APIs and Data Access

In real-world scenarios, whether it's a mobile app, desktop, service, or web apps, they heavily rely on Application Programming Interfaces (APIs) to interact with systems to submit or fetch data. APIs typically act as a gateway between client applications and a database to perform any data operations between systems. Often, APIs provide instructions and a specific format to clients on how to interact with the system to perform data transactions. Thus, APIs and data access work together to achieve two main goals: serving and taking data.

Here is the list of the main topics that we'll go through in the module:

- Understanding what ORM and Entity Framework Core are
- Reviewing the different design workflows supported by EF Core
- Learning database-first development
- Learning code-first development and migrations
- Learning the basics of LINQ to query data against conceptual models
- Reviewing what the ASP.NET Core API is
- Building Web APIs that implement the most commonly used HTTP methods for serving data
- Testing APIs with Postman

In this module, we are going to learn about the different approaches to working with a real database in Entity Framework (EF) Core. We will take a look at how to use EF Core with an existing database, as well as implementing APIs that talk to a real database using the EF Core code-first approach. We will look into ASP.NET Core Web APIs in concert with Entity Framework Core to perform data operations in an SQL Server database. We will also learn how to implement the most commonly used HTTP methods (verbs) for exposing some API endpoints.

It is important to understand that ASP.NET Core is not only limited to Entity Framework Core and SQL Server. You can always use whatever data access frameworks you prefer.

For example, you can always use Dapper, NHibernate, or even use the good old plain ADO.NET as your data access mechanism. You can also use MySQL or Postgres as your database provider if you'd like.

Technical requirements

This module uses Visual Studio 2019 to demonstrate building different applications. A basic understanding of databases, ASP.NET Core, and C# in general, is required because we're not going to cover their fundamentals in this module.

Understanding Entity Framework Core

In the software engineering world, most applications require a database to store data. So, we all need code to read/write the data stored in a database. Creating and maintaining code for a database is tedious work and it is a real challenge for us as developers. That's where Object Relational Mappers (ORMs) like Entity Framework come into play.

Entity Framework Core is an ORM and a data access technology that enables C# developers to interact with a database without having to manually write SQL scripts.

ORMs like EF Core help you build data-driven applications quickly by working through .NET objects instead of interacting directly with the database schema. These .NET objects are simply classes, which are typically referred to as Entities. With EF Core, C# developers can take advantage of their existing skills and leverage the power of Language Integrated Query (LINQ) to manipulate the dataset against the conceptual Entity Models, otherwise simply referred to as Models. We'll be using the term models from here on as shown in figure below:

The preceding diagram depicts the process of interacting with a database using EF Core. In the traditional ADO.NET, you would typically write SQL queries by hand to perform database operations. While performance varies according to how your queries are written, still, the ADO.NET way brings a performance advantage over ORMs as you can inject your SQL queries directly into your code and run it against the database. However, this leads to your code becoming hard to maintain because any SQL query changes will result in changing your application code as well; with the exception of using stored procedures. Also, debugging your code can be painful as you will be dealing with a plain string to write your SQL queries and any typos or syntax errors can be easily overlooked.

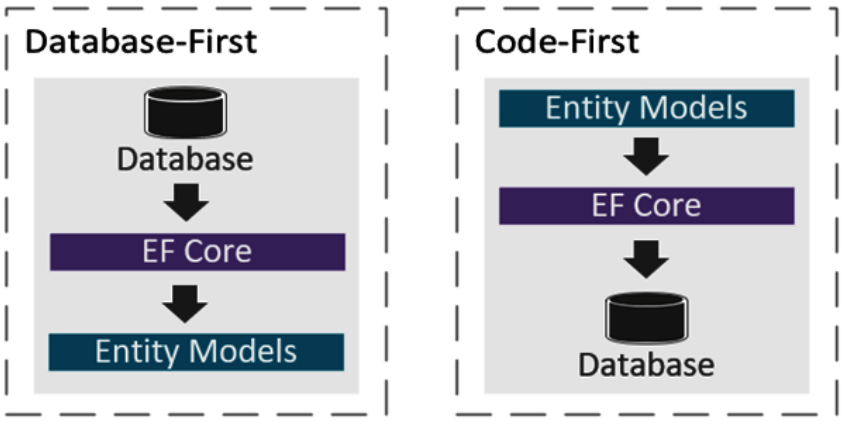
With EF Core, you don't have to worry about writing SQL scripts yourself. Instead, you will use LINQ to query strongly-typed objects and let the framework handle the rest, such as generating and executing SQL queries.

Keep in mind that EF Core is not limited to SQL Server databases. The framework supports various database providers that you can integrate with your application, such as Postgres, MySQL, SQLite, Cosmos, and many others.

Reviewing EF Core design workflows

There are two main design workflows supported by EF Core: the database-first approach and the code-first approach.

The following figure depicts the difference between the two design workflows:



In the preceding figure, we can see that the database-first workflow begins with an existing database and EF Core will generate models based on the database schema. The code-first workflow, on the other hand, begins with writing models and EF Core will generate the corresponding database schema via EF migrations. Migration is a process that keeps your models and database schema in sync without losing existing data.

The following table outlines recommendations for which design workflow to consider when building an application:

	Database-First	Code-First
Code Generation	You create table definitions in the database and import them using a script to scaffold everything for you.	You write classes that represent the entity models that you want to keep track of as well as defining relationships between other models.
Syncing Changes	You write SQL migration scripts for the changes and run them against the database. You then manually configure your application code to sync models with your database changes.	Use the EF migration tool when you need to sync model changes to your database. This lets you manage database schema changes without you having to write any SQL manually.
Type of project best suited for	You may use database-first for projects where you have full control over databases or when a database is developed and owned by another team.	You may use the code-first approach if you're building applications from scratch where business requirements change constantly. This enables you to evolve your models as you develop.

It's very important to understand the differences between the design workflows so you know when to apply them to your projects.

Now that you've learned the difference between the two design workflows, let's move on to the next section and learn how to implement each approach with hands-on coding exercises.

Learning database-first development

In this section, we will build a .NET Core console application to explore the database-first approach and see how entity models are created from an existing database (reverse engineering).

Creating a .NET Core console app

To create a new .NET Core console app, follow these steps:

1. Open Visual Studio 2019 and select **Create a new project**.
2. Select the **Console App (.NET Core) project** template.
3. Click **Next**. On the next screen, name the project EFCore_DatabaseFirst.
4. Click **Create** to let Visual Studio generate the default files for you.

Now, we are going to add the required Entity Framework Core packages in our application for us to work with our existing database using the database-first approach.

Integrating Entity Framework Core

The Entity Framework Core feature was implemented as a separate NuGet package to allow developers to easily integrate features that the application needs.

As you may have already learned from modules Razor View Engine; Getting Started with Blazor; and Exploring Blazor Web Frameworks, there are many ways to add NuGet package dependencies in Visual Studio; you could either use the Package Manager Console (PMC) or NuGet Package Manager (NPM). In this exercise, we are going to use the console.

By default, the PMC window is enabled and you can find it in the bottom-left portion of Visual Studio.

If, for some reason, you can't find the PMC window, you can manually navigate to it by going to the **Visual Studio** menu under **Tools > NuGet Package Manager > Package Manager Console**.

Now, let's install a few NuGet packages by running the following commands in the console individually:

```
PM> Install-Package Microsoft.EntityFrameworkCore.Tools

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design -Pre
```

The commands in the preceding code will install the NuGet packages as dependencies in your application. The -Pre command instructs to install the latest preview version of Entity Framework Core packages. In this case, the current version as of this time of writing is 5.0.0 for the SQL Server and Tools packages, and 2.0.0-preview1-final for the SqlServer.Design package.

Now that we have installed the necessary tools and dependencies for us to work with an existing database, let's move on to the next step.

Creating a database

To simulate working with an existing database, we will need to create a database from scratch. In this example, we will just be creating a single table that houses some simple columns for simplicity. You can use SQL Server Express if you have it installed or use the local database built into Visual Studio.

To create a new database in Visual Studio, follow these simple steps:

1. Go to **View > SQL Server Object Explorer**.
2. Drill down to **SQL Server > (localdb)\MSSQLLocalDB**.
3. Right-click on the Databases folder.
4. Click **Add New Database**.
5. Name it DbFirstDemo and click **OK**.
6. Right-click on the DbFirstDemo database and then select **New Query**.
7. Copy the following SQL script:

```
CREATE TABLE [dbo].[Person]
(
  [Id] INT NOT NULL PRIMARY KEY IDENTITY(1,1),
  [FirstName] NVARCHAR(30) NOT NULL,
  [LastName] NVARCHAR(30) NOT NULL,
  [DateOfBirth] DATETIME NOT NULL
)
```

8. Run the script and it should create a new table called Person in your local database.

Now that we have a database, let's move on to the next section and create .NET class objects for us to work with the data using EF Core.

Generating models from an existing database

As of the time of writing, there are two ways to generate models from an existing database.

You can either use PMC or .NET Core Command-Line Interface (CLI) commands. Let's see how we can do this in the following section.

Using the Scaffold-DbContext command

The first thing that you need to do is to grab the ConnectionString value for you to connect to the database. You can get this value from the Properties window of the DbFirstDemo database in Visual Studio.

Now navigate back to the PMC and run the following command to create the corresponding Models from the existing database:

```
PM> Scaffold-DbContext "INSERT THE VALUE OF CONNECTION STRING HERE"
Microsoft.EntityFrameworkCore.SqlServer -o Db
```

The Scaffold-DbContext command in the preceding code is part of the Microsoft.EntityFrameworkCore.Tools package, which is responsible for the reverse engineering process. This process will create a DbContext and Model classes based on the existing database.

We've passed in three main parameters in the Scaffold-DbContext command:

- **Connection string:** The first parameter is the connection string that instructs how to connect to the database.
- **Provider:** The database provider that will be used to execute the connection string against. In this case, we've used Microsoft.EntityFrameworkCore.SqlServer as the provider.

- **Output directory:** The -o option is shorthand for –OutputDir, which enables you to specify the location of the files to be generated. In this case, we've set it to Db.

Using the dotnet ef dbcontext scaffold command

The second option to generate Models from an existing database is using the EF Core tools via .NET Core CLI. In order to do this, we need to use the command-line prompt. In Visual Studio, you can go to **Tools > Command Line > Developer Command Prompt**. This process will launch a Command Prompt window at the folder where the solution file (.sln) is located. Since we need to execute the command at the level where the project file (.csproj) is located, then we need to move the directory one folder down. So, in Command Prompt, do the following:

```
cd EFCore_DatabaseFirst
```

The preceding command will set the current directory to where the project file is located.

Another approach is to navigate to the EFCore_DatabaseFirst folder outside Visual Studio and then press Shift + Right-click and select Open command window here or Open PowerShell window here. This process will directly open Command Prompt in the project file directory.

In Command Prompt, let's first install the EF Core CLI tools by running the following command:

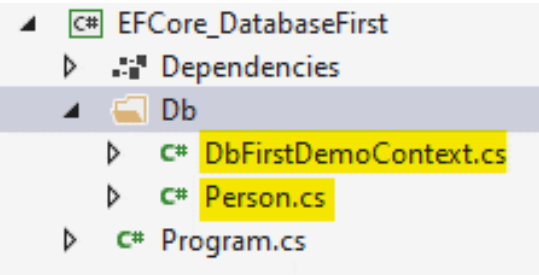
```
Dotnet tool install-–global dotnet-ef
```

The preceding code will install the EF Core tools globally on your machine. Now, run the following command:

```
dotnet ef dbcontext scaffold "INSERT THE VALUE OF CONNECTION STRING HERE" Microsoft.EntityFrameworkCore.SqlServer -o Db
```

The preceding code is quite similar to using the Scaffold-DbContext command, except we've used the dotnet ef dbcontext scaffold command, which is specific to CLI-based EF Core tools.

Both options will give you the same results and will create a DbContext and Model classes within the Db folder, as shown in figure below:



Take a moment to examine each file generated and see what code is generated.

When you open the DbFirstDemoContext.cs file, you can see that the class is declared as partial class and it derives from the DbContext class. DbContext is the main requirement in Entity Framework Core. In this example, the DbFirstDemoContext class represents the DbContext that manages the connection with the database and provides various capabilities such as building models, data mapping, change tracking, database connections, caching, transaction management, querying, and persisting data.

You'll also see the following code within the DbFirstDemoContext class:

```
    public virtual DbSet<Person> People { get; set; }
```

The preceding code represents an entity. Entities are defined as the type of DbSet that represents your model. EF Core requires an Entity so it can read, write, and migrate data to the database. To put it in simple terms, DbSet<Person> represents your database table called Person. Now, instead of you writing SQL script to perform database operations such as insert, update, fetch or delete, you will simply perform database operations against the DbSet called People and leverage the power of LINQ to manipulate data with strongly-typed code. This helps you, as a developer, boost productivity by programming against a conceptual application model with full IntelliSense support, instead of programming directly against a relational storage schema. Notice how EF automatically sets the DbSet property name to its plural form. It's just awesome!

The other thing that you'll see within the DbFirstDemoContext class is OnConfiguring(). This method configures the application to use Microsoft SQL Server as the provider using the UseSqlServer() extension method and passing the ConnectionString value. In the actual generated code, you will see that the value is being passed directly to the UseSqlServer() method.

> **Note**
>
> In real-world applications, you should avoid injecting the actual value directly and instead store your ConnectionString value in a key vault or secrets manager for security's sake.

Finally, you will see a method called OnModelCreating() within the DbFirstDemoContext class. The OnModelCreating() method configures a ModelBuilder for your Models. The method is defined from the DbContext class and marked as virtual, allowing us to override its default implementation. You'll use this method to configure Model relationships, data annotations, column mappings, data types, and validations. In this particular example, when EF Core generates the models, it applies the corresponding configuration that we have in our dbo.Person database table.

> **Note**
>
> Any changes you've made to the DbContext class and Entity models will be lost when running the database-first command again.

Now that we have a DbContext configured, let's move on to the next section and run some tests to perform some simple database operations.

Performing basic database operations

Since this is a console application, we are going to perform simple insert, update, select, and delete database operations in the Program.cs file for the simplicity of this exercise.

Let's start by inserting new data into the database.

Adding a record

Go ahead and add the following code within the Program class:

```
static readonly DbFirstDemoContext _dbContext = new DbFirstDemoContext();
static int GetRecordCount() {
  return _dbContext.People.ToList().Count;
}
static void AddRecord() {
  var person = new Person {
    FirstName = "Vjor", LastName = "Durano",
    DateOfBirth = Convert.ToDateTime("06 / 19 / 2020")
  };
  _dbContext.Add(person);
  _dbContext.SaveChanges();
}
```

The preceding code defines a static readonly instance of the DbFirstDemoContext class. We need the DbContext so that we can access the DbSet and perform database operations against it.

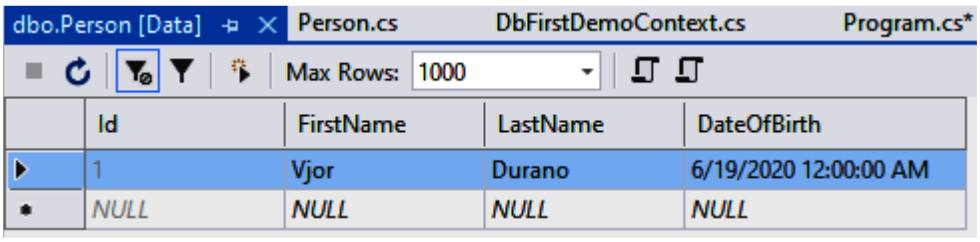
The GetRecordCount() method simply returns the number of record counts stored in the database. The AddRecord() method is responsible for inserting a new record into the database. In this example, we just defined some static values for the Person Model for simplicity. The _dbContext.Add() method takes a Model as the parameter. In this case, we've passed the person variable to it and then invoked the SaveChanges() method of the DbContext class. Any changes you've made to the DbContext won't be reflected in the underlying database – not unless you call the SaveChanges() method.

Now, what's left for us to do here is to call the methods in the preceding code. Go ahead and copy the following code in the Main method of the Program class:

```
static void Main(string[] args) {
  AddRecord();
  Console.WriteLine($"Record count: { GetRecordCount()}");
}
```

Running the preceding code will insert a new record into the database and output the value 1 as the record count.

You can verify that the record has been created in the database by going to the **SQL Server Object Explorer** pane in Visual Studio. Drill down to the dbo.Person table, right-click on it, and select **View Data**. It should show the newly added record in the database, as shown in figure below:



Cool! Now, let's continue and do some other database operations.

Updating a record

Let's perform a simple update to an existing record in the database. Append the following code within the Program class:

```
static void UpdateRecord(int id) {
  var person = _dbContext.People.Find(id);
  // removed null check validation for brevity
  person.FirstName = "Vynn Markus";
  person.DateOfBirth = Convert.ToDateTime("11 / 22 / 2016");
  _dbContext.Update(person);
  _dbContext.SaveChanges();
}
```

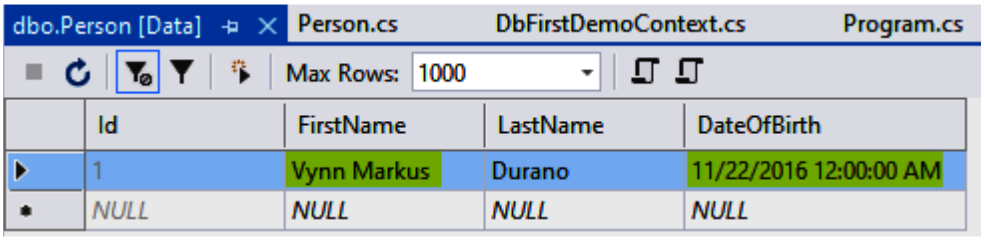
The preceding code takes an id as an argument. It then queries the database using the Find() method of the DbContext. We then check whether the id that we passed in has an associated record in the database. If the Find() method returns null, we simply do nothing and return directly to the caller. Otherwise, if the given id existed in the database, we perform a database update. In this case, we've simply replaced the value of the FirstName and DateOfBirth properties.

Now, let's call the UpdateRecord() method in the Main method of the Program class as in the following:

```
static void Main(string[] args) {
  UpdateRecord(1);
}
```

In the preceding code, we manually pass the value of 1 as the id. That value represents an existing record in the database when we performed insertion in the previous section.

Running the code should update the values for the FirstName and DateOfBirth columns as shown in figure below:



Great! Now, let's continue with other database operations.

Querying a record

Go ahead and copy the following code within the Program class:

```
static Person GetRecord(int id) {
  return _dbContext.People.SingleOrDefault(p => p.Id.Equals(id));
}
```

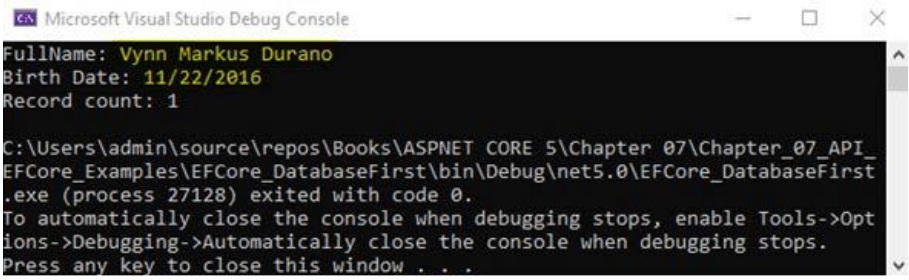
The preceding code also takes an id as an argument so it can identify which record to fetch. What it does is it queries the database using the LINQ SingleOrDefault() extension method and uses a lambda expression to perform value comparisons with the given id value. If the id matches with a record from the database, then we return a Person object to the caller.

Now, let's invoke the GetRecord() method by copying the following code within the Main method of the Program class:

```
static void Main(string[] args) {
  var p = GetRecord(1);
  if (p != null) {
    Console.WriteLine($"FullName: { p.FirstName}{ p.LastName}");
    Console.WriteLine($"Birth Date: {p.DateOfBirth.ToShortDateString()}");
  }
}
```

In the preceding code, we've manually passed the value of 1 again as the parameter to the GetRecord() method. This is to ensure that we are getting a record back since we only have one record in the database at the moment. If you pass an id value that doesn't exist in the database, then the GetRecord() method will return null. That's why you see we have implemented a basic validation to check against null so that the application won't break. We then print the values to the console window.

Running the code will result in the following as shown in figure below:



It's that simple! There are many things that you can do with LINQ to query data, especially complex data. In this example, we are just doing basic querying with a single database for you to better understand how it works.

Deleting a record

Now, let's see how we can easily perform deletion with EF Core. Copy the following code within the Program class:

```
static void DeleteRecord(int id) {
  var person = _dbContext.People.Find(id);
  // removed null check validation for brevity
  _dbContext.Remove(person);
  _dbContext.SaveChanges();
}
```

Just like in the database update operation, the preceding code checks for the existing record first using the Find() method. If the record exists, we invoke the Remove() method of the DbContext and save the changes to reflect the deletion in the database.

Now, copy the following code in the Main method of the Program class:

```
static void Main(string[] args) {
  DeleteRecord(1);
  Console.WriteLine($"Record count: { GetRecordCount()}");
}
```

Running the code will delete the record in the database with an id value equal to 1. The call to the GetRecordCount() method will now return 0 as we don't have any other records in the database.

Now that you've learned about implementing a database-first approach with EF Core, let's move on to the next section and explore the EF Core code-first approach in concert with ASP.NET Core Web API.

Learning code-first development

In this section, we are going to explore EF Core code-first development by building a simple ASP.NET Core Web API application to perform basic database operations.

Before we get our hands dirty with coding, let's first review what ASP.NET Core Web API is.

Reviewing ASP.NET Core Web API

There are many ways to enable various systems to access data from one application to another. A few examples of communications are HTTP-based APIs, web services, WCF servers, event-based communication, message queues, and many others. Nowadays, HTTP-based APIs are the most commonly used means of communication between applications. There are a few ways to use HTTP as the transport protocol for building APIs: OpenAPI, **R**emote **P**rocedure **C**all (gRPC), and **RE**presentational **S**tate **T**ransfer (REST).

ASP.NET Core Web API is an HTTP-based framework for building RESTful APIs that allow other applications on different platforms to consume and pass data over HTTP. In the ASP.NET Core application, Web APIs are very similar to MVC except that they return data as the response to the client instead of a View. The term client in the context of APIs refers to either a web app, mobile app, desktop app, another Web API, or any other type of service that supports the HTTP protocol.

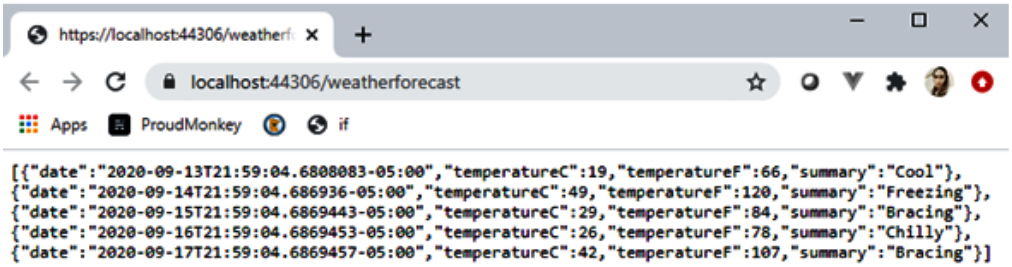
Creating a Web API project

Now that you know what Web API is all about, let's see how we can build a simple, yet realistic RESTFul API application that serves data from a real database. Keep in mind though that we're not going to cover all the constraints and guidelines of REST as it would be a huge task to cover them all in a single module. Instead, we will just be covering some of the basic guidelines for you to be able to get a good grasp and a head-start with building APIs in ASP.NET Core.

To create a new Web API project, fire up Visual Studio 2019 and follow the steps given here:

1. Select the **Create a new project** option.
2. On the next screen, select **ASP.NET Core Web Application** and then click **Next**.
3. On the **Configure your new project** dialog, set the project name to EFCore_CodeFirst and choose the location that you want the project to be created at.
4. Click Create. On the next screen, select the **API** project template and click Create.

You should see the default files generated by Visual Studio for the Web API template. The default generated template includes WeatherForecastController to simulate a simple HTTP GET request using static data. To ensure that the project works, run the application by pressing the **Ctrl + F5** keys and you should be presented with the following output when everything is fine as shown in figure below:



At this point, we can conclude that the default project is working properly. Now let's move on to the next step and set up the data access part of the application.

Configuring data access

The first thing that we need to do here is to integrate the required NuGet package dependencies for the application. Just like what we did in the Integrating Entity Framework Core section, install the following NuGet packages:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.SqlServer

At the minimum, we need to add these dependencies so we can work with EF Core, use SQL Server as the database provider, and finally, use EF Core commands to create migrations and database synchronization.

After successfully installing the required NuGet package dependencies, let's jump to the next step and create our Models.

Creating entity models

As we learned in the code-first workflow, we are going to begin creating the conceptual Models that represent entities.

Create a new folder called Db at the root of the application and create a sub-folder called Models. To make this exercise more fun, we are going to define a few Models that contain relationships. We are going to be building an API where music players can submit their information along with the musical instruments that they play. To achieve this requirement, we are going to need a few models to hold different information.

Now, create the following classes within the Models folder:

- InstrumentType.cs
- PlayerInstrument.cs
- Player.cs

The following is the class definition of the InstrumentType.cs file:

```
public class InstrumentType {
  public int InstrumentTypeId { get; set; }
  public string Name { get; set; }
}
```

The following is the class definition of the PlayerInstrument.cs file:

```
public class PlayerInstrument {
  public int PlayerInstrumentId { get; set; }
  public int PlayerId { get; set; }
  public int InstrumentTypeId { get; set; }
  public string ModelName { get; set; }
  public string Level { get; set; }
}
```

The following is the class definition of the Player.cs file:

```
public class Player {
  public int PlayerId { get; set; }
  public string NickName { get; set; }
  public List<PlayerInstrument> Instruments { get; set; }
  public DateTime JoinedDate { get; set; }
}
```

The classes in the preceding code are nothing but plain classes that house some properties that are required for us to build some API endpoints. These classes represent our Models that we are going to migrate as database tables later on. Keep in mind that, for simplicity's sake, we are using an int type as identifiers in this example. In a real application, you may want to consider using the Globally Unique Identifier (GUID) type instead so that it can't be easily guessed when you expose these identifiers in your API endpoints.

Seeding data

Next, we'll create an extension method to demonstrate preloading data into our lookup table called InstrumentType. Go ahead and create a new class called DbSeeder within the Db folder, then copy the following code:

```
public static class DbSeeder {
  public static void Seed(this ModelBuilder modelBuilder) {
    modelBuilder.Entity<InstrumentType>().HasData(
    new InstrumentType {
      InstrumentTypeId = 1, Name =    "Acoustic Guitar" },
    new InstrumentType {
      InstrumentTypeId = 2, Name =    "Electric Guitar" },
    new InstrumentType {
      InstrumentTypeId = 3, Name =    "Drums" },
    new InstrumentType {
      InstrumentTypeId = 4, Name =    "Bass" },
    new InstrumentType {
      InstrumentTypeId = 5, Name =    "Keyboard" }
    );
  }
}
```

The preceding code initializes some data for the InstrumentType Model using the HasData() method of the EntityTypeBuilder<T> object. We will invoke the Seed() extension method in the next step when we configure our DbContext.

Defining a DbContext

Create a new class called CodeFirstDemoContext.cs and copy the following code:

```
public class CodeFirstDemoContext : DbContext {
  public CodeFirstDemoContext(DbContextOptions<CodeFirstDemoContext> options)
    : base(options) { }
  public DbSet<Player> Players { get; set; }
  public DbSet<PlayerInstrument> PlayerInstruments {
    get;
    set;
  }
  public DbSet<InstrumentType> InstrumentTypes { get; set; }
  protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Player>()
      .HasMany(p => p.Instruments)
      .WithOne();
    modelBuilder.Seed();
  }
}
```

The preceding code defines a few DbSet entities for the Player, PlayerInstrument, and InstrumentType Models. In the OnModelCreating() method, we've configured a one-to-many relationship between the Player and PlayerInstrument Models. The HasMany() method instructs the framework that the Player

entity can contain one or more PlayerInstrument entries. The call to the modelBuilder.Seed() method will prepopulate the InstrumentType table in the database with data at the time it is created.

Keep in mind that the DbContext features extension methods to do database CRUD operations and already manages transactions. So, there's really no need for you to create a generic repository and unit of work pattern, not unless it's really needed to add more value.

Registering the DbContext as a service

Within the Db folder, go ahead and create a new class called DbServiceExtension.cs and copy the following code:

```
public static class DbServiceExtension {
  public static void AddDatabaseService(this
    IServiceCollection services, string connectionString)=> services.
    AddDbContext<CodeFirstDemoContext>(options => options.
    UseSqlServer(connectionString));
}
```

The preceding code defines a static method called AddDatabaseService(), which is responsible for registering the DbContext that uses the SQL Server database provider in the DI container.

Now that we have our DbContext, let's move on to the next step and wire up the remaining pieces to make the database migration work.

Setting the database ConnectionString

In this exercise, we will also use a local database built into Visual Studio. However, this time, we won't be injecting the ConnectionString value into our code. Instead, we'll use a configuration file to store it. Now, open the appsettings.json file and append the following configuration:

```
"ConnectionStrings": {
"CodeFirstDemoDb": "DataSource = (localdb)\\MSSQLLocalDB; Initial Catalog =
CodeFirstDemo; Integrated Security = True; Connect Timeout = 30; Encrypt = False;
TrustServerCertificate = False; ApplicationIntent = ReadWrite; MultiSubnetFailover =
False"
}
```

The preceding code uses the same ConnectionStrings value that we used in the previous example about **learning database-first development**, except that we are changing the Initial Catalog value to CodeFirstDemo. This value will automatically become the database name once the migration has been executed in SQL Server.

> **Note**
>
> As a reminder, always consider storing the ConnectionStrings value and other sensitive data in a key vault or secrets manager when developing a real application. This is to prevent exposing sensitive information to malicious users when hosting your source code in a version control repository.

Modifying the Startup class

Let's update the ConfigureServices() method of the Startup class to the following code:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddDatabaseService(Configuration.GetConnectionString("CodeFirstDemoDb"));
  //Removed other code for brevity
}
```

In the preceding code, we've invoked the AddDatabaseService() extension method that we created earlier. Registering the DbContext as a service in the DI container enables us to reference an instance of this service in any class within the application via DI.

Managing database migrations

In real-world development scenarios, business requirements often change and so do your Models. In cases like this, the migration features in EF Core come in handy to keep your conceptual Model in sync with the database.

To recap, migrations in EF Core are managed by executing commands either using the PMC or via .NET Core CLI. In this section, we are going to learn how we can perform the commands to do migrations.

First, let's start with creating a migration.

Creating a migration

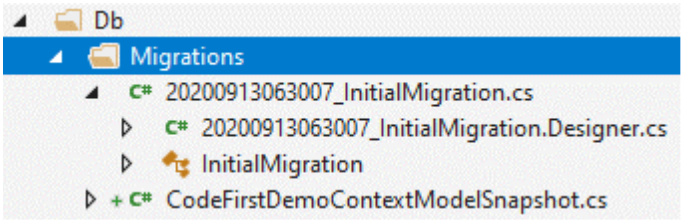
Open the PMC in Visual Studio and run the following command:

```
PM> Add-Migration InitialMigration -o Db/Migrations
```

Alternatively, you can also run the following command using the .NET Core CLI:

```
dotnet ef migrations add InitialMigration -o Db/Migrations
```

Both migration commands should generate the migration files under the Db/Migrations folder, as shown in figure below:



EF Core will use the generated migration files in the preceding screenshot to apply migrations in the database. The 20200913063007_InitialMigration.cs file contains Up() and Down() methods that accept MigrationBuilder as an argument.The Up() method gets executed when you apply Model changes to the database. The Down() method discards any changes and restores the database state based on the previous migration. The CodeFirstDemoContextModelSnapshot file contains a snapshot of the database every time you add a migration.

You may have noticed that the naming convention for the migration files is prefixed with a timestamp. This is because the framework will use these files in comparing the current state of the Models against the previous database snapshot when you create a new migration.

Now that we have the migration files, the next thing that we need to do is to apply the created migration to reflect the changes in the database.

Applying Migration

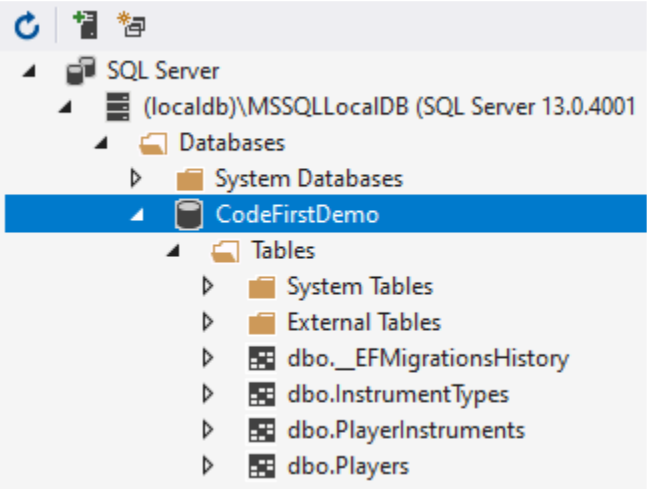
Navigate back to the PMC window and run the following command:

```
PM> Update-Database
```

The .NET Core CLI equivalent command is the following:

```
dotnet ef database update
```

The preceding commands will generate a database called CodeFirstDemo with the corresponding tables based on the Models along with a special migrations history table named _EFMigrationsHistory as shown in figure below:



The dbo._EFMigrationsHistory table stores the name of the migration file and EF Core version used to execute the migration. This table will be used by the framework to automatically apply changes based on the new migration. The dbo.InstrumentTypes table will also be preloaded with data.

At this point, you should now have the data access all set up and ready for use in the application.

Reviewing DTO classes

Before we deep dive into the implementation details. Let's first review what DTOs are, as we will be creating them later in this exercise.

Data Transfer Objects (DTOs) are classes that define a Model with sometimes predefined validation in place for HTTP responses and requests. You can think of DTOs as ViewModels in MVC where you only want to expose relevant data to the View. The basic idea of having DTOs is to decouple them from the actual Entity Model classes that are used by the data access layer to populate the data. This way, when a requirement changes or if your Entity Model properties are changed, they won't be affected and won't break your API. Your Entity Model classes should only be used for database related processes. Your DTOs should only be used for taking requests input and response output, and should only expose properties that you want your client to see.

Now, let's move on to the next step and create a few API endpoints for serving and consuming data.

Creating Web API endpoints

Most examples on the internet teach you how to create Web API endpoints by implementing the logic directly inside the Controllers for simplicity. For this exercise, we won't do that, instead, we will create APIs by applying some recommended guidelines and practices. This way, you will be able to use the techniques and apply them when building real-world applications.

For this exercise, we are going to cover the most commonly used **HTTP methods** (verbs) for implementing Web API endpoints, such as GET, POST, PUT, and DELETE.

Implementing an HTTP POST endpoint

Let's start off by implementing a POST API endpoint for adding a new record in the database.

Defining DTOs

First, go ahead and create a new folder called Dto at the root of the application. The way you want to structure your project files is based on preference and you are free to organize them however you want. For this demo, we wanted to have a clean separation of concerns so we can easily navigate and modify code without affecting other code. So, within the Dto folder, create a subfolder called PlayerInstruments and then create a new class called CreatePlayerInstrumentRequest with the following code:

```
public class CreatePlayerInstrumentRequest {
  public int InstrumentTypeId { get; set; }
  public string ModelName { get; set; }
  public string Level { get; set; }
}
```

The preceding code is a class that represents a DTO. Remember, DTOs should only contain properties that we need to expose from the outside world or consumers. In essence, DTOs are meant to be light classes.

Create another sub-folder called Players and copy the following code:

```
public class CreatePlayerRequest {
  [Required]
  public string NickName { get; set; }
  [Required]
  public List<CreatePlayerInstrumentRequest>
  PlayerInstruments { get; set; }
}
```

The preceding code contains a couple of properties. Notice that we've referenced the CreatePlayerInstrumentRequest class in a List type representation. This is to enable a one-to-many relation when you create a new player with multiple instruments. You can see that each property has been decorated with the [Required] attribute to ensure that the properties will not be left empty when submitting a request. The [Required] attribute is built into the framework and sits under the System. ComponentModel.DataAnnotations namespace. The process of enforcing validations to Models is called **data annotation**. If you want to have a clean Model definition and perform complex predefined validations in a fluent way, then you may try considering using FluentValidation instead.

Defining an interface

As you may have seen in the previous module's examples, we can directly pass an instance of the DbContext in the Controller via constructor injection. However, when building real applications, you should make your Controllers as thin as possible and take business logic and data processing outside your Controllers. Your Controllers should only handle things like routing, Model validations, and delegating the data processing to a separate service. With that said, we are going to create a service that handles the communication between the Controllers and DbContext.

Implementing the code logic in a separate service is a way of making your Controller thin and simple. However, we don't want the Controller to directly depend on the actual service implementation as it can lead to tightly coupled dependencies. Instead, we will create an interface abstraction to decouple the actual service dependency. This makes your code more testable, extensible, and easier to manage. You may review module Dependency Injection, for details about interface abstraction.

Now, create a new folder called interfaces at the root of the application. Within the folder, create a new interface called IPlayerService and copy the following code:

```
public interface IPlayerService {
  Task CreatePlayerAsync(CreatePlayerRequest playerRequest);
}
```

The preceding code defines a method that takes the CreatePlayerRequest class that we created earlier. The method returns a Task, which denotes that the method will be invoked asynchronously.

Now that we have an interface defined, we should now be able to create a service that implements it. Let's see how to do that in the next step.

Implementing the service

In this section, we are going to implement the interface we defined earlier to build the actual logic for the method defined in the interface.

Go ahead and create a new folder called Services at the root of the application and then replace the default generated code with the following:

```
public class PlayerService : IPlayerService {
  private readonly CodeFirstDemoContext _dbContext;
  public PlayerService(CodeFirstDemoContext dbContext) {
    _dbContext = dbContext;
  }
}
```

In the preceding code, we've defined a private and readonly field of the CodeFirstDemoContext and added a class constructor that injects the CodeFirstDemoContext as a dependency of the PlayerService class. By applying **dependency injection** in the constructor, any methods within the class will be able to access the instance of the CodeFirstDemoContext, allowing us to invoke all its available methods and properties.

You may also notice that the class implements the IPlayerService interface. Since an interface defines a contract that a class should follow, then the next step that we are going to take is to implement the CreatePlayerAsync() method. Go ahead and append the following code within the PlayerService class:

```
public async Task CreatePlayerAsync(CreatePlayerRequest playerRequest) {
  using var transaction = await _dbContext.Database.
  BeginTransactionAsync();
  try {
    var player = new Player {
      NickName = playerRequest.NickName,
      JoinedDate = DateTime.Now
    };
    await _dbContext.Players.AddAsync(player);
    await _dbContext.SaveChangesAsync();
    var playerId = player.PlayerId;
  } catch {
    await transaction.RollbackAsync();
    throw;
  }
}
```

In the preceding code, the method was implemented as asynchronous by marking it with the async keyword. What the code does is it first adds a new Player entry in the database and gets back the PlayerId that has been generated.

To complete the CreatePlayerAsync() method. Copy the following code within the try block after the `var playerId = player.PlayerId;` line:

```
var playerInstruments = new List<PlayerInstrument>();
foreach (var instrument in playerRequest.PlayerInstruments) {
  playerInstruments.Add(new PlayerInstrument {
    PlayerId = playerId,
    InstrumentTypeId = instrument.InstrumentTypeId,
    ModelName = instrument.ModelName,
    Level = instrument.Level
  });
}
_dbContext.PlayerInstruments.AddRange(playerInstruments);
await _dbContext.SaveChangesAsync();
await transaction.CommitAsync();
```

The preceding code iterates through the playerRequest.PlayerInstruments collection and creates the associated PlayerInstrument in the database along with the playerId.

Since the dbo.PlayerInstruments table depends on the dbo.Players table, we've used the EF Core database transaction feature to ensure that records in both tables will only be created on a successful operation. This is to avoid the data being corrupted when one database operation is failing. You can see it by invoking the transaction.CommitAsync() method when everything runs successfully and invoking the transaction.RollbackAsync() method within the catch block to revert any changes when an error occurs.

Let's proceed to the next step and register the service.

Registering the service

We need to register the interface mapping into the DI container in order for us to inject the interface into any other classes within the application. Add the following code within the ConfigureServices() method of the Startup.cs file:

```
services.AddTransient<IPlayerService, PlayerService>();
```

The preceding code registers the PlayerService class in the DI container as an IPlayerService interface type with a transient scope. This tells the framework to resolve interface dependency we inject it into the Controller class constructor at runtime.

Now that we have implemented the service and wired up the piece in the DI container, we can now inject the IPlayerService as a dependency of the Controller class, which we are going to create in the next step.

Creating the API controller

Go ahead and right-click on the Controllers folder and then select **Add > Controller > API Controller Empty**, and then click **Add**.

Name the class PlayersController.cs and then click **Add**. Now, copy the following code so it will look similar to this:

```
[Route("api/[controller]")]
[ApiController]
public class PlayersController : ControllerBase {
  private readonly IPlayerService _playerService;
  public PlayersController(IPlayerService playerService) {
    _playerService = playerService;
  }
}
```

The preceding code is the typical structure of an API Controller class. Web API controllers use the same routing middleware that's used for MVC except that it uses **attribute routing** to define the routes. The [Route] attribute enables you to specify whatever route for your API endpoints. The ASP.NET Core API default convention uses the format api/[controller] where the [controller] segment represents a token placeholder to automatically build the route based on the Controller class prefixed name. For this example, the route api/[controller] will be translated to api/players where players came from the PlayersController class name. The [ApiController] attribute enables the Controller to apply API-specific behaviors for your APIs, such as attribute routing requirements, automatic handling of HTTP 404 and 405 responses, problem details for errors, and more.

Web APIs should derive from the ControllerBase abstract class to utilize the existing functionalities built into the framework for building RESTful APIs. In the preceding code, you can see that we've now injected the IPlayerService as a dependency instead of the DbContext itself. This decouples your data access implementation from the Controller class, allowing more flexibility when you decide to change the underlying implementation of the service, as well as making your Controller thin and clean.

Now, append the following code for the POST endpoint:

```
[HttpPost]
public async Task<IActionResult> PostPlayerAsync([FromBody]
CreatePlayerRequest playerRequest) {
  if (!ModelState.IsValid) { return BadRequest(); }
  await _playerService.CreatePlayerAsync(playerRequest);
  return Ok("Record has been added successfully.");
}
```

The preceding code takes a CreatePlayerRequest class as an argument. By marking the argument with the [FromBody] attribute, we tell the framework to only accept values from the body of the request for this endpoint. You can also see that the PostPlayerAsync() method has been decorated with the [HttpPost] attribute, which signifies that the method can only be invoked for HTTP POST requests. You can see that the method implementation is now much cleaner as it only validates the DTO and delegates the actual data processing to the service. ModelState.IsValid() will check for any predefined validation rules for the CreatePlayerRequest Model and returns a Boolean to indicate whether the validation failed or passed. In this example, it only checks whether both properties in the CreatePlayerRequest class are not empty by checking against the [Required] attribute annotated for each property.

At this point, you should now have the POST endpoint available. Let's do a quick test to ensure that the endpoint is working as we expect.

Testing the POST endpoint

We will use Postman to test our API endpoints. Postman is really a handy tool to test APIs without having to create a UI, and it's absolutely free. Go ahead and download it here:

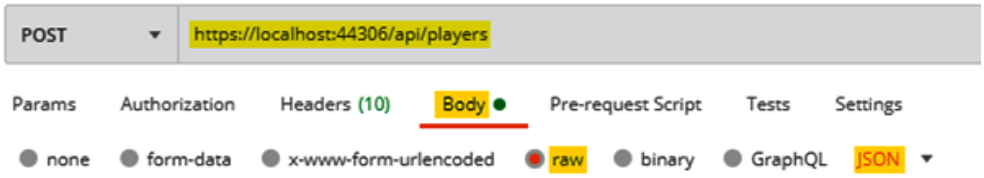
https://www.getpostman.com/.

After downloading Postman, install it on your machine so you can start testing. Now, run the application first, by pressing the **Ctrl + F5** keys to launch the application in the browser.

Open Postman and then make a POST request with the following URL: https://localhost:44306/api/players.

Note that port 44306 might be different in your case, so make sure to replace that value with the actual port your local application is running at. You can see launchSettings.json under the Properties folder in your project to learn more about how launch URL profiles are configured.

Let's continue with the testing. In Postman, switch to the Body tab, select the raw option, and select JSON as the format. Refer to the following figure below for a visual reference:

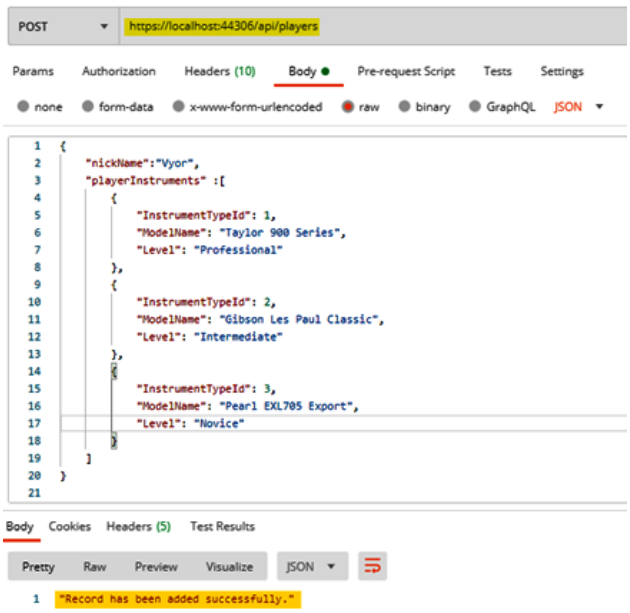


Now, in the raw textbox, copy the following JSON as the request payload:

```
{
  "nickName": "Vianne",
  "playerInstruments": [
    {
      "InstrumentTypeId": 1,
      "ModelName": "Taylor 900 Series",
      "Level": "Professional"
    },
    {
      "InstrumentTypeId": 2,
      "ModelName": "Gibson Les Paul Classic",
      "Level": "Intermediate"
    },
    {
      "InstrumentTypeId": 3,
      "ModelName": "Pearl EXL705 Export",
      "Level": "Novice"
    }
  ]
}
```

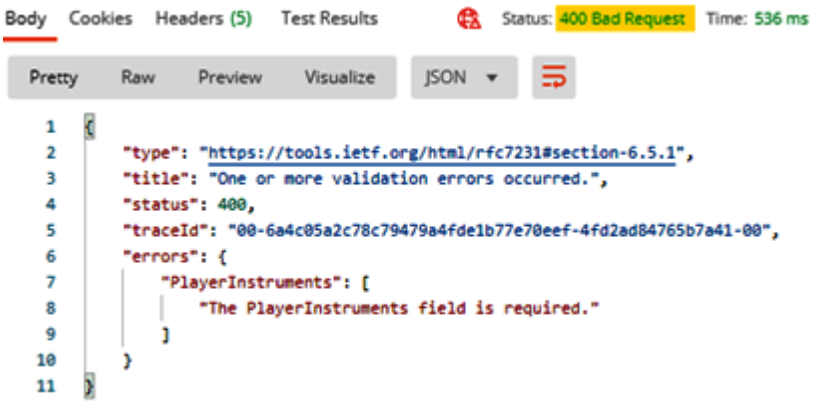
The preceding code is the JSON request body that the /api/players endpoint expects. If you remember, the POST endpoint expects CreatePlayerRequest as an argument. The JSON payload in the preceding code represents that.

Now, click the Send button in Postman to invoke the HTTP POST endpoint and you should be presented with the following result as shown in figure below:



The preceding screenshot returns a 200 HTTP status with a response message indicating that the record has been created successfully in the database. You can verify the newly inserted data by looking at the dbo.Players and dbo.PlayerInstruments database table.

Now, let's test the Model validation. The following figure shows the result if we omit the playerInstruments attribute in the request body and hit the Send button:



The preceding screenshot shows a validation error in ProblemDetails format with the 400 HTTP Status code. This is how the response is going to look when you annotate a Model property to be required and you don't supply it when invoking the API endpoint.

Now that you've learned the basics of creating a Web API endpoint for a POST request, let's continue to get our hands dirty by exploring other examples.

Implementing HTTP GET endpoints

In this section, we'll create a couple of HTTP GET endpoints for you to learn some of the basic ways to fetch data from the database.

Defining the DTO

Just like what we did for the POST endpoint, the first step that we need to do is to create a DTO class for us to define the properties that we need to expose. Create a new class called GetPlayerResponse within the Dto/Players folder and copy the following code:

```
public class GetPlayerResponse {
  public int PlayerId { get; set; }
  public string NickName { get; set; }
  public DateTime JoinedDate { get; set; }
  public int InstrumentSubmittedCount { get; set; }
}
```

The preceding code is just a plain class that holds a few properties. These are the properties that we are going to return to the client as the response.

For this endpoint implementation, we are not going to return all records from the database to the client because it would be very inefficient. Imagine you have thousands or millions of records in your database and your API endpoint tries to return all of them at once. That would definitely blow down the entire performance of your application and, worse, it could make your application unusable.

Implementing GET with pagination

To prevent potential performance issues from happening, we will implement a pagination feature to value performance. This will enable us to limit the amount of data to return to the client and maintain performance even if the data in the database grows.

Now, go ahead and create a new class called PagedResponse within the Dto folder. Copy the following code:

```
public class PagedResponse<T> {
  const int _maxPageSize = 100;
  public int CurrentPageNumber { get; set; }
  public int PageCount { get; set; }
  public int PageSize {
    get => 20;
    set => _ = (value > _maxPageSize) ? _maxPageSize :
    value;
  }
  public int TotalRecordCount { get; set; }
  public IList<T> Result { get; set; }
  public PagedResponse() {
    Result = new List<T>();
  }
}
```

The preceding code defines some basic metadata for the paged Model. Notice that we've set the constant _maxPageSize variable to 100. This is the value of the maximum number of records that the API GET endpoint will return to the client. The PageSize property is set to 20 as the default in case the client won't specify the value when invoking the endpoint. Another thing to notice is we've defined a generic property Result of type IList<T>. The T can be of any Model that you want to return as paginated.

Next, let's create a new class called UrlQueryParameters within the Dto folder. Copy the following code:

```
public class UrlQueryParameters {
  public int PageNumber { get; set; };
  public int PageSize { get; set; };
}
```

The preceding code will be used as the method argument for the GET endpoint that we are going to implement later. This is to allow clients to set the page size and number when requesting the data.

Next, create a new folder called Extensions at the root of the application. Within the Extensions folder, create a new class called PagerExtension and copy the following code:

```
public static class PagerExtension {
  public static async Task<PagedResponse<T>>
  PaginateAsync<T>(
  this IQueryable<T> query,
  int pageNumber,
  int pageSize)
  where T : class {
    var paged = new PagedResponse<T>();
    pageNumber = (pageNumber < 0) ? 1 : pageNumber;
    paged.CurrentPageNumber = pageNumber;
    paged.PageSize = pageSize;
    paged.TotalRecordCount = await query.CountAsync();
    var pageCount = (double)paged.TotalRecordCount / pageSize;
    paged.PageCount = (int)Math.Ceiling(pageCount);
    var startRow = (pageNumber - 1) * pageSize;
    paged.Result = await query.Skip(startRow).
```

```
    Take(pageSize).ToListAsync();
    return paged;
  }
}
```

The preceding code is where the actual pagination and calculation is happening. The PaginateAsync() method takes three parameters in order to perform pagination and returns a Task of type PagedResponse<T>. The this keyword in the method argument denotes that the method is an extension method of the type IQueryable<T>. Notice that the code uses the LINQ Skip() and Take() methods to paginate the result.

Now that we have defined the DTO and implemented an extension method to paginate the data, let's continue to the next step and add a new method signature in the IPlayerService interface.

Updating the interface

Go ahead and add the following code within the IPlayerService interface:

```
Task<PagedResponse<GetPlayerResponse>>
GetPlayersAsync(UrlQueryParameters urlQueryParameters);
```

The preceding code defines a method that takes UrlQueryParameters as an argument and returns PagedResponse of type GetPlayerResponse Model. Next, we'll update the PlayerService to implement this method.

Updating the service

Add the following code within the PlayerService class:

```
public async Task<PagedResponse<GetPlayerResponse>>GetPlayersAsync(
      UrlQueryParameters parameters) {
  var query = await _dbContext.Players
  .AsNoTracking()
  .Include(p => p.Instruments)
  .PaginateAsync(parameters.PageNumber,parameters.PageSize);

  return new PagedResponse<GetPlayerResponse> {
    PageCount = query.PageCount,
    CurrentPageNumber = query.CurrentPageNumber,
    PageSize = query.PageSize,
    TotalRecordCount = query.TotalRecordCount,
    Result = query.Result.Select(p => new GetPlayerResponse {
      PlayerId = p.PlayerId,
      NickName = p.NickName,
      JoinedDate = p.JoinedDate,
      InstrumentSubmittedCount = p.Instruments.Count
    }).ToList()
  };
}
```

The preceding code shows the EF Core way of querying data from the database. Since we are only fetching data, we've used the AsNoTracking() method to improve the query performance. No tracking queries are much quicker because they eliminate the need to set up change tracking information for the entity, thus they are quicker to execute and improve query performance for read-only data. The Include() method allows us to load the associated data in the query results. We then call the PaginateAsync() extension method that we implemented earlier to chunk the data based on UrlQueryParameters property values. Finally, we construct the return response using a **LINQ method-based query**. In this case, we return a PagedResponse object with the GetPlayerResponse. type

To see the actual SQL script generated by EF Core, or if you prefer to use raw SQL script to query the data, check out the links in the Further reading section of this module.

Let's move on to the next step and update the Controller class to define the GET endpoint.

Updating the controller

Add the following code within the PlayersController class:

```
[HttpGet]
public async Task<IActionResult> GetPlayersAsync(
  [FromQuery]UrlQueryParameters urlQueryParameters) {
  var player = await _playerService.
  GetPlayersAsync(urlQueryParameters);
  //removed null validation check for brevity
  return Ok(player);
}
```

The preceding code takes UrlQueryParameters as the request parameter. By decorating the parameter with the [FromQuery] attribute, we tell the framework to evaluate and get the request values from the query string. The method invokes GetPlayersAsync() from the IPlayerService interface and passes along UrlQueryParameters as the argument. If the result is null, we return NotFound(); otherwise, we return Ok() along with the result.

Now, let's test the endpoint to ensure we get what we expect

Testing the endpoint

Now run the application and open Postman. Make an HTTP GET request with the following endpoint:

```
https://localhost:44306/api/players?pageNumber=1&pageSize=2
```

You can set the value of pageNumber and pageSize to whatever you want and then hit the Send button. The following Figure 7.13 is a sample screenshot of the response output:

Sweet! Now, let's try another GET endpoint example.

Implementing GET by ID

In this section, we will learn how to fetch data from the database by passing the ID of the record. We will see how we can query the related data from each database table and return a response to the client containing detailed information coming from the different tables.

Defining the DTOs

Without further ado, let's go ahead and create a new class called GetPlayerInstrumentResponse within the Dto/PlayerInstrument folder. Copy the following code:

```
public class GetPlayerInstrumentResponse {
  public string InstrumentTypeName { get; set; }
  public string ModelName { get; set; }
  public string Level { get; set; }
}
```

Create another new class called GetPlayerDetailResponse with the Dto/Players folder and then copy the following code:

```
public class GetPlayerDetailResponse {
  public string NickName { get; set; }
  public DateTime JoinedDate { get; set; }
  public List<GetPlayerInstrumentResponse> PlayerInstruments { get; set; }
}
```

The preceding classes represent the response DTO or Model that we are going to expose to the client. Let's move on to the next step and define a new method in the IPlayerService interface.

Updating the interface

Add the following code within the IPlayerService interface:

```
Task<GetPlayerDetailResponse> GetPlayerDetailAsync(int id);
```

The preceding code is the method signature that we are going to implement in the service. Let's go ahead and do that.

Updating the service

Add the following code within the PlayerService class:

```
public async Task<GetPlayerDetailResponse>
GetPlayerDetailAsync(int id) {
  var player = await _dbContext.Players.FindAsync(id);
  //removed null validation check for brevity
  var instruments = await
  (from pi in _dbContext.PlayerInstruments
   join it in _dbContext.InstrumentTypes
 on pi.InstrumentTypeId equals it.InstrumentTypeId
   where pi.PlayerId.Equals(id)
   select new GetPlayerInstrumentResponse {
     InstrumentTypeName = it.Name, ModelName = pi.ModelName,
     Level = pi.Level
   }).ToListAsync();
  return new GetPlayerDetailResponse {
    NickName = player.NickName,
    JoinedDate = player.JoinedDate,
    PlayerInstruments = instruments
  };

}
```

The preceding code contains the actual implementation of the GetPlayerDetailAsync() method. The method in asynchronous that takes an id as the argument and returns a GetPlayerDetailResponse type. The code first checks whether the given id has associated records in the database using the FindAsync() method. If the result is null, we return default or null; otherwise, we query the database by joining the related tables using LINQ query expressions. If you've written T-SQL before, you'll notice that the query syntax is pretty much similar to SQL except that it manipulates the conceptual Entity Models providing strongly-typed code with rich IntelliSense support.

Now that we have our method implementation in place, let's move on to the next step and update the Controller class to define another GET endpoint.

Updating the controller

Add the following code within the PlayersController class:

```
[HttpGet("{id:long}/detail")]
public async Task<IActionResult> GetPlayerDetailAsync(int id) {
  var player = await _playerService.GetPlayerDetailAsync(id);
  //removed null validation check for brevity
  return Ok(player);

}
```

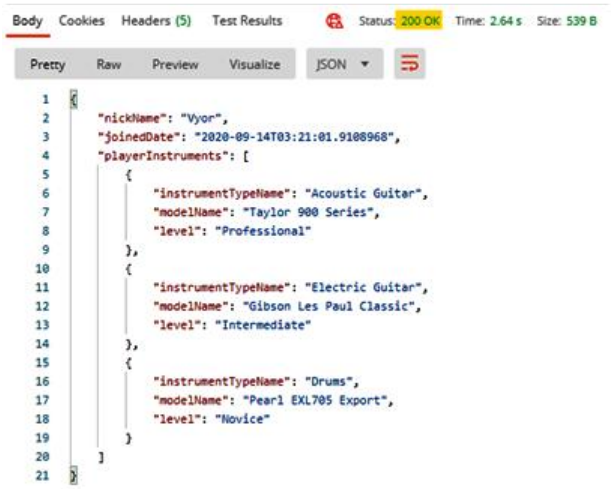
The preceding code defines a GET endpoint with a route configured to "{id:long}/detail". The id in the route represents a parameter that you can set in the URL. As a friendly reminder, consider using GUID as record identifiers when exposing a resource ID to the outside world instead of identity seed. This is to reduce the risk of exposing data to malicious users trying to sniff your endpoints by just incrementing the id value.

Testing the endpoint

Run the application and make a `GET` request in Postman with the following endpoint:

```
https://localhost:44306/api/players/1/detail
```

The following Figure below is a sample screenshot of the response output:



Now that you've learned various ways to implement HTTP GET endpoints, let's move on to the next section and see how we can implement the PUT endpoint.

Implementing an HTTP PUT endpoint

In this section, we are going to learn how to update a record in the database by utilizing the HTTP PUT method.

Defining a DTO

To make this example simple, let's just update a single column in the database. Go ahead and create a new class called UpdatePlayerRequest within the Dto/Players folder.

Copy the following code:

```
public class UpdatePlayerRequest {
  [Required]
  public string NickName { get; set; }
}
```

Next, we'll update the IPlayerService interface to include a new method for performing a database update.

Updating the interface

Add the following code within the IPlayerService interface:

```
Task<bool> UpdatePlayerAsync(int id, UpdatePlayerRequest playerRequest);
```

The preceding code is the method signature for updating the dbo.Players table in the database. Let's move on to the next step and implement this method in the service.

Updating the service

Add the following code within the IPlayerService class:

```
public async Task<bool> UpdatePlayerAsync(int id,UpdatePlayerRequest playerRequest) {
  var playerToUpdate = await _dbContext.Players.  FindAsync(id);
  //removed null validation check for brevity
  playerToUpdate.NickName = playerRequest.NickName;
  _dbContext.Update(playerToUpdate);
  return await _dbContext.SaveChangesAsync() > 0;
}
```

The preceding code is pretty much straightforward. First, it checks whether the id has an associated record in the database. If the result is null, we return false; otherwise, we update the database with the new value of the NickName property. Now, let's move on to the next step and update the Controller class to invoke this method.

Updating the controller

Add the following code within the PlayersController class:

```
[HttpPut("{id:long}")]
public async Task<IActionResult> PutPlayerAsync(int id,
  [FromBody] UpdatePlayerRequest playerRequest) {
  if (!ModelState.IsValid) { return BadRequest(); }
  var isUpdated = await _playerService.UpdatePlayerAsync(id,playerRequest);
  if (!isUpdated) {
    return NotFound($"PlayerId { id } not found.");
  }
  return Ok("Record has been updated successfully.");
}
```

The preceding code takes an id and an UpdatePlayerRequest Model from the request body. The method is decorated with [HttpPut("{id:long}")], which signifies that the method can only be invoked in an HTTP PUT request. The id in the route denotes a parameter in the URL.

Testing the PUT endpoint

Run the application and make a PUT request in Postman with the following endpoint:

```
https://localhost:44306/api/players/1
```

Now, just like in the POST request, copy the following code in the raw textbox:

```
{
  "nickName": "Vynn"
}
```

The preceding code is the required parameter for the PUT endpoint. In this particular example, we will change the NickName value to “Vynn” for id equal to 1. Clicking the Send button should update the record in the database.

Now, when you perform a GET request by id via /api/players/1/detail, you should see that the NickName for id holding the value of 1 has been updated. In this case, the value “Vjor” is updated to “Vynn”.

Implementing an HTTP Delete endpoint

In this section, we are going to learn how to implement an API endpoint that performs database record deletion. For this example, we don’t need to create a DTO since we are just going to pass the id in the route for the delete endpoint. So, let’s jump right in by updating the IPlayerService interface to include a new method for deletion.

Updating the interface

Add the following code within the IPlayerService interface:

```
Task<bool> DeletePlayerAsync(int id);
```

The preceding code is the method signature that we are going to implement in the next section. Notice that the signature is similar to the update method except that we are not passing a DTO or Model as an argument.

Let’s move on to the next step and implement the method in the service.

Updating the service

Add the following code within the PlayerService class:

```
public async Task<bool> DeletePlayerAsync(int id) {
  var playerToDelete = await _dbContext.Players
  .Include(p => p.Instruments)
  .FirstAsync(p => p.PlayerId.
  Equals(id));
  //removed null validation check for brevity
  _dbContext.Remove(playerToDelete);
  return await _dbContext.SaveChangesAsync() > 0;

}
```

The preceding code uses the Include() method to perform cascading deletions with the associated records in the dbo.PlayerIntruments table. We then use the FirstAsync() method to filter the record to be deleted based on the id value. If the result is null, we return false; otherwise, we perform the record deletion using the _dbContext.Remove() method. Now, let’s update the Controller class to invoke this method.

Updating the controller

Add the following code within the PlayersController class:

```
[HttpDelete("{id:long}”)]
public async Task<IActionResult> DeletePlayerAsync(int id)
{
  var isDeleted = await _playerService.DeletePlayerAsync(id);
  if (!isDeleted) {
    return NotFound($"PlayerId { id } not found.");
  }
  return Ok("Record has been deleted successfully.");

}
```

The implementation in the preceding code is also similar to the update method, except that the method is now decorated with the [HttpDelete] attribute. Now, let's test the DELETE API endpoint.

Testing the DELETE endpoint

Run the application again and make a DELETE request in Postman with the following endpoint:

```
https://localhost:44306/api/players/1
```

Clicking the **Send** button should show a successful response output when the record with id equal to 1 has been deleted from the database.

That's it! If you've made it this far, then you should now be familiar with building APIs in ASP.NET Core and be able to apply the things that you've learned in this module when building your own APIs. As you may know, there are many things that you could do to improve this project. You could try incorporating features such as logging, caching, HTTP response consistency, error handling, validations, authentication, authorization, Swagger documentation, and exploring other HTTP methods such as PATCH.

Summary

In this module, we've covered the concepts and the different design workflows for implementing Entity Framework Core as your data access mechanism. Understanding how the database-first and code-first workflows work is very important when deciding how you want to design your data access layer. We've learned how APIs and data access work together to serve and consume data from various clients. Learning how to create APIs that deal with a real database from scratch gives you a better understanding of how the underlying backend application works, especially if you will be working with real applications that use the same technology stack.

We've learned how to implement the common HTTP methods in ASP.NET Core Web API with practical hands-on coding exercises. We've also learned how to design an API to make it more testable and maintainable by leveraging interface abstraction, and learned about the concepts of having DTOs to value the separation of concerns and how to make API controllers as thin as possible. Learning this technique enables you to easily manage your code, without affecting much of your application code when you decide to refactor your application. Finally, we've learned how to easily test API endpoints using Postman.

In the next module, you are going to learn about ASP.NET Core Identity for securing web apps, APIs, managing user accounts, and more.