a Promise completes, using the async and await syntax, which we will explore next.

Async and await

We have seen that the JavaScript runtime is single threaded and will push any asynchronous calls onto a particular queue within its memory, to be executed later. Using Promises helps us to structure our code to ensure that we only execute a particular section of code once the asynchronous call has completed. We still need to bear in mind, however, that the JavaScript runtime will continue to process our code line by line. This quirk of the language can often lead to weird results, or unwanted errors, if we do not take care when writing code that will be executed asynchronously.

Oftentimes, however, we need to make a series of calls to one asynchronous function after another. In these cases, it would actually be far better if we could pause the execution of our code until the asynchronous code completes. This is what the `async` and `await` keywords can do for us. In this section of the chapter, we will explore how to mark functions with the `async` keyword, in order to allow the use of the `await` keyword, which will actually pause execution of the code block until the Promise has completed.

The async and await language features of JavaScript were adopted in the ES2017 language specification, meaning that only JavaScript runtimes that support this version can use the new async and await keywords. TypeScript, however, has incorporated support for these features for ES target versions all the way back to ES3. This means that we can safely use async and await within any body of TypeScript code, and it will behave as if the runtime was running ES2017. We can start using these language features now, and TypeScript will take care of the rest.

Await syntax

Let's dive right in and show how to use `async` and `await` using an example. First up, a Promise that will delay for a second as follows:

```
export function delayedPromise(): Promise<void> {
    return new Promise<void>(
        (
            resolve: () => void,
            reject: () => void) => {
            setTimeout(() => {
                console.log(`2. calling resolve()`)
                resolve();
            }, 1000);
        }
    )
}
```

Here, we have a function named `delayedPromise` that returns a Promise of type void. Within this Promise, we are calling the `setTimeout` function, which will call the `resolve` function following a delay of one second. Nothing new here.

Let's now examine how we can call this function using the `async` and `await` keywords, as follows:

```
async function callDelayedPromise() {
    console.log(`1. before calling delayedPromise`);
    await delayedPromise();
    console.log(`3. after calling delayedPromise`)
}

callDelayedPromise();
```

Here, we have a function called `callDelayedPromise` that is prefixed with the `async` keyword. The `async` keyword marks the entire function as being asynchronous. In other words, the compiler knows that the code inside this function will be executed asynchronously. Functions that are marked as `async` always return a Promise.

The body of this function logs a message to the console, and then calls the `delayedPromise` function, but here, we have prefixed the function call with the `await` keyword. The `await` keyword will pause the execution of the function code block until the Promise actually resolves. Once it resolves, we log another message to the console. Running this code will produce the following output:

```
1. before calling delayedPromise
... one second pause ...
2. calling resolve()
3. after calling delayedPromise
```

Here, we can see the sequence of events when using `async` and `await`. Our code will log the first message to the console, and then pause for a second until the Promise is resolved. Once the Promise resolves, or completes, we log the third message to the console.

The async and await technique really helps us when writing code that is sequential in nature. If we have many steps in our code that must be executed one after the other, then this technique is invaluable in producing easy-to-read and easy-to-maintain code.

Await errors

When using Promises, we should always include a `catch` block to trap any errors that may have occurred. When using `async` and `await` syntax, it is best practice to wrap any `await` calls within a `try catch` block to trap errors in the same way. Some JavaScript runtimes will allow us to get away with unhandled Promise rejections, and some will not. Node, for example, will terminate the running process if a Promise is rejected, and is not handled within a `catch` block. Other frameworks, such as Angular, will fail silently, which can cause unwanted side effects.

Let's take a look at this technique, starting with a Promise that will always return an error, as follows:

```
function errorPromise(): Promise<string> {
    return new Promise<string>(
        (
            resolve: (result: string) => void,
            reject: (error: string) => void) => {
            setTimeout(() => {
                console.log(`2. calling reject()`)
                reject("promise rejected");
            }, 1000);
        }
    );
}
```

Here, we have a function named `errorPromise` that returns a Promise object, similar to what we have seen before. This Promise, however, will call the `reject` callback after a second, in order to simulate an error. Note that we are using a Promise of type string here, and that the Promise code is returning an error message of `"promise rejected"`, when it invokes the `reject` callback. We can now use a `try catch` block to trap this error, as follows:

```
async function callErrorPromise() {
    try {
        console.log(`1. calling errorPromise()`);
        await errorPromise();

    } catch (error) {
        console.log(`3. await threw : ${error}`);
    }
}

callErrorPromise();
```

Here, we have a function named `callErrorPromise` that is marked as `async`, and is using a `try catch` block around the call to the `errorPromise` function. Within our `try` block, we log a message to the console, and then call the `errorPromise` function. Within our `catch` block, we log a message to the console, using the `error` argument that is made available. The output of this code is as follows:

```
1. calling errorPromise()
... one second pause ...
2. calling reject()
3. await threw : promise rejected
```

Here, we can see that the `errorPromise` function is being called, and then there is a one second pause before the Promise will invoke the `reject` callback. This `reject` callback will be caught by the `catch` block, and the `error` argument will contain the string `"promise rejected"`, in a similar manner to the `catch` clause we used when using Promise fluent syntax.

Await values

The async await syntax also allows values to be returned by Promises, in a similar manner to how we would use the `then` function blocks in Promise fluent syntax. Let's take a look at this in action, starting with a Promise that returns some values, as follows:

```
function promiseWithValues(): Promise<string[]> {
    return new Promise<string[]>(
        (
            resolve: (values: string[]) => void,
            reject: (error: string) => void
        ) => {
            resolve(["first", "second"]);
        }
    );
}
```

Here, we have a Promise named `promiseWithValues` that will return an array of strings when the Promise is resolved. The returned array of strings contains two elements, `"first"` and `"second"`. Let's now see how these results can be used with the `async await` syntax, as follows:

```
async function getValuesFromPromise() {
    let values = await promiseWithValues();
    for (let value of values) {
        console.log(`value : ${value}`)
    }
}

getValuesFromPromise();
```

Here we have an `async` function named `getValuesFromPromise`. Within this function, we simply create a variable named `values`, and assign to it the awaited result of the call to the `promiseWithValues` function. Once the Promise is resolved, we then use a `for` loop to loop through the values returned, and log a message to the console. The output of this code is as follows:

```
value : first
value : second
```

Here, we can see the results of logging a message for each value returned by the `promiseWithValues` Promise.

This concludes our exploration of the async await syntax. As we have seen, working with async await-style code provides a much simpler and more readable way of writing asynchronous code, particularly if we need to execute multiple asynchronous calls one after the other. The `await` keyword will pause execution of our code block until the Promise has returned. If we need to use the `await` keyword, then we must mark the function that it is used in with the `async` keyword.

If the Promise returns values, we can access these values through a simple variable assignment statement. We should also wrap any calls when using the async await syntax within a `try catch` block, to ensure that we can handle any errors that may occur.

Callbacks versus Promises versus async

As a refresher on the techniques that we have explored in this chapter, let's compare the techniques used when using callbacks, Promises, and async await all in one go. First up, the callback syntax is as follows:

```
function usingCallbacks() {
    function afterCallbackSuccess() {
        // execute when the callback succeeds
    }
    function afterCallbackFailure() {
        // execute when the callback fails
    }
    // call a function and provide both callbacks
    invokeAsync(afterCallbackSuccess, afterCallbackFailure);
    // code here does not wait for callback to execute
}
```

Here, we have defined a function named `usingCallbacks`. Within this function, we have defined two more functions, named `afterCallbackSuccess` and `afterCallbackFailure`. We then call an asynchronous function named `invokeAsync` and pass in these two functions as arguments. One of these functions will be invoked by the asynchronous code, depending on whether the call was successful.

Note that any code that appears after the call to `invokeAsync` will be executed immediately.

Our Promise syntax is as follows:

```
function usingPromises() {
    delayedPromise().then(
        () => {
            // execute on success
        }
    ).catch(
        () => {
            // execute on error
        }
    )
    // code here does not wait for promise to return
}
```

Here, we have a function named `usingPromises` that makes a call to the `delayedPromise` function, and is using fluent syntax. This fluent syntax allows us to attach a `then` function that will be executed if successful, as well as a `catch` function that will be executed if there is an error.

Note that any code that is outside the call to the `delayedPromise` function will not wait for the Promise to return and will be executed immediately.

Finally, our async await syntax is as follows:

```
async function usingAsync() {
    try {
        await delayedPromise();
        // continue to next line of code on error
    } catch(error) {
        // execute on error
    }
    // code here waits for async call to complete
}
```

Here, we have a function named `usingAsync` that has been marked with the `async` keyword. This function uses a `try catch` block to surround a call to the `delayedPromise` function. The call to this function has been prefixed by the `await` keyword. The effect of using the `await` keyword means that any code after the await call will be paused until the Promise is resolved.

Note that in an `async` function, code that is outside of the `try catch` block will not be executed until the Promise returns.

Summary

In this chapter, we focused on the asynchronous nature of the JavaScript runtime, and what techniques we can use to work with this feature. We started with callbacks, which is a standard way of handling asynchronous calls within JavaScript. We also explored the concept of callback hell, which is where using callbacks can become a bit of a nightmare. We then explored Promises, showing how the simple but strict syntax can be applied to any type of asynchronous processing. Finally, we examined the new async await syntax that can be used to pause execution of a code block until a Promise has completed.

In the next chapter, we will take a look at decorators, and how we can inject functionality into existing code using a simple decorator convention.

6

Decorators

Decorators in TypeScript provide a way of programmatically tapping into the process of defining a class. Remember that a class definition describes the shape of a class, what properties it has, and what methods it defines. When an instance of a class is created, these properties and methods become available on the class instance. Decorators, however, allow us to inject code into the actual definition of a class, before a class instance has been created. They are similar to attributes in C#, or annotations in Java.

JavaScript decorators are currently only at a draft or stage 2 level, meaning that it may take a while before they are adopted into the JavaScript standard. TypeScript, however, has supported decorators for quite some time, although they are marked as experimental. Decorators have also become popular due to their use within frameworks such as Angular, where they are primarily used for dependency injection, or Vue, where they are used to inject functions into a class definition.

It must be said that this chapter comes with a warning up-front. Since decorators are only at draft level, and with TypeScript only supporting them as experimental, the implementation and standards for both JavaScript and TypeScript could change at any time. This means that we could craft some very fine decorator code, but changes to the specifications could introduce breaking changes, forcing a significant amount of rework. The purpose of this chapter is therefore to introduce and understand decorators, particularly for those who are using them heavily in frameworks.

This chapter is broken up into two sections. The first section describes how we can set up our TypeScript project to support decorators, and what the syntax is for using decorators. The second section of this chapter will focus on each of the decorator types, what they are, how they are defined, and how they can be used. We will look at class, property, function, and method decorators.

Decorator overview

In this section of the chapter, we will take a look at the general setup and syntax of decorators, what we need to do to enable them, and how they are applied to classes. We will also show how multiple decorators can be used at the same time, and then discuss the different types of decorators. Finally, this section will take a look at decorator factories, and how we can pass parameters into decorator functions.

Decorator setup

Decorators are an experimental feature of the TypeScript compiler and are supported in ES5 and above. In order to use decorators, we need to enable a compile option in the `tsconfig.json` file. This option is named `experimentalDecorators`, and needs to be set to `true`, as follows:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "strict": true,
        "experimentalDecorators": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    }
}
```

Here, we have set the compiler option named `experimentalDecorators` to `true`. This will allow the use of decorators within our TypeScript code.

Decorator syntax

A decorator is a function that is called with a specific set of parameters. These parameters are automatically populated by the JavaScript runtime, and contain information about the class, method, or property to which the decorator has been applied. The number of parameters, and their types, determine where a decorator can be applied. To illustrate this syntax, let's define a class decorator, as follows:

```
function simpleDecorator(constructor: Function) {
    console.log('simpleDecorator called');
}
```

Here, we have a function named `simpleDecorator`, which has a single parameter named `constructor` of type Function, which logs a message to the console, indicating that it has been invoked. This function, due to the parameters that it defines, can be used as a class decorator function and can be applied to a class definition, as follows:

```
@simpleDecorator
class ClassWithSimpleDecorator {

}
```

Here, we have a class named `ClassWithSimpleDecorator` that has the `simpleDecorator` decorator applied to it. We apply a decorator using the "at" symbol (`@`), followed by the name of the decorator function. Running this code will produce the following output:

```
simpleDecorator called
```

Here, we can see that the `simpleDecorator` function has been invoked. What is interesting about this code sample, however, is that we have not created an instance of the class named `ClassWithSimpleDecorator` as yet. All that we have done is specify the class definition, added a decorator to it, and the decorator has been called by the JavaScript runtime automatically.

Not having to wait for the creation of an instance of a class tells us that decorators are applied when a class is defined. Let's prove this theory by creating a few instances of this class, as follows:

```
let instance_1 = new ClassWithSimpleDecorator();
let instance_2 = new ClassWithSimpleDecorator();

console.log(`instance_1 : ${JSON.stringify(instance_1)}`);
console.log(`instance_2 : ${JSON.stringify(instance_2)}`);
```

Here, we have created two new instances of `ClassWithSimpleDecorator`, named `instance_1` and `instance_2`. We then log a message to the console to output the value of each class instance. The output of this code is as follows:

```
simpleDecorator called
instance_1 : {}
instance_2 : {}
```

Here, we can see that the `simpleDecorator` function has only been called once, even though we have created two instances of the `ClassWithSimpleDecorator` class.

Decorators are only invoked once, when a class is defined.

Multiple decorators

Multiple decorators can be applied one after another on the same target. As an example of this, let's define a second decorator function as follows:

```
function secondDecorator(constructor: Function) {
    console.log(`secondDecorator called`);
}
```

Here, we have a decorator function named `secondDecorator`, which also logs a message to the console once it has been invoked. We can now apply both `simpleDecorator` (from our earlier code snippet) and `secondDecorator` as follows:

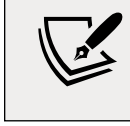
```
@simpleDecorator
@secondDecorator
class ClassWithMultipleDecorators {

}
```

Here, we have applied both decorators to a class named `ClassWithMultipleDecorators`. The output of this code is as follows:

```
secondDecorator called
simpleDecorator called
```

Here, we can see that both of the decorators have logged a message to the console. What is interesting, however, is the order in which they are called.



Decorators are called in the reverse order of their appearance within our code.

Types of decorators

Decorators, as mentioned earlier, are functions that are invoked by the JavaScript runtime when a class is defined. Depending on what type of decorator is used, these decorator functions will be invoked with different arguments. Let's take a quick look at the types of decorators, which are:

- **Class decorators**: These are decorators that can be applied to a class definition
- **Property decorators**: These are decorators that can be applied to a property within a class
- **Method decorators**: These are decorators that can be applied to a method on a class
- **Parameter decorators**: These are decorators that can be applied to a parameter of a method within a class

As an example of these types of decorators, consider the following code:

```
function classDecorator(
    constructor: Function) {}

function propertyDecorator(
    target: any,
    propertyKey: string) {}

function methodDecorator(
    target: any,
    methodName: string,
    descriptor?: PropertyDescriptor) {}

function parameterDecorator(
    target: any,
    methodName: string,
    parameterIndex: number) {}
```

Here, we have four functions, each with slightly different parameters.

The first function, named `classDecorator`, has a single parameter named `constructor` of type Function. This function can be used as a class decorator.

The second function, named `propertyDecorator`, has two parameters. The first parameter is named `target`, and is of type any. The second parameter is named `propertyKey` and is of type string. This function can be used as a property decorator.

The third function, named `methodDecorator`, has three parameters. The first parameter, named `target`, is of type any, and the second parameter is named `methodName`, and is of type string. The third parameter is an optional parameter named `descriptor`, and is of type `PropertyDescriptor`. This function can be used as a method decorator.

The fourth function is named `parameterDecorator`, and also has three parameters. The first parameter is named `target`, and is of type any. The second parameter is named `methodName`, and is of type string. The third parameter is named `parameterIndex`, and is of type number. This function can be used as a parameter decorator.

Let's now take a look at how we would use each of these decorators as follows:

```
@classDecorator
class ClassWithAllTypesOfDecorators {
    @propertyDecorator
    id: number = 1;

    @methodDecorator
    print() { }

    setId(@parameterDecorator id: number) { }
}
```

Here, we have a class named `ClassWithAllTypesOfDecorators`. This class has an `id` property of type number, a `print` method, and a `setId` method. The class itself has been decorated by our `classDecorator`, and the `id` property has been decorated by the `propertyDecorator`. The `print` method has been decorated by the `methodDecorator` function, and the `id` parameter of the `setId` function has been decorated by the `parameterDecorator`.

What is important to note about decorators is that it is the number of parameters and their types that distinguish whether they can be used as class, property, method, or parameter decorators. Again, the JavaScript runtime will fill in each of these parameters at runtime.

Decorator factories

On occasion, we will need to define a decorator that has parameters. In order to achieve this, we will need to use what is known as a decorator factory function. A decorator factory function is created by wrapping the decorator function itself within a function, as follows:

```
function decoratorFactory(name: string) {
    return (constructor: Function) => {
        console.log(`decorator function called with : ${name}`);
    }
}
```

Here, we have a function named `decoratorFactory` that accepts a single parameter named `name` of type string. Within this function, we return an anonymous function that has a single parameter named `constructor` of type Function. This anonymous function is our decorator function itself, and will be called by the JavaScript runtime with a single argument. Within the decorator function, we are logging a message to the console that includes the `name` parameter passed in to the `decoratorFactory` function. We can now use this decorator factory as follows:

```
@decoratorFactory('testName')
class ClassWithDecoratorFactory {

}
```

Here we have applied the decorator named `decoratorFactory` to a class named `ClassWithDecoratorFactory`, and supplied the string value of `"testName"` as the name argument. The output of this code is as follows:

```
decorator function called with : testName
```

Here, we can see that the anonymous function returned by the `decoratorFactory` function was invoked with the string `"testName"` as the value of the `name` argument.

There are two things to note regarding decorator factory functions. Firstly, they must return a function that has the correct number of parameters, and types of parameters, depending on what type of decorator they are. Secondly, the parameters defined for the decorator factory function can be used anywhere within the function definition, which includes within the anonymous decorator function itself.

This concludes our discussion of the setup and use of decorators. In the next section of this chapter, we will explore each of these types of decorators in a little more detail.

Exploring decorators

In this section of the chapter, we will work through each of the different types of decorators and experiment with the information that is provided by each decorator function. We will then use what we have learned in order to provide examples of some practical applications of decorators.

Class decorators

We know that in order to define a class decorator, we must define a function that has a single parameter, which is of type Function. Let's take a closer look at this parameter, as follows:

```
function classConstructorDec(constructor: Function) {
    console.log(`constructor : ${constructor}`);
}

@classConstructorDec
class ClassWithConstructor {
    constructor(id: number) { }
}
```

Here, we have a decorator function named `classConstructorDec`, which is logging the value of the `constructor` argument to the console. We have then applied this decorator to a class named `ClassWithConstructor`. This `ClassWithConstructor` class has a single `constructor` function that accepts a single parameter named `id`, of type number. The output of this code is as follows:

```
constructor : function ClassWithConstructor(id) {
    }
```

Here, we can see that the decorator was invoked with a single argument, which is the definition of the constructor function itself. Note that this definition is the JavaScript definition, and not the TypeScript definition, as there is no type for the `id` parameter. What this is showing us is that a class decorator will be called with the definition of the class constructor itself. Note that the exact output of this code depends on the target version that we have specified, which is `"es5"`. If we change this to `"es6"`, we will generate slightly different output.

Let's now update our `classConstructorDec` decorator and use it to modify the class definition itself, as follows:

```
function classConstructorDec(constructor: Function) {
    console.log(`constructor : ${constructor}`);
    constructor.prototype.testProperty = "testProperty_value";
}
```

Here, we have added a line to our `classConstructorDec` decorator that is using the `prototype` property to modify the class definition itself and has added a property named `testProperty`. The value of this `testProperty` is set to the string `"testProperty_value"`. We can see the effect of this decorator modifying the class definition when we construct an instance of this class as follows:

```
let classInstance = new ClassWithConstructor(1);
console.log(`classInstance.testProperty =
    ${(<any>classInstance).testProperty}`);
```

Here, we are creating an instance of the `ClassWithConstructor` class, named `classInstance`. We are then logging the value of the `testProperty` property of this class instance to the console. Note that we need to cast the `classInstance` variable to a type of any in order to access this property, as it does not appear on the initial class definition. The output of this code is as follows:

```
classInstance.testProperty =
    testProperty_value
```

Here, we can see that the class decorator has added a property named `testProperty` to the instance of the class and set its value to `"testProperty_value"`.

Property decorators

Property decorators, as we have seen, are decorators that can be used on class properties. Property decorators have two parameters, which are the class prototype itself and the property name. Let's take a look at these parameters, as follows:

```
function propertyDec(target: any, propertyName: string) {
    console.log(`target : ${target}`);
    console.log(`target.constructor : ${target.constructor}`);
    console.log(`propertyName : ${propertyName}`);
}
```

Here, we have defined a property decorator named `propertyDec`, which has two parameters. The first parameter is named `target`, and is of type any, and the second parameter is named `propertyName`, and is of type string. Within this decorator function, we are logging three things to the console. Firstly, we are logging the value of the `target` property itself, and then we are logging the value of the `constructor` property of the target object. Finally, we are logging the value of the `propertyName` parameter. Let's now use this decorator as follows:

```
class ClassWithPropertyDec {
    @propertyDec
    nameProperty: string | undefined;
}
```

Here, we have defined a class named `ClassWithPropertyDec`, which is decorating a single property named `nameProperty` with our `propertyDec` decorator. The output of this code is as follows:

```
target : [object Object]
target.constructor : function ClassWithPropertyDec() {
    }
propertyName : nameProperty
```

Here, we can see that the `target` argument that was passed to our decorator is an object, as we would expect, because it is, in fact, a class definition. We can also see from the second line of the console output that this object has a `constructor` function, and we are able to print its definition to the console. We also have the name of the property that we decorated passed in as the `propertyName` argument.

Static property decorators

Property decorators can also be applied to static class properties in the same way that they can be applied to normal properties. The resulting arguments that are passed into our decorator will be slightly different, however. Remember that the JavaScript runtime will fill in these decorator arguments for us, and we are therefore given the JavaScript version of these arguments.

Let's now try and decorate a static class property with the same decorator that we have just used, as follows:

```
class StaticClassWithPropertyDec {
    @propertyDec
    static staticProperty: string;
}
```

Here, we have applied the `propertyDec` decorator to a property named `staticProperty` on a class named `StaticClassWithPropertyDec`. We have marked this property as static, however. The outputs of the various console logs within our decorator are as follows:

```
target : function StaticClassWithPropertyDec() {
    }
target.constructor : function Function() { [native code] }
propertyName : staticProperty
```

Here, we can see that the `target` argument is now a function, where it was an object, or, more accurately, a class prototype object previously. The `constructor` property of this function is now a function, and the `propertyKey` argument contains the name of our property.

Let's now update our `propertyDec` property decorator to correctly identify the class name in both of these cases, as follows:

```
function propertyDec(target: any, propertyName: string) {
    if (typeof (target) === 'function') {
        console.log(`class name : ${target.name}`);
    } else {
        console.log(`class name : `
            + `${target.constructor.name}`);
    }
    console.log(`propertyName : ${propertyName}`);
}
```

Here, we have updated the `propertyDec` decorator function. Within this function, we are checking whether the type of `target` argument is, in fact, a function by using `typeof`. If it is a function, we are logging the `name` property of the function to the console. This will output the name of the class that has the static property. If the type of `target` argument is not a function, then we know that it will be an object prototype, and can then access the `name` property of the `constructor` property of this object prototype. The output of this code is as follows:

```
class name : ClassWithPropertyDec
propertyName : nameProperty
class name : StaticClassWithPropertyDec
propertyName : staticProperty
```

Here, we can see the results of our updated property decorator being applied to a normal class, and then being applied to a class with a static property. In both instances, we are now able to determine the name of the class, and the name of the property that was decorated, irrespective of whether the property was marked static or not.

Method decorators

Recall that method decorators can be applied to methods within a class and that they have three parameters. Let's take a look at a method decorator as follows:

```
function methodDec(
    target: any,
    methodName: string,
    descriptor?: PropertyDescriptor
) {
    console.log(`target: ${target}`);
    console.log(`methodName : ${methodName}`);
    console.log(`descriptor : ${JSON.stringify(descriptor)}`);

    console.log(`target[methodName] : ${target[methodName]}`);
}
```

Here, we have a method decorator named `methodDec`, which has three parameters, namely `target` of type any, `methodName` of type string, and an optional parameter named `descriptor` of type `PropertyDescriptor`. Within this decorator, we are logging the values of the `target`, `methodName`, and `descriptor` parameters to the console. Finally, we log the value of the `methodName` property of the `target` object to the console, using the syntax `target[methodName]`. Let's apply this decorator to a class as follows:

```
class ClassWithMethodDec {
    @methodDec
    print(output: string) {
        console.log(`ClassWithMethodDec.print`
            + `(${output}) called.`);
    }
}
```

Here, we have a class named `ClassWithMethodDec`, which has a single method named `print`, which has a single parameter named `output` of type string. This `print` method has been decorated with our `methodDec` decorator. The output of this code is as follows:

```
target: [object Object]
methodName : print
descriptor : {"writable":true,"enumerable":true,"configurable":true}
target[methodName] : function (output) {
        console.log("ClassWithMethodDec.print"
            + ("(" + output + ") called."));
    }
```

Here, we can see the values of each of the arguments that were passed into the decorator from the JavaScript runtime. The `target` argument indicates that this is an object, as it would be for a class definition. The `methodName` argument is the name of the method that was decorated, and is the string `"print"`. The `descriptor` argument shows the properties of this particular method, which is that it is enumerable, writable, and configurable. These property descriptors can be used to make a class method `readonly`, for instance, by setting the `writable` property to `false`.

The final message logged to the console is the result of `target[methodName]`, which is the definition of the function itself. Similar to the `constructor` argument that is seen in class decorators, we have the full definition of this particular method at our disposal.

With these arguments at hand, we can actually modify this method definition to do something slightly different, as we will see next.

Using method decorators

When a method decorator is called by the JavaScript runtime, it includes the `target` object that the method is available on, and it also includes the name of the method. We can then use this information to modify the original function. Let's take a look at an example that will create an audit trail of some sort and log a message to the console every time a function is called.

Consider the following method decorator:

```
function auditLogDec(target: any,
    methodName: string,
    descriptor?: PropertyDescriptor) {

    let originalFunction = target[methodName];

    let auditFunction = function (this: any) {
        console.log(`1. auditLogDec : overide of `
            + ` ${methodName} called`);
```

```
            for (let i = 0; i < arguments.length; i++) {
                console.log(`2. arg : ${i} = ${arguments[i]}`);
            }
            originalFunction.apply(this, arguments);
        }

        target[methodName] = auditFunction;
        return target;
    }
```

Here, we have a method decorator named `auditLogDec`, with the three parameters needed for a method decorator named `target`, `methodName`, and `descriptor`. Within this function, we create a variable named `originalFunction`, and set its value to the function of the decorated class, that is, `target[methodName]`. We then create a new function named `auditFunction` that does three things. Firstly, it logs a message to the console. Secondly, it loops through all of the arguments that were provided to it and then logs a message to the console with their values. Thirdly, it calls the `apply` function on the `originalFunction` variable, passing in a `this` argument, and the `arguments` argument that it was invoked with.

After defining `auditFunction` and, within it, invoking the `originalFunction` of our decorated class, we set the value of the original class function to the new function, on the line `target[methodName] = auditFunction`, and return the updated class definition named `target`.

In essence, we have made a copy of the original class function that we decorated, wrapped the call to this function within a function of our own making, and then substituted the original function with our own function. In so doing, we have modified the decorated method on the class definition itself, all within a decorator.

To show this technique in action, let's create a class definition as follows:

```
class ClassWithAuditDec {
    @auditLogDec
    print(arg1: string, arg2: string) {
        console.log(`3. ClassWithMethodDec.print`
            + `(${arg1}, ${arg2}) called.`);
```

```
    }
}

let auditClass = new ClassWithAuditDec();
auditClass.print("test1", "test2");
```

Here, we have a class named `ClassWithAuditDec`, which has a single class method named `print`. This print function has been decorated with our `auditLogDec` method decorator. The `print` function logs a single message to the console, indicating what arguments it was invoked with. We then create an instance of this class, named `auditClass`, and call the `print` function. The output of this code is as follows:

```
1. auditLogDec : overide of print called
2. arg : 0 = test1
2. arg : 1 = test2
3. ClassWithMethodDec.print(test1, test2) called.
```

Here, we can clearly see the effects of our decorator function being applied to the `print` method of the `ClassWithAuditDec` class. Our decorator function has intercepted the call to the `print` function, logged a few messages to the console, and then invoked the original function itself.

Using method decorators provides us with a powerful technique of injecting extra functionality into a class method.

Parameter decorators

Parameter decorators can be used to decorate a specific parameter within a method of a class. As an example, consider the following decorator:

```
function parameterDec(target: any,
    methodName: string,
    parameterIndex: number) {

    console.log(`target: ${target}`);
    console.log(`methodName : ${methodName}`);
    console.log(`parameterIndex : ${parameterIndex}`);

}
```

Here, we have defined a parameter decorator named `parameterDec`, which has the required number and types of parameters such that it can be used as a parameter decorator. The first parameter is named `target`, and is of type any, and will contain the object definition for the class itself. The second parameter is named `methodName`, and is a string that will contain the name of the method that has been called.
The third parameter is named `parameterIndex`, and will contain the index of the parameter that has been decorated, which is of type number. Let's now use this parameter decorator as follows:

```
class ClassWithParamDec {
    print(@parameterDec value: string) {
    }
}
```

Here, we have a class named `ClassWithParamDec`, which has a single function named `print`. This print function has a single parameter named `value`, of type string. We have decorated this `value` parameter with our `parameterDec` parameter decorator. The output of this code is as follows:

```
target: [object Object]
methodName : print
parameterIndex : 0
```

Here, we can see that the values of the arguments that were passed into our parameter decorator are what we expected. The `target` argument contains the object definition itself, the `methodName` argument contains the name of the method that was used, and the `parameterIndex` argument shows that this is the first argument of the class method.

Note that we are not given any information by the JavaScript runtime about the parameter that we are decorating. We are not told what type it is, or what name the parameter is. All we are told is that the parameter is at index `0` of the method named `print`.

Decorator metadata

The TypeScript compiler includes experimental support for decorators to carry extra metadata when they are used. This metadata provides us with a little more information with regard to how a decorator is used. In order to activate this feature, we will need to set the `emitDecoratorMetadata` flag in our `tsconfig.json` file to `true`, as follows:

```
{
    "compilerOptions": {
        // other compiler options
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

Let's now take a closer look at the effect that this compile option has on our generated JavaScript. Consider the following parameter decorator:

```
function metadataParameterDec(
    target: any,
    methodName: string,
    parameterIndex: number
) {}

class ClassWithMetadata {
    print(
        @metadataParameterDec id: number, name: string
    ) {}
}
```

Here, we have a parameter decorator named `metadataParameterDec`. This decorator is being applied to the `id` parameter of the `print` method of the `ClassWithMetadata` class definition.

If the `emitDecoratorMetadata` flag of our `tsconfig.json` file is set to `false`, or is not present, then the compiler will emit the following JavaScript:

```
function metadataParameterDec(target, methodName, parameterIndex) {
}
var ClassWithMetadata = /** @class */ (function () {
    function ClassWithMetadata() {
    }
    ClassWithMetadata.prototype.print = function (id, name) {
    };
    __decorate([
        __param(0, metadataParameterDec)
    ], ClassWithMetadata.prototype, "print", null);
    return ClassWithMetadata;
}());
```

Here, the generated JavaScript defines a standard JavaScript closure for our class named `ClassWithMetadata`. The code that is of interest is near the bottom of this closure, where the TypeScript compiler has injected a method named `__decorate`. We will not concern ourselves with the full functionality of this `__decorate` method, but what is of interest is the fact that it contains information about the parameter at index `0`, and that it is decorated with the `metadataParameterDec` decorator.

When the `emitDecoratorMetadata` option in our `tsconfig.json` file is set to `true`, however, the generated JavaScript will contain more information, as follows:

```
function metadataParameterDec(target, methodName, parameterIndex) {
}
var ClassWithMetadata = /** @class */ (function () {
    function ClassWithMetadata() {
    }
    ClassWithMetadata.prototype.print = function (id, name) {
    };
    __decorate([
        __param(0, metadataParameterDec),
        __metadata("design:type", Function),
        __metadata("design:paramtypes", [Number, String]),
        __metadata("design:returntype", void 0)
    ], ClassWithMetadata.prototype, "print", null);
    return ClassWithMetadata;
}());
```

Here, we can see that the call to the `__decorate` function now includes three additional array elements, each being a call to a new function named `__metadata`. Again, we will not go into depth on what these function calls are, other than to demonstrate that the compiler is registering information about the class method itself.

The `"design:type"` key is used to specify that the `print` method is, in fact, a function. The `"design:paramtypes"` key is used to specify the types of each method parameter, which in this case is Number and String, and the `"design:returntype"` key indicates what the return type of this function is, which in this case is void.

Let's now explore how we can use these metadata values to extract information about a method at runtime.

Using decorator metadata

In order to put this extra information that is provided to a decorator to use, we will need to install a third-party library named `reflect-metadata`, which can be installed using npm:

```
npm install reflect-metadata
```

Once this package has been installed, we can import this library as follows:

```
import 'reflect-metadata';
```

We can now start to use this metadata by calling the `Reflect.getMetadata` function that this library provides, as follows:

```
function reflectParameterDec(target: any,
    methodName: string,
    parameterIndex: number)
{

    let designType = Reflect.getMetadata(
        "design:type", target, methodName);
    console.log(`design type: ${designType.name}`)

    let designParamTypes = Reflect.getMetadata(
        "design:paramtypes", target, methodName);
    for (let paramType of designParamTypes) {
        console.log(`param type : ${paramType.name}`);
    }

    let designReturnType = Reflect.getMetadata(
        "design:returntype", target, methodName);
    console.log(`return types : ${designReturnType.name}`);
}
```

Here, we have a decorator named `reflectParameterDec`, which has the required parameters and types to be used as a parameter decorator. Within this function, we are making three calls to the `Reflect.getMetadata` function. Each call uses the keys that we saw earlier, namely, `"design:type"`, `"design:paramtypes"`, and `"design:returntype"`, and logs the results to the console. Let's now apply this decorator to a parameter, as follows:

```
class ClassWithReflectMetaData {
    print(
        @reflectParameterDec
        id: number,
        name: string
    ): number
    {
        return 1000;
    }
}
```

Here, we have a class definition for a class named `ClassWithReflectMetaData`, which has a single method named `print`. The `print` method has two parameters, named `id` of type number, and `name` of type string. The method returns a number. We have decorated the first parameter, named `id`, with our `reflectParameterDec` decorator. The output of this code is as follows:

```
design type: Function
param type : Number
param type : String
return types : Number
```

Here, we can see that we have quite a lot of information with regard to the `print` method. We can see that the `"design:type"` metadata records the fact that the `print` method is, in fact, a function. We can also see that the `"design:paramtypes"` metadata records an array holding the types of each of the two parameters that are part of the method definition. The first parameter is a Number, and the second parameter is a String. Finally, we can see that this function has a return type of type Number.

The information that is recorded by the TypeScript compiler when using the `emitDecoratorMetadata` compiler flag can be read and interpreted at runtime. Remember that type information which is used by our code, and the TypeScript compiler, is compiled away in the resulting JavaScript. Using decorator metadata allows us to retain some of this type information and opens the door to using this type of information to generate code analysis tools, for example, or to write frameworks for dependency injection.

Summary

In this chapter, we have explored the use of decorators within TypeScript. We started by setting up our environment to make use of decorators, and then discussed the syntax used to apply decorators. We learned that we can apply decorators to classes, class properties, class methods, and even class parameters. Each of these decorators has its own set of required parameters and parameter types, based on where it is to be used. We then discussed the use of decorators and saw how we can build an audit log trail using method decorators. In the final part of this chapter, we took a look at decorator metadata, and how this metadata provides additional information about our classes at runtime. In the next chapter, we will explore the integration between TypeScript and JavaScript, and how these two languages can co-exist within the same project.

7
Integration with JavaScript

One of the most appealing facets of JavaScript development is the wealth of external JavaScript libraries that have already been published, are tried and tested, and are available for re-use. Libraries such as jQuery, Underscore, Backbone, and Moment have been around for years, are well documented, and can be used to enhance the JavaScript development experience. As we know, TypeScript generates JavaScript, so we can easily use these libraries and frameworks in TypeScript.

JavaScript is also valid TypeScript, and we can even rename a standard JavaScript file to a TypeScript file just by changing the `.js` file extension to a `.ts` file extension, if we were converting existing JavaScript files to TypeScript. Remember that the strict typing syntax that is used in TypeScript is entirely optional, and we can therefore slowly start to introduce types into a renamed file. The strict typing syntax is also known as syntactic sugar, which can be sprinkled on top of any JavaScript code as and when we please.

In this chapter, we will take a look at how we can enhance a JavaScript library and add this syntactic sugar through declaration files, and import them using npm. We will then take a quick look at how we can use almost all of the TypeScript language features within a declaration file. Finally, we will discuss the use of a few compiler options that assist with integrating JavaScript and TypeScript source files within the same project, and how to generate declaration files from TypeScript source files. This chapter will cover:

- Declaration files, including:
    - Global variables
    - JavaScript code embedded in HTML

    - Finding declaration files with npm
    - Writing declaration files
    - Using the `module` keyword
- Declaration file typing
- The `allowJs` and `outDir` options
- Compiling JavaScript
- The `declaration` option

We'll start by looking at declaration files.

Declaration files

A declaration file is a special type of file used by the TypeScript compiler. It is only used during the compilation step and is used as a sort of reference file to describe JavaScript. Declaration files are similar to the header files used in C or C++ or the interfaces used in Java. They simply describe the structure of available functions and properties but do not provide an implementation. In this section of the chapter, we will take a look at these declaration files, what they are, and how to write them.

Global variables

Most modern websites use some sort of server engine to generate the HTML pages that we use. If you are familiar with the Microsoft stack of technologies, then you will know that ASP.NET MVC is a very popular server-side rendering engine. If you are a Node developer, then you may be using one of the popular Node packages to help construct web pages through templates, such as Jade, Handlebars, or **Embedded JavaScript** (**EJS**).

Within these templating engines, you may sometimes include some JavaScript within the rendered HTML as a result of your server-side logic. As an example of this, let's assume that you keep a list of contact email addresses on your server, and then surface them through a global variable, as follows:

```
<body>
    <script type="text/javascript">
        var CONTACT_EMAIL_ARRAY = [
            "help@site.com",
            "contactus@site.com",
            "webmaster@site.com"
```

```
        ]
    </script>
</body>
```

Here, we can see the `body` section of an HTML page that includes a `script` tag to define a variable named `CONTACT_EMAIL_ARRAY`. This array holds a number of email addresses. The `CONTACT_EMAIL_ARRAY` variable is available to any JavaScript `<script>` block that runs after this on the web page, and is an example of a global variable.

Let's assume that we are writing some TypeScript that needs to read this global variable, as follows:

```
class GlobalLogger {
    public static logGlobalsToConsole() {
        for (let email of CONTACT_EMAIL_ARRAY) {
            console.log(`found contact : ${email}`);
        }
    }
}

window.onload = () => {
    GlobalLogger.logGlobalsToConsole();
}
```

Here, we have a class named `GlobalLogger` that has a single static function named `logGlobalsToConsole`. Within this function, we are looping through the array elements of the `CONTACT_EMAIL_ARRAY` variable and logging a message to the console. We have also defined a function that will be called once the HTML page has loaded, and this function calls the `logGlobalsToConsole` static function.

If we compile this TypeScript code, we will generate the following error:

```
error TS2304: Cannot find name 'CONTACT_EMAIL_ARRAY'
```

This error indicates that TypeScript does not know anything about the variable named `CONTACT_EMAIL_ARRAY`. It does not even know that it is an array. Remember that this array is a global variable and only exists once the HTML page is up and running.

To solve our compilation problem, and make this `CONTACT_EMAIL_ARRAY` variable visible to TypeScript, we will make use of a declaration file. Let's create a file named `globals.d.ts`, that has the special `.d.ts` extension as follows:

```
declare const CONTACT_EMAIL_ARRAY: string[];
```

The first thing to note about this code is that we have used the TypeScript keyword `declare`. The `declare` keyword tells the compiler that we want to define something, but that the implementation of this object (or variable or function) will be resolved at runtime. Following the `declare` keyword, we have a standard variable definition for the variable `CONTACT_EMAIL_ARRAY`, and have set it to be of type string array. This `declare` keyword then does two things for us. Firstly, it allows the variable to be used within our TypeScript code, and secondly, it strongly types this variable to be an array of strings.

Let's update our HTML file to load the generated `GlobalLogger.js` JavaScript file as follows:

```
<head>
    <script src="GlobalLogger.js"></script>
</head>
```

Here, we have included a `script` tag within the header section of our HTML file to load the `GlobalLogger.js` file.

In order to load this HTML file in a browser, however, we will need to start up a web server and serve this page somehow. Luckily, there is a handy Node-based web server package that can be used for this exact purpose, named `http-server`. We can install this package from the command line as follows:

```
npm install -g http-server
```

With the `http-server` package installed, we can just run it from the command line, as follows:

```
http-server
```

Running this command from the command line will start up a server within the current directory, and start a server on port `8080`, as can be seen from the console output:

```
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://192.168.0.6:8080
Hit CTRL-C to stop the server
```

With the server running, we can now navigate to `http://localhost:8080/index.html` and check the console for the output of our `GlobalLogger` class:

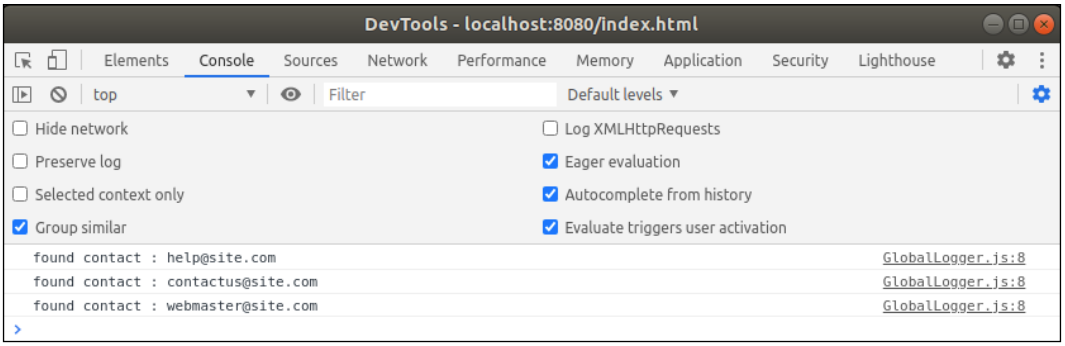


Figure 7.1: Console tab of Chrome Debugger Tools showing log messages

Here, we can see that the `logGlobalsToConsole` function was invoked, and it found the three strings within the `CONTACT_EMAIL_ARRAY` global variable.

So what have we accomplished thus far? We have seen that in some cases, a piece of JavaScript may be loaded into an HTML page and set a variable that we need access to from within TypeScript code. This particular variable is outside of our TypeScript code and is only set once the HTML page is actually run. TypeScript, therefore, needs to know about this global variable, and what type it is in order for the compilation step to proceed. To accomplish this, we created a declaration file that declared the variable, and its type.

Declaration files also provide IntelliSense capabilities within our IDE. As the TypeScript compiler knows what types are being used, it will load these into memory using the language server feature, just as it would for standard TypeScript files.

JavaScript code in HTML

The sample that we have been working with is an example of tight coupling between the generated HTML content (which contains JavaScript code in script blocks) and our running TypeScript generated code. You may argue, however, that this is a design flaw, and that the web page itself should send a request to an API to load this information. While this is a very valid argument, there are cases where including content in the rendered HTML is actually faster.

There used to be a time where the internet seemed to be capable of sending and receiving vast amounts of information in the blink of an eye. Bandwidth and speed on the internet were growing exponentially, and desktops were getting larger amounts of RAM and faster CPUs. As developers during this stage of the internet highway, we may have stopped thinking about how much RAM a typical user had on their machine.

We may also have stopped thinking about how much data we were sending across the wire, as a modern desktop at the time had sufficient connectivity, RAM, and CPU cycles to easily handle whatever we threw at it.

With the advent of the smartphone, however, people started to want access to their favorite sites wherever they were: on the bus, in the car, or while out jogging. These phones, however, had limited RAM, smaller CPUs, and sometimes a very limited amount of bandwidth to work with. The reason for including some JavaScript within the HTML code is also about latency. It takes time for the application to receive data, and this time could significantly impact an initial screen render.

For these reasons, we must carefully think about the users of our applications. How much data are they going to need before the website renders anything on their screen? The technique of including JavaScript variables or smaller static JSON data within the rendered HTML page often provides us with the fastest way to render a screen on a mobile device. Many popular websites use this technique to quickly render the general structure of the page (the header, side panels, and footers) before the main content is delivered through standard API calls. This technique works well, because the site is rendered extremely quickly and gives the user faster visual feedback.

Finding declaration files

Now that we know what a declaration file is, how do we find one that matches the JavaScript library that we are trying to use? Soon after TypeScript was released, Boris Yankov set up a GitHub repository to house TypeScript declaration files for third-party libraries. This repository, named Definitely Typed, quickly became very popular and is now the go-to repository for declaration files. One of the major issues that needed to be addressed in relation to declaration files was the need to match a particular JavaScript library version with the Definitely Typed version that matched that release. The community has, over time, built a number of command-line tools to help with this, including tsd, typings, and NuGet extensions. As of version 2.0 of TypeScript, we are able to use npm to install declaration files. As an example of this, let's install the underscore package and its corresponding types as follows:

```
npm install underscore
npm install @types/underscore --save-dev
```

Here, we are using `npm` to install the JavaScript library `underscore`, and then we are using `npm` to install the corresponding declaration files by prefixing the library name with `@types/`. In fact, if we attempt to use a library without installing the associated `@types` declaration files, the TypeScript compiler will generate an error, as follows:

```
Could not find a declaration file for module 'underscore'
Try `npm i --save-dev @types/underscore` if it exists
```

```
or add a new declaration (.d.ts) file containing `declare module
'underscore';
```

Here, we have attempted to use the underscore library without the associated declaration files. Note that the compiler gives us two solutions here, and the first is to install the `@types` declaration files, if they exist, and the second is to create a new declaration file that just contains the text:

```
declare module 'underscore';
```

Here, we are declaring that there is a module named `'underscore'` that we wish to use, but we are not providing a declaration file for it. This solution is really the last resort and should be avoided where possible. The reason for this is that we will not have any types declared for this library, and it will just be of type any.

Note that over time, some JavaScript libraries have begun to include declaration files within their main package, and therefore we do not even need to install an `@types` package in order to use it.

The popularity and usefulness of the Definitely Typed repository has meant that more and more contributors have written declaration files for just about every JavaScript library that you could wish to use. In general, the only exception to this is when a particular library has had a major version upgrade and the types are lagging behind slightly. This popularity has also meant that the TypeScript team themselves have included a tool for searching for types on their website, named Type Search (`https://www.typescriptlang.org/dt/search?search=`), as shown in the following screenshot:

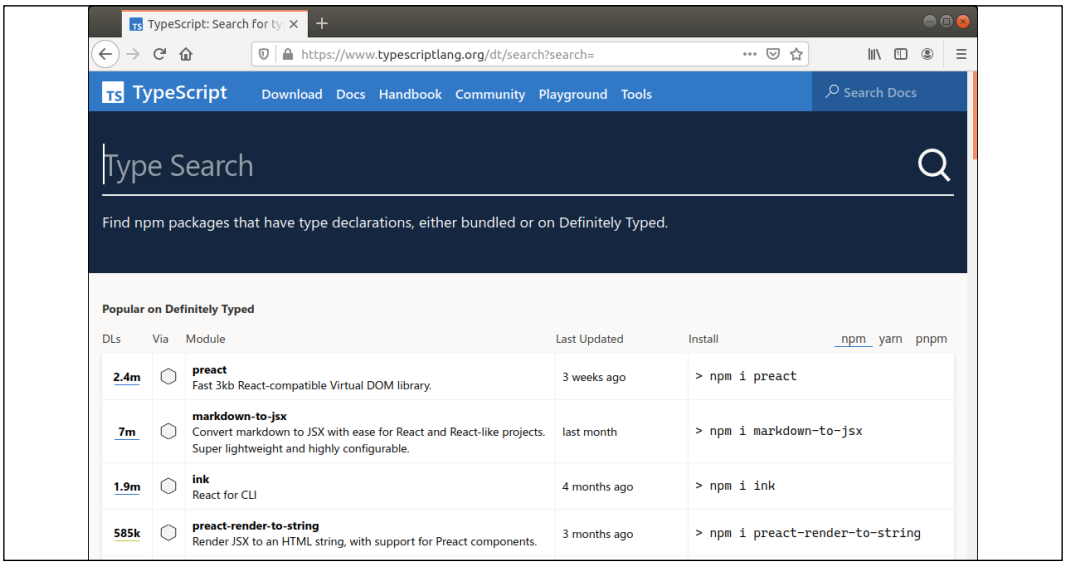


Figure 7.2: The Type Search website main page

Here, we can search for type declaration files and, as can be seen in the help text, declaration files that are either bundled or in the Definitely Typed repository.

Writing declaration files

As we have seen, in order to use JavaScript within a TypeScript project, we will need a declaration file that tells the compiler what functions or objects are made available and what types they are. There may be times, particularly if we are working in a large, established code base, where we need to integrate some home-built JavaScript into our TypeScript project. In these cases, we will need to write a declaration file of our own.

In this section, let's assume that you need to integrate an existing JavaScript helper function into your TypeScript code. As an example of this, consider the following JavaScript code:

```
var ErrorHelper = (function () {
    return {
        containsErrors: function (response) {
            if (!response || !response.responseText) {
                return false;
            }

            var errorValue = response.responseText;

            if (String(errorValue.failure) === "true"
                || Boolean(errorValue.failure)) {
                return true;
            }
            return false;
        },
        trace: function (msg) {
            var traceMessage = msg;
            if (msg.responseText) {
                traceMessage = msg.responseText.errorMessage;
            }
            console.log("[" + new Date().toLocaleTimeString()
                + "] " + traceMessage);
        }
    }
})();
```

Here, we have a variable named `ErrorHelper` that is assigned to what is known as an **Immediately Invoked Function Expression**, or **IIFE**. If we removed the body of this IIFE function, we would be left with the following:

```
var ErrorHelper = (function () { ... })();
```

Here, we can see the simplified version of an IIFE. The `ErrorHelper` variable is assigned to a function, which is wrapped by the parentheses `(` and `)`. After wrapping this function, we invoke it immediately as follows: `();`. This means that the JavaScript runtime will create the `ErrorHelper` object, and assign it to the function as soon as it starts up. IIFEs are the mechanism that older versions of JavaScript use to simulate classes and namespaces.

The `ErrorHelper` IIFE returns an object that has two functions. The first is named `containsErrors` and takes a single parameter named `response`. The `response` parameter is checked by this code to see if it has any errors. We will not go through the logic of this function at this stage, but rather will focus on the usage of this function a little later. The second function is named `trace` and also has a single parameter named `msg`. This function logs a message to the console, which includes information found in the `msg` parameter.

The logic within each of these functions checks for various properties within the parameters that are passed in and executes a different code path depending on what it finds. As an example of the usage of the `containsErrors` and `trace` functions, consider the following JavaScript code:

```
var failureMessage = {
    responseText: {
        "failure": true,
        "errorMessage": "Message From failureMessage"
    }
};

var failureMessageString = {
    responseText: {
        "failure": "true",
        "errorMessage": "Message from failureMessageString"
    }
};

var successMessage = {
    responseText: {
        "failure": false
```

```
    }
};

if (ErrorHelper.containsErrors(failureMessage))
    ErrorHelper.trace(failureMessage);

if (ErrorHelper.containsErrors(failureMessageString))
    ErrorHelper.trace(failureMessageString);

if (!ErrorHelper.containsErrors(successMessage))
    ErrorHelper.trace("success");
```

Here, we can see the various objects that are being checked for errors. The first is named `failureMessage` and has a `responseText` property that has two sub-properties named `failure` and `errorMessage`. The `failure` property is set to the boolean value of `true`, and the `errorMessage` property contains a string value for the error. The second object that is being checked for errors is named `failureMessageString` and has the same structure as the previous object, except that the `failure` property contains a string value of `"true"` instead of a boolean value of `true`. The final object being checked for errors is named `successMessage` and has its `responseText.failure` property set to a boolean value of `false`.

The code then checks the return value from the `containsErrors` function of the `ErrorHelper` class, passing in each of our message objects. In each case, the `trace` function is then called, as each of these `if` statements will return true. Note that the `trace` function is either called with the message object, or it can be called with just a string. The output of this code is as follows:

```
[8:18:18 PM] Message From failureMessage
[8:18:18 PM] Message from failureMessageString
[8:18:18 PM] success
```

Here, we can see that the `containsError` function is returning true for the `failureMessage` object and that the `trace` function is logging the `errorMessage` value to the console. The same behavior is seen for the `failureMessageString` object. The `containsError` function is returning false for the `successMessage` object and is therefore logging the value passed in as the `msg` argument to the console.

What this example is highlighting is that both the `containsErrors` function and the `trace` function can be called with arguments of different types, and with different object structures. Unfortunately, what these object structures are supposed to look like is not exposed by this JavaScript at all. This is one of the underlying drawbacks of a loosely typed language like JavaScript. We can pass anything as a parameter, and the function itself must ensure that it still works.

Let's now write a declaration file for the `ErrorHelper` utility functions and see how it can be used to ensure that this piece of JavaScript is always called with the correct object structures from our TypeScript code.

The module keyword

When writing declaration files, we need a way of grouping functions and properties of classes under the class name. TypeScript uses the `module` keyword as a means of creating a namespace, such that all functions or properties of a class can be grouped together within the class namespace.

Let's take a look at a first version of a declaration file for our `ErrorHelper` JavaScript class, in the file named `globals.d.ts`, as follows:

```
declare module ErrorHelper {
    function containsErrors(response: any): boolean;
    function trace(message: any): void;
}
```

Here, we are using the `declare` keyword combined with the `module` keyword to define a namespace named `ErrorHelper`. Within this module declaration, we have two function definitions, one for the `containsErrors` function and another for the `trace` function. This module declaration is acting as a namespace, meaning that we need to reference each of these functions by their fully qualified names, that is, `ErrorHelper.containsErrors`, and `ErrorHelper.trace`.

Note that even though we have declared the two functions that are available on the `ErrorHelper` class, we are still missing some crucial information about them. The `containsError` function has a single parameter named `response`, which is of type any, and the `trace` function also has a single parameter named `message` of type any. While this definition works, we can really do much better than this, and remove the type of any completely, as follows:

```
interface IResponse {
    responseText: IFailureMessage;
}

interface IFailureMessage {
    failure: boolean | string;
    errorMessage?: string;
}

declare module ErrorHelper {
```

```
    function containsErrors(response: IResponse): boolean;
    function trace(message: IResponse | string): void;
}
```

Here, we have defined an interface named `IResponse` that has a single parameter named `responseText`, of type `IFailureMessage`. The `IFailureMessage` interface has two properties, named `failure` and `errorMessage`. The `failure` property can be of type boolean, or of type string, which matches the use cases that we saw earlier. The `errorMessage` property is of type string. We have then updated the module declaration for the `ErrorHelper` object to properly type the `response` parameter of the `containsErrors` function, to be of type `IResponse`. The `trace` function has also been updated to properly type the message parameter to be of type `IResponse` or type string.

With these changes in place, any TypeScript code that uses these functions will ensure that the desired message structure is adhered to.

Having a detailed declaration file for external JavaScript functions and libraries enhances the available documentation, as it can be integrated directly into the code completion or code hinting engine of the IDE you are using. Declaration files describe code as if it were written directly in TypeScript and will enforce the same strict typing rules for any code that we write using the external JavaScript libraries.

Declaration file typing

As we have seen, declaration files use the `declare` and `module` keywords to define objects and namespaces. We have also seen that we can use interfaces in the same way that we do within TypeScript, in order to define custom types for variables. Declaration files allow us to use the same syntax that we would in TypeScript to describe types. These types can be used everywhere types are used in normal TypeScript, including function overloading, type unions, classes, and optional properties. Let's take a quick look at these techniques, with a few simple code samples, to illustrate this feature further. This section will cover:

- Function overloading
- Nested namespaces
- Classes
- Static properties and functions
- Abstract classes
- Generics
- Conditional types and inference

We'll begin by taking a look at function overloading.

Function overloading

Declaration files allow for function overloads, where the same function can be declared with different arguments, as follows:

```
declare function trace(arg: string | number | boolean);
declare function trace(arg: { id: number; name: string });
```

Here, we have a function named `trace` that is declared twice: once with a single parameter named `arg`, of type string or number or boolean, and once with the same `arg` parameter, which is a custom type. This overloaded declaration allows for all of the following valid code:

```
trace("trace with string");
trace(true);
trace(1);
trace({ id: 1, name: "test" });
```

Here, we have exercised the various combinations of arguments that are allowed by our function overloads.

Nested namespaces

Declaration files allow for module names to be nested. This in turn translates to nested namespaces, as follows:

```
declare module FirstNamespace {
    module SecondNamespace {
        module ThirdNamespace {
            function log(msg: string);
        }
    }
}
```

Here, we have declared a module named `FirstNamespace` that exposes a module named `SecondNamespace`, which in turn exposes a third module named `ThirdNamespace`. The `ThirdNamespace` module defines a function named `log`. This declaration will result in all three namespaces needing to referenced in order to call the `log` function, as follows:

```
FirstNamespace.SecondNamespace.ThirdNamespace.log("test");
```

Here, we are explicitly referencing each named namespace in order to call the `log` function within the `ThirdNamespace` module.

Classes

Class definitions are specified in module definitions using the `class` keyword, as it would be in normal TypeScript files, as follows:

```
declare class MyModuleClass {
    public print(): void;
}
```

Here, we have declared a class named `MyModuleClass` that has a public `print` function that returns `void`. This class definition can then be used as follows:

```
let myClass = new MyModuleClass();
myClass.print();
```

Here, we are creating an instance of the `MyModuleClass` class that has been declared in our declaration file. We then are calling the `print` function of the class instance named `myClass`.

Declaring a class in a declaration file is very similar to defining an interface for it. We do not provide any implementations of the functions, we are only declaring to the TypeScript compiler that the class exists, and what properties and functions are available to it.

Static properties and functions

In the same manner that we can mark a property or function as static in TypeScript, we can use the same syntax in a declaration file, as follows:

```
declare class MyModuleStatic {
    static print(): void;
    static id: number;
}
```

Here, we have declared a class named `MyModuleStatic` that has a static `print` function and a static `id` property. We can use these static properties and functions as follows:

```
MyModuleStatic.id = 10;
MyModuleStatic.print();
```

Here, we can see that the declaration of a static function or property follows the same usage rules as if we had defined it in TypeScript.

Abstract classes

Declaration files can define abstract classes and functions as follows:

```
declare abstract class MyModuleAbstract {
    abstract print(): void
}
```

Here, we have defined an abstract class named `MyModuleAbstract` that has a single function named `print` that has also been marked as abstract. We can use this abstract class declaration in a TypeScript file as follows:

```
class DerivedFromAbstract extends MyModuleAbstract {
    print() { }
}
```

Here, we have defined a class named `DerivedFromAbstract` that extends the `MyModuleAbstract` class from our declaration file. Note that we will also need to provide an implementation of the `print` function within this class definition, as the `print` function has been marked as an abstract function in the class declaration.

Generics

Declaration files allow generic syntax to be used, as follows:

```
declare function sort<T extends number | string>
    (input: Array<T>): Array<T> { }
```

Here, we are declaring a function named `sort` that is using generic syntax to specify the type of `T` to be either a number or a string, or both. This `sort` function has a single parameter named `input` that is an array of type `T` and returns an array of type `T`. This declaration will allow the following usage:

```
let sortedStringArray: Array<string> = sort(["first", "second"]);
let sortedNumericArray: Array<number> = sort([1, 2, 3]);
```

Here, we have defined a variable named `sortedStringArray` to hold the return value of a call to the `sort` function, where we pass in an array of strings as the only argument. The type of the `sortedStringArray` variable will be `Array<string>`. Next, we define a variable named `sortedNumericArray` to hold the return value of another call to the `sort` function, where we pass in an array of numbers as the only argument. The `sortedNumericArray` variable will be of type `Array<number>`.

Conditional types

Declaration files can also define conditional types and distributed conditional types, as follows:

```
declare type stringOrNumberOrBoolean<T> =
    T extends string ? string :
    T extends number ? number :
    T extends boolean ? boolean : never;
```

Here, we have declared a type named `stringOrNumberOrBoolean` that is using generic syntax to define a type of `T`. If `T` extends string, then the type will be string. If `T` extends number, then the type will be number. If the type of `T` extends boolean, then the type will be boolean. If `T` does not extend any of these types, then the type will be never.

We can now use this distributed conditional type as follows:

```
type myNever = stringOrNumberOrBoolean<[string,number]>;
```

Here, we have defined a type named `myNever` that is the result of the distributed conditional type named `stringOrNumberOrBoolean`, and has defined a type of `T` as a tuple of string and number. As this tuple does not match any of the types we are checking for, the type of the `myNever` variable will be never.

Conditional type inference

Declaration files allow for conditional type inference, as can be seen from the following example:

```
declare type inferFromPropertyType<T> =
    T extends { id: infer U } ? U : never;
```

Here, we have a type named `inferFromPropertyType` that will infer the type named `U` from the property named `id` of the type `T`. If the `id` property does not exist, then the type will be never. We can now use this type as follows:

```
type myString = inferFromPropertyType<{ id: string }>;
type myNumber = inferFromPropertyType<{ id: number }>;
```

Here, we have defined two types, named `myString` and `myNumber`, that are using the inferred conditional type named `inferFromPropertyType`. As the type of the `id` property is a string in the first line of this code, then `myString` will be of type string. The type of the `id` property in the second line of code is of type number, and therefore `myNumber` will be of type number.

Declaration file summary

What we have seen in this section of the chapter is that the type rules and typing techniques that TypeScript provides can all be used within declaration files. The purpose of a declaration file is to tell the TypeScript compiler ahead of time what the structure of a JavaScript library looks like. We have seen that we can use all of the TypeScript keywords and language features within a declaration file.

Integration compiler options

There are a few compiler options that we can configure in order to help with the integration of JavaScript and TypeScript. These options will allow us to include JavaScript files within the same project as TypeScript files, as well as allowing us to do some type checking on JavaScript files. We are also able to generate declaration files from our TypeScript code, if we are building libraries for general consumption. In this section of the chapter, we will explore these compiler options.

The allowJs and outDir options

The default TypeScript compilation configuration, as seen in the `tsconfig.json` file, will generate JavaScript files in the same directory as the source TypeScript files. The `outDir` compilation option is used to specify a different output directory for the generated JavaScript, as follows:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "outDir": "./dist",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    }
}
```

Here, we have a `tsconfig.json` file that has specified the `outDir` property for generated JavaScript, which is set to the `./dist` directory. When the TypeScript compiler is invoked, all of the generated JavaScript will be written to the `./dist` directory, instead of being generated in the same directory as the source TypeScript files.

This compile option is generally used in order to create a distributable version of the generated JavaScript. By writing output to a separate directory, we are able to keep the TypeScript source files away from the generated JavaScript and create a version of our project that is just pure JavaScript. Having a distributable version of the generated JavaScript also allows us to run post-compilation steps, such as minification, uglification, or bundling.

In a TypeScript-only project, in other words, where all of the source files are written in TypeScript, this is a convenient and viable option. Unfortunately, if we are mixing JavaScript source files with TypeScript source files in the same project, only the TypeScript generated JavaScript will be written to this directory.

If our project includes both JavaScript source files as well as TypeScript source files, we can turn on another compilation option in the `tsconfig.json` file named `allowJs` as follows:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "allowJs" : true,
        "outDir": "./dist",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    }
}
```

Here, we have set the `allowJs` option of the `tsconfig.json` file to `true`, which will also include any JavaScript files found in our source directory in the output directory of `./dist`.

Let's take a quick look at this in action, with the following directory structure for our source files:

```
.
├── TypeScriptSourceFile.ts
├── tsconfig.json
└── jsFiles
    └── JavaScriptSourceFile.js
```

Here, we have a directory structure that has a TypeScript source file named `TypeScriptSourceFile.ts` and the `tsconfig.json` file in the project root directory. We then have a subdirectory named `jsFiles` that includes a source JavaScript file named `JavaScriptSourceFile.js`. Once we have executed the compilation step, the directory structure becomes the following:

```
.
├── TypeScriptSourceFile.ts
├── tsconfig.json
├── jsFiles
│   └── JavaScriptSourceFile.js
└── dist
    ├── TypeScriptSourceFile.js
    └── jsFiles
        └── JavaScriptSourceFile.js
```

Here, we can see that the TypeScript compiler has created a subdirectory named `dist` to house the generated files. The `TypeScriptSourceFile.js` file has been created within this directory, as well as a subdirectory named `jsFiles`, which contains the included JavaScript. The directory structure within the `dist` directory matches the original directory structure of our source code.

Note that the `tsconfig.json` file has not been copied into the `dist` directory. The `dist` directory only contains the output of the compilation step.

When using the `allowJs` compiler option, and compiling directly with `tsc`, we must also specify an `outDir` option. This is because the compiler will attempt to generate a JavaScript file for each source JavaScript file. If we are not using the `outDir` option, the compiler will attempt to overwrite the original JavaScript file with the generated JavaScript file within the same directory. As these files are both the same name, an error will occur.

Compiling JavaScript

When we initialize a TypeScript project and issue the `tsc --init` command from the command line, TypeScript will generate a `tsconfig.json` file for us, with the default compile options. This file actually includes all of the available command-line options within it, but they are commented out and therefore not turned on.

Along with the full list of available options, the `tsconfig.json` file also includes a comment for each of the command-line options, in order to remind us what each of these options is used for. Let's take a look at the comments for the `allowJs` option as follows:

```
"allowJs": true, /* Allow javascript files to be compiled. */
```

Here, we can see that the `allowJs` compiler option has a comment stating that we are allowing source JavaScript files to be compiled. What this means is that the TypeScript compiler has the ability do to a TypeScript compilation step, even if the source file is JavaScript.

Let's put this theory to the test and create a JavaScript file that uses ES6 syntax as follows:

```
class MyEs6Class {
    _id;
    constructor(id) {
        this._id = id;
    }
    get id() {
        return this._id;
    }
}
```

Here, we have an ES6 JavaScript class named `MyEs6Class` that has an `_id` property, a `constructor` function, and a `get` function name `id` that will return the internal `_id` property. What is interesting to note, however, is that the `class` keyword has only been made available in ES6 and above and is not available for use in earlier JavaScript versions. In the same manner, `get` and `set` functions were also only introduced in the ES6 JavaScript standard and are also not available for use in earlier JavaScript versions.

So what we have here, then, is an ES6-compatible JavaScript class included in our TypeScript project.

The default target property in our `tsconfig.json` file is set to `"es5"`, meaning that the TypeScript compiler will generate ES5-compatible JavaScript for all source files. Let's now take a look at the generated JavaScript for this JavaScript source file in the `dist` directory, as follows:

```
"use strict";
var MyEs6Class = /** @class */ (function () {
    function MyEs6Class(id) {
        this._id = id;
    }
    Object.defineProperty(MyEs6Class.prototype, "id", {
        get: function () {
            return this._id;
        },
        enumerable: false,
```

```
        configurable: true
    });
    return MyEs6Class;
}());
```

Here, we can see that the generated JavaScript is, in fact, ES5-compatible, and that the `class` keyword within our ES6 JavaScript source file has been converted into a variable named `MyEs6Class`, and is using an IIFE to create compatible ES5 JavaScript. The `constructor` function has been transformed into a standard function, and the `get` function has been transformed into a property using the `Object.defineProperty` function.

What this shows us is that the TypeScript compiler is generating ES5-compatible JavaScript from our ES6 JavaScript source file. This is what the comment `"Allow javascript files to be compiled"` really means.

The declaration option

The final TypeScript compilation option that we will discuss in this chapter is the `declaration` option, which will generate declaration files from our source TypeScript, or our source JavaScript files. We can turn this option on by un-commenting it in the `tsconfig.json` file as follows:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "allowJs": true,
        "declaration": true,
        ... other options
    }
}
```

Here, we have set the `declaration` compile option to `true`. Running the TypeScript compiler will now generate a declaration file for each of our source files. Let's see what the declaration file would look like for the `MyEs6Class` instance that we wrote earlier in ES6 JavaScript, as follows:

```
declare class MyEs6Class {
    constructor(id: any);
    _id: any;
    get id(): any;
}
```

Here, we can see that a class named `MyEs6Class` has been declared. It has a `constructor` function that has a single parameter named `id`, an internal `_id` property, and a `get` function named `id`. This declaration matches the functionality of the class that we explored earlier. Note, however, that the types used are all any. The `_id` property of the class is of type any, the `id` parameter of the constructor function is of type any, and the `get id` function returns a type of any.

The use of any in the declaration file makes sense, as JavaScript does not have type annotations in the same way that TypeScript does. All properties, function parameters, function return types, and variables are, in fact, of type any.

Let's now take a look at the declaration file generated for a TypeScript function as follows:

```
interface IFilterable {
    name?: string;
}

function filterUndefined<T extends IFilterable>
    (input: Array<T>): Array<T>
{
    let output: Array<T> = [];
    for (let item of input) {
        if (item.name?.length) {
            output.push(item);
        }
    }
    return output;
}
```

Here, we have defined an interface named `IFilterable` that has a single property named `name` of type string that is optional. We then have a function named `filterUndefined` that is using generic syntax to define a type named `T`. The type `T` extends from the `IFilterable` interface, so this function is specifically looking for types that have a property named `name` of type string. The `filterUndefined` function has a single parameter named `input`, which is an array of type `T` and returns an array of type `T`.

Within this function, we are checking whether the `name` property is defined, and if it is, we add it to the array named `output` that will be returned by the function. Let's now look at the generated declaration file for this function, as follows:

```
interface IFilterable {
    name?: string;
}
declare function filterUndefined<T extends IFilterable>
    (input: Array<T>): Array<T>;
```

Here, we can see that the TypeScript compiler has retained the interface `IFilterable` within the declaration file and has created a `declare function` declaration that matches the definition of the `filterUndefined` function.

Summary

In this chapter, we explored the integration points that TypeScript allows when using JavaScript. We discussed how to use external JavaScript libraries or external JavaScript code within TypeScript code through the use of declaration files. We then built a declaration file from some JavaScript and learned how to describe the JavaScript types correctly for use within TypeScript. We then had a quick overview of how types are described in declaration files, and found that almost every language construct and keyword in the TypeScript language can be used within a declaration file. Finally, we took a look at how we can use some compiler options to allow both JavaScript and TypeScript files within the same project, and how TypeScript can generate declaration files for us.

In the next chapter, we will take a look at the strict compiler options that are made available to us, and how these can be used to check for potential errors within our code.

8

Strict Compiler Options

The TypeScript compiler uses the `tsconfig.json` file to specify a number of compilation options. These options include what version of JavaScript we would like to generate for, what the output directory should be, and whether or not to allow JavaScript source files within the project directory. We have already discussed a few of these compiler options on our journey with TypeScript so far.

One of the most important options is simply named `strict`, and this single option turns on a group of other compiler options. The purpose of the strict compiler options is to apply a number of checks to our code and determine whether any of these checks fail. As an example, these strict options can determine if a variable could be undefined at the time of use, or if the variable itself is never used. If we set the value of the strict option to false, or turn it off, then each of these options needs to be turned on individually. In this chapter, we will explore these other strict options.

The strict option is the default that is used when creating a new TypeScript project, and we should strive to keep this option turned on. Relaxing the strict compiler options used by the TypeScript compiler should always be done very carefully, and should be a deliberate act. There are, however, occasions when we should allow for the strict options to be relaxed.

If we are tasked with converting a large JavaScript project into TypeScript, then this is one of these occasions. Airbnb recently went through the process of converting their existing React-based JavaScript projects to TypeScript, and have published an open-source project named `ts-migrate` to help other teams do the same thing. The strategy that Airbnb used was a realistic and pragmatic approach, where the conversion was done in stages. At each stage of the migration project, a small portion of the strict compiler options were introduced, and then errors were subsequently fixed within the code.

In this chapter, we will take an in-depth look at these strict compiler options and a few other options that help with finding potential problems in our code. Understanding these options and what causes code to fail the strict checking rules will help us avoid common programming mistakes when writing TypeScript. Specifically, we will cover the following topics:

- Using nested `tsconfig.json` files
- `strictNullChecks`
- `strictPropertyInitialization`
- `strictBindCallApply`
- `strictFunctionTypes`
- `noImplicityAny`
- `noUnusedLocals`
- `noUnusedParameters`
- `noImplicitReturns`
- `noFallthroughCasesInSwitch`
- `noImplicitThis`

Let's start with nested configuration.

Nested configuration

The TypeScript compiler is able to re-use a `tsconfig.json` file in another directory when compiling code in the current directory. This feature is handy if we would like to override a compiler option when running `tsc` within a specific directory. The `tsconfig.json` file uses the `"extends"` option for this purpose. As an example of this nested configuration, consider the following source tree:

```
├── sub1
│   ├── SampleJsFile.js
│   └── tsconfig.json
├── SampleTsFile.ts
└── tsconfig.json
```

Here, we have a `tsconfig.json` file in the project root directory, as well as a TypeScript file named `SampleTsFile.ts`. We also have a subdirectory named `sub1` that contains a `tsconfig.json` file, and a JavaScript file named `SampleJsFile.js`. The `tsconfig.json` file in the project root directory is as follows:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "outDir": "./dist",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    }
}
```

Here, we have a default `tsconfig.json` file and have specified the `outDir` property to generate all JavaScript output into the `dist` directory. Running `tsc` in the project base directory will only compile `SampleTsFile.ts` and ignore any other source files in the `sub1` directory. This is because the only file in the `sub1` directory is, in fact, a JavaScript source file.

Let's take a look at the `tsconfig.json` file in the `sub1` directory, as follows:

```
{
    "extends": "../tsconfig",
    "compilerOptions": {
        /* Strict Type-Checking Options */
        "outDir": "../dist",
        "allowJs": true,
        "strict": false
    }
}
```

Here, we can see that the configuration is using the `"extends"` property, which is set to `"../tsconfig"`. This means that the TypeScript compiler will load the `tsconfig.json` file found one directory level up and use that as a base for configuration options. Any compiler options present in this file will override the options in the base configuration.

In this instance, we have specified a different option for the `outDir`, `allowJs`, and `strict` options. The `outDir` option has been set to the `dist` directory one directory level up. The `allowJs` option has been set to true, in order to include the `SampleJsFile.js` file in the compilation step. We have also specified a value of false for the `strict` option.

Note that in order for the compiler to use this `tsconfig.json` file, we will need to change into the `sub1` directory and make this the current directory, as follows;

```
cd sub1
```

Running the compiler once we are in the `sub1` directory will now use this `tsconfig.json` file and compile the `SampleJsFile.js` file. The output of this compilation step will add `SampleJsFile.js` to the `dist` directory of the project root, as we have specified the `allowJs` option to compile JavaScript files.

Note that the TypeScript compiler will use the `tsconfig.json` file in the current directory in order to compile our source files. When compiling, it will also apply these settings to any source files found in any sub-directories. If a `tsconfig.json` file exists in a subdirectory, however, the compiler will ignore it. This is why the output of the `tsc` compilation step in the project root directory will only generate the `SampleTsFile.js` file, whereas running the `tsc` compilation step in the `sub1` directory will then generate the `SampleJsFile.js` output file.

The sample code that accompanies this chapter has been written using this technique. In the rest of this chapter, we will start to turn on specific compiler options that are relevant to each topic and work through some code samples to show the effect of these options. Each compiler option that we will explore will have its own subdirectory, which will use the base `tsconfig.json` file, and each section will turn on the relevant compiler options for that topic.

Strict Options

In this section of the chapter, we will explore the strict set of compiler options, all of which come into play when we turn the main `"strict"` option from true to false. Note that if we turn the main `"strict"` option to false, then we must set the desired `"strict"` option to true in order for it to become active.

strictNullChecks

The `strictNullChecks` compiler option is used to find instances in our code where the value of a variable could be null or undefined at the time of usage. This means that when the variable is actually used, if it has not been properly initialized, the compiler will generate an error message. Consider the following code:

```
let a: number;
let b = a;
```

Here, we have defined a variable named `a` that is of type number. We then define a variable named `b` and assign the value of `a` to it. This code will generate the following error:

```
error TS2454: Variable 'a' is used before being assigned
```

This error is telling us that we are attempting to use the value of the variable `a` before it has been assigned a value. Remember that because it has not yet been assigned a value, it could still be undefined.

There are two ways to remove this error message. Firstly, we can ensure that the variable has a value before being used, as follows:

```
let a: number = 2;
let b = a;
```

Here, we have simply assigned the value of `2` to the variable named `a`, and this will remove the error.

The other way to fix this error is to let the compiler know that we are aware that the variable may be unassigned at the time of usage, as follows:

```
let a: number | undefined;
let b = a;
```

Here, we are using a type union to indicate that the variable `a` is either a number or undefined. In this way, we are informing the compiler that we are aware that the variable `a` could be undefined and will take care of this condition ourselves.

strictPropertyInitialization

The `strictPropertyInitialization` compiler option will check that all properties within a class have been initialized correctly. The concept is similar to the `strictNullChecks` option that we just discussed but extends into class properties. Consider the following class definition:

```
class WithoutInit {
    a: number;
    b: string;
}
```

Here, we have a class named `WithoutInit` that has two properties, named `a` of type number and `b` of type string. This code will generate the following errors:

```
error TS2564: Property 'a' has no initializer and is not definitely
assigned in the constructor
error TS2564: Property 'b' has no initializer and is not definitely
assigned in the constructor
```

These errors are being generated because both the `a` and `b` properties of the class have not been initialized. Remember that class properties are public by default, and this means that when we create an instance of this class, we have access to them. This error is warning us that we may inadvertently use these properties while they are still undefined.

There are four ways that we can fix this code. Firstly, we can use a type union, as we did with `strictNullChecks`, as follows:

```
class WithoutInit {
    a: number | undefined;
    b: string | undefined;
}
```

Here, we are using a type union to add the `undefined` type to both the `a` and `b` properties. Again, this lets the compiler know that we are aware that these properties could be undefined and will handle the consequences ourselves.

The second way that we can fix these errors is to use the definite assignment assertion operator, as follows:

```
class WithoutInit {
    a!: number;
    b!: string;
}
```

Here, we have added the `!` operator after each property, which tells the compiler that we are aware that these properties have not been initialized.

The third way to fix these errors is to assign a value to each of these properties, as follows:

```
class WithoutInit {
    a: number = 1;
    b: string = "test";
}
```

Here, we have assigned the numeric value of `1` to the `a` property and the string value of `"test"` to the `b` property. As both of these properties have now been assigned a value, the error message disappears.

The fourth method of fixing these errors is to use a constructor, as follows:

```
class WithoutInit {
    a: number;
    b: string;
    constructor(a: number) {
        this.a = a;
        this.b = "test";
    }
}
```

Here, we have defined a constructor function with a single parameter named `a` of type number. Within the constructor, we are assigning the value of the `a` parameter to the internal `a` property. We are also assigning the string value `"test"` to the property named `b`. With this constructor in place, both of the properties `a` and `b` have been correctly initialized, and the code will not generate any errors.

In order to use the `strictPropertyInitialization` option, we also need to enable the `strictNullChecks` option at the same time, or the compiler will return an error stating this.

strictBindCallApply

JavaScript provides the `bind`, `call`, and `apply` functions that are used to override the value of the `this` variable inside a function. When using the `bind`, `call`, and `apply` functions, we essentially provide a particular version of the object that the function should use as the value for `this` and then invoke the function with the parameters it requires. The `strictBindCallApply` option is used to ensure that we provide these function parameters with the correct types. To illustrate this concept, consider the following code:

```
class MyBoundClass {
    name: string = "defaultNameValue";

    printName(index: number, description: string) {
        console.log(`this.name : ${this.name}`);
        console.log(`index: ${index}`);
        console.log(`description : ${description}`);
    }
}
```

Here, we have a class definition for a class named `MyBoundClass` that has a `name` property of type string, and a function called `printName`. The `printName` function has two parameters, named `index` of type number, and `description` of type string. The `printName` function logs three messages to the console. The first log outputs the value of the `name` property, using the syntax `this.name`. The second log outputs the value of the `index` parameter, and the third log outputs the value of the `description` parameter.

We can now use this class as follows:

```
let testBoundClass = new MyBoundClass();
testBoundClass.printName(1, "testDescr");
```

Here, we have created an instance of the class `MyBoundClass`, and assigned it to a variable named `testBoundClass`. We then call the `printName` function with the arguments of `1` and `testDescr`. The output of this code is as follows:

```
this.name : defaultNameValue
index: 1
description : testDescr
```

Here, we can see that the output matches our expectations. The value of the internal `name` property is the value `defaultNameValue`, and the values of the `index` and `description` arguments are what were passed into the `printName` function.

Using the JavaScript `call` function, however, we can modify the value of the `this` variable. Consider the following call to the `printName` function:

```
testBoundClass.printName.call(
    { name: `overridden name property` }, 1, `whoa !`
);
```

Here, we are invoking the `printName` function on the variable `testBoundClass`, which holds our instance of the class `MyBoundClass`. Note that we are using the JavaScript `call` function to invoke the `printName` function. This `call` function's first parameter is what will be used as the value for `this` inside the class instance. The other parameters match the original function definition, and as such must be a number and a string to match the index and description parameters.

The output of this code is as follows:

```
this.name : overridden name property
index: 1
description : whoa !
```

Here, we can see that the value of `this.name` has been overridden by the value provided to the `call` function. The `call` function provided an object as its first argument, and this object had a property named `name`. When the `printName` function is invoked, therefore, the value of `this.name` is not what was defined in the class definition, but is now what was passed in to the `call` function. The value of the `index` argument and the value of the `description` argument are what were provided to the `call` function.

The `strictBindCallApply` compiler option will check to ensure that the parameters used in the `call` function match the parameters as defined in the original function definition. As an example of this, we might attempt the following:

```
testBoundClass.printName.call(
    { name: `overridden name property` }, "test", `whoa !`
);
```

Here, we are using the `call` function to override the value of `this` within the `printName` function, as we saw earlier, which is the first argument. The second and third arguments, which are the values `"test"` and `` `whoa !` ``, correspond to the `index` and `description` parameters of the original `printName` function. This code will generate the following error:

```
error TS2345: Argument of type '"test"' is not assignable to parameter
of type 'number'.
```

Here, we can see that the compiler is correctly identifying that the `index` parameter of the `printName` function should be of type number and not of type string, hence the error.

When using `bind`, `call` or `apply`, the compiler is therefore checking that we are still honoring the function definition and providing the correct types for each of the function parameters.

The `apply` function is very similar to the `call` function, except that the `apply` function uses an array of arguments, as follows:

```
testBoundClass.printName.apply(
    { name: `apply override` },
    [1, 'whoa !!']
);
```

Here, we are using the `apply` function to override the value that the class will use as `this` and are providing the arguments to the original `printName` function as an array. The `strictBindCallApply` compiler option will check that the arguments provided within the array match the types that are defined in the original `printName` function, as we saw with the `call` function.

The `bind` function follows the same sort of pattern as the `call` and `apply` functions, in that it allows us to override the value of `this`. The difference with the `bind` function, however, is that it returns a function that has this override permanently set. As an example of this, consider the following code:

```
let boundThis = {
    name: `boundThis`
};

let boundPrintNameFunction = testBoundClass.printName.bind(
    boundThis, 1, `testDescription`
);
```

Here, we have defined an object named `boundThis` that has a single property named `name`. We are then creating a variable named `boundPrintNameFunction` that holds the result of a call to the `bind` function. The first argument in the `bind` function is the overridden value of `this`, which in this case is the object named `boundThis`. The second and third arguments correspond to the `index` and `description` parameters for the original `printName` function, as we have seen before. As the `bind` function returns a function, we can now invoke it as follows:

```
boundPrintNameFunction();
```

Here, we are invoking the function that was assigned to the `boundPrintNameFunction` variable. The output of this code is as follows:

```
this.name : boundThis
index: 1
description : testDescription
```

Here, we can see that the `bind` function has overridden the value of `this` to be the value of the `boundThis` object. As we saw with the `call` and `apply` functions, the `strictBindCallApply` compiler option will ensure that a call to the `bind` function has the correct arguments and that the argument types match the original function declaration. We will explore the use of the `bind` function in later chapters, where it is often used to handle callbacks or DOM events, where the value of `this` is modified such that it refers to an instance of a class that is responsible for handling an event.

strictFunctionTypes

When we define types in TypeScript and then attempt to assign them to each other, the compiler makes sure that the types are consistent. Unfortunately, this strict typing rule did not apply correctly in some circumstances when dealing with functions. The `strictFunctionTypes` option corrects this behavior. Let's dive into an example to explain this, as follows:

```
let numberOrString: number | string;
let numberOnly: number = numberOrString;
```

Here, we have defined a variable named `numberOrString`, which is of type number or `string`. We then define a variable named `numberOnly`, which is of type number, and we then attempt to assign the `numberOrString` variable to the `numberOnly` variable. This code will generate the following error:

```
Type 'string | number' is not assignable to type 'number'.
```

Here, we can see that we are not allowed to assign variables that are not the same type to each other. Interestingly, the `numberOrString` variable could possibly be a number, but it could possibly be a string too. The `numberOnly` variable could only possibly be a number, which is why the compiler will not allow this assignment.

Let's now apply the same type error within a callback function, as follows:

```
function withCallback(
    fn: (a: number | string) => void
) {
    fn("test");
}

function withNumberOnly(a: number) {
    console.log(`a : ${a}`);
}

withCallback(withNumberOnly);
```

Here, we define a function named `withCallback` that takes a function named `fn` as its only argument. Note the callback signature of the `fn` function. It has a single parameter named `a`, which is of type number or string. We then define a function named `withNumberOnly`, which takes a single parameter named `a` of type number. We then use the `withNumberOnly` as the callback function in a call to the `withCallback` function.

The `withNumberOnly` function can only be called with a parameter of type number, but it is being used as a callback function where the parameter could be of type number or string. This can easily cause errors in our code. With the `strictFunctionTypes` compiler option turned on, this code will generate the following error:

```
Argument of type '(a: number) => void' is not assignable to parameter
of type '(a: string | number) => void'.
```

Here, we can see that the compiler is correctly identifying the mismatch in the function signatures that we are using.

These function signature mismatches can also occur when using inheritance, as can be seen in the following example:

```
class WithPrint {
    print() { }
}

class WithPrintAndRun extends WithPrint {
    run() { }
}

function usePrint(
    fn: (withPrint: WithPrint) => void
) {
    let withPrint = new WithPrint();
    fn(withPrint);
}

usePrint((withRun: WithPrintAndRun) => {
    withRun.run();
});
```

Here, we have a class named `WithPrint` that has a `print` member function. We then have a class named `WithPrintAndRun` that derives from the `WithPrint` class and defines a `run` member function. We then have a function named `usePrint` that defines a callback signature as its only parameter. This callback signature accepts a single parameter named `withPrint` of type `WithPrint`. This `usePrint` function then creates an instance of the `WithPrint` class and invokes the callback.

We then call the `usePrint` function and pass in a function as the callback that uses the `WithPrintAndRun` class as the type for the `withRun` parameter. So we are using a derived class, `WithPrintAndRun` as a type within the callback, when we should be using only the base class of `WithPrint` as the type.

Note that there is a glaring flaw in our logic here that will cause a runtime exception. The `usePrint` function constructs a class of type `WithPrint`, which it uses to invoke the callback function named `fn`. This class instance will only have a `print()` function and will not have a `run()` function, but the callback function that we defined is assuming that the argument used will be of type `WithPrintAndRun`. This means that when we call `withRun.run()`, within our callback function, we will always generate a runtime exception.

With the `strictFunctionTypes` option turned on, this code will generate the following error:

```
Argument of type '(withRun: WithPrintAndRun) => void' is not assignable
to parameter of type '(withPrint: WithPrint) => void'.
```

Here, we can see that the compiler is correctly identifying the mismatch in our function signatures, even if we are using inheritance.

no compiler options

There are also a number of compiler options that are prefixed with the word `"no"`. These options are similar to the `"strict"` options, in that they further guard our code against things like unused parameters, implicit returns, and implicit any. In this section of the chapter, we will take a look at these compiler options and how they can detect potential errors within our code. These parameters are similar in nature to `"strict"` parameters, in that they can be turned on or off and can be introduced into a code base gradually.

Note that if the `"strict"` compiler option has been set to true, then all of these options will be true as well.

noImplicitAny

When a type has not been specified for a property, parameter, or function return type, the TypeScript compiler will automatically assume that it is of type any. In strict mode, this will generate an error. Consider the following code:

```
declare function testImplicityAny();
```

Here, we have a function declaration named `testImplicitAny`, which will generate the following error:

```
error TS7010: 'testImplicityAny', which lacks return-type annotation,
implicitly has an 'any' return type
```

What this error is indicating is that the `testImplicitAny` function has not specified a return type, and therefore will return a type of `any`. This can be fixed by specifying a return type, as follows:

```
declare function testImplicityAny(): void;
```

Here, we have specified that the `testImplicityAny` function returns a type of void, and the error is removed.

If a function parameter or a class property has not specified a type, then similar errors are generated. Consider the following code:

```
function testNoParamType(value) { }

class TestAny {
    id;
}
```

Here, we have a function named `testNoParamType` that has a single parameter named `value`. The `value` parameter does not have a type specified. We also have a class named `TestAny` that has a single property named `id` and also does not have a type specified. This code will produce two errors, as follows:

```
error TS7006: Parameter 'value' implicitly has an 'any' type.
error TS7008: Member 'id' implicitly has an 'any' type.
```

Here, we can see that the compiler is identifying both the `value` parameter and the `id` property as implicitly being of type any. These errors can be fixed by specifying a type in each of these cases, as follows:

```
function testNoParamType(value: string) { }

class TestAny {
    id : any;
}
```

Here, we have modified the function signature of the `testNoParamType` function to type the `value` parameter to be of type string. We have also updated the class property named `id` of the class `TestAny` to be of type any. These two changes will remove the compiler error.

Note that we have set the `id` property to be of type any. While the use of the `any` type is severely discouraged, it does show an interesting aspect of the `noImplicityAny` option. We are still able to use the type any, but we are not allowed to inadvertently leave off a type. If we do, we are implying the any type, which this compiler option will generate errors for.

noUnusedLocals and noUnusedParameters

The `noUnusedLocals` and `noUnusedParameter` compiler options are used to detect variables or parameters that are not used and are therefore superfluous. Consider the following code:

```
function testFunction(input: string): boolean {
    let test;
    return false;
}
```

Here, we have a function named `testFunction` that has a single parameter named `input` of type string and that returns a boolean type. Within the body of the function, we define a variable named `test` and then return the value false. Compiling this code will generate the following errors:

```
error TS6133: 'input' is declared but its value is never read.
error TS6133: 'test' is declared but its value is never read.
```

Here, we can see that the compiler is detecting that we have an unused parameter and that we have an unused local variable. The parameter named `input`, of type string, is defined in the function definition but is never actually used within the function body. In a similar manner, the variable named `test` is defined within the function body, but it is never assigned a value and is also never used.

Note that while this is a trivial example and we can clearly see that these local variables and parameters are never used in this code, it may not be so easily spotted in a larger code base or larger functions, so is best left for the compiler to find these unused variables for us.

noImplicitReturns

If a function has declared that it will return a value, then the `noImplicitReturns` compiler option will ensure that it does. Consider the following code:

```
function isLargeNumber(value: number): boolean {
    if (value > 1_000_000)
        return true;
}

console.log(`isLargeNumber(1) : ${isLargeNumber(1)}`);
```

Here, we have a function named `isLargeNumber` that has a single parameter named `value` of type number and returns a boolean value. Within the body of the function, we are checking if the value passed in as an argument is greater than one million. If it is, we return the boolean value `true`. Unfortunately, there is a logic flaw with this code. If the `value` argument is not greater than one million, we have forgotten to return anything at all.

To test this code, we have included a console log that calls the `isLargeNumber` function with a value of `1`. The output of this code is as follows:

```
isLargeNumber(1) : undefined
```

Here, we can see that the `isLargeNumber` function will return `undefined` for any value of the argument `value` that is less than one million. Running the compiler, however, with the `noImplicitReturns` option set to true will now generate an error:

```
error TS7030: Not all code paths return a value.
```

Here, we can see the compiler is detecting that the `isLargeNumber` function may not return a value if the false code path is taken. Our erroneous function can easily be fixed as follows:

```
function isLargeNumber(value: number): boolean {
    if (value > 1_000_000)
        return true;
    return false;
}
```

Here, we have added a statement at the end of the function that will return false. Our function will now always return a value, either true or false, and the error is not generated.

noFallthroughCasesInSwitch

The TypeScript compiler option named `noFallthroughCasesInSwitch` is used to trap a particular logic error within `switch` statements. Consider the following code:

```
enum SwitchEnum {
    ONE,
    TWO
}

function testEnumSwitch(value: SwitchEnum): string {
    let returnValue = "";
```

```
    switch (value) {
        case SwitchEnum.ONE:
            returnValue = "One";
        case SwitchEnum.TWO:
            returnValue = "Two";
    }
    return returnValue;
}
```

Here, we have defined an enum named `SwitchEnum` that has two values, `ONE` and `TWO`. We then have a function named `testEnumSwitch` that has a single parameter of type `SwitchEnum`. Within this function, we are setting the value of the `returnValue` variable to either `"One"` or `"Two"` based on the value of the incoming `value` parameter. We can use this function as follows:

```
console.log(`SwitchEnum.ONE = ${testEnumSwitch(SwitchEnum.ONE)}`);
console.log(`SwitchEnum.TWO = ${testEnumSwitch(SwitchEnum.TWO)}`);
```

Here, we are calling the `testEnumSwitch` function twice, and logging the return value to the console. The first call to the `testEnumSwitch` function uses the `SwitchEnum.ONE` value as an argument and the second call uses `SwitchEnum.TWO`. The output of this code is as follows:

```
SwitchEnum.ONE = Two
SwitchEnum.TWO = Two
```

This is obviously an error. What is happening here is that the switch statement for the value `SwitchEnum.ONE` is setting the value of the `returnValue` variable to `"One"`, but it is then falling through to the switch statement for the case `SwitchEnum.TWO`, which sets the `returnValue` variable to the value of `"Two"`. With the `noFallthroughCasesInSwitch` compiler option set to true, this code will generate the following error:

```
error TS7029: Fallthrough case in switch
case SwitchEnum.ONE:
```

Here, we can see that the compiler is correctly identifying the flaw in our logic. The compiler will also let us know where it found this error and what `switch` statement was to blame. We can fix this error as follows:

```
function testEnumSwitch(value: SwitchEnum): string {
    let returnValue = "";
    switch (value) {
        case SwitchEnum.ONE:
```

```
            returnValue = "One";
            break;
        case SwitchEnum.TWO:
            returnValue = "Two";
    }
    return returnValue;
}
```

Here, we have added a `break` statement in the case statement for `SwitchEnum.ONE`. This resolves our compile error. Running the code now will produce the following:

```
SwitchEnum.ONE = One
SwitchEnum.TWO = Two
```

Here, we can see that the flaw in the logic of our `switch` statement has been resolved. Again, these errors may be easy to spot in simple samples like this, but in larger code bases they can easily slip by. With this compiler options set, we can rely on the compiler to find these types of logic errors for us.

noImplicitThis

The `noImplicitThis` compiler option is used to detect logic errors when the `this` variable is accessed incorrectly. Consider the following code:

```
class NoImplicitThisClass {
    id: number = 1;
    printAfterWait() {
        let callback = function () {
            console.log(`this.id : ${this.id}`);
        }
        setTimeout(callback, 1000);
    }
}
```

Here, we have a class named `noImplicitThisClass` that has a single property named `id`, which is set to the numeric value of `1`. The class then defines a function named `printAfterWait`. Within this `printAfterWait` function, we are defining a variable named `callback`, which is a function that logs the value of the `this.id` property to the console. We then call `setTimeout`, in order to execute the `callback` function after a one-second delay. We can use this class as follows:

```
let classInstance = new NoImplicitThisClass();
classInstance.printAfterWait();
```

Here, we are creating an instance of the `NoImplicitThisClass` named `classInstance` and calling the `printAfterWait` function. The output of this code is as follows:

```
this.id : undefined
```

Here, we can see that the value of the `this.id` property when we log it to the console is, in fact, undefined. We should be able to access the `this` property within our class, and it should be referencing the class instance itself, but in this case it does not.

The reason for this is the scoping of the `this` property within JavaScript. When a function is invoked, it gets its own copy of the `this` property. So in our case, the function that is assigned to the local `callback` variable gets its own copy of `this`. As the `id` property is defined on the `this` instance of the outer class, the `id` property of `this` within the callback function will be undefined.

The `noImplicitThis` compiler option will trap these sorts of errors. With it turned on, this code will generate the following error:

```
error TS2683: 'this' implicitly has type 'any' because it does not have
a type annotation
```

Here, the compiler is correctly identifying that our reference to `this.id` within the callback function is not referencing the `this` property of the `NoImplicitThisClass` class. The `this` property within the callback function therefore has a type of `any`, hence the error.

There are two ways that we could correct this code. Firstly, we could pass the `this` property into the `callback` function as follows:

```
let callback = function (_this) {
    console.log(`this.id : ${_this.id}`);
}
setTimeout(callback, 1000, this);
```

Here, we have added a parameter named `_this` to the `callback` function and then passed the value of `this` into the `setTimeout` call.

Another way to correct this code is to use an arrow function, as follows:

```
let callback = () => {
    console.log(`this.id : ${this.id}`);
}
setTimeout(callback, 1000);
```

Here, we have removed the `function` keyword and used the arrow syntax instead. The output of both solutions is now correct, as follows:

```
this.id : 1
```

Here, we can see that both solutions now have access to the correct `this` property. Arguably, the cleanest of the two solutions is to use the arrow syntax. Arrow functions, when they are used, do not get their own copy of the `this` property; they use the outer value of the `this` property.

Summary

In this chapter, we took a look at the various strict compiler options available within the TypeScript compiler. We started with an example of nested configuration, where a `tsconfig.json` file can reference another one as a base. We then discussed four strict options that check our code base for possible errors, by detecting null variables, properties that have not been initialized, parameter types when using bind, call, or apply, and strict function types. We then went through the no compiler options, which help us to identify variables that could be of type any, unused variables, implicit returns, switch case errors, and incorrect scoping of this.

It is best practice for any new TypeScript project to leave the default strict option set to true. This will ensure that all of the options we have discussed will always be on and will help to trap a large portion of possible errors within our code. The only time that we should really be modifying these options is if we are slowly migrating an existing JavaScript source code base to TypeScript.

In the next chapter, we will explore a popular and powerful external library named RxJs and how we can use Observables to transform data.

9
Using Observables to Transform Data

One of the most intriguing, somewhat difficult, but fun parts of JavaScript programming is based around the handling of events. Whenever we issue an asynchronous call, we are essentially waiting for an event to occur that will trigger our callback processing. We generally have no control over how long an asynchronous call might take, especially if this call is to a server somewhere via an API call. In a similar vein, when running a JavaScript application on the browser, we are often waiting for a user to interact with our software and generate an event via a button click, or a keypress. Again, we have no control over how long the user will take to click that button, or indeed the order in which buttons could be clicked. Think of a small child playing with a keyboard. They may hit a group of buttons at the same time by using their fist to smash the keyboard, or they may hit a completely random combination of keys. It is up to our software to handle this seemingly random stream of events.

One of the most powerful and popular JavaScript libraries that specializes in event processing is the Reactive Extensions for JavaScript library, or simply RxJS. RxJS uses the **Gang of Four** (**GoF**) design pattern named the Observable pattern as the basis for registering interest in an event, as well as doing something when an event has been triggered. Along with these basic principles of the Observer design pattern, RxJS provides a plethora of utility functions to transform event data, as and when it comes in. At the heart of the RxJS library is the concept of Observables, which are source event streams. This has given rise to using the term Observables to describe RxJS source streams, and what can be done to them. So when someone says use Observables, they really mean use the RxJS library with its source streams and utility functions.

In this chapter, we will explore Observables, which really means the RxJS library, how it is used, and how it can help us when working with event-based, or asynchronous data.

In this chapter, we will cover the following topics:

- An introduction to Observables
- `pipe` and `map`
- Combining operators
- Time-based Observables
- Handling errors
- `mergeMap`, `concatMap`, and `forkJoin`
- Observable Subject

Introduction to Observables

To begin the discussion on Observables, let's first install the RxJS library as follows:

```
npm install rxjs
```

The RxJS library already includes the declaration files that are needed by TypeScript, so there is no need to install them separately using `@types`.

To generate an Observable, we can use the `of` function as follows:

```
import { of, Observable } from "rxjs";

const emitter : Observable<number> = of(1, 2, 3, 4);
```

Here, we start by importing the `of` function and the `Observable` type from the `rxjs` library. We are then defining a constant variable named `emitter`, which is using generic syntax to define its type as an Observable of type `number`. We then assign the result of the `of` function to the emitter variable, which will create an Observable from the numbers 1 through 4. We can now create an Observer as follows:

```
emitter.subscribe((value: number) => {
    console.log(`value: ${value}`)
});
```

Here, we are calling the `subscribe` function on the variable `emitter`. As the `emitter` variable is of type Observable, it automatically exposes the `subscribe` function in order to register Observers. The `subscribe` function takes a function as a parameter, and this function will be called once for each value that is emitted by the Observable. The output of this code is as follows:

```
value: 1
value: 2
value: 3
value: 4
```

Here, we can see that the function we passed into the `subscribe` function has indeed been called once for each value that is emitted by the Observable.

Note that only when calling the `subscribe` function on an Observable will the Observable start to emit values. Calling the `subscribe` function is known as subscribing to an Observable, and the values that are produced by the Observable are also known as the Observable stream.

The `of` function has a partner function named `from`, which uses an array as input into the Observable, as follows:

```
const emitArray : Observable<number> = from([1, 2, 3, 4]);

emitArray.subscribe((value: number) => {
    console.log(`arr: ${value}`);
});
```

Here, we have a variable named `emitArray`, which is of type `Observable<number>`, and is using the `from` function to create an Observable out of an array. Again, we call the `subscribe` function on the Observable named `emitArray`, and provide a function to be called for each value emitted by the Observable. The output of this code is as follows:

```
arr: 1
arr: 2
arr: 3
arr: 4
```

Here, we can see that the `from` function has created an Observable stream from the array input, and that the function we provided to the `subscribe` function is being called once for each value that is emitted by the Observable.

pipe and map

The RxJS library provides a `pipe` function to all Observables, similar to the `subscribe` function. This `pipe` function takes a variable number of functions as parameters and will execute these functions on each value that is emitted by the Observable. The functions that are provided to the `pipe` function are generally known as Observable operators, which all accept an Observable as input, and return an Observable as output. The `pipe` function emits an Observable stream.

This concept is best explained by reading some code, as in the following example:

```
import { map } from "rxjs/operators";
const emitter = of(1, 2, 3, 4);

const modulus = emitter.pipe(
    map((value: number) => {
        console.log(`received : ${value}`);
        return value % 2;
    }));

modulus.subscribe((value: number) => {
    console.log(`modulus : ${value}`);
});
```

Here, we start with an Observable named `emitter`, which will emit the values `1` through `4`. We then define a variable named `modulus` to hold the results of calling the `pipe` function on the `emitter` Observable. The only argument we are providing to the `pipe` function is a call to the `map` function, which is one of RxJS' operator functions.

The `map` function takes a single function as a parameter and will call this function for each value that is emitted by the Observable. The `map` function is used to map one value to another, or to modify the value emitted in some way. In this sample, we are returning the result of applying the modulus of two to each value.

Finally, we subscribe to the Observable and log its value to the console. The output of this code is as follows:

```
received : 1
modulus : 1
received : 2
modulus : 0
received : 3
modulus : 1
received : 4
modulus : 0
```

Here, we can see that `emitter` Observable emits the values one through four, and that the modulus Observable is emitting the modulus of `2` for each valued received.

Note that in these code samples, we have not explicitly set the type for our Observables. The `emitter` Observable and the `modulus` Observable could be explicitly typed as follows:

```
const emitter : Observable<number> = of(1, 2, 3, 4);

const modulus : Observable<number> = emitter.pipe(
    ...
);
```

Here, we have specified the type of both the `emitter` Observable, and the modulus Observable. This is not strictly necessary, as the TypeScript compiler will determine the correct return types when working with Observables. It does, however, explicitly state what we are expecting out of the Observable stream, and in larger, or more complex Observable transformations, explicitly setting the expected return type makes the code more readable and can prevent errors.

Combining operators

The `pipe` function allows us to combine multiple operator functions, which will each be applied to the values emitted by an Observable. Consider the following code:

```
const emitter = of(1, 2, 3, 4);

const stringMap = emitter.pipe(
    map((value: number) => { return value * 2 }),
    map((value: number) => { return `str_${value}` })
);

stringMap.subscribe((value: string) => {
    console.log(`stringMap emitted : ${value}`);
});
```

Here, we have an Observable named `emitter` that will emit the values `1` through `4`. We then have a variable named `stringMap` that holds the result of the `pipe` function on the `emitter` Observable. Within this `pipe` function, we have two `map` functions. The first `map` function will multiply the incoming numeric value by `2`, and the second `map` function will convert it to a string, with the prefix `str_`.

We then subscribe to the Observable and log each value to the console. The output of this code is as follows:

```
stringMap emitted : str_2
stringMap emitted : str_4
stringMap emitted : str_6
stringMap emitted : str_8
```

Here, we can see that both `map` functions have been applied to each value emitted by the `emitter` Observable. Note that we have actually modified the type of each value from type number to type string, in our second map function. This is why the type specified for the `value` parameter in our `subscribe` function is of type string.

Avoid swallowing values

When writing functions that are used by the RxJS operator functions, we need to be careful that we continue to return values, and do not swallow them unexpectedly. Consider the following code:

```
const emitOneTwo = of(1, 2);

const swallowedValues = emitOneTwo.pipe(
    map((value: number) => {
        console.log(`swallowing ${value}`);
        // not returning a value;
    })
);

swallowedValues.subscribe((value: void) => {
    console.log(`subscriber received value: ${value}`)
});
```

Here, we have an Observable named `emitOneTwo` that will emit the values `1` and `2`. We then pipe these values into the `swallowedValues` Observable. Note the `map` function that we have provided here. All it is doing is logging a message to the console. It is not returning a value.

As it is not returning a value, our `subscribe` function must define the type of the incoming `value` parameter as void. The output of this code is as follows:

```
swallowing 1
subscriber received value: undefined
swallowing 2
subscriber received value: undefined
```

Here, we can see the side effects of not returning a value from a `map` function. Firstly, the `map` function will be called for each value that is emitted from the `emitOneTwo` Observable, as usual. Secondly, the `subscribe` function will still be called for each emitted value. Unfortunately, the emitted value will be `undefined`, as we have not returned anything within our `map` function.

If we truly do want to not return a value, then a better option is to make this decision explicitly, as follows:

```
const swallowedValues: Observable<number | null> =
    emitOneTwo.pipe(
        map((value: number) => {
            if (value < 2) {
                return null;
            }
            return value;
        })
    );

swallowedValues.subscribe((value: number | null) => {
    console.log(`subscriber received value: ${value}`)
});
```

Here, we have modified our `map` function to always return a value. If the incoming value is less than `2`, we return `null`, otherwise we simply return the value passed in. Note that we have also explicitly typed the value of the `swallowedValues` Observable to be of type `Observable<number | null>`. This is an example of where we might want to explicitly type an Observable, to ensure that all subscribers know what types it is emitting.

We have also modified our `subscribe` function, to allow the `value` parameter to be either of type number, or of type null. The output of this version of code is as follows:

```
subscriber received value: null
subscriber received value: 2
```

Here, we can see that the `subscribe` function was called with the value of `null`, and then with the value of `2`. Making sure that we always return a value from a `map` function ensures that we are making conscious decisions about how the subscriber should react.

Note that the TypeScript compiler will give us some indication of possible errors when we fail to return values within an Observable.

Failing to return a value will automatically make the entire Observable return a type of `unknown`. It is therefore good practice to strongly type the parameters of `subscribe` functions, so that we know what value we are expecting an Observable to emit.

Time-based Observables

One of the main features of using RxJS Observables is that they are able to work seamlessly with asynchronous events. In other words, if you subscribe to an Observable, and the Observable only emits a value after 10 seconds, you will still be notified of this event when it happens.

This ability to handle asynchronous events with Observables is very handy and is used extensively in some libraries for making requests to an API. We do not know how long the server may take to respond, so we can therefore create an Observable out of a REST request and then simply subscribe to it. We will see examples of this technique when using the `HttpClient` class within Angular in a later chapter.

To illustrate time-based events, we can use another function made available by the RxJS library name `interval`, as follows:

```
const sourceInterval = interval(1000);

const fiveNumbers = sourceInterval.pipe(
    take(5),
    map((value: number) => {
        console.log(`map received : ${value}`)
        return `string_${value * 2}`;
    })
);

fiveNumbers.subscribe((value: string) => {
    console.log(`${new Date().toLocaleTimeString()} ${value}`);
});
```

Here, we have defined an Observable named `sourceInterval`, which is using the `interval` function with the argument `1000`. The `interval` function will emit an ever-increasing integer value, starting at `0`, every `1000` milliseconds, in other words, every second. We are then piping this Observable into two functions. The first function is named `take`, which is another function provided by the RxJS library. The `take` function has a single parameter, which is the number of Observable values to `"take"`. We have set this value to `5`, and therefore, the `map` function will be provided with five values, one per second. Once we have received all five values, the Observable stream will stop.

Our `map` function is logging the received value to the console, and then multiplying its value by two, and converting it to a string with the prefix of `string_`.

Finally, we are subscribing to the new Observable stream named `fiveNumbers`, and logging each string value received to the console. For the purposes of this exercise, we are also logging the current time to the console. The output of this code is as follows:

```
map received : 0
5:06:20 PM string_0
map received : 1
5:06:21 PM string_2
map received : 2
5:06:22 PM string_4
map received : 3
5:06:23 PM string_6
map received : 4
5:06:24 PM string_8
```

Here, we can see the results of our time-based Observable stream. The value `0` is received by our `map` function, converted to a string, and then received by our subscribe function, at `5:06:20`. One second later, at `5:06:21` the `map` function receives the value `1`, which is passed onto our subscribe function. Once we have received five values, that is, `0` to `4`, the Observable stream stops.

Observable errors

So what happens when something goes wrong within an Observable stream? Obviously, we will need a mechanism to catch these errors, so that we can do something sensible with them. As an example of a faulty Observable stream, consider the following code:

```
interface IValue {
    value: number
}
interface INestedObj {
    id?: IValue;
}
const objEmit : Observable<INestedObj> = of(
    { id: { value: 1 } },
    {},
    { id: { value: 2 } }
);
```

Here, we start with two interfaces, named `IValue` and `INestedObj`. The `IValue` interface has a property named `value` of type number, and the `INestedObj` has a single optional parameter named `id` of type `IValue`. We then create an Observable named `objEmit` that emits three values. The first value has the nested structure described by the `INestedObj` interface, and the second is a blank object. Now consider the following Observable stream:

```
const returnIdValue = objEmit.pipe(
    map((value: INestedObj) => {
        return value.id!.value;
    })
);

returnIdValue.subscribe((value: number) => {
    console.log(`received ${value} `)
});
```

Here, we have an Observable stream named `returnIdValue` that is returning the `id.value` property for the incoming stream value named `value`. Note how we have had to use the definite assignment operator in this return statement, that is, `value!.id!.value`, as the TypeScript compiler will normally generate errors if we attempt to access values that could be null or undefined. We then subscribe to this stream, and log the value received to the console. Our `map` function code will work perfectly with the first value in the Observable stream but will cause the program to crash on the second value of the stream. The output of this code is as follows:

```
received 1
TypeError: Cannot read property 'value' of undefined
```

Here, we can see that the first value of the stream was processed correctly, but the second value caused the code to crash. This is because the second value emitted in our Observable stream does not have an `id` property, it is undefined, and we cannot access the `value` property of an undefined value.

While this may seem a fairly contrived example, it can happen quite often when receiving data in a nested JSON structure. It is always good practice to use optional chaining when dealing with nested properties coming from an outside source, where you can't be sure whether the value has been set correctly.

In order to catch this error correctly, we can provide an error handling function when we subscribe to our Observable stream, as follows:

```
returnIdValue.subscribe(
    // called for each observable value
```

```
    (value: number | null) => {
        console.log(`received ${value} `);
    },
    // called if an error occurs
    (error: unknown) => {
        console.log(`error : ${error}`);
    },
    // complete function
    () => {
        console.log(`complete`);
    }
);
```

Here, we have provided three functions as arguments to the `subscribe` function. The first function will be called for every value that is emitted by the Observable stream. The second function is an error function that will be called if an error occurs. The final function will be called when the subscribe function completes. The output of this code is as follows:

```
received 1
error : TypeError: Cannot read property 'value' of undefined
```

Here, we can see the sequence of events that are occurring with our Observable stream. The first value emitted is handled by the first function provided to `subscribe` and is receiving the value `1`. The second value emitted by the Observable stream is causing the Observable stream to throw an error, and this error is being trapped by the error function provided to the `subscribe` function. Note, too, that the complete function is not called if an error occurs with the Observable stream.

catchError

The code we have explored thus far will trap any errors occurring in the Observable stream, but this error trap is actually outside of the stream itself. It is an error handler for the `subscribe` function. We can, however, use the `catchError` operator within an Observable stream itself, in order to trap errors earlier, but still maintain the integrity of the stream. Consider the following code:

```
const returnIdValue = objEmit.pipe(
    map((value: INestedObj) => {
        return value!.id!.value;
    }),
    catchError((error: unknown) => {
        console.log(`stream caught : ${error}`);
```

```
            return of(null);
        })
    );
```

Here, we have updated our earlier Observable stream and added a `catchError` operator following our `map` operator. This `catchError` operator is a function that has a single parameter named `error` of type unknown. Within this function, we are logging a message to the console, and then returning an Observable value of `null`.

This code now produces the following output:

```
received 1
stream caught : TypeError: Cannot read property 'value' of undefined
received null
complete
```

There are a few interesting things to note about this output. Firstly, our `catchError` function is being called with the error `"Cannot read property 'value' of undefined"`, which is the error that is generated by the `map` function, for the second value in the Observable stream. Secondly, our `catchError` is emitting the value null to the stream, which is then being picked up in our `subscribe` function. The third interesting thing is that the complete function of our subscribe code is now being executed.

What this means is that by trapping the error within the Observable stream itself and emitting a value from the stream, the stream is allowed to complete, which will call the complete function on the subscriber. As we saw in our earlier code sample, which did not have a `catchError` function, allowing the Observable stream to throw an error, will cause only the error function to be executed, and not the complete function. Trapping an error with `catchError`, and putting a value on the stream when this happens, will trigger the complete function.

Note that if an error occurs within an Observable stream, the stream will stop emitting values. This happens whether or not we emit a value from the `catchError` function or not. In our output, we can see that once the `catchError` occurs, and we emit a `null` value, the stream does not emit the value `{ id: { value: 2 } }`.

Observables returning Observables

Quite often, when working with Observables, we need to return a new Observable stream while already dealing with an Observable stream. In other words, for each Observable value in a stream, create a new Observable stream. While this might sound complicated, in the real world, it can happen fairly regularly.

Suppose that we are working with a REST API that tells us what products are sold within a sales catalog. This particular API call returns an array of product IDs that are associated with a particular catalog. For each of these product IDs, we then need to initiate a new REST API call to retrieve the information for this particular product, such as its name and description.

Let's assume that we are using an HTTP client that returns an Observable for each API call. This means that the first Observable stream will be the list of products within a catalogue, say, `[1,2,3]`. For each value in this stream, we then need to initiate a new API call to fetch the name and description for the product, which will also return an Observable. So while dealing with each value of the original Observable stream, we then need to deal with another Observable stream that contains the product details.

To illustrate this concept, consider the following interfaces:

```
interface IProductId {
    id: number;
}

interface IProductDescription {
    name: string;
    description: string;
}
```

Here, we have an interface named `IProductId` that has a single property named `id` of type number. We then have an interface name `IProductDescription` that has two properties named `name` and `description` of type string. These interfaces represent the information returned by two different Observables, as follows:

```
const productList = <Observable<IProductId>>from(
    [{ id: 1 }, { id: 2 }, { id: 3 }]
);

function getProductName(id: number):
    Observable<IProductDescription> {
    return of(
        {
            id: id,
            name: `Product_${id}`,
            description: `Description_${id}`
        }
    );
}
```

Here, we have an Observable named `productList` that is returning values of type `IProductId`. This Observable will emit the values `{id: 1}`, then `{id: 2}`, and then `{id: 3}`. We then have a function named `getProductName` that accepts a single parameter named `id` of type number, which is the `id` of the product to return details for. This function also returns an Observable, but this time of type `IProductDescription`. We can now use these two Observable streams as follows:

```
productList.pipe(
    map((value: IProductId) => {
        console.log(`Product id: ${value.id}`);
        return getProductName(value.id);
    })
).subscribe((value: Observable<IProductDescription>) => {
    value.subscribe((value: IProductDescription) => {
        console.log(`product name : ${value.name}`);
        console.log(`product desc : ${value.description}`);
    });
});
```

Here, we are piping the `productList` Observable stream into a `map` function. Within the `map` function, we are calling the `getProductName` function for each value in the stream. We are then subscribing to this new stream, which will emit an Observable of type `IProductDescription`. Within our `subscribe` function, we then need to call the `subscribe` function on the new Observable stream, which is generated by the `getProductName` function. This second `subscribe` function is logging the value of the `name` and `description` properties of the returned `value` argument, which is of type `IProductDescription`. The output of this code is as follows:

```
Product id: 1
product name : Product_1
product desc : Description_1
Product id: 2
product name : Product_2
product desc : Description_2
Product id: 3
product name : Product_3
product desc : Description_3
```

Here, we can see that each value of the initial Observable stream, which returns a value of type `IProductId`, is being processed by the second `subscribe` function, and logging the expected values to the console.

The use of inner Observables can be optimized, however, by using another operator function named `mergeMap`.

mergeMap

The `mergeMap` operator is used to return a single value from an Observable stream, so that we do not need to subscribe to the inner Observable. This behavior is best described through an example, as follows:

```
productList.pipe(
    mergeMap((value: IProductId):
        Observable<IProductDescription> => {
        console.log(`Product id: ${value?.id}`);
        return getProductName(value.id);
    })
).subscribe((value: IProductDescription) => {
    console.log(`product name : ${value.name}`)
    console.log(`product desc : ${value.description}`)
});
```

Here, we are piping the `productList` Observable stream into a `mergeMap` function. Note that the code for this `mergeMap` function is identical to our earlier `map` function, where it logs a message to the console, and then calls the `getProductName` function with the `id` property of the value passed in as an argument. The difference between this code sample and the previous code sample is in the `subscribe` function. Note that the type of the `value` parameter has changed from type `Observable<IProductDescription>` to just `IProductDescription`.

The `mergeMap` operator, therefore, has removed the need for us to `subscribe` to the Observable that was emitted from the `getProductName` function. The `mergeMap` function is used to flatten an inner Observable.

concatMap

When an Observable emits values, the values emitted may arrive at a subscriber out of order. Let's assume that an Observable takes three seconds to emit a single value, and then two seconds to emit another value, and finally one second to emit the third value. This can be simulated as follows:

```
const emitTreeTwoOne = of(3, 2, 1);

const delayedEmit = emitTreeTwoOne.pipe(
    mergeMap((value: number) => {
        console.log(
            `>> emit >>
            ${new Date().toLocaleTimeString()}
```

```
            value : ${value},
            delaying : ${1000 * value} ms`
        );
        return of(value).pipe(delay(1000 * value))
    })
);
```

Here, we have an Observable named `emitThreeTwoOne`, which will emit the values `3`, then `2`, then `1`. We then pipe this Observable into a `mergeMap` function that logs the emitted value to the console, along with a timestamp. This function then emits the value received after a delay, using the `delay` function from the RxJS library. We can then subscribe to the `delayedEmit` Observable as follows:

```
delayedEmit.subscribe(value => {
    console.log(`<< receive <<
        ${new Date().toLocaleTimeString()}
        received value : ${value}`);
});
```

The output of the subscription to this `delayedEmit` observable is as follows:

```
>> emit >>
        11:05:26 PM
        value : 3,
        delaying : 3000 ms
>> emit >>
        11:05:26 PM
        value : 2,
        delaying : 2000 ms
>> emit >>
        11:05:26 PM
        value : 1,
        delaying : 1000 ms
<< receive <<
        11:05:27 PM
        received value : 1
<< receive <<
        11:05:28 PM
        received value : 2
<< receive <<
        11:05:29 PM
        received value : 3
```

Here, we can see that the `emitThreeTwoOne` Observable is emitting the values `3`, `2`, and `1`, one after another. Note the timestamps that are logged from the `mergeMap` function. All three emit events occur one after the other, within the same second. Our `delayedEmit` Observable, however, is delaying the emission of these values by `3` seconds, then `2` seconds, then `1` second. Our subscription to the `delayedEmit` Observable shows, from the timestamps, that we will receive the value `1` first, as it was only delayed by one second, and then the value `2`, and then the value `3`. In other words, we are receiving values emitted by the Observable based on the time that they were emitted.

Note too that we have three emit events, followed by three receive events. This means that each value received by the `delayedEmit` Observable is processed immediately.

If it is important to process the emitted values in order, no matter when they arrived, we can use the `concatMap` function instead of the `mergeMap` function. The `concatMap` function will only subscribe to the next Observable when the previous one completes. To illustrate this behavior, let's update the `delayedEmit` Observable as follows:

```
const delayedEmit = emitTreeTwoOne.pipe(
    concatMap((value: number) => {
        console.log(
            `>> emit >>
             ${new Date().toLocaleTimeString()}
             value : ${value},
             delaying : ${1000 * value} ms`
        );
        return of(value).pipe(delay(1000 * value))
    })
);
```

Here, the only change to our code is to use the `concatMap` function within our `pipe` function, instead of the `mergeMap` function that we used earlier. Let's take a look at the output of our code now, as follows:

```
>> emit >>
        11:10:03 PM
        value : 3,
        delaying : 3000 ms
<< receive <<
        11:10:06 PM
        received value : 3
>> emit >>
```

```
        11:10:06 PM
        value : 2,
        delaying : 2000 ms
<< receive <<
        11:10:08 PM
        received value : 2
>> emit >>
        11:10:08 PM
        value : 1,
        delaying : 1000 ms
<< receive <<
        11:10:09 PM
        received value : 1
```

Here, we can see the effect of using `concatMap` instead of `mergeMap`. The `concatMap` function will only subscribe to the next Observable when the first completes. This is clearly visible from the output where the `emit` and `receive` log messages occur in pairs. The first Observable value from the `emitThreeTwoOne` stream is emitted, and is piped into our `delayedEmit` Observable. Within this `delayedEmit` pipe, we are using `concatMap` to delay the emission of the value by three seconds. Once the delay is up, the value is emitted and received by our subscription to this Observable. This sequence marks the completion of the first Observable, and then the `concatMap` function is ready to receive another value. It then emits the value `2` after a delay of two seconds, which is again received by the subscription, and finally emits the value `1` after one second.

The `concatMap` function is used to process Observable sequences in order. So if you are needing to wait until an Observable completes before wanting to process the next value, use `concatMap`.

forkJoin

When we have a number of Observable streams that need to all complete before we do something, we can use the `forkJoin` function. This situation occurs quite often when dealing with REST requests at the start of a page load, where the page may need to load data from a number of different REST APIs before displaying the page.

Let's assume that we are building a web page to show products available in a catalog. We may need one REST request that loads a store catalog based on the current date, and another REST request that loads sales specials for the day. Our page may want to display the sale items on the top of the page, or in a scrolling banner, as well as all store items in the main body of the page.

We may also need a further REST request to load information related to the customer, such as their logged-in status, or their country of origin. Only when all of these requests have completed can we display the page in full and allow the customer to add items to their shopping basket.

To illustrate this concept, let's examine a few Observable streams, as follows:

```
const onePerSecond = interval(1000);

const threeNumbers: Observable<number[]> = onePerSecond.pipe(
    take(3),
    map((value: number) => {
        console.log(`>> threeNumbers emitting : ${value}`);
        return value;
    }),
    toArray()
);

const twoStrings: Observable<string[]> = onePerSecond.pipe(
    take(2),
    map((value: number) => {
        console.log(`>> twoStrings emitting : value_${value}`);
        return `value_${value}`;
    }),
    toArray()
);
```

Here, we have an Observable stream named `onePerSecond`, which will emit an incrementing integer value starting at `0` every second. We then have an Observable named `threeNumbers` that pipes the `onePerSecond` stream into a series of RxJS functions. The first function within this pipe section is `take`, which has the effect of only processing, or taking the first *n* values of the initial stream, as we have seen before.

The next function in our pipe section is a `map` function, which is used to log the value of the Observable stream to the console, and then re-emit it. The final function in this pipe section is the `toArray` function, which combine all emitted values into an array and emit this array as a single Observable value. So our `threeNumbers` Observable stream will process the values `0`, `1`, and `2`, and emit the array `[0,1,2]`.

In the same manner, the `twoStrings` Observable will take only two values from the `onePerSecond` stream, use the `map` function to return a string, and emit an array. It will therefore process the values `0` and `1`, and return the array `["value_0", "value_1"]`.

What is interesting about this example is that the `twoStrings` Observable will emit its string array before the `threeNumbers` Observable completes. This gives us a timing issue if we are trying to wait for both Observables to complete before we continue processing. Let's see how the `forkJoin` function helps in this case as follows:

```
forkJoin(
    [threeNumbers,
        twoStrings]
).subscribe((values) => {
    console.log(`<< threeNumbers returned : ${values[0]}`);
    console.log(`<< twoStrings returned : ${values[1]}`);
});
```

Here, we are using the `forkJoin` function, which accepts an array of Observables that must all complete before it will execute the `subscribe` function. Our `subscribe` function in this example just logs the values that were emitted by each Observable to the console. The output of this code is as follows:

```
>> threeNumbers emitting : 0
>> twoStrings emitting : value_0
>> threeNumbers emitting : 1
>> twoStrings emitting : value_1
>> threeNumbers emitting : 2
<< threeNumbers returned : 0,1,2
<< twoStrings returned : value_0,value_1
```

Here, we can see both the `threeNumbers` Observable stream and the `twoStrings` Observable stream receiving values from the `onePerSecond` stream, and emitting three numbers and two strings. In both cases, the `toArray` function is combining the emitted values into an array. Interestingly, though, the `forkJoin` function will wait until both streams have emitted values before running the `subscribe` function, as can be seen in the last two logs to the console.

Note how we have referenced the array within our `subscribe` function for this `forkJoin`, as `values[0]` for the output of the `threeNumbers` stream, and `values[1]` for the output of the `twoStrings` stream. We can also use array syntax to destructure these array values into named variables as follows:

```
forkJoin(
    [threeNumbers,
        twoStrings]
).subscribe((
```

```
    [threeNumbersOutput, twoStringsOutput]
) => {
    console.log(`<< threeNumbersOutput: ${threeNumbersOutput}`);
    console.log(`<< twoStringsOutput: ${twoStringsOutput}`);
});
```

Here, we have replaced the `values` parameter of the `subscribe` function with a destructured and named array, `[threeNumbersOutput, twoStringsOutput]`. This means that we do not need to reference the values using array syntax, or `values[0]`; we can just reference the named parameter, which would be `threeNumbersOutput`, or `twoStringsOutput`. Using this named destructuring technique is preferable to using the array technique, for two reasons. Firstly, our code becomes more readable, as a named variable, such as `threeNumbersOutput`, is easier to understand than `value[0]`. Secondly, we are safeguarding our code from referencing array values that do not exist. Within our `subscribe` function, we could erroneously reference `values[3]`, which would return `undefined` as the `forkJoin` only returned two values.

As we have seen, `forkJoin` will wait for all Observable streams to complete before executing the `subscribe` function.

Observable Subject

Thus far, we have worked with Observables that emit values and seen how to subscribe to Observable streams. The Observable itself is responsible for emitting values, and the subscribers react to values being emitted. When an Observable stream is complete, all subscribers complete their processing, and their execution stops. In essence, subscribers are alive as long as the Observable stream is emitting values.

So what if we want to keep an Observable stream open and register one or more subscribers that will wait around until a new value is emitted? Think in terms of an event bus, where multiple subscribers register their interest in a topic on an event bus, and then react as and when an event is raised that they are interested in. RxJS provides the `Subject` class for this express purpose.

A Subject maintains a list of listeners that have registered their interest. A Subject is also an Observable stream, and therefore listeners can subscribe to the stream and use the same syntax and functions that are used for normal Observable streams. What makes a Subject interesting is that has the ability to multicast, which means that it allows multiple subscribers to the same stream and will notify all interested subscribers when an event happens.

To illustrate this behavior, let's build a minimal event bus using a `Subject`. To start with, let's define an interface for a broadcast event, as follows:

```
enum EventKeys {
    ALL = "all-events",
    SINGLE = "single-event"
}

export interface IBroadcastEvent {
    key: EventKeys;
    data: string;
}
```

Here, we have defined an enum named `EventKeys` that defines two string values of `"all-events"` and `"single-event"`. We then define an interface named `IBroadcastEvent` that has two properties, named `key` of type `EventKeys` and `data` of type string. The purpose of this `EventKeys` enum and the `IBroadcastEvent` interface is to define the shape of any event that will be broadcast on our event bus. Each event of type `IBroadcastEvent` will contain a `key` value that is limited to our enum set and a `data` property of type string.

Let's now take a look at an implementation of an event bus, as follows:

```
export class BroadcastService {
    private _eventBus = new Subject<IBroadcastEvent>();

    on(key: EventKeys): Observable<string> {
        return this._eventBus.asObservable().pipe(
            filter(
                event => event.key === key ||
                    event.key === EventKeys.ALL),
            map(event => event.data));
    }

    broadcast(key: EventKeys, data: string) {
        this._eventBus.next({ key, data });
    }
}
```

Here, we have a class named `BroadcastService` that has a single private property named `_eventBus` of type `Subject<IBroadcastEvent>`. We then have an `on` function that has a single parameter named `key` of type `EventKeys`. This `on` function returns an Observable stream of type string.

The purpose of the `on` function is to allow subscribers to register for a particular event. In other words, the subscribers are notified on the emission of an event with a specific key.

Note the implementation of this `on` function. Within it, we are returning the `_eventBus` Subject as an Observable, using the `asObservable` function from RxJS. This function, from the documentation, "Creates a new Observable with this Subject as the source." In other words, it exposes the Subject named `_eventBus` as a new Observable stream to any caller of this function. Note too that we are piping the `_eventBus` Observable into a `filter` and then a `map` function. This `filter` function will check to see if the incoming argument named `key` matches the key of the event being raised. If it matches, or if the event key is `EventKeys.ALL`, then notify the registered Observer.

This `on` function, therefore, will notify all registered Observers of events that have the key `EventKeys.ALL`, but will only notify specific Observers if the key is `EventKey.SINGLE`.

The `BroadcastService` class also has a function named `broadcast`, which has two parameters, named `key` of type `EventKeys` and `data` of type string. This function can be called to broadcast an event to any registered listeners.

Let's now look at the implementation of a listener on this event bus, as follows:

```
class Listener {
    private eventSubscription: Subscription;
    constructor(
        broadCastService: BroadcastService,
        eventKey: EventKeys,
        private listenerName: string
    ) {
        _.bindAll(this, "reactToEvent");
        this.eventSubscription =
            broadCastService.on(eventKey)
                .subscribe(this.reactToEvent);
    }
    private reactToEvent(event: string) {
        console.log(`Listener [${this.listenerName}]
            received event : ${event}`);
    }
    public unregister() {
        this.eventSubscription.unsubscribe();
    }
}
```

Here, we have defined a class named `Listener` that has a single private property named `eventSubscription` of type `Subscription`. This class has a constructor that has three parameters. The first parameter is named `broadCastService` of type `BroadcastService`, which is the instance of the `BroadcastService` class acting as the event bus. The second parameter is named `eventKey` of type `EventKeys`, and it is used to indicate which events this listener is interested in. The third parameter is named `listenerName`, which is a private property to differentiate the `Listener` class instance from other `Listener` class instances.

The implementation of the constructor starts with a call to the underscore function named `bindAll`, which binds the value of `this` within a class function to the class instance itself. Remember that when an instance of the class `BroadcastService` emits an event, it will emit the event with its own scoped value of `this`. In other words, when an event is emitted by the Broadcast service, the `this` variable will be scoped to be `this` as seen by the `BroadcastService` class instance, and not `this` as seen by the instance of the `Listener` class instance. Calling the `bindAll` function with `this` as seen by the `Listener` class instance, and the name of the function to bind to, `"reactToEvent"`, will ensure that when the `reactToEvent` is called, `this` will refer to `this` as scoped to the `Listener` class instance.

Following our `bindAll` function call, the constructor then sets the class property named `eventSubscription` to the Observable returned by the `on` function of the `BroadcastService`. Note how we are calling the `on` function with the value of the `eventKey` parameter, in order to register for specific events we are interested in. We then subscribe to the Observable stream returned by the `on` function and call the `reactToEvent` function defined in the `Listener` class.

The `reactToEvent` function logs a message to the console, with the value of the event that was emitted. Note that we have also defined an `unregister` function on the `Listener` class that will unsubscribe to the Observable stream.

With both the `BroadcastService` class and the `Listener` class defined, we can now use these classes as follows:

```
const broadCastService = new BroadcastService();

const listenOne = new Listener(
    broadCastService,
    EventKeys.ALL, "first");
const listenTwo = new Listener(
    broadCastService,
    EventKeys.SINGLE, "second");
```

Here, we start by creating an instance of the `BroadcastService` class named `broadCastService`. We then create two instances of the `Listener` class named `listenOne` and `listenTwo`. The `listenOne` class instance is interested in all events that are broadcast on the event bus, and the `listenTwo` class instance is interested in the event `EventKeys.SINGLE`. Note that both listeners will receive any event that is broadcast with the `EventKeys.ALL` key. We can now start to broadcast events on this event bus, and see what happens as follows:

```
broadCastService.broadcast(
    EventKeys.ALL, "ALL event broadcast");
broadCastService.broadcast(
    EventKeys.SINGLE, "single event broadcast");
broadCastService.broadcast(
    EventKeys.ALL, "Another ALL event broadcast");
```

Here, we start by using the instance of the `BroadcastService` named `broadCastService` to call the `broadcast` function three times. The first call broadcasts an event with the key `EventKeys.ALL`, with the value `"ALL event broadcast"`. This means that all listeners will receive the event. The second call to the `broadcast` function uses the key `EventKeys.SINGLE` to only broadcast an event to listeners that have registered with this particular key. The third call to the `broadcast` function again uses the `EventKeys.ALL` key, but this time with a different message. The output of this code is as follows:

```
Listener [first] received event : ALL event broadcast
Listener [second] received event : ALL event broadcast
Listener [second] received event : single event broadcast
Listener [first] received event : Another ALL event broadcast
Listener [second] received event : Another ALL event broadcast
```

Here, we can see which events are picked up by our listeners. The first event that we broadcast was using the key `EventKeys.ALL`, and as such is picked up by the listener named `"first"` and by the listener named `"second"`. The second event was broadcast was using the key `EventKeys.SINGLE`, and as such is only picked up by the listener named `"second"`, which registered for this event specifically. The third event that was broadcast was also using the key `EventKeys.ALL`, and as such is picked up by both listeners.

Let's now unregister one event listener as follows:

```
listenOne.unregister();
broadCastService.broadcast(
    EventKeys.ALL, "final ALL event broadcast");
```

Here, we have called the `unregister` function on the class instance named `listenOne`, which was listening to events with the key `EventKeys.ALL`. We then broadcast an event with the key `EventKeys.ALL`. The output of this code is as follows:

```
Listener [second] received event : final ALL event broadcast
```

Here, we can see that because we have unregistered the first event listener, the only listener that is still available to receive an event is the listener named `"second"`.

Using an event bus within our code is a powerful design pattern that can be used when completely unrelated components need to be notified of a particular event. Think of the case where a user logs out of a web site. They may have an open shopping cart and may also be viewing content that is only available to registered users. We can build an event bus to notify all interested components of an event such as the user logging out, and each component can then take the necessary steps to modify their rendered content accordingly. An event bus allows us to utilize a design pattern known as the Domain Events Pattern, where multiple components can react to a specific domain event. This design pattern strengthens the quality and extensibility of our code base, by allowing each component to focus on their particular area of interest, and also reacting to external events that occur across the entire domain.

Summary

In this chapter, we have explored parts of the RxJS library and the fundamental concept of Observables that it provides. We have seen how we can create Observables easily using the `of` and `from` functions, as well as how we can transform values emitted from an Observable stream using the `pipe`, `map`, `take`, `toArray`, `mergeMap`, and `contactMap` functions. We also explored error handling through the error function available on a subscription, and how to handle errors within an Observable stream using the `catchError` function. We then explored how to wait for multiple Observable streams to complete with the `forkJoin` function and rounded out the chapter with a discussion on Subjects, and an implementation of a simple event bus.

We have only just scratched the surface of the functionality available in the RxJS library, however, but have covered some of the main topics and concepts. Feel free to head over to the RxJS website to view the wealth of available operators and functions that cover just about anything you may want to do to transform a stream of data.

In the next chapter, we will focus on unit testing by exploring the powerful Jest unit testing framework.

10

Test-Driven Development

In the modern world of JavaScript development, there are many different frontend frameworks that we can use to write applications, from older frameworks such as Backbone.js, to newer ones such as Angular, React, and Vue. These frameworks will generally use either the **Model View Controller** (**MVC**) design pattern, or some variation of it, such as the **Model View Presenter** (**MVP**), or **Model View View Model** (**MVVM**). When discussing this group of patterns together, they are described by some as **Model View Whatever** (**MVW**), or simply **MV***.

Some of the benefits of this MV* style of writing applications include modularity and separation of concerns, but one of the biggest advantages is the ability to write testable JavaScript. Using MV* allows us to unit test the Models we use, the Views we use, and the Controllers we use. We can write tests for individual classes, and then extend these tests to cover groups of classes. We can also test our rendering functions and ensure that the DOM elements on a web page are displaying correctly. We can simulate button clicks, drop-down selects, form input, and even animations. These tests can then be extended into page transitions, including simulating login pages and access rights. By building a large set of tests for our applications, we gain confidence that our code works as expected, and allows us to refactor our code at any time.

Refactoring code refers to the ability to modify the underlying implementation of a block of code, or a set of functionalities, without fear that we will introduce bugs unwittingly. This means that if we have a set of tests, then we are free to rewrite any part of the underlying code, as long as the tests continue to pass. There is an old saying that without tests, we are not refactoring code, we are just randomly changing things. In a large body of code, even a one-line change could have unwanted side effects that are not easily found without unit tests in place.

In this chapter, we will look at test-driven development in relation to TypeScript. We will focus on the popular Jest testing framework for unit testing, exploring how tests are structured using Jest. We will then discuss how to write asynchronous tests, or tests that deal with asynchronous code, and how to write tests that actually modify the DOM. Finally, we will discuss end-to-end testing, or testing against a running website, using Protractor and Selenium.

The testing paradigm

Test-driven development (**TDD**) is really a way of thinking, or a paradigm if you like, that should be baked into any standard development process. This paradigm starts with tests and drives the momentum of a piece of production code through these tests. TDD means asking the question "How do I know that I have solved the problem?", instead of just "How do I solve the problem?". This is an important idea to grasp. We write code in order to solve a problem, but we should also be able to prove that we have solved the problem through the use of automated tests.

The basic steps of a test-driven approach are as follows:

- Write a test that fails.
- Run the test to ensure that it fails.
- Write code to make the test pass.
- Run the test to see that it now passes.
- Run all tests to see that the new code does not break other tests.
- Repeat.

Using TDD is really a mindset. Some developers follow this approach and write tests first, while others write their code and their tests afterward. Then, there are some that don't write tests at all. If you fall into the last category, then hopefully the techniques presented in this chapter will help you to get started in the right direction.

There are so many excuses out there for not writing unit tests. Things such as "the test framework was not in our original quote," or "it will add 20% to the development time," or "the tests are outdated, so we don't run them anymore." The truth is, though, that in this day and age, we cannot afford not to write unit tests. Applications grow in size and complexity, and requirements change over time. An application that has a good suite of tests can be modified far more quickly, and will be more resilient to future requirement changes than one that does not have tests. This is when the real cost savings of unit testing becomes apparent. By writing unit tests for your application, you are future-proofing it, and ensuring that any change to the code base over time does not break existing functionality.

We also want to write applications that stand the test of time. The code we write now could be in a production environment for years to come, which means that sometimes, you will need to make enhancements or bug fixes to code that was written years ago. Tests can also help to explain a piece of code, as well as the limitations put on allowed ranges for variables, or under exactly which circumstances a particular piece of code will execute.

Ideally, our tests should run within a Continuous Integration, or CI, environment. This means that as soon as we check a piece of code into source control, a CI server will extract the latest version, build it from scratch, and run all of the tests for the code in an automated fashion. Having a suite of tests will not really help unless they are run regularly, and having a CI server run automated tests on every check-in will ensure that our checked-in code always passes all of our tests.

Unit, integration, and acceptance tests

Automated tests can be broken up into three general areas, or types of tests – unit tests, integration tests, and acceptance tests. We can also describe these tests as either black-box or white-box tests. White-box tests are where the internal logic or structure of the code are known to the tester. Black-box tests, on the other hand, are tests where the internal design or logic are not known to the tester.

Unit tests

A unit test is typically a white-box test where all of the external interfaces to a block of code are mocked or stubbed out. If we are testing some code that does an asynchronous call to load a block of JSON, for example, unit testing of this code would require mocking out the returned JSON. This technique ensures that the object under test is always given a known set of data. When new requirements come along, this known set of data can grow and expand, of course. Objects under test should be designed to interact with interfaces, so that those interfaces can be easily mocked or stubbed in a unit test scenario.

Integration tests

Integration tests are another form of white-box tests that allow the object under test to run in an environment close to how it would look in a real deployment. In our earlier example, where some code needs to do an asynchronous call to load a block of JSON data, an integration test would need to actually call the REST services that generated the JSON. If this REST service relied upon data from a database, then the integration test would need data in the database that matched the integration test scenario. If we were to describe a unit test as having a boundary around the object under test, then an integration test is an expansion of this boundary to include dependent objects or services.

Building automated integration tests for your applications will improve the quality of your product immensely. Consider the case for the scenario we have been discussing – where a block of code calls a REST service for some JSON data. Someone could easily change the structure of the JSON data that the REST service returns. Our unit tests would still pass, as they are not actually calling the REST service, but our application would be broken because the returned JSON is not what we are expecting.

Without integration tests, these types of errors will only be picked up in the later stages of manual testing, or possibly only in production. The later a failure in our system is identified, the more expensive it is to fix.

Acceptance tests

Acceptance tests are black-box tests and are generally scenario-based. They may incorporate multiple user screens or user interactions in order to pass. Although these tests are generally carried out by a testing team, they can be automated fairly easily with the wealth of modern testing tools that are readily available.

Automating acceptance testing is really the holy grail of the testing tree. Humans can easily make mistakes and using a testing team to repeatedly run hundreds of acceptance tests is not always reliable, is expensive, and takes a long time. Computers are very good at doing repetitive tasks over and over again, and they can also do them faster than humans. Having an automated acceptance test suite the runs overnight, for example, can give the development team early access to any test failures, which will reduce costs immensely. Having a full suite of automated acceptance tests also proves that the application works, and that new features have not inadvertently broken older ones.

Unit testing frameworks

Over the years, there have been many unit testing frameworks that have been built for JavaScript. Some of the oldest, and therefore most mature, frameworks include Jasmine (`https://jasmine.github.io/`), QUnit (`https://qunitjs.com/`) and Mocha (`https://mochajs.org/`). There have also been attempts to write unit testing frameworks in TypeScript, including MaxUnit or tsUnit, but these frameworks never really took off.

When using a unit testing framework, one of the most important things to consider is the range of testing features that it has. Does it have the ability to raise errors, for example? Can it mock out dependencies easily, such as an asynchronous call to a REST server, or returning rows from a database? Can it allow for special testing code to replace a function call completely? Can it test throwing exceptions, and can it test rendered DOM elements?

The ease with which TypeScript can integrate with JavaScript libraries means that we can use a fully featured JavaScript unit testing framework as if it were written in TypeScript. Most testing frameworks have a similar suite of features, including a watch mode that will detect changes to test files and automatically re-run tests. One of the popular testing frameworks currently available is Jest, which was written by the Facebook team, and integrates with Babel, TypeScript, Node, React, Angular, and Vue. In this chapter, we will use Jest to explore the capabilities of a fully featured JavaScript testing framework.

Jest

Jest is a simple-to-configure and powerful JavaScript unit testing framework that is built on top of the popular Jasmine framework. Jasmine has been around for a very long time, and is a mature, fully featured, and widely used testing framework. Jest enhances Jasmine by making it easier to configure, as well as providing a wealth of extra features. Jest tests can also be run concurrently, which significantly speeds up the length of time a test suite will take to run. Jest is available through npm, and will therefore require an npm environment, which can be created as follows:

```
npm init
```

Here, we have initialized an npm project and can now install the required Jest packages, as follows:

```
npm install jest --save-dev
```

With Jest installed, we can either run it using the command `npx jest`, or we can modify our `package.json` file to specify that Jest will be used when we run `npm test`. Let's update our `package.json` file as follows:

```
{
  "name": "src",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {
    "jest": "^26.5.3"
  }
}
```

Here, we have updated the `test` sub property of the `scripts` property to specify `jest` as our main testing framework. We can now run our unit tests with npm as follows:

```
npm test
```

Here, we are starting the `jest` test runner using npm. Unfortunately, we do not have any tests written as yet, and we will therefore find an error written to the console as follows:

```
No tests found, exiting with code 1
```

Jest, by default, will look for files named `*.spec.js` for tests to run. The word spec is short for the word specification, and these spec files therefore contain all of our test specifications. Generally, our test spec files will sit alongside our normal components, in the same directory, so that a component named `search.js` will have a corresponding test spec file named `search.spec.js`.

So, let's go ahead and write some test specifications.

ts-jest

Jest is a JavaScript test framework, and as such, will look for JavaScript tests to run within our project. We can either run the TypeScript compiler to generate JavaScript files, or we can use a framework such as ts-jest. ts-jest is a TypeScript to Jest bridge, and will take care of the compilation step, and integration with Jest for us. In fact, ts-jest will compile and execute our TypeScript tests without even generating JavaScript files. This means that we can write unit tests in TypeScript, and ts-jest will run them within Jest seamlessly.

We can install and configure ts-jest as follows:

```
npm install ts-jest --save-dev
npx ts-jest config:init
npm install typescript --save-dev
```

Here, we have installed the ts-jest library using npm as usual. We then run ts-jest with npx and specify that it should create a configuration file using the command-line option `config:init`. This is similar to configuring an npm project using `npm init`, or configuring a TypeScript project with `tsc --init`. Note that ts-jest relies on a local version of TypeScript to be installed within our project, so we also need to install the TypeScript npm package in our project directory.

With all of our configuration in place, let's write a simple test in a file named `hello_jest.spec.ts`, as follows:

```
test('should be false', () => {
    expect(true).toBeFalsy();
});
```

Here, we start a test by calling the function named `test`, which accepts two arguments. The first argument is a string value and is the name of the test, which in this case is `'should be false'`. The second argument is a function that is the test itself. Within this function, we are using the Jest `expect` function, which returns a value that can be assessed by what is known as a matcher. The matcher in this case is `toBeFalsy`. So, this test is expecting that the value `true` should be false, which is clearly not the case, so this test should fail. Running `npm test` with this file in place will run Jest and show the results of our test run, which can be seen in the following screenshot:

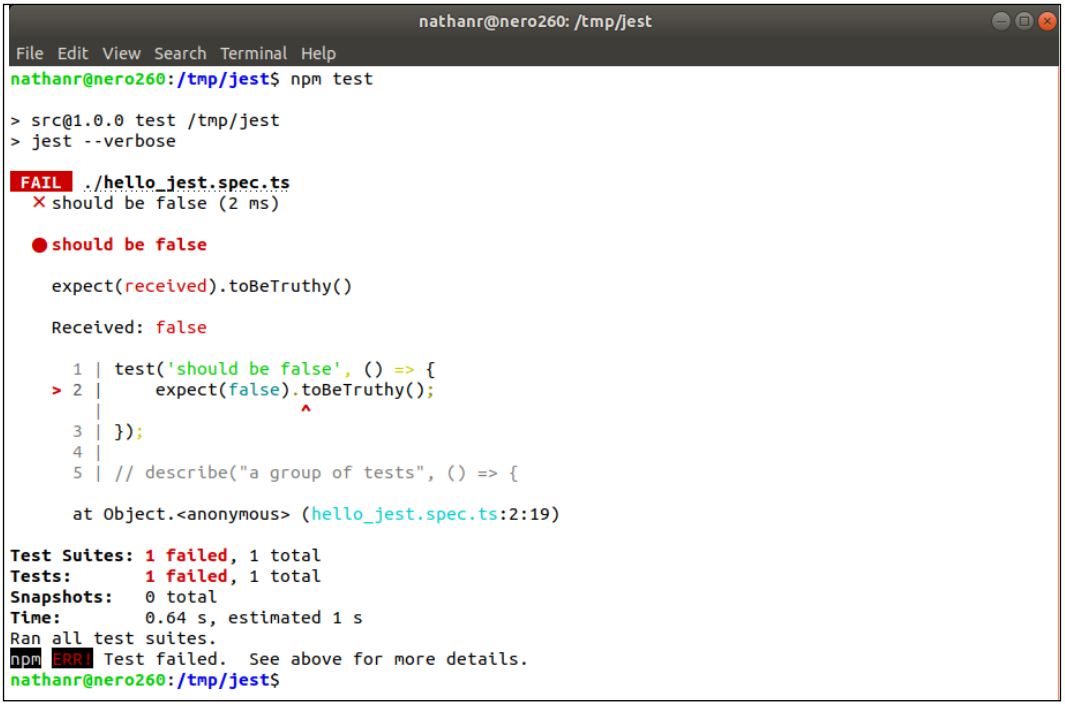


Figure 10.1: Console output showing Jest running a failed test

Here, we can see the results of our first test run using ts-jest. The first thing to note is that the entire test run has failed, and that Jest is letting us know what test failed, and exactly where it failed. We can see that the failing test is in a file named `hello_jest_spec.ts`, that the test itself was named `'should be false'`, and that the `expect` function received a value of `true`, where it was expecting a value of `false`. We can also see a code snippet of the code that is to blame.

While this may seem a lot of information for a single failed test, bear in mind that a mature application may have hundreds, if not thousands, of unit tests in a single test run. Having information about what test failed, what file it was in, what was expected, and the exact code that is to blame, is an invaluable time saver when trying to fix failing tests.

Following on from the test-driven development paradigm, we should now write some code that causes this test to pass, as follows:

```
test('should be false', () => {
    expect(false).toBeFalsy();
});
```

Here, we have changed the value that is passed into our `expect` function from `true` to `false`. So, reading the test code like we would read a sentence, this test is stating `"we expect the value of false to be false"`, which it is, and therefore our test run will pass, as shown in the following screenshot:

Figure 10.2: Command-line output showing a successful Jest test run

Here, we can see the results of a successful test run. Jest will show us what tests were run by naming each test with a green tick next to it. We also have information around how many test suites were run, how many tests were run, and how long the entire test run took.

Watch mode

Jest, like other test frameworks, can also be run in watch mode. This means that it will watch all files within a project, and automatically re-run any tests if any files change. Let's update our `package.json` file as follows:

```
{
    ...
```

```
    "scripts": {
        "test": "jest --watchAll --verbose"
    },
    ...
}
```

Here, we have added the `--watchAll` command-line argument to the `test` property. This means that Jest will be executed with the `--watchAll` argument and will execute in watch mode. Note, too, that we have also added the `--verbose` command-line option. This option will output the name of each test that we have run to the console.

Running `npm test` with this in place shows Jest running in watch mode, as shown in the following screenshot:

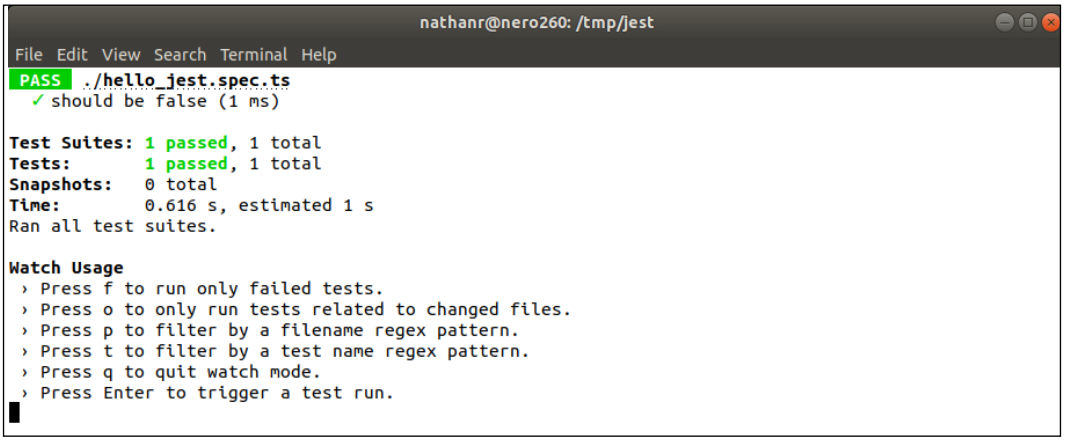


Figure 10.3: Command-line output showing Jest running in watch mode

Here, Jest is running in watch mode, which will, by default, run all tests as soon as any of our source files change. Note that there are also a number of keyboard shortcuts available to run a subset of tests. Hitting the *f* key will run only failed tests, and hitting the *o* key will run tests related to changed files only. The *p* and *t* options allow for filtering tests based on regex patterns.

Grouping tests

Within a test specification file, we may want to group our tests into logical sets. Jest uses the function `describe` for this purpose, as seen in the following tests:

```
describe("a group of tests", () => {
    test("first test", () => {
        expect("string value").toEqual("string value")
    })
```

```
    it("second test", () => {
        expect("abc").not.toEqual("def");
    })
});
```

Here, we start our tests with the `describe` function. The `describe` function is very similar to the `test` function, and also has two parameters. The first parameter is a string value for the name of the group of tests, or the test suite, and the second parameter is a function containing the set of tests. Within this `describe` function, we have a test named `"first test"`, and another test named `"second test"`. Note, however, that the second test is using the function `it`, and not the function `test`. These two function names are synonymous, as the `it` function is the default Jasmine function for describing tests, and the `test` function is Jest's default.

The output of this test run can be seen in the following screenshot:

Figure 10.4: Command-line output showing Jest running a group of tests

Here, we see that Jest has output the name of our group of tests, which was `"a group of tests"`, and has grouped the `"first test"` test, and the `"second test"` test under it. Having a set of tests run as a group has a few advantages, as we will see next.

Forcing and skipping tests

When working with an application that has a number of tests written for it, we often want to run only a specific test, or a specific group of tests. This act is known as forcing tests, as we are forcing the entire test suite to only run specific tests. This can be done in two ways, as follows:

```
describe("a group of tests", () => {
    test.only("first test", () => {
        expect("string value").toEqual("string value")
    })
```

```
    fit("second test", () => {
        expect("abc").not.toEqual("def");
    })
});
```

Here, we are using the `test.only` function instead of just the `test` function to force the test named `"first test"` to be run. We are also using the `fit` function instead of just the `it` function to force the test named `"second test"` to be run. The output of this test run is as follows:

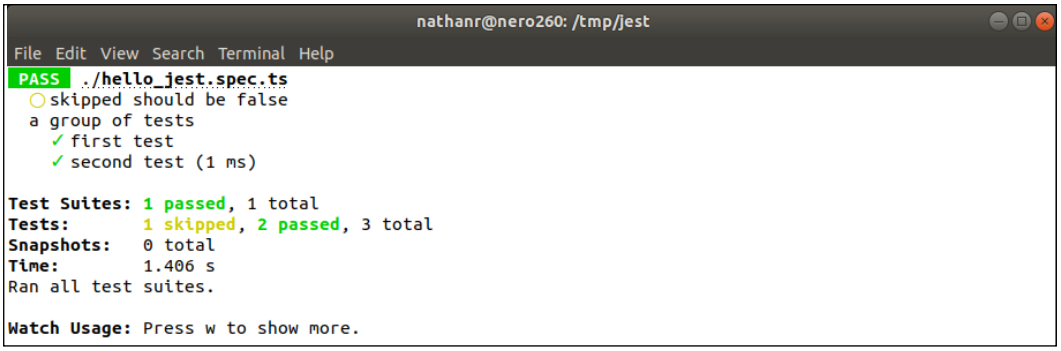


Figure 10.5: Command-line output of Jest with forced tests

Here, we can see that the test run is only including the tests named `"first test"` and `"second test"`, which were the tests that we forced. Note too, that we now have skipped tests, and that Jest is reporting the number of skipped tests alongside the number of actual tests that were run.

Using the `it` function (Jasmine default) instead of the `test` function (Jest default) for tests means that we only have to prefix an `it` test with the letter `f` in order to force it, that is, `fit` versus `it`. This is quite a time saver, instead of having to type `test.only` to force a test instead of the normal `test`. For this reason, we will adopt the Jasmine style `it` test naming convention for creating tests through the remainder of this chapter.

Groups of tests can also be forced by prefixing the letter `f` to the `describe` function, as follows:

```
fdescribe("a group of tests", () => {
    test("first test", () => {
        expect("string value").toEqual("string value")
    })
    it("second test", () => {
        expect("abc").not.toEqual("def");
    })
});
```

Here, we have used the `fdescribe` function for our test group named `"a group of tests"`, instead of the standard `describe` function. Using `fdescribe` will force the entire group of tests to be included in the test run.

The opposite of forcing tests is to skip tests. To skip a test, we can prefix the test with the letter `x`, so it becomes `xit`, as follows:

```
fdescribe("a group of tests", () => {
    test("first test", () => {
        expect("string value").toEqual("string value")
    })
    xit("second test", () => {
        expect("abc").not.toEqual("def");
    })
});
```

Here, we are forcing the group of tests named `"a group of tests"` to be run using `fdescribe` and are also skipping the test named `"second test"` by using `xit` instead of `it`. This test run can be seen in the following screenshot:

Figure 10.6: Command-line output showing Jest skipping a test

Here, we can see the combination of both forcing a group of tests using `fdescribe`, and skipping a test using `xit`. The test group named `"a group of tests"` is being forced, and therefore only the tests `"first test"` and `"second test"` will be run. We have, however, skipped the test named `"second test"` by using `xit` instead of just `it`. This means that the only test that is within the test group, and is not being skipped, is the test `"first test"`.

Note that skipping tests is not a good idea at all. The point of writing unit tests is to ensure that our application behaves as expected, and tests are a way of describing what that behavior should be.

If a test has been written, then it should be respected, and any code changes should ensure that the test still passes, or that the test is updated to accommodate the new behavior. Skipping tests just because they are failing is the antithesis of test-driven development and should be avoided at all costs.

Matchers

Jest uses what are known as matchers to match the expected values in a test to the received values. Let's have a quick look at some of these matchers, as follows:

```
it("should match with toBe", () => {
    expect(1).toBe(2);
});
```

Here, we are using the `toBe` matcher to test whether the value `1` is the same as the value `2`. Obviously, this test will fail with the following message:

```
expect(received).toBe(expected) // Object.is equality
Expected: 2
Received: 1
```

Here, we can see that Jest is expecting the value `2`, but it received the value `1`. The interesting thing about this message is that the `toBe` matcher is using `Object.is` equality. This means that the following test will pass:

```
it("should match with toBe using assignment", () => {
    let objA = { id: 1 };
    let objB = objA;
    expect(objA).toBe(objB);
});
```

Here, we are creating an instance of an object named `objA`, and then assigning its value to another variable named `objB`. This means that both variables `objA` and `objB` are pointing to the same object in memory. We may attempt to use the `toBe` matcher on two different objects, as follows:

```
it("should match with toBe", () => {
    let objA = { id: 1 };
    let objB = { id: 1 };
    expect(objA).toBe(objB);
});
```

Here, `objA` and `objB` have the same shape, and they have the same values, but they are not the same object in memory. This test will produce the following test failure:

```
expect(received).toBe(expected) // Object.is equality
If it should pass with deep equality, replace "toBe" with
"toStrictEqual"
Expected: {"id": 1}
Received: serializes to the same string
```

Here, we can see that our test is failing, as both `objA` and `objB` are two different objects. The error output from Jest is also suggesting that we should replace `toBe` with `toStrictEqual`. We can, however, achieve the same result with the `toEqual` matcher, as follows:

```
it("should match with toEqual", () => {
    let objA = { id: 1 };
    let objB = { id: 1 };
    expect(objA).toEqual(objB);
});
```

Here, our test passes, as the `toEqual` matcher will correctly interpret the shape and values of both objects as equal.

Jest also provides a number of variations of the `toContain` matcher, which can be used to test for inclusion of a value in another value, as follows:

```
it("should contain a value", () => {
    expect("abcde").toContain("cde");
});
```

Here, our test is expecting that the value `"abcde"` contains the value `"cde"`. We can also use this concept on arrays, as follows:

```
it("should contain an array item", () => {
    let objArray = [
        { id: 1 },
        { id: 2 }
    ];
    expect(objArray).toContainEqual({ id: 2 });
});
```

Here, we have an array of objects named `objArray`, and our test is expecting that the `objArray` array should contain the value `{ id: 2 }`. Note that we are using the `toContainEqual` matcher in this case, which is used to check that an array contains a specific item.

Note that any expectation can be reversed using the `not` property, as follows:

```
it("should not contain a value", () => {
    expect("abcde").not.toContain("123");
});
```

Here, we are expecting that the value `"abcde"` should not contain the value `"123"`. All of Jest's matchers can be prefixed with the `not` property, in order to test for inequality.

Matchers also have the ability to test for errors when thrown. Consider the following function:

```
function throwsError() {
    throw new Error("this is an error");
}
```

Here, we have a function named `throwsError` that throws an error with the message `"this is an error"`. We can write a test that expects this error to be thrown as follows:

```
it("should throw an error", () => {
    expect(
        () => { throwsError() }
    ).toThrowError(new Error("this is an error"));
});
```

Here, we have a test that is using the matcher named `toThrowError` to test that the function `throwsError` actually does throw an error, with the correct error message. Note that we are providing an anonymous function within the `expect` that calls the `throwsError` function. When testing for errors to be thrown, we must wrap our call to the function that throws an error within an anonymous function, or else the test itself will not execute correctly.

Test setup and teardown

Before we run a particular test, we may wish to exercise some code beforehand. This may be to initialize a particular variable, or to make sure that the dependencies of an object have been set up. In the same vein, we may wish to execute some code once a particular test has run, or even after a full test suite has run. To illustrate this, consider the following class:

```
class GlobalCounter {
    count: number = 0;
    increment(): void {
```

```
        this.count++;
    }
}
```

Here, we have a class named `GlobalCounter` that has a `count` property, and an `increment` function. The count property is set to `0` when the class is instantiated, and the `increment` function increments its value. We can use our start up and tear-down functions in a test suite as follows:

```
describe("test setup and teardown", () => {

    let globalCounter: GlobalCounter;
    beforeAll(() => {
        globalCounter = new GlobalCounter();
    })

    beforeEach(() => {
        globalCounter.count = 0;
    });

    afterEach(() => {
        console.log(`globalCounter.count =
            ${globalCounter.count}`);
    });

});
```

Here, we have a test suite named `"test setup and teardown"`. Within this test suite, we have a variable named `globalCounter` of type `GlobalCounter`. We then use the `beforeAll` function to create an instance of the `GlobalCounter` class and assign it to the variable `globalCounter`. The `beforeAll` function will run once before all tests within a suite are run.

We then use the `beforeEach` function to assign the value `0` to the `count` property of the `globalCounter` class instance. The `beforeEach` function will run once before each test within the suite runs.

Finally, we have an `afterEach` function, which logs the value of the `globalCounter.count` property to the console. The `afterEach` function will run after each test has completed. Note that Jest also provides an `afterAll` function, which will run once after all tests within a suite have completed.

With these setup and teardown functions in place, we can write a few tests as follows:

```
it("should increment", () => {
    globalCounter.increment();
    expect(globalCounter.count).toEqual(1);
});

it("should increment twice", () => {
    globalCounter.increment();
    globalCounter.increment();
    expect(globalCounter.count).toEqual(2);
});
```

Here, we have two tests. The first test calls the `increment` function on the `globalCounter` variable once, and then checks that the count property is equal to `1`. The second test calls the `increment` function twice, and then checks that the count property is equal to `2`. Note that our `beforeEach` function is being called before each of these tests, and it resets the value of the count property back to `0` every time. The output of this test also shows that the `afterEach` function is logging the value of the `count` property to the console, as follows:

```
    console.log
      globalCounter.count = 1
      at Object.<anonymous> (matchers.spec.ts:55:17)

    console.log
      globalCounter.count = 2
      at Object.<anonymous> (matchers.spec.ts:55:17)
```

Here, we can see the value of the `count` property after each test, as logged by the `afterEach` function.

Data-driven tests

Quite often, we need the same test to be run multiple times, just with different input values. As an example of this, consider the following test:

```
[1, 2, 3, 4, 5]
    .forEach((value: number) => {
        it(`${value} should be less than 5`, () => {
            expect(value).toBeLessThan(5);
        })
    });
```

Here, we have defined an array with the numbers one through five. We are then calling the `forEach` function, which takes a function as an argument, and will execute that function once for each value in the array. Note how we have then defined a test within this `forEach` function, and that the test is expecting that the value passed in should be less than `5`. Note too, how we have used a template string in the actual name of the test, so that we can distinguish these tests by value. This test will fail, as follows:

```
FAIL  ./data_driven.spec.ts
  ● data driven tests › 5 should be less than 5

    expect(received).toBeLessThan(expected)

    Expected: < 5
    Received:   5
```

Here, we can see that the test that failed is named `"5 should be less than 5"`, and that Jest is letting us know that the value `5` is, in fact, not less than `5`.

Using `forEach` in this manner gives us a quick way of setting up a series of tests with a range of data. Let's now take a look at how we can write some data-driven tests for a validation function, which is as follows:

```
function hasValueNoWhiteSpace(value: string): boolean {
    if (
        value &&
        value.length > 0 &&
        value.trim().length > 0) {
        return true;
    }
    return false;
}
```

Here, we have a function named `hasValueNoWhiteSpace`, which takes a single string parameter, and returns a `boolean`. This code is checking whether the input value is an empty string, or whether it is an empty string after trimming. To test this function, we would expect that the value `"  "` would return `false`, but that the value `" a "` would return `true`.

Our test cases would then be similar to the following:

```
[
    [undefined, false],
    [null, false],
```

```
    [" ", false],
    ["  ", false],
    [" a ", true]
]
```

Here, we have defined an array of tuples, where each tuple has two values. The first value is the value we would like to test with, and the second value is the expected result of the `hasValueNoWhiteSpace` function. In order to create a data-driven test with this sort of input, let's write a small utility function as follows:

```
function testUsing<T>
    (values: T[], func: Function) {
    for (let value of values) {
        func.apply(Object, [value]);
    }
}
```

Here, we have a function named `testUsing` that will work with any type, `T`. This function has two parameters, which are named `values` and `func`. The `values` parameter is an array of type `T`, and the `func` parameter is a function callback. Within this `testUsing` function, we are looping through each element of the array, and calling the `apply` function to invoke the function `func` with the current `array` element. We can now use this `testUsing` function as follows:

```
testUsing(
    [
        [undefined, false],
        [null, false],
        [" ", false],
        ["  ", false],
        [" a ", true]
    ]
    , ([value, isValid]: [string, boolean]) => {

        it(`"${value}" hasValueNoWhiteSpace ? ${isValid}`,
            () => {
            isValid ?
                expect(hasValueNoWhiteSpace(value)).toBeTruthy() :
                expect(hasValueNoWhiteSpace(value)).toBeFalsy();
            }
        );

    });
```

Here, we are invoking the `testUsing` function with two arguments. The first argument is the array of values that we would like to run tests with. The second argument is a function that is destructuring each tuple within the array, into the variables named `value` and `isValid`. We then call the `it` function to create a test with each value of our array. The output of this data-driven test sequence is as follows:

```
    ✓ "undefined" hasValueNoWhiteSpace ? false
    ✓ "null" hasValueNoWhiteSpace ? false
    ✓ " " hasValueNoWhiteSpace ? false
    ✓ "  " hasValueNoWhiteSpace ? false (1 ms)
    ✓ " a " hasValueNoWhiteSpace ? true
```

Here, we can see that the `testUsing` function is running a test for each value in our data array. Depending on the second value of each tuple, the test will determine whether the `hasValueNoWhiteSpace` function returns the correct boolean value.

Data-driven tests are a convenient way of writing unit tests where the only real change to a series of tests is either an input or a resulting value, but the body of the test itself remains the same.

Jest mocks

When testing our code, we often have the situation where we want to ensure that a particular function was called, or that it was called with the correct parameters. This is most often seen when a particular function calls other functions in a sequence, in order to execute some business logic. We may call an `initialize` function, for example, and this `initialize` function may call a number of REST services to load data. When writing a test for our `initialize` function, we would want to ensure that all of the calls to REST services were called. To ensure that functions are called, we use Jest mocks, or Jest spies.

As an example of where we can use a Jest mock, consider the following class:

```
class MyCallbackClass {
    executeCallback(
        value: string,
        callbackFn: (value: string) => null
    ) {
        console.log(`executeCallback invoking callbackFn`);
        callbackFn(value);
    }
}
```

Here, we have a class named `MyCallbackClass`, which has a single method named `executeCallback`. The `executeCallback` function accepts two parameters, named `value` of type string, and `callbackFn`, which is a function that has a single parameter named `value` of type string. The `executeCallback` method logs a value to the console, and then invokes the callback function that was passed in, along with the string value that was passed in.

Let's take a look at how we can use a Jest mock function to use as the `callbackFn` argument, as follows:

```
it("should mock callback function", () => {
    let mock = jest.fn();

    let myCallbackClass = new MyCallbackClass();
    myCallbackClass.executeCallback('test', mock);

    expect(mock).toHaveBeenCalled();
});
```

Here, we start our test by creating a variable named `mock` that is assigned to the result of calling the `jest.fn` function. The `jest.fn` function essentially creates a mock function, which can then be used as a quick replacement for a callback function.

Our test then creates an instance of the `MyCallbackClass` named `myCallbackClass`, and then calls the method `executeCallback`, passing in the string value `'test'` and our mock function as arguments. We then use the `toHaveBeenCalled` matcher on this mock function instance, which will test that the `executeCallback` method actually invokes the callback function that we passed in as an argument.

We can also check that the function that was passed in as an argument is called with the correct arguments, as follows:

```
it("should call testFunction with argument using mock", () => {
    let mock = jest.fn();

    let myCallbackClass = new MyCallbackClass();
    myCallbackClass.executeCallback("argument_1", mock);
    expect(mock).toHaveBeenCalledWith("argument_1");
});
```

Here, we are using the `toHaveBeenCalledWith` matcher, instead of the `toHaveBeenCalled` matcher, as we used in our previous test. This gives us the ability to check that the function passed in as an argument was called with the correct arguments.

Jest mocks are a quick and convenient way of creating callback functions.

Jest spies

Jest also provides us with the ability to check whether a particular class method has been called, using what is known as a spy. Consider the following class definition:

```
class MySpiedClass {
    testFunction() {
        console.log(`testFunction() called`);
        this.testSpiedFunction();
    }
    testSpiedFunction() {
        console.log(`testSpiedFunction called`)
    }
}
```

Here, we have a class named `MySpiedClass`, which has two methods. The first is named `testFunction`, and the second method is named `testSpiedFunction`. Note that the `testFunction` function logs a message to the console, and then invokes the `testSpiedFunction` within the method body. To properly test this code, we will require a test that ensures that the `testSpiedFunction` is invoked when we call the `testFunction` method. This can be accomplished with a spy, as follows:

```
it("should call testSpiedFunction", () => {
    let mySpiedClass = new MySpiedClass();
    const testFunctionSpy = jest.spyOn(
        mySpiedClass, 'testSpiedFunction');

    mySpiedClass.testFunction();
    expect(testFunctionSpy).toHaveBeenCalled();
});
```

Here, we have a test that starts by creating an instance of the `MySpiedClass` and assigns it to a variable named `mySpiedClass`. We then create a variable named `testFunctionSpy` and assign to it the result of the call to `jest.spyOn`. The `jest.spyOn` function takes two arguments. The first argument is the class instance that we would like to spy on, and the second argument is the function name of the class method that we wish to create the spy on. In this test, we have created a spy on the `testSpiedFunction` method, which will be invoked when we call the `testFunction` method.

Our test expectation in this case is that the `testFunctionSpy`, which is spying on the `testSpiedFunction` method, will have been called. Using spies, therefore, allows us to test our code paths and ensure that they call other methods when we expect them to.

The output of this test is as follows:

```
  console.log
    testFunction() called

  console.log
    testSpiedFunction called
```

Here, we can see that both of our console logs from the class methods have been called. When we invoke the `testFunction` method, it will log the first message, `"testFunction() called"`, to the console. The `testFunction` method then invokes the `testSpiedFunction` method, which will log the second message, `"testSpiedFunction called"` to the console.

Note that we can also provide a mock function implementation on a spy. Consider the following test:

```
it("should call mock of testFunction", () => {
    let mySpiedClass = new MySpiedClass();
    const testFunctionSpy = jest.spyOn(
        mySpiedClass, 'testFunction')
        .mockImplementation(() => {
            console.log(`mockImplementation called`);
        });

    mySpiedClass.testFunction();
    expect(testFunctionSpy).toHaveBeenCalled();
});
```

Here, we have used the `mockImplementation` function on our spy to provide an implementation of the function that will be called during the test. This mock implementation will log a message to the console showing that it will be called instead of the class method. The output of this code is as follows:

```
  console.log
    mockImplementation called

      at MySpiedClass.<anonymous> (spies_mocks.spec.ts:61:25)
```

Here, we can see that the mock implementation of the `testFunction` method was invoked instead of the actual implementation of the `testFunction` method. This behavior is interesting to note.

When we create a spy on a method, we are able to check whether the method was invoked, and whether it was invoked with the correct parameters. Creating a spy, however, will not prevent the body of the method being run. If we are wanting to override the body of the method, and not allow the body of the method to be invoked, then we need to provide a mock implementation.

This distinction of whether or not the body of the method is invoked is extremely important when writing tests. As an example, let's assume that a method will connect to a database, run a query, and return results. In this instance, we do not want the body of the method to be run, as we do not have a database instance to connect to. We want to mock out any interactions with a database completely. In these cases, we will need to provide a mock implementation.

Spies returning values

When we wish to mock out the return value of a function, we can easily just return a value from a mock implementation. As an example of this, consider the following class:

```
class MyMockedClass {
    functionToBeMocked(): number {
        return 5;
    }
}
```

Here, we have a class named `MyMockedClass`, which has a single function named `functionToBeMocked`, that returns the value `5`. We can override this return value with a mock implementation as follows:

```
it("should return value from mocked", () => {
    let myMockedClass = new MyMockedClass();
    jest.spyOn(myMockedClass, 'functionToBeMocked')
        .mockImplementation((): number => {
            return 10;
        });

    expect(myMockedClass.functionToBeMocked()).toEqual(10);
});
```

Here, our test creates an instance of the `MyMockedClass` named `myMockedClass`, and then spies on the `functionToBeMocked` function. Our mock implementation of this function returns the value `10`. Our test then expects the value of the `functionToBeMocked` method to return the value `10`. This test will pass as expected, as we have replaced the original method implementation that returned the value `5` with the mock implementation that returns the value `10`.

Returning values from mock implementations means that we can simulate any sort of external interaction with other systems within our tests. We can mock out calls to a database, or calls to a REST endpoint, and inject standard values that we can test against.

Asynchronous tests

As we have seen with our exploration of JavaScript and TypeScript, a lot of code we write is asynchronous. This means that we have no control of exactly when a callback will be invoked, or a Promise will resolve, as we are waiting for an event to occur that is outside of our control. This often presents problems in our unit testing, where we need to wait for an asynchronous event to complete before we can continue with our test. As an example of this, consider the following class:

```
class MockAsync {
    executeSlowFunction(
        complete: (value: string) => void
    ) {
        setTimeout(() => {
            complete(`completed`);
        }, 1000);
    }
}
```

Here, we have a class named `MockAsync` that has a single method named `executeSlowFunction`. This function takes a callback function named `complete` as its only parameter, and then invokes it after 1 second. We might write a test for this class as follows:

```
describe("failing async tests", () => {
    it("should wait for callback to complete", () => {
        let mockAsync = new MockAsync();
        console.log(`1. calling executeSlowFunction`);
        let returnedValue !: string;
        mockAsync.executeSlowFunction((value: string) => {
```

```
            console.log(`2. complete called`);
            returnedValue = value;
        });
        console.log(`3. checking return value`);
        expect(returnedValue).toBe("completed")
    });
});
```

Here, we have a test suite named `"failing async tests"`, and a test named `"should wait for callback to complete"`. This test starts by creating an instance of the `MockAsync` class, named `mockAsync`. It then logs a message to the console, and creates a variable named `returnedValue` to hold the value that will be returned by a call to the `executeSlowFunction` method. It then invokes the `executeSlowValueFunction` function, and defines a callback function. This callback function logs a message to the console, and then stores the returned value from the callback in the `returnedValue` variable. The test then logs a third message to the console, and checks that the `returnedValue` variable contains the string value `"completed"`.

Running this test, however, gives the following failure output:

```
1. calling executeSlowFunction
3. checking return value

 ● failing async tests › should wait for callback to complete

    expect(received).toBe(expected) // Object.is equality

    Expected: "completed"
    Received: undefined

      21 |         });
      22 |         console.log(`3. checking return value`);
    > 23 |         expect(returnedValue).toBe("completed")
         |                               ^
      24 |     });
      25 | });
      26 |
```

Here, we can see that the messages logged to the console are `"1. calling executeSlowFunction"`, and `"3. checking return value"`. We are completely missing the message `"2. executeSlowFunctionReturned"`. Our test is also failing, as the expected value of the `returnedValue` variable should be `"completed"`, but is, in fact, `undefined`.

What is causing this test to fail is the fact that the test itself is not waiting for 1 second for the `executeSlowFunction` function to call the `complete` callback. What we really need is a way to signal to our test that it should only execute the test expectation once the asynchronous call has completed.

Using done

Jest provides a method named `done` to signify that the test run should wait for an asynchronous call to complete. The `done` function can be passed in as an argument in any `beforeAll`, `beforeEach`, or `it` function, and will allow our asynchronous test to wait for the `done` function to be called before continuing. Let's rewrite our previous failing test using `done` as follows:

```
describe("async test with done ", () => {
    let returnedValue!: string;

    beforeEach((done: jest.DoneCallback) => {
        let mockAsync = new MockAsync();
        console.log(`1. calling executeSlowFunction`);
        mockAsync.executeSlowFunction((value: string) => {
            console.log(`2. executeSlowFunction returned`);
            returnedValue = value;
            done();
        })
    });

    it("should return value after 1 second", () => {
        console.log(`3. checking returnedValue`);
        expect(returnedValue).toEqual("completed");
    })
});
```

Here, we have a test suite named `"async test with done"`. Within the body of this suite, we have declared a variable named `returnedValue` to hold the value that is returned by our asynchronous callback. We then define a `beforeEach` function, which has a single argument named `done`, of type `jest.DoneCallback`. Within this `beforeEach` function, we create an instance of the class `MockAsync`, log a message to the console, and then invoke the `executeSlowFunction` method with a callback function. Within this callback function, we log a message to the console, set the value of the `returnedValue` variable, and call the `done` function.

When using `done`, the `beforeEach` function will wait until the `done` function is actually called before continuing with the test run. This `done` function is only called once the 1 second delay is up, and the `executeSlowFunction` invokes the callback that we provided.

The test that we are running is named `"should return value after 1 second"`, and logs a message to the console before checking the value of the `returnedValue` variable. Our test now succeeds with the following messages being logged to the console:

```
  console.log
    1. calling executeSlowFunction
    2. executeSlowFunction returned
    3. checking returnedValue
```

Here, we can see that the order of console logs matches the order of execution that we are expecting in our test. The `beforeEach` function is waiting for the `executeSlowFunction` to invoke its callback, and when it does, the `done` function is invoked, which will continue the test run.

From the Jasmine documentation:

"The spec will not start until the `done` function is called in the call to `beforeEach`, and this spec will not complete until its `done` function is called. By default, Jasmine will wait for 5 seconds before causing a timeout failure. This can be overridden using the `jasmine.DEFAULT_TIMEOUT_INTERVAL` variable."

Using async await

If an asynchronous function is using Promises, then we can use the async await syntax to run tests, in the same way that we would normally. As an example, consider the following class that is using a Promise to return a value, as follows:

```
class AsyncWithPromise {
    delayedPromise(): Promise<string> {
        return new Promise<string>(
            (resolve: (str: string) => void,
                reject: (str: string) => void) => {
                setTimeout(() => {
```

```
                    console.log(`2. returning success`);
                    resolve("success");
                }, 1000)
            }
        )
    }
}
```

Here, we have a class named `AsyncWithPromise` that has a single method named `delayedPromise`, which returns a Promise of type string. This promise will log a message to the console, and then call the `resolve` function with the string value of `"success"`, after a 1 second delay. We can now write a unit test as follows:

```
describe("async test", () => {
    it("should wait 1 second for promise to resolve",
        async () => {
        let asyncWithPromise = new AsyncWithPromise();
        console.log(`1. calling delayedPromise`);
        let returnValue = await asyncWithPromise.delayedPromise();
        console.log(`3. after await`);
        expect(returnValue).toEqual("success");
    })
});
```

Here, we have a test suite named `"async test"`, which has a single test within it named `"should wait 1 second for promise to resolve"`. Note how our test function is prefixed with the `async` keyword.

Within the body of this test, we create an instance of the class `AsyncWithPromise`, named `asyncWithPromise`. We then log a message to the console and set the value of a variable named `returnValue` to the result of the call to the `delayedPromise` method. Note how we are using the `await` keyword to pause test execution until this Promise is resolved. Our test then logs a third message to the console, and finally checks the value of the `returnValue` variable against the expected string, `"success"`. The output of this test is as follows:

```
  async test
    ✓ should wait 1 second for promise to resolve (1012 ms)

  console.log
    1. calling delayedPromise
    2. returning success
    3. after await
```

Here, we can see that the test is passing, and that the execution time of the test was 1,012 milliseconds. This means that the test waited for the Promise to be resolved for 1,000 milliseconds, and that the rest of the test execution took 12 milliseconds. Note too, that the messages logged to the console are also in the correct order. The first message was logged just before we called the `delayedPromise` function, the second message was logged within the `delayedPromise` function when the 1,000 millisecond timeout elapsed, and the third message was logged after our `await` call.

Using async await syntax within a unit test is exactly the same as using it in normal code. As long as we mark the test function with the `async` keyword, the test will pause when it encounters the `await` keyword and continue when the asynchronous code completes.

HTML-based tests

Jest uses a library named jsdom to allow for testing HTML elements and interactions. Jsdom is not an actual browser; it is a library that implements the JavaScript DOM API, and can, therefore, simulate a full-blown browser experience. The benefit of using jsdom is in the speed at which we can run our tests, and the fact that we do not have to provide an environment that can run a full browser. Running a full internet browser generally assumes that the tests are running on a standard computer, and as such, has an actual screen. This means that virtual machines that run tests must be configured with a screen, and adds the extra requirements of having a screen driver that can run at the required resolution.

We can install the jsdom library using npm as follows:

```
npm install jsdom --save-dev
```

While we are at it, let's install the jquery library as well:

```
npm install jquery
```

And the `@types` declaration files for both libraries as follows:

```
npm install @types/jsdom --save-dev
npm install @types/jquery --save-dev
```

With jsdom and jquery installed, we can now write a test that checks whether the DOM has been updated. Consider the following code:

```
function setTestDiv(text: string) {
    $('#test_div').html(`<p>${text}</p>`);
}
```

Here, we have a function named `setTestDiv`, which has a single parameter named `text` of type string. This function is using jQuery to set the HTML of the DOM element with the id `test_div`. The `$('#test_div')` function call will find the DOM element with the id `test_div`, and the `html` function will set the HTML of the element to the provided string. We have set the HTML in this instance to a paragraph element containing the text provided as the `text` parameter.

We can now write a test for this function as follows:

```
it("should set text on div", () => {
    document.body.innerHTML =
        `<div id="test_div"></div>`;

    let htmlElement = $('#test_div');
    expect(htmlElement.length).toBeGreaterThan(0);

    setTestDiv("Hello World");

    expect(htmlElement.html()).toContain("Hello World");
});
```

Here, we have a test named `"should set text on div"`, which starts by creating a DOM div element with the id `"test_div"`. We are assigning a string value to the `document.body.innerHTML` property, which will create the DOM elements. Our test then uses the jQuery search function to find the element with the id `"test_div"`, and assign it to a variable named `htmlElement`. We then test whether the variable `htmlElement` has a length that is greater than `0`. If jQuery does not find the named element in the DOM, it will return an empty object, or `{}`, which has a length of `0`.

We then call the `setTestDiv` function, which will update the HTML of the `test_div` element. Finally, our test calls the `html` function, which will retrieve the updated HTML for the element, and checks to see that it contains the text `"Hello World"`. The div element's HTML will, in fact, be set to `<p>Hello World</p>`, which will allow our test to pass.

DOM events

There are times when we need to test DOM events, such as an `onclick` or `onselect` event, which are tied to a particular JavaScript function held within an HTML script tag. The jsdom library will parse these script tags and JavaScript functions and execute them, as long as we have set up the test to do so. As an example of this, consider the following test string:

```
const htmlWithClickEvent = `
<body>
    <script type="text/javascript">
        function handle_click_event() {
            console.log("handle_click_event() called.");
        }
    </script>
    <div id="click_handler_div"
        onclick="handle_click_event()"
    >Click Here</div>
</body>
`;
```

Here, we have a string constant named `htmlWithClickEvent`. This variable defines some HTML that we wish to test. It starts with a `body` tag, and within this `body` tag, has a `script` tag with a JavaScript function named `handle_click_event`. This function just logs a message to the console. We then have a `div` element, with the id of `"click_handler_div"`, which defines an `onclick` DOM event, which will execute the `handle_click_event` function. This `div` element has the text `"Click Here"`.

If this were HTML embedded within a standard page, then when we click on the text `"Click Here"`, the `onclick` handler will trigger the `handle_click_event` function, and the message `"handle_click_event() called."` will be logged to the console.

In order to unit test this HTML block, we can write a test as follows:

```
it("should trigger an onclick DOM event", () => {

    let dom = new JSDOM(
        htmlWithClickEvent,
        { runScripts: "dangerously" });

    let clickHandler = <HTMLElement>
        dom.window.document.querySelector("#click_handler_div");
    let clickEventSpy = jest.spyOn(clickHandler, "click");
    clickHandler.click();
```

```
    expect(clickEventSpy).toHaveBeenCalled();
});
```

Here, we have a test named `"should trigger an onclick DOM event"`, which starts by creating a variable named `dom`, which is an instance of a new `JSDOM` class. This `JSDOM` class is being constructed with two arguments. The first argument is the string value that represents the HTML itself, and the second argument is a configuration object with the property `runScripts`, which has been set to `"dangerously"`. This option will run both the `onclick` DOM events, and any JavaScript that is contained within our source HTML. Without this option, these will be ignored.

Our test then declares a variable named `clickHandler` that is using the `querySelector` function to find a DOM element with the id of `"click_handler_div"`. Note that we must use the `querySelector` function on the `dom.window.document` property, which represents our HTML DOM root. We then create a jest spy on the `"click"` function of this `clickHandler` object and invoke it by calling the `click` function. Our test then expects that the spy we set up should have been called.

Note that the console output for this test is as follows:

```
  console.log
    handle_click_event() called.
```

Here, we can see that the JavaScript function named `handle_click_event` has been invoked, as it is logging the message to the console. This means that the `JSDOM` class has loaded our HTML block, parsed it as standard HTML, and recognized that this function is inside a script tag, and is therefore executable.

We can use this sort of technique for other DOM events, including `onchange`, `onfocus`, `ondrag`, or anything else. Having the ability to construct snippets of HTML and test them is a very powerful feature of jest and jsdom. We can fill in forms, click on the **submit**, **cancel**, or **OK** buttons, and generally simulate user interaction with our application.

Protractor

Protractor is a Node-based test runner that is used to tackle end-to-end or automated acceptance testing. Essentially, it is a tool that allows us to programmatically control a web browser. Just like in manual testing, Protractor has the ability to browse to a specific page by entering its URL, and then interact with the elements on the page itself. As an example of how it can be used, suppose that we have a website that has a login page, and all further interaction with the site requires a valid login. We can use Protractor to browse to the login page at the start of each test, enter valid credentials, hit the **Login** button, and then interact with the site's pages.

Protractor is installed as an npm package as follows:

```
npm install -g protractor
```

Here, we have installed the protractor package as a globally accessible node package.

We will get into running Protractor a little later, but first, let's discuss the engine that Protractor uses under the hood to drive the browser, a suite named Selenium.

Selenium

Selenium is a driver for web browsers. It allows the programmatic control of web browsers and can be used to create automated tests in Java, C#, Python, Ruby, PHP, Perl, and, of course, JavaScript. Protractor uses the Selenium driver under the hood to control web browser instances. Note that Selenium requires a Java runtime to have been installed, so go ahead and install Java if required.

Let's start by installing the Selenium Server for use with Protractor as follows:

```
webdriver-manager update
```

This command will download a few packages, including the correct version of chromedriver for our operating system. Once it has completed, we now need to start the Selenium server as follows:

```
webdriver-manager start
```

If all goes well, Selenium will report that the server has been started, as follows:

```
Selenium Server is up and running on port 4444
```

Once the server is up and running, we will need a configuration file for Protractor that tells Protractor where to find the Selenium server. Let's create a directory named `protractor`, and within it a file named `protractor.conf.js`, containing the following:

```
exports.config = {
    seleniumAddress: 'http://localhost:4444/wd/hub',
    specs: ['*.spec.js']
}
```

Here, we are assigning some properties to the `exports.config` object. The first property is named `seleniumAddress`, and points to the URL of the Selenium server. The second property, named `specs`, lists an array of tests to run. In this example, we are using a wildcard entry to specify the fact that any file that ends in `.spec.js` contains a Protractor test specification.

Let's now write a Protractor test, in a file named `test_google.spec.ts` as follows:

```
it("should navigate to google and find a title", async () => {
    browser.driver.get('https://www.google.com');
    expect(browser.driver.getTitle()).toContain("Google");
    let title = await browser.driver.getTitle();
    console.log(`await getTitle() returned : ${title}`);
});
```

Here, we have a test named `"should navigate to google and find a title"`. Note that this test has been marked as `async`, so that we can use an await call within the body of the test. We then use the `browser.driver` object, which represents the browser instance, and call the `get` function with a URL. This will cause the browser to navigate to the URL we provided. Our test expectation is checking whether the title of the web page is, in fact, `"Google"`, using the `getTitle` function.

Our test then creates a variable named `title`, and calls the `browser.driver.getTitle` function using await. Note that calls to Selenium are, by nature, asynchronous, as Selenium needs to interact with the browser instance, and we don't know how long it will take to navigate to a web page, or access any of its elements. Our test is then logging the value that was returned by the `getTitle` function call to the console.

If you keep an eye on your screen while running this test, you will notice that Protractor is starting a new instance of a Chrome browser session, navigating to the Google home page, and then running the tests. The output of this test is as follows:

```
Started
await getTitle() returned : Google
.
1 spec, 0 failures
Finished in 1.707 seconds
```

Here, we can see that the test started, and that the `console.log` message we were expecting in our test has been logged to the console.

Finding page elements

Selenium has a number of functions that we can use in order to find HTML elements on a page during testing. We can search for an element using its `id` property, or we can use a CSS selector, or xpath. As an example of this, consider the following test:

```
it("should search for the term TypeScript", async () => {
    browser.driver.get('https://www.google.com');
```

```
    await browser.driver.findElement(
        By.css(`input[type=text]`))
        .sendKeys("TypeScript");

    await browser.driver.findElement(
        By.xpath(
            `//*[@id="tsf"]/div[2]/div[1]/div[1]/div/div[2]/input`
        )).sendKeys(Key.ENTER);

    let title = await browser.driver.getTitle();

    expect(title).toContain(`TypeScript`);
    console.log(`await getTitle() returned : ${title}`);
});
```

Here, we have a test named `"should search for the term TypeScript"`. This test navigates to the Google home page and then uses the Selenium function named `findElement` to find an HTML element on the page. We are using the static `By.css` function in order to find the `input` element of `type="text"`. Once we have found the element, we input the text `"TypeScript"` by using the `sendKeys` function.

We then have a second call to the `findElement` function, but this time are using the `By.xpath` static function. Note that in this test, both the `By.css` and the `By.xpath` selectors will find the same input element. We are then using the `sendKeys` function to simulate the user hitting the *Enter* key. Our test then finds the page title, and checks that it contains the string `"TypeScript"`. Note that if you watch the behavior of the browser during this test, you will notice that the browser is actually being redirected to another page, which contains the results of the search. Google is modifying the page title for this second page to `"TypeScript - Google Search"`, and so our test is actually waiting for this second page to render before checking the page title.

We can use the Chrome developer tools to find the correct screen elements in our tests by right-clicking on the element in question, and selecting the menu option **Inspect**. This will open up the Chrome developer window, and highlight the HTML that is rendering the screen element. With the screen element highlighted, another right-click will bring up a context menu, and we can then choose the **Copy** menu item, and select what to copy from the various options, as shown in the following screenshot:

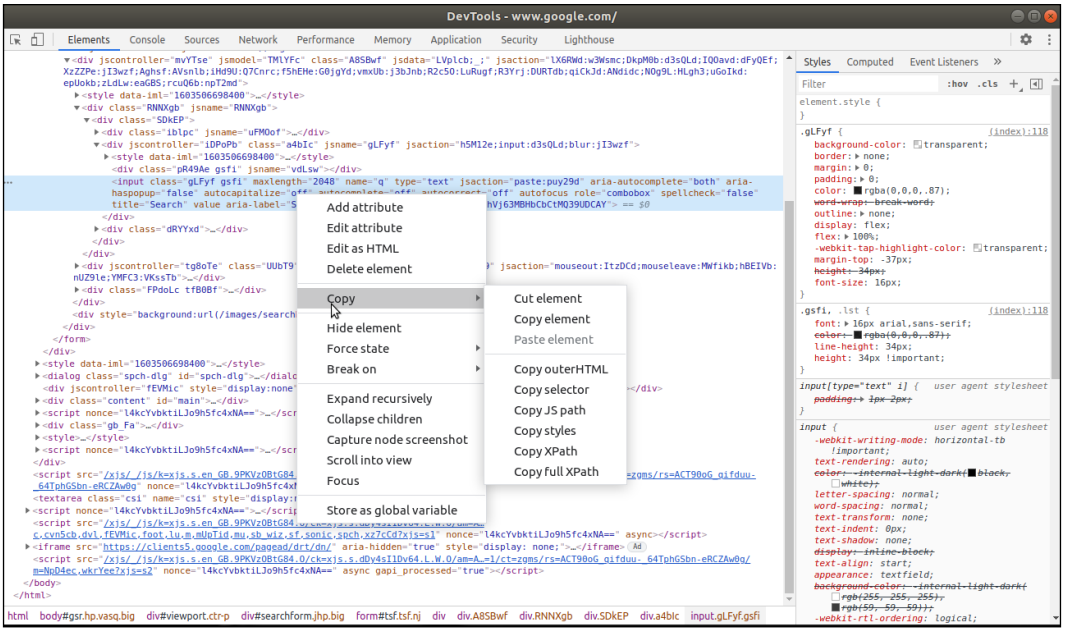


Figure 10.7: Chrome developer tools showing copy options for an HTML element

Here, we can see that we are able to copy a selector, a JS path, an XPath, or a full XPath for this element, among other options.

Finding page elements in this way allows us to quickly and easily incorporate selectors into our tests.

Summary

In this chapter, we have explored the concepts of test-driven development from the ground up. We have discussed the various types of testing, including unit, integration, and acceptance tests, along with black-box or white-box testing. We then explored the Jest testing framework, how to set it up, and how to use matchers, spies, and mocks. We explored the concepts surrounding asynchronous testing, and also how to inject HTML into the DOM using jsdom. Finally, we explored Protractor and Selenium, which are used for black-box and end-to-end tests of HTML pages.

In the next chapter, we will explore the Angular framework, and how it can be used to write single-page applications in TypeScript.

11
Angular

One of the watershed moments in the story of the TypeScript language came when it was announced that the Microsoft and Google teams had been working together on Angular 2. Angular 2 was a much anticipated update to the popular Angular (or Angular 1) single-page application framework, which was written in JavaScript. The Google team that built Angular 2 originally proposed a new language named AtScript, which would allow Angular 2 applications to use newer ECMAScript 6 and 7 language features, and make the Angular 2 syntax cleaner and easier to understand. It was, in fact, intended for AtScript to run on top of TypeScript.

Following several months of collaboration, it was announced that all of the necessary features of the AtScript language would be absorbed into the TypeScript language, and that Angular 2 would be written in TypeScript. This meant that the providers of the language (Microsoft) and the consumers of the language (Google) were able to agree on the requirements and immediate future of the language. This collaboration showed that the TypeScript language went through intense scrutiny from a well-renowned JavaScript framework team, and passed with flying colors.

The original Angular framework (version 1) was renamed AngularJS, and the new Angular 2 framework was simply named Angular. In the years since Angular 2's release, in 2016, it has maintained a steady release pipeline, and is currently at version 11.

In this chapter, we will build an Angular application, and discuss the various elements that Angular uses. In particular, we will cover the following topics:

- Setting up an Angular application
- Angular modules and shared modules
- Angular Material

- DOM events
- Angular services
- Dependency injection
- Reactive forms
- Unit testing

Angular setup

Angular uses a command-line tool known as the Angular CLI to facilitate the creation of Angular applications and components. The Angular CLI can be installed using npm as follows:

```
npm install -g @angular/cli
```

Here we are installing the package `@angular/cli` globally using npm. Once installed, the Angular CLI provides a utility named `ng`, which can be used to create an Angular application as follows:

```
ng new angular-app
```

Here, we are invoking the Angular CLI and specifying that we wish to create a new angular application named `angular-app`. The Angular CLI will ask a few questions when creating an application, such as whether we would like to enforce strict type checking, whether to include Angular routing, and which stylesheet format we would like to use. In this sample application, we chose *Yes* for strict type checking, *Yes* for Angular routing, and *SCSS* for the stylesheet format.

Once the Angular CLI has completed, let's switch to the newly created `angular-app` directory and start up a development server as follows:

```
cd angular-app
npm start
```

Here, we are calling the `start` script that Angular has included in the `package.json` file. This script will invoke the Angular CLI with the command `ng serve`, which will compile our application, start a local web server, and additionally run the compiler in watch mode. By default, the web server will run on port `4200`, which we can then browse to as follows:

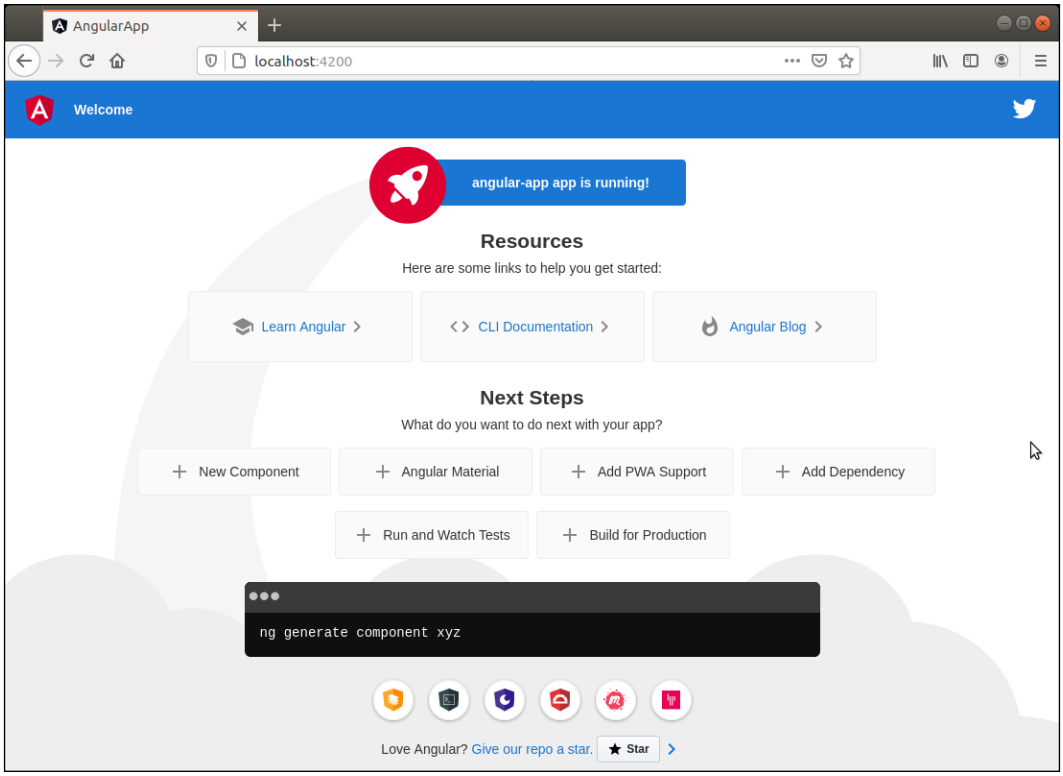


Figure 11.1: Default Angular application running on port 4200

Our stock standard Angular application is up and running, which has been set up and configured automatically by the Angular CLI.

Application structure

Before we go ahead and modify our new Angular application, let's take a quick tour of the application structure that the Angular CLI has put in place for us. In the project root directory, we can find a `tsconfig.json` file, a `package.json` file, and an `angular.json` file, among others. We already know that the `tsconfig.json` file will be used for TypeScript compilation options, and that the `package.json` file will be used to store the package dependencies that we are using. The `angular.json` file is used for Angular specific settings. As an example, the `"build"` option specifies what stylesheets should be included, where the output path is, and the configuration options that are used when building production code, as opposed to development code. Interestingly, there are also options for linting, internationalization, unit testing, and end-to-end testing.

The Angular CLI will also create a `src` directory, and under this directory, an `app`, `assets`, and `environments` directory. The `src/app` directory contains the Angular components and modules that are used to build our application, and the `assets` directory is used for static content, such as images or fonts. Within the `src` directory, let's take a quick look at the `index.html` file that has been created for us, as follows:

```
<!doctype html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>AngularApp</title>
    <base href="/">
    <meta name="viewport"
        content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

<body>
    <app-root></app-root>
</body>

</html>
```

Here, we have a pretty standard HTML file that sets the page title and the favorite icon within the `<head>` tag. The interesting part of this file is within the `<body>` tag, where there is a single HTML tag named `<app-root>`. This tag tells Angular which component to render on the main page. If we now open the `src/app/app.component.ts` file, we will see where this `<app-root>` tag is defined, as follows:

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent {
    title = 'angular-app';
}
```

Here, we can see the definition of a class named `AppComponent`, which is using a class decorator named `@Component` to specify three properties for this class. The first property is named `selector`, and the value of this property is `'app-root'`, which matches the `<app-root>` tag within the `index.html` page. It is this selector that is used within HTML in order to render a component. The second property is named `templateUrl`, and specifies a file named `'./app.component.html'`. This is the file that contains all of the HTML that the component uses. The third property is named `styleUrls`, and this references the `'app.component.scss'` file that will be used for any Sass style CSS that is needed for the component.

Angular modules

An Angular module is a grouping of Angular components into a logical set. Modules also allow us to specify any dependencies that a set of components may have, which may include other modules, or service providers. To explore how modules work within Angular, let's first create a new component named header, as follows:

```
ng generate component header
```

Here, we have called the Angular CLI with the argument `generate component`, and have specified that the component should be named `header`. The Angular CLI will then output the following:

```
CREATE src/app/header/header.component.scss (0 bytes)
CREATE src/app/header/header.component.html (21 bytes)
CREATE src/app/header/header.component.spec.ts (626 bytes)
CREATE src/app/header/header.component.ts (276 bytes)
UPDATE src/app/app.module.ts (503 bytes)
```

Here, we can see that the Angular CLI has created four files in the `src/app/header` directory, which include the Sass styles, the HTML file, a unit test file, and the component itself. Note on the last line of this output that the Angular CLI has also updated the file `src/app/app.module.ts`. Let's now take a closer look at this file as follows:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
```

```
@NgModule({
    declarations: [
        AppComponent,
        HeaderComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, we can see that this file is importing `BrowserModule` and `NgModule` from Angular, and is then importing `AppRoutingModule` from the file `app-routing.module.ts`. We will cover routing in a later chapter. This file then imports both the `AppComponent` class and our new `HeaderComponent` class.

The file then defines a class named `AppModule`, which is using the `@NgModule` decorator to define four properties of the `AppModule` class. The `declarations` property contains an array of all of the HTML components that this module uses. This essentially registers the tag `<app-root>` and the new `<app-header>` tag as being available for use within our HTML files. The `imports` property lists modules that this module uses, which in this case are `BrowserModule` and `AppRoutingModule`. We then have a `providers` property, which lists the service providers that are used in this module, and is currently empty. Finally, the `bootstrap` property tells Angular to create the `AppComponent` class, and render it into the DOM.

Now that we have an updated `AppModule` class, let's replace all of the HTML in the file `app.component.html` with the following:

```
<app-header></app-header>
```

Here, we have added an `<app-header>` tag to our HTML, which matches the `selector` property of our newly created header component. What this means, then, is that the `index.html` file references the `<app-root>` tag, which loads the `AppComponent` class, which then loads the `HeaderComponent` class.

Our `HeaderComponent` class uses the `header.component.html` file as its template, which contains the following:

```
<p>header works!</p>
```

Here, we can see that the generated HTML file contains a paragraph `<div>` element with the text `"header works!"`. Note that if we are running Angular in watch mode, which is the default when running `ng serve`, our page will automatically update, once Angular has detected and processed our change, as follows:

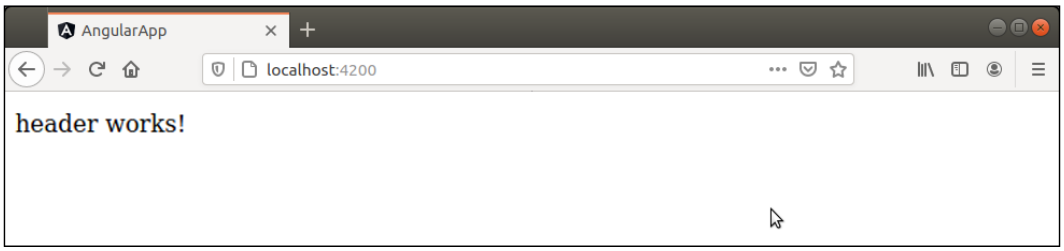


Figure 11.2: Angular application showing the default output of the header components

We have now created a new component, integrated this component within our Angular application, and rendered it on to the page.

Angular Material

The Angular team at Google also maintains a set of user interface components for use within Angular named Angular Material. This set of components includes buttons, header bars, icons, dropdowns, menu items, date pickers, and dialogues, along with a host of useful and stylish components. Angular Material can be added to an Angular project using the Angular CLI as follows:

```
ng add @angular/material
```

This command will download and install the necessary packages via npm. It will also ask a number of questions about choosing a prebuilt theme, topography, and animations. Feel free to choose any of the themes presented, and then choose the *Yes* option for the following two questions, as follows:

```
? Choose a prebuilt theme name, or "custom" for a custom theme:
Indigo/Pink        [ Preview: https://material.angular.io?theme=indigo-
pink ]
? Set up global Angular Material typography styles? Yes
? Set up browser animations for Angular Material? Yes
```

The installation of Angular Material will also update the existing Angular application with modifications to the `angular.json` file, the `index.html` file, and the `app.module.ts` file.

Let's move on and use the Material toolbar in our header component as follows:

```
<mat-toolbar color="primary">
    <span>Switch Sales</span>
    <span class="example-spacer"></span>
</mat-toolbar>
```

Here, we are using the `<mat-toolbar>` tag to include a material toolbar in our header component. This toolbar has a span with the text `"Switch Sales"`, and another span that has the CSS class of `"example-spacer"`.

Compiling our application now, however, will fail with the following error message:

```
'mat-toolbar' is not a known element
If 'mat-toolbar' is an Angular component, then verify that it is part
of this module
```

Here, we can see that the Angular compiler is attempting to find the module that has the `selector` property of `mat-toolbar`, but it can't find it. The simplest solution to this problem is to add the correct module to the list of `imports` that the application is using by modifying the `app.module.ts` file as follows:

```
import { MatToolbarModule } from '@angular/material/toolbar';

@NgModule({
    declarations: [
        AppComponent,
        HeaderComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule,
        BrowserAnimationsModule,
        MatToolbarModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, we have imported the `MatToolbarModule` from the Angular Material library and added it to the `imports` array. Note that each component in the Angular Material library is documented on the Angular Material website, and each component has an API tab that will show us the name of the module that needs to be imported.

Note that when Angular runs in watch mode, it will monitor our source code files for changes, automatically recompile them, and then refresh our browser when done. Unfortunately, this process does sometimes fail, and these failures generally occur when we add new packages to the project, or when we add or move modules around. It is generally a good idea to restart the `npm start` process when we add or remove packages to an Angular project, or if we see a large amount of errors being generated on the command line.

Our header component, using the Material toolbar, should now look like the following screenshot:

Figure 11.3: Screenshot of the Angular header component with the Material toolbar

Now that we have integrated the Angular Material library, let's take a look at how we can reuse this library on multiple pages, through the use of a shared module.

A shared module

The Angular Material library has quite a few of these individual modules, and adding each of them to the `app.module.ts` module file can become quite tedious. This can also be a problem when we start to unit test our Angular components, as each unit test will need to import each of these modules individually in order for our tests to run. What we can do, however, is build a shared module that contains all of the Angular Material modules in one hit, and then all we need to do is to import this shared module when we require access to Angular Material components.

We can create a module using the Angular CLI, as follows:

```
ng generate module shared
```

Here, we are calling the Angular CLI to generate a module named `shared`. This command will create a stub module definition in the file `shared/shared.module.ts`. We can now import all of the Angular Material modules that we will be using throughout our application into this shared module as follows:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from
    '@angular/platform-browser/animations';

import { MatToolbarModule } from "@angular/material/toolbar";
import { MatIconModule } from "@angular/material/icon";
import { MatTooltipModule } from '@angular/material/tooltip';
import { MatSidenavModule } from '@angular/material/sidenav'
import { ReactiveFormsModule } from '@angular/forms';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';

@NgModule({
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        MatToolbarModule,
        MatIconModule,
        MatTooltipModule,
        MatSidenavModule,
        ReactiveFormsModule,
        MatFormFieldModule,
        MatInputModule,
        MatButtonModule
    ],
    exports: [
        BrowserModule,
        BrowserAnimationsModule,
        MatToolbarModule,
        MatIconModule,
        MatTooltipModule,
```

```
        MatSidenavModule,
        ReactiveFormsModule,
        MatFormFieldModule,
        MatInputModule,
        MatButtonModule
    ]
})
export class SharedModule { }
```

Here, we have the definition of a class named `SharedModule`, which is using the `@NgModule` decorator to declare a module. What is important about this module is that it is using the `imports` array to import all of the Angular Material modules that we will be using in this application, including `MatToolbarModule`, `MatIconModule`, and various others. It is also then using the `exports` array to re-export all of these modules and make them available to other modules that are using this shared module.

We can now update our `app.module.ts` file to use this shared module as follows:

```
import { SharedModule } from './shared/shared.module';

@NgModule({
    declarations: [
        AppComponent,
        HeaderComponent
    ],
    imports: [
        AppRoutingModule,
        SharedModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, we are importing the `SharedModule` definition from the `shared/shared.module.ts` file, and we are then importing it within the `imports` array. This means that the `AppModule` module, and any components that it declares, will automatically have access to the Angular Material modules that we have defined in our `SharedModule` module.

Note once again that if you get lots of errors from the Angular compiler following these changes, remember to restart the `npm start` process.

An Angular application

Now that we have Angular set up, and have imported the Angular Material modules, we can focus on building an application. The application that we are going to build will be focused on a user logging in and logging out of our application. A screenshot of the elements in this application is as follows:

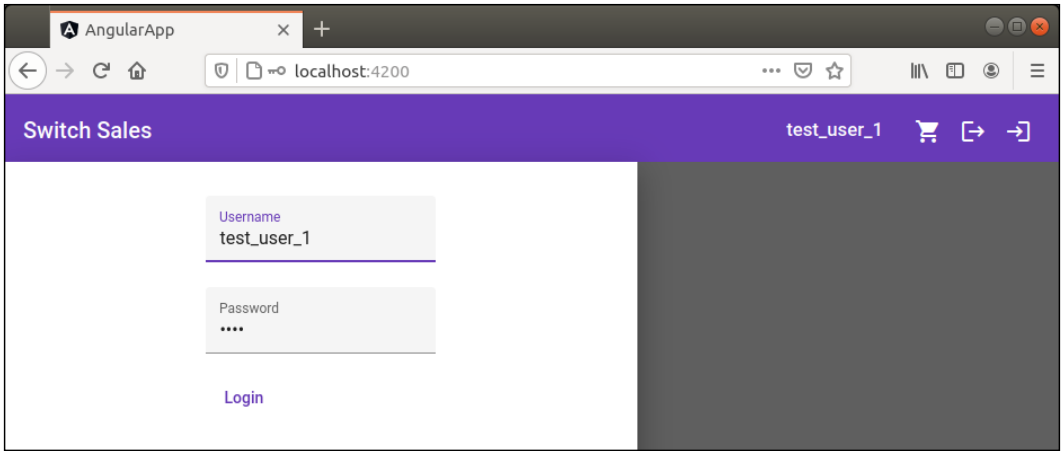


Figure 11.4: A screenshot of the elements of our Angular application

Here, we can see that we have a toolbar at the top of the page, with the name of our application called `"Switch Sales"`. On the right of the toolbar, we have the logged-in username and a couple of icons. The shopping cart icon will be used to access a user's shopping cart, and we then have a logout button and a login button. Note that in the final version of this toolbar implementation, we will hide or show these buttons based on whether the user is actually logged in.

We also have a panel that slides out from the left when a user clicks on the **Login** button, which is the login form itself. This login form has a **Username** entry field and a **Password** entry field, along with the **Login** button itself.

We will be using four main components in this application, as follows:

- `AppComponent` houses the entire page itself, and is responsible for any major changes to the body of the HTML page.
- `HeaderComponent` is responsible for rendering the header panel at the top of the page.
- `UserDetailsComponent` actually sits inside the `HeaderComponent`, and provides the username and buttons on the top right of the header panel. This component must notify other page elements when a user has clicked on any of the buttons that are available.

- `LoginComponent` is responsible for presenting the login form, and capturing the input values for the **Username** and **Password**, and for reacting to the **Login** button event click.

Angular DOM events

Our application already has an app component and a header component that are rendered to the DOM using the `<app-root>` and `<app-header>` HTML tags. Let's now create a user-details component in the same manner, and see how we can render material icons and then trap the DOM event when a user clicks on one of these icons. We will start by creating the component using the Angular CLI as follows:

```
ng generate component user-details
```

Here, we have specified that the Angular CLI should generate a component named `user-details`. The files that are generated for us are output to the console, as follows:

```
CREATE src/app/user-details/user-details.component.scss (0 bytes)
CREATE src/app/user-details/user-details.component.html (27 bytes)
CREATE src/app/user-details/user-details.component.spec.ts (662 bytes)
CREATE src/app/user-details/user-details.component.ts (299 bytes)
UPDATE src/app/app.module.ts (808 bytes)
```

Here, we can see that the Angular CLI has generated the four files that we will need for our component, and has also updated the `app.module.ts` file to include our new component in the `declarations` section.

Let's now include this component in our header component by adding the following to the `header.component.html` file:

```
<mat-toolbar color="primary">
    <span>Switch Sales</span>
    <span class="example-spacer"></span>
    <app-user-details></app-user-details>
</mat-toolbar>
```

Here, we have included the `<app-user-details>` HTML tag in the HTML for the header component. This means that the user-details component is a child of the header component, as rendering the header component will also render the user-details component. We will require a little CSS in the `header.component.scss` file as follows:

```
.example-spacer {
    flex: 1 1 auto;
  }
```

Here, we have added the `.example-spacer` style, which will push the user-details component to the right of the screen. Our application toolbar now looks like this:

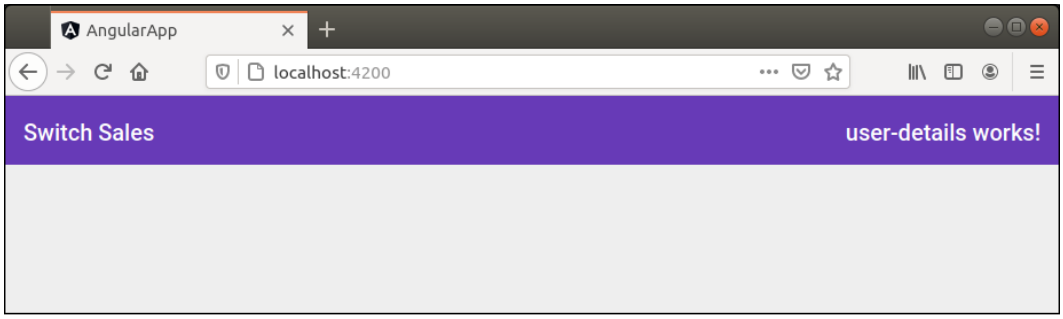


Figure 11.5: The header component rendering the child user-details component

Here, we can see that the user-details component is being rendered within the header component. Let's now add a few elements to the `user-details.component.html` file as follows:

```
<span class="username-span">{{loggedInUserName}}</span>

<button mat-icon-button color="white">
    <mat-icon>shopping_cart</mat-icon>
</button>

<button mat-icon-button color="white">
    <mat-icon>logout</mat-icon>
</button>

<button mat-icon-button color="white" (click)="onLoginClicked()">
    <mat-icon>login</mat-icon>
</button>
```

Here, we have added a span with the class `username-span`, and are then injecting the value of the property named `loggedInUserName` of the `UserDetailsComponent` class into the DOM, using the double curly braces syntax, that is, `{{loggedInUserName}}`. Angular will take care of updating the DOM value for us whenever it detects a change in the property of the class.

We have then created three buttons, which are using the Material Icon module and the `<mat-icon>` tag. Note that the last button has an event handler named `(click)`, which is set to the value of `onLoginClicked()`. This is how Angular defines the link between a DOM event (the clicking of the button), and the function to call when this occurs (the `onLoginClicked` function).

Let's now update our `UserDetailsComponent` class as follows:

```
import { Component, OnInit } from '@angular/core';

@Component({
    selector: 'app-user-details',
    templateUrl: './user-details.component.html',
    styleUrls: ['./user-details.component.scss']
})
export class UserDetailsComponent implements OnInit {

    loggedInUserName: string = "logged_in_user";

    constructor() { }

    ngOnInit(): void {
    }

    onLoginClicked() {
        console.log(`UserDetailsComponent : onLoginClicked()`);
    }
}
```

There are two important changes that we have made to this file. Firstly, we have created a property on the class named `loggedInUserName`, which is of type string, and has been set to the value `"logged_in_user"`. This property is what will be rendered to the DOM when the HTML template specifies the property `{{loggedInUserName}}`. The second change to this component is the `onLoginClicked` function, which logs a message to the console. This is the function that will be invoked when the user clicks on the login icon, as specified in the HTML template as `(click)="onLoginClicked()"`.

With these changes in place, our header panel now includes the icons that we specified, and will react to the DOM event and call the `onLoginClicked` function, as seen in the following screenshot:

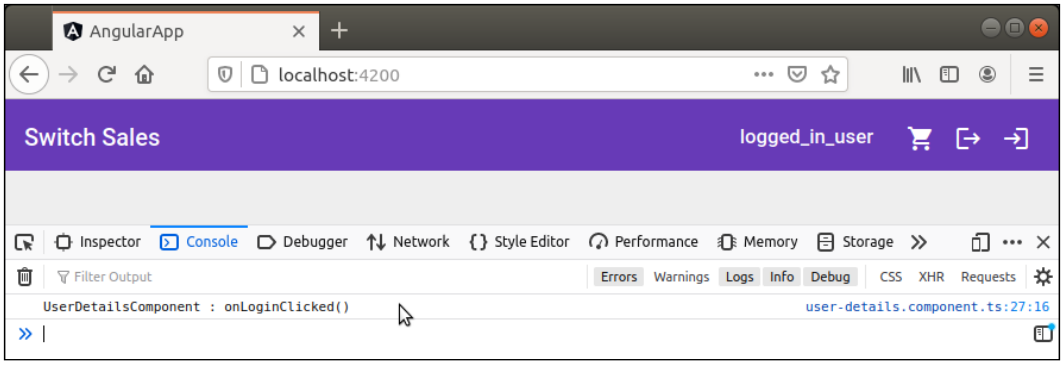


Figure 11.6: Updated user details component showing buttons and console log for the DOM clicked event

Here, we can see the name of the logged in user and the buttons that we created in our HTML template. Note that the screenshot also shows the console log message that occurs when we click on the login button.

Angular EventEmitter

When a user clicks on the login button, we will need to show the login panel, which will allow the user to enter their username and password. The event handler that traps the click event on the login button, however, is a part of the user-details component that renders the buttons onto the screen. Angular provides an `EventEmitter` class, which allows child components to send messages to their parents. Let's explore this mechanism by modifying our `UserDetailsComponent` class as follows:

```
import { EventEmitter, Output } from '@angular/core';
export class UserDetailsComponent implements OnInit {

    .. existing code

    @Output() notify = new EventEmitter();

    onLoginClicked() {
        console.log(`UserDetailsComponent : onLoginClicked()`);
        this.notify.emit("UserDetailsComponent : emit value");
    }
}
```

Here, we have added a class property named `notify` that is a new instance of the `EventEmitter` class. Note that we have also used a class property decorator named `@Output()` to mark the `notify` property as an output property.

We have also updated our `onLoginClicked` function to call the `emit` function of the `notify` property. This `emit` function takes a single parameter of type `any`. In this case, we have supplied a single string argument.

Now that we are emitting an event, we will need to consume this event, and do something with it. Angular provides the `(notify)` HTML template event, which is available on the parent component of the component that is emitting the event. This means that we will need to modify the `header.component.html` file to register for this event, and also modify the `header.component.ts` file to provide an event handler. Let's update the HTML template in the `header.component.html` file as follows:

```
<mat-toolbar color="primary">
    <span>Board Sales</span>
    <span class="example-spacer"></span>
    <app-user-details (notify)="onUserDetailsEvent($event)">

    </app-user-details>
</mat-toolbar>
```

Here, we have added a `(notify)="onUserDetailsEvent($event)"` attribute to the `<app-user-details>` HTML tag. There are two points to note about this attribute. Firstly, the `(notify)` attribute specifies the function that will be called when this event is fired by the child component. This means that we will need to create a function named `onUserDetailsEvent` on our `HeaderComponent` class to process the event. The second thing to notice is that the argument for the `onUserDetailsEvent` function is prefixed by a `$` symbol, and is `$event`, instead of just `event`. This syntax is necessary in order to attach the child event correctly.

With our interest in the event registered through the HTML template, we can now write an event hander as follows:

```
export class HeaderComponent implements OnInit {

    constructor() { }

    ngOnInit(): void {
    }

    onUserDetailsEvent(event: any) {
```

```
        console.log(`event : ${event}`)
    }

}
```

Here, we have added a class method named `onUserDetailsEvent`, which will be called when the event is fired. Note that the type of the event parameter is of type any, which is not ideal. Also, we do not need to name this parameter with a `$` symbol as a prefix.

With these modifications in place, when we click on the login button on the user-details component, our event is received and processed by the parent header component.

Unfortunately, there are a number of drawbacks to using this event "bubbling" mechanism. Firstly, if we need to send a message to a component other than the immediate parent component, then we need to have a "chain" of `EventEmitter` classes, where an event received by a parent component is emitted to the parent's parent component, and so on, until the correct component has been notified.

The second drawback to this event mechanism is that events are of type any by nature. This does not easily allow us to ensure that messages with a specific structure are handled correctly. We can, however, make use of Angular services to help restructure our event mechanism a little better.

Angular services

A better solution to our event mechanism is to use the Domain Events Design Pattern, which essentially allows various classes in our domain to either register their interest in an event, or to generate domain events. This means that any class within our application can generate an event, and any class within our application can respond to an event. As an example of this pattern, let's assume that a user has clicked a button to add an item to their shopping cart. The class that handles this event can then broadcast to any interested listeners that an item has been added to the cart. The class that is responsible for the actual shopping cart would be interested in this event, and a class that is responsible for showing the shopping cart icon in a toolbar would also be interested in this event.

The basic structure of the Domain Events Design Pattern is for a single event bus to be created that all classes have access to. Classes can then either register their interest in an event or can broadcast an event on the event bus to any interested parties.

Angular provides what are known as services to provide supporting functionality across components. A service is essentially a singleton instance that is available to all classes within an application.

In *Chapter 9*, *Using Observables to Transform Data*, we built a `BroadcastService` class that acted as an event bus, and enabled events to be raised, and for interested parties to register for these events. The `BroadcastService` is just what we need within our Angular application to work with domain events.

We can create an Angular service using the Angular CLI as follows:

```
ng generate service services/broadcast
```

Here, we are using the generate argument of the Angular CLI to generate a service named `broadcast`, which will be created in a new directory named `services`. We will reuse the implementation of the `BroadcastService`, which we created in *Chapter 9*, *Using Observables to Transform Data*, which we built when working with Observable Subjects, as follows:

```
export interface IBroadcastEvent {
    key: EventKeys;
    data?: any;
}

export enum EventKeys {
    ALL = "all-events",
    LOGIN_BUTTON_CLICKED = "login_button_clicked",
    USER_LOGIN_EVENT = "user_login_event"
}

@Injectable({
    providedIn: 'root'
})
export class BroadcastService {

    ... existing implementation
}
```

Here, we have exported the `IBroadcastEvent` interface, which defines a domain event. We have also updated the `EventKeys` enum to contain two enum values, which are `LOGIN_BUTTON_CLICKED`, which will correlate to the user clicking on the **Login** button, and the `USER_LOGIN_EVENT`, which will correlate to the user completing the login form and then pressing the **Submit** button.

The `BroadcastService` class implementation is unchanged, but note that it has been prefixed by a decorator named `@Injectable`. This decorator marks our `BroadcastService` as being injectable by Angular's Dependency Injection framework, which we will discuss a little later. Note that Angular uses RxJS too, so it will be installed by default for an Angular application.

As we have seen with Angular components, we will need to register this service somehow, and ensure that all components or classes have access to it. The place to do this is in the `app.module.ts` file, as follows:

```
@NgModule({
    declarations: [
        AppComponent,
        HeaderComponent,
        UserDetailsComponent
    ],
    imports: [
        AppRoutingModule,
        SharedModule
    ],
    providers: [
        BroadcastService
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, we have included `BroadcastService` under the `providers` array for the `AppModule`. This means that any component within our application will have access to the `BroadcastService`. Note, too, that Angular Services are essentially a singleton instance, meaning that one instance of the service will be created at startup.

Angular Dependency Injection

Angular uses a technique known as **Dependency Injection** (**DI**) to provide services to components. This means that a class can request an instance of a service that it depends on, and Angular will resolve this dependency at runtime. DI in Angular works through the constructor function of a component, as follows:

```
export class UserDetailsComponent implements OnInit {

    loggedInUserName: string = "logged_in_user";

    constructor(private broadcastService: BroadcastService) {
    }

    ngOnInit(): void {
    }
```

```
    @Output() notify = new EventEmitter();

    onLoginClicked() {
        console.log(`UserDetailsComponent : onLoginClicked()`);
        this.notify.emit("UserDetailsComponent : emit value");
        this.broadcastService.broadcast(
            EventKeys.LOGIN_BUTTON_CLICKED,
            "UserDetailsComponent:  LOGIN_BUTTON_CLICKED"
        );
    }
}
```

Here, we have updated the `UserDetailsComponent` class in two areas. Firstly, we have added a private member variable through our constructor function named `broadcastService`, of type `BroadcastService`. By adding this private member variable into the constructor and specifying its type, the Angular DI process will take over when creating this class and pass the instance of the `BroadcastService` into this component as an argument.

The second change we have made to this class is to call the `broadcast` function on our local instance of the `BroadcastService` and send an `EventKeys.LOGIN_BUTTON_CLICKED` event to the event bus. Note that we have included a string value as the data argument, just for debugging purposes.

The DI framework that Angular uses is very easy to use and becomes second nature when dealing with services. The other benefit to using DI is when we build unit tests for components. Within a unit test, we can easily create a mock implementation of a service, and pass this in as the instance of the service that we wish to use. We will discuss this technique a bit later in this chapter.

Child components

Now that we have defined and broadcast a domain event, we can react to it. Remember that the component that is generating the `LOGIN_BUTTON_CLICKED` event is the `UserDetailsComponent`, which is a child of the `HeaderComponent`. The component that must react to this domain event is actually the `AppComponent`, which is in charge of the entire page.

Our `AppComponent` will use an Angular Material Sidenav component to either show or hide the login panel. The Sidenav control can be manipulated programmatically, which means that we can call methods on the Sidenav control itself.

Angular provides the `@ViewChild` decorator, which allows our `AppComponent` class to access its `MatSidenav` child component. To use a child component in our code, we will need to first update our `app.component.html` file as follows:

```
<app-header></app-header>

<mat-sidenav-container class="full-height-container">
    <mat-sidenav #sidenav mode="over"
        class="content-padding"
        [fixedInViewport]="true"
        [fixedTopGap]="60"
        [fixedBottomGap]="0"
        [opened]="false">
        Login form to go here.
    </mat-sidenav>
    <mat-sidenav-content>
        <div class="content-padding">
            Main Content goes here.
        </div>

    </mat-sidenav-content>
</mat-sidenav-container>

<router-outlet></router-outlet>
```

Here, we have added a `<mat-sidenav-container>` element, and within this, a `<mat-sidenav>` and a `<mat-sidenav-content>` element. These elements work together to provide a side navigation panel that will slide in from the left. Note that the `<mat-sidenav>` element is the side navigation panel itself, while `<mat-sidenav-content>` holds the main content of the page.

The important thing to note in this code snippet is that we have given the `<mat-sidenav>` tag an attribute of `#sidenav`. This gives this component an `"id"` attribute that Angular uses to tie controls defined in the HTML template back to objects in our class definition. We can now obtain a reference to this control in our component as follows:

```
export class AppComponent {
    title = 'angular-app';
    @ViewChild("sidenav") sidenav: MatSidenav | null = null;

    constructor(broadCastService: BroadcastService) {
```

```
        _.bindAll(this, "onLoginClicked");
        broadCastService.on(EventKeys.LOGIN_BUTTON_CLICKED)
            .subscribe(this.onLoginClicked);
    }

    onLoginClicked(event: string) {
        console.log(`AppComponent received : ${event}`);

        this.sidenav?.open();
    }
}
```

Here, we have added a class member variable named `sidenav` that is of type `MatSidenav` or null. This variable has been decorated with the decorator `@ViewChild`. This decorator takes a single string argument of `"sidenav"`, which must match the `#sidenav` attribute of the HTML template. Note that this child component is set to null when the class is first constructed and will become available once the HTML view has been rendered.

We have updated our constructor function for the `AppComponent` class and again used Angular's DI framework to gain a handle on the `BroadcastService` service. We are also using the `bindAll` function from `underscore` to ensure that the `onLoginClicked` function is called, with `this` bound correctly to the class instance. We then attach the `onLoginClicked` function to the event `LOGIN_BUTTON_CLICKED`. Note that we will need to install `underscore` and `@types/underscore` here.

Our `onLoginClicked` function logs a message to the console, and then uses the `sidenav` member variable to programmatically control the `MatSidenav` instance. Note that we are calling the `open` function on the `MatSidenav` class, which will slide the control in from the left.

We can control the width of the `mat-sidenav` component using some CSS in the `app.component.scss` file, as follows:

```
mat-sidenav {
    width: 60%;
}
```

Here, we have set the width of the `mat-sidenav` component to `60` percent of the screen width.

If we fire up our browser, we will now be able to click on the login button on the user-details component, and the `UserDetailsComponent` class will then emit a domain event. The `AppComponent` class will receive this event, and open the side navigation panel, as shown in the following screenshot:

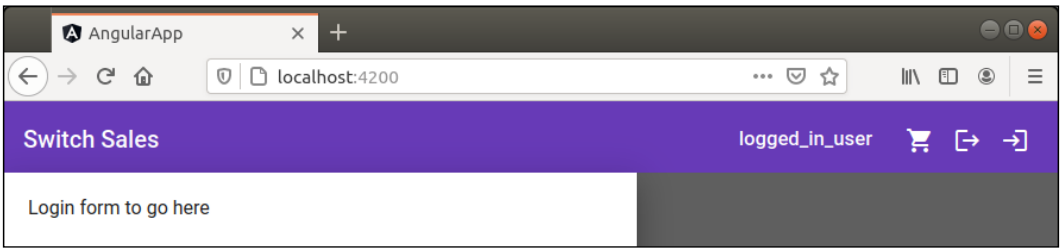


Figure 11.7: Angular application showing the sidebar once the login button is clicked

Our Angular application is starting to take shape. We have a main header panel, a user-details panel, and now have a sidebar that is shown when the correct domain event is fired. We have also been sticking to the Single Responsibility pattern, which states that each component is to have a single responsibility. So, do one thing and do it well. We have also implemented the Domain Events pattern, which allows a control to fire a domain event to any interested component, and for that component to react to the event in a meaningful manner. In the next section of this chapter, we will take a look at how Angular deals with user input, through the use of forms.

Angular forms

When building single-page applications, we will often need a user to input data of some sort using a form. Angular uses a two-way data binding process to bind values that are entered on a form to variables within a component itself. This process is two-way, as Angular will take care of synchronizing what is shown in the DOM with the member variables of a component. So if we change a value programmatically within our code, this value will be updated in the HTML. Similarly, when a user modifies the HTML values, these values will automatically update our class member variables.

Angular actually has two different methods of creating two-way data binding forms. The first method is named template forms, which allows us to bind an input control directly to a property on our class, as follows:

```
<input type="text" [(ngModel)]="name" />
```

Here, we have an `input` control within our HTML template, which is using the `[(ngModel)]` attribute with a value of `"name"`. This tells the Angular runtime that it must synchronize the input value of this control with a property named `"name"` on the component class, as follows:

```
@Component({
... selectors
})
export class SampleComponent  {

    name: string = "";
}
```

Here, we have an Angular component named `SampleComponent` with a property named `name`. Again, if we set this value within our component class, then the value will be shown within the HTML template, and if we modify the value from the HTML template, this property will automatically be updated.

While template forms are very simple to implement, they do have limitations. As an example, we may want to show or hide certain form fields depending on a selection that the user has made. We may also wish to provide extra validation on the input, say to ensure that the user has entered a valid price in the range 0.01 to 10.00. In these cases, template forms do not provide the flexibility that we need for advanced forms.

Reactive forms

To overcome the limitations of template forms, Angular has introduced the concept of Reactive forms. Reactive forms are forms that are defined in code, rather than being defined as individually bound member variables. Before we define a form, however, let's create a login component, and embed it in our `<mat-sidenav>` panel as follows:

```
ng generate component login
```

Here, we are using the Angular CLI to generate a login component, and automatically add it to our `AppModule` module. We can now include this form in the `<mat-sidenav>` panel by updating the `app.component.html` template as follows:

```
<mat-sidenav #sidenav mode="over"
    class="content-padding"
    [fixedInViewport]="true"
    [fixedTopGap]="60"
    [fixedBottomGap]="0"
    [opened]="false">
    <app-login></app-login>

</mat-sidenav>
```

Here, we have replaced the text `"Login form to go here"` with the `<app-login>` tag, which will now render the login component.

To use Reactive forms in our `LoginComponent`, we will need to make a few changes as follows:

```
export class LoginComponent implements OnInit {

    loginForm: FormGroup | null = null;

    constructor(
        private formBuilder: FormBuilder,
        private broadcastService: BroadcastService
    ) { }

    ngOnInit(): void {
        this.buildForm();
    }

    buildForm() {
        let form = {
            username: new FormControl({}, Validators.required),
            password: new FormControl({}, Validators.required)
        }

        let formState = {
            username: {
                value: "",
                disabled: false
            },
            password: {
                value: "",
                disabled: false
            }
        }

        this.loginForm = this.formBuilder.group(form);
        this.loginForm.reset(formState);
    }

    isFormValid() {
        return this.loginForm?.valid;
    }
}
```

The first thing we have done is to create a member variable named `loginForm`, which is of type `FormGroup` or null. We have then used Angular's DI framework to request an instance of the `FormBuilder` class. Our `ngOnInit` function is now calling a new function on our class named `buildForm`.

The `buildForm` function is responsible for constructing the form itself and starts by creating an object named `form` that has a property for each of the form fields we need, which in this case is `username` and `password`. Each form field property is a new instance of a `FormControl` class, which is using a blank object as the first argument, and a `Validators.required` static function as the second argument. Validators are used to validate the input of a form, and in this instance, all we have specified is that both the `username` and `password` fields are required before the form can be submitted.

We then construct an object named `formState`, which specifies what the current value for both of these form controls should be, and whether they are disabled or not. You can imagine that if a user is filling in a form that was previously saved, we would want to fill in the form with what they previously entered, and the `formState` object allows us to do this.

We then call the `group` function on the `formBuilder` instance to create our Reactive form and assign it to the local member variable named `loginForm`. We then call the `reset` function on our newly created `loginForm` form, and pass in the `formState` object. Again, the reason for separating the `formState` object from the form definition allows us to reset all values to blank, for example, if the user clicks a button named **Clear**, or **Undo**. So we create the form once, but can reset the state many times.

Finally, we have a function named `isFormValid` that is returning the value of the `valid` property of the `loginForm` itself. Remember that we have marked both the `username` and `password` fields as being required, so the form will not be valid until both fields have been filled in.

Reactive form templates

Now that we have a Reactive form defined in our class, we can bind this to our HTML template, in the `login.component.html` file, as follows:

```
<div class="login-container">
    <div *ngIf="loginForm">
        <form [formGroup]="loginForm"
            (ngSubmit)="onSubmit()">
            <mat-form-field appearance="fill">
                <mat-label>Username</mat-label>
                <input matInput
                    formControlName="username">
```

```
            </mat-form-field>
            <br>
            <mat-form-field appearance="fill">
                <mat-label>Password</mat-label>
                <input matInput type="password"
                    formControlName="password">
            </mat-form-field>
            <br>
            <button mat-button color="primary"
                type="submit"
                [disabled]="!isFormValid()">
                Login
            </button>
        </form>
    </div>
</div>
```

Here, we have the HTML template for our login form, which has four interesting things to note.

Firstly, we are using an Angular directive, which is `*ngIf="loginForm"`. The `*ngIf` is known as an Angular `if` directive, and it is used to only show the containing div if the result of the expression is true. In this case, the `if` directive will only render the div if the class member variable, `loginForm`, evaluates to true, and is not null or undefined. Remember that Angular must load our component class, then parse the component HTML template and bind values before updating the DOM. This process can take some time, and the net effect is that Angular may render the template to the DOM before the `loginForm` class member variable has been created.

The second thing to note in this HTML snippet is that we have defined a `<form>` tag and set the `[formGroup]` attribute to the `"loginForm"` class member variable. This will set up the two-way data binding between our `loginForm` member variable and the HTML elements on the screen. This `<form>` element also has an event named `(ngSubmit)`, which will call the class member function named `onSubmit` when the form itself has been submitted.

The third item of interest is that we have `<input>` tags in the HTML template that have a `formControlName` attribute set to the name of the form controls that we set up in our `buildForm` function. Again, this attribute tells Angular which HTML element matches the `FormControl` instances within our form definition.

The fourth and final item of interest in this HTML template is that we have a submit button whose `disabled` attribute is tied back to the `isFormValid()` function on our class. This means that this button will be disabled if the form itself is not valid.

Reading form values

With our Reactive form defined in our class, and the Reactive form template built in our HTML file, we can now turn our attention to reading the values out of the form once the user hits the **Login** button. Remember that the form itself has specified the `onSubmit` function to be called when the form has been submitted. This function is as follows:

```
onSubmit() {
    console.log(`onSubmit: username :
        ${this.loginForm?.value.username}`);
    console.log(`onSubmit: password :
        ${this.loginForm?.value.password}`);

    this.broadcastService.broadcast(
        EventKeys.USER_LOGIN_EVENT,
        this.loginForm?.value.username);
}
```

Here, our `onSubmit` function is logging the values entered by the user in both the `username` and `password` input controls to the console. Note how we access these values, through the `loginForm` instance. This instance has a `value` property, and a corresponding property for each control within the form.

If we run our application in the browser, and enter some values, we will see these console logs in our log output as follows:

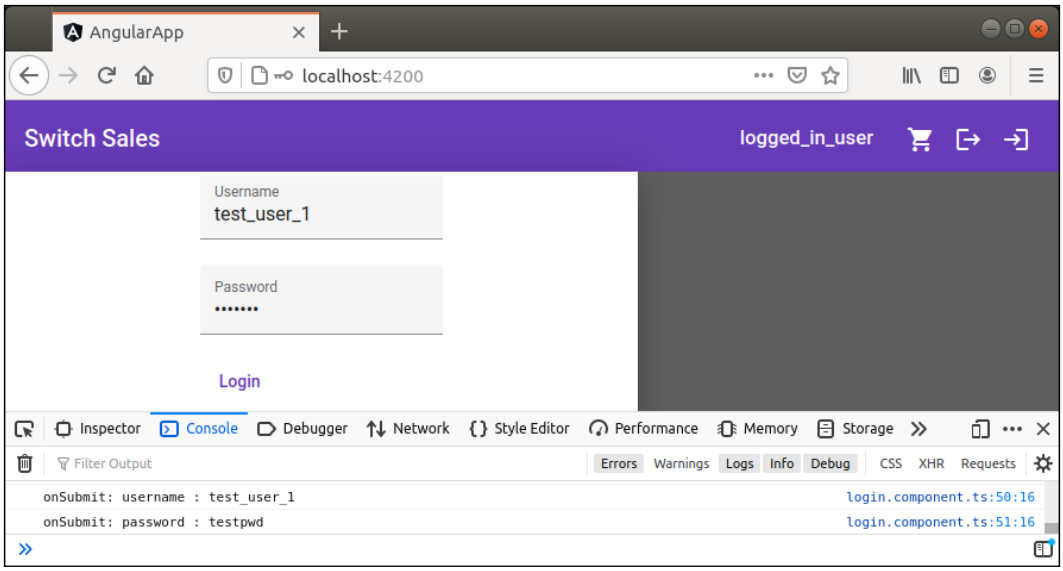


Figure 11.8: Angular application showing logs of form values read from a FormGroup

Note that in our `onSubmit` function, we are broadcasting a domain event with the key `USER_LOGIN_EVENT`, and the event's `data` property is set to the value that the user entered in the `username` form control. We will hook up some consumers for this event to round out our application a little later, but first let's take a look at how we can unit test forms.

Angular unit testing

Angular provides a rich set of unit testing components that enable us to unit test all aspects of our application. For the login component, we should write a set of unit tests that cover at least the basic logic flow of the form. We really need to test the following behaviors of the login component:

- When the form is first presented, both the username and password fields should be blank.
- If only the username field is filled in, then the **Login** button should be disabled.
- If both the username and password fields are filled in, then the **Login** button should be enabled.
- When the **Login** button is clicked, a domain event should be broadcast with the correct event key and data packet.

The Angular CLI will automatically generate a `.spec.ts` file for each component or service that we create using `ng generate`. Let's now take a look at the default unit test that was generated for our component as follows:

```
describe('LoginComponent', () => {
    let component: LoginComponent;
    let fixture: ComponentFixture<LoginComponent>;

    beforeEach(async () => {
        await TestBed.configureTestingModule({
            declarations: [
                LoginComponent
            ]
        }).compileComponents();
    });

    beforeEach(() => {
        fixture = TestBed.createComponent(LoginComponent);
        component = fixture.componentInstance;
```

```
        fixture.detectChanges();
    });

    it('should create', () => {
        expect(component).toBeTruthy();
    });
});
```

Here, we can see the basic structure of an Angular unit test. The test suite starts by defining a variable named `component`, which will house the instance of our `LoginComponent` class. The test then defines a variable named `fixture`, which is of type `ComponentFixture<LoginComponent>`. The `fixture` instance will house the rendered HTML template for the login component, and can be used to query or manipulate the DOM. Think of the `component` variable as the class instance, and the `fixture` variable as the HTML.

Our test then has a `beforeEach` function that has been marked as `async`. Within this `async` function, we are awaiting a call to the function `compileComponents`, which is returned by a call to the static function, `TestBed.configureTestingModule`. The `configureTestingModule` function is used to provide any class instances that are required by the DI framework, and to declare any imports or services that the class needs, similar to what we have seen with our `AppModule` class.

We then have another `beforeEach` function that sets the value of the `fixture` and `component` variables for use within the test. The `beforeEach` function then calls the `fixture.detectChanges` function, which will simulate the Angular change detection process, and bind any two-way inputs that the class and HTML templates need. Once this `beforeEach` function has completed, we are ready to start our test.

This test suite has a single test named `"should create"` and is just checking whether the `LoginComponent` was created correctly. We can now run our Angular tests from the command line as follows:

```
npm test
```

Here, we are invoking the `"test"` script that is included in our `package.json` file, which, under the hood, will run `ng test` and output the results to the console. Note that this also runs in watch mode, and as such, will re-run all tests when changes are detected.

Unfortunately, running this automatically generated test will produce the following error:

```
NullInjectorError: No provider for FormBuilder!
```

This error is being generated due to the Angular DI framework not having an instance of the `FormBuilder` class to inject into our `LoginComponent` constructor. We will need to make sure that each of the DI injected classes are also available within the confines of this unit test, as follows:

```
beforeEach(async () => {
    await TestBed.configureTestingModule({
        imports: [
            SharedModule
        ],
        declarations: [
            LoginComponent
        ],
        providers: [
            FormBuilder,
            BroadcastService
        ]
    }).compileComponents();
});
```

Here, we have updated the configuration object that we are passing into the `configureTestingModule` function in two places. Firstly, we have added an `imports` array, and within it, have specified the `SharedModule` as an import. Remember that the `SharedModule` module exports all of the Angular Material components that we may be using within our component. Secondly, we have added a `providers` array, and specified the services that we will make available to this unit test, which are the `FormBuilder` and `BroadcastService`. This configuration object is very similar to an Angular module, which also needs to provide the same sort of configuration.

With this configuration in place, our `LoginComponent` can be created correctly, and our test now passes.

Unit testing forms

Now that the `LoginComponent` is being created correctly, and has all of the services that it needs, we can focus on testing the form itself. Let's start with a test for the initial state of the form, as follows:

```
it('should set form fields correctly on startup', () => {
    expect(component.loginForm).toBeDefined();
    expect(component.loginForm?.value.username).toEqual("");
    expect(component.loginForm?.value.password).toEqual("");
});
```

Here, we have a test that is checking whether the form is created correctly when the form is first loaded. Remember that when we call the `fixture.detectChanges` function, this will trigger the Angular change detection routine, which will ensure that our component has been rendered to the DOM. Part of this change detection will call the `ngOnInit` function on our component, which in turn calls the `buildForm` function to set up the form.

Our test is making sure that the `loginForm` member variable has been initialized and is then checking that the `username` and `password` form values have been set to a blank string.

Our next test will fill in values for these form controls and check whether the **Login** button has been enabled, as follows:

```
it('should set form validity correctly', () => {
    expect(component.isFormValid()).toBeFalse();
    component.loginForm?.controls["username"]
        .setValue("test_username");
    expect(component.isFormValid()).toBeFalse();
    component.loginForm?.controls["password"]
        .setValue("test_password");
    expect(component.isFormValid()).toBeTrue();
});
```

Here, our test starts by checking whether the `isFormValid` function returns false, which in turn, means that our **Login** button will be disabled, as the form itself is not valid. We then fill in a value of `"test_username"` for the `username` input field by calling the `setValue` function of the control on the form, which is `controls["username"].setValue`. We then check that the form is still not valid. Finally, we set the value of the `password` input field, and then check that the form is now valid.

For our final test, we would like to ensure that a domain event is raised when the user clicks on the **Login** button. In order to check that the domain event is raised, we will need to spy on the `broadcast` function of the `BroadcastService`. This means that we will need a handle on the injected instance of the `BroadcastService` within our test. This can be accomplished as follows:

```
describe('/src/app/login/login.component.ts', () => {
    let component: LoginComponent;
    let fixture: ComponentFixture<LoginComponent>;
    let broadcastService: BroadcastService;

    beforeEach(async () => {
    .. existing code
```

```
    });

    beforeEach(() => {
        fixture = TestBed.createComponent(LoginComponent);
        component = fixture.componentInstance;
        broadcastService = TestBed.inject(BroadcastService);
        spyOn(broadcastService, 'broadcast');
        fixture.detectChanges();
    });
    ... existing tests
});
```

Here, we have declared a variable named `broadcastService` of type `BroadcastService`, which will hold the instance of the `BroadcastService` service that has been injected into the test. In our `beforeEach` function, we are calling the `TestBed.inject` static function to obtain a handle on the injected instance of the `BroadcastService` service. Once we have a handle on this instance, we call the Jasmine `spyOn` function to create a spy on the `broadcast` function of the `BroadcastService` service. This mechanism can be used to obtain a handle on any one of the services that are injected by Angular's DI mechanism. We can now write a test as follows:

```
it('should broadcast an event when the Login button is clicked',
    () => {
    component.loginForm?.controls["username"]
        .setValue("test_username");
    component.loginForm?.controls["password"]
        .setValue("test_password");

    fixture.detectChanges();
    const loginButton = fixture.debugElement
        .nativeElement.querySelector("#submit_button");
    expect(loginButton.disabled).toBeFalsy();

    loginButton.click();

    expect(broadcastService.broadcast)
        .toHaveBeenCalledWith(
            EventKeys.USER_LOGIN_EVENT,
            "test_username");
});
```

Here, our test starts by filling in the values of the `username` and `password` input form fields. We then call the `fixture.detectChanges` function to trigger Angular's change detection routine, which will enable the **Login** button on the HTML template. Once the form is valid, we can access the DOM by using the `fixture.debugElement.nativeElement` property. We search the DOM using the `querySelector` function, and find the button with an `id` attribute of `"submit_button"`. We then check the disabled attribute of the button to ensure that it is not disabled.

Finally, we click the button and check to see whether the code has called the `broadcast` function on the `BroadcastService` instance with the correct event values.

Reacting to domain events

Our Angular application is almost complete. There are a few things that we still need to do in order to round it out. Firstly, once a user has completed the login form, we should close the login sidebar, and show the main content once again. Secondly, we should update the user-details component to show the currently logged-in username, as well as hide the login button. These changes can all occur on the back of the domain event `USER_LOGIN_EVENT`.

To hide the sidebar, we need to react to the `USER_LOGIN_EVENT` within the app component, as it is in charge of showing and hiding the sidebar. This can easily be accomplished by updating the `app.component.ts` file as follows:

```
export class AppComponent {
    .. existing code
    constructor(broadCastService: BroadcastService) {
        _.bindAll(this, "onLoginClicked", "onLoginEvent");
        broadCastService.on(EventKeys.LOGIN_BUTTON_CLICKED)
            .subscribe(this.onLoginClicked);
        broadCastService.on(EventKeys.USER_LOGIN_EVENT)
            .subscribe(this.onLoginEvent);
    }

    .. existing code

    onLoginEvent() {
        this.sidenav?.close();
    }
}
```

Here, we have subscribed to the `USER_LOGIN_EVENT` and configured a function named `onLoginEvent` as an event handler. Within this `onLoginEvent` function, we call the `close` function of the child component `sidenav`.

We can now turn our attention to the user-details component, and how it will react to the `USER_LOGIN_EVENT`. Let's update the `user-details.component.ts` file as follows:

```
export class UserDetailsComponent implements OnInit {
    loggedInUserName: string = "";
    isLoggedIn: boolean = false;

    constructor(private broadcastService: BroadcastService) {
        _.bindAll(this, "loginSuccessful");
        this.broadcastService.on(EventKeys.USER_LOGIN_EVENT)
            .subscribe(this.loginSuccessful);
    }

    ... existing code

    loginSuccessful(event: any): void {
        console.log(
            `UserDetailsComponent.loginSuccessful : ${event}`);
        this.loggedInUserName = event;
        this.isLoggedIn = true;
    }
    onLogoutClicked(): void {
        this.loggedInUserName = "";
        this.isLoggedIn = false;
    }
}
```

Here, we have made a few significant changes to the user-details component. Firstly, we have introduced a member variable named `isLoggedIn` to indicate whether we have a logged-in user or not. Secondly, we have subscribed to the domain event of `USER_LOGIN_EVENT`, and when this event is received, we call the `loginSuccessful` function. This function will set the value of the `loggedInUserName` variable to the username that is part of the domain event, and then we set the `isLoggedIn` member variable to true. We have also included an `onLogoutClicked` function that will reset these variables when the logout button is clicked.

With these domain event handlers in place, we can make a slight change to the `user-details.component.html` template file as follows:

```
<span *ngIf="isLoggedIn"
    class="username-span">{{loggedInUserName}}</span>

<button *ngIf="isLoggedIn" mat-icon-button color="white">
    <mat-icon>shopping_cart</mat-icon>
</button>

<button *ngIf="isLoggedIn" mat-icon-button color="white"
    (click)="onLogoutClicked()">
    <mat-icon>logout</mat-icon>
</button>

<button *ngIf="!isLoggedIn" mat-icon-button color="white"
    (click)="onLoginClicked()">
    <mat-icon>login</mat-icon>
</button>
```

Here, we have added an `*ngIf` directive to our span that shows the logged-in username, as well as the settings and logout button, based on the class member property named `isLoggedIn`. If a user has logged in, then these buttons and span will be shown, and if a user has not logged in as yet, the only button that will be shown is the login button.

Our Angular application is now complete.

Summary

In this chapter, we have explored the Angular SPA framework, and built an Angular application. We have seen how to incorporate the Angular Material set of UI controls, shown what Angular modules are used for, and built an Angular module of our own. We then discussed how to react to DOM events in Angular, such as a button click, and the event handling capabilities built into the Angular framework. We then explored Angular services, and incorporated the event bus that we built in a previous chapter into an Angular application, using Angular's DI capabilities. Finally, we explored the two types of Angular forms, and incorporated an Angular Reactive form within our application. In the next chapter, we will explore another popular TypeScript-compatible SPA framework – React.

12
React

React is a JavaScript framework originally developed by Facebook, and is widely used and very popular within the JavaScript community. React uses a specific inline syntax for combining HTML templates and JavaScript code in the same file, which is named JSX, or JavaScript XML. Interestingly, React is only geared toward rendering components to the DOM, and does not provide an out-of-the-box framework for things like routing. For this reason, React is often combined with other frameworks, such as Redux for state management, or React Router for routing capabilities. That said, React is an extremely efficient rendering engine, and is generally a lot faster than other frameworks in terms of rendering speed. There are some, but not many, frameworks that render faster than React, the most notable of these being Backbone.

Along with its JSX syntax for combining HTML and JavaScript, React is actually a very simple framework to understand and use. Its primary principle is that rendering a particular area of the DOM should be handled by a single React component. React uses what is known as a virtual DOM in order to calculate what updates need to be made to the DOM, so it is able to render only those elements that have changed. This means that if we were to render a collection of items to the DOM, then each element of the collection should become a React component, and the collection itself should be a React component. If only a single item in the collection is updated, then React will not need to re-render the entire collection; it will only need to render the updated item.

In this chapter, we will build a React application that will render a collection of products to the screen. We will then show how we can use DOM events within React to co-ordinate what happens when a user selects a particular product.

Finally, we will build a form within React, and show how to populate and interrogate form controls as and when we need to. Specifically, we will cover the following topics:

- Setting up a React application
- JSX
- React props
- React event handling
- React state
- Building a React application
- Using React forms

Introduction to React

In this section of the chapter, we will set up a React development environment, and explore the JSX syntax that React uses. We will also discuss how parent components pass information to child components through properties, or props, and how a component can track its own properties through what is known as React state.

React setup

Setting up a React development environment has, in the past, been a rather time-consuming and delicate operation. Getting React to work properly with TypeScript in the past only seemed to add to this complexity. The React community, however, has introduced a set of handy utilities to make setting up a React development environment a breeze.

We can create a fully fledged React application, complete with unit testing and TypeScript integration, by running the following command:

```
npx create-react-app product-list --template typescript
```

Here, we are running the `create-react-app` script directly with `npx`, instead of installing it globally. We have provided the name of the directory for the React app, which is `product-list`, and we have also specified that we wish to use the TypeScript template. After a few minutes, the `create-react-app` script will complete. To start our newly created React application, we will need to change into the `product-list` directory, and then start the React development environment as follows:

```
cd product-list
npm start
```

Here, we have started the React development environment in the `product-list` directory. This will compile our application, start a development server, and open a browser, as shown in the following screenshot:

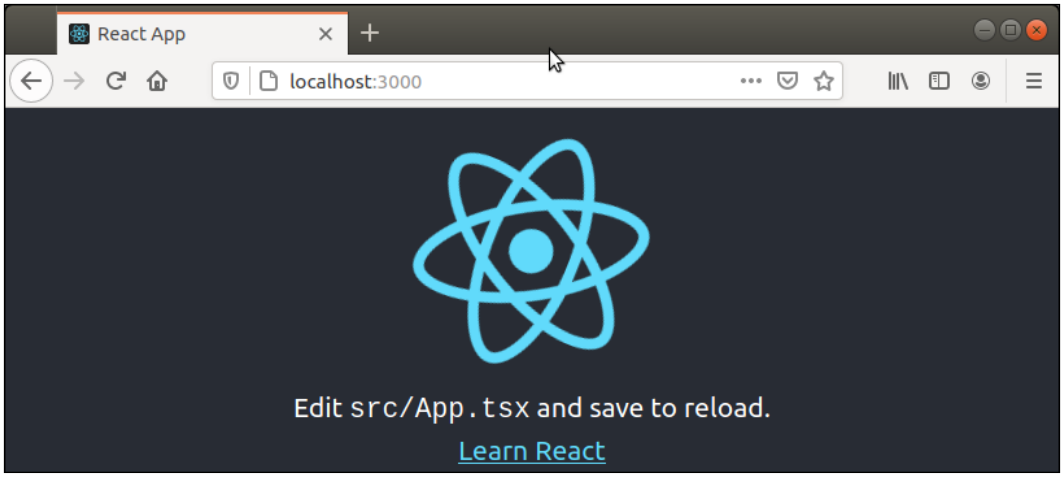


Figure 12.1: A default React app running in a development environment

Here, we can see that React will start a local server on port `3000`, and open a browser for us. Note that React will also run in watch mode, which means that any changes to our source files will automatically trigger a compilation step, and will also cause the browser to reload the application.

JSX

As mentioned earlier, React uses a specific syntax for embedding what looks like HTML directly into our JavaScript code. This syntax is known as JavaScript XML, or simply JSX. TypeScript has supported the JSX syntax from very early releases, with one main caveat. If you need to use JSX syntax, then your TypeScript file should be named `.tsx`, instead of the normal `.ts`. With this in mind, let's modify the generated `src/App.tsx` file and replace the contents with the following:

```
import React from 'react';
import './App.css';

class App extends React.Component {
    render() {
        return <div>Hello JSX</div>
    }
}
export default App;
```

Here, we have the simplest example of a React component named `App`, and how it uses JSX syntax. We have defined a class named `App` that extends the class `React.Component`. This class has a single `render` function that returns JSX, which is a `<div>` HTML element containing the text `"Hello JSX"`. Note how our HTML template is embedded within the `render` method. The last line of this code snippet exports the `App` class by default. Reloading our browser now will render the screen as follows:

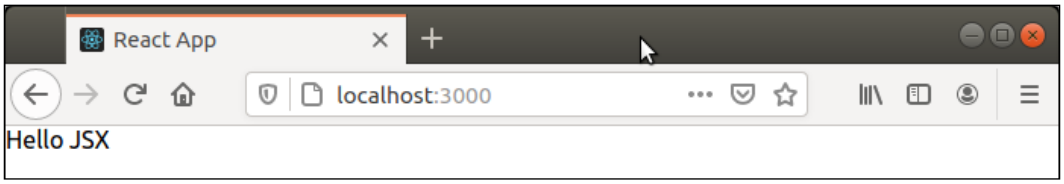


Figure 12.2: A React application rendering a JSX div to the screen

There are a few things to note about the `render` function, and the structure of React components. Firstly, a `render` function can only return a single outer HTML element, or a user-defined element. In other words, every HTML element rendered must be a child of a single parent element. This feeds into the design of React components, where the over-arching principle is that each child element should be its own React component, to ensure that we are only updating the portion of the DOM that we need to. We will see examples of this design philosophy as we build out our React application.

Secondly, the outer HTML element must be on the same line as the `return` statement, unless we wrap the JSX in parentheses, or `(` and `)`, as follows:

```
render() {
    return
    <div>Hello JSX</div>
    // this will cause an error
}
```

Here, our outer `<div>` HTML element is not on the same line as the `return` statement, and will cause a compilation error as follows:

```
Error: App(...): Nothing was returned from render. This usually means a
return statement is missing.
```

The solution to this error, and a good coding practice, is to wrap all returned JSX within parentheses, as follows:

```
render() {
    return (
        <div>Hello JSX</div>
    )
}
```

Here, we have wrapped our JSX within parentheses, that is, an opening `(` and a closing `)`, which resolves the error.

JSX and logic

The syntax of JSX allows us to embed normal TypeScript logic within our `render` functions, but this logic must be performed outside of the `return` statement. This is best described by taking a look at a `render` function that uses logic to determine what to display, as follows:

```
class SampleApp extends React.Component {
    render() {
        let item = null;
        if (Math.random() < 0.5) {
            item = (
                <p>Random less than 0.5</p>
            )
        } else {
            item = (
                <p>Random is greater than 0.5.</p>
            )
        }

        return (
            <div>
                {item}
            </div>
        )
    }
}
```

Here, we have a class named `SampleApp`, which is a React component. Within our `render` function, we define a variable named `item` that is set to null. We then have an `if` statement that is checking whether the value of the call to `Math.random()` is less than `0.5`. If it is, we set the value of the `item` variable to an HTML `<p>` element with the text `"Random less than 0.5"`. If it is not, we set the `item` variable to an HTML `<p>` element with the text `"Random is greater than 0.5."`.

Our `render` function then returns a top-level `<div>` element, and is using curly braces around the `item` variable, that is, `{item}`, to embed the HTML contained in the `item` variable within the resulting JSX. This syntax is how we use the output of logic in our React component to drive the rendered HTML.

React props

When we create a React component, we are able to give it properties that are handed down from the parent component to the child component. These properties can include both display properties and event handlers. As an example of this, let's put a button on the screen and see how to both handle the `onClick` DOM event within the component itself and how to trigger an event on the parent component.

Before we do this, however, let's install the Material-UI library built for React, which will give us access to a wide range of standard components, such as buttons and input controls, as follows:

```
npm install @material-ui/core
npm install @material-ui/icons
```

Here, we are installing the React Material-UI core package, as well as the icons package.

We will create a new React component in a file named `MyButton.tsx`, as follows:

```
import React from "react";

export class MyButton extends React.Component {
    render() {
        return (
            <Button color="primary">Click me</Button>
        )
    }
}
```

Here, we have created a new React component named `MyButton` that extends `React.Component` and has a single `render` function. Within the `render` function, we are adding a React Material-UI component named `Button`, which will render a button to the screen with the text `"Click me"`. Note that we have also exported the `MyButton` class so that we can import it in the `App.tsx` file as follows:

```
import { MyButton } from './MyButton';

class App extends React.Component {
    render() {
        return (
               <MyButton/>
        )
    }
}
```

Here, we are importing the `MyButton` component from the `MyButton.tsx` file, and then using it in the `render` function.

Running the application now will render a button, as shown in the following screenshot:

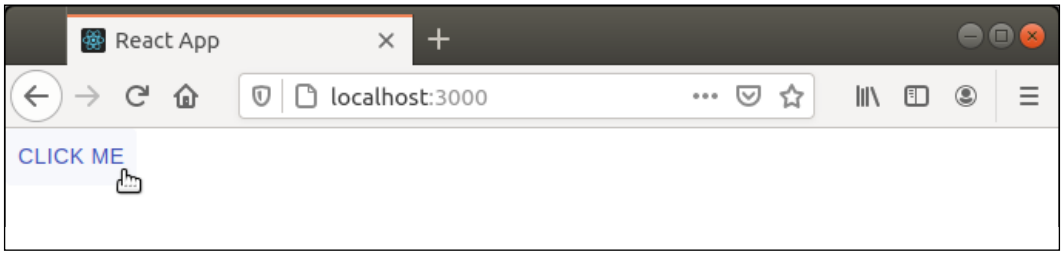


Figure 12.3: React application rendering a child component and a Material-UI button

Here, we can see that the `App` component is rendering the `MyButton` component, which in turn is rendering a Material-UI `Button` component.

In order to customize our `MyButton` component, we will need to define a set of properties, or props, that are available to set via the parent component. This is accomplished by defining our `MyButton` component with some properties, as follows:

```
export interface IMyButtonProps {
    buttonName: string;
    handleButtonClick(): void;
}
export class MyButton extends
    React.Component<IMyButtonProps> {
    render() {
        return (
            <Button color="primary"
                onClick={this.onButtonClicked}>
                {this.props.buttonName}
            </Button>
        )
    }
}
```

Here, we have defined an interface named `IMyButtonProps` that has a single member variable named `buttonName`, and a function named `handleButtonClick`. We have then modified the definition of the `MyButton` class, and are using generic syntax to specify that the `MyButton` component extends `React.Component` with the type `IMyButtonProps`.

What this means is that our `MyButton` component now exposes the properties of the `IMyButtonProps` interface as externally available for setting and internally available for use. The effect of this can be seen in the `render` function, where we are now using the `buttonName` property, which is made available on the `props` member variable of the React component.

In order to set this property, we now need to update the use of this `MyButton` component, and pass in the `buttonName` property, which means that we will need to update our `App.tsx` file as follows:

```
render() {
    return (
        <MyButton
            buttonName="Click here"
            handleButtonClick={this.handleClick}>
        </MyButton>
    )
}
handleClick() {
    console.log(`App.handleClick() called`);
}
```

Here, we have updated the `render` function and are now specifying the property `buttonName` as an attribute of the `MyButton` component. This will then set the `buttonName` property of the `MyButton` component when it is rendered.

Note that we have also set the `handleButtonClick` attribute to a function on the `App` class, named `this.handleClick`. What this means is that a function named `handleButtonClick` is made available within the `MyButton` component, as part of the `props` property, to use as a callback to the parent component.

Our `App` member function named `handleClick` is being passed down as this callback function into the `MyButton` component. The `handleClick` function just logs a message to the console indicating that it has been invoked.

React event handling

Thus far, we have specified that the `MyButton` component has two properties: firstly, a property named `buttonName`, which is a string, and secondly, a function named `handleButtonClick`. Let's now update our `MyButton` component to trap the `onClick` DOM event when our button is clicked, and then call the `handleButtonClick` function on the parent `App` component as follows:

```
export class MyButton extends
    React.Component<IMyButtonProps> {
    constructor(props: IMyButtonProps) {
        super(props);
        this.onButtonClicked =
            this.onButtonClicked.bind(this);
    }
    render() {
        return (
            <div>
                <Button color="primary"
                    onClick={this.onButtonClicked}>
                    {this.props.buttonName}
                </Button>
            </div>
        )
    }
    onButtonClicked() {
        console.log(`MyButton.onButtonClicked() called`);
        this.props.handleButtonClick();
    }
}
```

There are four important changes that we have made to the `MyButton` class. Firstly, we have introduced a `constructor` function, which has a single parameter named `props` of type `IMyButtonProps`. Note that the type of the `props` parameter must match the type that was specified for the `React.Component` generic argument. This constructor calls `super`, passing in the `props` argument. Remember that all constructors for derived classes must call the `super` function, and in this case, must pass in all of the arguments that were provided when constructing the class.

The second important change we have made is to bind the `onButtonClicked` function to the correct instance of `this`. This is easily accomplished by calling the `bind` function and supplying the class instance's `this` as the only argument. We then set the `onButtonClicked` function to be the bound version. This is necessary so that when the `onButtonClicked` function is called, it has access to the member variables and functions on the `MyButton` component instance.

The third important change we have made is to set the `onClick` event handler of the `Button` component to the `onButtonClicked` function. This will cause our function to be called when the button is clicked, and the DOM event is fired.

The fourth and last important change we have made is to define the `onButtonClicked` function, which logs a message to the console, and then calls the `handleButtonClick` function that is passed down as one of the component properties. Remember that this function is provided to the instance of the `MyButton` component by the `App` component.

We will also need to update our `App` component, and make sure that we bind the called `handleClick` function to the correct version of `this`, as follows:

```
export interface IAppProps { };

class App extends
    React.Component<IAppProps> {

    constructor(props: IAppProps) {
        super(props);
        this.handleClick =
            this.handleClick.bind(this);
    }

    render() {
    ... render function
    }
    handleClick() {
        console.log(`App.handleClick() called`);
    }
}
```

Here, we have added an interface named `IAppProps`, which is used as the only generic argument to the `React.Component` definition. We have then added a constructor function, in the same manner as we did with the `MyButton` component, in order to bind the `handleClick` function to the correct instance of `this`.

With these changes in place, we can now fire up our browser and click on the button as shown in the following screenshot:

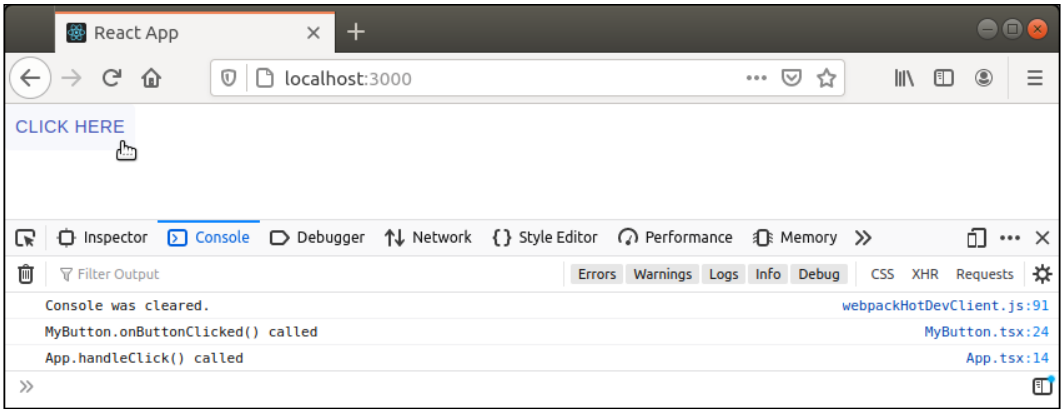


Figure 12.4: React application showing console log messages after clicking on a button

Here, we can see the two function calls being invoked when we click on the button. The first function that is called is the `onButtonClicked` function of the `MyButton` component, which is trapping the `onClick` DOM event. The `onButtonClicked` function is then calling the `handleClick` function on the `App` component. We can therefore see that in the same way that properties are handed down from a parent component to a child component, function callbacks can be handed down from parent to child component in the same manner, as in the end, both functions and properties are just values.

React state

Each React component can also have what is known as a component state. The difference between a component's properties and its state is that properties are set and managed by the parent component, but state is managed by the component itself.

As an example of how we can use component state, let's update our `App` component, and set an internal property that will toggle on or off when we click on our button. To indicate that a React component has state, we add a second argument to our component definition as follows:

```
export interface IAppProps { };
export interface IAppState {
    showDetails: boolean;
}
```

```
class App extends
    React.Component<IAppProps, IAppState> {

    constructor(props: IAppProps) {
        super(props);
        this.handleClick =
            this.handleClick.bind(this);
        this.state = {
            showDetails: false
        }
    }

    render() {
        ... existing render function
    }
    handleClick() {
        console.log(`App.handleClick() called`);
        this.setState({
            showDetails: !this.state.showDetails
        });
    }
}
```

Here, we have made four changes to our `App` component. Firstly, we have defined a new interface named `IAppState` that has a single property named `showDetails`. Secondly, we are now using both the `IAppProps` and `IAppState` interfaces as arguments to our generic component definition, that is, `React.Component<IAppProps,IAppState>`. The effect of this is that the `App` component will now have both a `props` property, and a `state` property that exposes the `IAppState` interface as an internal component state.

The third change we have made is to set the initial value of the `state` property to an object with the required properties within our constructor function. As can be seen in the code snippet, we are starting up with the `showDetails` property of our `state` instance set to false.

The fourth change we have made is to toggle the value of the `showDetails` property whenever the `handleClick` callback function is called. Note that we must call the `setState` function on our class instance in order to correctly trigger a state change. Simply assigning a new value to the `state` property, as we did in our constructor, will not work. When the `setState` function is called, React will trigger a call to our `render` function in order to update the component display.

We can show the value of the internal `state` property on the screen by updating our `render` function as follows:

```
render() {
    return (
        <div>
            <p>showDetails =
                {this.state.showDetails ? "true" : "false"}
            </p>
            <MyButton
                buttonName="Click here"
                handleButtonClick={this.handleClick}>
            </MyButton>
        </div>
    )
}
```

Here, the only change that we have made is to add a `<p>` element, and render the text `"showDetails ="`. We then interrogate the `showDetails` property of the component's `state` property, and render either the value `"true"` or the value `"false"`, depending on what the current internal state is.

Running our application now will allow us to toggle the value on the screen from `"false"`, which is the default value, to `"true"` when we click on the button as shown in the following screenshot:

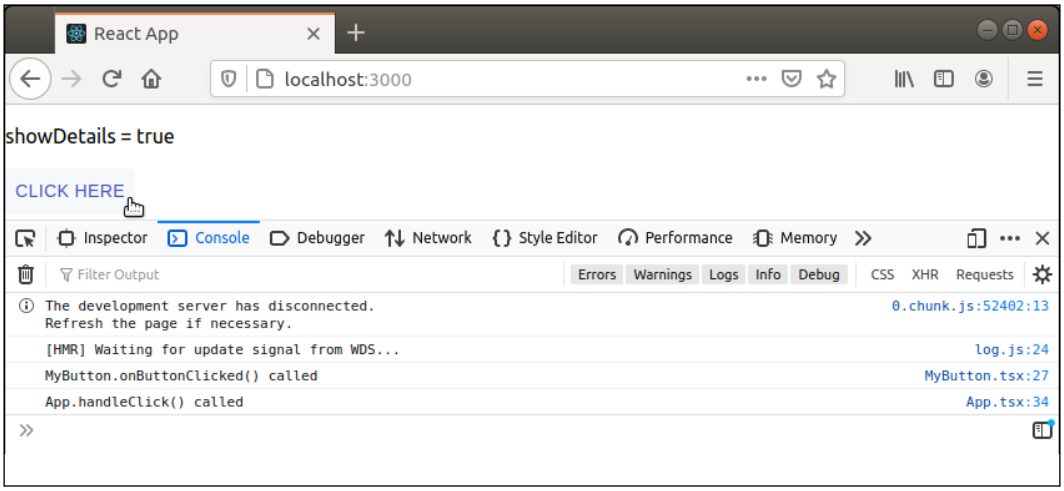


Figure 12.5: React application showing internal state values

Here, we can see that after the button was clicked, the `handleClick` function has been called, which will update the internal `showDetails` property on the component's state. Continuing to click this button will toggle the `showDetails` state from true to false and back again.

In this section of the chapter, we have introduced the main concepts to keep in mind when building React components. We discussed JSX, and how we can include logic within JSX expressions. We also looked at component props, how we can set them, and how we can use them within a component. This discussion on component props included a look at how to use callback functions within a component, and how we can trigger functions on a parent component, as long as we have bound the function to the correct instance of `this`. Finally, we discussed component state, and took a look at how we can update component state within our application.

A React application

In this section of the chapter, we will build a React application that shows us a list of products. Clicking on any one of the products will show a full-screen dialog that will slide in from the right, with a detailed description of the product and some product specifications. The detail view dialog will also include a form that will allow us to specify how many products we would like to order.

Application overview

The starting screen for our application will be the product list, which shows a short-form version of each product, which includes just the name of the product, the switch type, and an image, as shown in the following screenshot:

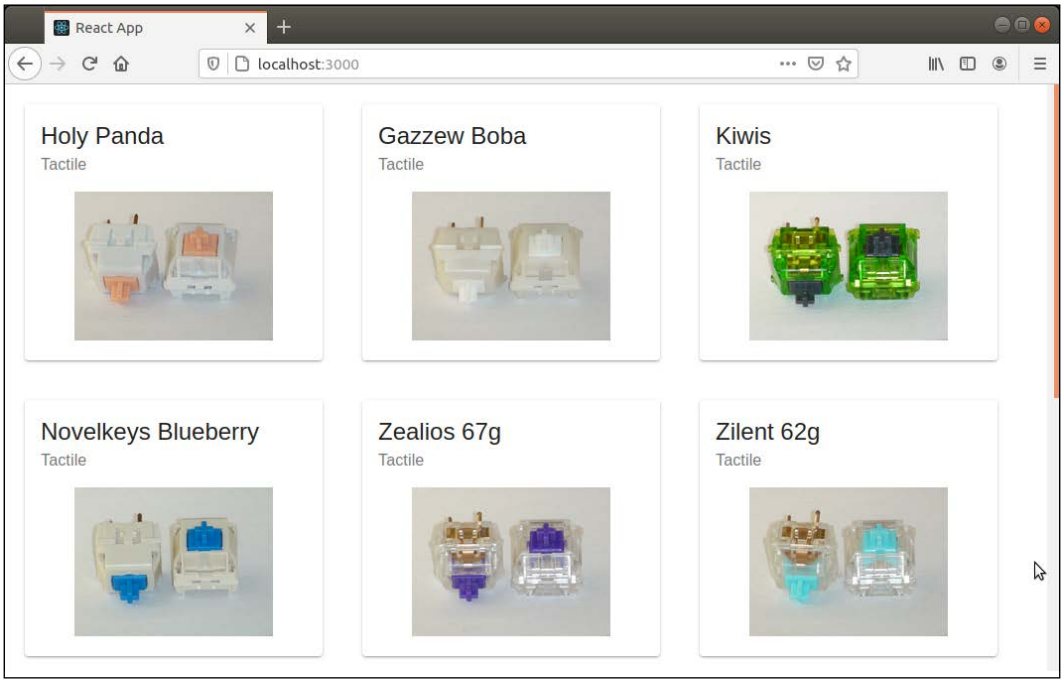


Figure 12.6: React application showing a list of products for sale

Here, we can see a summary view of all the products that are available for sale. Each product is rendered in its own area, with the product name and switch type shown, along with a thumbnail image of the product itself. Clicking on any one of these products will slide a panel in from the right, and show details about the product, as shown in the following screenshot:

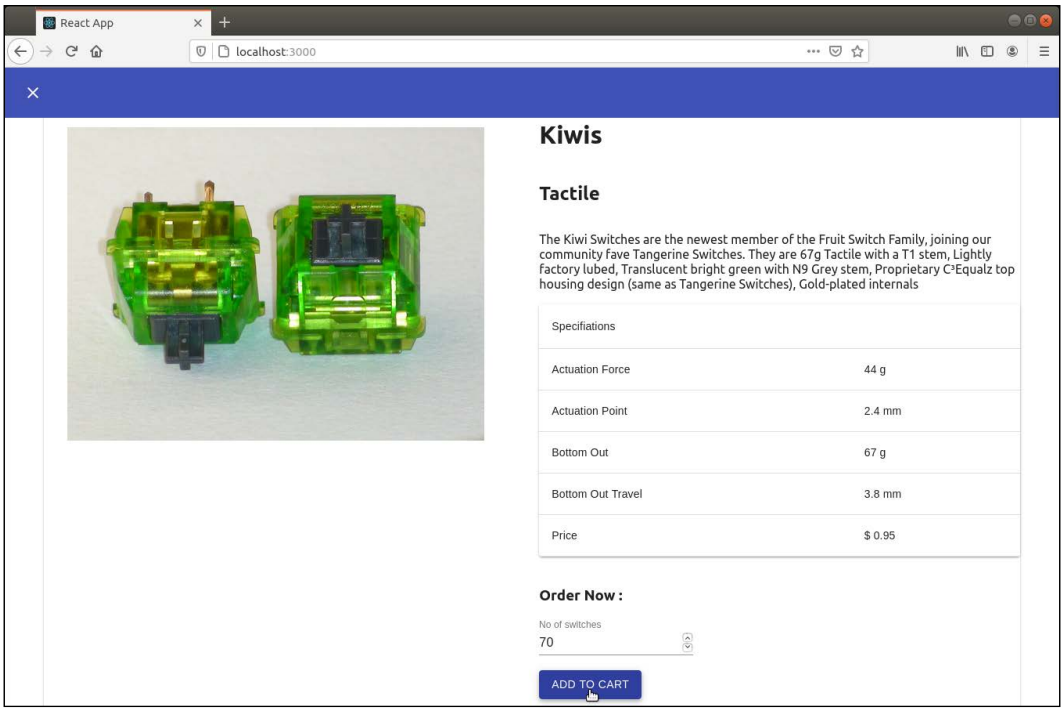


Figure 12.7: React application showing product details

Here, we can see the details screen for one of the products. On the top left, we have a close button that will return us to the main product screen. The product details panel shows a larger image of the product, as well as a product description and some product specifications. On the bottom right of the product screen is a form that allows us to specify how many units of the product we wish to purchase, along with an **ADD TO CART** button.

Mechanical keyboard switches

For those readers who are not aware of what these products are, they are different types of mechanical keyboard switches. Computer keyboards are either a standard, off-the-shelf membrane keyboard, or they can be made using mechanical switches. Membrane keyboards are cheap and easy to manufacture, and use a rubber membrane that has a small inverted "dome" under each switch. When a key is pressed, this dome collapses, which then presses two thin layers of plastic together to register a current on the switch.

Mechanical keyboards, however, use a mechanical switch mechanism to register a keypress. When the internal stem of a switch (which is on a spring) is pressed down, it forces two copper leaves together that register current on the switch, and thus register a keypress. Using mechanical switches allows a multitude of variations in the "feel" of the keyboard, and can use different "weights" of spring to make them heavier or lighter to depress, as well as variations in the stem itself, providing what are known as linear, tactile, or clicky switches.

Mechanical keyboards help to reduce typing fatigue, are more ergonomic, and are just a pleasure to type on. The range of switches means that they can be customized to suit every user's typing needs. They are, however, quite a lot more expensive than standard membrane keyboards. This book, in fact, has been written on a custom-built, brass, plate-mounted, split, tented, 60% keyboard with Zilent v2 67g switches, and GMK Laser keycaps.

Application components

Our React application will need a few components, with each one handling the rendering of one particular element on the screen. Our main screen will need a `CollectionView` component, which will be used to render the entire collection of products. Each item in the list of products will be rendered by an `ItemView`, which will render the thumbnail image and the product name. When we click on an item, the `App` component will show a full-screen dialog, and this will be rendered by the `DetailView` component.

A logical view of how the application components fit together can be seen in the following diagram:

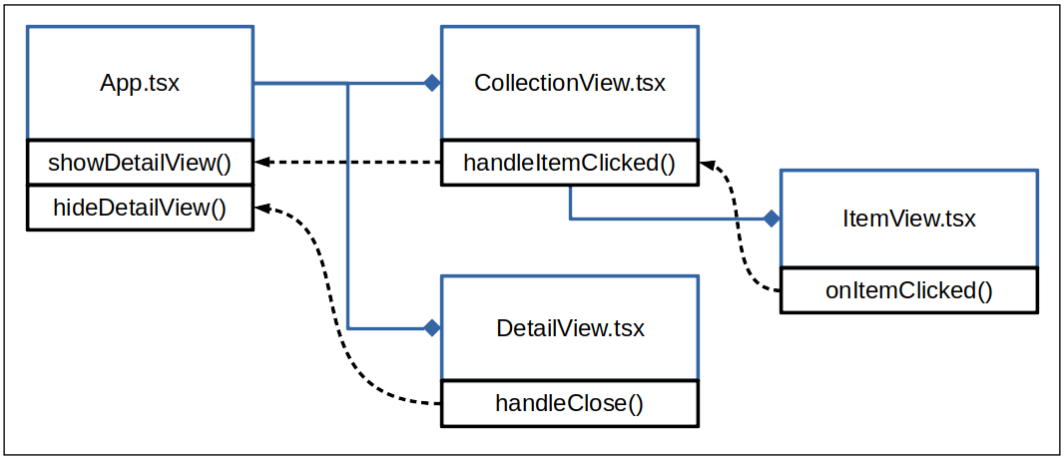


Figure 12.8: Logical view of the React application components

Here, we can see the relationships between the application components. The `App` component renders both the `CollectionView` and the `DetailView` components. The `CollectionView` will loop through each item in the product collection, and render an individual `ItemView`. When a user clicks on an `ItemView`, the `onItemClicked` event handler will be invoked, which will then call the `handleItemClicked` function on the `CollectionView`. This will then need to call the `showDetailView` function on the `App` component, which will then show the `DetailView` for the item selected. When a user clicks on the close button on the `DetailView`, the `handleClose` function will be triggered, and this event needs to trigger the `hideDetailView` function on the `App` component.

Our application will need a list of products, which will just be a simple array, in a file named `Products.tsx`, as follows:

```
export interface IProduct {
    id: number;
    name: string;
    type: string;
    image: string;
    longDescription: string;
    specs?: ISpecs;
}
export interface ISpecs {
```

```
    actuationForce: string;
    actuationPoint: string;
    bottomOut: string;
    bottomOutTravel: string;
    price: string;
}

export class Collection {
    items: IProduct[] = [
        {
            id: 1,
            name: "Holy Panda",
            type: "Tactile",
            image: "/images/holy_panda.png",
            longDescription: "..."
            , specs: {
                actuationForce: "44",
                actuationPoint: "2.4",
                bottomOut: "62",
                bottomOutTravel: "3.8",
                price: "1.60"
            }
        }
    ... other items
    ]
    handleItemClicked(id: number) { }
}
```

Here, we start with two interfaces, named `IProduct` and `ISpecs`. The `IProduct` interface defines the data that we will need on a particular product, such as its name, the switch type, an image, and a long description. The `ISpecs` interface holds data about the actuation forces and the price.

The `Collection` class has a property named `items`, which is an array of type `IProduct`, and will hold all data for each of our products. Note that the `Collection` class also has a function named `handleItemClicked`, which has a single argument named `id` of type number. Remember that the `CollectionView` is an intermediate component between the `App` component and the `ItemView` component. In order for an `ItemView` event to be raised with the `App` component, we will need to route this through the `handleItemClicked` function.

The CollectionView component

Let's start with the `CollectionView` component, which is responsible for rendering the entire collection of products, in the `CollectionView.tsx` file, as follows:

```
export class CollectionView
    extends React.Component<Collection> {
    constructor(props: Collection) {
        super(props)
        this.handleItemClicked =
            this.handleItemClicked.bind(this);
    }

    handleItemClicked(id: number) {
        console.log(`handleItemClick : ${id}`);
        this.props.handleItemClicked(id);
    }

    render() {
        let items = this.props.items.map((item) => {
            return (
                <ItemView {...item} key={item.id}
                    onItemClicked={this.handleItemClicked} />
            )
        })
        return (
            <Box display="flex" flexWrap="wrap">
                {items}
            </Box>
        )
    }
}
```

Here, we have a class named `CollectionView` that extends from `React.Component`, with the `Collection` class as its generic `props` type. This class has a `constructor` function, which is used primarily to bind the `handleItemClicked` function to the correct instance of `this`. The class also has a `handleItemClicked` function definition that will call the `handleItemClicked` function that is passed down from the parent component through the `props` argument.

The `render` function loops through all of the products in the product list that are passed into the component through the `props.items` property. For each item, we generate some JSX that includes an `ItemView` component.

Note how we are able to pass in the current product properties all in one go by using spread syntax, that is, `{...item}`, instead of having to pass in each named property. Each `ItemView` component also has a `key` attribute, which we are setting to the value of the `id` property of our `item`. This `key` attribute will be used by React to identify which areas of the DOM need to be re-rendered, if it detects a change.

We also set the `ItemView` callback function named `onItemClicked` to the `handleItemClicked` function. Finally, our `render` function renders a Material-UI `Box` component that contains the built-up JSX for our list of products.

The ItemView component

As we have seen, each product in our list will be rendered by an `ItemView` component, in the `ItemView.tsx` file, as follows:

```
import './ItemView.css';
import { Card, CardHeader, CardMedia } from "@material-ui/core";

export interface IItemView extends IProduct {
    onItemClicked(id: number): void;
}

export class ItemView
    extends React.Component<IItemView> {
    constructor(props: IItemView) {
        super(props)
        this.onItemClicked =
            this.onItemClicked.bind(this);
    }
    render() {
        return (
            <div className="item-view-card">
                <Card onClick={this.onItemClicked}>
                    <CardHeader title={this.props.name}
                        subheader={this.props.type}
                    />
                    <CardMedia
                        className="card-media-image"
                        image={this.props.image}
                    />
                </Card>
            </div>
```

```
            )
        }
        onItemClicked() {
            this.props.onItemClicked(this.props.id)
        }
    }
```

Here, we start by importing a CSS file named `ItemView.css`. This file will hold a few CSS styles that will be used by this component. Note that there are a number of ways to include styles for React components, which we will not get into detail about here, but the simplest method is to import a CSS file as we have done here.

We then have an interface definition named `IItemView` that extends the `IProduct` interface that we used to define the structure of each of our products. It has a single event handler defined, named `onItemClicked`, which has a single argument named `id` of type number. This will allow each `ItemView` component to tell its parent the `id` that it has for its product, when a user clicks on the `ItemView` itself.

The `ItemView` class, which is used to render an item, extends from `React.Component`, and uses the `IItemView` interface as its property definition.

The `ItemView` class has a constructor function, as we have seen before, and it also has a `render` function. Note that we have tied the `onClick` DOM event handler to the top-level `Card` element, so that clicking anywhere on the `Card` element itself will trigger a click event. The `onClick` event handler is set to trigger the `onItemClicked` function on the component, which will then call the `onItemClicked` function on the parent `CollectionView` component, through the `props` property. Note that we are calling this `onItemClicked` function with the `id` property for this product by referencing the `this.props.id` property.

Our `Collection` and `ItemView` components are almost ready to display; all we need now is two small pieces of CSS in the `ItemView.css` file as follows:

```
.card-media-image {
    height: 150px;
```

```
    background-size: contain !important;
    margin-bottom: 20px;
}

.item-view-card {
    padding: 20px;
    min-width: 300px;
}
```

Here, we have defined a style for the image that has shown for each item named `card-media-image`. This CSS just sets the `height` of each image, and that the `background-size` property should be set to `contain`, so that each image is shown with the same dimensions. The `item-view-card` style sets some padding, and a minimum width for the `<div>` surrounding our `Card` element.

We can now update our `App.tsx` file, and render the `CollectionView` as follows:

```
const collectionInstance = new Collection();

class App extends
    React.Component<IAppProps, IAppState> {
    ... existing constructor

    render() {
        return (
            <div>
                <CollectionView
                    {...collectionInstance}
                />
            </div>
        )
    }
    ... existing functions
}
```

Here, we create an instance of the `Collection` class, which has all of our products defined, and we then pass it into the `CollectionView` class, again using spread syntax. With this in place, we now have a working product list, as shown in the following screenshot:

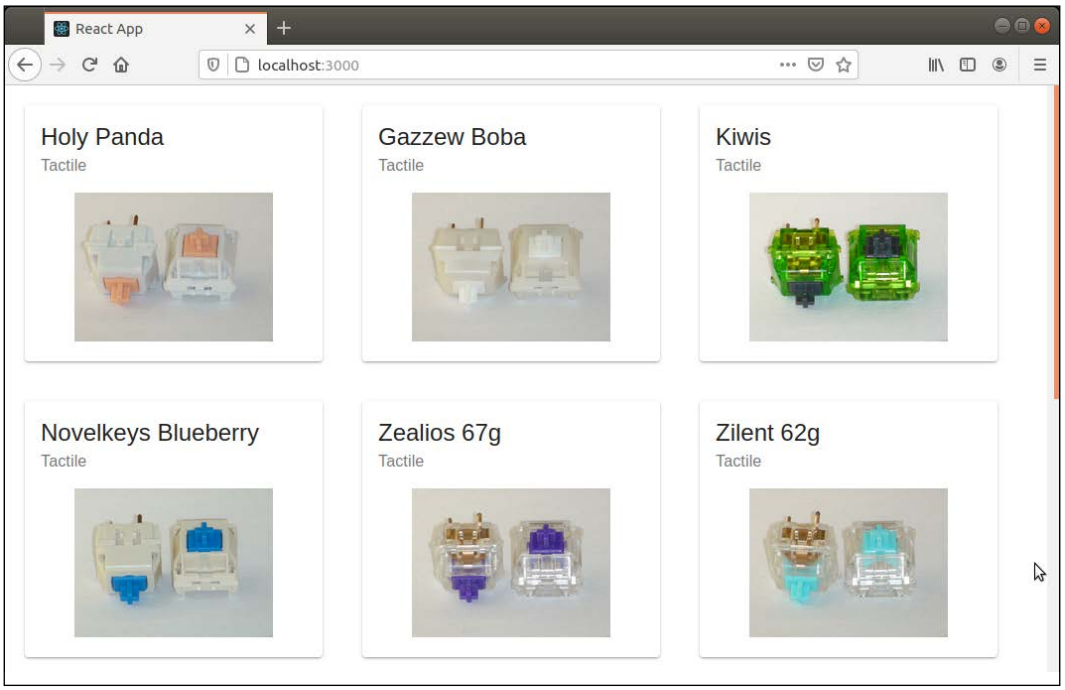


Figure 12.9: React application showing a list of products

Here we can see that the `App` component is rendering a `CollectionView` component, which in turn is rendering an `ItemView` component for each product in our product list.

The DetailView component

When a user clicks on one of our products, we would like a panel to slide in from the right of the screen with a detailed view of the product in question. This `DetailView` component, in the `DetailView.tsx` file, is as follows:

```
export interface IDetailsProps {
    open: boolean;
    product: IProduct | null;
    handleClose(): void;
}

const Transition = React.forwardRef(function Transition(
```

```
    props: TransitionProps & { children?: React.ReactElement },
    ref: React.Ref<unknown>,
) {
    return <Slide direction="left" ref={ref} {...props} />;
});

export class DetailView extends React.Component<IDetailsProps> {

    constructor(props: IDetailsProps) {
        super(props);
        this.handleClose = this.handleClose.bind(this);
    }

    render() {
        return (
            <div className="full-screen-details-dialogue">
                <Dialog
                    fullScreen
                    open={this.props.open}
                    TransitionComponent={Transition}>
                    <AppBar>
                        <Toolbar>
                            <IconButton
                                edge="start"
                                color="inherit"
                                onClick={this.handleClose}
                                aria-label="close">
                                <Close></Close>
                            </IconButton>
                        </Toolbar>
                    </AppBar>

                    ... other screen elements

                </Dialog>
            </div>
        )
    }
    handleClose() {
        console.log(`Details: handleClose()`)
        this.props.handleClose();
    }

}
```

Here, we start with an interface named `IDetailsProps` that contains the properties and methods that our `DetailView` component will need. The first property is named `open`, is of type boolean, and controls whether the full-screen `Dialog` component is open or closed. The second property is named `product`, and will contain the currently selected product. We then have a function named `handleClose`, which is the callback function that is passed into our `DetailView` component by the parent component.

We then define a constant named `Transition`, which is used by the Material-UI library to control animated transitions. We will not analyze this code here, but just note that in the `render` function of our `DetailView`, we are setting the `TransitionComponent` attribute on the `Dialog` element to use these transition settings.

Finally, we define our class named `DetailView`, which is a standard React component with the `IDetailsProps` interface defined as the component properties.

The `render` function of this `DetailView` component has two interesting things to note. Firstly, there is an attribute on the Material-UI `Dialog` component named `open`, and this is used to either show or hide the dialog itself, by setting it to either true or false. We are setting this attribute to the value that is set in the component's properties, which is the `this.props.open` property. This means that the parent component is in control of whether the dialog is shown or not. If the `props.open` property is set to true by the parent component, the dialog will show, and if it is false, then the dialog will not show.

The second thing to note about the `render` function is that within the `Dialog` element, we have an `AppToolbar` element, which contains a `Toolbar` element, and finally an `IconButton` element. We have set the `onClick` attribute of this `IconButton` element to trigger the `handleClose` function on our `DetailView` component. This function will invoke the `handleClose` callback function from the parent element, which will set the `props.open` property to false, and effectively close the dialog itself.

Note that we have not shown all of the screen elements in the `render` function in this sample, as it is quite lengthy and repetitive. Please refer to the sample code for a full working example.

The App component

Now that we have a `DetailView` component in place, we can turn our attention to the changes required in the `App` component to either show or hide the `DetailView` dialog, as follows:

```
export interface IAppState {
    showDetails: boolean;
    product: IProduct | null;
```

```
}

const collectionInstance = new Collection();

class App extends
    React.Component<IAppProps, IAppState> {

    constructor(props: IAppProps) {
        super(props);
        this.showDetailView =
            this.showDetailView.bind(this);
        this.handleClose = this.handleClose.bind(this);
        this.state = {
            showDetails: false,
            product: null
        }
    }

    render() {
        return (
            <div>
                <CollectionView
                    {...collectionInstance}
                    handleItemClicked={this.showDetailView} >
                </CollectionView>
                <DetailView open={this.state.showDetails}
                    product={this.state.product}
                    handleClose={this.handleClose}></DetailView>
            </div>
        )
    }
    showDetailView(id: number) {
        let foundItem = _.find(
            collectionInstance.items,
            (item: IProduct) => {
                return item.id === id
            });

        if (foundItem) {
            this.setState({
                showDetails: true,
                product: foundItem
```

```
            });
        }
    }
    handleClose() {
        console.log(`App : handleClose()`);
        this.setState({
            showDetails: false,
            product: null
        })
    }
}
```

There are five updates that we have made to the `App` component. Firstly, we have added a property to our `IAppState` interface, named `product`, which will hold the currently selected product, or null if no product has been selected. Secondly, we have set the currently selected product to null in our constructor function. Thirdly, we have added a function named `showDetailView` as the callback function for the prop `handleItemClick` on our `CollectionView` instance.

The fourth update we have made is to render the `DetailView`. Note that we have set the `open` property of the `DetailView` to the `showDetails` property of the `App` component's state. This means that when the internal state of the `App` component changes, it will automatically propagate these changes to the `DetailView` child component, and as such will show or hide the full-screen dialog when required. We have also attached the `handleClose` member function to the `handleClose` callback function for the `DetailView` component, which will trigger when the user closes the dialog.

The fifth and final update to the `App` component is to set the state of the component when the `handleClose` function is triggered. Remember that the `showDetailView` function is triggered when a user clicks on a product, which will set the `state.showDetails` property to true. The `handleClose` function, therefore, sets the `state.showDetails` property to false, which will hide the `DetailView` dialog.

With these changes in place, we can click on a product and view the product details, as shown in the following screenshot:

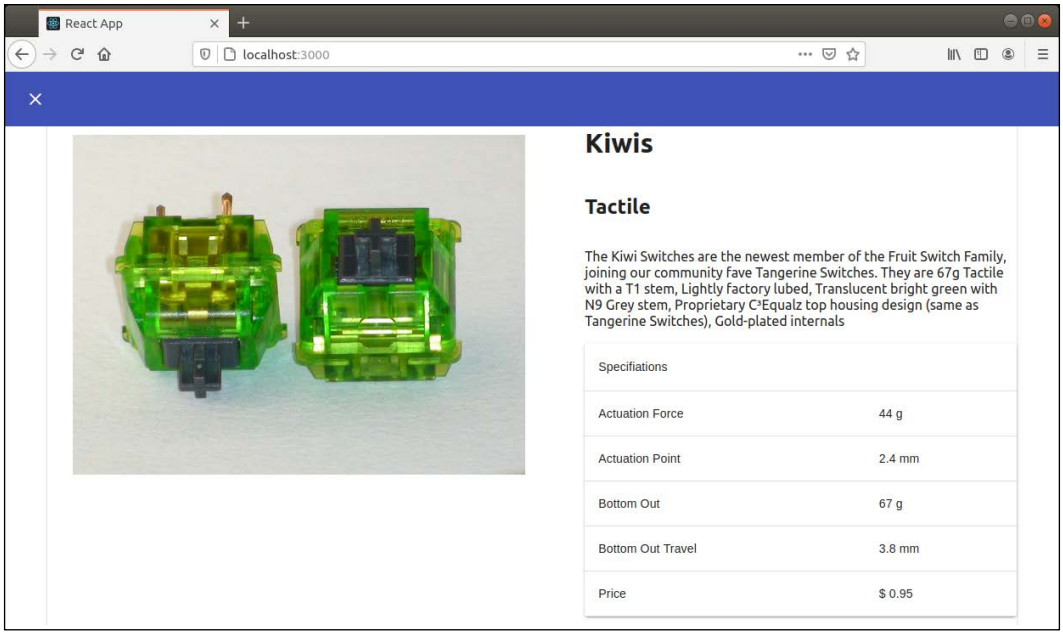


Figure 12.10: React detail view showing product details and specifications

Here, we can see the `DetailView` component rendering product details. We have a large image of the product on the left, and a long description and product specifications on the right. Note on the top left of the screen is the close dialog button.

React forms

Our product details page just needs an **Add to cart** button in order for our React application to be complete. We will, however, also need to know how many switches a user would like to purchase. Mechanical keyboards come in a variety of sizes, with the most popular being 65%, which has around 70 keys. A ten-keyless keyboard, which does not have the right-most number pad, is around 100 keys, and a full-size keyboard has around 120 keys.

In the final section of this chapter, we will take a look at how React handles form input values and form submission, which we will need to gather how many switches a user would like to add to their cart.

We will make updates to our `DetailView.tsx` file in a few stages. Firstly, let's introduce an interface to hold the number of switches a user wishes to purchase as follows:

```
export interface IDetailsState {
    noSwitches: number;
}

...existing code

export class DetailView
    extends React.Component<
        IDetailsProps,
        IDetailsState
    > {
    constructor(props: IDetailsProps) {
        super(props);
        this.handleClose = this.handleClose.bind(this);
        this.state = {
            noSwitches: 70
        }
    }
    ... existing functions
}
```

Here, we have made three changes to our `DetailView.tsx` file. Firstly, we have introduced an interface named `IDetailsState` that has a single property named `noSwitches`, of type number, which will hold the number of switches that the user enters into the form. Secondly, we have modified the generic types passed into the `React.Component` to include the `IDetailsState` interface, which will strongly type any use of the `state` property within the component to be of type `IDetailsState`.

Thirdly, we have set the initial value of the `noSwitches` state property to be `70`, which is an example of pre-populating a form with values. Remember that if we have stored information input into a form previously, or have persisted the form data to the database, we would want to populate our form data with what the user has already stored on our site.

Let's now look at the actual form element that we will render within our JSX, as follows:

```
render() {
    return (
    ... existing JSX
```

```
        <Grid item >
            <h3>Order Now :</h3>
        </Grid>
        <form noValidate
            autoComplete="off"
            onSubmit={this.onSubmit}>

            <Grid item >
                <TextField
                    name="noSwitches"
                    id="standard-basic"
                    label="No of switches"
                    onChange={this.onChange}
                    value={this.state.noSwitches}
                    type="number"
                    />
            </Grid>
            <Grid item >
                 
            </Grid>

            <Grid item >
                <Button
                    type="submit"
                    variant="contained"
                    color="primary">
                    Add to Cart
            </Button>

            </Grid>
        </form>

    ... existing JSX
    )
}
```

Here, we have a snippet of our full `render` function that just shows the `<form>` HTML element itself. There are a couple of things that are of interest in this JSX snippet. Firstly, we have defined a `<form>` HTML element, and set the `onSubmit` attribute to a function on the `DetailView` class named `onSubmit`. This function will be invoked when the form itself is submitted.

The second thing to note in this snippet is the `onChange` and `value` attributes of the `TextField` element. The `onChange` event will be called whenever the value entered into the input field is modified by the user, which we will see a little later. The `value` attribute is set to the `noSwitches` property of the component's `state`. Again, this is how we can pre-populate input values from either a persistent store or with defaults, as we are doing here.

Finally, we have a button with the text `"Add to Cart"` that has a `type` attribute of `"submit"`, meaning that the entire form will be submitted when this button is pressed. When the form is submitted, the `onSubmit` function will be invoked. Let's take a look at the `onSubmit` class member function, and the `onChange` function, as follows:

```
onChange(event: React.ChangeEvent<HTMLInputElement>) {
    this.setState({
        noSwitches: parseInt(event.target.value)
    });
    console.log(`onChange :
        ${event.target.name} :
        ${event.target.value}`);
}

onSubmit(e: React.FormEvent) {
    console.log(`submit :
        ${this.state.noSwitches}`);
    e.preventDefault();
}
```

Here, we have the definition of the `onChange` and `onSubmit` member functions. The `onChange` function has a single argument named `event`, which is of type `React.ChangeEvent<HTMLInputElement>`. This React type will give us access to the properties and functions that are available on each DOM event, which are quite extensive and we will not explore here. Note that our `onChange` event starts by calling the `setState` function, with the new value that we retrieved from the control, by referencing `event.target.value`. This is one way of keeping the internal state of a component synchronized with the user's input, and is known as a controlled component. React also provides what are known as uncontrolled components, which use a reference to access the value of a control.

Within our `onChangeEvent`, we are then logging a message to the console. If we modify the default value of `70` within the input control to a value of `120`, we will notice this function being called for every keystroke, as follows:

```
onChange : noSwitches : 7
onChange : noSwitches :
onChange : noSwitches : 1
onChange : noSwitches : 12
onChange : noSwitches : 120
```

Here, we can see that the first `onChange` event on the `noSwitches` input element was the change from the value `70` to the value `7`, then to a blank input, and finally to the values of `1`, `12`, and `120`. For each of these events, we are calling the `setState` function.

The `onSubmit` function, which is called when the user presses the **Add to Cart** button, is just logging the current value of the internal `state` property named `noSwitches`. Note that it is also calling the `preventDefault` function in order to prevent the entire application from re-loading.

With our form in place, our `DetailView` page now looks like the following screenshot:

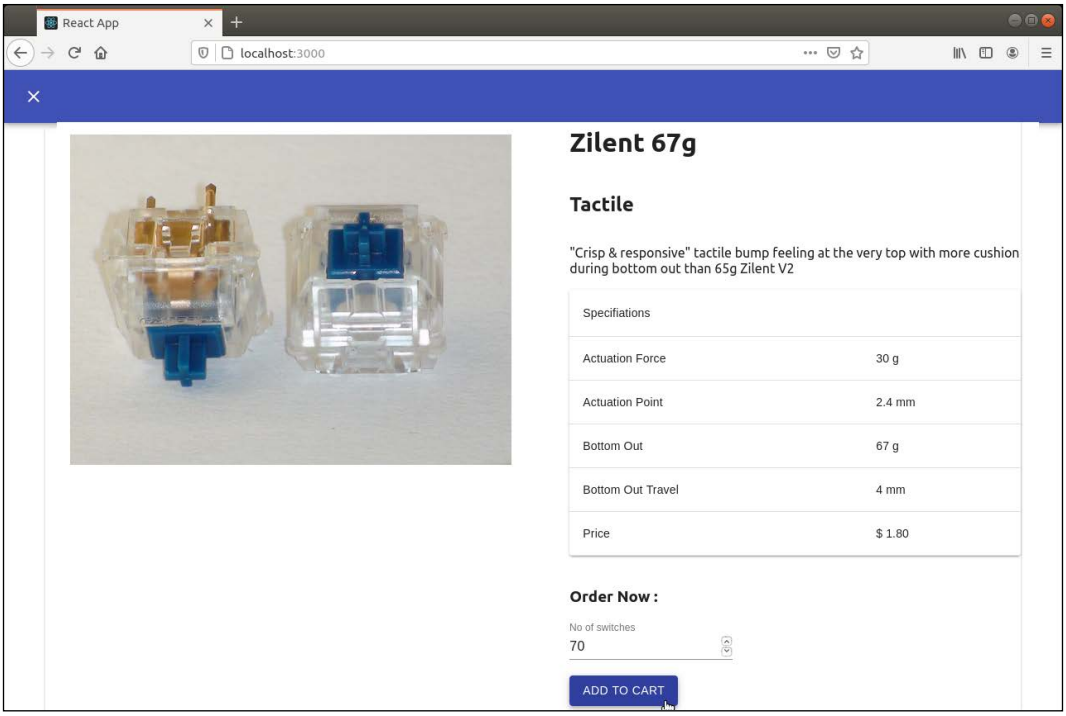


Figure 12.11: React form showing input control

Here, we can see the rendered input control and the `"Add to Cart"` button, which will trigger the form submit.

With these changes in place, our React application is complete.

Summary

We have covered a fair bit of ground in this chapter, as we built a sample React application. We started with the basics of React and JSX syntax, and explored the interaction of logic within JSX returning render functions. We then discussed how properties are used within React components, and what React component state is. We then built a React application to show a product list, and discussed the various components we needed to render the list itself, and a product details panel. We finished off with a discussion on React form input. In the next chapter, we will discuss another very popular JavaScript framework, named Vue, and will build a shopping cart-style application.

13
Vue

Vue.js, or just Vue, is a JavaScript framework that has been steadily gaining popularity among the community. Vue describes itself as being "progressive," meaning that the framework can be quickly put to use for user interfaces, but can then be extended to include more and more functionality as required by your Single-Page Application. This extensibility includes features such as routing and advanced state management. Vue was created by Evan You, who was working at Google at the time, and was using AngularJS (or Angular 1) quite extensively. The first official release of Vue was in February 2014.

Version 3 of Vue, which was released in September 2020, was written in TypeScript, and as such, Vue supports TypeScript as a "first-class citizen," to quote the Vue website. While it was possible to use TypeScript and Vue prior to version 3, the integration and support for TypeScript have now become rather seamless.

In this chapter, we will explore the syntax and structure of a Vue application, and build a basic shopping cart, utilizing the TypeScript integration aspects of Vue. Specifically, we will cover the following topics:

- Setting up a Vue application
- Structure of a Vue component
- Child components
- Setting component props
- Component state
- Computed properties, conditionals, and loops
- Building a Vue application
- Material Design for Vue

Introduction to Vue

Vue is somewhat unique in its component structure, as it combines HTML templates with TypeScript logic, and also component-specific CSS within the same file. In this section of the chapter, we will set up a Vue development environment, and explore some of the concepts of Vue components.

Vue setup

Similar to what we have seen with Angular and React, Vue also provides a command-line utility for generating a bare-bones Vue application, which is known as the Vue CLI, or command-line interface. We can install the Vue CLI globally, as follows:

```
npm install -g @vue/cli
```

Here, we are using `npm` to install the Vue CLI, and making it available globally with the `-g` flag. Once it's installed, we can check the version of the Vue CLI as follows:

```
vue -V
```

Here, we are invoking the Vue CLI, which is simply called `vue`, and using the `-V` flag to retrieve the currently installed version, which at the time of writing returns:

```
@vue/cli 4.5.9
```

Here, we can see that our globally installed version of the Vue CLI is at version 4.5.9. With the CLI installed, we can now create a Vue application as follows:

```
vue create shopping-cart
```

Here, we are invoking the Vue CLI with the option `create`, which will create a Vue application, as well as the directory we wish to create the application in, which is `shopping-cart`. Once the CLI is invoked, it will prompt us with a few questions on how we wish to configure our application. To configure Vue to use TypeScript, we must select **Manually select features** as the option for the first question, which is **Please pick a preset**.

The features that we will need for our project are shown in the following screenshot:

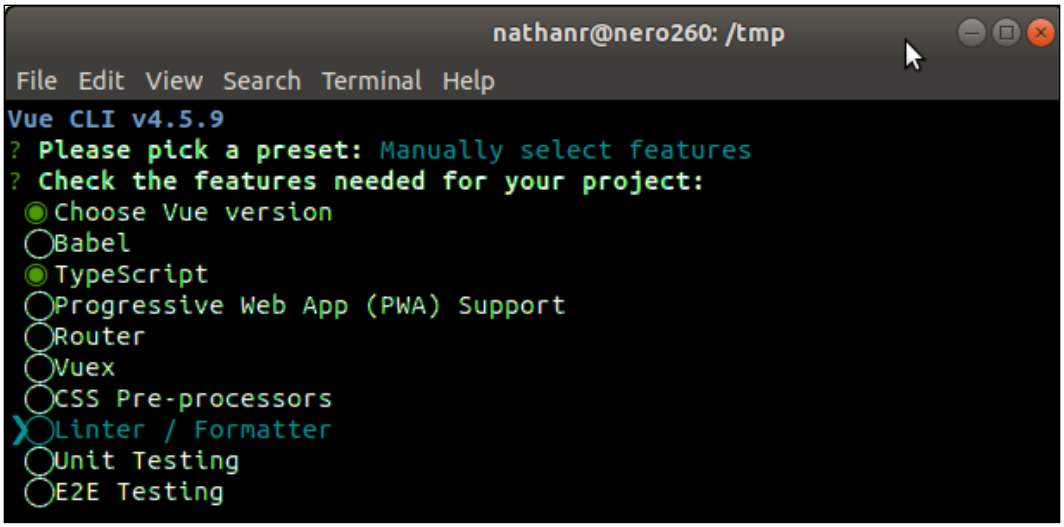


Figure 13.1: Vue CLI showing options available for the creation of a new application

Here, we have selected the **Choose Vue version** and **TypeScript** options for the features that will be included in our new Vue application. Hitting *Enter* now will give us a series of new prompts that will govern the application structure, starting with the option **Choose a version of Vue.js**. We will select **3.x (Preview)**, to use version 3 of Vue.

The next option we are presented with is **Use class-style component syntax**, to which we will answer `Y`. This will allow us to use native TypeScript classes within our Vue components, which we will explore a little later. The next option asks whether we would like to use Babel alongside TypeScript, to which we can answer `N`, and then how to store config files, to which we can answer **In dedicated config files**. The final question is whether we would like to use these options as presets for any other project we create, to which we can also answer `N`.

Once the options have been configured, Vue will download the necessary npm packages, and then configure our project according to the options we selected. Once complete, we can change into the newly created folder named `shopping-cart`, and start a development server as follows:

```
cd shopping-cart
npm run serve
```

Here we are starting the development server in the `shopping-cart` directory by using the `npm run` command, followed by the `serve` keyword. This will compile our application source code, start a server at `localhost:8080`, and then run in watch mode in the background to re-compile our source should any of our source files change. Pointing a browser to this URL will bring up the default Vue application, as shown in the following screenshot:

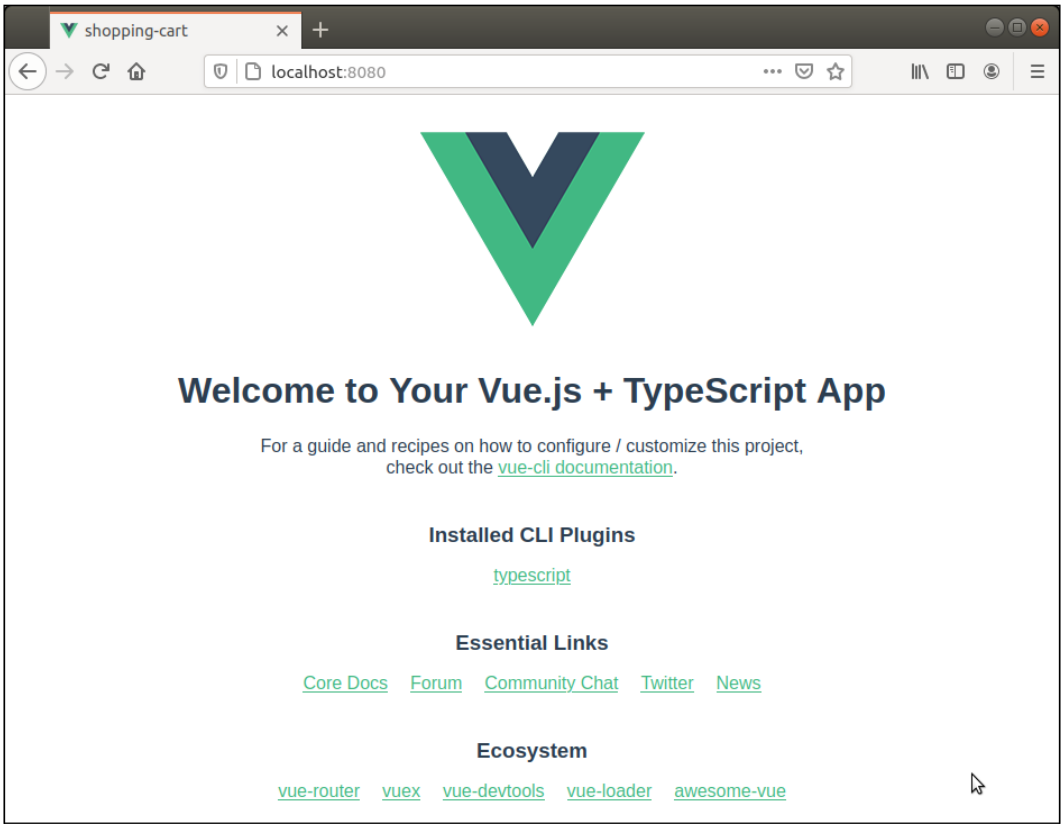


Figure 13.2: Default Vue application after running and configuring the Vue CLI

Here, we can see that Vue has created an application, and also included some default content, which includes a link to the **vue-cli documentation** and some essential links.

Component structure

If we explore the files and directory structure that the Vue CLI has generated for our application, we will notice three top-level directories named `node_modules`, `public`, and `src`.

Outside of the `node_modules` directory, which contains the modules required to generate the application, the `public` directory contains the `index.html` file, and all other Vue files are housed in the `src` directory. If we examine the `src` directory, we will notice a `main.ts` file, with the following contents:

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Here, we can see that the `main.ts` file is importing the `App` class from the `App.vue` file, and calling the `createApp` function with the `App` class as its only argument. It then calls the `mount` function with the string `'#app'`, which tells Vue to mount the generated HTML from the `App` class to a DOM element with an `id` of `app`, or `<div id="app">`, which can be found in the `public/index.html` file. Note that all Vue source files have the extension `.vue`.

The `App.vue` file contains our `App` component. Let's modify the default `App.vue` file with the following content:

```
<template>
    Hello Vue !
</template>

<script lang="ts">
import { Vue } from 'vue-class-component';

export default class App extends Vue {}
</script>

<style scoped>
</style>
```

Here, we can see the main structure of a Vue component. There are three main sections to a `.vue` file, denoted by the `<template>`, `<script>`, and `<style>` elements. The `<template>` section contains the HTML snippet that will be rendered by the component. In this snippet, this just contains the text `"Hello Vue !"`.

The `<script>` section has an attribute named `lang`, which is set to the value of `"ts"`. This marks any code that is within the `<script>` element as TypeScript code, and will therefore follow any of our existing TypeScript language syntax and compilation rules. Within this TypeScript code, we have defined a class named `App` that extends the class `Vue`. This is how we create a Vue component.

The `<style>` section can contain any CSS styles that need to be applied. Note that we have added a `scoped` attribute to the style element. This will ensure that the styles contained in this file are only applied to the elements within this file's `<template>` section, and will not be applied to any other elements. If the `scoped` attribute is removed, then these styles become available to the entire application.

Child components and props

As we have seen with the other frameworks that we have worked with, it is a good idea to separate out our components, and let each component focus on one area, or be in control of a specific set of functionality. Often, we will need to set properties on a child component from the parent component, in order to tailor it. As an example of a child component with properties, let's modify the `component/HelloWorld.vue` file to contain the following:

```
<template>
    <p>Hello World</p>
    <p>msg prop = {{ msg }}</p>
</template>

<script lang="ts">
import { Options, Vue } from 'vue-class-component';

@Options({
    props: {
        msg: String
    }
})
export default class HelloWorld extends Vue {
    msg!: string
}
</script>

<style scoped>
</style>
```

Here, we have a Vue component named `HelloWorld`, which contains the three sections of a Vue component. The `<template>` section will render a paragraph to the screen with the text `"Hello World"`, and then another paragraph with the text `"msg prop ="`. Following this, the template will inject the value of a property named `msg`, by using the double-curly braces syntax to reference the property, that is, `{{ msg }}`.

Our `<script>` section contains a class named `HelloWorld` that extends from the `Vue` class, and is exported. Note how we have used a Vue decorator named `@Options` to define an object that contains a single property named `props`. Within this `props` property, we have defined a property named `msg`, which is using the `String` constructor. The Vue `props` property exposes component properties to parent components, and can contain any type of object.

We have also created a class property named `msg` within the `HelloWorld` class, which matches the exported `props` name of `msg`. Note that we are using a definite assignment operator on the `msg` class property, so that we do not generate compiler errors. Vue will actually handle the value assignment of this class property for us, and Vue does not use constructor functions in its component life-cycle, hence the need for the definite assignment.

In order to use this `HelloWorld` component, we can now update our `App.vue` file as follows:

```
<template>
    Hello Vue !
    <HelloWorld
        msg="msg from App.vue" >
    </HelloWorld>
</template>

<script lang="ts">
import { Vue, Options } from 'vue-class-component';
import HelloWorld from "./components/HelloWorld.vue";

@Options({
    components: {
        HelloWorld,
    },
})
export default class App extends Vue {}
</script>

<style scoped>
</style>
```

Here, we have updated our `App.vue` file in three places. Firstly, we have modified our template to include an HTML element of `<HelloWorld>`, which will render the `HelloWorld` component to the screen. Note that we have also set the `msg` attribute on the component to a value of `"msg from App.vue"`, which will set the value of the `msg` property on the `HelloWorld` component.

The second change we have made is to import the `HelloWorld` component within our script section. Note that we must import the full name of the file containing the `HelloWorld.vue` component here.

The third change we have made is to decorate the `App` class with an `@Options` decorator, and have specified a `components` property, with a single property of `HelloWorld`. This instructs Vue to load the `HelloWorld` component when the `App` component is created. With these changes in place, our screen will now render both components, as shown in the following screenshot:

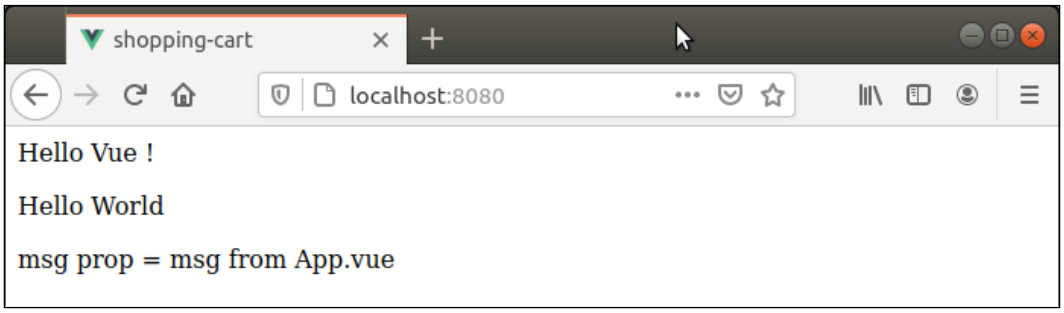


Figure 13.3: Vue application showing both parent and child components

Here, we can see that the `App` component is rendering the `"Hello Vue !"` text to the screen, and the `HelloWorld` component is rendering its template to the screen. We can also see the `msg` property that was set by the `App` component is being rendered correctly by the `HelloWorld` component.

Component state

The properties that a component exposes to a parent component are read-only. This means that only the parent component can change these properties, and that the flow of data is one-way, from parent to child. If our component needs to maintain its own state and update an internal property, such as the value from an input control, we will need a place to store this state. Vue components use a function named `data` that returns a structure for storing the internal state.

Let's update our `HelloWorld` component, and introduce an internal property named `myText` as follows:

```
<template>
    <p>Hello World</p>
    <p>msg prop = {{ msg }}</p>
    <input type="text" v-model="myText"/>
    <button v-on:click="clicked()" >Submit</button>
```

```
</template>

<script lang="ts">
import { Options, Vue } from 'vue-class-component';

@Options({
    props: {
        msg: String
    }
})
export default class HelloWorld extends Vue {
    msg!: string;
    myText!: string;
    data() {
        return {
            myText: this.msg
        }
    }
    clicked() {
        console.log(`this.myText = ${this.myText}`);
    }
}
</script>
```

Here, we have made a number of changes to our component. Firstly, we have introduced an `input` element within our HTML template that is using the `v-model` attribute to bind the value entered in the input control to a property named `myText`. This binding is a two-way binding, so the value shown on the screen and the internal value of the `myText` property are kept synchronized.

The second change we have made is to introduce a button element, and have set the `v-on:click` attribute to a method named `clicked()`. The `v-on` attribute is the method that Vue uses to trap DOM events, and in this case, we are trapping the `click` event.

Thirdly, we have introduced a class property named `myText` of type string on the `HelloWorld` class. The fourth change we have made is to create a member function named `data` that returns an object with a property named `myText`. The `data` function is used to define data that a Vue class needs for storing internal values. Note that we have also set the value of the `myText` property within the `data` function to the value of `this.msg`. This means that when the component is rendered, the default value of the internal `myText` property will be set to the incoming value of the `msg` prop, which is passed in from the parent component.

The final change we have made is to include a `clicked` member function on the `HelloWorld` class instance, which will be the event handler for our button. Within this function, we log the value of the `myText` state property to the console. Let's now fire up our browser, and modify the text in the input control to see what happens, as shown in the following screenshot:

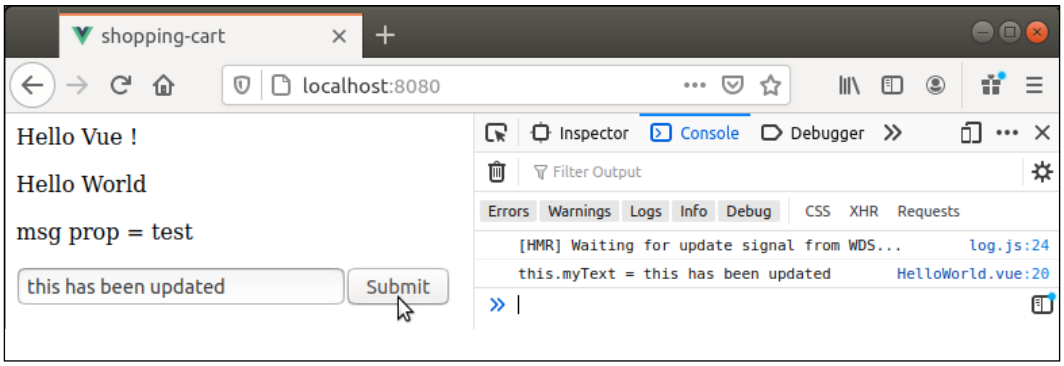


Figure 13.4: Vue application showing updates to an internal state property

Here, we can see that we have modified the value of the input control, and clicked on the **Submit** button. Our internal `myText` property has been updated, and is shown in the console logs.

Component events

A parent component sometimes needs to be notified when an event happens on a child component. In Vue, the mechanism for communication between children and parents is for the child to emit an event, and for the parent to supply a callback function for this event. As an example of this, let's update our `HelloWorld` component to emit an event when the **Submit** button is clicked, as follows:

```
@Options({
    props: {
        msg: String
    },
    emits: ["onSubmitClicked"]
})
export default class HelloWorld extends Vue {
    msg!: string;
    myText!:string;
    data() {
        return {
            myText: this.msg
```

```
        }
    }
    clicked() {
        console.log(`this.myText = ${this.myText}`);
        this.$emit("onSubmitClicked", this.myText);
    }

}
</script>
```

Here, we have made two changes to our `HelloWorld` component. Firstly, we have added an `emits` property to our `@Options` decorator object, and specified that this component will emit an event named `onSubmitClicked`. Secondly, we have called the `$emit` function on our instance of `this` within the `clicked` function, which is being called with two arguments. The first argument is the name of the event to emit, and the second argument is the parameters that will be used when this function is called. In this instance, we are sending the current value of the `myText` state property.

Now that we are emitting an event, let's take a look at how we specify a callback function that will react to this event in the `App.vue` parent component, as follows:

```
<template>
    Hello Vue !
    <HelloWorld
        @on-submit-clicked="onAppSubmitClicked"
        :msg="appMsg"
         >
    </HelloWorld>
</template>


<script lang="ts">
import { Vue, Options } from 'vue-class-component';
import HelloWorld from "./components/HelloWorld.vue";

@Options({
    components: {
        HelloWorld,
    },
})
export default class App extends Vue {
    appMsg:string = "test";
    data() {
```

```
        return {
            appMsg: String
        }
    }
    onAppSubmitClicked(msgValue: string) {
        console.log(`App:onSubmitClicked() : ${msgValue}`);
        this.appMsg = msgValue;
    }
}
</script>
```

Here, we have made three major changes to our `App.vue` component. Firstly, we have set an event handler attribute as part of our declaration of the `HelloWorld` component within our HTML template. This event handler is the attribute `@on-submit-clicked`, and is set to a function within our component class named `onAppSubmitClicked`. Note the kebab-case that we are using to name this attribute. The event that is being raised by the `HelloWorld` child component is named in its `emits` property as `"onSubmitClicked"`. The format of an attribute name for the event handler is to convert this `"onSubmitClicked"` name to kebab-case, so it must be named `"on-submit-clicked"`.

The second major change we have made is to convert the `msg` property that we are passing down into the `HelloWorld` component from a static string into a class state property named `appMsg`. Note that the syntax for the static string has been changed from `msg="msg from App.vue"` to the syntax for accessing a member variable, and is now `:msg="appMsg"`. This change will therefore inject the current value of the `App` component's `appMsg` property into the `msg` property for the `HelloWorld` child component. The structure of our `App` class has also been changed to include an `appMsg` class property, as well as defining a `data` function.

The third major change to our `App` class is to create the `onAppSubmitClicked` function, which is the callback function that will be invoked when the `HelloWorld` component emits the `onSubmitClicked` event. This function has a single parameter named `msgValue`, of type string. The function logs a message to the console, and then sets the value of the `appMsg` property on the `App` component to the incoming value. This change will trigger a change to the `msg` property on the `HelloWorld` component. Remember that data in a Vue application flows from parent to child, and as soon as the parent value changes, the child component will be re-rendered with the new value. We can see the effects of this change in the following screenshot:

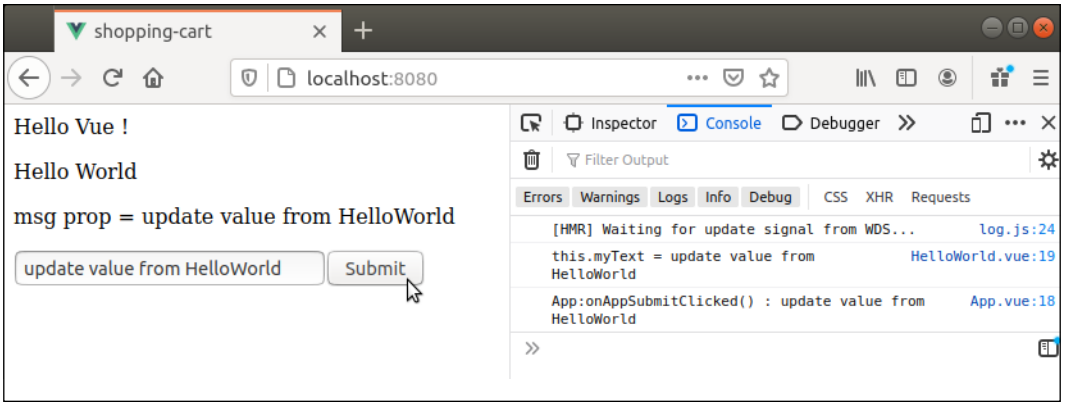


Figure 13.5: Vue component showing updates to the child component through a dynamic property

Here, we can see from the console messages what happens when we update the input control on the screen, and click the **Submit** button. Firstly, the `HelloWorld` component logs the updated `myText` value to the console, and then it emits an event. This event is received by the `App` component, which then updates its internal `appMsg` property. As this property is bound to the `msg` property of the `HelloWorld` component, the `HelloWorld` component is re-rendered to show the updated property.

Computed props, conditionals, and loops

Vue components also allow us to define what are known as computed properties, which are read only, and are derived from some component logic. Along with computed properties, we are also able to define conditional statements in our HTML templates, and deal with arrays of data, within a template loop. As an example of this, let's update our `HelloWorld` component as follows:

```
<template>
    <p>Hello World</p>
    <p>msg prop = {{ msg }}</p>
    <input type="text" v-model="myText"/>
    <button v-on:click="clicked()" >Submit</button>
    <div v-if="stringLength > 0">
        <p>text length is : {{ stringLength }}</p>
        <ul>words are :
            <li v-for="word in words">{{word}}</li>
        </ul>
    </div>
</template>
```

```
<script lang="ts">
import { Options, Vue } from 'vue-class-component';

@Options({
    props: {
        msg: String
    },
    computed: {
        stringLength() {
            return this.myText.length;
        },
        words() {
            return this.myText.split(' ')
        }
    },
    emits: ["onSubmitClicked"]
})
export default class HelloWorld extends Vue {
    ... existing class
}
</script>
```

Here, we have updated our HTML template and added a `<div>` element that is using a `v-if` attribute, which will render if the conditional statement returns true. Our conditional statement is checking whether the computed property named `stringLength` is greater than `0`. If it is, then the `<div>` will be rendered. Within this conditional element, we then render the value of the `stringLength` computed property, and then have a `<ul>` element that contains a `<li>` element. Note how we have specified a `v-for` attribute for the `<li>` element with a value of `"word in words"`. This is the mechanism that Vue uses to generate an HTML element for every item in an array.

Vue will interrogate the `words` array, which is also a computed property, and for each element in this array, will make the array element available through the variable `word`. In our template, we are just rendering the variable to the screen.

We have also updated our `HelloWorld` class through the `@Options` decorator and specified a property named `computed`. The `computed` property exposes two functions named `stringLength` and `words`, which are computed properties. The `stringLength` function returns the length of the `myText` property. Likewise, the `words` function returns an array for each string it finds in the `myText` property, by splitting it by the space character.

We can see this code in operation by firing up our browser, and entering a value in the input element, as shown in the following screenshot:

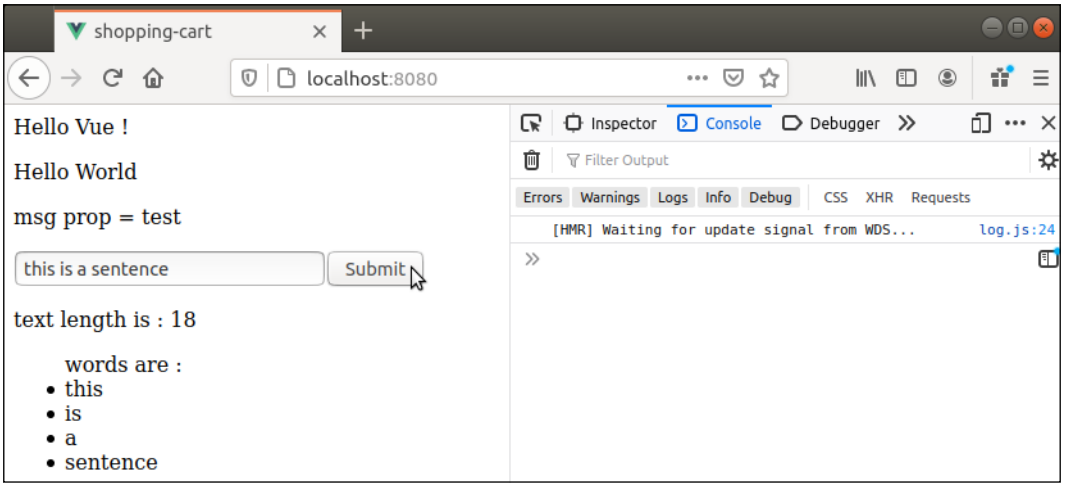


Figure 13.6: Vue component with computed properties, conditional elements, and loops

Here, we can see that the computed properties are returning the value of `18` for the `stringLength` property, and an array of strings for the `words` property. These values are based on the value of the internal `myText` property, which is bound to the input control. Note that if we clear the input control, such that it does not contain any text, then the `<div>` element based on our conditional statement will not show.

Another thing to note, which is not apparent in a static screenshot, is that as we type something into the input control, the values of the `stringLength` and `words` computed properties change on the fly. This behavior shows that Vue is responding to any change in our input control, which is bound to our `myText` property, and immediately re-computing any computed properties, as we type.

Thus far in this chapter, we have explored some of the main features of Vue applications, and worked through some examples of child components, props, internal state, events, and computed properties. We have also discussed conditional statements and how to loop through arrays in our component properties. In the next section of this chapter, we will put these techniques into use, and build an application.

A Vue application

In this section of the chapter, we will build a shopping cart application using the techniques that we have learned up until now with Vue. In our previous chapter, we built a product list and product details application using React, with the ability to specify how many of a particular item we would like to purchase.

This Vue application is intended to receive this information, and take over the display of a shopping cart. Our Vue application will need the ability to adjust the amount of a particular item we would like to purchase, as well as calculate the total cost (including tax and shipping) that a purchase would incur.

Application overview

An overview of the components we will build is shown in the following diagram:

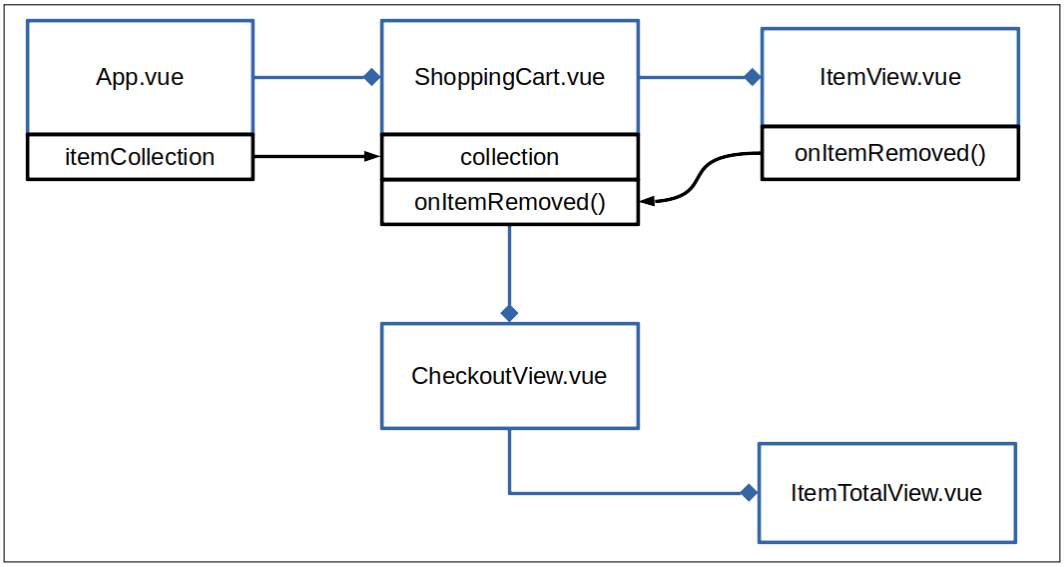


Figure 13.7: Logical view of Vue components for the shopping cart application

Here, we start with the `App.vue` component, which will instantiate a collection of items in our shopping cart, and store it as a local state property, named `itemCollection`. The `App` component will render a `ShoppingCart.vue` component, and pass in `itemCollection` as a property named `collection`. The `ShoppingCart` component has an internal state that will either show all items in the cart using the `ItemView` component, or show the `CheckoutView` component, which will be used to show tax amounts and shipping costs. The `ItemView` component will allow the user to modify the amount of a particular item they wish to purchase, and also remove an item from the cart. If an item is removed, an event will be emitted from the `ItemVue` component to the parent `ShoppingCart` component to remove the item. This event is named `onItemRemoved`.

The `CheckoutView` component will render an `ItemTotalView` component for each product in the cart, which will use computed properties to show the total cost for each shopping cart item.

The `ShoppingCart` component will render all items in the cart as shown in the following screenshot:

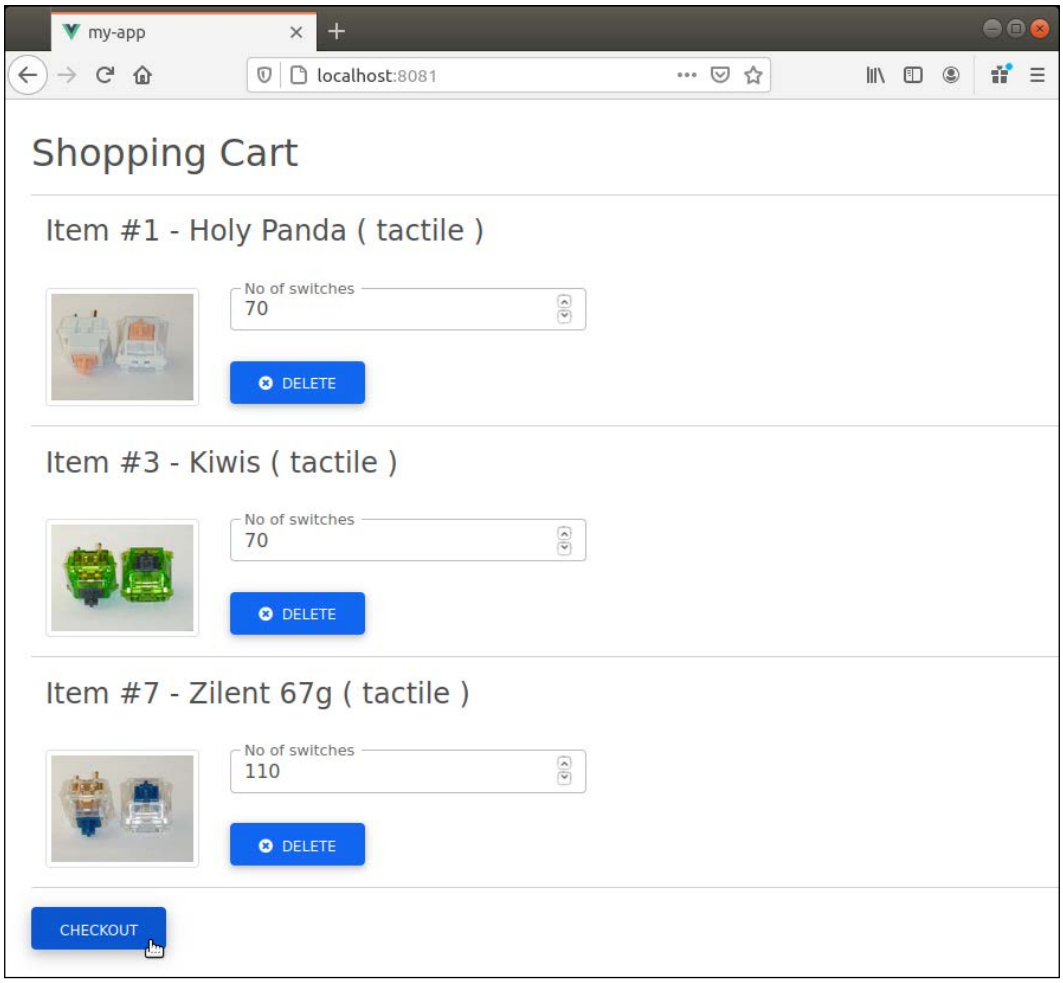


Figure 13.8: Vue shopping cart application showing all items in the shopping cart

Here, we can see that there are three items in our shopping cart, and each item shows the number of switches that have been added to the cart. There is a **DELETE** button for each item in the cart, allowing us to remove it from our cart, and we can also adjust the number of switches we would like to purchase. On the bottom of the screen is a **CHECKOUT** button, which will show the checkout screen as shown in the following screenshot:

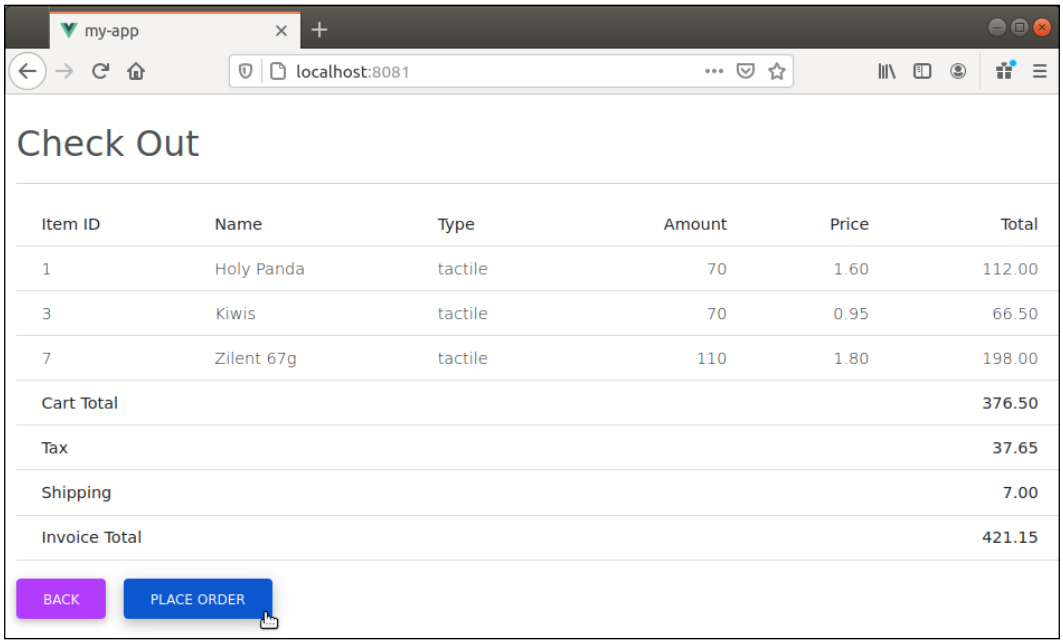


Figure 13.9: Vue shopping cart application showing the checkout screen, and total values

Here, we can see a tabular view of each item in our shopping cart, as well as the total amount for each item. We also have a **Cart Total** amount, which is used to calculate the amount of tax to charge. Add in some shipping costs, and we can show the **Invoice Total** for all items in the cart. We also have two buttons that are available on the screen, one to go back to our cart view and another to place an order.

Material Design for Bootstrap

Before we start building our application, we will install a CSS library named Material Design for Bootstrap. Bootstrap is a well-known set of CSS styles that have become popular for building web applications, and also includes some JavaScript to aid the creation of web components, such as buttons, input fields, and data pickers. We will use the Material Design version of this library to give our application the same look and feel as the Angular and React applications that also used Material Design. We can install it using npm as follows:

```
npm install bootstrap
npm install mdb-ui-kit
npm install @fortawesome/fontawesome-free
```

Here, we have installed `bootstrap` and `mdb-ui-kit`, as well as a set of free icons from Font Awesome. In order to include these libraries in our application, we can either import them as CSS or JavaScript scripts, or we can add the CDN versions to our `public/index.html` file, as follows:

```
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<link rel="icon" href="<%= BASE_URL %>favicon.ico">
<title><%= htmlWebpackPlugin.options.title %></title>
<script
type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/mdb-ui-kit/3.0.0/mdb.min.
js">
</script>

<link
href="https://cdnjs.cloudflare.com/ajax/libs/mdb-ui-kit/3.0.0/mdb.min.
css"
rel="stylesheet"
/>
</head>
```

Here, we have added a `<script>` tag within the `<head>` tag that points to the `mdb.min.js` file hosted on Cloudflare. We have also added a `<link>` tag to include the CSS found in the `mdb.min.css` file hosted on Cloudflare.

We will also need to use one of the JavaScript functions available in the `mdb` library, which currently does not have TypeScript types available for it. This means that we will need to declare the JavaScript module by editing the `shims-vue.d.ts` declaration file, and adding a line at the bottom, as follows:

```
declare module '*.vue' {
  import type { DefineComponent } from 'vue'
  const component: DefineComponent<{}, {}, any>
  export default component
}

declare module 'mdb-ui-kit/js/mdb.min.js';
```

Here, we have added a single line that declares a module named `'mdb-ui-kit/js/mdb.min.js'`, so that it will become available for importing later.

App component

Let's start our application by updating our `App.vue` component. The `App` component will just be the entry point to our application, and will hold the collection of items that are in our shopping cart. For the purposes of this application, we will just create a static set of items, in a file named `CartItems.ts`, as follows:

```
export interface IProduct {
    id: number;
    name: string;
    type: string;
    image: string;
    longDescription?: string;
    amount?: number;
    specs: ISpecs;
}
export interface ISpecs {
    actuationForce?: string;
    actuationPoint?: string;
    bottomOut?: string;
    bottomOutTravel?: string;
    price: string;
}

export class CartCollection {
    items: IProduct[];
    constructor() {
        this.items = [
            {
                id: 1,
                name: "Holy Panda",
                type: "Tactile",
                image: "holy_panda.png",
                amount: 70,
                specs: {
                    price: "1.60"
                }
            }
```

```
            ... other items
        ];
    }
}
```

Here, we have an interface named `IProduct` that represents a single item in our shopping cart, and has a number of properties that we will need, including the item's `id`, `name`, `type`, `image`, `longDescription`, and the `amount` of items requested. It also has a property named `specs`, of type `ISpecs`, that will contain the specification details of the item.

We also have a class named `CartCollection` that has a single property named `items`, which is an array of type `IProduct`. The `constructor` function creates the `items` array, and adds some items to it. This items array is set to a manually created list, but will be populated by an API call in the final version of the application. Again, we have not shown the full source of the constructor function for the sake of brevity.

Our `App.vue` component is as follows:

```
<template>
  <ShoppingCart :collection="cartItems"/>
</template>

<script lang="ts">
import { Vue, Options } from 'vue-class-component';
import { CartCollection } from "./CartItems";
import ShoppingCart from "./components/ShoppingCart.vue";

const shoppingCartItems = new CartCollection();

@Options({
    components: {
        ShoppingCart,
    },
})
export default class App extends Vue {
    data() {
        return {
            cartItems: shoppingCartItems
        }
    }
}
</script>
```

Here, we can see that the component's template just creates a `ShoppingCart` component, and binds the `collection` prop of the `ShoppingCart` component to the internal state property of the `App` component named `cartItems`. Within our script section of the file, we create a new instance of the `CartCollection` class, and then set the value of the `cartItems` state property to the class instance.

ShoppingCart component

The `ShoppingCart` component receives the collection of items in the cart, and renders either the shopping cart screen or the checkout screen. The option to show either screen is controlled by a single boolean value, which is either on or off. Let's take a look at the template section of the `ShoppingCart.vue` component first up, as follows:

```
<template>
    <div class="container">
        <div
            v-if="!isCheckingOut"
        >
         
        <h2>Shopping Cart</h2>
        <hr/>
        <div
            v-bind:key="item.id"
            v-for="item in collection.items"
        >
            <ItemView
                :item="item"
                @on-remove="onItemRemoved"
            ></ItemView>
        </div>
        <button
            class="btn btn-primary"
            v-on:click="checkout"
        >Checkout</button>
        </div>
        <div v-if="isCheckingOut">
                 
        <h2>Check Out</h2>
        <hr/>
        <CheckoutView
            :basket="collection"
        ></CheckoutView>
```

```
            <button
                class="btn btn-secondary"
                v-on:click="back()"
            >Back</button>
             
            <button
                class="btn btn-primary"
            >Place Order</button>
            </div>
        </div>
    </template>
```

Here, our template starts with a `<div>` element with the Bootstrap class of `"container"`. We then have a `<div>` element that is toggled on or off based on the property `isCheckingOut`, using the `v-if` directive. If the screen is not in checkout mode, we loop through the items in the internal state property named `collection.items`, and render an `ItemView` element for each one of them. Note that we have bound the `item` property of each `ItemView` component to the current item in the array. We also have an event listener, which will trigger the `onItemRemoved` callback function if a user decides to remove the item from their shopping cart. We then have a button named `Checkout` that will call the `checkout` function when clicked.

We then have another conditional `<div>` element, which is checking the value of the `isCheckingOut` property, and if true, will render the `CheckoutView` component. The `CheckoutView` component exposes a prop named `basket`, which is bound to the `collection` property of the `ShoppingCart` component. Finally, we have a button that will call the `back` function when clicked.

Let's now take a look at the script section of the `ShoppingCart` component, as follows:

```
@Options({
    props: {
        collection: CartCollection,
    },
    components: {
        ItemView,
        CheckoutView,
    }
})
export default class ShoppingCart extends Vue {
    collection!: CartCollection;
    isCheckingOut!: boolean;
    data() {
```

```
        return {
            collection: this.collection,
            isCheckingOut: false,
        };
    }

    onItemRemoved(id: number) {
        const index =
            this.collection.items.findIndex(
                (item) => item.id === id
            );
        this.collection.items.splice(index, 1);
    }
    checkout() {
        this.isCheckingOut = true;
    }
    back() {
        this.isCheckingOut = false;
    }
}
```

Here, we can see that the `ShoppingCart` component is exposing a prop named `collection`, of type `CartCollection`. The component also has two internal properties, named `collection` and `isCheckingOut`, and the `data` function sets both values. Note that the `collection` prop that is exposed by the component has the same name as the internal state property named `collection`. Our component also has an `onItemRemoved` callback function, which will find the index within the collection with the correct ID, and remove it.

Finally, our component has a `checkout` function and a `back` function, which just toggle the value of the `isCheckingOut` property.

ItemView component

The `ItemView` component is responsible for displaying a single shopping cart item, with an image of the item, as well as an input control for the number of switches in the cart. Each item also has a **Delete** button, which can be used to remove the item from the cart.

We will not show or discuss the full HTML template, as it is quite lengthy, but will instead focus on a single element of the template, which is the `input` element, as follows:

```
<div
    class="form-outline"
    ref="inputAmount"
>
    <input
        type="number"
        class="form-control"
        v-model="item.amount"
    />
    <label
        class="form-label"
    >No of switches</label>
</div>
```

Here, we have a `<div>` element with the class of `"form-outline"`, and an attribute named `ref`, which is set to the value `"inputAmount"`. Vue allows us to generate a reference to a particular HTML element that we may need to access within our code. These references are similar to finding a DOM element using CSS searches, or searching by `id`. The reason that we need a reference to this element is due to the way that the Material Design for Bootstrap controls work.

Our `input` element has an attribute named `type`, which is set to `"number"`, and it also has a class of `"form-control"`. We then follow this `input` element with a `label` element, with the class of `"form-label"`, and the text value of `"No of switches"`. When the `mdb` library initially renders this label and input control, it assumes that the user has not entered a value as yet, and renders the label directly on top of the input control. This effect can be seen in the following screenshot:

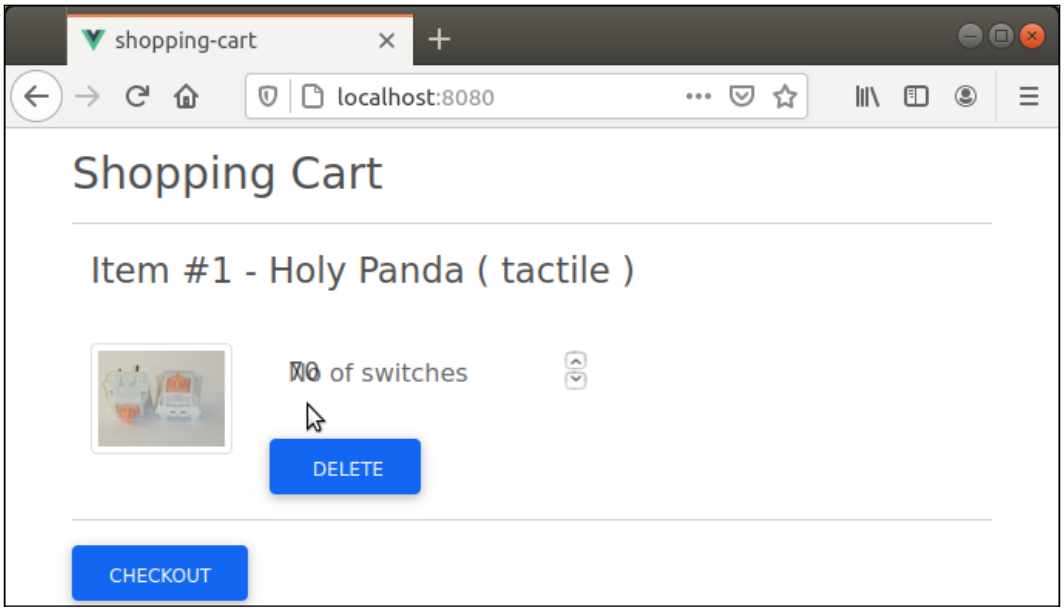


Figure 13.10: Material Design for Bootstrap showing a label on top of an input control

Here, we can see that the text `"No of switches"` is overlapping the input control value of `70`, which becomes unreadable. The reason for this issue, according to the documentation, is because we have set an initial value for the input control. The solution is to initialize the control using a bit of JavaScript that the `mdb` library provides. Let's take a look at the `ItemView` component, to see how this works, as follows:

```
import * as mdb from 'mdb-ui-kit/js/mdb.min.js';

@Options({
    props: {
```

```
        item: Object
    },

    emits: ["onRemove"],
    computed : {
        imageSource() {
            return `images/${this.item.image}`;
      }
    },
    mounted() {
        let inputBts = this.$refs.inputAmount;
        new mdb.Input(inputBts).init();
    }
})
export default class ItemView extends Vue {
    item!: IProduct;
    data() {
        return {
            item: this.item,
        };
    }
    onItemRemove() {
        this.$emit("onRemove", this.item.id);
    }
}
```

Here, we start by importing the `mdb.min.js` file, which will allow us to reference the `mdb` library using the prefix `mdb`. Our `@Options` decorator then defines a `props` property, an `emits` property, and a `computed` property, which define parts of our component.

Note, however, that we have included a function in our `@Options` structure named `mounted`. The `mounted` function will be called once our Vue component has been rendered to the screen, and is part of the Vue component life-cycle. There are other methods that are available on the component life-cycle, including `created` and `updated`, which we do not need to use here. The `mounted` function creates a local variable named `inputBts`, which is set to the value of `this.$refs.inputAmount`. The `this.$refs` component property gives us access to any HTML element that we have set a reference to in our template. So in essence, the `inputBts` variable is the DOM element with the `ref="inputAmount"` attribute.

Once we have a reference to this DOM element, we create a new `mdb.Input` class, passing in the DOM element in the constructor, and then call the `init` function. This has the effect of initializing the input control, such that the label is correctly displayed above the value of the input control, as can be seen in the following screenshot:

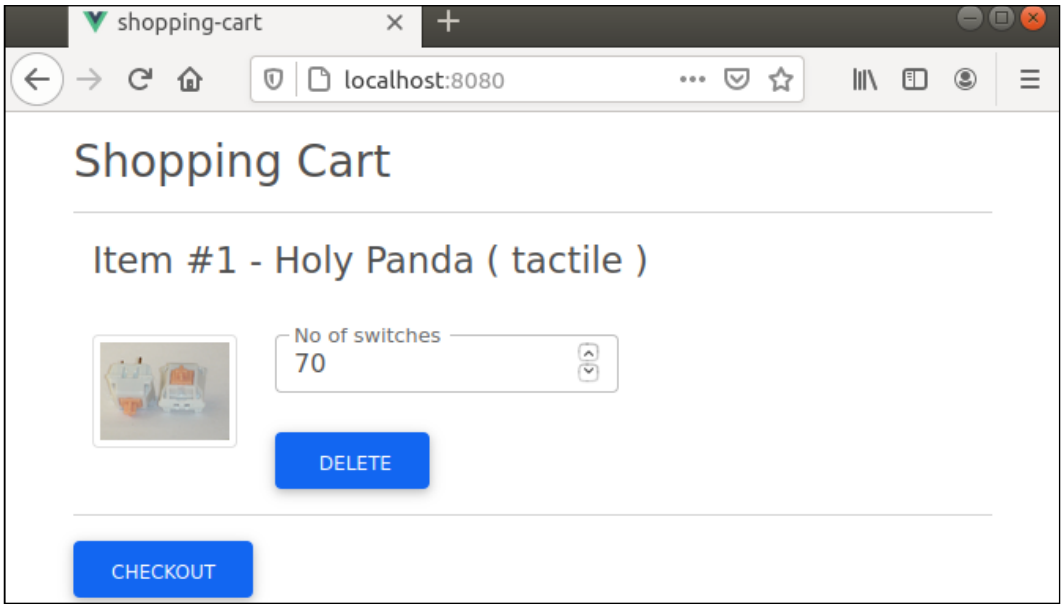


Figure 13.11: Material Design for Bootstrap input control displaying correctly after initialization

Here, we can see that the input control and the label have been initialized correctly, and are therefore displaying correctly.

The rest of the `ItemView` component is fairly straightforward, and includes an `item` state property, as well as an `onItemRemove` function that will emit an event to the parent control when the **DELETE** button is clicked.

CheckoutView component

The second-to-last component that we will need is the `CheckoutView` component. Remember that this component is displayed by the `ShoppingCart` component when the user clicks on the **CHECKOUT** button. There is nothing unusual about this component, other than that the template uses a table to display the items in the shopping cart, and has a number of computed properties. Again, in the interest of brevity, we will not show the entire template for this component; please refer to the sample code for this.

Our `CheckoutView` component calculates a few values. The first of these values is the total amount of all items that are in the shopping cart, as follows:

```
totalValue() {
    let total = 0;
    for (let item of this.basket.items) {
        total += <number>item.amount * +item.specs.price;
    }
    return total * 100;
}
```

Here, we have a variable named `total` that is set to `0`. We then loop through each `item` within our shopping basket, and multiply the number of switches that the user has requested by the `item.specs.price` of each switch. This value is added to our grand total. Note that we are multiplying the returned value by `100`, in order to return an integer that is calculated in cents. This makes it easier to calculate tax, as can be seen in the function that calculates it, as follows:

```
taxValue() {
    return Math.round(this.totalValue() * 0.1);
}
```

Here, we have the definition of the function that calculates the total tax amount of our shopping cart. We are just calculating `10` percent of the total value of the shopping cart, and rounding this value up to the nearest integer. When we need to display a value on the screen, we use another function for this, as follows:

```
displayValue(amount: number): string {
    return (amount / 100).toFixed(2);
}
```

Here, we have a class member function named `displayValue`, which takes a number as a single argument. The function then divides the input amount by `100`, and then calls the `toFixed` function to return a value that can be displayed on the screen. This function will take care of the value `0.1` being displayed as `0.10`, or the value `1.5` being displayed as `1.50`, which we are used to when dealing with invoices and values.

The `CheckoutView` component will generate a table row for each item in the shopping cart, as can be seen in the template, as follows:

```
<tr v-for="item in basket.items" v-bind:key="item.id">
    <ItemTotalView :item="item"></ItemTotalView>
</tr>
```

Here, we can see that we are using a `v-for` attribute to loop through all of the array items in `basket.items`. Each array element will render an `ItemTotalView` component.

ItemTotalView component

The final component in our Vue application is the `ItemTotalView` component. This component is responsible for rendering a table row for each item in our shopping basket. Again, for the purpose of brevity, we will not show the entire component, as it is fairly simple, so please refer to the sample code for the full listing. `ItemTotalView` uses the following template:

```
<template>
    <td scope="col">{{ item.id }}</td>
    <td scope="col">{{ item.name }}</td>
    <td scope="col">{{ item.type }}</td>
    <td scope="col"
        class="right-aligned-text">
        {{ item.amount }}
    </td>
    <td scope="col"
        class="right-aligned-text">
        {{ itemValueFormatted }}
    </td>
    <td scope="col"
        class="right-aligned-text">
        {{ itemTotalValue }}
    </td>
</template>
```

Here, we can see that this component renders a `<td>` HTML element for each property in our shopping cart item. There are two computed properties that are rendered, namely `itemValueFormatted` and `itemTotalValue`. The `itemValueFormatted` property will convert the price of the item from `1.5` to `1.50` for display purposes. `itemTotalValue` will calculate the total price of the item, by multiplying the number of switches specified by the price of each switch.

With the `CheckoutView` and `ItemTotalView` components in place, our application now renders the price of the shopping cart as shown in the following screenshot:

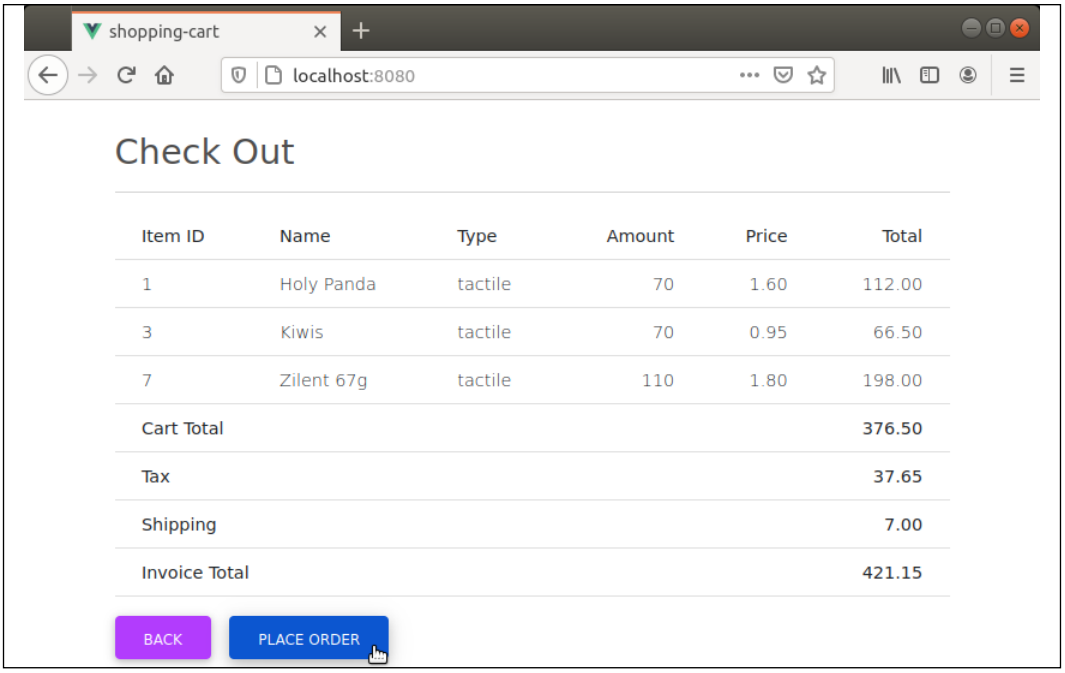


Figure 13.12: Vue CheckoutView component rendering the invoice totals

Here, we can see that the `CheckoutView` component is rendering an individual `ItemTotalView` component for each item on our shopping cart, and calculating the totals as expected.

Our Vue shopping cart application is now complete.

Summary

In this chapter, we have explored the Vue JavaScript framework, and built a shopping cart application using it. We set up a Vue development environment, and then explored the structure of a Vue application that uses props and state to manage data between parent and child components. We then discussed DOM events, and how components can communicate by emitting custom events. To round out the first section of this chapter, we discussed computed properties, conditional template elements, and loops.

In the second section of this chapter, we built a shopping cart application using Vue, and broke down the application into various components. In the final chapter of this book, we will combine the Angular application that we have built with the React product view application, and then the Vue shopping cart application, to see how we can build applications using micro front-ends.

14

Node and Express

JavaScript has traditionally been used within web browsers in order to enhance or improve the usability or style of web pages. With the growth of the web, each browser competed to provide the best and fastest JavaScript engine they could. One of these JavaScript engines was the V8 engine, which was initially built for Google Chrome, and was released as open source in 2008. Using this engine, Ryan Dahl wrote a JavaScript engine that could be used as a web server, and run on the command line, named Node.js, or simply Node. A year after Node was released, the first versions of the Express framework for Node were released, which provided a set of features to simplify building server-side applications using JavaScript.

Node, using the single-threaded execution of JavaScript code, is able to handle thousands of concurrent web server requests from servers that are relatively small and cheap. Using the callback mechanisms of JavaScript, this also means that programmers do not need to think about threading, or the many side effects of multi-threaded applications. There are many large-scale corporations that use Node to run their mission-critical websites.

In this chapter, we will take a quick tour of Node and Express, and show how to build a web server with just a few lines of code. We will then build a functional Express application that will use the Handlebars template engine to render different HTML pages, and show how to redirect a browser to a different internal URL. Finally, we will explore the handling of JSON data as part of a POST operation, and look at using sessions to store data between requests.

Express introduction

In this section of the chapter, we will set up a Node and Express development environment, and show how to build the simplest web server with just a few lines of code. We will then discuss routes, and show how to split up our code base into modules for ease of maintenance and readability. Finally, we will discuss how to set configuration parameters for a Node application.

Express setup

In order to build a Node and Express application, we just need to initialize a Node environment, and install a few npm packages, as well as their corresponding declaration files as follows:

```
mkdir node-express-app
cd node-express-app
npm init
npm install express
npm install @types/express --save-dev
```

Here, we have created a directory named `node-express-app`, changed into this directory, and then initialized a Node environment. We then install the `express` module using `npm`, and install the TypeScript declaration files for Express, as in `@types/express`. We will also need to initialize a TypeScript environment within the same directory, which can be done as follows:

```
tsc --init
```

We can now write a very minimal web application using Express, by creating a file named `minimal_app.ts` with the following content:

```
import express from "express";

const app = express();

app.get("/",
    (
        req: express.Request,
        res: express.Response
    ) => {
        console.log(`request URL : ${req.url}`);
```

```
        res.send(`Hello Express !`);
    });

app.listen(3000, () => {
    console.log(`listening on port 3000`);
});
```

Here, we start by importing the `express` module, and then creating a variable named `app`, which is set to the result of invoking the `express()` function. This will initialize the Express engine, and allow us to call some functions on the `app` instance variable.

The first function that we call on the `app` instance is the `get` function, which will handle an HTTP GET request for the path that we specify as the first argument, which in this case is just `"/"`. The second argument to the `get` function is a callback function that will be invoked when the browser issues a GET request that matches the path specified. This callback function has two parameters, named `req` and `res`. The `req` parameter is of type `express.Request`, and is the incoming HTTP request. The `res` parameter is of type `express.Response`, and is the outgoing HTTP response.

Our callback function is logging a message to the console with the value of the `url` property from the original HTTP request. It is then calling the `send` function on the HTTP response object with the text `"Hello Express !"`. The effect of this `send` call is that an HTTP response will be sent back to the browser with the text provided. This request-response pattern is the nature of web servers. They receive an HTTP request, interpret what the request is looking for, and then return an HTTP response.

The last thing that we do within our Node application, which can be seen on the last lines of this code snippet, is to call the `listen` function on the `app` instance, and specify a port to listen on. This `listen` function will start our web server application, effectively listening on the specified port for any HTTP requests. The `listen` function can also specify a callback function to execute once the `listen` command is running. Within the callback function that we have defined, we just log a message to the console.

We can now compile and run our newly minted Express web application as follows:

```
tsc
node minimal_app
```

Here, we are invoking the TypeScript compiler to compile our application, and are then using the Node engine to execute the `minimal_app.js` file. We will then see the following log on the console:

```
listening on port 3000
```

Here, we can see that our application is running, and listening for HTTP requests on port `3000`. We can now fire up a web browser, and point it to `http://localhost:3000`, as shown in the following screenshot:

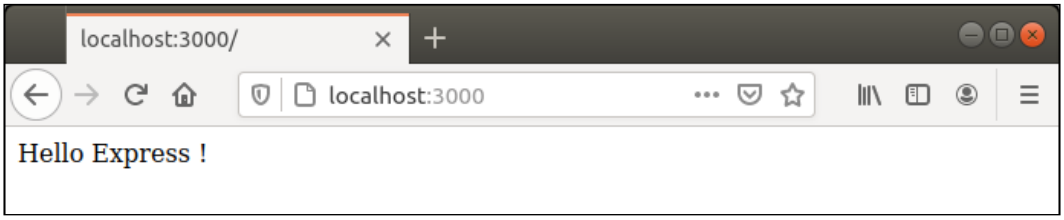


Figure 14.1: Minimal Express web server serving a simple message

Here, we can see that an HTTP request to the server running at `localhost:3000` responds with the text `"Hello Express !"`. Note too that the console log on our command line logs the request URL as follows:

```
listening on port 3000
request URL : /
```

Here, we can see that the value of the request's `url` property as seen by our Express GET handler is `"/"`.

Express router

A large-scale web application would have many different paths for many different web pages. We may have a `/login` URL path to render a login page, and a `/products` URL path to show a list of products, for example. Each of these URLs may have to handle GET requests, POST requests, PUT requests, DELETE requests, or PATCH requests. If we had an application that had hundreds of endpoints, then having each handler of an Express application within the same file would quickly become a maintenance nightmare. Express provides a `Router` object that can be used to split these routes into separate files, in such a way that all requests that use the `/login` URL, for example, can be handled by code within a single file, and all requests for `/products` can be handled by code within another file.

Let's create a `routes` directory within our application, and create two route files named `index.ts` and `login.ts` to explore this technique. Firstly, the `index.ts` file, as follows:

```
import express from "express";

let router = express.Router();

router.get(`/`,
```

```
    (
        req: express.Request,
        res: express.Response
    ) => {
        res.send(`Index module processed ${req.url}`);
    });

export { router };
```

Here, we start by importing the `express` module, as we have done previously. We then call the `Router()` function on the `express` module, and assign this a local variable named `router`. This `Router()` function acts like a singleton instance, meaning that the call to `express.Router()` returns the same router instance no matter where it was called from. In this way, we can attach new routes to the same global Express route handler. We then call the `get` function on the router instance, and supply two arguments, which are a string with the route matcher, and a callback function. Within this callback function, we are just returning a string to the browser with a message and the value of the `url` property of the request.

The `get` function sets up a route handler for an HTTP GET request. This means that when a GET request comes in for the URL at `"/"`, this route handler will be executed. Express provides other functions to handle all other combinations of HTTP requests, such as the `post` function to handle POST requests, and the `put` function to handle PUT requests.

Note the last line of this code snippet, where we are exporting the variable named `router`. This makes the changes that we have made to the global Express router within this file available to the application as a whole.

Let's now look at the login module, within the file `login.ts`, which is an almost identical route handler, as follows:

```
import express from "express";

let router = express.Router();

router.get(`/login`,
    (
        req: express.Request,
        res: express.Response
    ) => {
        res.send(`Login module processed ${req.url}`);
    });

export { router };
```

Here, we have a standard route configuration, with only two differences in the code, compared to our `index.ts` module. The first difference is that the `get` function now uses the string value of `"/login"` as its first argument, meaning that any GET requests to a URL of `/login` will be handled by this route handler. The only other difference is in the message that is sent back to the browser, in the call to the `res.send` function. This message now starts with the text `"Login module"`, instead of `"Index module"`.

In order to use these route modules, let's now create a new application file in the project base directory named `main.ts`, as follows:

```
import express from "express";

let app = express();

import * as Index from "./routes/index";
import * as Login from "./routes/login";

app.use(`/`, Index.router);
app.use(`/`, Login.router);

app.listen(3000, () => {
    console.log(`listening on port 3000`);
});
```

Here, we have a minimal Node Express application that imports the `express` module, and sets up a local variable named `app`, as we have seen earlier. We then have two `import` statements, which are importing our `index` and `login` modules. Once these modules have been imported, we then call the `use` function on our `app` variable in order to register them as route handlers. Note that we are referencing the exported `router` variable from each module, as seen in the use of `Index.router` and `Login.router`. The `app.use` function is essentially registering each of these route handlers.

Our application then listens for HTTP requests on port `3000`. If we fire up a browser now, we will see the `index` router handle our initial request, as shown in the following screenshot:

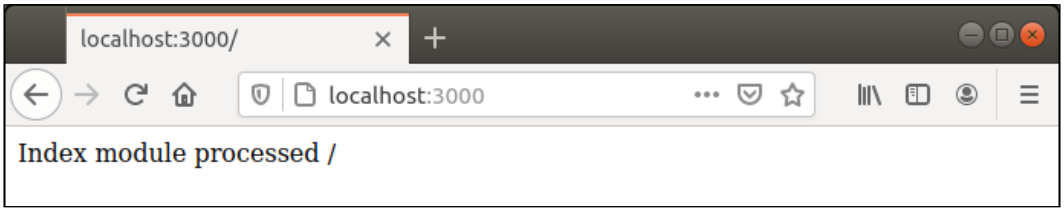


Figure 14.2: Express index router handling a GET request at the root URL

Here, we can see that the `index.ts` router has handled this HTTP request. If we now type in the `/login` URL, we will see the `login.ts` router handling the request, as shown in the following screenshot:

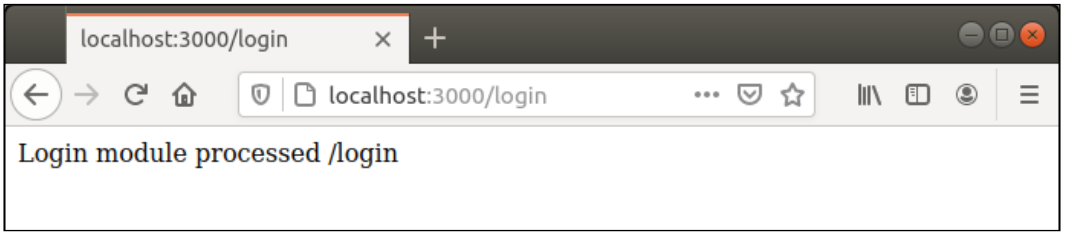


Figure 14.3: Express login router handling a GET request for the /login URL

Here, we can see that the `login.ts` router has handled this HTTP request, as the URL that we typed in is `localhost:3000/login`.

Express configuration

Currently, the port that our Express application is listening on is hard-coded to port `3000`. In a production-like environment, this could be set to port `80`, or for an application using secure communication to port `443`, which is the default HTTPS port, or any other port, for that matter, if some kind of proxy server is being used. Ideally, the port that this application is listening on should be controlled by a configuration file, or a configuration setting of some sort, instead of being hard-coded. We really should move settings like these into a configuration file of some sort that the application can consume.

Fortunately, setting and using configuration settings in an Express application is relatively simple, with the use of purpose-built libraries, such as the config library. Let's go ahead and install the config library for this purpose, using npm as follows:

```
npm install config
```

We will also need the associated type information, which can be installed as follows:

```
npm install @types/config --save-dev
```

With the config library installed, we can now update our `main.ts` file as follows:

```
... existing code

import config from "config";

enum ConfigOptions {
```

```
    PORT = 'port'
}

let port = 3000;

if (config.has(ConfigOptions.PORT)) {
    port = config.get(ConfigOptions.PORT)
} else {
    console.log(`no port config found, using default ${port}`);
}

app.listen(port, () => {
    console.log(`listening on port ${port}`);
});
```

Here, our changes start with an import of the config library, and an `enum` definition named `ConfigOptions`, which has a single value named `PORT`. This `'port'` configuration option will be used to define the port number that the application will listen on. This will give us the ability to vary the listening port that our application uses, which could be `3000` for a development environment, or `443` for a production environment.

Our code then defines a local variable named `port`, and sets its default value to `3000`. We then have an `if` statement that is checking if the `config.has` function returns true for the enum value of `ConfigOptions.PORT`. If it returns true, we set the value of the `port` variable to the result of calling the `config.get` function. If the `config.has` function call returns false, we will log a message to the console, and leave the default value as it is. Note that our `app.listen` function call is now using the local `port` variable, instead of the hard-coded value of `3000`.

What we have effectively done by using the `config` library is to ask it if it has found a configured value for the string `'port'`, and if so, we will use it. If it has not found a configured value, we will use the default value of `3000`. There are a number of ways to set this `'port'` value, with the simplest method being to create a directory named `config`, and within it a file named `default.json`, containing the following:

```
{
    "port" : 9999
}
```

Here, we have specified that the value of the `'port'` setting used by the config library should be set to `9999`. If we now fire up our Express application, we will see that this configuration setting has been applied, as follows:

```
node-express-app$ node main
listening on port 9999
```

Here, we can see that when we start the application with the command of `node main`, the `config/default.json` config file has been consumed by our application, and the `'port'` configuration has been applied. This is why the application is now running on port `9999`.

The config library can be used for any sort of application configuration, which may include things like setting the date format, time format, or even the time zone that an application is running in. Having an external file that contains all of this configuration information allows us to tweak our application for any environment it needs to run in.

An Express application

Now that we have an idea about the basics of Express routing and setup, let's build a two-page application that handles a user login form. Along the way, we will learn about how Express renders application HTML pages, how they can be combined with a generic layout page, and how to serve static files such as CSS and icons. We will also discuss how to handle form input, and how to use session data.

Express templating

Our current route handlers are returning simple messages to the browser. In a real-world application, however, we will need to render complete HTML pages, with a standard HTML structure, including links to CSS stylesheets if necessary, and a header section. Express uses a template engine that allows us to specify the HTML we need for each page, and also provides the mechanism of injecting run-time values into these templates, similar to the template mechanisms of Angular, React, or Vue.

Express supports many different templating engines, including Pug, Mustache, Jade, Dust, and many more. Introducing a template engine into our Express application is a matter of installing the engine of choice via `npm`, and configuring Express to use it.

In this application, we will use the Handlebars template engine. Handlebars uses standard HTML snippets, along with substitution variables using a double brace syntax, in order to inject values into our HTML. This is similar to what we have seen with our SPA frameworks. Some template engines, such as Pug and Jade, use their own custom formats to represent HTML elements, and are a mix of HTML keywords, class names, and variable substitution formats. As a quick comparison, consider a Handlebars template as follows:

```
<!DOCTYPE html>
<html>
<head>
    <title>{{title}}</title>
    <link rel="stylesheet" type='text/css' href="/css/styles.css" />
</head>
<body>
    {{{body}}}
</body>
</html>
```

Here, we have a Handlebars HTML template that looks very much like standard HTML, with a few substitution variables, such as `{{title}}` and `{{{body}}}`. Again, this looks very similar to the Angular, React, or Vue HTML templates that we have already seen. Note that a similar Jade template would be as follows:

```
doctype
html
  head title #{title}
  link(rel='stylesheet', href='/stylesheets/style.css')
    body
```

Here, we can see that the template does not look like standard HTML. We do not have tags that are enclosed with angled brackets, such as `<html>` or `<body>`. Instead, we have keywords that look familiar, such as `html`, `head`, and `link`, but they seem to be following their own syntax rules. Jade is designed in such a way that it will generate valid HTML based on the keywords found in the template.

While this Jade template may save us a lot of typing, it does mean that we will need to learn and understand the various keywords and subtle syntax used in Jade, in order to render valid HTML. This, therefore, adds another layer of complexity. For the sake of simplicity, then, and to avoid learning a completely new syntax for HTML templates, we will use Handlebars as our template engine.

Handlebars configuration

Handlebars can be installed via `npm` as follows:

```
npm install hbs
```

Once Handlebars has been installed, we can configure it for use within our application by modifying our `main.ts` file as follows:

```
import express from "express";

let app = express();

import * as Index from "./routes/index";
import * as Login from "./routes/login";

import * as path from "path";
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

app.use(`/`, Index.router);
app.use(`/`, Login.router);

   ... existing config code

app.listen(port, () => {
    console.log(`listening on port ${port}`);
});
```

Here, we have added three lines to our `main.ts` file. The first is to `import` the `path` module, which is part of the Node engine. The `path` module allows us to use several handy functions when working with directory path names within Node. We can use the `__dirname` variable, which holds the full path name of the current directory, in a call to the `path.join` function, on the second line that we have added to this file.

The `path.join` function will return the full path name to a local `views` directory. We are then setting the `'views'` global Express parameter to this directory. This configures Handlebars to use this path when it is looking for template files.

The final change to the `main.ts` file is to call `app.set` with the argument `'view engine'`, and the value `'hbs'`. This call configures our Express application to use Handlebars as the template engine.

Using templates

Now that we have registered a template library, we can update our routes to render a template, instead of returning a simple message to the browser. Let's update our `routes/index.ts` file as follows:

```
import express from "express";

let router = express.Router();

router.get(`/`,
    (
        req: express.Request,
        res: express.Response
    ) => {
        res.render('index',
            {
                title: 'Express App',
                welcomeMsg: 'Welcome to the Express App'
            }
        )
    });

export { router };
```

Here, we have replaced the call to `res.send` that we used earlier, with a call to the `res.render` function. This function takes two arguments. The first argument is the name of the view to render, which is `'index'`, and the second argument is an object with properties that we may wish to render on the screen. We have specified two properties here, named `title` and `welcomeMsg`, which both contain string values.

Let's now create the corresponding index view template file, by creating a `views` directory, and within it, a file named `index.hbs`. Note that Handlebars uses the `.hbs` file extension for template files, but they are referenced in our `res.render` function without this `.hbs` extension. The contents of the `index.hbs` file is as follows:

```
<h1>{{welcomeMsg}}</h1>
```

Here, we have a single `<h1>` element, which will use template substitution to replace the `{{welcomeMsg}}` tag with the value of the `welcomeMsg` property that we specified within our route handler. With the template in place, if we rebuild our Node application and refresh our browser, we will see the message rendered, as shown in the following screenshot:

Figure 14.4: Express application using a Handlebars template to display a message

Here, we can see that our Express application has applied the `index.hbs` template to render the message to the screen.

Our rendered HTML page is starting to come together, but still needs a proper structure, including the `<doctype>`, `<head>`, and `<body>` elements, for it to be complete. Handlebars allows us to specify a base layout that can be used for all pages, which is by default named `layout.hbs`. Let's create this `layout.hbs` file in the `views` directory, as follows:

```
<!DOCTYPE html>
<html>
<head>
    <title>{{title}}</title>
    <link
        rel="stylesheet"
        type='text/css'
        href="/css/bootstrap.min.css" />
</head>
<body>
    {{{body}}}
</body>
</html>
```

Here, we have defined the base layout page template that all views will use. Note that the `{{{body}}}` tag uses a triple brace syntax, which will inject whatever template we specify into the `<body>` element of this template. We have also included a `<title>` tag, which allows us to inject the title of the page using our standard template syntax. Our template also includes a `<link>` tag that is pointing to a file named `css/bootstrap.min.css`.

With this base layout page in place, our page now renders with a title, as shown in the following screenshot:

Figure 14.5: Express application showing the page title from the default layout view

Here, we can see that the page title is now **Express App**, which is specified as the `title` property in our `index.ts` route handler, and is being rendered by the default `layout.hbs` view.

Static files

Our page template includes a `<link>` element that is referencing a file named `css/bootstrap.min.css`. This is a static resource file, in that it is not rendered by any logic; it is served as is. We will need to configure our Express app to serve these static files, which we can do in our `main.ts` file as follows:

```
.. existing code

app.use(`/`, Index.router);
app.use(`/`, Login.router);

app.use(express.static(path.join(__dirname, 'resources')));

app.listen(port, () => {
    console.log(`listening on port ${port}`);
});
```

Here, we have added another call to the `app.use` function to our code in our main application. We have provided a single argument to the `app.use` function, which is a call to the `express.static` function. We have provided a single argument to the `express.static` function, which is the full path to a directory named `resources`, again using the `path.join` function as we did earlier. This will configure our Express application to serve static files from the `resources` directory. Our HTML template is looking for the `bootstrap.min.css` file, which is part of the Bootstrap library, so let's install it as follows:

```
npm install bootstrap
```

We can now create the `resources` directory, and within it a `css` directory, as follows:

```
mkdir resources
cd resources
mkdir css
cd css
cp ../../node_modules/bootstrap/dist/css/bootstrap.min.css .
```

Here, we are just creating the `resources/css` directory, and copying the `bootstrap.min.css` file from the `node_modules` directory into it. If we now stop and start our Express server, and then refresh our browser, we will see a minor change to our web page, as shown in the following screenshot:

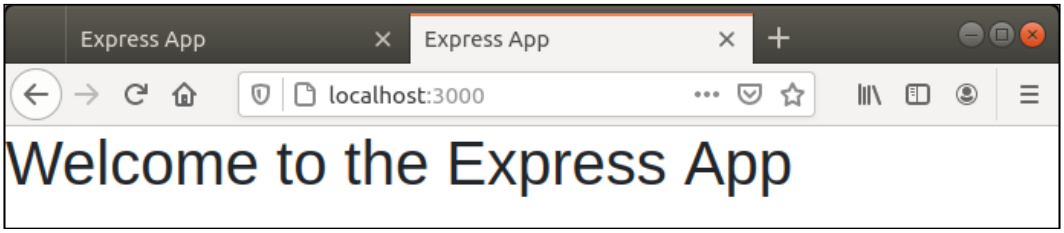


Figure 14.6: Express application showing content with the Bootstrap styles applied

Here, we can see that the font for our text has changed very slightly, which is a result of applying the `bootstrap.min.css` styles. This file is a static file that has been served up by our Express application through the static file configuration.

Express forms

Let's now take a look at how Express can interpret data that is submitted from a form on a web page. To do this, we will create a `login.hbs` template file, and modify our `login.ts` route handler to accept a POST request, along with the existing GET request. We can create a file in the `views` directory named `login.hbs` as follows:

```
<h1>Login</h1>
<form method="post">
    <p>{{errorMessage}}</p>
    <p>Username :
        <input name="username"/>
    </p>
    <p>Password :
        <input name="password" type="password"/>
    </p>
```

```
    <button
        type="submit"
        class="btn btn-primary">
        Login
    </button>
</form>
```

Here, we have an HTML template that contains a form. Within this form, we have a paragraph element that will show the `errorMessage` property, if it exists. We then have an input control with the name attribute set to `username`, and another input control with the name attribute set to `password`. Finally, we have a button control with a type attribute of `submit`, and the text `Login`.

We will now need to update our `routes/login.ts` file to render this template as follows:

```
import express from "express";

let router = express.Router();

router.get(`/login`,
    (
        req: express.Request,
        res: express.Response
    ) => {
        res.render(`login`, {
            title: `Express Login`
        });
    });

export { router };
```

Here, we have made a minor change to our `login.ts` file, in order to render the `'login'` template, with the `title` property set to the value `'Express Login'`.

If we point our browser to the URL `localhost:3000/login` now, we will see the form rendered to the screen as shown in the following screenshot:

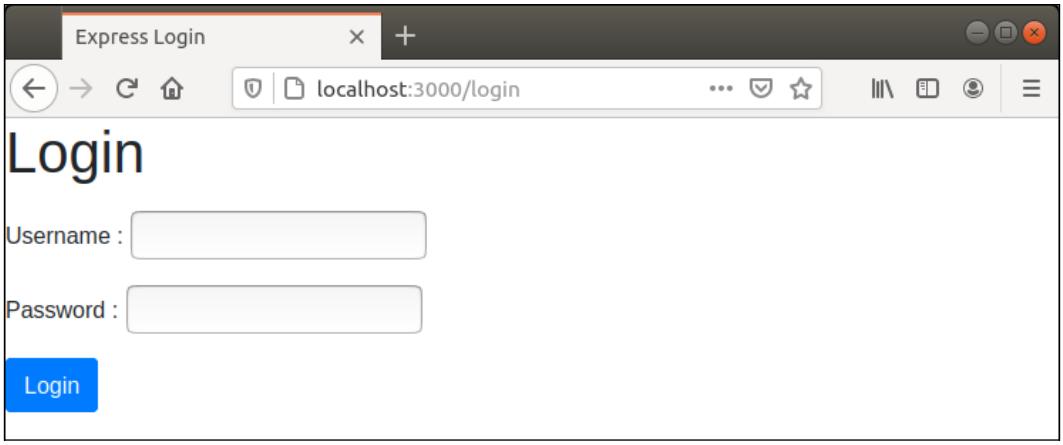


Figure 14.7: Express login template rendering a form to the screen

Here, we can see that the input controls for username and password and the **Login** button have been rendered to the screen from our template.

Before we update our `login.ts` route handler to process this form, let's install and configure a package named `body-parser`, as follows:

```
npm install body-parser
```

With the `body-parser` package installed, we can now configure our Express application to use it, by adding two lines to our `main.ts` file as follows:

```
... existing code

app.set(`views`, path.join(__dirname, `views`));
app.set(`view engine`, `hbs`);

import bodyParser from "body-parser";
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false}));

app.use(`/`, Index.router);
app.use(`/`, Login.router);

... existing code
```

Here, we are again calling the `app.use` function to register the `body-parser` library. The `json` function made available by the `body-parser` library call will initialize middleware code that will convert incoming requests into JSON objects that are attached to the body of an incoming request. This means that we can use the property `body.req` to find parameters passed with the POST request. Note that we also need to call the `urlencoded` function on the `body-parser` library in order to allow JSON-like objects to be exposed on our request `body` property.

With this configuration in place, we can now create a POST handler function in our `login.ts` route handler as follows:

```
router.post(`/login`,
    (
        req: express.Request,
        res: express.Response,
        next: express.NextFunction
    ) => {
        console.log(`req.body.username : ${req.body.username}`);
    });
```

Here, we have invoked the `router.post` function in order to register a handler function for an HTTP POST request, for the URL `'/login'`. Within our handler function, we are logging a message to the console with the value of the `req.body.username` property. As mentioned earlier, the `body-parser` library will attach any form data from the HTML `<form>` tag to the request `body` property, with a property name matching the `name` attribute for each input element of our form. If we enter some text into the username input control, and hit the **Login** button, our POST handler will now log a message to the console, as follows:

```
listening on port 3000
req.body.username : test
```

Here, we can see that entering the value `test` into the username input control will make the `req.body.username` property available with our form data.

Express session data and redirects

Now that we have access to the username and password data from the login form, we can set an application variable that can tell our application whether or not the user has been logged in. To do this, we will store the username in a session variable, so that it is persisted between application screens. We will make use of the `express-session` library to handle session storage, which can be installed via `npm` as follows:

```
npm install express-session
npm install @types/express-session
```

Here, we have installed both the `express-session` library and its associated declaration files. We can now configure our application to use this library, by updating our `main.ts` file with the following changes:

```
... existing code

import expressSession from 'express-session';

app.use(expressSession(
    {
        secret: `asdfghjkl`,
        resave: false,
        saveUninitialized: true
    }
));

app.use(`/`, Index.router);
app.use(`/`, Login.router);

... existing code
```

Here, we are importing the `express-session` library, and then calling the `app.use` function with the `expressSession` function in order to configure it. The `expressSession` function uses a configuration object to set the `secret`, `resave`, and `saveUninitialized` required properties.

The `secret` property, as described by the `express-session` documentation, is used to sign the session ID cookie. This means that the session cookie values will become invalid if they are modified somehow, and provides an extra level of security to ensure that our session values are legitimate. The `resave` option controls whether the session will be saved to the session store for each request. We are not using a production-level session store in this sample, so we will leave this as the recommended value of `false`. The `saveUninitialized` option controls whether new sessions are automatically saved to the store, which we will leave as the recommended value of `true`.

Now that we have the `express-session` library configured, we can store a value in the request's session by modifying the `login.ts` route handler as follows:

```
... existing code

router.post(`/login`,
(
    req: express.Request,
    res: express.Response,
    next: express.NextFunction
) => {
    console.log(`req.body.username : ${req.body.username}`);
    if (req.body.username?.length > 0) {
        console.log(`found body.name`);
        (<ISessionData>req.session).username =
            req.body.username;
        res.redirect(`/`);
    } else {
        res.render(`login`, {
            title: `Express Login`,
            errorMessage: `Please enter a username and password`
        })
    }

});

... existing code
```

Here, we have updated our POST request handler with an `if` statement that checks for the existence of the `req.body.username` property. If the `length` of this property is greater than `0`, we log a message to the console, and then store the value in the `req.session` object, with the property name of `username`. Note how we are casting the `req.session` property to an interface named `ISessionData`. The definition of this interface is in the `SessionData.ts` file, as follows:

```
import session from "express-session";

export interface ISessionData
    extends session.Session {
    username: string;
}
```

Here, we have defined an interface that derives from, or extends, the `session.Session` class, and adds a property named `username` of type string. By creating an interface like this, we can ensure that any use of the `req.session` object will only reference properties that have been defined within our code.

Once we have stored the value of the `req.body.username` property into the `req.session.username` property, we call the `res.redirect` function to redirect the browser to the URL at `"/"`. This will cause the `index.ts` route handler to be invoked, and re-render the `index.hbs` template.

Note that if the length of the `req.body.username` property is less than or equal to `0`, which means that nothing has been entered in the username input control on our form, we re-render the `login.hbs` template with an object that contains a property named `errorMessage`.

Now that we have stored the username within the session object, and handled the error condition, let's update our `index.hbs` template as follows:

```
<h1>{{welcomeMsg}}</h1>

{{#if username}}
<p>User : {{username}} logged in.
    {{else}}
<p>Click <a href="/login">here to login</a></p>
{{/if}}
```

Here, we are using a Handlebars conditional statement with the syntax of `{{#if username}}`. This will check if the `username` property exists on the object that was used to render this view. If this property exists, we will render a paragraph with the name of the user that is logged in. If it is not, we render a link to the `"/login"` URL.

All that remains for us to do is to read the session information when we render the `index.hbs` view, by updating the `index.ts` file as follows:

```
router.get(`/`,
(
    req: express.Request,
    res: express.Response
) => {
    res.render('index',
        {
            title: 'Express App',
```

```
            welcomeMsg: 'Welcome to the Express App',
            username: (<ISessionData>req.session).username
        }
    )
});
```

Here, we have added a single property to the object that is being used to render the `index.hbs` template. This property is named `username`, and is being set to the value of the `username` property from the session object. Note that we are re-using the `ISessionData` interface when referencing the `req.session` object, to ensure that we get the property names correct.

The full flow of the Express application can now be seen in the following series of screenshots:

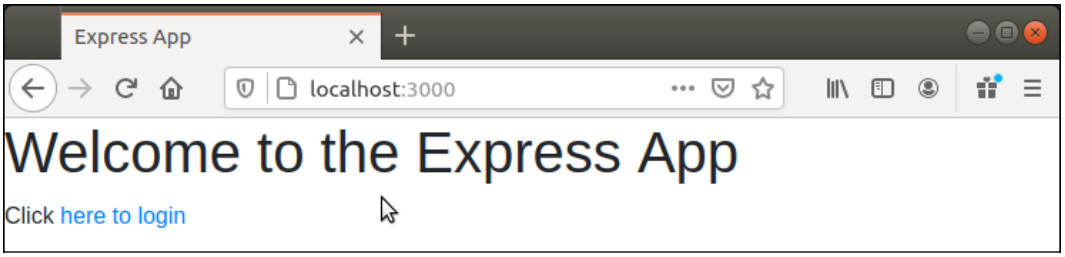


Figure 14.8: Express application showing the default page

Here, we have the default page that is being served by the `index.ts` router. If we click on the link to log in, we are re-directed to the `/login` page, as follows:

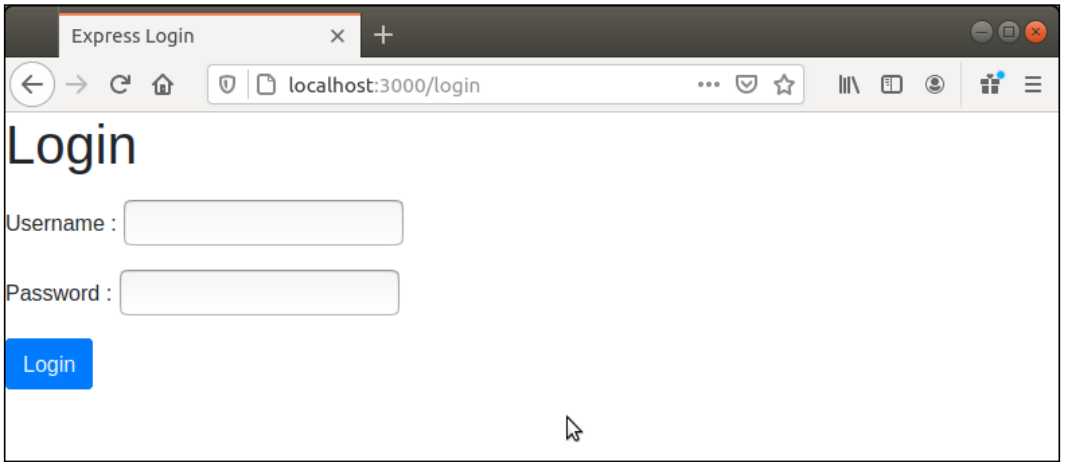


Figure 14.9: Express application showing the login page

Here, we have the login page that is being served by the `login.ts` router. If we now click on the **Login** button, but do not enter a username, the login page is re-rendered with an error message as follows:

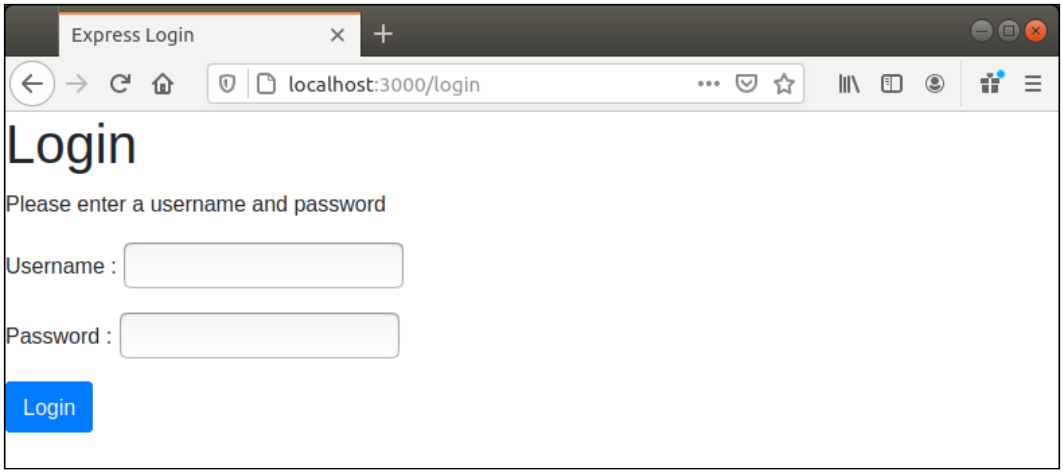


Figure 14.10: Express application showing an error message on the login page

Here, we can see the error message being shown by the `login.ts` route handler. If we now enter a username and a password and click on the **Login** button, we will be re-directed to the default page as follows:

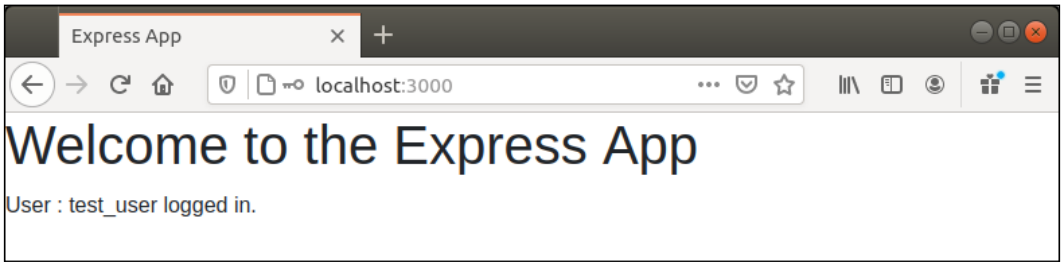


Figure 14.11: Express application showing the username that has been retrieved from the session

Here, we can see that the username of `test_user` is being displayed on the default page. This username has been retrieved from the session information that was stored for this user.

Our Node and Express application is now complete.

Summary

In this chapter, we explored writing a Node application, and particularly a web application, using the Express library. We worked through setting up a very minimal Express application, and then went on to discuss Express routes and configuration. We then explored Express template engines, and installed and configured Handlebars to render two different pages, through our route handlers. We then discussed static files and implemented a login page, showing how to write data to session information. In the next chapter, we will explore the world of REST APIs, and build an API using Amazon Web Services.

15
An AWS Serverless API

Node has been rather a game changer in the web application world. One of the reasons for this is the lightweight hardware specifications that are needed to run a Node web server. Traditionally, web server engines, such as Apache, or Microsoft's IIS web server, needed some pretty beefy servers in order to accommodate thousands of HTTP requests per second.

Node, as we have discussed, uses a single-threaded architecture, and each instruction that needs to wait, for any reason, is put onto a queue for processing at a later time. This means that the server is only running a single thread of execution at any particular time, and therefore can handle a large number of simultaneous requests with a surprisingly little amount of CPU or RAM. In the modern age of cloud computing, this means that many more Node web servers can be run on a single piece of physical hardware, compared to other traditional web servers.

Most cloud services, including Azure, Google, and Amazon, have taken this concept a step further, and now offer the ability to run code without the need for a server at all. Our code is provided with a runtime environment that has all of the dependencies we may need. If we take Node as an example, as long as we have the basic Node modules available at runtime, setting up some code to handle a single web request requires just a few lines of code.

Running code to handle requests within these environments is known as serverless architecture. We do not have to worry about maintaining a server to handle a particular request; the cloud service provider will handle all of this for us.

In this chapter, we will explore the use of a serverless architecture, and build a REST API using Amazon's serverless infrastructure. Amazon Web Services, or AWS, provides what are known as Lambda functions, which are serverless runtime environments to execute code. This code can be written in practically any language, with the most popular being .NET, Go, Java, Python, and JavaScript. There are equivalent concepts within the other cloud service providers, with Microsoft Azure using what are known as Azure functions, which run code in a serverless environment. We will, in particular, be using the AWS Serverless Application Model command-line interface, or the AWS SAM CLI, to accomplish this.

This chapter is split into two main sections. In the first section, we will cover serverless architecture, and get an AWS SAM environment set up. Specifically, we will cover the following topics:

- AWS Lambda architecture
- Installing and initializing a SAM application
- Deploying a SAM application

In the second half of the chapter, we will build an API, using DynamoDB as our database, and will cover the following topics:

- DynamoDB tables and NoSQL Workbench
- Lambda functions
- Running Lambdas locally
- Lambda path parameters
- Processing database records

Let's begin by setting up a serverless environment.

Serverless setup

In this section of the chapter, we will set up a serverless development environment using the Amazon Serverless Application Model command-line interface. We will use this command-line interface to initialize an API application, which will also generate some sample code for us. We will then deploy this application to the cloud, again using the command-line interface.

AWS Lambda architecture

Before we go ahead and build a Lambda function, let's explore the pieces of the AWS architecture that we will use in order to respond to a web request. These elements are shown in the following diagram:

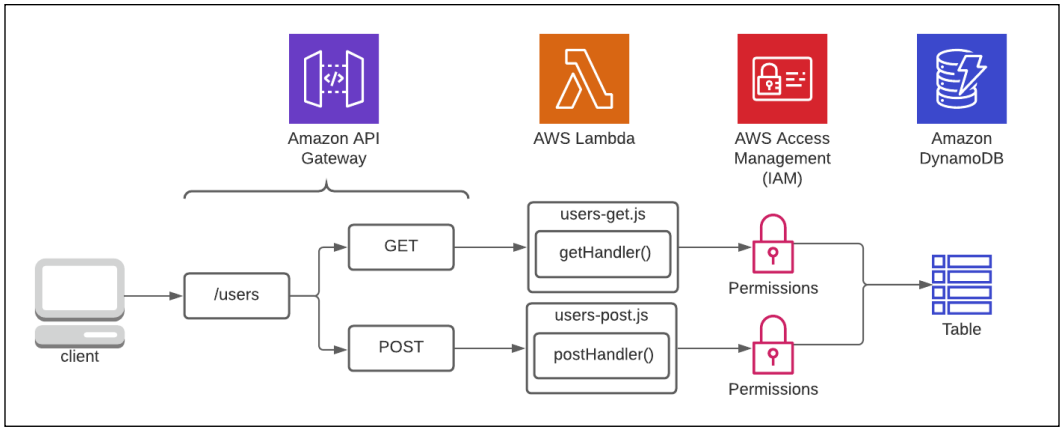


Figure 15.1: AWS Lambda architectural components

Here, we start with a client request, on the far left-hand side of the diagram. This request takes the form of an HTTP request for a GET call on the `/users` API endpoint. The first architectural element that will respond to this request is Amazon API Gateway. Amazon API Gateway is an AWS service that is used to provide REST services, HTTP services, and even WebSocket services to clients.

Amazon API Gateway is essentially a repository of configured endpoints. It also provides all of the infrastructure that is required to handle the processing of multiple incoming API calls. In the preceding diagram, we can see that we have configured a GET handler for a REST request to the endpoint `/users`, and we have also configured a POST handler for the same `/users` endpoint. Each of these endpoints, and the handlers that will respond to these requests, will need to be registered with Amazon API Gateway.

We can also configure an API Gateway endpoint to execute an AWS Lambda function. In our preceding diagram, we have configured the GET request on the `/users` endpoint to invoke a function within a file named `users-get.js`, with the function itself named `getHandler`. We have also configured the POST request on the `/users` endpoint to invoke a function named `postHandler` within a file named `users-post.js`. Both of these functions are our Lambda functions. Note that the JavaScript files are shown in this diagram, as Lambda functions only support JavaScript. We will be generating these JavaScript files from TypeScript.

Each of our Lambda functions will need access to an Amazon DynamoDB table. In order to allow these functions to access our database tables, we will also need to use the AWS **Identity and Access Management service** (**IAM**) to provide the appropriate access. This will ensure that only the configured resources will have access to our data store.

Finally, we will need to define and configure some tables using Amazon DynamoDB, which will be used to store and retrieve information that we need.

Amazon provides a web-based interface known as the management console, which allows us to configure each of these services and set up the links between them. In practice, however, using this web-based interface can become overwhelming, tedious, and difficult to manage. Configuring AWS services through the management console is also a manual process, and as such can easily lead to mistakes and misconfiguration, particularly if we need to configure a large amount of services. Luckily, Amazon provides a configuration API that can be used to automate all of these processes, as well as a command-line interface that can be used to interact with this API.

The command-line interface for configuring Amazon Web Services is known as the AWS CLI, and the command-line interface for configuring serverless applications is known as the AWS Serverless Application Model CLI, or AWS SAM CLI. The SAM CLI uses template files written in either YAML or JSON, which configure each of our architectural components, and can be used to great effect to configure serverless applications. We will install and use this AWS SAM CLI to configure and even deploy our serverless Lambda functions, and to configure and deploy our DynamoDB tables.

Installing the SAM CLI

The installation process for installing the SAM CLI can be a little complex, depending on the operating system that you are working with. We will not cover the entire installation process here, but rather point to the official installation instructions found on the AWS website (`https://aws.amazon.com/serverless/sam/`).

In a nutshell, the components that need to be installed are as follows:

- The AWS CLI: This is the command-line interface to AWS, some functions of which are used by the AWS SAM CLI
- Docker: The virtual container that will simulate the Lambda environment
- Homebrew (for macOS systems): This is the package manager for macOS systems that the AWS SAM CLI uses for installation
- The AWS SAM CLI: This is the specific AWS command-line option for working with serverless applications

We will also need an active AWS account in order to create resources within AWS. The good news is that we can create a free account, and use a multitude of AWS services for free using the Free Tier of AWS services. AWS generally uses a threshold model for the services available on its free tiers.

In other words, as long as the number of requests, or CPU usage, or disk space, or whatever metric that service is measured by, is below a certain threshold, it remains free of charge. Once these thresholds are breached, we will then start to attract fees. This model is great for experimentation, low-usage applications, and can even provide a large discount for corporate applications using this model. The thresholds are generous, meaning that we can go a very long way in terms of usage before fees start to kick in.

For the purposes of this chapter, we will assume that an AWS account has been created, and that the AWS CLI and the AWS SAM CLI have been installed correctly. To ensure that the AWS SAM CLI is working, we can issue the following command on the command line:

```
sam --version
```

Here, we are using the `sam` command to query the version of the SAM CLI that is currently installed. At the time of writing, this version is as follows:

```
SAM CLI, version 1.20.0
```

Here, we can see that the SAM CLI is correctly installed, and is currently at version 1.20.0.

Before we can use the SAM CLI from the command line, we will also need to ensure that our credentials have been configured correctly for the AWS CLI. This can be accomplished by running the following command:

```
aws configure
```

Here, we are invoking the AWS CLI with the command `configure`, which is used to set up the credentials that we will use from the command line to connect to AWS. The `configure` command will ask four questions, as follows:

```
~/.aws$ aws configure
AWS Access Key ID [None]: [ paste enter access key ]
AWS Secret Access Key [None]: [ paste secret access key ]
Default region name [us-east-1]:
Default output format [None]:
```

Here, the first question that the AWS CLI asks is for us to enter our AWS Access Key ID. This information will be provided when we first sign up to Amazon, or it can be regenerated through the IAM settings. The second question is the AWS Secret Access Key, which is also provided on first sign-up, or through IAM. The next question is which region should be used as the default, and the last question is what the default output format should be. We have left the defaults for region name and output format as suggested.

Initializing a SAM application

With the SAM CLI installed, we can create a SAM application as follows:

```
sam init --name api-app
```

Here, we are invoking the SAM CLI with the command `init`, which is used to initialize a SAM application, and we are also using the `--name` option to name the application `"api-app"`. This will start the initialization routine, which will ask a number of questions on the command line as follows:

```
Which template source would you like to use?
     1 - AWS Quick Start Templates
     2 - Custom Template Location
Choice:
```

Here, we will choose the first option, which is to use AWS Quick Start Templates, and respond by typing `1`. The next question is which package type to use, as follows:

```
What package type would you like to use?
     1 - Zip (artifact is a zip uploaded to S3)
     2 - Image (artifact is an image uploaded to an ECR image
repository)
Package type:
```

Here, we can again choose option `1` to use Zip artifacts uploaded via an S3 bucket. The next question we are presented with is to choose the runtime for our Lambda functions, as follows:

```
Which runtime would you like to use?
     1 - nodejs14.x
     2 - python3.8
... other options
Runtime:
```

Here, we will choose option `1`, which is to use the `nodejs14.x` runtime. This options list actually has 14 different options, which are not shown here, meaning that the list of available languages with which to write AWS Lambdas is quite extensive.

The final question we will need to answer is which quick start template to use for our application, as follows:

```
AWS quick start application templates:
     1 - Hello World Example
```

```
     2 - Step Functions Sample App (Stock Trader)
... other options
Template selection:
```

Here, we will again choose option `1`, which will give us a classic `"Hello World"` example. The SAM CLI will now create an application for us, with the following output:

```
    -----------------------
    Generating application:
    -----------------------
    Name: api-app
    Runtime: nodejs14.x
    Dependency Manager: npm
    Application Template: hello-world
    Output Directory: .

    Next steps can be found in the README file at ./api-app/README.md
```

Here, we can see that we have generated a SAM application named `api-app` that is using the `nodejs14.x` runtime, uses the `npm` tool as a dependency manager, and is using the `hello-world` application template.

Generated structure

Let's now take a quick look at the generated application structure that the SAM CLI has generated for us, based on the `"Hello World"` example, which we selected when initializing the application. The files and folders that are created for us are as follows:

```
.
└── api-app
    ├── template.yaml
    ├── README.md
    ├── hello-world
    │   ├── tests
    │   │   ├── unit
    │   │   │   └── test-handler.js
    │   ├── package-lock.json
    │   ├── package.json
    │   └── app.js
    └── events
        └── event.json
```

Here, we can see that the SAM CLI has created a directory named `api-app`, which corresponds to the name that we used when initializing the application. Within this directory, we have a file named `template.yaml` and a `README.md` file. The `template.yaml` file is used to describe all of the AWS elements that we will need within our application. This includes the endpoint names that Amazon API Gateway will use, the names of the JavaScript files that will be used as Lambda handler functions, and also the IAM access control settings. We can also include elements that describe DynamoDB tables, and assign different database access rights to each of our Lambda functions.

The `template.yaml` file is used when we deploy a SAM application to AWS, and is also used to configure endpoints and Lambda handlers when we run a local copy of our API.

If we take a sneak peek at a section of the `template.yaml` file, we will find that it references files within the generated directory structure, as follows:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: hello-world/
      Handler: app.lambdaHandler
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

Here, we have a YAML file that describes a resource named `HelloWorldFunction` of type `AWS::Serverless::Function`. This resource has a set of properties, including one named `CodeUri`. This `CodeUri` property specifies the directory within which to find the source for this serverless function, and is set to `hello-world/`. The `Handler` property tells us to look in a file named `app.js` for an exported function named `lambdaHandler`, which will be run when this Lambda is invoked. Putting these values together, we will therefore invoke an exported function named `lambdaHandler` within a file named `hello-world/app.js`.

The `Runtime` property specifies the environment in which to run this Lambda function, which is set to `nodejs14.x`. We then have an `Events` section, which lists `HelloWorld` as an event, of `Type: Api`. This section is actually describing the API Gateway configuration, which has a property with a `Path` set to `/hello`, and the `Method` set to `get`. This means that we have configured a GET operation at the endpoint `/hello`, which will call our Lambda function.

Deploying a SAM application

Without making any changes to our sample application, we can immediately deploy it to AWS by invoking the `deploy` command of the SAM CLI. The first time we run this command, however, we will need to answer a few questions concerning our configuration, which will be stored for later use. To deploy a SAM application for the first time, we can invoke the SAM CLI as follows:

```
sam deploy --guided
```

Here, we have invoked the `deploy` command with the parameter `--guided`. This option will provide us with an opportunity to set some variables that are required for deployment to AWS. A summary of these questions and answers is as follows:

```
Setting default arguments for 'sam deploy'
=========================================
Stack Name [sam-app]: api-app
AWS Region [us-east-1]:
#Shows you resources changes to be deployed and require a 'Y' to
initiate deploy
Confirm changes before deploy [Y/n]: Y
#SAM needs permission to be able to create roles to connect to the
resources in your template
Allow SAM CLI IAM role creation [Y/n]: Y
HelloWorldFunction may not have authorization defined, Is this okay?
[y/N]: Y
Save arguments to configuration file [Y/n]: Y
SAM configuration file [samconfig.toml]:
SAM configuration environment [default]:
```

Here, we have specified a number of options regarding deployment, including the stack name, and the AWS region. Note that these options are written to a file named `samconfig.toml`, so that we do not have to answer them the next time we run a `sam deploy` command.

Once this command has completed, the output will include the full API Gateway URL that has been created, as follows:

```
    https://d6vnhxnc83.execute-api.us-east-1.amazonaws.com/Prod/hello/
```

Here, we can see the full URL that will invoke our Lambda function. Copying and pasting this value into the browser will invoke our Lambda as follows:

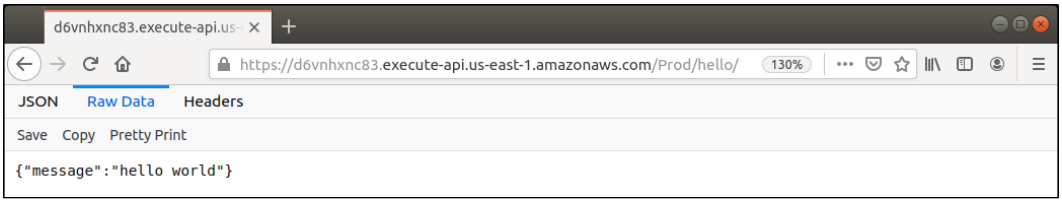


Figure 15.2: Browser showing results of a GET call to our Hello World Lambda function

Here, we can see that our Lambda function is responding to a GET request on the endpoint `/hello`.

In a few steps, we have initialized an AWS serverless application using the SAM CLI, deployed it to an AWS account, and can test the deployed version of our Lambda function almost immediately.

Building an API

Now that we have a working AWS SAM CLI environment configured, we can build an API. The API that we will build in this chapter will be used to support the three existing applications that we have been building throughout this book, in Angular, React, and Vue. Our Angular application, which we built in *Chapter 11*, *Angular*, revolved around a user logging in to an application. This means that we will need to store and retrieve user details from a database.

The React application that we built in *Chapter 12*, *React*, showed a list of products for sale, and when a particular product was chosen, it would show the details of the product. To support this, we will need to store and then retrieve our products from a database, and allow for the generation of a list of available products. We also included the option of selecting how many of these products to add to a shopping cart.

The Vue application that we built in *Chapter 13*, *Vue*, was responsible for displaying a user's current shopping cart, and allowed for updating, as well as deleting the products in the cart. In order to support this functionality in our API, we will need a mechanism to store and retrieve a user's shopping cart information, including the products they have ordered, and the amount of each product they have selected.

DynamoDB tables

To support the functionality that our applications require, we will need three DynamoDB database tables, as shown in the following diagram:

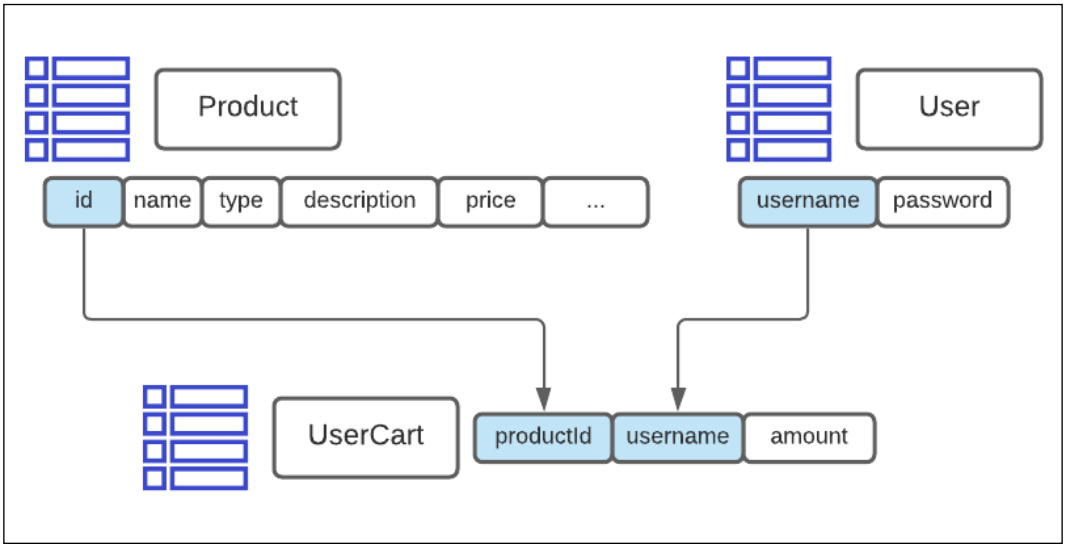


Figure 15.3: DynamoDB tables that will be used within the API

Here, we have a `Product` table, a `User` table, and a `UserCart` table. The `Product` table has a primary key named `id`, and fields to hold the product information, such as `name`, `type`, `description`, `price`, and others. The `User` table has a primary key, which will be the `username`, and a field to hold the `password`. The `UserCart` table has a primary key that is made up of both the `productId` from the `Product` table and the `username` from the `User` table, as well as one additional field to hold the `amount` of the particular product the user has ordered.

Note that DynamoDB does not distinguish between an insert and an update operation on a particular table; it has a single "put" operation. This "put" operation will either create a database record if it does not exist, or update it if it does exist. This create or update operation is based on the primary key. For our `Product` and `User` tables, each product will have a distinct `id`, and each user will have a distinct `username`. When it comes to the `UserCart` table, however, the combination of `productId` and `username` together determines a unique row in the table.

Let's now create these table definitions within our `template.yaml` file, starting with the `Product` table as follows:

```
Resources:
  ProductTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: ProductTable
      AttributeDefinitions:
        - AttributeName: "id"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "id"
          KeyType: "HASH"
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
  HelloWorldFunction:
... existing Lambda attributes
```

Here, we have an entry in our `Resources` section named `ProductTable`, which is of type `AWS::DynamoDB::Table`. This table definition has four properties, which are the `TableName`, the `AttributeDefinitions`, the `KeySchema`, and `ProvisionedThroughput`. The `TableName` property is self-explanatory, and is set to the value `ProductTable`.

Our `AttributeDefinitions` property has a single entry with the `AttributeName` of `"id"`, and the `AttributeType` of `"S"`, which denotes a string. Our `KeySchema` property specifies the attribute that will be used as the key for this table, which is the `"id"` field.

DynamoDB tables can hold pretty much any information within a table, as long as the key conforms to the type that is defined in the table definition. This means that a DynamoDB table is really an object store, rather than a strict table as seen in most relational databases, which have fixed column names. This is why a DynamoDB table is only defined by specifying the type of its key attribute in the `AttributeDefinitions` property, and the key type in the `KeySchema` property.

The only other property we need in order to define a DynamoDB table is the `ProvisionedThroughput` for reading and writing. These units specify how much data we expect to read and write, and are measured in 4K blocks, or 4,096 bytes. We will not discuss these options here, but bear in mind that these parameters will affect reading and writing performance if not tuned correctly.

The `User` table definition is almost the same as the `Product` table definition, as follows:

```
Resources:
  ProductTable:
... existing definition
  UserTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: UserTable
      AttributeDefinitions:
        - AttributeName: "username"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "username"
          KeyType: "HASH"
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
```

Here, we have a similar table definition for the `UserTable` table, with two minor differences. The first difference is that the `TableName` is now `UserTable`, and the second difference is that the name of the primary key field is `username`. Note that the name of the primary key field is used in both the `AttributeDefinitions` property and the `KeySchema` property.

The `UserCart` table definition is as follows:

```
UserCartTable:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: UserCartTable
    AttributeDefinitions:
      - AttributeName: "username"
        AttributeType: "S"
      - AttributeName: "productId"
        AttributeType: "S"
    KeySchema:
      - AttributeName: "username"
        KeyType: "HASH"
      - AttributeName: "productId"
        KeyType: "RANGE"
    ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
```

Here, we have a standard table DynamoDB table definition that uses a table name of `UserCartTable`. This table definition, however, has two fields that make up the primary key, named `username` and `productId`. Note that both of these fields must be defined in both the `AttributeDefinitions` property and the `KeySchema` properties.

With our database table definitions in place, we can now execute a `sam deploy` command, which will create the DynamoDB tables on our AWS account.

NoSQL Workbench

When working with DynamoDB tables, Amazon provides a GUI utility named NoSQL Workbench for DynamoDB, which helps with both interrogating data within a table, and generating code that can read and write data. The installation for this utility is fairly simple, and is a matter of finding the correct download link for your operating system from AWS (`https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/workbench.settingup.html`).

Once installed, we will need to add a remote connection to our DynamoDB instance, by specifying an access key ID, and a secret access key. For the purposes of this discussion, we will assume that a connection has been set up, and the region set correctly, such that when we choose the **Operation builder** option, we can see the three tables that we created earlier, as shown in the following screenshot:

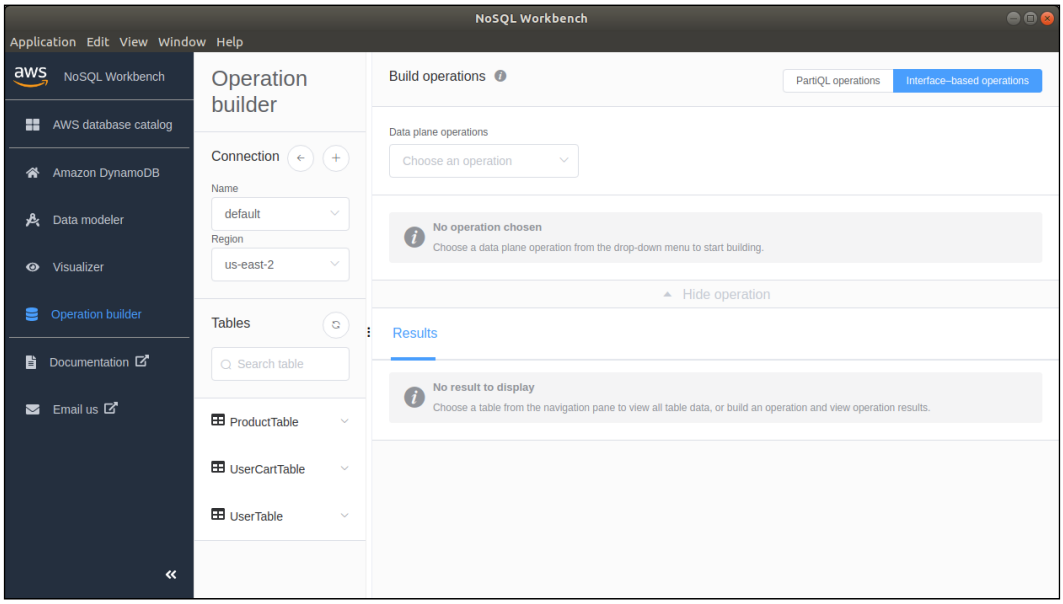


Figure 15.4: NoSQL Workbench in Operation builder mode

Here, we can see that we are in **Operation builder** mode, which is shown on the left-hand side panel, and that our three tables are shown in the **Tables** list.

Let's now choose an operation named `PutItem` in the **Build operations** panel, and choose the `ProductTable` from the list of tables, as follows:

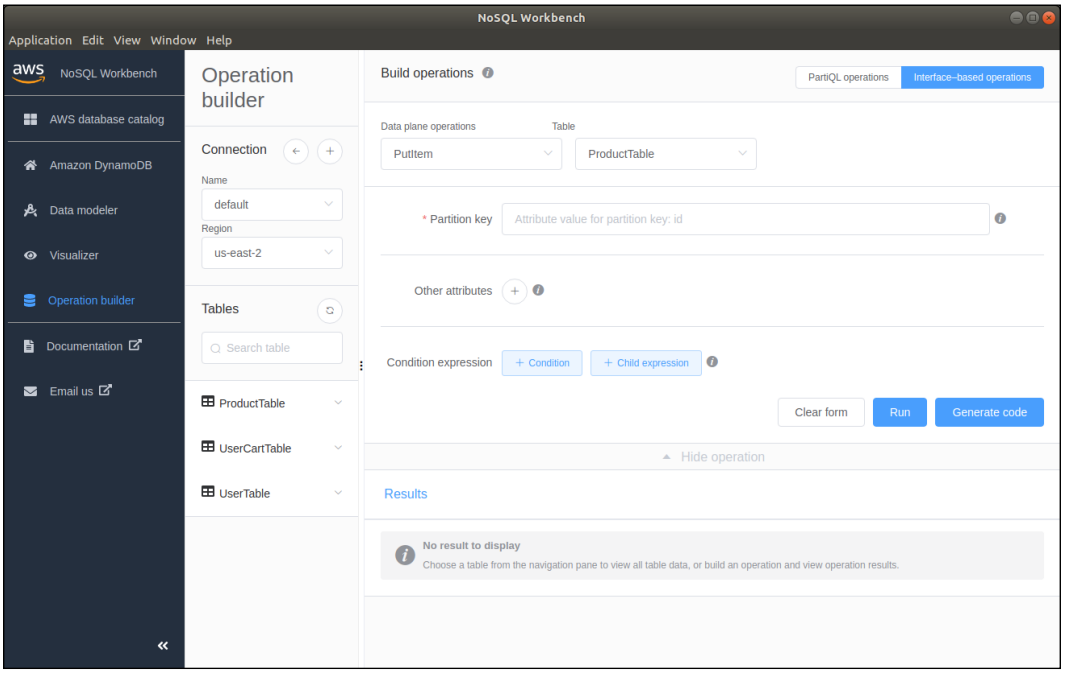


Figure 15.5: NoSQL Workbench with a table selected in Build operations mode

Here, we now have a screen that will allow us to build a PUT operation, or a `putItem` operation, on the `ProductTable` with the attributes that we specify. As the `ProductTable` has a single primary key named `id`, the **Partition key** field is indicating that we should specify what this key is.

We can also add other attributes to this PUT operation by clicking on the **+** button next to **Other attributes**, and by specifying an attribute name, an attribute type, and an attribute value, as shown in the following screenshot:

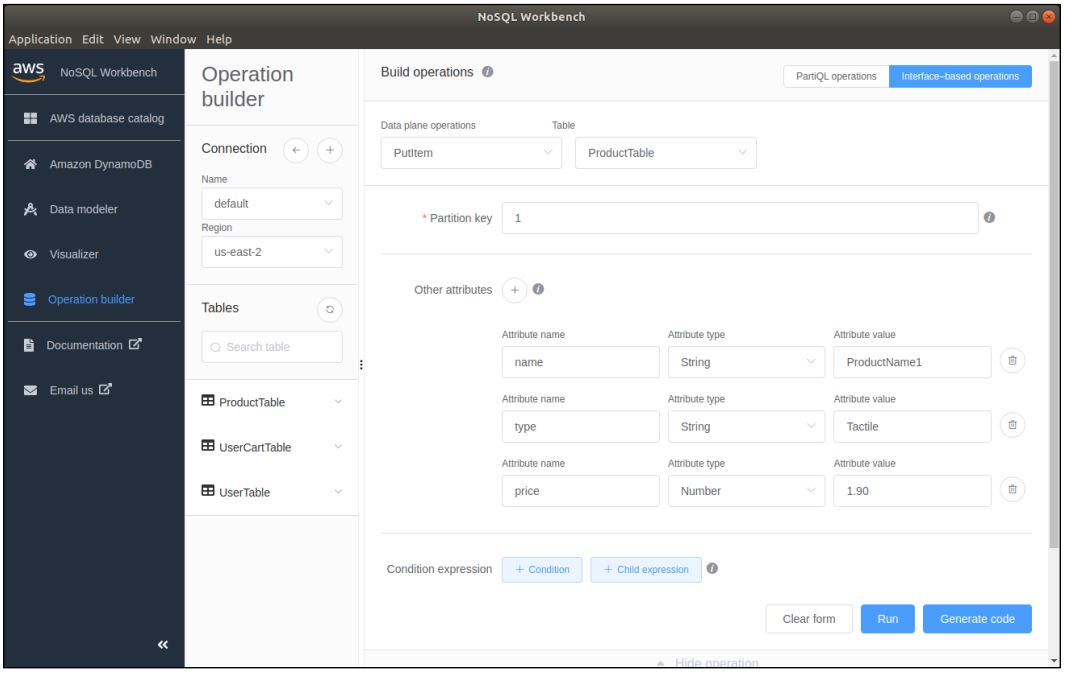


Figure 15.6: NoSQL Workbench PutItem operation with multiple fields as attributes

Here, we can see that we have entered the value of `1` as the partition key and then have added three additional attributes. The first of these attributes is named `name`, is of type `String`, and has the value `ProductName1`. The second attribute is named `type`, is also of type `String`, and has the value `Tactile`. The third attribute is named `price`, is of type `Number`, and has the value `1.90`.

This screen has a **Run** button in the bottom-right corner of the screen, and also has a **Generate code** button. Hitting the **Run** button will execute this operation, and actually insert data into our `ProductTable` database table. Instead of doing this, let's hit the **Generate code** button, which will generate some Python, JavaScript, and Java code samples in three tabs on the bottom of the screen.

If we select the **JavaScript (Node.js)** tab, we will see generated JavaScript code that is compatible with Node, as shown in the following screenshot:

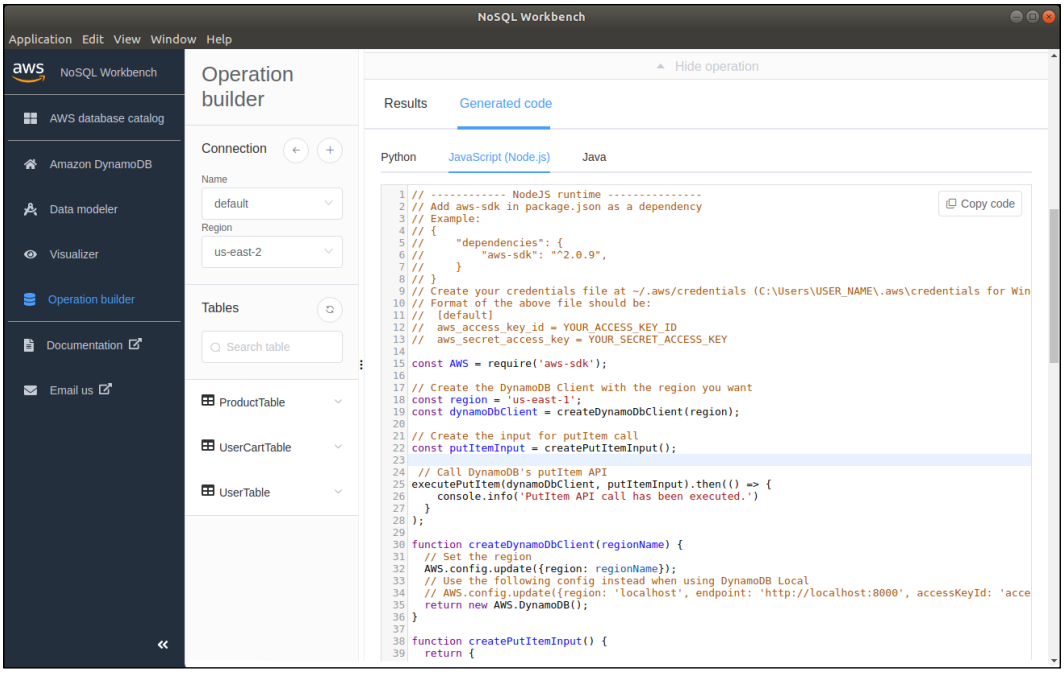


Figure 15.7: NoSQL Workbench showing generated JavaScript code for a PutItem operation

Here, we can see that the NoSQL Workbench tool has generated some JavaScript code that can be run from a Node environment.

The generated code has three main objectives, as follows:

- Create a connection to the database
- Construct an insert or `putItem` instruction
- Execute the instruction

Let's focus initially on the second objective, which is to construct a `putItem` instruction. The generated code for this is as follows:

```
function createPutItemInput() {
  return {
    "TableName": "ProductTable",
    "Item": {
      "id": {
        "S": "1"
      },
```

```
      "name": {
        "S": "ProductName1"
      },
      "type": {
        "S": "Tactile"
      },
      "price": {
        "N": "1.90"
      }
    }
  }
}
```

Here, we have a function named `createPutItemInput`, which is returning an object that has a property named `TableName`, which is the table we are inserting data into, and then an `Item` property, which contains child properties for each of the fields on the table that we are trying to insert. This structure is fairly self-explanatory. We can easily modify this function to insert any data into this table as follows:

```
function createPutItemInput(
    productId: string,
    productName: string,
    productType: string,
    price: number
) {
    return {
      "TableName": "ProductTable",
      "Item": {
        "id": {
          "S": `${productId}`
        },
        "name": {
          "S": `${productName}`
        },
        "type": {
          "S": `${productType}`
        },
        "price": {
          "N": `${price}`
        }
      }
    }
  }
```

Here, we have added four parameters to our `createPutItemInput` function, named `productId`, `productName`, `productType`, and `price`. We have then modified the returned object, and substituted these values as strings directly into the object values.

Once we have an object that can be used to insert values into our table, we can turn our attention to the generated code that executes the insert statement, as follows:

```
const region = 'us-east-2';
const dynamoDbClient = createDynamoDbClient(region);

function createDynamoDbClient(regionName) {
  // Set the region
  AWS.config.update({region: regionName});
  return new AWS.DynamoDB();
}

// Create the input for putItem call
const putItemInput =
    createPutItemInput("1", "Holy Panda", "Tactile", 1.90);

async function executePutItem(dynamoDbClient, putItemInput) {
  // Call DynamoDB's putItem API
  try {
    const putItemOutput =
        await dynamoDbClient.putItem(putItemInput).promise();
    console.info('Successfully put item.');
    // Handle putItemOutput
  } catch (err) {
    handlePutItemError(err);
  }
}
```

Here, we have created a `const` value named `dynamoDbClient`, which is the result of calling the `createDynamoDbClient` function with a `region` argument. The `createDynamoDbClient` function calls the `AWS.config.update` function, and then returns a new instance of the `AWS.DynamoDB` class. Essentially, this is creating a connection to our database.

We then create a `const` value named `putItemInput`, which is the result of calling our `createPutItemInput` function. We then have an `async` function named `executePutItem`, which has two parameters. The first is our `dynamoDbClient` connection object, and the second is our `putItem` object. This function will execute the `putItem` function on the `dynamoDbClient` and wait for the call to complete.

In this manner, we are able to insert data into our `ProductTable` database table. The sample code that accompanies this chapter has a Node program named `api-handlers/helpers/insert_products.ts` that uses these techniques to insert all of the data that we used in *Chapter 12*, *React*, into our `ProductTable` table.

Using the NoSQL Workbench utility, and tweaking the generated JavaScript code very slightly, gives us a quick and convenient way of creating code that can interact with our DynamoDB tables. The NoSQL Workbench utility can also be used to run commands immediately, which we can use to test queries against our database, and make sure that our generated code will work correctly.

Application API endpoints

In order to support the various use cases that our application will need to support, we will need to write a number of API endpoints. These are shown in the following diagram:

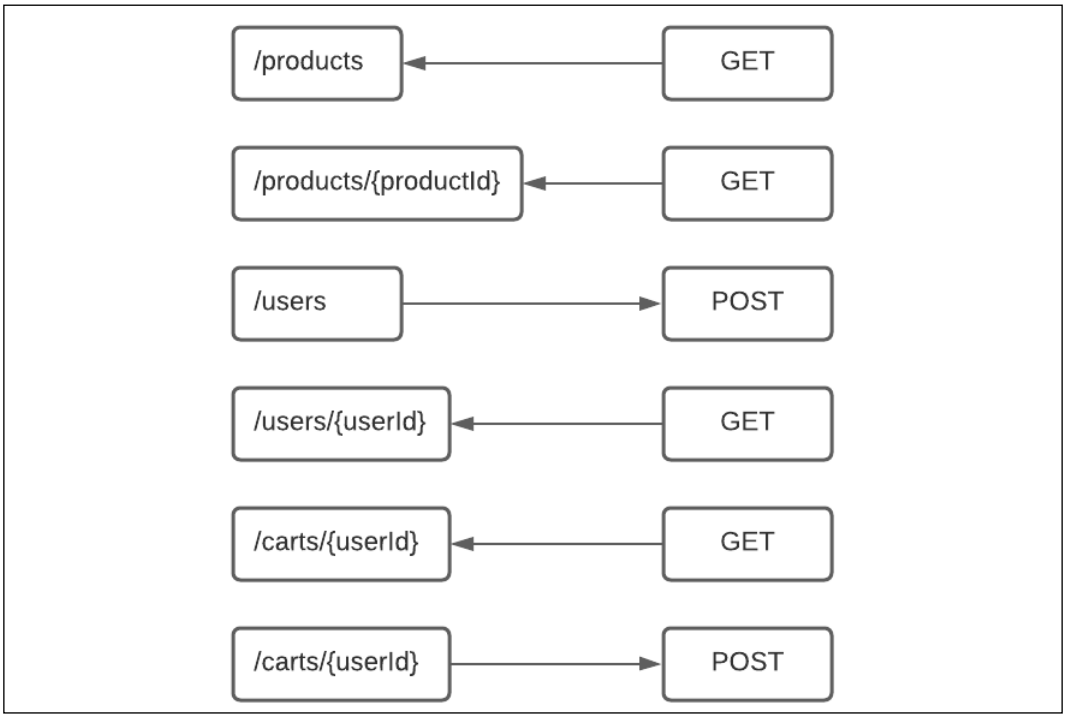


Figure 15.8: List of API endpoints used in the application

Here, we start with a GET endpoint on the `/products` URL, which will fetch all products that are available for sale. We can then get a specific product by calling the `/products/{productId}` endpoint, where the parameter `{productId}` is replaced with the actual ID of the product that we are interested in, such as `/products/1`, or `/products/2`.

Our `/users` URL is only used as a POST endpoint, and is used when a user registers with the application. Once a user is registered, we are able to issue a GET API call to the `/users/{userId}` URL to fetch a particular user's data.

Finally, we have an endpoint named `/carts/{userId}`, which will allow a GET API call to fetch the user's cart contents, and a POST call to update the cart contents.

We can now map the database tables that each API call will use, as shown in the following diagram:

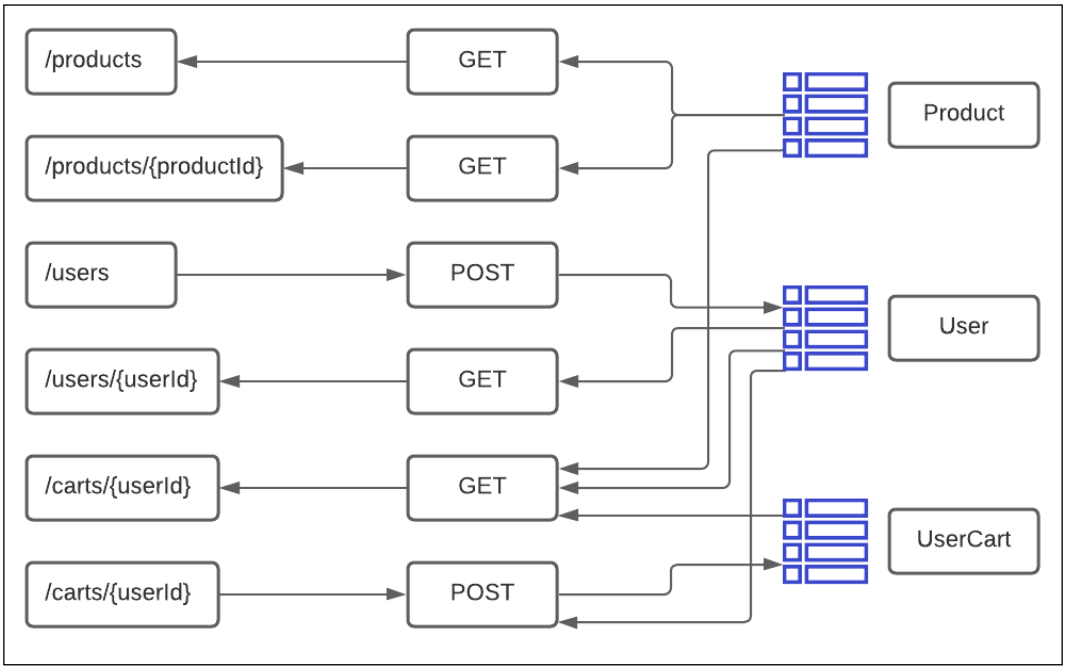


Figure 15.9: List of API endpoints and their read and write interactions with database tables

Here, we can see that the `/products` endpoints will only ever read from the `Product` table. The `/users` POST API endpoint will write to the `User` table, and the `/users/{userId}` GET API endpoint will read from the `User` table.

Our `/carts/{userId}` GET API endpoint, however, will read from all three tables. This endpoint must verify that the `userId` that is passed in as part of the REST path is, indeed, a valid user. If it is, it can then read the `UserCart` table, and match each `productId` found in the table with the corresponding product information in the `Product` table.

The `/carts/{userId}` POST API endpoint will also need to verify that the `userId`, as sent in via the REST endpoint, is a valid user, and will therefore also need to read from the `User` table. If it is a valid user, it can then write to the `UserCart` table.

A Lambda function

Now that we know what we need to build, let's start with the `/users` POST API endpoint. A Lambda function for this will need to be defined within the `template.yaml` file, as follows:

```
  UserApiFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: api-handlers/
      Handler: users.postHandler
      Runtime: nodejs14.x
      Events:
        UserPostEvent:
          Type: Api
          Properties:
            Path: /users
            Method: post
      Policies:
        - DynamoDBCrudPolicy:
            TableName: !Ref UserTable
```

Here, we have defined a section within our YAML file with the name `UserApiFunction`, with a `Type` property of `AWS::Serverless::Function`. We then define a few properties for this serverless function. The `CodeUri` property configures the root directory, and the `Handler` property specifies both the filename and the function that will be used. In this example, we will therefore create a file named `api-handlers/users.js` that exports a function named `postHandler`. This Lambda function will be configured by the API gateway to respond to POST methods at the URL `/users`. Note that we will show how to compile the files within the `api-handlers` directory in the next section.

Note that we also have a `Policies` section that specifies a property named `DynamoDBCrudPolicy`. This policy will grant this Lambda function CRUD rights to the table that is specified, which in this case is a reference to the `UserTable`, which appears earlier in our template definition.

Let's now take a look at the body of the Lambda function itself, as follows:

```
import {
    APIGatewayProxyEvent,
    Context
} from 'aws-lambda';
import { dynamoDbClient } from './db-functions';

export const postHandler = async (
    event: APIGatewayProxyEvent, context: Context
) => {
    let response = {};
    try {

        let bodyJson = JSON.parse(<string>event.body);

        let username: string = bodyJson.username;
        let password: string = bodyJson.password;

        await dynamoDbClient.putItem({
            "TableName": "UserTable",
            "Item": {
                "username": {
                    "S": username
                },
                "password": {
                    "S": password
                }
            },
            "ConditionExpression": "attribute_not_exists(#3f9c0)",
            "ExpressionAttributeNames": {
                "#3f9c0": "username"
            }
        }).promise();
```

```
        response = {
            'statusCode': 200,
            'body': `User created`
        }
    } catch (err) {
        console.log(err);
        // return err;
        response = {
            'statusCode': err.statusCode,
            'body': `${err.message} : an item with this id already
exists`
        }
    }

    return response;
};
```

We start by importing some types from the `aws-lambda` library, and then import a variable named `dynamoDBClient` from a file named `db-functions`. This `db-functions` file, which is available with the sample code, creates the database connection that we will need in all of our Lambdas, and exposes it as the variable `dynamoDBClient`. In this way, we have a single definition for our database connection, which can be re-used between any one of our Lambda functions.

We then `export` an `async` function named `postHandler`, which has two parameters. The first is named `event`, and is of type `APIGatewayProxyEvent`. This `event` parameter contains information that the API Gateway passes into the Lambda function, including the protocol used at the time of the event, and a whole raft of extra information regarding this particular web request. The `context` parameter also contains information that the API Gateway injects, including things such as `logGroupName`, and possibly an identity of type `CognitoIdentity`, along with other information in regards to the context of this call.

Our function begins by creating a variable named `bodyJson`, which is the result of a `JSON.parse` function call using the property `event.body`. This is effectively turning a POST `body` property into a JSON structure, where the content type of the POST request has been set to `application/json`. We then extract the `username` and `password` values from this JSON.

Our Lambda function then calls a `putItem` function on our `dynamoDbClient` connection and passes in a structure with information to store in the `UserTable` table. This information is the `username` and `password`. Note, however, that we have included a property named `ConditionalExpression`, and a property named `ExpressionAttributeNames`.

Note that the `ConditionalExpression` value uses a generated variable named `#3f9c0`, which is the property name in the `ExpressionAttributeNames` object. These generated variables will tie a conditional expression to the value that needs to be used.

These two properties are going to check whether an entry in the table with the same username already exists, and will fail if it does. This is to ensure that we cannot POST the same `username` to the endpoint if it has already been created. Conditional expressions are fairly easy to generate, which can be done using the NoSQL Workbench, as shown in the following screenshot:

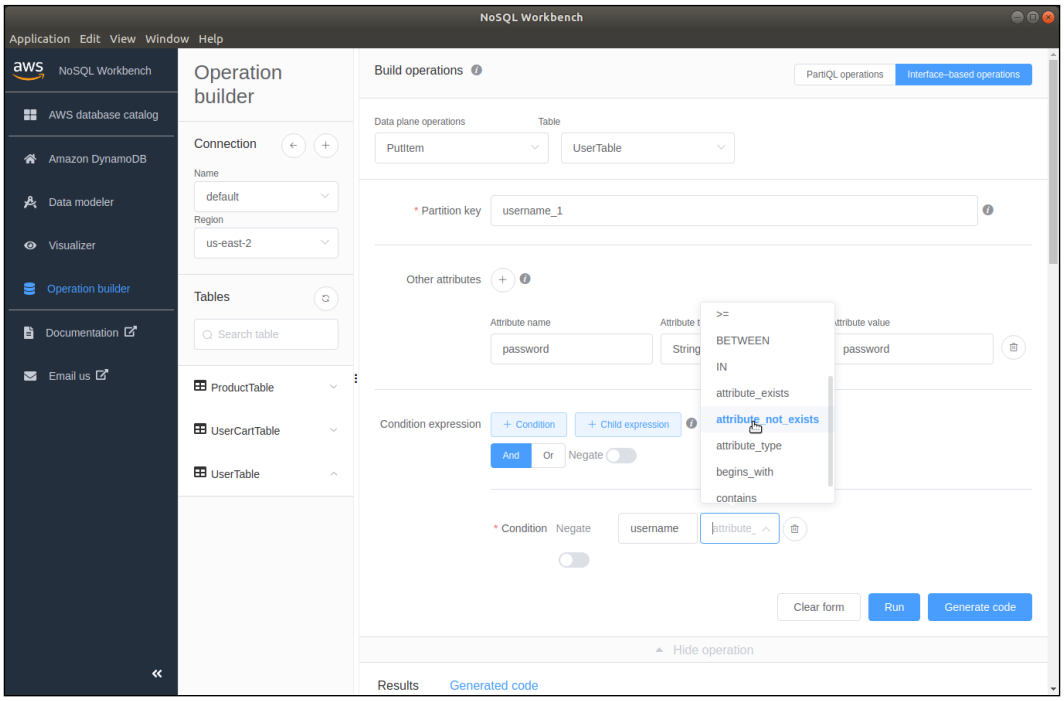


Figure 15.10: NoSQL Workbench showing how to generate a conditional expression

Here, we have used the NoSQL Workbench GUI to generate the code we need for a conditional statement. The **Partition key** value we have entered, `username_1`, is what will be used for the value of the `username` field. We then add another attribute with the **Attribute name** of `password`, of type `String` (partially obscured), and the **Attribute value** of `password`. We have then used the **Condition expression** controls to specify that the `PutItem` operation is only allowed if the `username` attribute does not exist. Using the NoSQL Workbench GUI is a quick way of generating code that can be used in DynamoDB queries.

Our Lambda function will return an object with the `statusCode` set to `200`, and the `body` set to `"User Created"` if the `putItem` operation succeeds. If it violates the `attribute_not_exists` condition, then our `catch` clause will be triggered, and respond with a `statusCode` set to the error status code, and an error message.

Compiling Lambdas

Lambda functions support JavaScript, so we will need to compile our TypeScript code into JavaScript before we can use them. This means that we will need to create a TypeScript environment within each of our sub-directories, such that the `users.ts` file generates a `users.js` file in the same directory. We will therefore need to initiate a TypeScript environment and a Node environment within our `api-handlers` subdirectory as follows:

```
cd api-handlers
npm init
tsc --init
npm install aws-sdk
npm install aws-lambda
npm install @types/aws-lambda --save-dev
```

Here, we have switched to the `api-handlers` directory, and issued both an `npm init` command and a `tsc --init` command. We have also installed the `aws-sdk` and `aws-lambda` npm libraries, as well as the `@types/aws-lambda` package.

Note that the runtime that we will be using within AWS is at Node version 14, which supports much more up-to-date JavaScript versions than browsers do. For this reason, we can modify the `target` property in our `tsconfig.json` file to target `ES2019`, which will generate ES2019 JavaScript for use within our Lambdas. This will simplify the JavaScript that is generated for our Lambda functions.

Running Lambdas locally

The SAM CLI allows us to run our Lambda functions locally, or on our local machine, without the need to deploy them to AWS first. This can be done by typing the following command:

```
sam local start-api
```

Here, we are using the SAM CLI to start up a fully fledged Lambda environment on our local machine. The output of this command will show us what API functions are available, as follows:

```
Mounting HelloWorldFunction at http://127.0.0.1:3000/hello [GET]
Mounting UserApiFunction at http://127.0.0.1:3000/users [POST]
You can now browse to the above endpoints to invoke your functions.
2021-02-13 17:10:29  * Running on http://127.0.0.1:3000/ (Press CTRL+C
to quit)
```

Here, we can see that our two Lambdas are running on port `3000`, with a GET handler at the endpoint `/hello`, and a POST handler at the endpoint `/users`. We can now generate a POST event, using a tool such as Postman.

Postman (`www.postman.com`) is a GUI tool that is specifically geared toward helping develop API endpoints. We can generate a POST event as shown in the following screenshot:

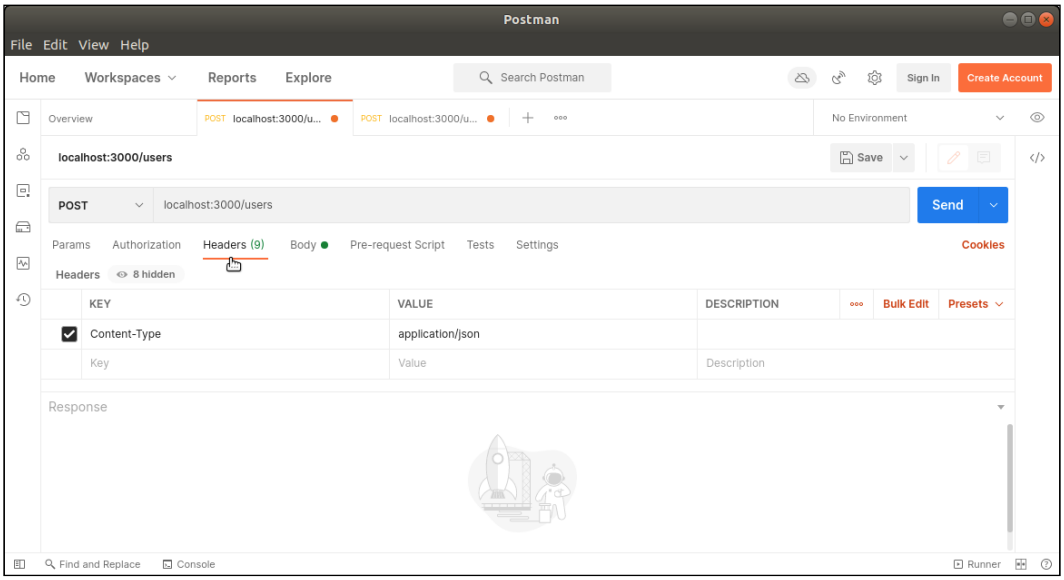


Figure 15.11: Postman GUI showing headers set to application/json

Here, we have selected a **POST** operation, and typed in the URL `localhost:3000/users`. We have also added a row in the `Headers` table to specify the content type to be `application/json`. Before we POST this call, we will also need to switch to the **Body** tab and enter a payload, as shown in the following screenshot:

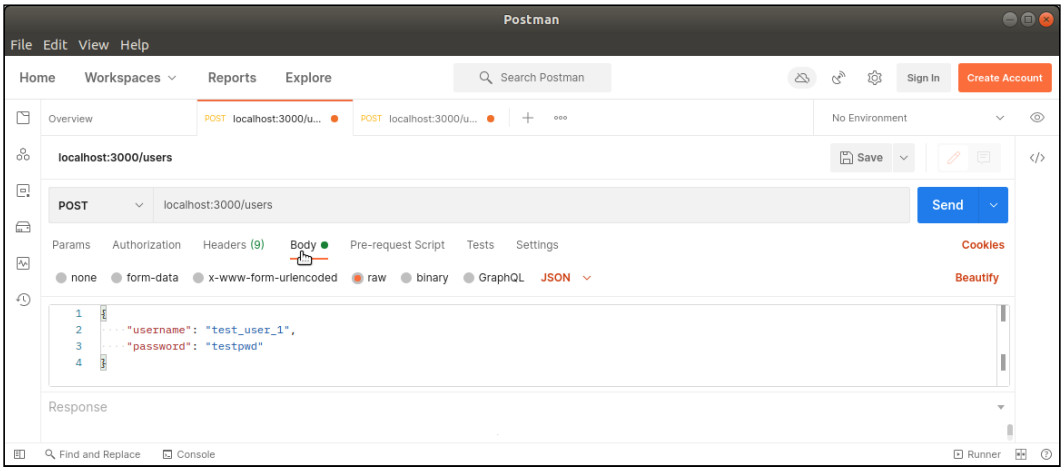


Figure 15.12: Postman GUI showing the contents of the body for a POST operation

Here, we have set a JSON object as the body to use in our POST operation. Hitting the **Send** button now will invoke a POST request to our `/users` endpoint, which is running on our local machine, which should respond with a `200 OK` message, as shown in the following screenshot:

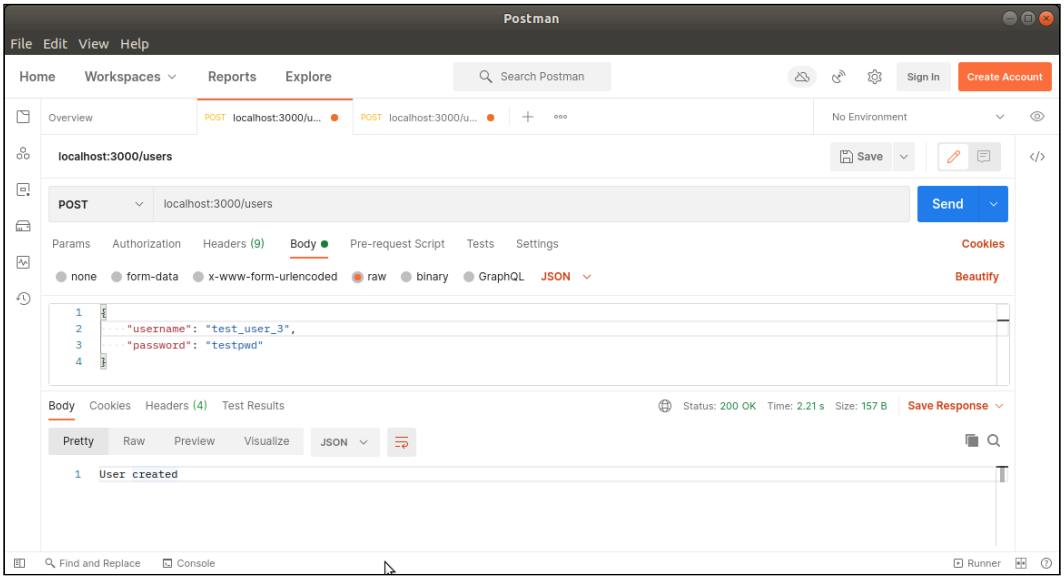


Figure 15.13: Postman GUI showing a successful POST to the user's endpoint

Here, we can see the results of the POST operation, with the `User created` message showing in the response at the bottom of the screen.

Note that if we fire up our NoSQL Workbench and initiate a scan on the `UserTable` table, we will see that this record has been created within our DynamoDB table.

Lambda path parameters

A number of our Lambda functions have a path parameter within them. So, to GET details of a particular user, we will need to use the URL `/users/{userId}`, where we substitute the `{userId}` variable with the actual value. In other words, to GET the details of a user named `test_user_1`, our path would need to be `/users/test_user_1`. This is a standard REST pattern, where the URL itself contains a parameter. Let's now take a look at how the definition of the user's GET Lambda function in our `template.yaml` file supports this syntax as follows:

```
UserGetApiFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: api-handlers/
    Handler: users.getHandler
    Runtime: nodejs14.x
    Events:
      HelloWorld:
        Type: Api
        Properties:
          Path: /users/{userId}
          Method: get
    Policies:
      - DynamoDBCrudPolicy:
          TableName: !Ref UserTable
```

Here, the definition of the Lambda function named `UserGetApiFunction` is almost identical to the previous definition, with two notable exceptions. Firstly, we have specified that the handler function is named `getHandler`, within the `users.js` file, so we will need to create an exported function with this name. Secondly, the `Path` property is using the substitution syntax to indicate that this API function will handle anything that matches a GET request to the `/users/<anything>` path.

Before we write this handler, we can simplify our code slightly by writing a utility function to check whether the user exists in our database, as follows:

```
export async function userExists(
    username: string
): Promise<boolean> {
    const scanOutput = await dynamoDbClient.scan({
        "TableName": "UserTable",
        "ConsistentRead": false,
        "FilterExpression": "#87ea0 = :87ea0",
        "ExpressionAttributeValues": {
            ":87ea0": {
                "S": `${username}`
            }
        },
        "ExpressionAttributeNames": {
            "#87ea0": "username"
        }
    }).promise();

    if (scanOutput.Items && scanOutput.Items?.length > 0) {
        return true;
    } else {
        return false;
    }
}
```

Here, we have an exported `async` function named `userExists` that accepts a single parameter named `username` of type string. Within this function, we create a variable named `scanOutput`, which will hold the value returned by the `scan` function with the expression that we have provided. The `scan` function in DynamoDB is a search function, and this expression searches the `UserTable` table for entries with a `username` field equal to the value of the `username` variable.

Our function then checks whether the `scanOutput` has an `Items` property, and whether the `Items` property has a length greater than `0`. If it does, this means that a record has been found, and we return `true`. If the `scanOutput` does not contain an `Items` collection with at least one value, we return `false`.

We can now use this function in our Lambda function as follows:

```
export const getHandler = async (
    event: APIGatewayProxyEvent, context: Context
```

```
) => {
    let response = {};
    try {
        let userId = (<any>event.pathParameters).userId;

        let isUser = await userExists(userId);

        if (isUser) {
            response = {
                'statusCode': 200,
                'body': `User exists`
            }
        } else {
            response = {
                'statusCode': 404,
                'body': `Not found`
            }
        }
    } catch (err) {
        console.log(err);
        // return err;
        response = {
            'statusCode': err.statusCode,
            'body':
            `${err.message} : an item with this id already exists`
        }
    }

    return response;
};
```

Here, we have exported an `async` function named `getHandler`, which is our Lambda function. As usual, it has two parameters, named `event` and `context`. Note how we use the `event.pathParameters` variable to access the value of the property named `userId`. When our Lambda function is invoked with the path `/users/test_user_1`, the `test_user_1` path parameter will be made available in the variable `event.pathParameters.userId`. Unfortunately, we must cast this `pathParameters` variable to the type of `any` in this case in order to access our `userId` property.

Once we have the `userId` variable from the path parameter, we can invoke the `userExists` function to find out whether the username exists in our database. If it does, we respond with a `200 OK`, `User exists` response, and if we cannot find the corresponding user in our database, we respond with a `404`, `Not found` response.

We can now invoke this Lambda function using Postman, as shown in the following screenshot:

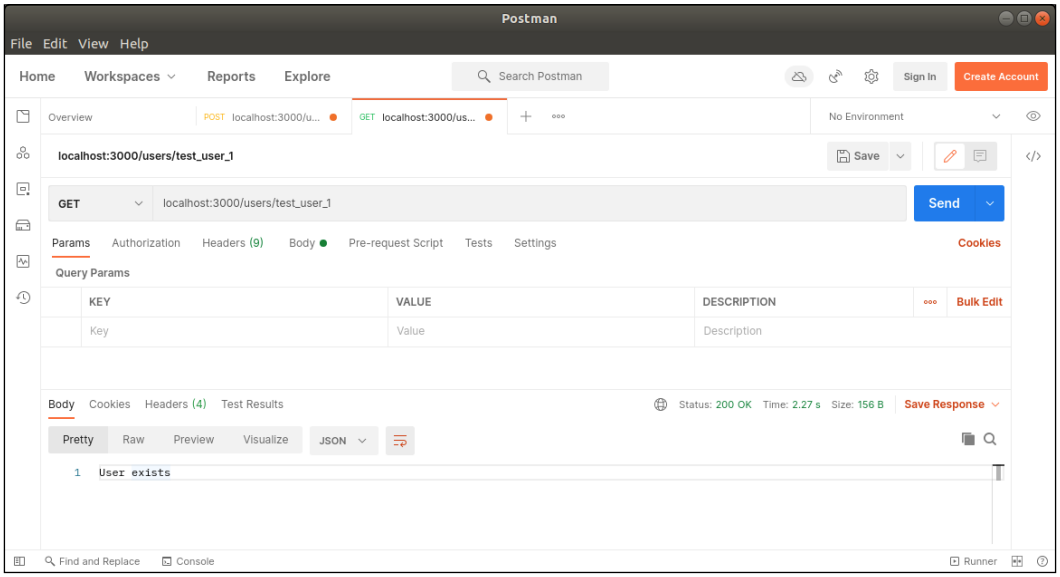


Figure 15.14: Postman UI showing results of a GET request to the /users /test_user_1 endpoint

Here, we can see that a GET request to the endpoint `/users/test_user_1` responds with a `200 OK` response containing the message `User exists`.

Processing database records

Let's now turn our attention to the Lambda function that returns our list of products, which is a GET handler at the endpoint `/products`. This handler will need to query or scan our `ProductTable`, and generate a JSON response, which is an array of each product found in the database. We will not discuss the `template.yaml` entry for this Lambda function as it is nearly identical to the other template entries. Please refer to the source code that accompanies this chapter for a full listing.

Our Lambda function, in the file named `products.ts`, is as follows:

```
export const getHandler = async (
    event: APIGatewayProxyEvent,
    context: Context
) => {
    let response = {};
    try {
```

```
        let scanResults =
            await executeScan(
                dynamoDbClient,
                getProductScanParameters()
            );
        let outputArray = [];

        if (scanResults?.Items) {
            for (let item of scanResults.Items) {
                outputArray.push(getProduct(item));
            }
        }

        response = {
            'statusCode': 200,
            'body': JSON.stringify(outputArray)
        }
    } catch (err) {
        console.log(err);
        return err;
    }

    return response;
};
```

Here, we have the standard structure for a Lambda function. Our function starts by creating a variable named `scanResults`, which will hold the results of querying the entire `ProductTable` table. Again, please refer to the source code for a full listing of the `getProductScanParameters` function. We then create an array named `outputArray`, which will hold our array of products.

Our code then checks whether the query on the database returned any records, and if it did, we loop through each record that was returned and call the `getProduct` function with this database record. The `getProduct` function is as follows:

```
export function getProduct(item: DynamoDB.AttributeMap): IProduct {
    let product: IProduct = {
        id: parseInt(<string>item["id"].S),
        name: <string>item["name"].S,
        type: <string>item["type"].S,
        image: <string>item["image"].S,
        longDescription: <string>item["longDescription"].S,
```

```
            specs: {
                actuationForce: <string>item["actuationForce"].N,
                actuationPoint: <string>item["actuationPoint"].N,
                bottomOut: <string>item["bottomOut"].N,
                bottomOutTravel: <string>item["bottomOutTravel"].N,
                price:
                    item["price"].N ?
                        (parseInt(item["price"].N) / 100)
                            .toFixed(2) : "",
            }
        }
        return product;
    }
```

Here, we are returning an object that conforms to the `IProduct` interface that we used in *Chapter 12*, *React*. Note how we are extracting the value of a particular database field by referencing the column name followed by either an `S` or an `N` for a string or number type, `item["id"].S`, or `item["actuationForce"].N`, as an example. We are also casting each value to a string type, using the syntax `<string>item["id"].S`. This is because the resulting field value of `S` is defined as of type string or undefined, and the field value of `N` is also of type string or undefined.

Note that this code should really be a lot more defensive, and check whether each field exists, before referencing the `S` or `N` value, as it is prone to errors if the value `item["id"]` returns `undefined`, for example. In this case, our code will crash with a `Cannot read property 'S' of undefined` error.

Our Lambda function will now read all of our database entries in the `ProductTable` table and return a collection of objects when called, as shown in the following screenshot:

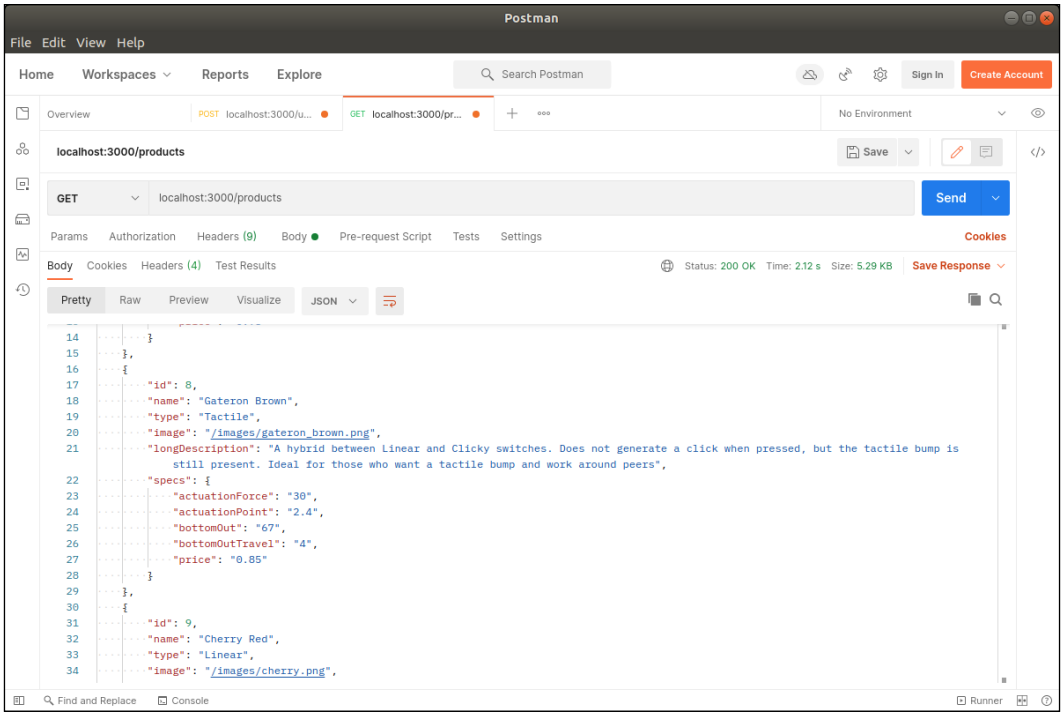


Figure 15.15: Postman UI showing the results of a GET call to the /products endpoint

Here, we can see that a GET call to the `/products` endpoint will return a list of objects, each representing a product that is held within our database.

API summary

We have covered a lot of ground in this section of the chapter and have built endpoints for the POST to `/users`, a GET to `/user/{userId}`, and a GET to `/products`. We will not cover the remainder of the endpoints that we need in detail here, so please refer to the sample code that is attached to this chapter for the full source.

Each of the remaining Lambda functions is defined in the same way, through the `template.yaml` file, and they access the database in the same way that we have already discussed. There are only slight differences between them. Please feel free to use the NoSQL Workbench tool to generate queries to satisfy the rest of the endpoints that we have not covered, and as an exercise, try to put it all together without referring to the attached source code.

Summary

In this chapter, we have covered the architecture of an AWS Lambda function and installed the SAM CLI to help us to generate Lambda functions, deploy Lambda functions, and run them locally. We have also discussed the DynamoDB database, and worked through some examples of using NoSQL Workbench to generate code for us, which we can use to interact with our data. We also created a few TypeScript-based Lambda functions, which were able to use either data sent as part of a POST operation or path parameters to guide their behavior.

In the next, and final, chapter of this book, we will combine the work that we have done over the past few chapters into a single application using micro front-end techniques.

16

Micro Front-ends

In recent years, the programming community has been exploring and implementing micro services architecture. The goal of this architecture is to split up monolithic applications into smaller-sized chunks, such that a group of smaller services now work together to provide application functionality. The benefit of doing things this way centers around the idea that each micro service can be independently deployed, can have a completely independent build and release cycle, and multiple copies of these services can be easily spun up to provide scaling capabilities. Each micro service becomes independent of any others, and can therefore use its own choice of technology stack, deployment pipeline, and testing regime. A micro service can also evolve its functionality over time, without impacting other services, as it is an independent unit that does a particular job within a larger community of services.

An extension of the micro-services architecture is the concept of micro front-ends, where a single application can be made up of several user interfaces that are built independently of each other. In other words, if a micro service is responsible for providing a specific set of functionality, then a micro front-end is a front-end that surfaces this data and functionality to a user. If we think of "silos" of functionality, complete with services, data storage, and user interfaces, then this "silo" is a micro front-end. This micro front-end can be owned by a specific team, and the act of isolating the services, data, and front-end has certain benefits. They can be deployed independently of other front-ends, they can follow their own development cycles and release cycles, and they would have no impact on other "silos" of functionality. Most importantly, though, is that the application itself is not a monolithic application; it is built up of independent component parts, and becomes more modular and easier to change.

Thus far in this book, we have built three different web applications, using Angular, React, and Vue. The Angular application that we built was centered around a user logging in, and handled the login screens and associated logic. The React application that we built showed a list of products, and also showed the details of a product when selected. The Vue application that we put together was centered around a shopping cart, allowing a user to view and update products in their cart, and also generate a summary view of their items.

In this chapter, we will bring each of these applications together into a single application, and treat each existing application as a micro front-end. Specifically, we will cover the following topics:

- Micro front-end design concepts:
    - How do we build a micro front-end?
    - How do front-ends communicate with each other?
    - Domain events
    - An Event Bus
- Building a micro front-end application:
    - A global Event Bus
    - React updates
    - Vue updates
    - An Angular micro front-end
    - Micro front-end summary

Design concepts

So what exactly is a micro front-end application? In theory, it involves breaking down our websites into smaller functional pieces, which are independently deployed and managed. Breaking up a website in this way means thinking about "silos" of functionality that are self-contained and are able to function as if completely independent of other areas of the site.

Designing an application using micro front-ends has its benefits, but it will also present us with a few unique challenges. It is best suited to websites where a number of different teams are working together to produce a single user experience. There may be a team that is solely focused on products and product management. This team would be responsible for building a front-end that shows what products are available on the site.

They may need to implement a search algorithm, for example, or need to show current specials that are available to a certain geographic region. The team that handles all things related to products should be able to build, test, and deploy functionality in their own rhythm, and without affecting other areas of the site.

We may have another team that is solely responsible for handling payment for goods on the site. This team may take all things shopping cart, including integration with credit card payment providers, and calculation of shipping costs and taxes. An update of the site from this team may revolve around the integration of another payment provider, like PayPay, for example, and should not affect anything to do with the product team.

This way of thinking about large applications mirrors what happens in a real-world business. When a business starts out, all aspects from product procurement to sales through to delivery may be handled by a single person. As the business grows, however, separate departments would be created with the focus on a single aspect of the business, such as procurement or sales.

If each "functional team" has their own front-end that they are responsible for, then they are also free to choose whichever framework they wish to use.

Before we jump in and build an application out of our micro front-ends, let's take a step back and discuss what we will need to get this technique right. A conceptual model of our three front-ends is as follows:

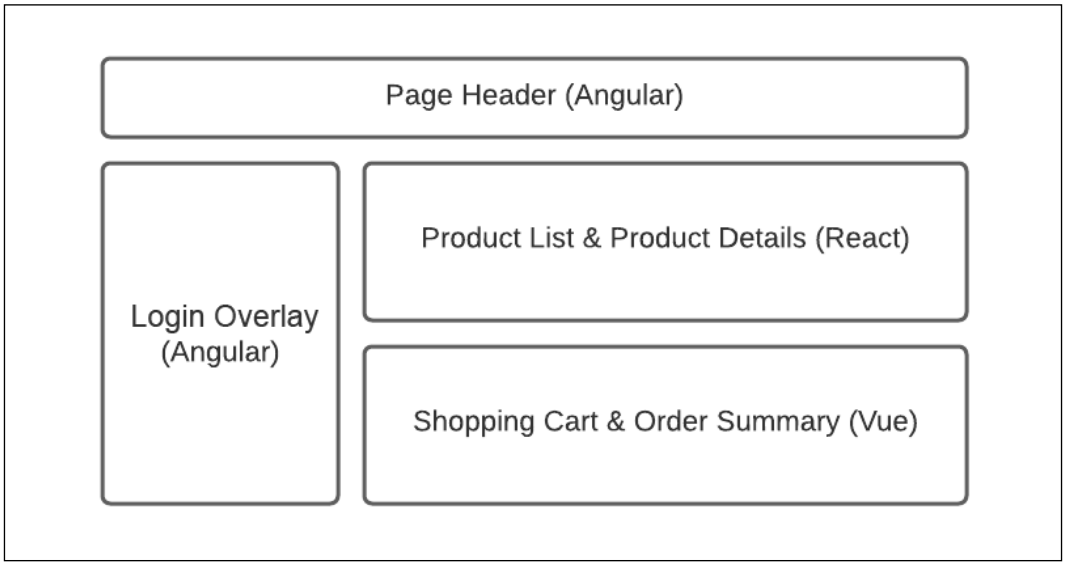


Figure 16.1: Conceptual model of the micro front-ends used in the application

Here, we can see that the **Page Header** is controlled by Angular, as well as the **Login Overlay** on the left. The **Product List** and **Product Details** are built using React, and the **Shopping Cart** and **Order Summary** are built using Vue.

There are two main questions that need to be answered in order to build an application using micro front-ends. These are:

- How do we render a page with multiple sources?
- How will these front-ends communicate, if they need to?

The answers to these questions really depend on which mechanism we choose in order to build a micro front-end application.

Micro front-end mechanisms

There are three main mechanisms that can be used to build micro front-ends. We can make use of iframes, we can use JavaScript, or we can use some sort of registry to put together applications out of components. Let's explore these mechanisms a little further.

The iframe technique

One of the simplest solutions to building a micro front-end is to house each application within its own iframe, and to build an HTML page that includes each of these iframes. iframes have been used to embed content within HTML pages for years, and this technique works well to a certain extent. Building an HTML page is relatively simple, and each micro front-end can be hosted separately.

The disadvantage of using iframes comes when components need to communicate with each other. Embedding content using an iframe is simple when it is a "fire and forget" approach, and the containing page does not need to know anything about the content once the iframe has been created. This is great for things like embedded video or adverts, but does not work as well with micro front-ends. iframes use the same-origin security policy by default, to prevent access to resources that originate from a different URL. While we can override this behavior, it does make cross-iframe communication a little more difficult.

In order to allow two iframes to communicate, we need each iframe to be made aware of the origin of the other, and to explicitly allow messages to be transferred across the iframe boundary. This origin includes both the host name of the server, as well as the port number, which means that an iframe on `myserver.com:8000` can't talk to an iframe hosted on `myserver.com:8001`.

The use of iframes is a viable approach for building micro front-ends, and certainly ensures that each individual front-end is completely isolated from another.

The JavaScript technique

Modern JavaScript frameworks all have a "build" option, which will generate production-ready versions of our applications. These production build steps use bundlers like webpack or Babel, which will bundle all of the necessary JavaScript required to run the application into a few small JavaScript files. These bundlers are also capable of minimizing the resulting files by removing any libraries or functions that are not used, in a technique referred to as "tree shaking." The result of these build steps is generally a single HTML page, and a few of these "bundled" JavaScript files.

Using the output of these bundlers, we can quite easily just create an HTML page that includes the necessary scripts and HTML tags required for each front-end. This may work if the layout of the page is fairly simple, and each of the front-ends is visible to the end user at the same time.

More complex applications, however, need to be able to co-ordinate the layout and events across different micro front-ends. This means that we may only render a particular front-end at a particular time, or we may wish to control where on the page it is displayed. In these situations, we can register each micro front-end with a controlling JavaScript application, which can be used to set up the HTML, and co-ordinate the display of each of the micro front-ends in various regions on the page.

Most micro front-end libraries and frameworks that are currently available use URL paths as the mechanism for determining what is to be shown at a particular time. Using our applications as an example, if the URL path is `/products`, then it shows the product list, and if the URL path is `/shopping-cart`, then it shows the shopping cart.

The JavaScript technique for controlling applications is the most powerful, and the most flexible, option for composing pages using micro front-ends.

The Registry technique

One final method of implementing micro front-ends is to set up a registry of available components and allow an application builder to pick and choose which ones they wish to work with. Solutions like Bit (`bit.dev`) allow the publication of separate components that are all available as a "toolbox" to the end application developer. Installing and using components in this way allows for far easier code re-use, and is particularly useful for large-scale teams that are all working within the same JavaScript framework.

This technique, however, is not really a true reflection of the micro front-end ethos; it is more of a mechanism for component re-use. Bit, as an example, will only allow components to be shared if they are all written in the same framework. In other words, React components are not able to use Vue components, and Vue components are not able to use React components.

The registry technique for building micro front-ends is a simple, viable, and flexible way of re-using components, with the caveat that all front-ends are written with the same JavaScript framework.

What we will use

In this chapter, we will use the JavaScript technique in order to register and co-ordinate our various front-ends into a single, coherent application. This is not the simplest approach for building a micro front-end application, but it does provide the greatest level of flexibility.

By exploring this technique, we will gain an understanding of how micro front-ends can be built using different JavaScript frameworks, and it will give us a reference point that can help to assess third-party solutions. Each implementation of a micro front-end architecture will differ, and the decision to move in the direction of micro front-ends is by no means a simple one to make. By working through the concepts in this chapter, we will see what it takes to build a micro front-end-capable solution with the most flexibility, even if this introduces some complexity.

Communication mechanisms

In a micro front-end application, we will need to have some sort of communication mechanism between each of the front-ends. The mechanism we use for this communication will be different based on which micro front-end solution we choose to proceed with.

As an example of front-end-to-front-end communication, how does our application know when to show the list of products and when to show the shopping cart? How does each front-end know whether the user is a registered user or not?

Iframe-based micro front-ends can use the `postMessage` method to send messages across an iframe boundary. As discussed earlier, this method needs to ensure that messages are not susceptible to cross-site scripting attacks, and as such, each front-end domain must be known and trusted by the other domains that may wish to send or receive messages.

The `postMessage` method of communication allows any data to be sent as part of the message itself. As long as the origin of the message is trusted, we can send complete JavaScript objects across the iframe boundary, to be interpreted by the receiving iframe as they see fit.

JavaScript-based micro front-end mechanisms tend to use URL paths for communication between front-ends. This means that each front-end must understand what is required of it to display when a specific URL is encountered. This technique is fairly simple to implement, as most JavaScript frameworks are able to interpret routes easily and respond accordingly. The use of URL paths, however, is fairly coarse-grained, and is a limited mechanism for effective communication. We cannot send particular data in relation to a URL without using URL parameters, which are visible to the end user and can easily be manipulated.

An effective mechanism for communication between front-ends is the use of a shared Event Bus, where each front-end can register its interest in a particular message, or can publish a specific message. This means that one front-end can register to receive events, and another can produce these events without each front-end knowing of the existence of the other. The concept of Event Buses is a widely used pattern in modern application architecture, and there are countless versions of Event Buses available for any operating system and any architectural design.

The DOM itself already provides a mechanism for event subscription, through the use of the `addEventListener` and `removeEventListener` functions. This could potentially be re-purposed in order to create a shared Event Bus.

We have, however, already come across a JavaScript event bus in our discussion on the RxJS library, and the use of Observables and Subjects. We also implemented a version of an Event Bus in the chapter on Angular, where we broadcasted and subscribed to messages in order to simplify communication between Angular components. This implementation followed the concept of the Domain Event design pattern, where a particular event that is important within a domain is raised by a component, and other components react to this event accordingly. The sender of the event is de-coupled from the consumer of the event.

In this chapter, we will use this Domain Event design pattern, and create a `MicroEventBus` class that is based on the `BroadcastService` as an Event Bus in order to communicate events between micro front-ends. This gives us the greatest level of flexibility for communication between front-ends, both in terms of the data that is contained within an event and the registration and propagation of events across micro front-ends.

Domain events

Let's now take a look at the overall structure of our application, and identify which domain events will need to be communicated across our front-ends, as shown in the following diagram:

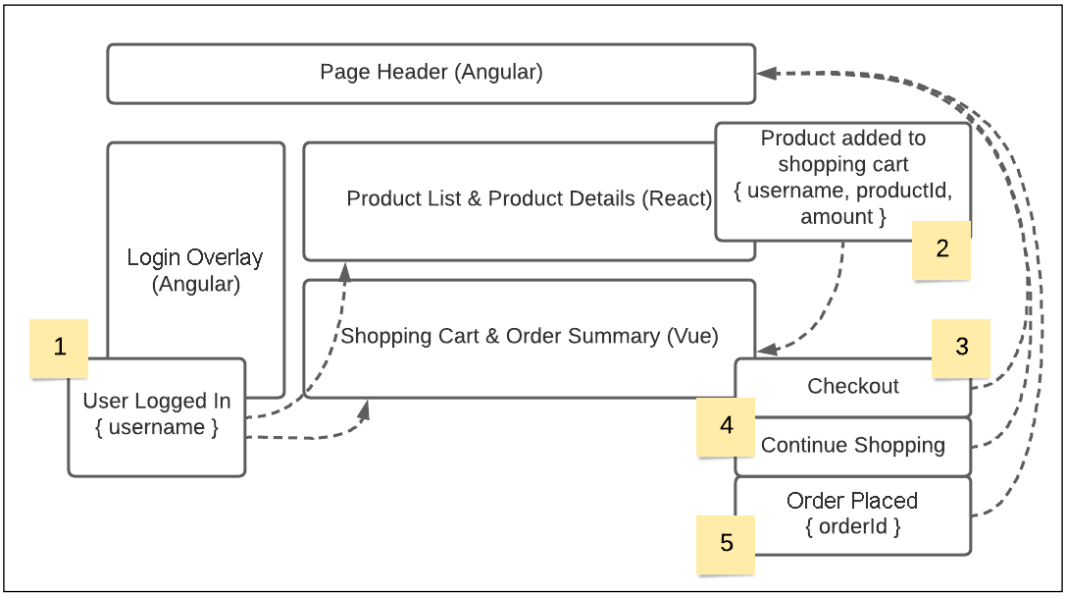


Figure 16.2: Logical view of micro front-ends showing domain event producers and consumers

Here, we start with an event, numbered 1 on the diagram, that is broadcast to all front-ends when a user logs in to the application. It contains a single piece of data, which is the `username`. The origin of this event is the Angular front-end, and the consumers of this event are the Product List and Shopping Cart front-ends. This event allows our front-ends to determine whether they are working with a registered user or not, and if so, what the `username` is.

The second event that we will need to broadcast, numbered 2 on the diagram, is when a user adds a product to their shopping cart. The producer of this event is the Product List front-end, and the consumer of this event is the Shopping Cart front-end. This event will include the `username` of the logged-in user, the `productId` that has been chosen, as well as the `amount` of items the user has selected.

The third, fourth, and fifth events that we will need originate from the Shopping Cart front-end, and are consumed by the Angular front-end. Event number 3, which is the Checkout event, signals that the user wishes to check out with their current shopping cart. If they change their mind, however, the front-end can raise event number 4, which is the Continue Shopping event.

This will allow the Angular front-end to co-ordinate the layout switch from the Checkout view back to the Product List view. The fifth and final event we will need is when a user places an order. This event will trigger a "Thank you" page, but in a real-world application would transition to some sort of payment option page.

The Event Bus

When a **Single-Page Application** (**SPA**) is packaged for production use, it generally contains a copy of all of the libraries and code that it needs in order to run. The output of this packaging step, or bundling step, is a single HTML page that loads JavaScript files that contain all of the dependent libraries. If the RxJS library was used within the application, for example, then the RxJS library files themselves will make up part of this bundled file. This ensures that each SPA has all the dependencies it needs in order to run.

This presents us with a small problem when trying to build an Event Bus that will communicate with each front-end. The Event Bus code must exist outside of the code base for each front-end but still be accessible within it. Luckily, we can attach an instance of the Event Bus to the top-level DOM element named `window`, as shown in the following diagram:

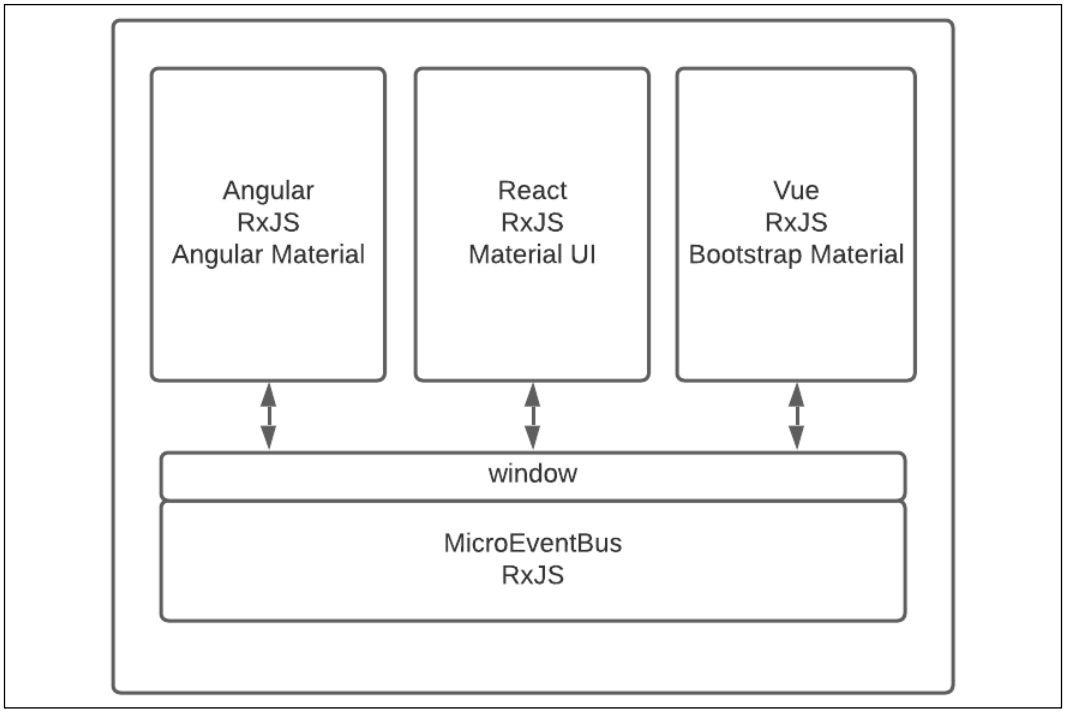


Figure 16.3: Logical view of front-ends communicating with the global Event Bus

Here, we can see that each front-end will have access to the Event Bus named `MicroEventBus`, which is attached to the top-level `window` DOM element. Note too that each of these front-ends and the Event Bus itself use the RxJS library, but that this library is copied into the bundled versions of each front-end.

To access the shared Event Bus, we will just need to reference the global version attached to the `window` DOM element, as in `window.microEventBus.broadcast`, or `window.microEventBus.on`. The Event Bus code itself will need to ensure that this global variable is created correctly.

Now that we have an idea of how we can build a micro front-end application, let's go ahead and put it together.

Building a micro front-end application

In this section of the chapter, we will go ahead and build our micro front-end application, and combine our Angular, React, and Vue front-ends into an application that behaves like a single store front. We will also integrate our front-ends with the REST API that we built in *Chapter 15*, *An AWS Serverless API*. This will ensure that both the React Product List front-end and the Vue Shopping Cart front-end work off the same list of products, as exposed by the `/products` endpoint.

In a production front-end, the general idea is that each application is a "silo," and therefore has its own REST endpoints and databases, in such a way that each team has the freedom to modify any piece of their architecture without affecting other front-ends. In reality, though, there will always be some sort of shared data that all front-end applications may need access to. As an example, a store of registered users might be shared across front-ends, or a store of products. This may take the form of replicated databases, or it may take the form of shared micro services.

In our discussions thus far, we have talked about using the JavaScript technique for building our micro front-end application. This technique utilizes a "host" application that is responsible for loading each front-end, and embedding it within the HTML page. We will use our existing Angular application as the "host" application, and work toward embedding our React and Vue front-ends into the Angular application.

This means that our Angular application will control the page flows, and will embed the other front-ends into either new or existing Angular pages. It will be responsible for showing or hiding front-ends based on the overall page state, and will also communicate with the Event Bus to raise events as we have seen in our diagrams earlier.

Throughout this chapter, we will be updating our existing Angular, React, and Vue applications, in order to get them to work within a micro front-end architecture.

We will, however, only show the modifications to each of these applications in this chapter, and will not discuss any code that has not been changed. In general, the updates to each of these applications are very minor, and will only affect a few files. For the purposes of this chapter, we will assume that the existing source code from chapters 11, 12, 13, and 15 have been copied into the existing directories:

- `/angular-app`: The Angular application from *Chapter 11*, *Angular*
- `/product-list`: The React product list application that we built in *Chapter 12*, *React*
- `/shopping-cart`: The Vue shopping cart application that we build in *Chapter 13*, *Vue*
- `/api-app`: The AWS REST API that we built in *Chapter 15*, *An AWS Serverless API*

Note that we will also be building a new Event Bus in the folder `/micro-event-bus`.

When referencing application source files in this chapter, we will prefix the filename with the above directories, in order to indicate which application files are being modified.

The global Event Bus

Before we start integrating our front-ends, let's build the Event Bus that we will use to facilitate communication of domain events. As each of our front-ends will either send or receive events through this bus, we will need an easy way to simulate this environment when developing each of our front-ends. In other words, if the React front-end must respond to the `user-logged-in` event, then we should be able to run our React application in isolation, and simulate one of these events.

In reality, simulation of these events takes a few lines of code in a `<script>` tag, and is fairly trivial, as we shall see.

To start with, all we need is a new npm and TypeScript project, as follows:

```
mkdir micro-event-bus
cd micro-event-bus
npm init
tsc --init
npm install rxjs
mkdir src
```

Here, we have created a directory named `micro-event-bus`, and set up a new npm and TypeScript project within it. We have also created a `src` directory to hold our source code.

The Event Bus that we will be building is an updated version of the `BroadcastService` that we used in our Angular application. In a micro front-end architecture, it is very important to clearly define both the messages that will be passed between front-ends and also the message contents. The messages effectively become the contract or API between front-ends, and therefore any changes to these messages, which could affect each front-end, must be clearly documented and reviewed. Let's therefore ensure that we describe each of these messages clearly, by defining them in TypeScript, and then expose them to our front-ends using a declaration file.

The `/micro-event-bus/src/MicroEventBus.ts` file will contain our definition of the available events as follows:

```
interface IDomainEventKey {
    "checking-out": null;
    "continue-shopping": null;
    "place-order": null;
    "user-logged-in": string;
    "add-user-cart-item": IAddUserCartItemData;
}

interface IAddUserCartItemData {
    productId: number;
    username: string;
    amount: number;
}

export interface IBroadcastEvent
    <T extends keyof IDomainEventKey>
{
    key: T;
    data?: IDomainEventKey[T];
}
```

Here, we start with an interface named `IDomainEventKey`, which holds the list of available events that any one of our front-ends can broadcast on the Event Bus. Each of the available messages is a string property of the interface, and each defines a type that is associated with the message. The `"checking-out"`, `"continue-shopping"`, and `"place-order"` messages do not have any data associated with them, and therefore we specify their type as null. When we raise a `"user-logged-in"` message, we will need to provide the username, which is a string, as part of the message. Similarly, when we raise an `"add-user-cart-item"` event, we will need to provide an object that has the properties named in the `IAddUserCartItemData` interface, which includes the `productId`, `username`, and `amount` properties.

Note the definition of the `IBroadcastEvent` interface. This interface is using generic syntax to define a type named `T` that `extends keyof IDomainEventKey`. In other words, `T` can only be a string that matches one of our defined event names or interface property names in the `IDomainEventKey` interface. The `IBroadcastEvent` interface has two properties, named `key` and `data`. The `key` property is of type `T`, and the data property is of type `IDomainEventKey[T]`. In other words, the `data` property is the type of the named property of the `IDomainEventKey` interface. This is how we can tie an event key, which is a string, to the data that the event must carry.

With these definitions, we can now write an Event Bus implementation in the file `micro-event-bus/src/MicroEventBus.ts`, as follows:

```
export class MicroEventBus {
    private eventBus = new Subject<any>();

    on<T extends keyof IDomainEventKey>
        (key: T): Observable<IDomainEventKey[T] | undefined> {
        console.log(`BCST: on(${key})`)
        return this.eventBus.asObservable().pipe(
            filter(
                (event: IBroadcastEvent<T>) => event.key === key),
            map(
                (event: IBroadcastEvent<T>) => event.data));
    }

    broadcast<T extends keyof IDomainEventKey>
        (key: T, data: IDomainEventKey[T]) {

        console.log(`BCST: broadcast: ${key} : ${data}`)
        this.eventBus.next({ key, data });
    }
}
```

Here, we have defined a class named `MicroEventBus` that has a single private property named `eventBus`, and two member functions, named `on` and `broadcast`. Note how we have used the `IDomainEventKey` interface in these member function definitions. The `on` function takes a single parameter named `key` of type `T`, and returns an Observable of type `IDomainEventKey[T]`, or undefined. This matches our definition of the `key` and `data` properties of the `IBroadcastEvent`. The `broadcast` function also defines the `key` and `data` parameters in the same way.

The implementation of the `MicroEventBus` is almost identical to the implementation of the `BroadcastService` that we used in *Chapter 11, Angular*. It allows interested parties to register their interest in a domain event using the `on` function, and allows the generation of a domain event through the `broadcast` function.

We can now create an instance of the `MicroEventBus` class, and attach it to the global `window` property as follows:

```
interface IWindowEventBus {
    microEventBus: MicroEventBus;
}

declare let window: IWindowEventBus;

if (window.microEventBus === undefined) {
    console.log(`BCST : window.microEventBus = undefined,
        setting window.microEventBus `);
    window.microEventBus = new MicroEventBus();
}
```

Here, we have described an interface named `IWindowEventBus` that contains a single property named `microEventBus` of type `MicroEventBus`. We then declare a variable named `window` of type `IWindowEventBus`. This allows us to associate a type with the global `window` instance, and specify that it has a property on it named `microEventBus`.

The final `if` statement in this file checks to see if the property named `microEventBus` exists on the global `window` object, and will create it if it is undefined. This will ensure that only a single instance of the `MicroEventBus` class is attached to the `window` property, no matter how many times this code is included or run.

Building a module

Our `MicroEventBus` class has a single dependency, which is the RxJS library. If we were to include the JavaScript file that we have generated, that is, `MicroEventBus.js`, directly into an HTML page and attempt to run it, we would find two errors logged to the console. These are:

```
Uncaught ReferenceError: exports is not defined
Uncaught TypeError: Cannot read property 'on' of undefined
```

These errors are indicating that we have a few dependencies in our code that have not been met. The first error is caused by the use of the `export` keyword in our TypeScript code. Unfortunately, the `export` keyword, and the `import` keyword for that matter, are used for module resolution. In other words, if we `import` a library, the runtime must be able to find that library somewhere and understand how to import it. Exports are the other way around, where we are defining something as being exported, and the runtime must know how to register this with the module resolving routines.

The second error is due to our code importing the RxJS library, as the `on` function that we are using is defined in this library. What this means is that we need to include the RxJS library as a dependency within our library output.

Both of these errors can be resolved by packaging our code correctly, which we can do by building a self-contained module. We will use webpack for this, so let's install the webpack npm library as follows:

```
npm install webpack --save-dev
npm install webpack-cli --save-dev
```

Here, we have installed `webpack` and the `webpack-cli` package as developer dependencies.

With webpack installed, we can now create a file at the root of the project named `micro-event-bus/webpack.config.js`, as follows:

```
const path = require('path');

module.exports = {
    entry: './src/MicroEventBus.js',
    output: {
        filename: 'MicroEventBus.js',
        path: path.resolve(__dirname, 'dist'),
    },
    mode: 'production'
};
```

Here, we define the `exports` property on the `module` namespace with the properties of `entry` and `output`. The `entry` property points to the JavaScript file that is generated by the TypeScript compilation step. The `output` property specifies that we wish to generate a file named `MicroEventBus.js` within the `dist` directory.

We can now run webpack as follows:

```
npx webpack
```

This command will execute the `webpack` CLI, read the `webpack.config.js` file, and produce a file named `MicrEventBus.js` in the `dist` directory. This single file will actually include all of the libraries that our code has imported, which includes the RxJS library.

We can now put our module to the test by writing a simple HTML page in the `dist` directory as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Getting Started</title>
</head>
<body>
<script src="MicroEventBus.js"></script>

<script>
    console.log('running script');
    testBroadcast();
    function testBroadcast() {
        console.log('subscribing');
        window.microEventBus.on('test-event')
            .subscribe(function (event)
            {
            console.log('IDX : testBroadcast got : ' + event)
            });
    }

    function sendMessage() {
        window.microEventBus
            .broadcast('test-event', 'test message');
    }
</script>
<button onclick="sendMessage()">Click me</button>
</body>
</html>
```

Here, we have an HTML page that includes the `MicroEventBus.js` file within a `<script>` tag. We then have a little bit of embedded JavaScript within another `<script>` tag. This embedded JavaScript has two functions, named `testBroadcast` and `sendMessage`. The `testBroadcast` function calls the `on` function of the `window.microEventBus` object in order to register interest in an event named `'test-event'`. It then subscribes to this Observable, and executes a function when the event is received. This function is the consumer of the event.

The `sendMessage` function sends an event through the `broadcast` function of the `window.microEventBus` object. This function is the provider of the event.

We then have a button at the bottom of the HTML page that invokes the `sendMessage` function when the button is clicked. If we now run a web server in the `dist` directory, using the `http-server` node application, and point our browser to this page, we will see these events firing, as shown in the following screenshot:

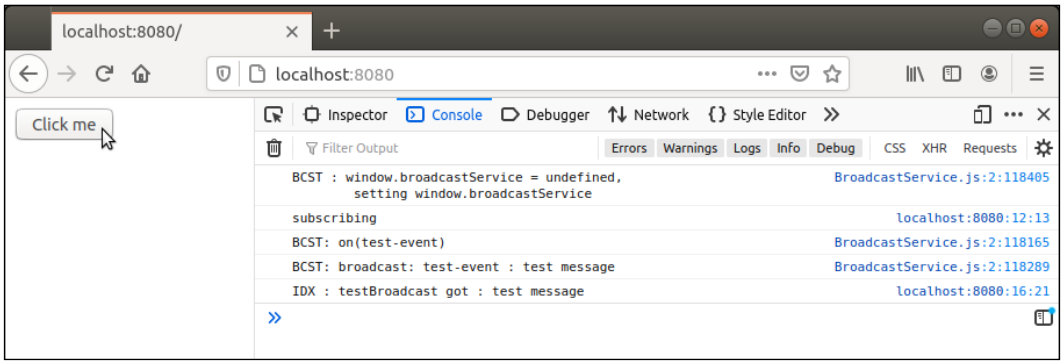


Figure 16.4: Console log showing both producer and consumer of events on the Event Bus

Here, we can see that there are four console log messages being produced. The first message comes from code within the `MicroEventBus.js` module, which is detecting that the `window.microEventBus` variable has not been assigned as of yet, and is therefore doing so. We then get a message from our embedded JavaScript stating that we are subscribing to the event named `'test-event'`. This call triggers another console log message from our `MicroEventBus`, which shows that it has added a listener to the event named `'test-event'`.

The final two messages are produced when we click the button, and the `sendMessage` function is raising an event named `'test-event'`, with the value `'test message'`. Our listener is receiving this event.

Our Event Bus is now working, and crucially, is working on the global variable named `window.microEventBus`. We will be using embedded JavaScript like this when we need to simulate events for each of our front-ends.

Note that the `'test-event'` event is not listed as an available event on our `IDomainEventKey` interface, but our Event Bus still broadcasts it, and our subscriber is notified. This shows that we are running a JavaScript version of our Event Bus, which is blissfully unaware of any types or type constraints. The only reason our Event Bus works is that the event name that both the broadcaster and subscriber are using happens to be the same.

Module typing

Now that we have defined our Event Bus, and have packaged it using webpack, we will need to ensure that all clients of our Event Bus have access to the types that we have defined for it. In other words, let's generate an `@types` definition for our `MicroEventBus` class, so that it can be used safely within our other applications. This can be achieved by generating a declaration file from our TypeScript code, and then including this declaration file in the definition of our module.

To generate a declaration file, we can set the compiler option named `"declaration"` in our `micro-event-bus/tsconfig.json` file, as follows:

```
{
    "compilerOptions": {
        "target": "es2015",
        "module": "commonjs",
        "declaration": true,
        ... other options

    }
}
```

Here, we have set the `"declaration"` compiler option to `true`. This will generate a declaration file named `MicroEvenBus.d.ts` when we invoke the compiler, by executing `tsc`.

This declaration file, however, will be created in the `micro-event-bus/src` directory. What we will need to do is to copy it into the `micro-event-bus/dist` directory in order to make it available for use. We can accomplish this by using the `copy-webpack-plugin` library, which can be installed as follows:

```
npm install copy-webpack-plugin --saveDev
```

With the library installed, we can now copy the declaration file from our `src` directory to the `dist` directory by updating our `webpack.config.js` file, as follows:

```
const path = require('path');
const copyWebpackPlugin = require('copy-webpack-plugin');

module.exports = [
    {
        entry: './src/MicroEventBus.js',
        output: {
            filename: 'MicroEventBus.js',
```

```
            path: path.resolve(__dirname, 'dist'),
        },
        mode: 'production',
        plugins: [
            new copyWebpackPlugin({
                patterns: [
                    {
                        from: "src/MicroEventBus.d.ts",
                        to: "MicroEventBus.d.ts"
                    },
                    {
                        from: "src/index.html",
                        to: "index.html"
                    }
                ]
            })
        ]
    }
];
```

Here, we have made a few updates to our `micro-event-bus/webpack.config.js` file. Firstly, we have imported the `copy-webpack-plugin` library and assigned it to a constant named `copyWebpackPlugin`. Secondly, we have added a `plugins` property to the configuration object, and are using the `copyWebpackPlugin` library to copy files into our `dist` directory. The `patterns` property specifies which files we need to copy, and in our case, we are specifying that the generated `src/MicroEventBus.d.ts` declaration file should be copied into the `dist` directory, along with the `src/index.html` file. Running `npx webpack` with these updates will copy these two files into our `dist` directory.

In order to use these types in our client applications, we will need to do two things. Firstly we will need to attach the declaration file to the library so that any TypeScript client can use it, and secondly, we will need to make this library available through npm.

To attach the declaration files for outside use, we will update our `micro-event-bus/package.json` file as follows:

```
{
    "name": "micro-event-bus",
    "version": "1.0.0",
    "description": "",
    "main": "dist/MicroEventBus.js",
    "types": "dist/MicroEventBus.d.ts",
    ... existing config
}
```

Here, we have updated the `"types"` property to point to the `"dist/MicroEventBus.d.ts"` declaration file. This means that any TypeScript user of this library will be able to find our declaration files.

Now that we have packaged our declaration file for our `MicroEventBus` class, we can package this for use by npm. Normally, this would mean publishing the library publicly, as in on a publicly available web server, but in development, we can publish this library locally with the following command:

```
npm link
```

Here, we are using the `npm link` command to make this library available as a global node module. On Linux systems, for example, this command creates a symbolic link between the directory `/usr/local/lib/node_modules/micro-event-bus` and our current source directory. This means that the `micro-event-bus` library is available for use by npm in our target applications on our local machine.

React updates

Let's now turn our attention to our Product List front-end that we built using React. We will need to update it in a few places, as follows:

- Fetch data from the `/products` API endpoint to populate our list of products
- Integrate the `MicroEventBus` in a test environment
- Listen for a `'user-logged-in'` event, and store the provided username locally
- Broadcast an event when a user adds a product to their shopping cart

Loading data from an API

Our React application has been displaying a list of products from a manually generated array. This array is defined in the `product-list/src/Products.tsx` file, and is the `items` property of the `Collection` class. We will now load these items from our `/products` API endpoint. We will still need the `Collection` class, but it now becomes just a holder for the array of `items`, as follows:

```
export class Collection {
    handleItemClicked(id: number) { }
    items: IProduct[] = [];
    //     these manually created items can be deleted
    //     [
    //         {
```

```
    //         id: 1,
    //         name: "Holy Panda"
    //         }
    //     ]

}
```

Here, we have removed the manually created array of items, and the `items` property of the `Collection` class is now just an empty array of items of type `IProduct`. This `items` property will be populated with the results of the API call.

The next update we will need to make is to make the instance of the `Collection` class part of the App component state, as follows:

```
export class AppState {
    showDetails: boolean = false;
    product: IProduct | null = null;
    collectionInstance: Collection;
}

// this class instance will now be part of the App state
// const collectionInstance = new Collection();
```

Here, we have added the property named `collectionInstance` to the `AppState` class, and have commented out the global instance of the `Collection` class. This change means that the list of products we are using is now part of the state of the App class, and when it changes, it will force a refresh of the entire component. Note that we will need to update any references to the global `collectionInstance` variable with the `this.state.collectionInstance` variable throughout the `App.tsx` class. One example of this is in the `render` function, where we pass the `collectionInstance` variable as a property down to the `CollectionView` component, as follows:

```
<CollectionView
    // {...collectionInstance}
    // replaced with:
    {...this.state.collectionInstance}
    handleItemClicked={this.showDetailView} >
</CollectionView>
```

Here, in our `render` function, we pass the value of the `collectionInstance` variable down into the `CollectionView` component. This will need to be updated to reference the `this.state.collectionInstance` instance instead.

After a React component has been mounted into the DOM, the `componentDidMount` function of the component will be called. This is where we are able to load our data from the API endpoint, as follows:

```
componentDidMount() {
    ajax.getJSON("http://127.0.0.1:3000/products")
        .subscribe((data) => {
            let collection = new Collection();
            collection.items = data as IProduct[];
            this.setState({
                showDetails: false,
                product: null,
                collectionInstance: collection
            });
        });
}
```

Here, we are using the `ajax` object that is part of the RxJS library, in order to fetch data from an API endpoint. We are calling the `getJSON` function and supplying this function with an endpoint that will return our product list. In this example, we are hard-coding the endpoint to be served by our local sam instance, and are using `localhost:3000/products` as our endpoint. We then `subscribe` to this function call, and when the product list is returned, we assign the returned JSON structure to the `collection.items` property. Once the `items` array has been populated, we call the `setState` function in order to re-render our component.

Note that we will need to install the RxJS library for use in our React application, by executing `npm install rxjs`, and we will also need to import the `ajax` object at the top of our `App.tsx` file as follows:

```
import { ajax } from 'rxjs/ajax';
```

If we refresh our application now, we will find that a blank page is rendered while the `componentDidMount` function is waiting for a response from the API endpoint. Let's tweak our rendered HTML a little, in order to show a progress indicator while we wait, by updating the `render` function as follows:

```
render() {
    return (
        <div>
            { this.state.collectionInstance.items.length < 1 ?
                <div className="App-header">
                    <CircularProgress />
                </div>
```

```
                : ""}
            <CollectionView
                {...this.state.collectionInstance}

                ... existing components

        </div>
    )
}
```

Here, we have introduced a little bit of logic into our `render` function that will check if the length of the array named `items` on our `collectionInstance` variable is less than `1`. If it is, we display a Material UI component named `CircularProgress`. The effect of this is that when we are waiting for the API endpoint to return a list of products, we will have a spinner progress control displayed, instead of simply a blank screen.

React domain events

There are two Domain Events that our React front-end needs to deal with. The first is an incoming event where we are notified that a user has logged in to the application. This event provides us with the username of the logged-in user, which we will need when we broadcast a Domain Event. The second Domain Event is one that we will need to broadcast is when a user clicks on the **Add to Cart** button in the `DetailView` component. This event will include the `username`, the `productId` that was selected, as well as the `amount` of switches the user has entered.

In order to test that our React application can consume and produce these Domain Events in a test environment, we will need to copy the `MicroEventBus.js` file from our `micro-event-bus/dist` directory into the `product-list/public` directory. We will also need to update our `product-list/public/index.html` file as follows:

```
<html lang="en">
  <head>
    .. existing tags
    <script src="/MicroEventBus.js"></script>
  </head>
  <body>
    <noscript>
        You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
```

```
    <script>
        testBroadcast();
        function testBroadcast() {
            window.microEventBus.on('add-user-cart-item')
                .subscribe(function (event)
            {
                console.log(
                    `IDX : add-user-cart-item received : `);
                console.log(
                    `IDX : ${JSON.stringify(event, null, 4)}`);
            });
        }

        function sendMessage() {
            window.microEventBus
                .broadcast('user-logged-in', 'test_user_1');
        }
    </script>
    <button onclick="sendMessage()">Send user event</button>
  </body>
</html>
```

Here, we have updated our `index.html` file and included the `MicroEventBus.js` file as a `<script>` resource. We have also added a `<script>` section to the `<body>` tag of this file, in a similar manner to the test code that we used in the `micro-event-bus/dist/index.html` file. This test code registers a listener to the `'add-user-cart-item'` domain event, which will log the values found as properties of the event to the console.

We have also added a `sendMessage` function to broadcast an event named `'user-logged-in'`, with the name of the user set to `'test_user_1'`. Our test code then includes a `<button>` element that will call the `sendMessage` function when clicked.

Our React application will now need to consume the `micro-event-bus` npm library that we created earlier, which we linked to our globally available `node_modules`. This can be accomplished in our local development environment by again using `npm link` as follows:

```
npm link micro-event-bus
```

Here, we are using the `npm link` command to find and link the `micro-event-bus` library for use within this project. This has the same effect as installing an npm library using `npm install`, but will reference the local version of this package instead.

The updates that we need to make to our application in order to react to the `'user-logged-in'` event are mostly in the `product-list/src/App.tsx` file as follows:

```
import { MicroEventBus } from 'micro-event-bus';
export let microEventBus: MicroEventBus =
    (window as any).microEventBus;

export interface IAppState {
    ... existing properties
    username: string;
}

class App extends
    React.Component<IAppProps, IAppState> {

    ... existing constructor

    componentDidMount() {
        this.processUserLoginEvent =
            this.processUserLoginEvent.bind(this);

        microEventBus.on('user-logged-in').subscribe(
            this.processUserLoginEvent
        );

        ... existing call to API
    }

    render() {
        return (
            <div>
                ... existing items
                <DetailView open={this.state.showDetails}
                    product={this.state.product}
                    handleClose={this.handleClose}
                    username={this.state.username}>
                </DetailView>
            </div>
        )
    }
    ... existing functions
```

```
    processUserLoginEvent(event: string | undefined) {
        console.log(`RCT : App.processUserLoginEvent : ${event}`);

        if (event) {
            this.setState({ username: event });
        }
    }
}
```

Here, we start by importing the `MicroEventBus` definition, and defining a variable named `microEventBus`, of type `MicroEventBus`. This variable is set to the global instance of our Event Bus that is attached to the `window` object. This is the handle that we need to either subscribe to or publish events on the Event Bus.

We have then added a `username` property to the `IAppState` interface, so that we can track it through our component state. We have also added a function named `processUserLoginEvent` that will be called when we receive the `'user-logged-in'` event, and we are binding the correct instance of `this` to the function, within the `componentDidMount` function.

We then subscribe to the `'user-logged-in'` event, and when it is raised, we invoke the `processUserLoginEvent` function. The function definition of the function that is called when the `'user-logged-in'` event is raised must match the definition that is in our declaration file for the `micro-event-bus` library. Remember that the `subscribe` function for the `'user-logged-in'` event has a type of string or undefined, and therefore the `processUserLoginEvent` function must match this function signature. The declaration file for our `MicroEventBus` class is ensuring that we define this function correctly. If we attempt to use a function signature that does not match, TypeScript will generate an error. This is a great way of using TypeScript to ensure that when an Event Bus message is received, we are aware of exactly what information it is carrying in its `data` property, and can process it correctly.

Note that the `DetailView` component also needs to know about the current username, so we pass this into the `DetailView` props within our `render` function. The implementation of the `processUserLoginEvent` function calls `setState` with the value coming in through the event.

Our `product-list/src/DetailView.tsx` file needs three minor updates, as follows:

```
import { microEventBus } from "./App";

export interface IDetailsProps {
    open: boolean;
    product: IProduct | null;
    handleClose(): void;
    username: string;
}

export class DetailView extends
    React.Component<IDetailsProps, IDetailsState>
{

    ... existing functions

    onSubmit(e: React.FormEvent) {
        console.log(`submit ; ${this.state.noSwitches}`);
        e.preventDefault();

        if (this.props.product) {
            microEventBus.broadcast('add-user-cart-item', {
                username: this.props.username,
                productId: this.props.product?.id,
                amount: this.state.noSwitches
            });
        }
    }
}
```

Here, we start by importing the `microEventBus` variable from the `App` component. We have then added the `username` property to the props that are exposed for the `DetailView` component, which will allow the `App` component to set it. Finally, in our `onSubmit` function, we are raising an event on the Event Bus by calling the `broadcast` function with the `'add-user-cart-item'` key, and an object containing the `username`, `productId`, and `amount` values.

With these changes in place, we can now fire up our browser, and see how our React application reacts to the domain events (no pun intended). This can be seen in the console log as shown in the following screenshot:

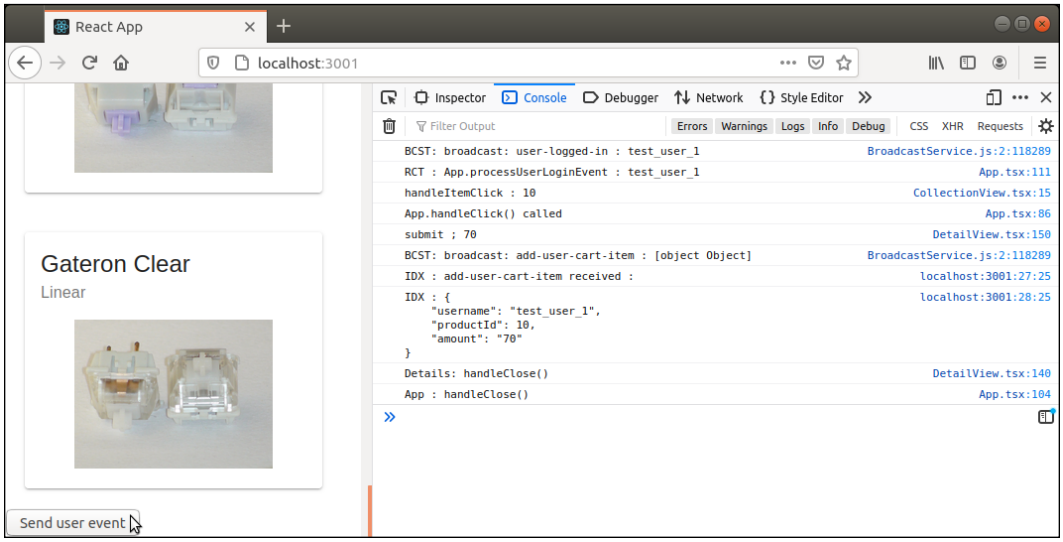


Figure 16.5: Console logs showing events from the Event Bus being processed in a test harness

Here, we can see that the first two logs are generated when we click the button named **Send user event** at the bottom of the page. The global `microEventBus` instance is sending an event named `'user-logged-in'` with the name of the user, which is `'test_user_1'`, and our React application is processing this event. Also, when we click on the **Add to Cart** button on a details page, React is raising an event named `'add-user-cart-item'`, and our test code that is running within our HTML page is picking up on this event.

Our changes to our React application are complete. With a few minor modifications to our code, it is ready to send and receive Domain Events through the global Event Bus. By using the declaration file that is included with our `micro-event-bus` library, we are also able to ensure that only defined events can be sent and received, and that the type of the data packet for each of these events is correctly typed.

Before we move on, however, we will need to build a production-ready version of our React application for inclusion as a micro front-end. This can be accomplished quite simply by running the following:

```
npm run-script build
```

Here, we are invoking the `react-scripts` CLI to generate a distributable version of our React application. The output of this will be placed into the `product-list/dist` folder. We will be using this output when we wish to inject the React application as a micro-front-end into our final application.

Vue updates

We can now turn our attention to our Vue application, which will have similar changes made to it to send and receive messages via the Event Bus. We will also need to integrate with our API. A list of changes to our Vue application are as follows:

- Listen for the `'user-logged-in'` event, and store the username
- Fetch the shopping cart from the API via the `/carts/{username}` endpoint
- Listen for the `'add-user-cart-item'` event, and add the item to the current shopping cart
- Add a new button called `'Update Cart'` that will call the API to store the shopping cart
- Send a `'checking-out'` event to the Event Bus
- Send a `'continue-shopping'` event to the Event Bus
- Send a `'place-order'` event to the Event Bus

Vue domain events

Let's start our updates to the Vue application by firstly copying the `micro-event-bus/dist/MicroEventBus.js` file into the `shopping-cart/public` directory, and then updating the `shopping-cart/public/index.html` file for testing, as follows:

```
... existing HTML

<script type="text/javascript" src="MicroEventBus.js">
</script>
</head>
  <body>

    ... existing HTML
    <script>
        testBroadcast();
        function testBroadcast() {
```

```
            window.microEventBus.on('continue-shopping')
                .subscribe(function (event)
            {
                console.log('IDX : continue-shopping : ' + event)
            });
            window.microEventBus.on('place-order')
                .subscribe(function (event)
            {
                console.log('IDX : place-order : ' + event)
            });

        }

        function sendUserLogin() {
            window.microEventBus
                .broadcast('user-logged-in', 'test_user_1');
        }
        function sendAddToCart() {
            window.microEventBus
                .broadcast('add-user-cart-item',
            {
                username : "test_user_1",
                productId: 3,
                amount : 303
            });
        }


    </script>
    <button onclick="sendUserLogin()">
        Send 'user-login'</button>
    <button onclick="sendAddToCart()">
        Send 'add-user-cart-item'</button>
  </body>
```

Here, we have made three changes to the file, in a very similar manner to the changes we made to our React `product-list/public/index.html` file for testing. Firstly, we have included the `MicroEventBus.js` file as a resource. Secondly, we are subscribing to the `'continue-shopping'` and `'place-order'` events, which are raised by our Vue application, and thirdly, we have hooked up a test button to send the `'user-logged-in'` event, and another button to send the `'add-user-cart-item'` event.

Fetching data in Vue

Let's now look at the updates that we need within our Vue application in order to process these incoming events. Before we start, we will again need to link the `micro-event-bus` npm library for use in our Vue application, by executing the following on the command line:

```
npm link micro-event-bus
```

This will make the declaration files available for use within our Vue application.

When we receive a `'user-logged-in'` event, we can make a call to the `/carts/{username}` API endpoint, and populate the list of items in the user's shopping cart. We will again use the RxJS utility functions in order to fetch data from our API, and update our `shopping-cart/src/App.vue` file as follows:

```
import { MicroEventBus } from 'micro-event-bus';
export let microEventBus: MicroEventBus =
    (window as any).microEventBus;

@Options({
    components: {
        ShoppingCart,
    },
    mounted() {
        console.log(`VUE: App:mounted()`);

        microEventBus
            .on('user-logged-in')
            .subscribe(this.handleUserLoggedIn);

    }
})
export default class App extends Vue {
    cartItems!: CartCollection;
    username!: string;
    loading: boolean = false;
    data() {
        return {
            cartItems: [],
            username: null,
            loading: true
```

```
        }
    }
    handleUserLoggedIn(event: string) {
        console.log(`VUE: App.handleUserLoggedIn : ${event}`);
        this.username = event;
        this.loading = true;
        this.fetchData();
    }

    fetchData() {
        console.log(`calling ajax with ${this.username}`);
        ajax.getJSON(
            `http://127.0.0.1:3000/carts/${this.username}`
        ).subscribe((data) =>
        {
            let cartItems = new CartCollection();
            cartItems.items = <IProduct[]>data;
            this.cartItems = cartItems;
            this.loading = false;
        }, (error) => {
            console.log(`VUE: get /carts/(username)
                error : ${error}`);
            this.loading = false;
        });
    }
}
```

Here, we have made a number of changes to our `App.vue` component code. Firstly, we have imported the `MicroEventBus` class definition, and exported a variable named `microEventBus`, which is set to the globally defined instance of our Event Bus, as we did within our React application.

Secondly, we have defined a function named `mounted` within the `@Options` decorator. This function will be invoked by the Vue runtime when a component has been mounted into the DOM, similar to the `componentDidMount` function we use in React, or the `ngOnInit` function in Angular. Within this `mounted` function, we register an event handler for the `'user-logged-in'` event, which will call the `handleUserLoggedIn` function.

The third change we have made is to define a set of properties for the `App` class. The first property will hold the items in the user's shopping cart named `cartItems`, and the second property will hold the username. We also have a boolean property named `loading` to indicate that the component is busy loading data. Our `data` function has also changed in order to set default values for each of these internal properties.

We have then defined a function named `handleUserLoggedIn`, which will store the `username` that was passed in as part of the domain event, set the `loading` property to `true`, and call the `fetchData` function.

The final change we have made is to define a `fetchData` function, which will call the endpoint at `/carts/{username}`, and process the results. When the API endpoint responds to our GET request, we set the internal `cartItems` variable to the result of the API call, and then set the `loading` flag back to `false`.

We can now update the `<template>` section of our `shopping-cart/src/App.vue` file to show when the application is loading a shopping cart as follows:

```
<template>
   <div v-if="loading" class="loading">
      Loading...
    </div>
     <div v-if="!loading" class="loading">
      <ShoppingCart
        :collection="cartItems" />
    </div>
</template>
```

Here, we have a `<div>` tag that will only be rendered when the `loading` class property is set to `true`, and our `ShoppingCart` component will only be shown if the `loading` class property is set to `false`. Our `App.vue` component is setting the value of the `loading` property to `true` just before we call the API, and it is setting it to `false` once the API has returned, and our `cartItems` property has been set.

Raising Events

Our Vue application needs to raise three domain event. These are the `'continue-shopping'` event, the `'checking-out'` event, and the `'place-order'` event. Each of these events will be broadcast from buttons on the `shopping-cart/src/components/ShoppingCart.vue` component, as follows:

```
<template>
    <div class="container">

  ... existing template

        <button
            class="btn btn-primary"
            v-on:click="continueShopping()"
        >Continue Shopping</button>
```

```
    </div>
</template>

<script lang="ts">
import { microEventBus } from "../App.vue";

... existing code

export default class ShoppingCart extends Vue {
    ... existing class definition

    continueShopping() {
        console.log(`VUE : continueShopping()`);
         microEventBus.broadcast(
                'continue-shopping',"");
    }

}
</script>
```

Here, we have added a button named `"Continue Shopping"` that will call the `continueShopping` function on our `ShoppingCart` class. The `continueShopping` function will raise an event with the key `'continue-shopping'`. The `'place-order'` and `'checking-out'` events are raised by other buttons in this component, in the same manner as shown here. Please refer to the source code that accompanies this chapter for a full listing.

We can now fire up our browser, and see these domain events flowing into our Vue application, and flowing out, as shown in the following screenshot:

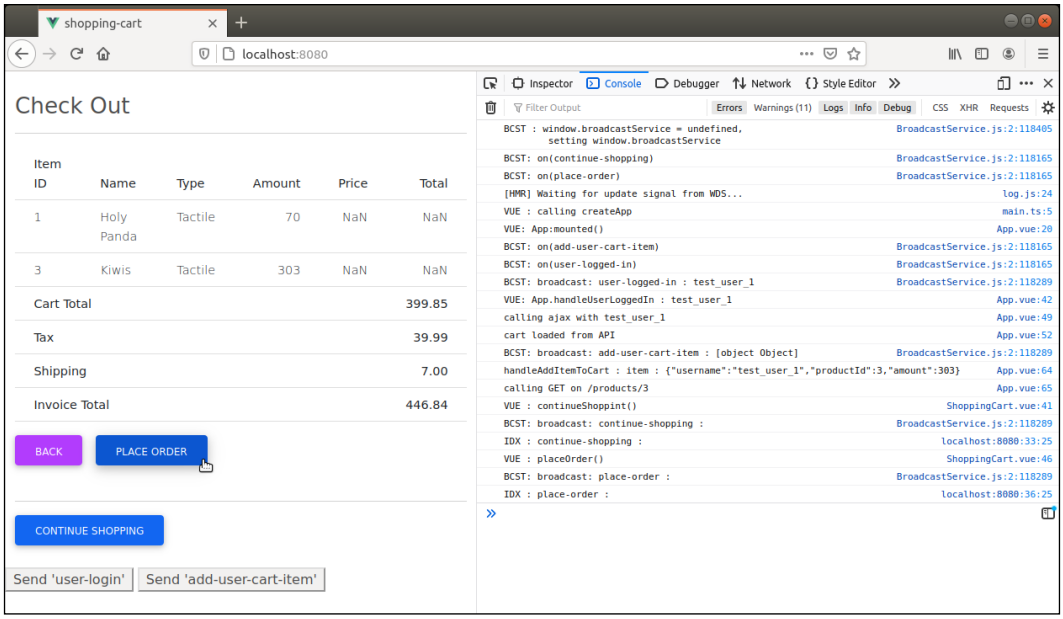


Figure 16.6: Vue application showing console logs of domain events

Here, we can see our two test buttons to generate incoming events at the bottom of the page. When we click on the **Send 'user-login'** button, our Vue application receives a `'user-logged-in'` event, which will load the current cart from the `/carts/{username}` API endpoint. If we then click on the **Send 'add-user-cart-item'** button, this will simulate the event that is generated when a user adds an item to their cart. This will call the `/products/${productId}` endpoint in order to load details about the product added to the cart. We can then see the result of clicking on the **CONTINUE SHOPPING** button as well as the **PLACE ORDER** button, which will raise the corresponding domain events.

Our updates to our Vue application are complete. Again, there is one final step that we must complete, which is to package our Vue application for production use.

Similar to how we packaged our React application into a distributable form, we can generate a package from our Vue application, by updating the `package.json` file as follows:

```
... existing properties
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "build-component":
      "vue-cli-service build --target lib --name ShoppingCartVue"
  },
... existing properties
```

Here, we have added a property named `"build-component"` to the `"scripts"` section within our `package.json` file. This `"build-component"` property will invoke the `vue-cli-service`, and generate a library in a file named `ShoppingCartVue`. Similar to how we used webpack to bundle our `MicroEventBus` along with all of its dependencies into a single file, this command will bundle our entire Vue application and all of its dependencies into a single file named `ShoppingCartVue.umd.js`, and all of our required CSS into a file named `ShoppingCartVue.css`. Note that it will also generate a `ShoppingCartVue.common.js` file, if we need to use CommonJS module loading.

Angular micro front-end

The time has come to finally put all three of our applications together as a single micro front-end application. As we discussed in the *Design concepts* section, we will be using a JavaScript technique to co-ordinate the display of front-ends, and in our case, we will use Angular to do this. This means that we will use our existing Angular application to set up regions within a page to inject our front-ends into.

Please note that the code examples as presented in this chapter are a logical progression of concepts in building our micro front-end. If you are following along, and find that something doesn't quite work as expected, please refer to the sample code that accompanies this chapter. We will be covering quite a bit of ground in terms of source code, and having a code editor open with the final version of the sample code at hand may help to understand how it all fits together.

Micro front-end DOM Containers

Our Angular application already has a header panel, and a login panel that is rendered as a Material Sidenav container. Let's update our `angular-app/src/app/app.component.html` file to set up container `<div>` elements for our micro front-ends to be injected into, as follows:

```
<app-header></app-header>
... existing mat-sidenav
    <mat-sidenav-content>
        <div class="content-padding">
              <div [hidden]="hideProducts" class="split left">
                <h2>Product List</h2>
                <div id="root"></div>
              </div>
              <div [hidden]="hideShoppingCartRegion"
                class="{{shoppingCartStyles}}">
                <h2>Shopping Cart</h2>
                <p *ngIf="hideShoppingCart">
                    Please login to view your shopping cart
                </p>
                <div [hidden]="hideShoppingCart"
                    id="vue-application"></div>
              </div>
              <div [hidden]="hideOrderPlaced" class="split left">
                <h2>Order Placed</h2>
            </div>
        </div>
    </mat-sidenav-content>
</mat-sidenav-container>

<router-outlet></router-outlet>
```

Here, we have updated our `<mat-sidenav-content>` region to include tree `<div>` elements, which have a `[hidden]` attribute that is being controlled by a component variable. For the first `div`, if the `hideProducts` variable on the `AppComponent` class is set to `true`, then this `<div>` element will be hidden. We will be rendering the React Product List front-end into this `<div>` element, and controlling visibility via the `hideProducts` member variable. We also have a `<div>` element for our Vue front-end, and another `<div>` element that will show when an order has been placed.

Note that we are not using an `*ngIf` directive here to control the visibility of a `<div>` element; we are using the `[hidden]` property instead. The reason for this is that both React and Vue, as micro front-ends, need a specific `<div>` element present in the DOM in order to render their content. The `*ngIf` directive will actually remove the `<div>` element from the DOM completely, if the condition is false. This means that we will remove the entire front-end from the DOM if we use an `*ngIf` directive, as it is attached to a specific DOM element. Using the `[hidden]` attribute, however, will just hide the element from view, and will not actually remove it from the DOM.

For the time being, we will set the `hideProducts` member variable to `false`, in order to show the Product List, and set the `hideShoppingCartRegion` variable to `false` in order to show our Shopping Cart, and set all the others to `true`.

Rendering the React front-end

In order to render the React front-end into our Angular page, let's first take a quick look at the output of the React production build, which we used to generate a distributable version of our product list application, in the `product-list/build` directory. This `build` directory for React contains an `index.html` file, among other files, and also includes a `/static/css` directory, and a `/static/js` directory.

The `/static/js` directory contains a number of JavaScript files, which, according to the React documentation, follow a standard naming convention, as follows:

- `main.[hash].chunk.js` – Contains our application code
- `[number].[hash].chunk.js` – Vendor code, or modules from `node_modules`
- `runtime-main.[hash].js` – A small webpack runtime used to load and run the application

If we also have a look at the `product-list/build/index.html` file, we will see that there are two scripts that appear in the `<body>` element, as follows:

```
<body>
    <div id="root"></div>

    ... existing scripts

    <script src="/static/js/2.13772dc5.chunk.js"></script>
    <script src="/static/js/main.0dd1336b.chunk.js"></script>
</body>
```

Here, we have a `<div>` element with an id attribute of `"root"`, which is the top-level DOM element that will contain our React application. We also have scripts named `2.13772dc5.chunk.js` and `main.0dd1336b.chunk.js`. Note that these filenames are generated automatically, so will be different for each application. If you build the React application on your local machine, then these filenames will also change.

In order to use these React files in our Angular application, we will do three things. Firstly, we will include all other files in the `/static/css` and `/static/js` directories, other than the two JavaScript files mentioned previously, into our Angular `index.html` file.

Secondly, we will create a `<div id="root"></div>` element in our `angular-app/src/app/app.component.html` template, so that the React front-end is rendered into this HTML element.

Thirdly, we will inject the two JavaScript files mentioned previously into the `<body>` element once our Angular component has been rendered, within the `ngOnInit` function.

This third step is important. We will need to wait for the `<div id="root">` element to be rendered into the DOM before we attempt to run our React front-end. This is so that the React front-end can attach to this element in the DOM. Remember that the Angular component life-cycle will create our App component, then render its template to the DOM, and only then call the `ngOnInit` function, so once we are in the `ngOnInit` function, we know that the DOM has the `<div>` tag that we need.

We will use a new service named `angular-app/src/app/services/ScriptLoaderService` in order to inject these two JavaScript files into the `<body>` element during our `ngOnInit` function, as follows:

```
ngOnInit() {
    this.scriptLoaderService.loadAllScripts(
        "/assets/react/build/static/js/2.13772dc5.chunk.js",
        "/assets/react/build/static/js/main.0dd1336b.chunk.js"
    ).subscribe(() => {
        console.log(`script loaded`);
    });
}
```

Here, we are calling the `loadAllScripts` function of the local instance of the `ScriptLoaderService` to inject our two JavaScript files into the `<body>` tag at runtime. We will not discuss the implementation of this `ScriptLoaderService` here, so please refer to the supplied source code for a full listing.

We can now update our `angular-app/src/app/app.component.html` file as follows:

```
<div [hidden]="hideProducts" class="split left">
    <h2>Product List</h2>
    <div id="root"></div>
</div>
```

Here, we have added the `<div id="root">` element into the panel that will show the product list. We must now include the remainder of the JavaScript files and the CSS file that are needed by React into our Angular `angular-app/src/index.html` file as follows:

```
<!doctype html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>AngularApp</title>
    <base href="/">

    <script src="/assets/MicroEventBus.js"></script>

    <script
     src="/assets/react/build/static/js/3.ae4dd273.chunk.js">
    </script>
    <script
     src="/assets/react/build/static/js/runtime-main.2fc4af22.js">
    </script>

    <link
     href="/assets/react/build/static/css/main.e4e3bc47.chunk.css"
     rel="stylesheet">

</head>

<body>
    <app-root></app-root>
</body>

</html>
```

Here, we have added the two remaining JavaScript files in `<script>` tags, and included the generated CSS file.

Our Angular application has, in effect, mimicked the `index.html` file that was generated as part of the React build. We have included the same JavaScript files, we have included the same CSS files, and we have included the DOM element for the React front-end to render our application into. The only difference, really, is timing. By injecting two of the React JavaScript files into the DOM during our Angular component's rendering cycle, we can ensure that the required DOM element is available for use, before actually launching our React application.

If we fire up our browser now, and load our Angular application, we will see the React front-end rendering inside our Angular framework, as follows:

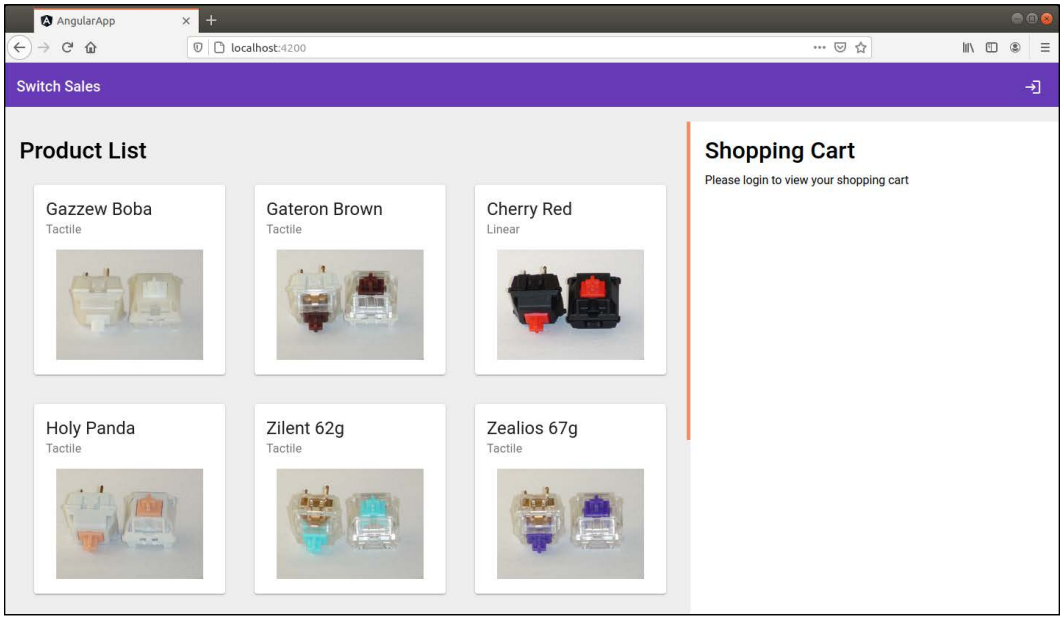


Figure 16.7: React micro front-end rendering inside our Angular application

Here, we can see that the left-hand panel within our Angular application has rendered the React micro front-end.

Rendering the Vue front-end

We can now turn our attention to rendering the Vue front-end into our Angular application. Let's again take a look at the `shopping-cart/dist/demo.html` file that Vue generated for us when we generated a distributable form of the application, as follows:

```
<meta charset="utf-8">
<title>ShoppingCartVue demo</title>
<script src="https://unpkg.com/vue"></script>
<script src="./ShoppingCartVue.umd.js"></script>

<link rel="stylesheet" href="./ShoppingCartVue.css">

<div id="app">
  <demo></demo>
</div>
```

```
<script>
new Vue({
  components: {
    demo: ShoppingCartVue
  }
}).$mount('#app')
</script>
```

There are a few interesting things to note about this HTML file, which will give us clues as to how we can render this Vue front-end into our Angular application.

The first thing to note is that the first `<script>` resource that is loaded is the Vue runtime itself. The second thing to note is that our Vue shopping cart application, and all of its library dependencies, have been bundled into a file named `ShoppingCartVue.umd.js`. The third thing to note is that there is a `<div id="app">` HTML element that our Vue application will be rendered into.

Finally, we have a small script at the end of the page that creates a new instance of Vue, and specifies that it should load our `ShoppingCartVue` as a component, and render this into the `div` named `"#app"`. This little script is an important clue as to when the Vue front-end is rendered, and we will need to simulate this in order to render the Vue application.

We can simulate this little script within our Angular App component by editing the `angular-app/src/app/app.component.ts` file as follows:

```
declare let Vue: any;
declare let ShoppingCartVue: any;

@Component({
    ... existing decorator
})
export class AppComponent {
    ... existing class code
    ngOnInit() {
        this.scriptLoaderService.loadAllScripts(
            "/assets/react/build/static/js/2.13772dc5.chunk.js",
            "/assets/react/build/static/js/main.0dd1336b.chunk.js"
        ).subscribe(() => {
            console.log(`script loaded`);
        });
        Vue.createApp(ShoppingCartVue).mount('#vue-application');
    }
    ... existing class code
}
```

Here, we start by declaring a variable named `Vue` of type `any`, and a variable named `ShoppingCartVue` of type `any`. Note that we will only use the `Vue` variable and the `ShoppingCartVue` variable once within this file, so for the sake of simplicity, we have decided to use the type `any` in this particular instance.

At the end of the `ngOnInit` function, just after we inject the React JavaScript files, we make a call to the `createApp` function of the `Vue` instance. Note that this is slightly different from the script that was generated in the `demo.html` file. It actually more closely resembles the `/shopping-cart/src/main.ts` version of creating a Vue application, as follows:

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#vue-application');
```

Here, we have the TypeScript-compatible version of creating a Vue application, and mounting it into a DOM element. Note that the `createApp` function is available when imported as a module, but is also available on the global `Vue` instance, which is what we have used in our `ngOnInit` function.

We can now include the required Vue scripts directly into our Angular `index.html` file as follows:

```
<!doctype html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>AngularApp</title>
    <base href="/">
    ... existing Angular scripts
    <script src="/assets/MicroEventBus.js"></script>

    ... existing React scripts

    <script
        src="https://cdnjs.cloudflare.com/ajax/libs/
            mdb-ui-kit/3.0.0/mdb.min.js">
    </script>
    <link
        href="https://cdnjs.cloudflare.com/ajax/libs/
            mdb-ui-kit/3.0.0/mdb.min.css"
        rel="stylesheet">
```

```
    <script src="https://unpkg.com/vue@3"></script>
    <script src="/assets/vue/ShoppingCartVue.umd.js"></script>

</head>

<body>
    <app-root></app-root>
</body>

</html>
```

Here, we have added two `<script>` tags to load firstly the Vue runtime itself, as in `https://unpkg.com/vue@3`, and then the `ShoppingCartVue.umd.js` files. Note that we are loading the `vue@3` script, and not simply the `vue` script, to ensure that we load Vue version 3.0 and not version 2.0. We have also loaded the `mdb.min.js` script and the `mdb.min.css` resource in order to provide the Material Design libraries that we have used in our Vue application.

Note that there is a marked difference between how we are rendering our Vue application and how we are rendering our React application. For React, we need to inject scripts into the DOM when we are ready to render the application, and for Vue, we just call the `createApp` script. The Vue method seems much simpler.

The reason for this is that React, at the time of writing, does not support creating libraries in the same way as Vue. Remember that we bundle the entirety of our Vue application into a single file named `ShoppingCartVue.umd.js`, which exposes a top-level class named `ShoppingCartVue`. We can then call the `Vue.createApp` function with this top-level class name, and mount it into a `<div>` element of our choosing. React, however, does not support this, which means that we need to use a mix of referenced JavaScript files in our `index.html` file, combined with the injection of JavaScript files into the DOM when we wish to render the application.

Angular domain events

The only remaining thing to do in our Angular application is to listen to any domain events that are raised from our Vue front-end. Again, we will need to use the declaration files for our micro-event-bus library, by importing them through npm as follows:

```
npm link micro-event-bus
```

Here, we are importing the type definitions for our `'micro-event-bus'` library into our Angular application. We can now listen to events by updating our `angular-app/src/app/app.component.ts` file as follows:

```
import { MicroEventBus } from 'micro-event-bus';
export let microEventBus: MicroEventBus =
    (window as any).microEventBus;

export class AppComponent {
    ... existing class member variables

    constructor(
        broadCastService: AngularBroadcastService,
        private scriptLoaderService: ScriptLoaderService,
        private cd: ChangeDetectorRef
    ) {
        _.bindAll(this, [
            "onLoginClicked",
            "onLoginEvent",
            "toggleCheckoutOnly",
            "toggleContinueShopping",
            "togglePlaceOrder"
        ]);
        ... internal broadcast event listeners

        microEventBus.on("checking-out")
            .subscribe(this.toggleCheckoutOnly);

        microEventBus.on("continue-shopping")
            .subscribe(this.toggleContinueShopping);

        microEventBus.on("place-order")
            .subscribe(this.togglePlaceOrder);

    }

    ... existing functions

    toggleCheckoutOnly() {
        this.shoppingCartStyles = "split";
```

```
            this.hideProducts = true;
            this.cd.detectChanges();
        }

        toggleContinueShopping() {
            this.shoppingCartStyles = "split right";
            this.hideProducts = false;
            this.cd.detectChanges();

        }
        togglePlaceOrder() {
            this.hideShoppingCartRegion = true;
            this.hideProducts = true;
            this.hideOrderPlaced = false;
            this.cd.detectChanges();
        }
    }
```

Here, we have made a few changes to our `AppComponent` class. Firstly, we have created a global variable named `microEventBus`, as we have in both our React and Vue applications. Secondly, we now have a dependency on an Angular internal service named `ChangeDetectorRef`, which we will discuss next. We have then subscribed to each of the domain events that are raised by our Vue front-end. These are the `"checking-out"`, `"continue-shopping"`, and `"place-order"` events. Each of these events will call a member function on our class.

So why do we need the `ChangeDetectorRef` service, then? If we examine the `toggleCheckoutOnly` function, we will see that we are updating the internal member property named `shoppingCartStyles`, and are then setting the `hideProducts` variable to `true`. These two variables will control what we see on our screen, as we will see in some screenshots in a little while. The final step of this function calls the `detectChanges` function on our instance of the `ChangeDetectorRef` service.

Bear in mind that when an event is generated from our Vue front-end, it is raising a Domain Event, and that event is picked up by our listener on the global instance of `microEventBus`. This event, and even the call to the `toggleCheckoutOnly` function, is essentially happening outside of our Angular application. Even if the call to this function is bound to the correct instance of `this`, which is the Angular class instance, the Angular runtime will not automatically detect this change, and will therefore not update our screen.

In order to always trigger the Angular change detection routines when an external event is received, we can call the `detectChanges` function, which will force Angular to run a change detection routine. When it does, it will realize that the `hideProducts` member variable and the `shoppingCartStyles` variables have changed, and update our screen.

Our updates to our Angular application are now complete, and we have a working micro front-end application. Let's now see how this application responds to our domain events.

Our micro front-end application

The following set of screenshots show how our Angular application is reacting to these domain events that are raised by our Vue application.

Firstly, once we have logged in with a valid username, the Vue application will load our current shopping cart as follows:

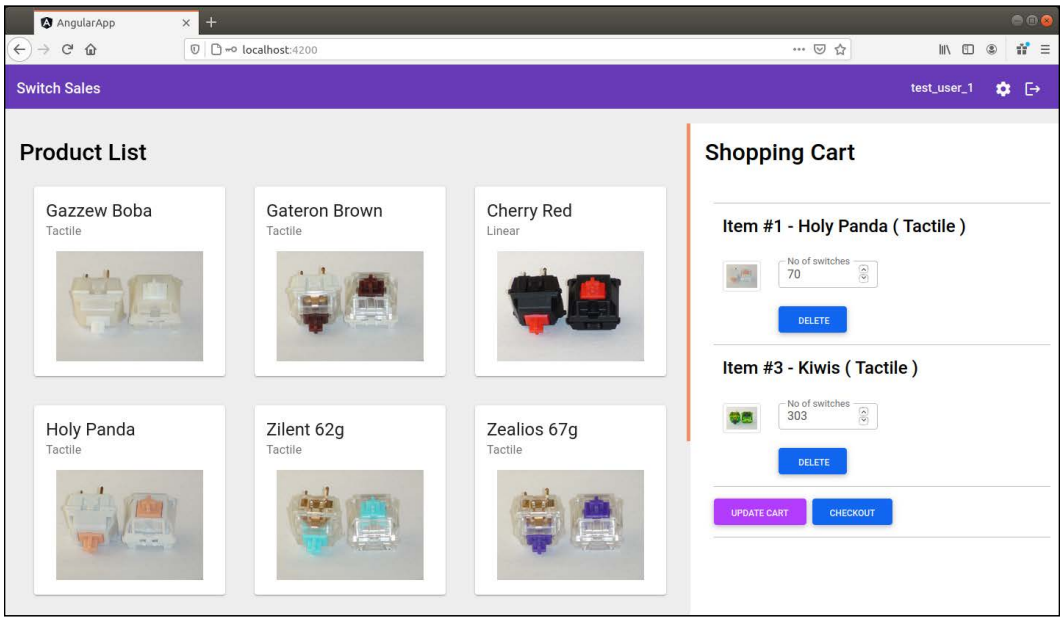


Figure 16.8: Vue front-end showing the loaded shopping cart on the right of the screen

Here, we can see that once we have logged in, and the `'user-logged-in'` event is received by the Vue front-end, it will trigger a call to the API, and load the current user's shopping cart within the Vue front-end.

If we now click on the **CHECKOUT** button on our **Shopping Cart** panel, the screen will change to show our invoice, as follows:

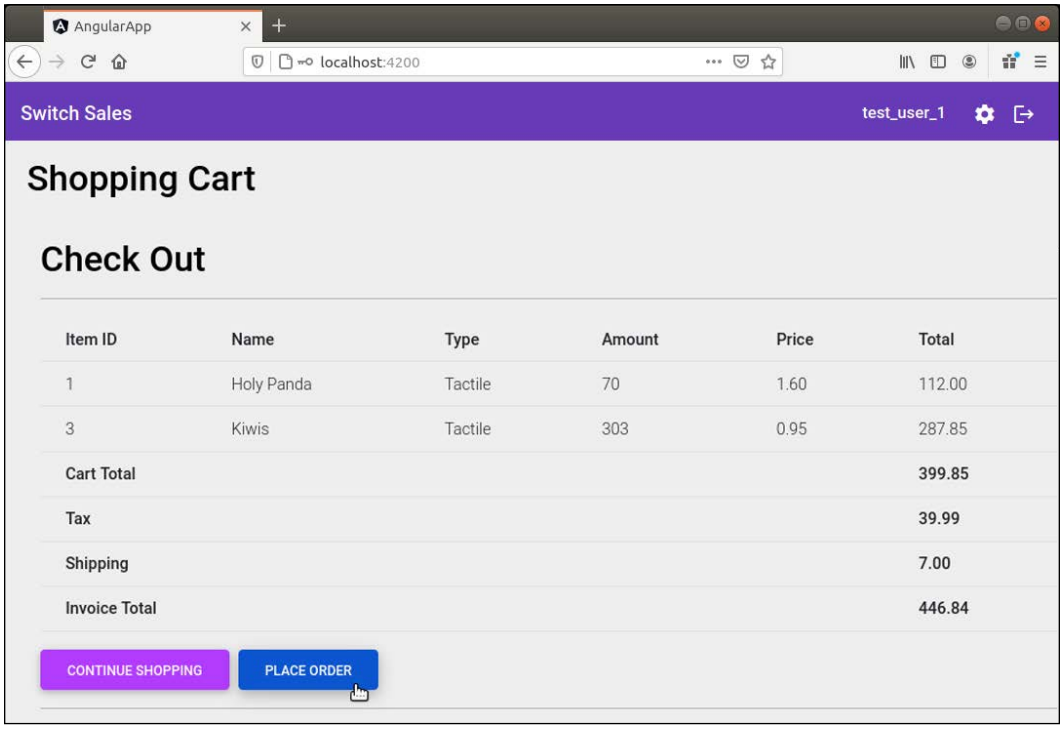


Figure 16.9: Vue front-end showing checkout screen, and Angular hiding other panels

Here, we can see that the Angular application has responded to the `'checking-out'` domain event, and has hidden the React product list front-end. Our Vue application has also switched to the Checkout view. If we click on the **CONTINUE SHOPPING** button, our Angular app will revert these changes, and we will be presented with the product list and shopping cart panels side by side again, as seen in the previous screenshot.

Clicking on the **PLACE ORDER** button will cause our Vue front-end to raise the `'place-order'` event, which will be received by the Angular application, and show the **Order Placed** screen, as shown in the following screenshot:

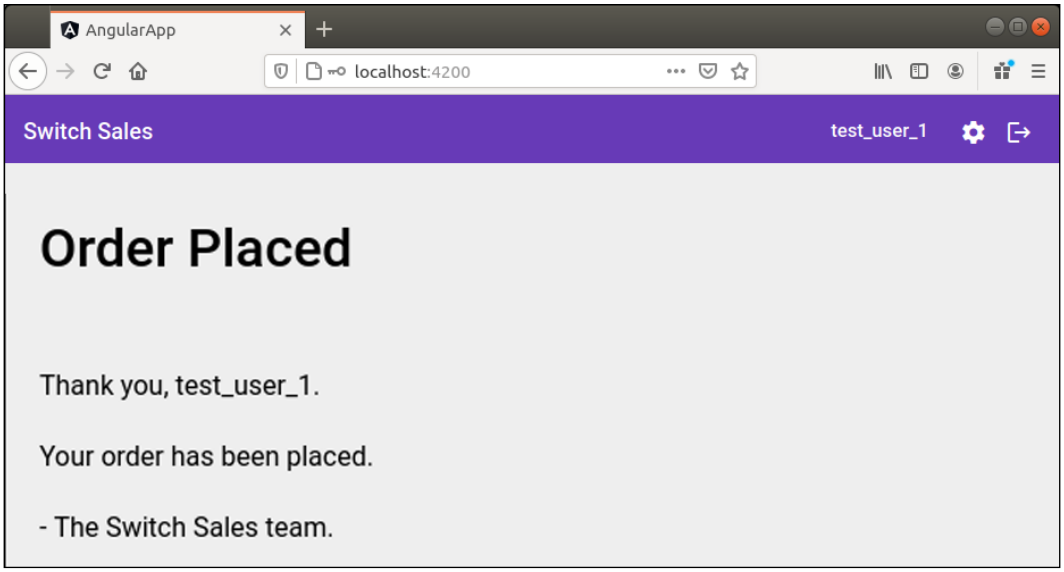


Figure 16.10: Angular application showing the Order Placed screen, after the domain event from the Vue front-end

Here, we have an **Order Placed** screen, which is not terribly exciting. It is, however, rendered by our Angular application, showing that we can control the display of our micro front-ends from Angular in a very fine-grained manner. Our screens have shown both the React and Vue front-ends side by side in a left and right screen configuration, or just the Vue front-end by itself, or just an Angular screen by itself.

Our discussion on micro-front ends is now complete. We have accomplished our goal of using a JavaScript technique to inject a React front-end and a Vue front-end into an Angular controlling application to provide a seamless experience to the user.

Please note that the source code samples provided with this chapter are a little more fleshed out than what we have discussed here. The sample code has a few extra features that integrate more completely with our API, and allow the registration of new users and the synchronization of a user's shopping cart with the back-end database.

Thoughts on micro front-ends

The work that we have done to implement our micro front-end was a combination of four distinct pieces of unrelated code, put together in a single application. We started with an implementation of an Event Bus, which was attached to a global variable through which all front-ends could communicate, our `MicroEventBus`. The packaging and use of this `MicroEventBus` was rather seamless, and easy to integrate with all of our front-ends.

The real benefit of using TypeScript in this implementation was due to the declaration files that we generated, which accomplished a few key things. Firstly, it ensured that we are only able to raise events that are specified by the Event Bus itself. This gives us a central repository for all events that are used within our micro front-end application. Secondly, it defined the data elements that each event carries. Some events have no data associated with them, and some have rather complex data structures. By using declaration files, we are able to ensure that both the producer of an event and the subscriber to the event have a clear understanding of what data is associated with each event.

One of the key goals of our micro front-end application was that each front-end could operate independently of any other front-end, and its only integration with the outside world was through Domain Events. We proved this with each of our applications through the use of some test events that were raised with a few lines of JavaScript in the `index.html` files. We were able to simulate how each of our front-ends reacted to these events, and also made sure that they generated events with the correct data structures.

We used our Angular application as the host application, within which we embedded our React application and our Vue application. This technique provided us with a mechanism of coordinating the rendering of each application at the right time, and allowed us to hide them when they were not needed. It also allowed us to build an application that felt as if it was a single application, instead of three distinct ones.

We explored two different techniques of rendering our applications into elements of the DOM. Rendering our React application required the careful injection of JavaScript files into the DOM at the right time, and rendering our Vue application was a matter of calling a Vue function. Unfortunately, there is currently no standard mechanism to do this that all popular frameworks adhere to. Over time, as the popularity of micro front-ends grows, we will hopefully see the adoption of standards for this, such as the existing standard for web components.

This chapter has been an exploration of what can be done with micro front-ends. As described at the start, the use of micro front-ends means that we can break up a large monolithic application into separate, discreet areas of functionality. This approach, in turn, can empower teams that are responsible for an area of functionality to be completely autonomous and independent. Each team can define their own release cycles and release cadence, and can be in charge of the direction for their own area of the product. Each team can contribute to the definition of domain events, or can request another team to publish a domain event for things that they are interested in.

The concepts of Domain Events, Event Buses, and indeed micro services have been explored and adopted by the general programming community as a means of building modular solutions. The extension of these ideas into micro front-ends is just the beginning of what solutions may look like when we build them in the future.

Summary

This book has been an exploration of all things TypeScript, starting with basic language constructs, and moving into more and more advanced features. In the first part of this book, we covered the language itself, integration with JavaScript, unit testing, and data transformation. In the second part of this book, we explored a number of popular JavaScript frameworks, including Angular, React, Vue, Express, and AWS Lambda, and finally integrated some of these into a micro front-end application. Hopefully you have enjoyed this journey, and have learned a little bit about how TypeScript can be used to successfully develop, test, and integrate a wide range of JavaScript applications.

Share your experience

Thank you for taking the time to read this book. If you enjoyed this book, help others to find it. Leave a review at `https://www.amazon.com/dp/1800564732`.

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

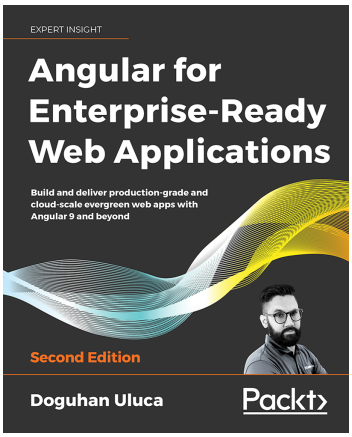
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

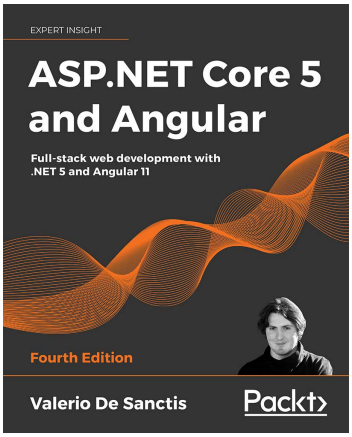


Angular for Enterprise-Ready Web Applications

Doguhan Uluca

ISBN: 978-1-83864-880-0

- Adopt a minimalist, value-first approach to delivering web apps
- Master Angular development fundamentals, RxJS, CLI tools, GitHub, and Docker
- Discover the flux pattern and NgRx
- Implement a RESTful APIs using Node.js, Express.js, and MongoDB
- Create secure and efficient web apps for any cloud provider or your own servers
- Deploy your app on highly available cloud infrastructure using DevOps, CircleCI, and AWS



ASP.NET Core 5 and Angular

Valerio De Sanctis

ISBN: 978-1-80056-033-8

- Implement a Web API interface with ASP.NET Core and consume it with Angular using RxJS Observables
- Set up an SQL database server using a local instance or a cloud data store
- Perform C# and TypeScript debugging using Visual Studio 2019
- Create TDD and BDD unit tests using xUnit, Jasmine, and Karma
- Perform DBMS structured logging using third-party providers such as SeriLog
- Deploy web apps to Windows and Linux web servers, or Azure App Service, using IIS, Kestrel, and nginx

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W