ASP.NET Web API

ASP.NET Web API is a framework for building HTTP services that can be accessed from any client including browsers and mobile devices. It is an ideal platform for building RESTful applications on the .NET Framework.

This Web API training will help you learn the essentials of ASP.NET Web API starting from the basics to advanced level. The training is broken down into sections, where each section contains several related topics that are packed with easy-to-understand explanations, real-world examples, useful tips and informative notes section.

This training is designed for beginners and professionals who want to learn Web API step-by-step.

**Table of Contents:**

* **What is Web API?**
* **Create Web API project**
* **Test Web API**
* **Web API Controllers**
* **Web API Routing**
* **Parameter Binding**
* **Action Method Return Type**
* **Web API Request/Response Data Formats**
* **ASP.NET Web API: Media-Type Formatters**
* **Web API Filters**
* **Create Web API for CRUD operation - Part 1**
* **Create Web API for CRUD operation - Part 2: Implement Get Method**
* **Create Web API for CRUD operation - Part 3: Implement Post Method**
* **Create Web API for CRUD operation - Part 4: Implement Put Method**
* **Create Web API for CRUD operation - Part 5: Implement Delete Method**
* **Consume Web API for CRUD operation**
* **Consume Web API Get method in ASP.NET MVC**
* **Consume Web API Post method in ASP.NET MVC**
* **Consume Web API Put method in ASP.NET MVC**
* **Consume Web API Delete Method in ASP.NET MVC**
* **Consume Web API in .NET using HttpClient**
* **Configure Dependency Injection with Web API**
* **Web API Hosting**

# **What is Web API?**

Before we understand what Web API is, let's see what an API (Application Programing Interface) is.

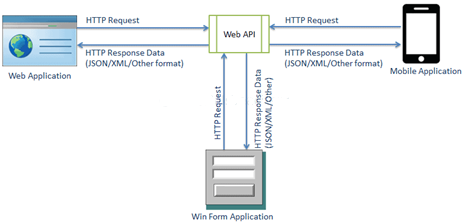
As per [Wikipedia's Definition of API](https://en.wikipedia.org/wiki/Application_programming_interface): In computer programming, an application programming interface (API) is a set of subroutine definitions, protocols, and tools for building software and applications.

To put it in simple terms, API is some kind of interface which has a set of functions that allow programmers to access specific features or data of an application, operating system or other services.

A Web API as the name suggests, is an API over the web which can be accessed using HTTP protocol. It is a concept and not a technology. We can build Web API using different technologies such as Java, .NET etc. For example, Twitter's [REST APIs](https://dev.twitter.com/rest/public) provide programmatic access to read and write data using which we can integrate twitter's capabilities into our own application.

ASP.NET Web API

The ASP.NET Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms such as the web, windows, mobile etc. It works the almost same way as ASP.NET MVC web application except that it sends data as a response instead of html view. It is like a webservice or WCF service, but the exception is that it only supports HTTP protocol.



Web API

ASP.NET Web API Characteristics

1. ASP.NET Web API is an ideal platform for building RESTful services.
2. ASP.NET Web API is built on top of ASP.NET and supports ASP.NET request/response pipeline
3. ASP.NET Web API maps HTTP verbs to method names.
4. ASP.NET Web API supports different formats of response data. Built-in support for JSON, XML, BSON format.
5. ASP.NET Web API can be hosted in IIS, Self-hosted or another web server that supports .NET 4.0+.
6. The ASP.NET Web API framework includes a new HttpClient to communicate with Web API server. HttpClient can be used in ASP.MVC server side, Windows Form application, Console application or other apps.

ASP.NET Web API Versions

| Web API Version | Supported .NET Framework | Coincides with | Supported in |
| --- | --- | --- | --- |
| Web API 1.0 | .NET Framework 4.0 | ASP.NET MVC 4 | VS 2010 |
| Web API 2 | .NET Framework 4.5 | ASP.NET MVC 5 | VS 2012, 2013, 2017, 2019 |

ASP.NET Web API vs WCF

| Web API | WCF |
| --- | --- |
| Open source and ships with .NET framework. | Ships with .NET framework |
| Supports only HTTP protocol. | Supports HTTP, TCP, UDP and custom transport protocol. |
| Maps http verbs to methods | Uses attributes-based programming model. |
| Uses routing and controller concept like ASP.NET MVC. | Uses Service, Operation and Data contracts. |
| Does not support Reliable Messaging and transaction. | Supports Reliable Messaging and Transactions. |
| Web API can be configured using HttpConfiguration class but not in web.config. | Uses web.config and attributes to configure a service. |
| Ideal for building RESTful services. | Supports RESTful services but with limitations. |

When to choose WCF?

* Choose WCF if you use .NET Framework 3.5. Web API does not support .NET 3.5 or below.
* Choose WCF if your service needs to support multiple protocols such as HTTP, TCP, Named pipe.
* Choose WCF if you want to build a service with WS-\* standards like Reliable Messaging, Transactions, Message Security.
* Choose WCF if you want to use Request-Reply, One Way, and Duplex message exchange patterns.

When to choose ASP.NET Web API?

* Choose Web API if you are using .NET framework 4.0 or above.
* Choose Web API if you want to build a service that supports only HTTP protocol.
* Choose Web API to build RESTful HTTP based services.
* Choose the Web API if you are familiar with ASP.NET MVC.

Let's begin by creating a simple ASP.NET Web API project using Visual Studio in the next section.

# **Create Web API project**

Here, you will learn how to create a new ASP.NET Web API project using Visual Studio.

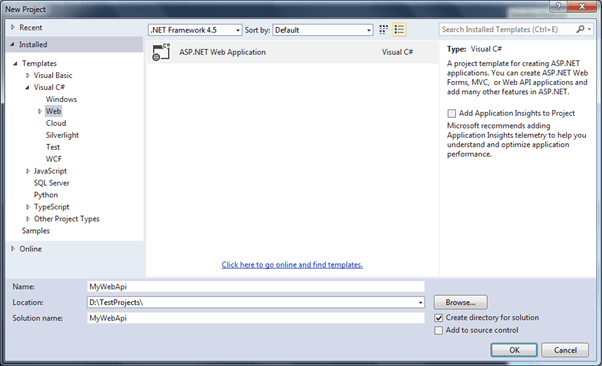
You can create a Web API project in two ways.

1. Web API with MVC Project
2. Stand-alone Web API Project

## **Web API with MVC Project**

Visual Studio (2013/2015/2017/2019) includes Web API template which creates a new Web API project with ASP.NET MVC application and includes all the necessary references to get started.

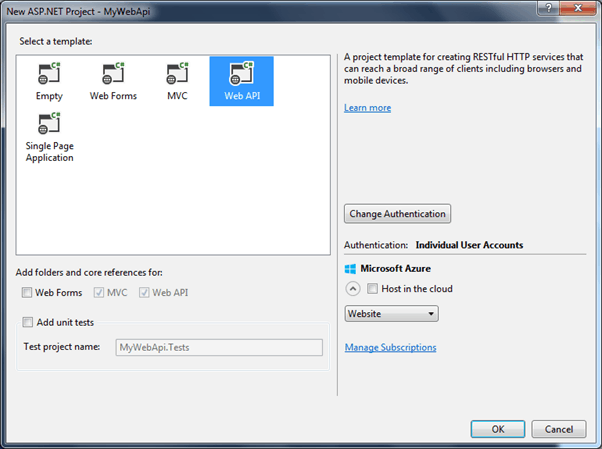
For this, open Visual Studio and click on **File** menu and click on **New Project.** This will open New Project popup as below.



Create Web API project

In the New Project popup, expand **Visual C#** and select **Web** node in the left pane. Select **ASP.NET Web Application** template in the middle pane and enter the name of a project and click **OK**. (ASP.NET Web Application (.NET Framework) template in Visual Studio)

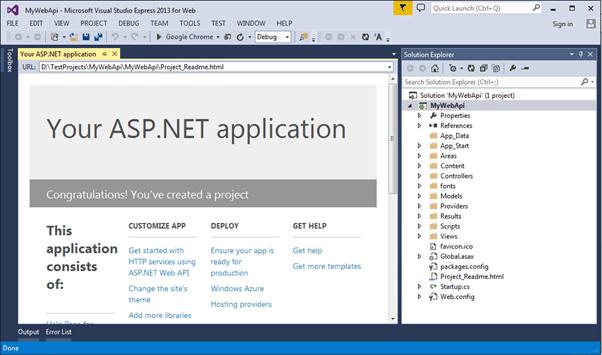
This will open New ASP.NET Project popup as shown below.



Select Web API Template

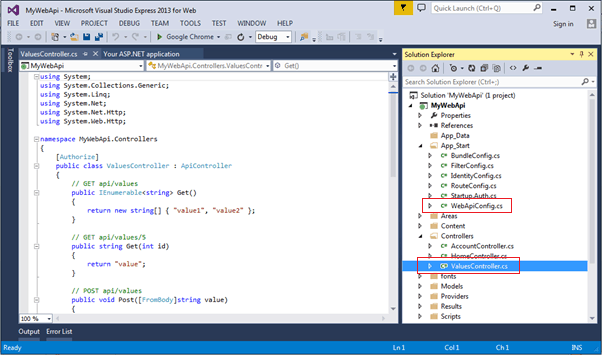
Select Web API in the above popup. Notice that this has selected and disabled MVC and Web API checkboxes. It means that it will add necessary folders and references for both MVC and Web API.

Click Ok to create a new Web API + MVC project as shown below.



Web API project

This project is same as default MVC project with two specific files for Web API, WebApiConfig.cs in **App\_Start** folder and ValuesController.cs in **Controllers** folder as shown below.

Web API project

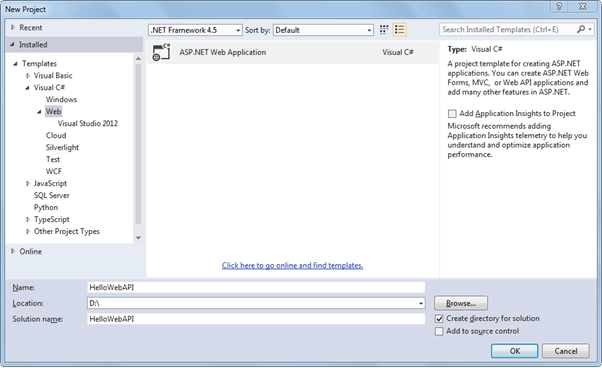
The WebApiConfig.cs is a configuration file for Web API. You can configure routes and other things for web API, same as RouteConfig.cs is used to configure MVC routes. It also creates Web API controller ValuesController.cs by default. You will learn about WebApiConfig and Controller in the next section.

Thus, you can create a Web API project with MVC to get started on your application.

## **Stand-alone Web API Project**

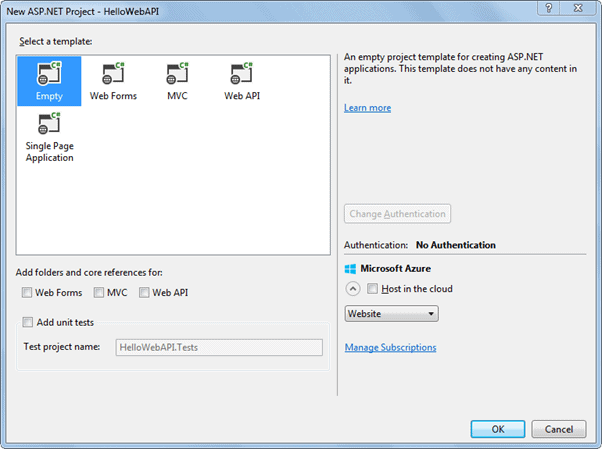
Here, we will create a new stand-alone Web API project without MVC project.

For this, open Visual Studio for Web -> go to **File** menu and select **New Project..**This will open**New Project** popup as below.



Create Web API Project

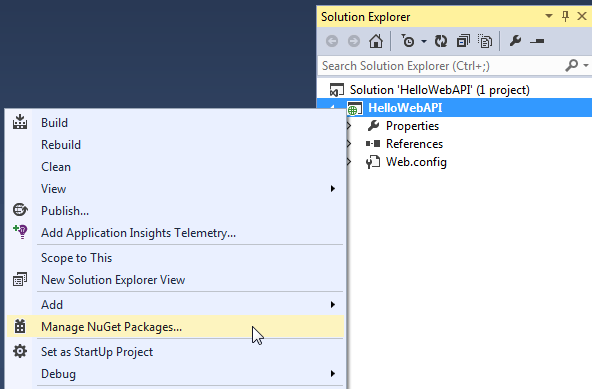
Select **Web** template in the left pane and **ASP.NET Web Application** in the middle pane. Enter the name of the project, location and Solution name as shown above. Clicking on **OK** will open a popup as shown below.



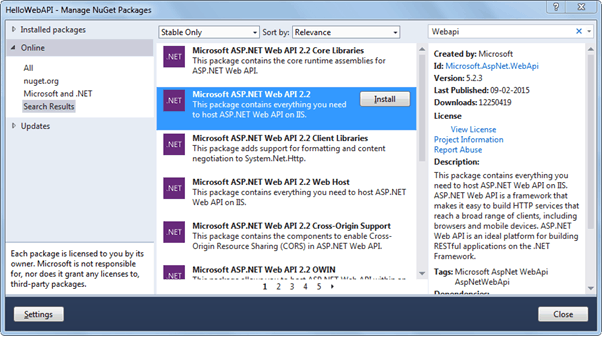
Select Project Template

In the above popup, select **Empty** as a template and click ok. This will create an empty "HelloWebAPI" project.

Now, we need to add the latest Web API references using **NuGet Package Manager**. Right Click on the project and click **Manage NuGet Packages..** as shown below.

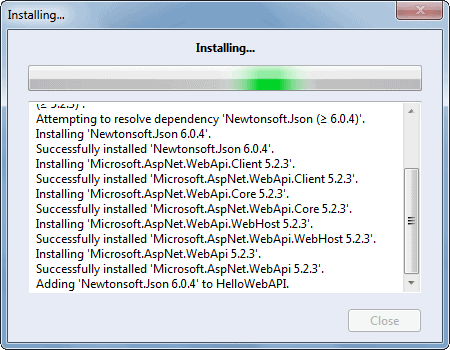
Open NuGet

This will open Manage NuGet Packages popup. Select **Online** in the left pane and search for Webapi (make sure that internet connection is on). This will display all the Web API related packages. Select **Microsoft ASP.NET Web API2.2** package and click on **Install** as shown below.



Install Web API Package

Accept the license agreement and continue.

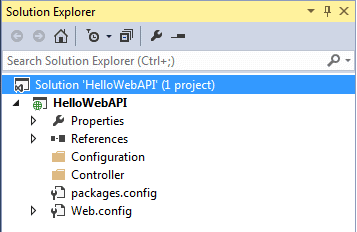


Install Web API Package

The following Web API packages are displayed upon successful installation.

Web API Packages

Now, create Controllers and Configuration folder in the HelloWebAPI project as shown below. We will add Web API controller in the Controllers folder and configuration class in the Configuration folder.

 Web API Project

Now, add a new class in the configuration folder and name it "HelloWebAPIConfig" with the following content. (You may give any appropriate name)

Example: Web API Configuration

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Web;

**using** System.Web.Http;

**namespace** HelloWebAPI.Configuration{

**public static class** HelloWebAPIConfig{

**public static void** Register(HttpConfiguration config){

// Web API routes

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

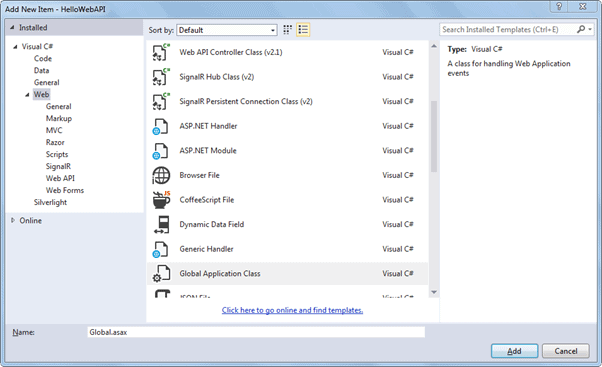
);

}

}

}

Now, add Global.asax by right clicking on the project -> select Add -> click New Item.. to open Add New Item popup as below. Select Global Application Class and click OK.

Add Global.asax

This will add Global.asax file into the project. We need to configure our Web API routes when the application starts. So, call HelloWebAPIConfig.Register() method in the Application\_Start event in the Global.asax as shown below.

Global.asax

**public class** Global : System.Web.HttpApplication{

**protected void** Application\_Start(**object** sender, EventArgs e){

GlobalConfiguration.Configure(HelloWebAPIConfig.Register);

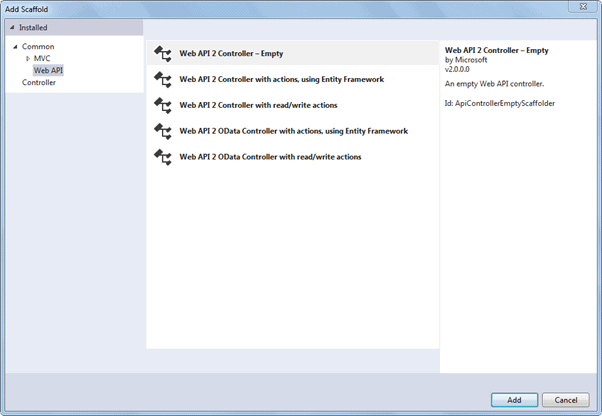
}

}

Web API is configured only using code-based configuration using GlobalConfiguration class. The Configure method requires a callback method where you have configured your Web API.

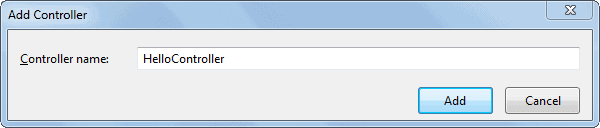
Thus, when an application starts it will call Application\_Start event which in turn calls HelloWebAPIConfig.Register() method to configure your Web API.

Now, let's add Web API controller by right clicking on the Controllers folder -> select Controller.. this will open popup as below.



Add Web API Controller

Select Web API in the left pane and Web API 2 Controller - Empty in the middle pane and click **Add**. This will open another popup to enter the name of your controller as below. Enter controller name and click **Add**.

 Enter Controller Name

This will create the following empty HelloController class in Controllers folder.

Example: Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** HelloWebAPI.Controller{

**public class** HelloController : ApiController{

}

}

Now, we need to add action methods. Here, we will add a simple Get action methods as shown below.

Example: Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** HelloWebAPI.Controller{

**public class** HelloController : ApiController{

**public string** Get(){

**return** "Hello World";

}

}

}

Now, compile and run the project and navigate to http://localhost:xxxx/api/hello in the browser. It will display the following result. (replace xxxx with your local port number)

 Access Web API in the Browser

So, in this way you can create a simple Web API from scratch with config and controller class.

During the development, programmers often need to test and check the result of Web API. Learn how to test Web API for development in the next section.

# **Test Web API**

In the previous section, we created our first simple Web API project. Now we will learn how to test Web API locally to check request & response during development.

We can use the following third-party tools for testing Web API.

* [Fiddler](http://www.telerik.com/fiddler)
* [Postman](https://www.getpostman.com/)

## **Fiddler**

Fiddler is a free debugging proxy for any browser. We can use it to compose and execute different HTTP requests to our Web API and check HTTP response.

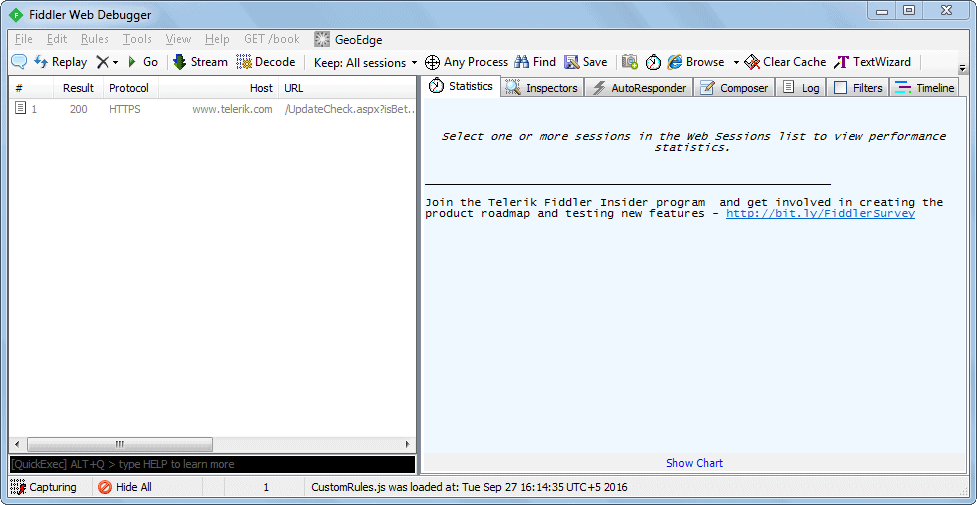
Let's see how to use Fiddler to send an HTTP request to our local Web API and check the response.

**Step 1:**

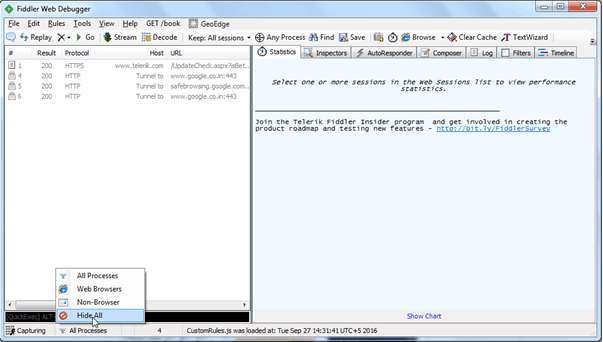
Download and install Fiddler from [here](https://www.telerik.com/download/fiddler).

**Step 2:**

After successful installation click on Fiddler.exe to open Fiddler. It will look like the image below.

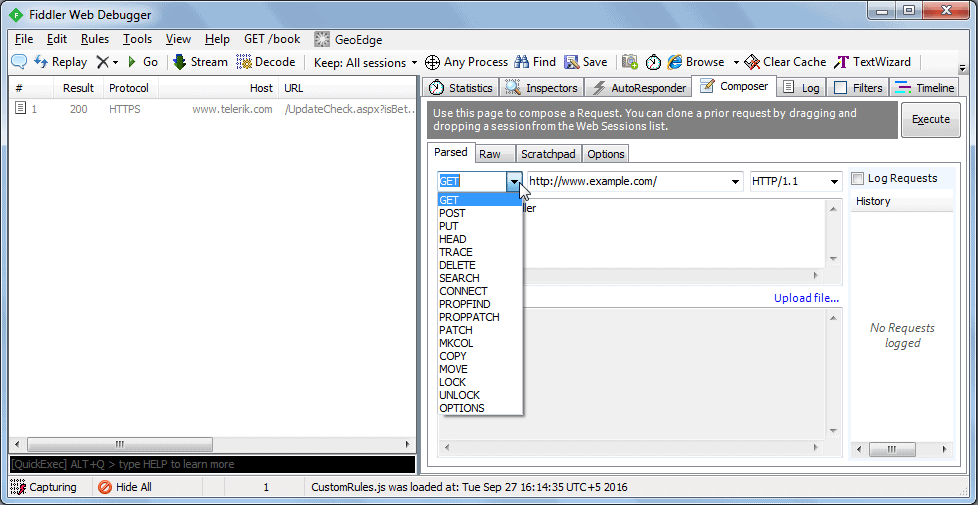
 Fiddler

Fiddler by default captures all processes. We are only interested in intercepting our local process. So, click on **All Processes** at the bottom left corner and select **Hide All**.

 Hide All Processes in Fiddler

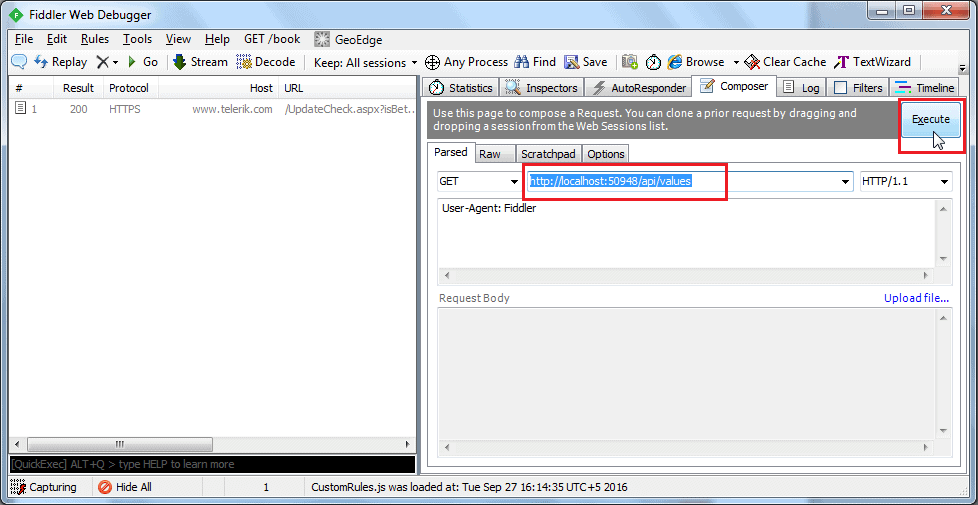
**Step 3:**

Click on the Composer tab. The first tab in the Composer tab is Parsed tab where we can configure HTTP requests and execute it. The first dropdown includes all HTTP Methods. Select a particular HTTP method for the request you want to execute. Here, we will select GET to execute HTTP GET request as shown below.



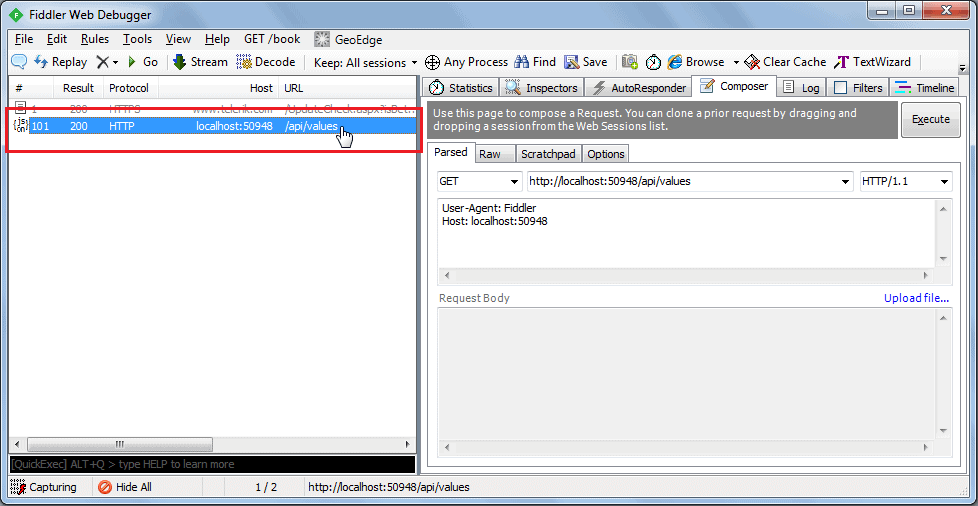
Select HTTP Method

Now, enter the URL of a request in the adjacent textbox. Here, we will execute HTTP request http://localhost:xxxx/api/values to the Web API which we created in the previous section as shown below.

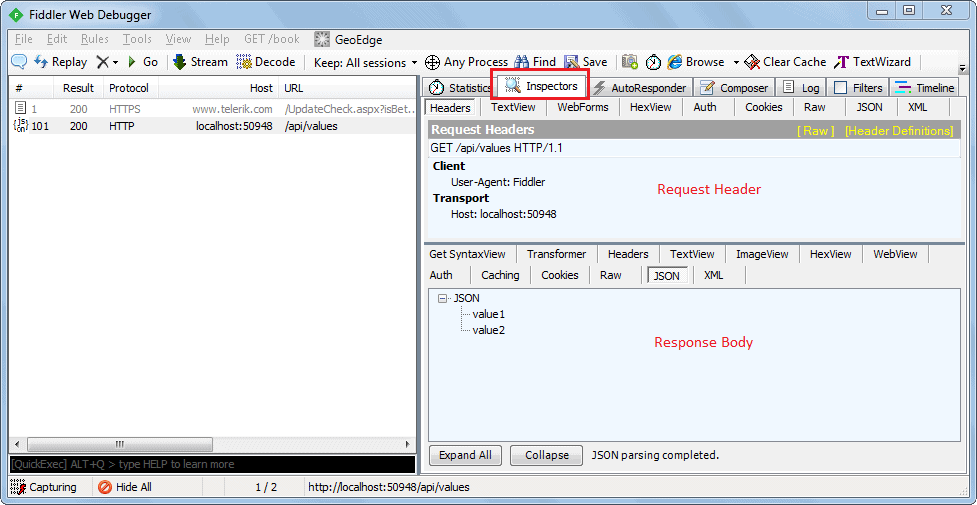


Enter URL and Execute

Click on the Execute button to send this HTTP request and it will immediately display the response in the left pane as shown below.

Response in Fiddler

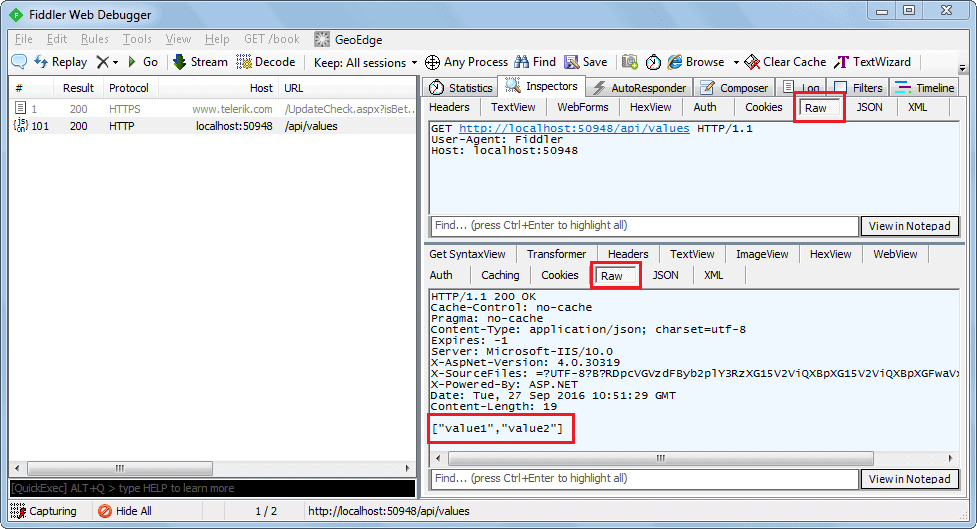
Double click on the result row above to open Inspector tab for the request as shown below.



Fiddler Request & Response

As you can see above, the top pane shows the Request header, and the bottom pane shows the response.

You can also see the raw request header and response by clicking on the Raw tab of request and response as shown below.



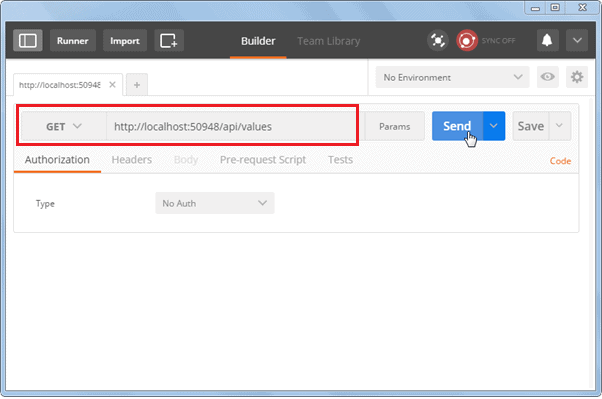
Fiddler Request & Response in Raw Format

You can also see other forms of request and response in Fiddler, but this is the basic way of executing an HTTP request and checking the response.

## **Postman**

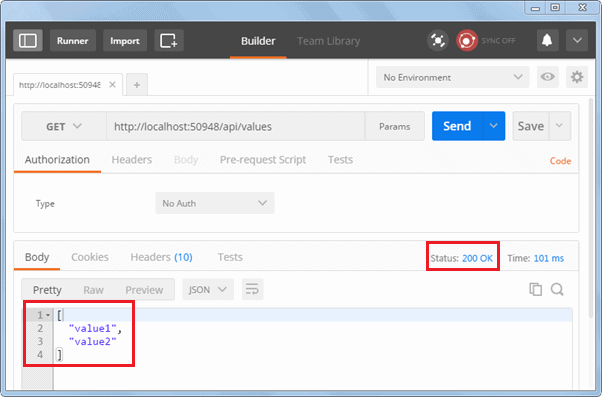
Postman is a free API debugging tool. You can install it on your Chrome browser or Mac. Install it for Chrome from [here](https://chrome.google.com/webstore/detail/postman-rest-client/fhbjgbiflinjbdggehcddcbncdddomop).

After successful installation, open it and select HTTP Method and enter the URL of your Web API as shown below.



Test Web API in Postman

Click on the Send button to send an HTTP request to the provided URL. The response is displayed below.



Response in Postman

As you can see above, HTTP response shows data and response status. Thus, you can use Postman to test your Web API.

We will use Fiddler throughout this training series.

# **Web API Controllers**

We created Web API with MVC project in the previous section where it generated a simple controller. Here, you will learn about Web API Controller in detail.

A Web API Controller is like an ASP.NET MVC controller. It handles incoming HTTP requests and sends responses back to the caller.

Web API controller is a class which can be created under the **Controllers** folder or any other folder under your project's root folder. The name of a controller class must end with "Controller" and it must be derived from System.Web.Http.**ApiController** class. All the public methods of the controller are called action methods.

The following is a simple controller class added by visual studio by default when we created a new Web API project in the Create Web API Project section.

Example: Simple Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** MyWebAPI.Controllers{

**public class** ValuesController : ApiController{

// GET: api/values

**public** IEnumerable<**string**> Get(){

**return new string**[] { "value1", "value2" };

}

// GET: api/values/5

**public string** Get(**int** id){

**return** "value";

}

// POST: api/values

**public void** Post([FromBody]**string value**){

}

// PUT: api/values/5

**public void** Put(**int** id, [FromBody]**string value**){

}

// DELETE: api/values/5

**public void** Delete(**int** id){

}

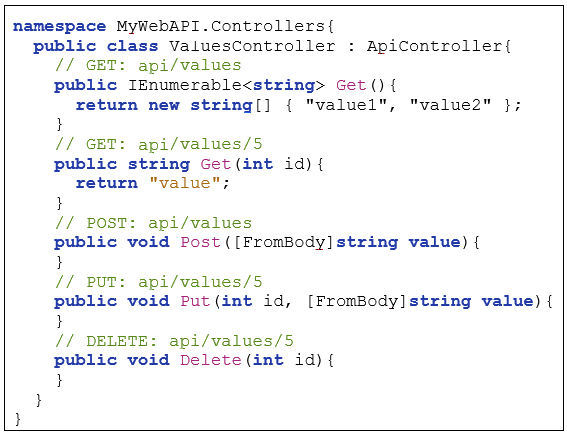
}

}

As you can see in the above example, ValuesController class is derived from ApiController and includes multiple action methods whose names match with HTTP verbs like Get, Post, Put and Delete.

Based on the incoming request URL and HTTP verb (GET/POST/PUT/PATCH/DELETE), Web API decides which Web API controller and action method to execute e.g. Get() method will handle HTTP GET request, Post() method will handle HTTP POST request, Put() mehtod will handle HTTP PUT request and Delete() method will handle HTTP DELETE request for the above Web API.

The following figure illustrates the significance of Web API controller and action methods.



Handles HTTP **DELETE** request

http://localhost:1234/api/values?id=1

Handles HTTP **PUT** request

http://localhost:1234/api/values?id=1

Handles HTTP **POST** request

http://localhost:1234/api/values

Handles HTTP **GET** request with query string

http://localhost:1234/api/values?id=1

Web API controller Base class

Handles HTTP **GET** request

http://localhost:1234/api/values

Web API Controller Overview

If you want to write methods that do not start with an HTTP verb then you can apply the appropriate http verb attribute on the method such as HttpGet, HttpPost, HttpPut etc. same as MVC controller.

Example: Simple Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** MyWebAPI.Controllers{

**public class** ValuesController : ApiController{

[HttpGet]

**public** IEnumerable<**string**> Values(){

**return new string**[] { "value1", "value2" };

}

[HttpGet]

**public string** Value(**int** id){

**return** "value";

}

[HttpPost]

**public void** SaveNewValue([FromBody]**string value**){

}

[HttpPut]

**public void** UpdateValue(**int** id, [FromBody]**string value**){

}

[HttpDelete]

**public void** RemoveValue(**int** id){

}

}

}

## **Web API Controller Characteristics**

1. It must be derived from System.Web.Http.ApiController class.
2. It can be created under any folder in the project's root folder. However, it is recommended to create controller classes in the **Controllers** folder as per the convention.
3. The Action method name can be the same as HTTP verb name or it can start with HTTP verb with any suffix (case in-sensitive) or you can apply Http verb attributes to method.
4. Return type of an action method can be any primitive or complex type. Learn more about it here.

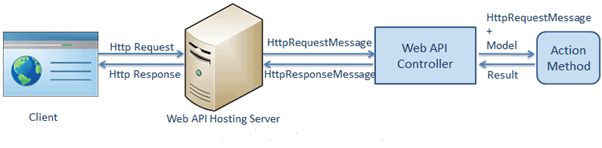
## **Action Method Naming Conventions**

As mentioned above, the name of the action methods in the Web API controller plays an important role. Action method name can be the same as HTTP verbs like Get, Post, Put, Patch or Delete as shown in the Web API Controller example above. However, you can append any suffix with HTTP verbs for more readability. For example, Get method can be GetAllNames(), GetStudents() or any other name which starts with Get.

The following table lists possible action method names for each HTTP method:

| HTTP Method | Possible Web API Action Method Name | Usage |
| --- | --- | --- |
| GET | Get() get() GET() GetAllStudent() \*any name starting with Get \* | Retrieves data. |
| POST | Post() post() POST() PostNewStudent() \*any name starting with Post\* | Inserts new record. |
| PUT | Put() put() PUT() PutStudent() \*any name starting with Put\* | Updates existing record. |
| PATCH | Patch() patch() PATCH() PatchStudent() \*any name starting with Patch\* | Updates record partially. |
| DELETE | Delete() delete() DELETE() DeleteStudent() \*any name starting with Delete\* | Deletes record. |

The following figure illustrates the overall request/response pipeline.



Web API Request Pipeline

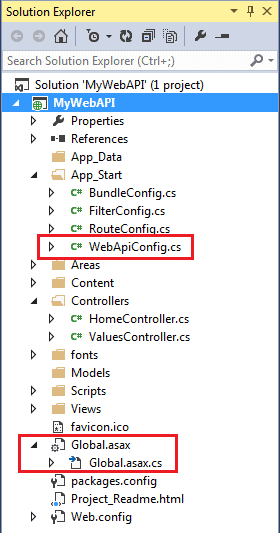
## **Difference between Web API and MVC controller**

| Web API Controller | MVC Controller |
| --- | --- |
| Derives from System.Web.Http.ApiController class | Derives from System.Web.Mvc.Controller class. |
| Method name must start with Http verbs otherwise apply http verbs attribute. | Must apply appropriate Http verbs attribute. |
| Specialized in returning data. | Specialized in rendering view. |
| Return data automatically formatted based on Accept-Type header attribute. Default to json or xml. | Returns ActionResult or any derived type. |
| Requires .NET 4.0 or above | Requires .NET 3.5 or above |

**Configure Web API**

Web API supports code-based configuration. It cannot be configured in web.config file. We can configure Web API to customize the behavior of Web API hosting infrastructure and components such as routes, formatters, filters, DependencyResolver, MessageHandlers, ParamterBindingRules, properties, services etc.

We created a simple Web API project in the Create Web API Project section. The Web API project includes default WebApiConfig class in the App\_Start folder and also includes Global.asax as shown below.

 Configure Web API

Global.asax

**public class** WebAPIApplication : System.Web.HttpApplication{

**protected void** Application\_Start(){

GlobalConfiguration.Configure(WebApiConfig.Register);

//other configuration

}

}

WebApiConfig

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

);

// configure additional webapi settings here..

}

}

Web API configuration process starts when the application starts. It calls GlobalConfiguration.Configure(WebApiConfig.Register) in the Application\_Start method. The Configure method requires the callback method where Web API has been configured in code. By default, this is the static WebApiConfig.Register() method.

As you can see above, WebApiConfig.Register() method includes a parameter of HttpConfiguration type which is then used to configure the Web API. The HttpConfiguration is the main class which includes the following properties using which you can override the default behaviour of Web API.

| Property | Description |
| --- | --- |
| DependencyResolver | Gets or sets the dependency resolver for dependency injection. |
| Filters | Gets or sets the filters. |
| Formatters | Gets or sets the media-type formatters. |
| IncludeErrorDetailPolicy | Gets or sets a value indicating whether error details should be included in error messages. |
| MessageHandlers | Gets or sets the message handlers. |
| ParameterBindingRules | Gets the collection of rules for how parameters should be bound. |
| Properties | Gets the properties associated with this Web API instance. |
| Routes | Gets the collection of routes configured for the Web API. |
| Services | Gets the Web API services. |

Visit MSDN to learn about all the members of [HttpConfiguration](https://msdn.microsoft.com/en-us/library/system.web.http.httpconfiguration(v=vs.118).aspx" \t "_blank).

Learn how to configure Web API routes in the next section.

# **Web API Routing**

In the previous section, we learned that Web API can be configured in WebApiConfig class. Here, we will learn how to configure Web API routes.

Web API routing is like ASP.NET MVC Routing. It routes an incoming HTTP request to a particular action method on a Web API controller.

Web API supports two types of routing:

1. Convention-based Routing
2. Attribute Routing

## **Convention-based Routing**

In convention-based routing, Web API uses route templates to determine which controller and action method to execute. At least one route template must be added to the route table to handle various HTTP requests.

When we created Web API project using WebAPI template in the Create Web API Project section, it also added WebApiConfig class in the App\_Start folder with default route as shown below.

Example: WebApiConfig with Default Route

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

// Enable attribute routing

config.MapHttpAttributeRoutes();

// Add default route using convention-based routing

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

);

}

}

In the above WebApiConfig.Register() method, config.MapHttpAttributeRoutes() enables attribute routing which we will learn later in this section. The config.Routes is a route table or route collection of type [HttpRouteCollection](https://msdn.microsoft.com/en-us/library/system.web.http.httproutecollection(v=vs.118).aspx" \t "_blank). The "DefaultApi" route is added in the route table using [MapHttpRoute()](https://msdn.microsoft.com/en-us/library/hh835153(v=vs.118).aspx" \t "_blank) extension method. The MapHttpRoute() extension method internally creates a new instance of [IHttpRoute](https://msdn.microsoft.com/en-us/library/system.web.http.routing.ihttproute(v=vs.118).aspx" \t "_blank) and adds it to an HttpRouteCollection. However, you can create a new route and add it into a collection manually as shown below.

Example: Add Default Route

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

config.MapHttpAttributeRoutes();

// define route

IHttpRoute defaultRoute =

config.Routes.CreateRoute("api/{controller}/{id}",

**new** { id = RouteParameter.Optional }, **null**);

// Add route

config.Routes.Add("DefaultApi", defaultRoute);

}

}

The following table lists parameters of MapHttpRoute() method.

| Parameter | Description |
| --- | --- |
| name | Name of the route |
| routeTemplate | URL pattern of the route |
| defaults | An object parameter that includes default route values |
| constraints | Regex expression to specify characteristic of route values |
| handler | The handler to which the request will be dispatched. |

Now, let's see how Web API handles an incoming http request and sends the response.

The following is a sample HTTP GET request.

Sample HTTP GET Request

GET http://localhost:1234/api/values/ HTTP/1.1

User-Agent: Fiddler

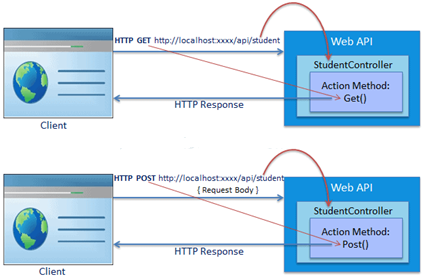
Host: localhost: 60464

Content-Type: application/json

Considering the DefaultApi route configured in the above WebApiConfig class, the above request will execute Get action method of the ValuesController because HTTP method is a GET and URL is http://localhost:1234/api/values which matches with DefaultApi's route template /api/{controller}/{id} where value of {controller} will be ValuesController. The default route has specified id as an optional parameter so if an id is not present in the url then {id} will be ignored. The request's HTTP method is GET so it will execute Get action method of ValueController.

If the Web API framework does not find matched routes for an incoming request, then it will send 404 error response.

The following figure illustrates Web API Routing.



Web API Routing

The following table displays which action method and controller will be executed on different incoming requests.

| Request URL | Request HTTP Method | Action method | Controller |
| --- | --- | --- | --- |
| http://localhost:1234/api/course | GET | Get() | CourseController |
| http://localhost:1234/api/product | POST | Post() | ProductController |
| http://localhost:1234/api/teacher | PUT | Put() | TeacherController |

 Note:

Web API also supports routing same as ASP.NET MVC by including action method name in the URL.

### **Configure Multiple Routes**

We configured a single route above. However, you can configure multiple routes in the Web API using HttpConfiguration object. The following example demonstrates configuring multiple routes.

Example: Multiple Routes

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

config.MapHttpAttributeRoutes();

// school route

config.Routes.MapHttpRoute(

**name**: "School",

**routeTemplate**: "api/myschool/{id}",

**defaults**: **new** { controller="school", id = RouteParameter.Optional }

**constraints**: **new** { id ="/d+" }

);

// default route

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

}

}

In the above example, School route is configured before DefaultApi route. So, any incoming request will be matched with the School route first and if incoming request url does not match with it then only it will be matched with DefaultApi route. For example, request url is http://localhost:1234/api/myschool is matched with School route template, so it will be handled by SchoolController.

Note: The reason to use api in the route template is just to avoid confusion between MVC controller and Web API controller. You can use any pattern based on your app architecture.

Visit asp.net to learn about [routing](https://www.asp.net/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api) in detail.

## **Attribute Routing**

Attribute routing is supported in Web API 2. As the name implies, attribute routing uses [Route()] attribute to define routes. The Route attribute can be applied on any controller or action method.

In order to use attribute routing with Web API, it must be enabled in WebApiConfig by calling config.MapHttpAttributeRoutes() method.

Consider the following example of attribute routing.

Example: Attribute Routing

**public class** StudentController : ApiController{

[Route("api/student/names")]

**public** IEnumerable<**string**> Get(){

**return new string**[] { "student1", "student2" };

}

}

In the above example, the Route attribute defines new route "api/student/names" which will be handled by the Get() action method of StudentController. Thus, an HTTP GET request http://localhost:1234/api/student/names will return list of student names.

Visit asp.net to learn about [attribute routing](https://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2) in detail.

# **Parameter Binding**

In the previous section we learned how Web API routes HTTP request to a controller and action method. Here, we will learn how Web API binds HTTP request data to the parameters of an action method.

Action methods in Web API controller can have one or more parameters of different types. It can be either primitive type or complex type. Web API binds action method parameters either with URL's query string or with request body depending on the parameter type. By default, if parameter type is of .NET primitive type such as int, bool, double, string, GUID, DateTime, decimal or any other type that can be converted from string type then it sets the value of a parameter from the query string. And if the parameter type is complex type, then Web API tries to get the value from request body by default.

The following table lists the default rules for parameter binding.

| HTTP Method | Query String | Request Body |
| --- | --- | --- |
| GET | Primitive Type, Complex Type | NA |
| POST | Primitive Type | Complex Type |
| PUT | Primitive Type | Complex Type |
| PATCH | Primitive Type | Complex Type |
| DELETE | Primitive Type, Complex Type | NA |

Let's see how Web API get values of action method parameters from HTTP request.

## **Get Action Method with Primitive Parameter**

Consider the following example of Get action method that includes single primitive type parameter.

Example: Primitive Parameter Binding

**public class** StudentController : ApiController{

**public** Student Get(**int** id){

}

}

As you can see above, the Get action method includes id parameter of int type. So, Web API will try to extract the value of id from the query string of requested URL, convert it into int and assign it to id parameter of Get action method. For example, if an HTTP request is http://localhost/api/student?id=1 then value of id parameter will be 1.

Followings are valid HTTP GET Requests for the above action method.

http://localhost/api/student?id=1

http://localhost/api/student?ID=1

 Note:

Query string parameter name and action method parameter name must be the same (case-insensitive). If names do not match, then values of the parameters will not be set. The order of the parameters can be different.

## **Multiple Primitive Parameters**

Consider the following example of Get action method with multiple primitive parameters.

Example: Multiple Parameters Binding

**public class** StudentController : ApiController{

**public** Student Get(**int** id, **string** name){

}

}

As you can see above, the Get method includes multiple primitive type parameters. So, Web API will try to extract the values from the query string of request URL. For example, if an HTTP request is http://localhost/api/student?id=1&name=steve then value of id parameter will be 1 and name will be "steve".

Followings are valid HTTP GET Requests for the above action method.

http://localhost/api/student?id=1&name=steve

http://localhost/api/student?ID=1&NAME=steve

http://localhost/api/student?name=steve&id=1

 Note:

Query string parameter names must match with the name of an action method parameter. However, they can be in different order.

## **POST Action Method with Primitive Parameter**

HTTP POST request is used to create new resources. It can include request data into HTTP request body and in query string.

Consider the following Post action method.

Example: Post Method with Primitive Parameter

**public class** StudentController : ApiController{

**public** Student Post(id id, **string** name){

}

}

As you can see above, the Post action method includes primitive type parameters id and name. So, by default, Web API will get values from the query string. For example, if an HTTP POST request is http://localhost/api/student?id=1&name=steve then the value of id will be 1 and name will be "steve" in the above the Post method.

Now, consider the following the Post method with complex type parameter.

Example: Post Method with Complex Type Parameter

**public class** Student{

**public int** Id { get; set; }

**public string** Name { get; set; }

}

**public class** StudentController : ApiController{

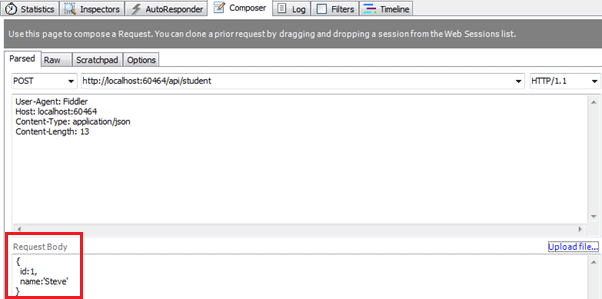
**public** Student Post(Student stud){

}

}

The above Post method includes Student type parameter. So, as a default rule, Web API will try to get the values of stud parameter from HTTP request body.

Following is a valid HTTP POST request in the fiddler for the above action method.



Parameter Binding

Web API will extract the JSON object from the Request body above and convert it into Student object automatically because names of JSON object properties match with the name of Student class properties (case-insensitive).

## **POST Method with Mixed Parameters**

Post action method can include primitive and complex type parameters. Consider the following example.

Example: Post Method with Primitive and Complex Type Parameters

**public class** Student{

**public int** Id { get; set; }

**public string** Name { get; set; }

}

**public class** StudentController : ApiController{

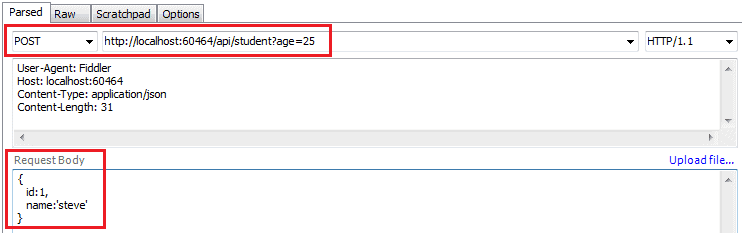
**public** Student Post(**int** age, Student student){

}

}

The above Post method includes both primitive and complex type parameters. So, by default, Web API will get the id parameter from query string and student parameter from the request body.

Following is a valid HTTP POST request in the fiddler for the above action method.



Parameter Binding

 Note:

Post action method cannot include multiple complex type parameters because at most one parameter is allowed to be read from the request body.

Parameter binding for Put and Patch method will be the same as Post method in Web API.

## **[FromUri] and [FromBody]**

You have seen that by default Web API gets the value of a primitive parameter from the query string and complex type parameter from the request body. But what if we want to change this default behavior?

Use [FromUri] attribute to force Web API to get the value of complex type from the query string and [FromBody] attribute to get the value of primitive type from the request body, opposite to the default rules.

For example, consider the following Get method.

Example: FormUri

**public class** StudentController : ApiController{

**public** Student Get([FromUri] Student stud){

}

}

In the above example, the Get method includes complex type parameter with [FromUri] attribute. So, Web API will try to get the value of Student type parameter from the query string. For example, if an HTTP GET request http://localhost:xxxx/api/student?id=1&name=steve then Web API will create Student object and set its id and name property values to the value of id and name query string.

 Note:

Name of the complex type properties and query string parameters must match.

The same way, consider the following example of the Post method.

Example: FromUri

**public class** StudentController : ApiController{

**public** Student Post([FromUri]Student stud){

}

}

As you can see above, we have applied [FromUri] attribute with the Student parameter. The Web API by default extracts the value of complex type from request body but here we have applied [FromUri] attribute. So now, Web API will extract the value of Student properties from the query string instead of request body.

The same way, apply [FromBody] attribute to get the value of primitive data type from the request body instead of query string, as shown below.

Example: FromBody

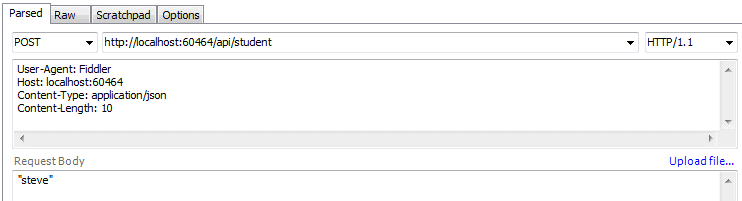
**public class** StudentController : ApiController{

**public** Student Post([FromBody]**string** name){

}

}

Following is a valid HTTP POST request in the fiddler for the above action method.

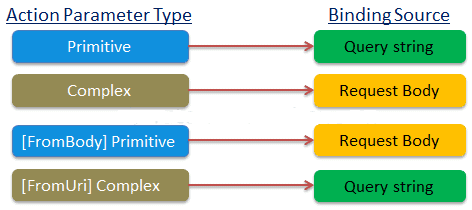


Parameter Binding

 Note:

FromBody attribute can be applied on only one primitive parameter of an action method. It cannot be applied on multiple primitive parameters of the same action method.

The following figure summarizes parameter binding rules.



Web API Parameter Bindings

# **Action Method Return Type**

In the previous section, you learned about parameter binding with Web API action method. Here, you will learn about the return types of action methods which in turn will be embedded in the Web API response sent to the client.

The Web API action method can have the following return types.

1. Void
2. Primitive type or Complex type
3. HttpResponseMessage
4. IHttpActionResult

## **Void**

It's not necessary that all action methods must return something. It can have void return type.

For example, consider the following Delete action method that just deletes the student from the data source and returns nothing.

Example: Void Return Type

**public class** StudentController : ApiController{

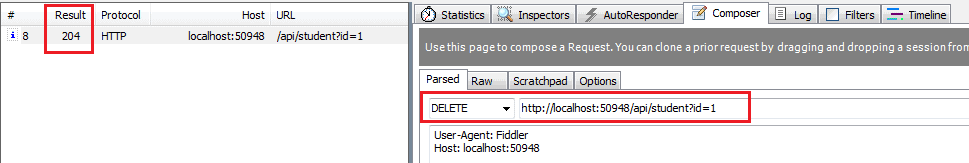
**public void** Delete(**int** id){

DeleteStudentFromDB(id);

}

}

As you can see above Delete action method returns void. It will send 204 "No Content" status code as a response when you send HTTP DELETE request as shown below.

Void Response Status

## **Primitive or Complex Type**

An action method can return primitive or other custom complex types as other normal methods.

Consider the following Get action methods.

Example: Primitive or Complex Return Type

**public class** Student{

**public int** Id { get; set; }

**public string** Name { get; set; }

}

**public class** StudentController : ApiController{

**public int** GetId(**string** name){

**int** id = GetStudentId(name);

**return** id;

}

**public** Student GetStudent(**int** id){

**var** student = GetStudentFromDB(id);

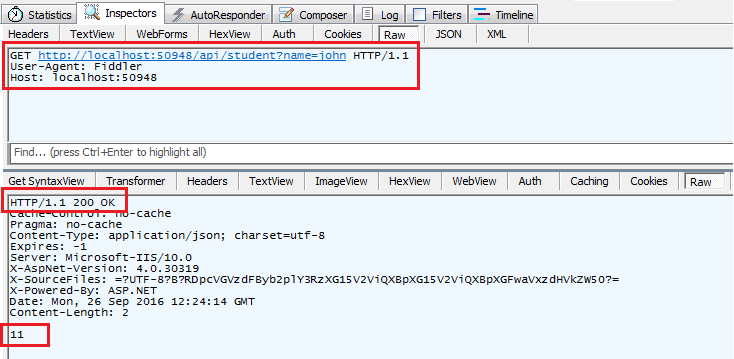
**return** student;

}

}

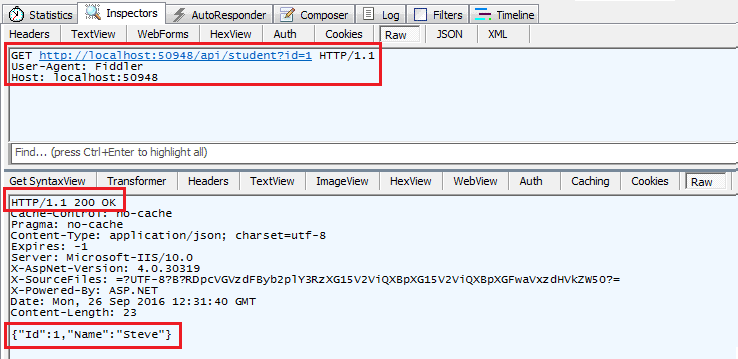
As you can see above, GetId action method returns an integer and GetStudent action method returns a Student type.

An HTTP GET request http://localhost:xxxx/api/student?name=john will return following response in Fiddler.



Primitive Return Type in Response

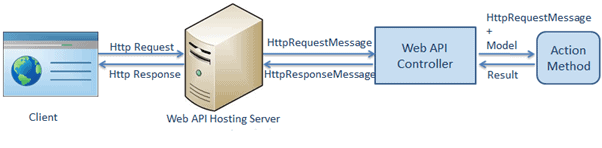
An HTTP GET request http://localhost:xxxx/api/student?id=1 will return following response in Fiddler.



Complex Return Type in Response

## **HttpResponseMessage**

Web API controller always returns an object of HttpResponseMessage to the hosting infrastructure. The following figure illustrates the overall Web API request/response pipeline.

Web API Request Pipeline

Visit [Web API HTTP Message Life Cycle Poster](https://www.asp.net/media/4071077/aspnet-web-api-poster.pdf) for more details.

As you can see in the above figure, the Web API controller returns HttpResponseMessage object. You can also create and return an object of HttpResponseMessage directly from an action method.

The advantage of sending HttpResponseMessage from an action method is that you can configure a response your way. You can set the status code, content or error message (if any) as per your requirement.

Example: Return HttpResponseMessage

**public** HttpResponseMessage Get(**int** id){

Student stud = GetStudentFromDB(id);

**if** (stud == **null**){

**return** Request.CreateResponse(HttpStatusCode.NotFound, id);

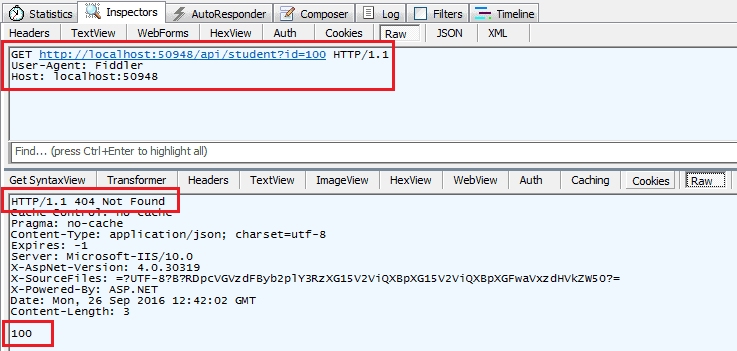
}

**return** Request.CreateResponse(HttpStatusCode.OK, stud);

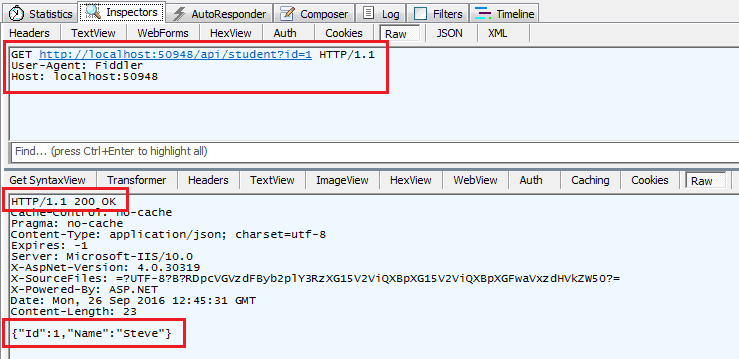
}

In the above action method, if there is no student with specified id in the DB then it will return HTTP 404 Not Found status code, otherwise it will return 200 OK status with student data.

For example, an http GET request http://localhost:xxxx/api/student?id=100 will get following response considering student with id=100 does not exist in the DB.

 Web API Response in Fiddler

The same way, an HTTP GET request http://localhost:60464/api/student?id=1 will get following response considering student with id=1 exists in the database.

 Web API Response in Fiddler

## **IHttpActionResult**

The *IHttpActionResult* was introduced in Web API 2 (.NET 4.5). An action method in Web API 2 can return an implementation of IHttpActionResult class which is like ActionResult class in ASP.NET MVC.

You can create your own class that implements IHttpActionResult or use various methods of ApiController class that returns an object that implements the IHttpActionResult.

Example: Return IHttpActionResult Type using Ok() and NotFound() Methods

**public** IHttpActionResult Get(**int** id){

Student stud = GetStudentFromDB(id);

**if** (stud == **null**)**return** NotFound();

**return** Ok(stud);

}

In the above example, if student with specified id does not exists in the database, then it will return response with the status code 404 otherwise it sends student data with status code 200 as a response. As you can see, we don't have to write much code because NotFound() and Ok() method does it all for us.

The following table lists all the methods of ApiController class that returns an object of a class that implements IHttpActionResult interface.

| **ApiController Method** | **Description** |
| --- | --- |
| BadRequest() | Creates a BadRequestResult object with status code 400. |
| Conflict() | Creates a ConflictResult object with status code 409. |
| Content() | Creates a NegotiatedContentResult with the specified status code and data. |
| Created() | Creates a CreatedNegotiatedContentResult with status code 201 Created. |
| CreatedAtRoute() | Creates a CreatedAtRouteNegotiatedContentResult with status code 201 created. |
| InternalServerError() | Creates an InternalServerErrorResult with status code 500 Internal server error. |
| NotFound() | Creates a NotFoundResult with status code404. |
| Ok() | Creates an OkResult with status code 200. |
| Redirect() | Creates a RedirectResult with status code 302. |
| RedirectToRoute() | Creates a RedirectToRouteResult with status code 302. |
| ResponseMessage() | Creates a ResponseMessageResult with the specified HttpResponseMessage. |
| StatusCode() | Creates a StatusCodeResult with the specified http status code. |
| Unauthorized() | Creates an UnauthorizedResult with status code 401. |

Visit MSDN to know all the members of [ApiController](https://msdn.microsoft.com/en-us/library/system.web.http.apicontroller(v=vs.118).aspx" \t "_blank).

## **Create Custom Result Type**

You can create your own custom class as a result type that implements IHttpActionResult interface.

The following example demonstrates implementing IHttpActionResult class.

Example: Create Custom Result Type

**public class** TextResult : IHttpActionResult{

**string** \_value;

HttpRequestMessage \_request;

**public** TextResult(**string value**, HttpRequestMessage request){

\_value = **value**;

\_request = request;

}

**public** Task<HttpResponseMessage>

ExecuteAsync(CancellationToken cancellationToken){

**var** response = **new** HttpResponseMessage(){

Content = **new** StringContent(\_value),

RequestMessage = \_request

};

**return** Task.FromResult(response);

}

}

Now, you can return TextResult object from the action method as shown below.

Example: Return Custom Result Type

**public** IHttpActionResult GetName(**int** id){

**string** name = GetStudentName(id);

**if** (String.IsNullOrEmpty(name)){

**return** NotFound();

}

**return new** TextResult(name, Request);

}

# **Web API Request/Response Data Formats**

Here, you will learn how Web API handles different formats of request and response data.

## **Media Type**

Media type (aka MIME type) specifies the format of the data as type/subtype e.g. text/html, text/xml, application/json, image/jpeg etc.

In HTTP request, MIME type is specified in the request header using **Accept** and **Content-Type** attributes. The Accept header attribute specifies the format of response data which the client expects, and the Content-Type header attribute specifies the format of the data in the request body so that receiver can parse it into appropriate format.

For example, if a client wants response data in JSON format, then it will send the following GET HTTP request with Accept header to the Web API.

HTTP GET Request:

GET http://localhost:60464/api/student HTTP/1.1

User-Agent: Fiddler

Host: localhost:1234

**Accept: application/json**

The same way, if a client includes JSON data in the request body to send it to the receiver, then it will send following POST HTTP request with Content-Type header with JSON data in the body.

HTTP POST Request:

POST http://localhost:60464/api/student?age=15 HTTP/1.1

User-Agent: Fiddler

Host: localhost:60464

**Content-Type: application/json**

Content-Length: 13

{

id:1,

name:'Steve'

}

Web API converts request data into CLR object and serialize CLR object into response data based on Accept and Content-Type headers. Web API includes built-in support for JSON, XML, BSON, and form-urlencoded data. It means it automatically converts request/response data into these formats OOB (out-of the box).

Example: Post Action Method

**public class** Student{

**public int** Id { get; set; }

**public string** Name { get; set; }

}

**public class** StudentController : ApiController{

**public** Student Post(Student student){

// save student into db

**var** insertedStudent = SaveStudent(student);

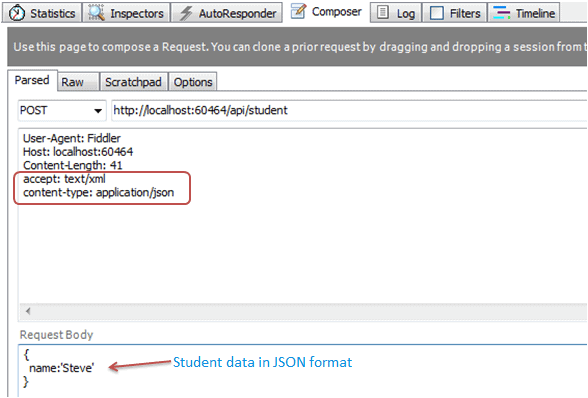
**return** insertedStudent;

}

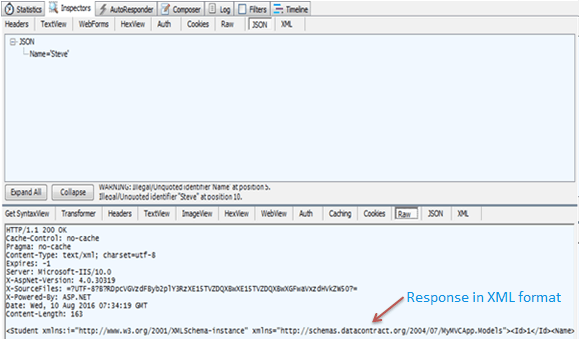
}

As you can see above, the Post action method accepts Student type parameter, saves that student into DB and returns inserted student with generated id. The above Web API handles HTTP POST request with JSON or XML data and parses it to a Student object based on Content-Type header value and the same way it converts insertedStudent object into JSON or XML based on Accept header value.

The following figure illustrates HTTP POST request in fiddler.

 Request-Response Data Format

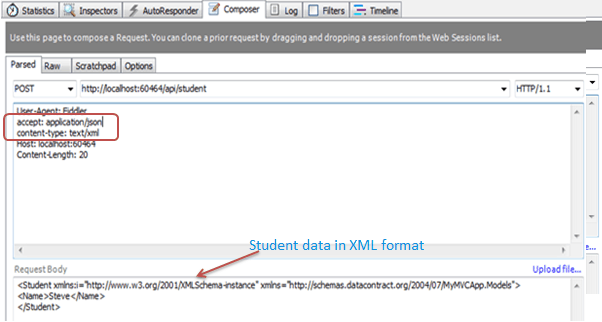
In the above figure, the Accept header specifies that it expects response data in XML format and Content-Type specifies that the student data into request body is in the JSON format. The following is the response upon execution of the above request.



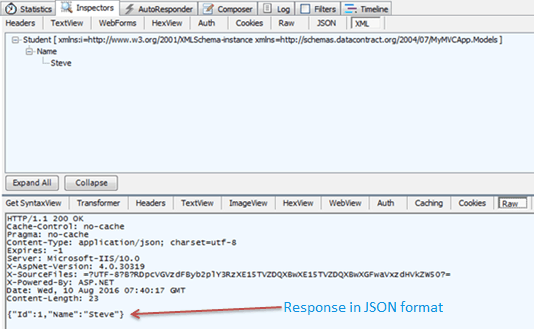
Request-Response Data Format

The same way, you can specify different request & response format using accept and content-type headers and Web API will handle them without any additional changes.

The following HTTP POST request sends data in XML format and receives data in JSON format.

 Web API Request

The above HTTP POST request will get the following response upon execution.

 Web API Response

Thus, Web API handles JSON and XML data by default. Learn how Web API formats request/response data using formatters in the next section.

# **ASP.NET Web API: Media-Type Formatters**

As you have seen in the previous section that Web API handles JSON and XML formats based on Accept and Content-Type headers. But how does it handle these different formats? The answer is: By using Media-Type formatters.

Media type formatters are classes responsible for serializing request/response data so that Web API can understand the request data format and send data in the format which client expects.

Web API includes following built-in media type formatters.

| Media Type Formatter Class | MIME Type | Description |
| --- | --- | --- |
| JsonMediaTypeFormatter | application/json, text/json | Handles JSON format |
| XmlMediaTypeFormatter | application/xml, text/json | Handles XML format |
| FormUrlEncodedMediaTypeFormatter | application/x-www-form-urlencoded | Handles HTML form URL-encoded data |
| JQueryMvcFormUrlEncodedFormatter | application/x-www-form-urlencoded | Handles model-bound HTML form URL-encoded data |

## **Retrieve Built-in Media Type Formatters**

As mentioned, Web API includes above listed media type formatter classes by default. However, you can also add, remove or change the order of formatters.

The following example demonstrates HTTP Get method that returns all built-in formatter classes.

Example: Retrieve Built-in Formatters in C#

**public class** FormattersController : ApiController{

**public** IEnumerable<**string**> Get(){

IList<**string**> formatters = **new** List<**string**>();

**foreach** (**var** item **in** GlobalConfiguration.Configuration.Formatters){

formatters.Add(item.ToString());

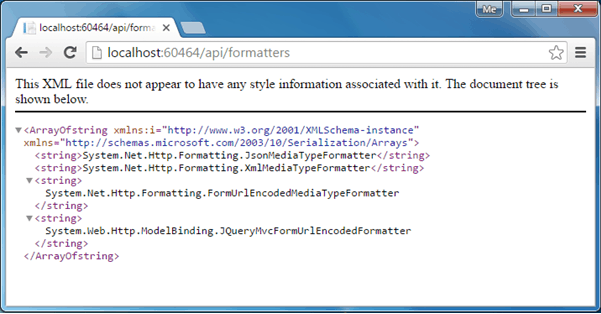
}

**return** formatters.AsEnumerable<**string**>();

}

}

In the above example, GlobalConfiguration.Configuration.Formatters returns MediaTypeFormatterCollection that includes all the formatter classes. The above example returns names of all the formatter classes as shown below.



Built-in Media-Type Formatters

Alternatively, MediaTypeFormatterCollection class defines convenience properties that provide direct access to three of the four built-in media type formatters. The following example demonstrates retrieving media type formatters using MediaTypeFormatterCollection's properties.

Example: Retrieve Built-in Formatters in C#

**public class** FormattersController : ApiController {

**public** IEnumerable<**string**> Get(){

IList<**string**> formatters = **new** List<**string**>();

formatters.Add(GlobalConfiguration.Configuration.Formatters.

JsonFormatter.GetType().FullName);

formatters.Add(GlobalConfiguration.Configuration.Formatters.

XmlFormatter.GetType().FullName);

formatters.Add(GlobalConfiguration.Configuration.Formatters.

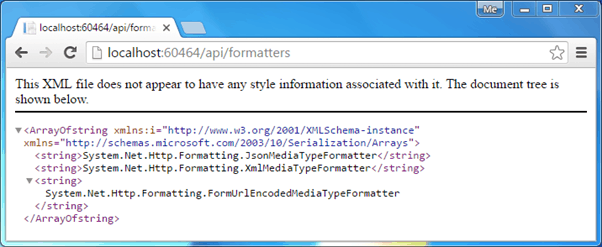
FormUrlEncodedFormatter.GetType().FullName);

**return** formatters.AsEnumerable<**string**>();

}

}

The above example returns following response to the browser.



Media-Type Formatters

### **BSON Formatter**

Web API also supports BSON format. As the name suggests, BSON is binary JSON, it is a binary-encoded serialization of JSON-like documents. Currently there is very little support for BSON and no JavaScript implementation is available for clients running browsers. This means that it is not possible to retrieve and automatically parse BSON data to JavaScript objects.

Web API includes built-in formatter class BsonMediaTypeFormatter for BSON but it is **disabled** by default. Learn more about BSON support in Web API [here](https://www.asp.net/web-api/overview/formats-and-model-binding/bson-support-in-web-api-21).

### **JSON Formatter**

As mentioned above, Web API includes JsonMediaTypeFormatter class that handles JSON format. The JsonMediaTypeFormatter converts JSON data in an HTTP request into CLR objects (object in C# or VB.NET) and converts CLR objects into JSON format that is embeded within HTTP response.

Internally, JsonMediaTypeFormatter uses third-party open-source library called [JSON.NET](https://json.codeplex.com/) to perform serialization.

### **Configure JSON Serialization**

JSON formatter can be configured in WebApiConfig class. The JsonMediaTypeFormatter class includes various properties and methods using which you can customize JSON serialization. For example, Web API writes JSON property names with PascalCase by default. To write JSON property names with camelCase, set the CamelCasePropertyNamesContractResolver on the serializer settings as shown below.

Example: Customize JSON Serialization in C#

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

);

// configure json formatter

JsonMediaTypeFormatter jsonFormatter = config.Formatters.JsonFormatter;

jsonFormatter.SerializerSettings.ContractResolver =

**new** CamelCasePropertyNamesContractResolver();

}

}

### **XML Formatter**

The XmlMediaTypeFormatter class is responsible for serializing model objects into XML data. It uses System.Runtime.DataContractSerializer class to generate XML data.

Learn more about configuring JSON and XML serialization [here](https://www.asp.net/web-api/overview/formats-and-model-binding/json-and-xml-serialization).

# **Web API Filters**

Web API includes filters to add extra logic before or after the action method execution. Filters can be used to provide cross-cutting features such as logging, exception handling, performance measurement, authentication and authorization.

Filters are attributes that can be applied on the Web API controller or one or more action methods. Every filter attribute class must implement IFilter interface included in System.Web.Http.Filters namespace. However, System.Web.Http.Filters includes other interfaces and classes that can be used to create filters for specific purposes.

The following table lists important interfaces and classes that can be used to create Web API filters.

| Filter Type | Interface | Class | Description |
| --- | --- | --- | --- |
| Simple Filter | IFilter | - | Defines the methods that are used in a filter |
| Action Filter | IActionFilter | ActionFilterAttribute | Used to add extra logic before or after the action methods execute. |
| Authentication Filter | IAuthenticationFilter | - | Used to force users or clients to be authenticated before action methods execute. |
| Authorization Filter | IAuthorizationFilter | AuthorizationFilterAttribute | Used to restrict access to action methods to specific users or groups. |
| Exception Filter | IExceptionFilter | ExceptionFilterAttribute | Used to handle all unhandled exception in Web API. |
| Override Filter | IOverrideFilter | - | Used to customize the behavior of other filter for individual action method. |

As you can see, the above table includes class as well as interface for some of the filter types. Interfaces include methods that must be implemented in your custom attribute class whereas filter class has already implemented necessary interfaces and provides virtual methods, so that they can be overridden to add extra logic. For example, ActionFilterAttribute class includes methods that can be overridden. We just need to override methods which we are interested in, whereas if you use IActionFilter attribute then you must implement all the methods.

Visit MSDN to know all the classes and interfaces available in [System.Web.Http.Filters](https://msdn.microsoft.com/en-us/library/system.web.http.filters(v=vs.118).aspx" \t "_blank).

Let's create a simple LogAttribute class for logging purpose to demonstrate action filter.

First, create a LogAttribute class derived from ActionFilterAttribute class as shown below.

Example: Web API Filter Class

**public class** LogAttribute : ActionFilterAttribute{

**public** LogAttribute(){

}

**public override void** OnActionExecuting(HttpActionContext actionContext){

Trace.WriteLine(**string**.Format("Action Method {0} executing at {1}",

actionContext.ActionDescriptor.ActionName,

DateTime.Now.ToShortDateString()), "Web API Logs");

}

**public override void** OnActionExecuted(

HttpActionExecutedContext actionExecutedContext){

Trace.WriteLine(**string**.Format("Action Method {0} executed at {1}",

actionExecutedContext.ActionContext.ActionDescriptor.ActionName,

DateTime.Now.ToShortDateString()), "Web API Logs");

}

}

In the above example, LogAttribute is derived from ActionFilterAttribute class and overrided OnActionExecuting and OnActionExecuted methods to log in the trace listeners. (You can use your own logging class to log in textfile or other medium.)

Another way of creating LogAttribute class is by implementing IActionFilter interface and deriving Attribute class as shown below.

Example: Web API Filter Class

**public class** LogAttribute : Attribute, IActionFilter{

**public** LogAttribute(){ }

**public** Task<HttpResponseMessage> ExecuteActionFilterAsync(

HttpActionContext actionContext, CancellationToken cancellationToken,

Func<Task<HttpResponseMessage>> continuation){

Trace.WriteLine(**string**.Format("Action Method {0} executing at {1}",

actionContext.ActionDescriptor.ActionName,

DateTime.Now.ToShortDateString()), "Web API Logs");

**var** result = continuation();

result.Wait();

Trace.WriteLine(**string**.Format("Action Method {0} executed at {1}",

actionContext.ActionDescriptor.ActionName,

DateTime.Now.ToShortDateString()), "Web API Logs");

**return** result;

}

**public bool** AllowMultiple{ **get** { return true; } }

}

In the above example, deriving from Attribute class makes it an attribute and implementing IActionFilter makes LogAttribute class as action filter. So now, you can apply [Log] attributes on controllers or action methods as shown below.

Example: Apply Web API Filter on Controller

[Log]

**public class** StudentController : ApiController{

**public** StudentController(){

}

**public** Student Get(){

//provide implementation

}

}

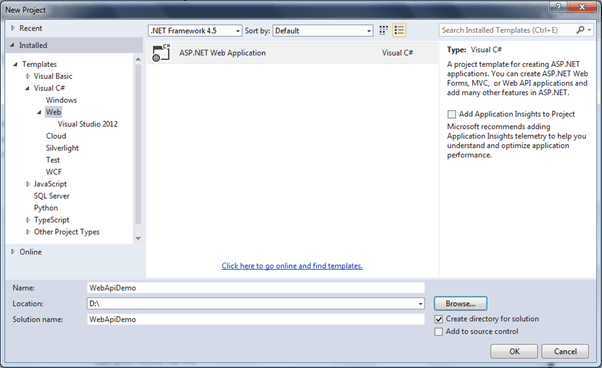
So now, it will log all the requests handled by the above StudentController. Thus, you can create filters for cross-cutting concerns.

# **Create Web API for CRUD operation - Part 1**

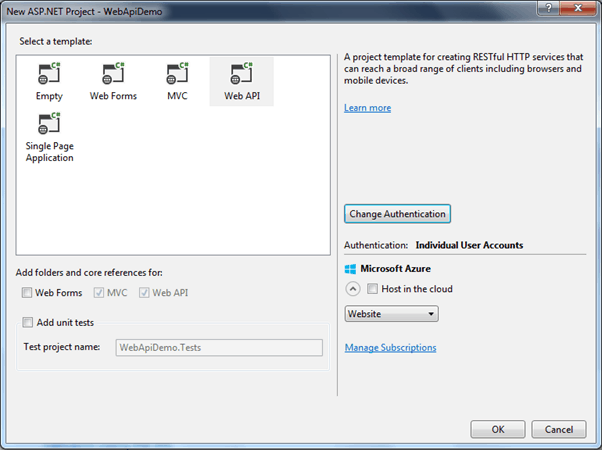
Here we will create a new Web API project and implement GET, POST, PUT and DELETE method for CRUD operation using Entity Framework.

First, create a new Web API project in Visual Studio.

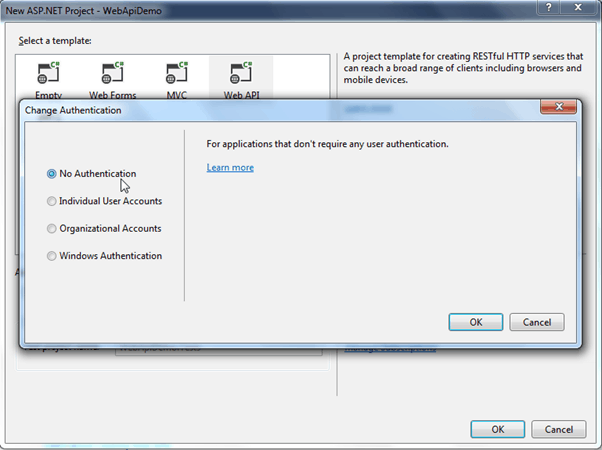
Open Visual Studio for Web and click on **File** menu -> **New Project..** This will open New Project popup as shown below.

 Create Web API Project

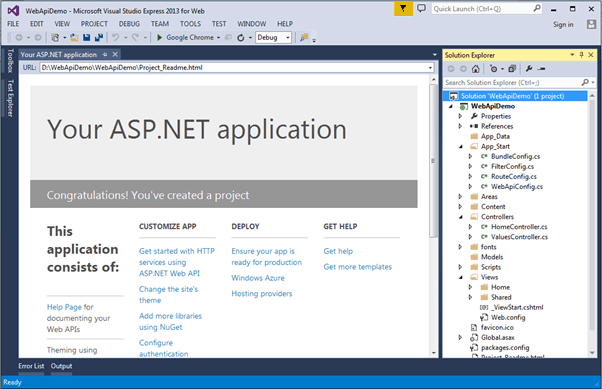
In the New Project popup, select **Web** template under **Visual C#**. Enter the project name WebApiDemo and the location where you want to create the project. Click **OK** to continue. This will open another popup to select a project template. Select Web API project as shown below.

 Select Web API Project Template

Here, we are not going to use any authentication in our demo project. So, click on the **Change Authentication** button to open Authentication popup and select **No Authentication** radio button and then click **OK** as shown below.

 Change Authentication

Now, click **OK** in New ASP.NET Project popup to create a project as shown below.

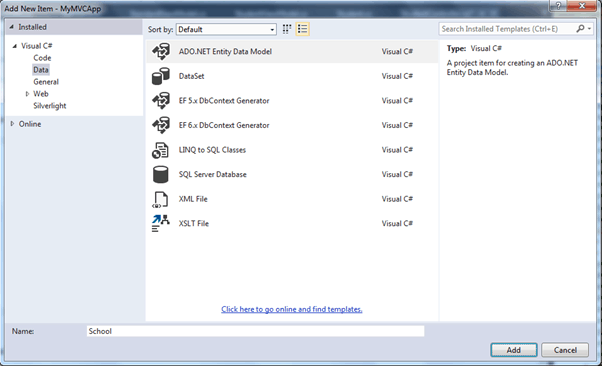
 Web API Project

As you can see, a new WebApiDemo project is created with all necessary files. It has also added default ValuesController. Since we will be adding our new Web API controller, we can delete the default ValuesController.

Here, we are going to use Entity Framework DB-First approach to access an existing school database. So, let's add EF data model for the school database using DB First approach.

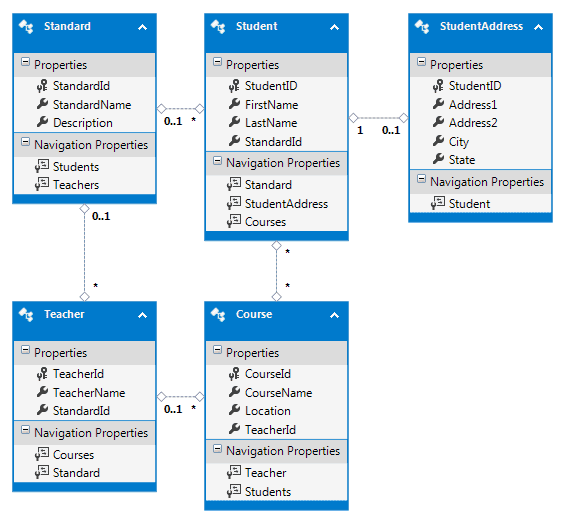
### **Add Entity Framework Data Model**

To add EF data model using DB-First approach, right click on your project -> click **New Item..** This will open **Add New Item** popup as shown below.

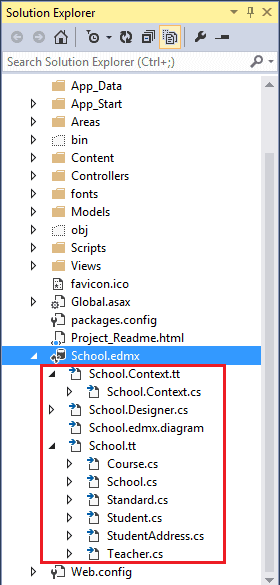
 Create Entity Data Model

Select **Data** in the left pane and select **ADO.NET Entity Data Model** in the middle pane and enter the name of a data model and click **Add**. This will open Entity Data Model Wizard using which you can generate Entity Data Model for an existing School database. [Download EF 6 demo project with Schoold Database from Github](https://github.com/entityframeworktutorial/EF6-DBFirst-Demo). The scope of the topic is limited to Web API so we have not covered how to generate EDM. Learn [how to create Entity Data Model in EF 6](https://www.entityframeworktutorial.net/entityframework6/create-entity-data-model.aspx).

EntityFramework will generate the following data model after completing all the steps of Entity Data Model Wizard.

Generated Entities in the EDM Designer

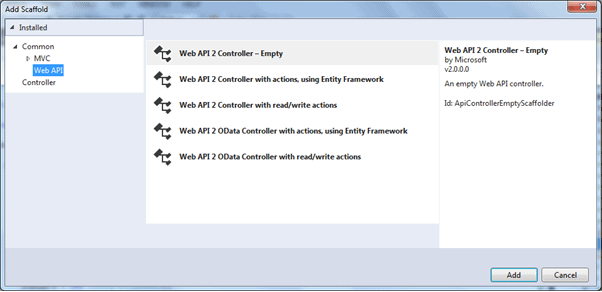
Entity Framework also generates entities and context classes as shown below.

. edmx in the Project

Now, we are ready to implement CRUD operation using Entity Framework in our Web API project. Now, let's add a Web API controller to our project.

## **Add Web API Controller**

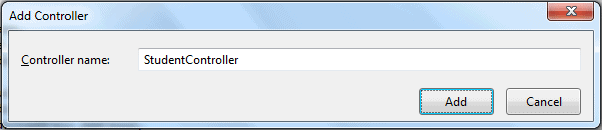
To add a Web API controller in your MVC project, right click on the **Controllers** folder or another folder where you want to add a Web API controller -> select **Add** -> select **Controller**. This will open **Add Scaffold** popup as shown below.



Create Web API Controller

In the Add Scaffold popup, select **Web API** in the left pane and select **Web API 2 Controller - Empty** in the middle pane and click **Add**. (We select Empty template as we plan to add action methods and Entity Framework by ourselves.)

This will open the **Add Controller** popup where you need to enter the name of your controller. Enter "StudentController" as a controller name and click **Add** as shown below.

Create Web API Controller

This will add empty StudentController class derived from ApiController as shown below.

Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** MyMVCApp.Controllers{

**public class** StudentController : ApiController{

}

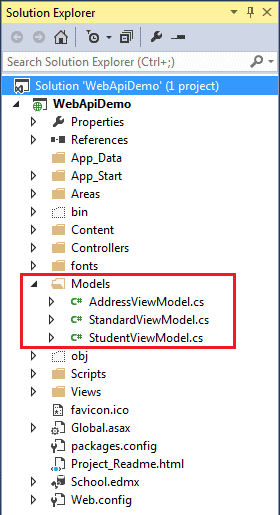
}

We will implement GET, POST, PUT and DELETE action methods in this controller in the subsequent sections.

### **Add Model**

We will be accessing the underlying database using Entity Framework (EF). As you have seen above, EF creates its own entity classes. Ideally, we should not return EF entity objects from the Web API. It is recommended to return DTO (Data Transfer Object) from Web API. As we have created a Web API project with MVC, we can also use MVC model classes which will be used in both MVC and Web API.

Here, we will return Student, Address and Standard from our Web API. So, create StudentViewModel, AddressViewModel and StandardViewModel in the Models folder as shown below.

 Models

The following are model classes.

Model Classes

**public class** StudentViewModel{

**public int** Id { get; set; }

**public string** FirstName { get; set; }

**public string** LastName { get; set; }

**public** AddressViewModel Address { get; set; }

**public** StandardViewModel Standard { get; set; }

}

**public class** StandardViewModel{

**public int** StandardId { get; set; }

**public string** Name { get; set; }

**public** ICollection<StudentViewModel> Students { get; set; }

}

**public class** AddressViewModel{

**public int** StudentId { get; set; }

**public string** Address1 { get; set; }

**public string** Address2 { get; set; }

**public string** City { get; set; }

**public string** State { get; set; }

}

 Note:

ViewModel classes or DTO classes are just for data transfer from Web API controller to clients. You may name it as per your choice.

Now, let's implement Get methods to handle various HTTP GET requests in the next section.

# **Create Web API for CRUD operation - Part 2: Implement Get Method**

This section is a continuation of the previous section where we created the necessary infrastructure for our Web API.

In this section we will implement Get action methods in our Web API controller class that will handle HTTP GET requests.

As per the Web API naming convention, action method that starts with a work "Get" will handle HTTP GET request. We can either name it only Get or with any suffix. Let's add our first Get action method and give it a name GetAllStudents because it will return all the students from the DB. Following an appropriate naming methodology increases readability and anybody can understand the purpose of a method easily.

The following GetAllStudents() action method in StudentController class (which we created in the previous section) returns all the students from the database using Entity Framework.

Example: Get Method in Web API Controller

**public class** StudentController : ApiController{

**public** IHttpActionResult GetAllStudents (){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress")

.Select(s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0) **return** NotFound();

**return** Ok(students);

}

}

As you can see in the above example, GetAllStudents() method returns all the students using EF. If no student exists in the DB then it will return 404 NotFound response otherwise it will return 200 OK response with students data. The NotFound() and Ok() methods defined in the ApiController returns 404 and 200 response respectively.

In the database, every student has zero or one address. Suppose you want to implement another GET method to get all the Students with its address then you may create another Get method as shown below.

Example: Get Methods in Web API Controller

**public class** StudentController : ApiController{

**public** IHttpActionResult GetAllStudents(){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress")

.Select(s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0) **return** NotFound();

**return** Ok(students);

}

**public** IHttpActionResult GetAllStudentsWithAddress(){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress").Select(

s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ? **null** :

**new** AddressViewModel()

{

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

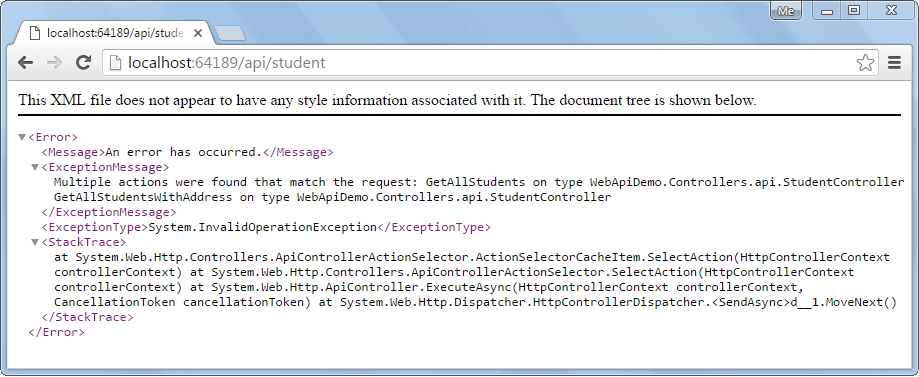
**if** (students.Count == 0) **return** NotFound();

**return** Ok(students);

}

}

The above web API example will compile without an error but when you execute HTTP GET request then it will respond with the following multiple actions found error.

Web API Error

This is because you cannot have multiple action methods with the same number of parameters with the same type. Both action methods above do not include any parameters. So, Web API does not understand which method to execute for the HTTP GET request http://localhost:64189/api/student.

The following example illustrates how to handle this kind of scenario.

Example: Get Method in Web API Controller

**public class** StudentController : ApiController{

**public** StudentController(){ }

**public** IHttpActionResult GetAllStudents(**bool** includeAddress = **false**){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress")

.Select(s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ||

includeAddress == **false** ? **null** :

**new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0) **return** NotFound();

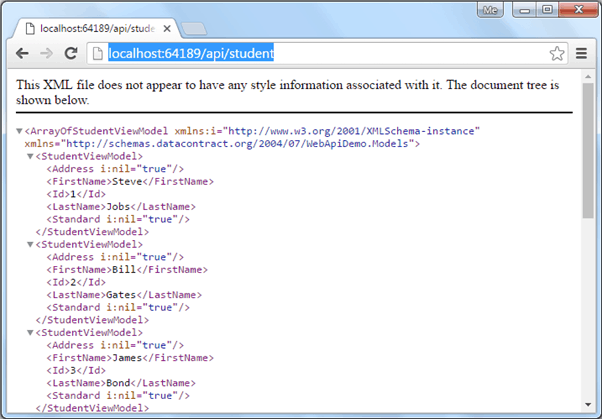
**return** Ok(students);

}

}

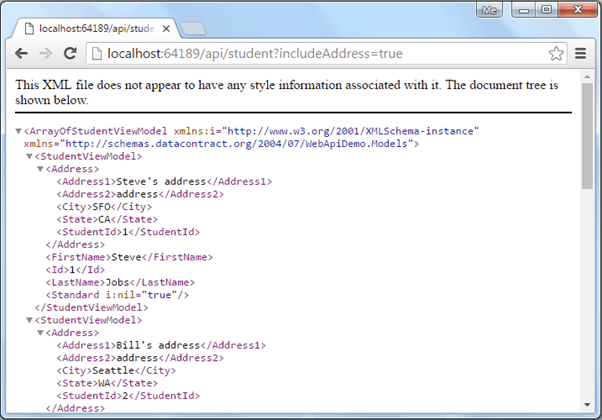
As you can see, GetAllStudents action method includes parameter includeAddress with default value false. If an HTTP request contains includeAddress parameter in the query string with value true, then it will return all students with its address otherwise it will return students without address.

For example, http://localhost:64189/api/student (64189 is a port number which can be different in your local) will return all students without address as shown below.



Access Web API GET Method in the Browser

An HTTP request http://localhost:64189/api/student?includeAddress=true will return all students with address as shown below.



Access Web API GET Method in the Browser

## **Implement Multiple GET methods**

As mentioned, Web API controller can include multiple Get methods with different parameters and types.

Let's add the following action methods in StudentController to demonstrate how Web API handles multiple HTTP GET requests.

| Action method | Purpose |
| --- | --- |
| GetStudentById(int id) | Returns student whose id matches with the specified id. |
| GetAllStudents(string name) | Returns list of students whose name matches with the specified name. |
| GetAllStudentsInSameStandard(int standardId) | Returns list of students who are in the specified standard. |

The following example implements the above action methods.

Example: Multiple Get Methods in Web API Controller

**public class** StudentController : ApiController{

**public** StudentController(){ }

**public** IHttpActionResult GetAllStudents(**bool** includeAddress = **false**){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress").Select(

s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ||

includeAddress == **false** ? **null** : **new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

**if** (students == **null**)**return** NotFound();

**return** Ok(students);

}

**public** IHttpActionResult GetStudentById(**int** id){

StudentViewModel student = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

student = ctx.Students.Include("StudentAddress")

.Where(s => s.StudentID == id)

.Select(s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName

}).FirstOrDefault<StudentViewModel>();

}

**if** (student == **null**)**return** NotFound();

**return** Ok(student);

}

**public** IHttpActionResult GetAllStudents(**string** name){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress")

.Where(s => s.FirstName.ToLower() == name.ToLower())

.Select(s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ? **null** : **new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0)**return** NotFound();

**return** Ok(students);

}

**public** IHttpActionResult GetAllStudentsInSameStandard(**int** standardId){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress").Include("Standard")

.Where(s => s.StandardId == standardId).Select(

s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ? **null** : **new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

},

Standard = **new** StandardViewModel(){

StandardId = s.Standard.StandardId,

Name = s.Standard.StandardName

}

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0)**return** NotFound();

**return** Ok(students);

}

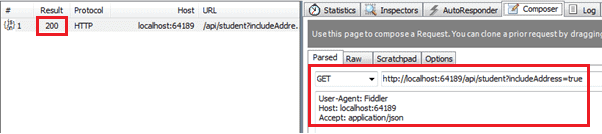
}

Now, the above Web API will handle the following HTTP GET requests.

| HTTP GET Request URL | Description |
| --- | --- |
| http://localhost:64189/api/student | Returns all students without associated address. |
| http://localhost:64189/api/student?includeAddress=false | Returns all students without associated address. |
| http://localhost:64189/api/student?includeAddress=true | Returns all students with address. |
| http://localhost:64189/api/student?id=123 | Returns student with the specified id. |
| http://localhost:64189/api/student?name=steve | Returns all students whose name is steve. |
| http://localhost:64189/api/student?standardId=5 | Returns all students who are in 5th standard. |

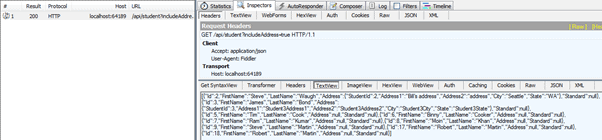
Similarly, you can implement Get methods to handle different HTTP GET requests in the Web API.

The following figure shows HTTP GET request in Fiddler.



Http GET request in Fiddler

The following figure shows HTTP GET response of above request in Fiddler.



Http GET response in Fiddler

Next, implement the Post action method to handle HTTP POST request in the Web API.

# **Create Web API for CRUD operation - Part 3: Implement Post Method**

This section is a continuation of the previous two sections where we created necessary infrastructure for the Web API and implemented GET methods. Here, we will implement the POST method in the Web API.

The HTTP POST request is used to create a new record in the data source in the RESTful architecture. So, let's create an action method in our StudentController to insert new student records in the database using Entity Framework.

The action method that will handle HTTP POST request must start with the word Post. It can be named either Post or with any suffix e.g. POST(), Post(), PostNewStudent(), PostStudents() are valid names for an action method that handles HTTP POST request.

The following example demonstrates a Post action method to handle HTTP POST request.

Example: Post Method in Web API Controller

**public class** StudentController : ApiController{

**public** StudentController(){

}

//Get action methods of the previous section

**public** IHttpActionResult PostNewStudent(StudentViewModel student){

**if** (!ModelState.IsValid) **return** BadRequest("Invalid data.");

**using** (**var** ctx = **new** SchoolDBEntities()){

ctx.Students.Add(**new** Student(){

StudentID = student.Id,

FirstName = student.FirstName,

LastName = student.LastName

});

ctx.SaveChanges();

}

**return** Ok();

}

}

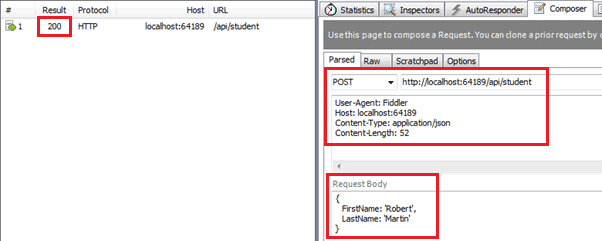
As you can see above, we named the action method as PostNewStudent. You can give any name as per your requirement, but it must start with the word "Post". The PostNewStudent() action method includes parameter of StudentViewModel type which includes all the information about new student.

In the Post method we first need to validate the model using ModelState.IsValid. This will make sure that the student object includes all the necessary information. If it is not valid then you can return BadRequest response. If it is valid then add a student using Entity Framework context and return 200 OK status response.

 Note:

This is just a demo project. However, you can return newly created student object with Id in the response.

Now, you can send HTTP POST request using Fiddler as shown below and see the response.

Execute HTTP POST request in Fiddler

As you can see in the above figure, HTTP POST request includes *StudentViewModel* object into JSON format in the request body. After successful execution the response status is 200 OK.

Next, implement the Put action method to handle HTTP PUT request in the Web API.

# **Create Web API for CRUD operation - Part 4: Implement Put Method**

This section is a continuation of the previous three sections where we created necessary infrastructure for the Web API and implemented GET & POST methods. Here, we will implement the PUT method in the Web API.

The HTTP PUT method is used to update an existing record in the data source in the RESTful architecture.

So, let's create an action method in our StudentController to update an existing student record in the database using Entity Framework. The action method that will handle HTTP PUT request must start with the word Put. It can be named either Put or with any suffix e.g. PUT(), Put(), PutStudent(), PutStudents() are valid names for an action method that handles HTTP PUT request.

The following example demonstrates the Put action method to handle HTTP PUT request.

Example: Put Method in Web API Controller

**public class** StudentController : ApiController{

**public** StudentController(){

}

**public** IHttpActionResult Put(StudentViewModel student){

**if** (!ModelState.IsValid) **return** BadRequest("Not a valid model");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** existingStudent = ctx.Students.Where(

s => s.StudentID == student.Id).FirstOrDefault<Student>();

**if** (existingStudent != **null**){

existingStudent.FirstName = student.FirstName;

existingStudent.LastName = student.LastName;

ctx.SaveChanges();

}

**else** **return** NotFound();

}

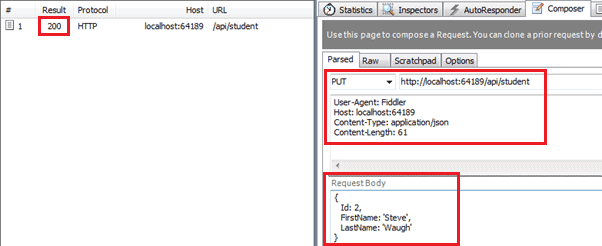
**return** Ok();

}

}

As you can see above, the Put action method includes a parameter of StudentViewModel. It then creates a new student entity using passed StudentViewModel object and then changes the state to be modified.

Now, you can send HTTP PUT request using Fiddler as shown below and see the response.

Execute PUT request in Fiddler

As you can see in the above figure, HTTP PUT request includes *StudentViewModel* object into JSON format in the request body. After successfull execution the response status is 200 OK.

Next, implement the Delete action method to handle HTTP DELETE request in the Web API.

# **Create Web API for CRUD operation - Part 5: Implement Delete Method**

This section is a continuation of the previous four sections where we created necessary infrastructure for the Web API and also implemented GET, POST & PUT methods. Here, we will implement the Delete action method in the Web API.

The HTTP DELETE request is used to delete an existing record in the data source in the RESTful architecture.

So let's create an action method in our StudentController to delete an existing student record in the database using Entity Framework. The action method that will handle HTTP DELETE request must start with the word "Delete". It can be named either Delete or with any suffix e.g. DELETE(), Delete(), DeleteStudent(), DeleteAllStudents() are valid names for an action method that handles HTTP DELETE request.

The following example demonstrates the Delete action method to handle HTTP DELETE request.

Example: Delete Method in Web API Controller

**public class** StudentController : ApiController{

**public** StudentController(){

}

**public** IHttpActionResult Delete(**int** id){

**if** (id <= 0) **return** BadRequest("Not a valid student id");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** student = ctx.Students

.Where(s => s.StudentID == id).FirstOrDefault();

ctx.Entry(student).State = System.Data.Entity.EntityState.Deleted;

ctx.SaveChanges();

}

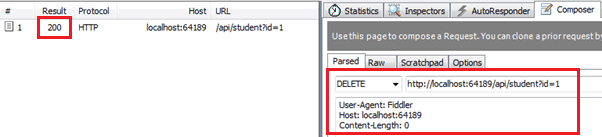
**return** Ok();

}

}

As you can see above, the Delete action method includes an id parameter of int type because it just needs an id to delete a record. It fetches an existing student from the database that matches with the specified id and then marks its status as deleted. This will delete a student from the database.

Now, you can send HTTP DELETE request using Fiddler as shown below and view the response.

Execute HTTP DELETE request in Fiddler

As you can see in the above figure, HTTP DELETE request URL http://localhost:64189/api/student?id=1 includes query string id. This id query string will be passed as an id parameter in the Delete method. After successful execution the response status is 200 OK.

Thus, you can create Get, Post, Put and Delete methods to implement HTTP GET, POST, PUT and DELETE requests respectively.

Now, let's consume this Web API into ASP.NET MVC and AngularJS application in the next sections.

# **Consume Web API for CRUD operation**

In the previous section, we created Web API with Get, Post, Put and Delete methods that handles HTTP GET, POST, PUT and DELETE requests respectively. Here, we will see how to consume (access) Web API for CRUD operation.

Web API can be accessed in the server-side code in .NET and also on client side using JavaScript frameworks such as jQuery, AnguarJS, KnockoutJS etc.

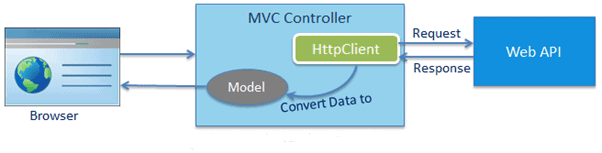
Here, we will consume our Web API (created in the previous section) in the following environments:

1. Consume Web API in ASP.NET MVC
2. Consume Web API in AngularJS

## **Consume Web API in ASP.NET MVC**

To consume Web API in ASP.NET MVC server side we can use HttpClient in the MVC controller. HttpClient sends a request to the Web API and receives a response. We then need to convert response data that came from Web API to a model and then render it into a view.

The following figure illustrates consuming Web API in ASP.NET MVC.

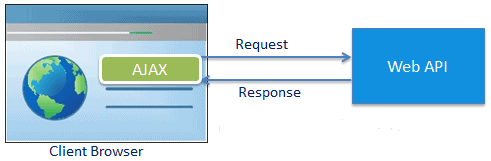
Consume Web API at Server side ASP.NET MVC

Note: AngularJS or any other JavaScript framework can be used in MVC view and can access Web API directly from the view using AJAX. We have taken ASP.NET MVC just to demonstrate how to access Web API from server-side code in case you do not use any JavaScript framework.

## **Consume Web API in AngularJS**

Web API can be accessed directly from the UI at client side using AJAX capabilities of any JavaScript framework such as AngularJS, KnockoutJS, Ext JS etc.

The following figure illustrates consuming Web API in client-side framework using AJAX.



Consume Web API at Client Side

Learn how to consume Get, Post, Put and Delete methods of Web API in ASP.NET MVC and Angular in the next coming sections.

# **Consume Web API Get method in ASP.NET MVC**

We created a Web API and implemented various Get methods to handle different HTTP GET requests in the Implement Get Method section. Here we will consume one of those Get methods named GetAllStudents() shown below.

Example: Sample Web API

**public class** StudentController : ApiController{

**public** StudentController(){ }

**public** IHttpActionResult GetAllStudents(**bool** includeAddress = **false**){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress").Select(

s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** || includeAddress == **false** ?

**null** : **new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

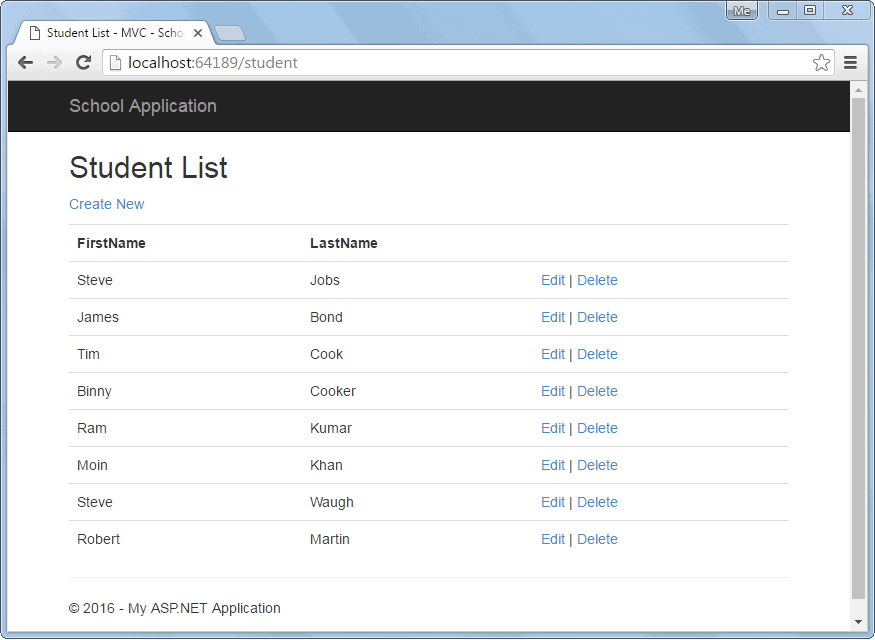
**if** (students.Count == 0)**return** NotFound();

**return** Ok(students);

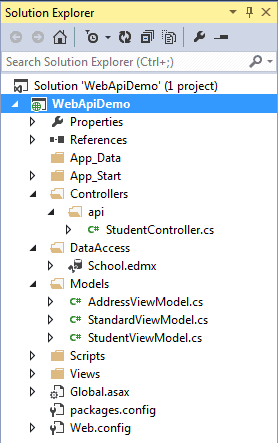
}

}

The above GetAllStudents() action method will handle HTTP GET request http://localhost:64189/api/student and will return a list of students. We will send this HTTP request to the ASP.NET MVC controller to get all the student records and display them in the MVC View. The view will look like below.

 Student List View

The following is a Web API + MVC project structure created in the previous sections. We will add necessary classes to this project.

 Web API Project

We have already created the following StudentViewModel class under Models folder.

Example: Model Class

**public class** StudentViewModel{

**public int** Id { get; set; }

**public string** FirstName { get; set; }

**public string** LastName { get; set; }

**public** AddressViewModel Address { get; set; }

**public** StandardViewModel Standard { get; set; }

}

Let's consume the above Web API into ASP.NET MVC application step by step.

**Step 1:**

First of all, create MVC controller class called StudentController in the Controllers folder as shown below. Right click on the Controllers folder > **Add..** > select **Controller..**

Example: MVC Controller

**public class** StudentController : Controller{

// GET: Student

**public** ActionResult Index(){

**return** View();

}

}

**Step 2:**

We need to access Web API in the Index() action method using HttpClient as shown below.

Example: MVC Controller

**public class** StudentController : Controller{

// GET: Student

**public** ActionResult Index(){

IEnumerable<StudentViewModel> students = **null**;

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/");

//HTTP GET

**var** responseTask = client.GetAsync("student");

responseTask.Wait();

**var** result = responseTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<IList<StudentViewModel>>();

readTask.Wait();

students = readTask.Result;

}

**else** {//web api sent error response log response status here..

students = Enumerable.Empty<StudentViewModel>();

ModelState.AddModelError(**string**.Empty,

"Server error. Please contact administrator.");

}

}

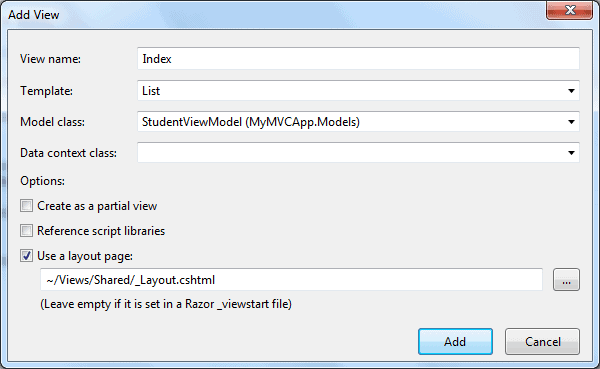
**return** View(students);

}

}

**Step 3:**

Now, we need to add Index view. Right click in the Index action method and select **Add View..** option. This will open the Add View popup as shown below. Now, select List as template and StudentViewModel as Model class as below (we already created StudentViewModel in the previous section).

 Add View in ASP.NET MVC

Click **Add** to add Index view in the **Views** folder. This will generate the following Index.cshtml.

Index.cshtml

@model IEnumerable<WebAPIDemo.Models.StudentViewModel>

@{

ViewBag.Title = "Index";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table **class**="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.FirstName)

</th>

<th>

@Html.DisplayNameFor(model => model.LastName)

</th>

<th></th>

</tr>

@**foreach** (**var** item **in** Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.FirstName)

</td>

<td>

@Html.DisplayFor(modelItem => item.LastName)

</td>

<td>

@Html.ActionLink("Edit", "Edit", **new** { id = item.Id }) |

@Html.ActionLink("Details", "Details", **new** { id = item.Id }) |

@Html.ActionLink("Delete", "Delete", **new** { id = item.Id })

</td>

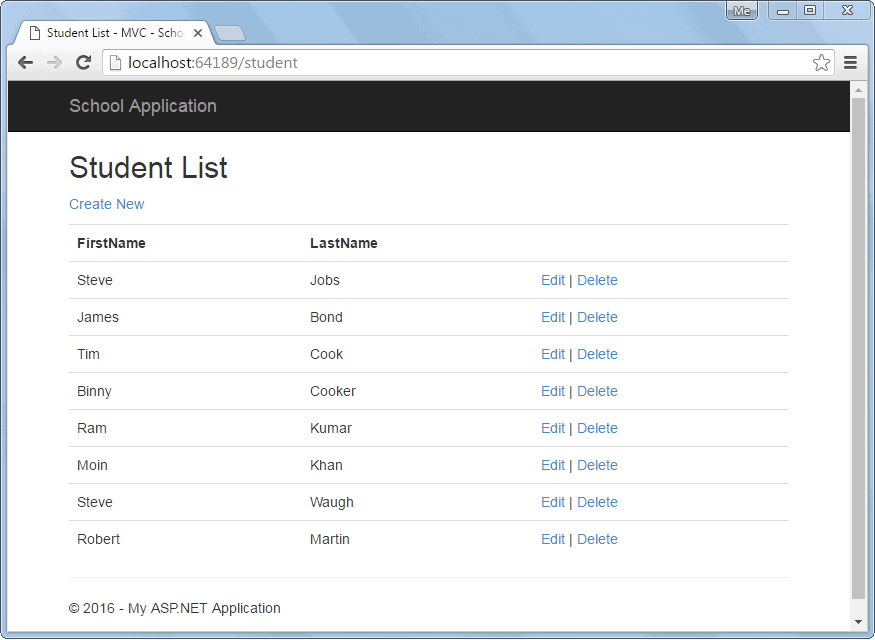
</tr>

}

</table>

Remove Details link from the View because we will not create Details page here.

Now, run the application and you will see list of students in the browser as shown below.

Student List View

### **Display Error**

We have successfully displayed records in the view above but what if Web API returns error response?

To display appropriate error message in the MVC view, add the ValidationSummary() as shown below.

Index.cshtml

@model IEnumerable<WebAPIDemo.Models.StudentViewModel>

@{

ViewBag.Title = "Index";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table **class**="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.FirstName)

</th>

<th>

@Html.DisplayNameFor(model => model.LastName)

</th>

<th></th>

</tr>

@**foreach** (**var** item **in** Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.FirstName)

</td>

<td>

@Html.DisplayFor(modelItem => item.LastName)

</td>

<td>

@Html.ActionLink("Edit", "Edit", **new** { id = item.Id }) |

@Html.ActionLink("Delete", "Delete", **new** { id = item.Id })

</td>

</tr>

}

<tr>

<td>

@Html.ValidationSummary(**true**, "", **new** { @class = "text-danger" })

</td>

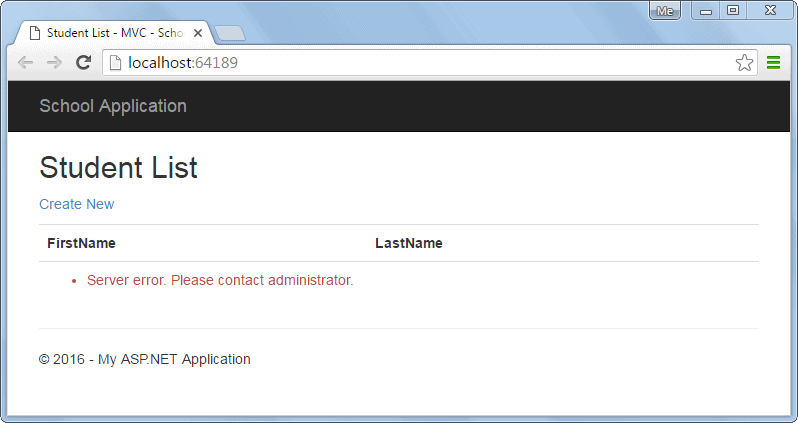
</tr>

</table>

In the above view, we have added @Html.ValidationSummary(true, "", new { @class = "text-danger" }) in the last row of the table. This is to display an error message if Web API returns error response with the status other than 200 OK.

Please note that we have added model error in the Index() action method in StudentController class created in step 2 if Web API responds with the status code other than 200 OK.

So now, if Web API returns any kind of error, then Student List view will display the message below.

Display Error Message

In the next section, we will use the Post method to create a new record in the underlying data source by clicking on Create New link in the above view.

# **Consume Web API Post method in ASP.NET MVC**

In the previous section, we learned how to consume the Web API Get method and display records in the ASP.NET View. Here, we will see how to consume Post method of Web API to create a new record in the data source.

We already created a Web API with Post method in the Implement Post Method section shown below.

Example: Sample Web API with Post Method

**public class** StudentController : ApiController{

**public** StudentController(){ }

//Get action methods of the previous section

**public** IHttpActionResult PostNewStudent(StudentViewModel student){

**if** (!ModelState.IsValid) **return** BadRequest("Not a valid model");

**using** (**var** ctx = **new** SchoolDBEntities()){

ctx.Students.Add(**new** Student(){

StudentID = student.Id,

FirstName = student.FirstName,

LastName = student.LastName

});

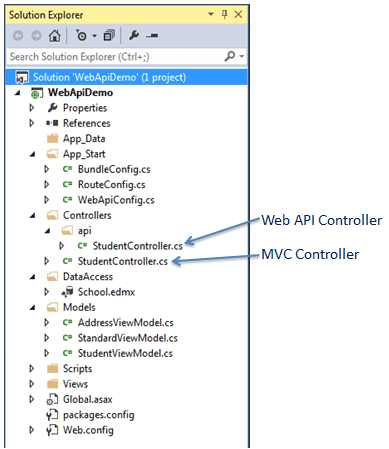
ctx.SaveChanges();

}

**return** Ok();

}

}

In the above Web API, PostNewStudent method will handle HTTP POST request http://localhost:64189/api/student. It will insert a new record in the database using Entity Framework and will return 200 OK response status.

The right side is a Web API + MVC project structure created in the previous sections. We will add necessary classes to this project.

Web API Project

We have already created the following StudentViewModel class under Models folder.

Example: Model Class

**public class** StudentViewModel{

**public int** Id { get; set; }

**public string** FirstName { get; set; }

**public string** LastName { get; set; }

**public** AddressViewModel Address { get; set; }

**public** StandardViewModel Standard { get; set; }

}

Now, let's create MVC view to create a new record by consuming the above Web API Post method.

**Step 1:**

First, we need to add the action method "create" which will render "Create New Student" view where user can enter data and submit it. We have already created StudentController class in the previous section to display student list view. Here, add "create" action method to render "Create New Student" view shown below.

Example: MVC Controller

**public class** StudentController : Controller{

**public** ActionResult Index(){

//consume Web API Get method here..

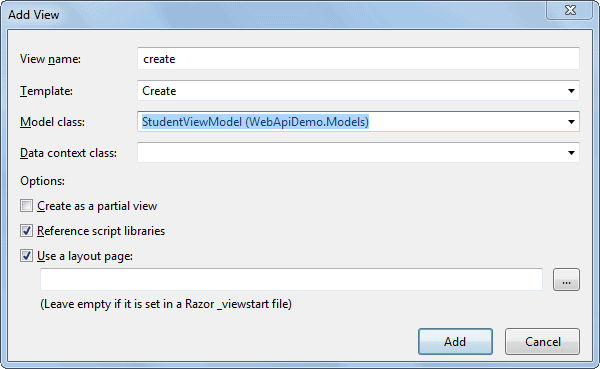
**return** View();

}

**public** ActionResult create(){ **return** View();}

}

Now, right click in the above action method and select **Add View..** This will open following Add View popup.

 Add View in ASP.NET MVC

Now, select Create Template, StudentViewModel class as a model and click on Add button as shown above. This will generate createcshtml in the Views > Student folder as below.

create.cshtml

@model WebApiDemo.Models.StudentViewModel

@{

ViewBag.Title = "Create New Student - MVC";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Create New Student</h2>

@**using** (Html.BeginForm()) {

@Html.AntiForgeryToken()

<div **class**="form-horizontal">

<hr />

@Html.ValidationSummary(**true**, "", **new** { @class = "text-danger" })

<div **class**="form-group">

@Html.LabelFor(model => model.FirstName, htmlAttributes:

**new** { @class = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.FirstName,

**new** { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.FirstName, "",

**new** { @class = "text-danger" })

</div>

</div>

<div **class**="form-group">

@Html.LabelFor(model => model.LastName, htmlAttributes:

**new** { @class = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.LastName,

**new** { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.LastName, "",

**new** { @class = "text-danger" })

</div>

</div>

<div **class**="form-group">

<div **class**="col-md-offset-2 col-md-10">

<input type="submit" **value**="Create" **class**="btn btn-default" />

</div>

</div>

</div>

}

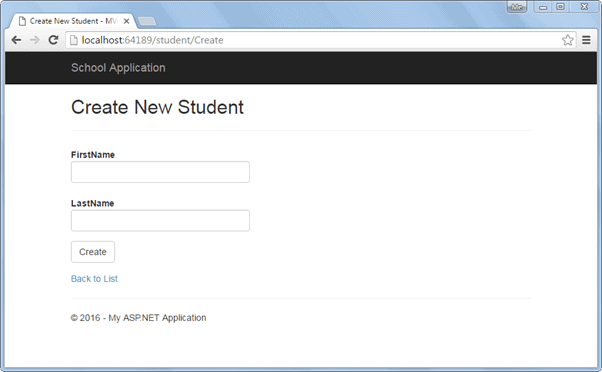
<div>

@Html.ActionLink("Back to List", "Index")

</div>

In the above view, Html.BeginForm() generates HTML form tag <form> action="/Student/Create" method="post" </form> which will send post request when user clicks on the create button.

Now, run the project and navigate to http://localhost:64189/student/create. It will display the simple data entry view as shown below.

 Create New Student View

As soon as the user enters student data and clicks on the **Create** button in the above view, it will send Post request to the Student MVC controller. To handle this post request, add HttpPost action method "create" as shown below.

Example: Post Method in MVC Controller

**public class** StudentController : Controller{

**public** ActionResult Index(){

//consume Web API Get method here..

**return** View();

}

**public** ActionResult create(){

**return** View();

}

[HttpPost]

**public** ActionResult create(StudentViewModel student){

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/student");

//HTTP POST

**var** postTask = client.PostAsJsonAsync<StudentViewModel>(

"student", student);

postTask.Wait();

**var** result = postTask.Result;

**if** (result.IsSuccessStatusCode){

**return** RedirectToAction("Index");

}

}

ModelState.AddModelError(**string**.Empty,

"Server Error. Please contact administrator.");

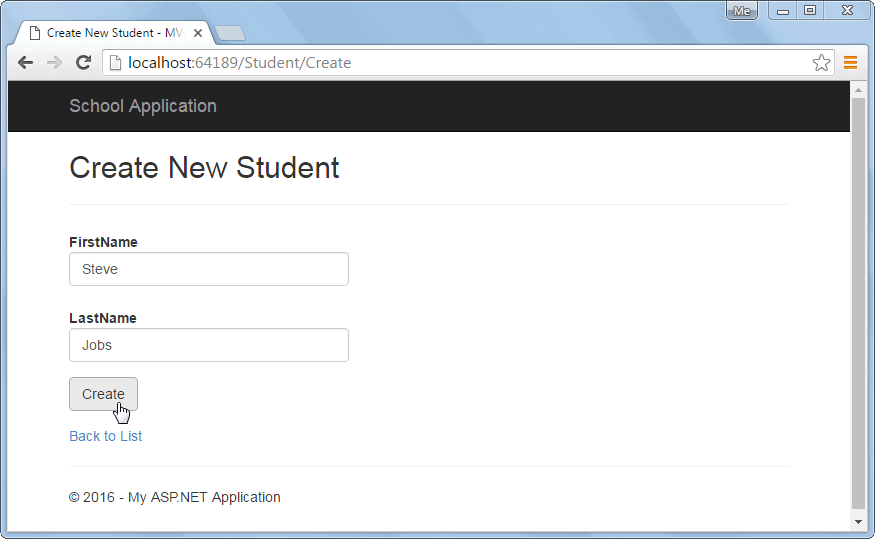
**return** View(student);

}

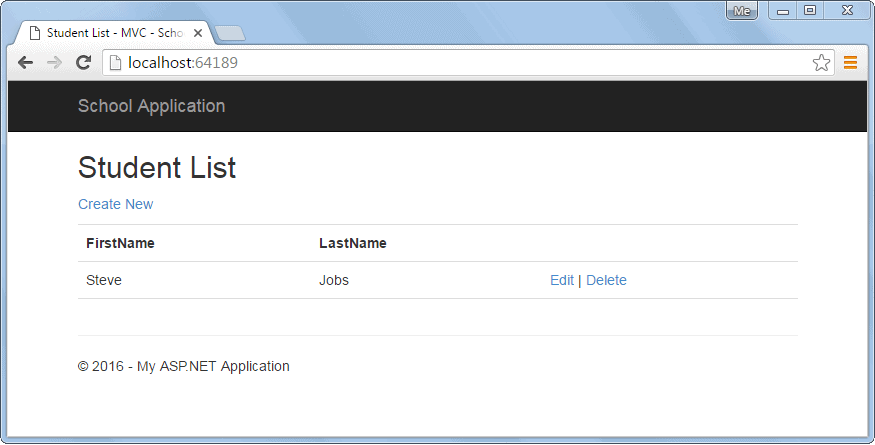
}

As you can see in the above HttpPost action method create() uses HttpClient to send HTTP POST request to Web API with StudentViewModel object. If the response returns success status, then it will redirect to the list view. Visit HttpClient section to learn more about it.

Now, run the project and navigate to http://localhost:64189/student/create, enter student information as shown below.

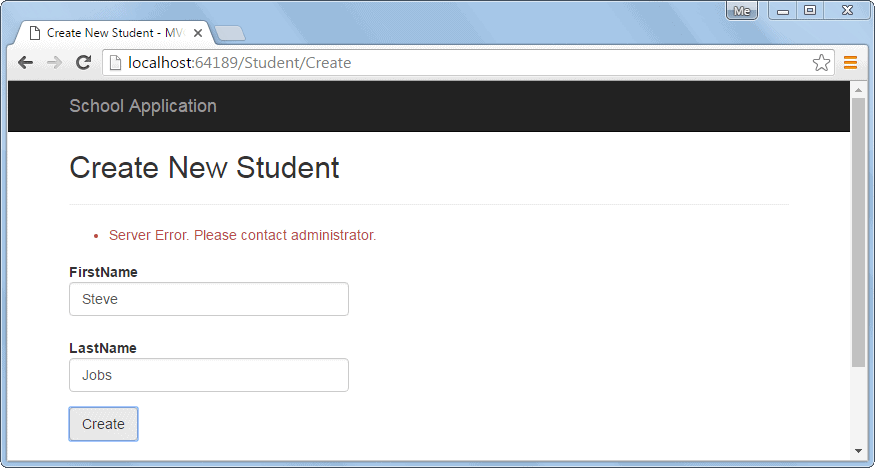
 Create a New Student

Now, on the click of create button above, it will insert a new record in the DB and redirect to the list view as shown below.



Redirect to Student List View

Also, the above create view will display an error message if Web API sends error response as shown below.



Display Error Message

So, in this way we can consume the Post method of Web API to execute HTTP POST request to create a new record.

Next, consume Put method of Web API to edit an existing record.

# **Consume Web API Put method in ASP.NET MVC**

In the previous two sections, we learned how to consume Web API Get and Post methods in the ASP.NET View. Here, we will see how to consume Put method of Web API to update an existing record.

We already created a Web API with Put method that handles HTTP PUT request in the Implement Put Method section as below.

Example: Sample Web API with Put method

**public class** StudentController : ApiController{

**public** StudentController(){ }

**public** IHttpActionResult Put(StudentViewModel student){

**if** (!ModelState.IsValid) **return** BadRequest("Not a valid data");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** existingStudent = ctx.Students.Where(

s => s.StudentID == student.Id).FirstOrDefault<Student>();

**if** (existingStudent != **null**){

existingStudent.FirstName = student.FirstName;

existingStudent.LastName = student.LastName;

ctx.SaveChanges();

}

**else** **return** NotFound();

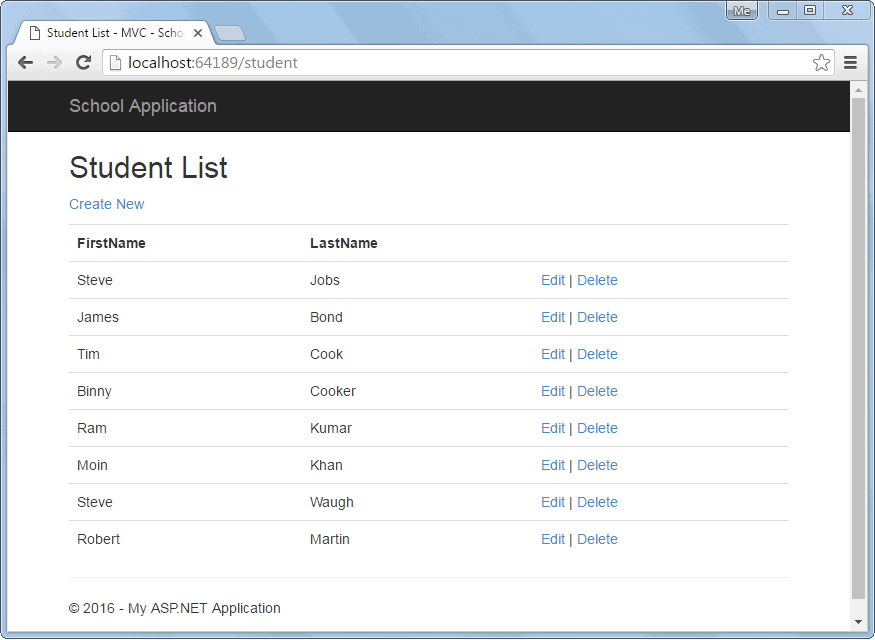
}

**return** Ok();

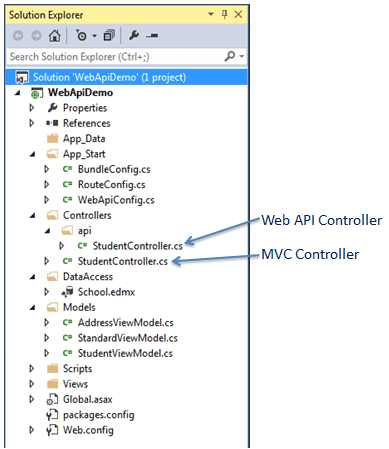
}

}

We created Student List view in the previous section as below. In the below view there is an edit link for each record to edit that record. We will handle editing functionality in this section.

 Student List View

The following is a Web API + MVC project structure created in the previous sections. We will add the necessary classes to this project.

Web API Project

We have already created the following StudentViewModel class under Models folder.

Example: Model Class

**public class** StudentViewModel{

**public int** Id { get; set; }

**public string** FirstName { get; set; }

**public string** LastName { get; set; }

**public** AddressViewModel Address { get; set; }

**public** StandardViewModel Standard { get; set; }

}

So, let's consume Web API Put method by implementing edit functionality.

**Step 1:**

In the above Student List view, when user clicks on the Edit link it will send HTTP GET request http://localhost:64189/student/edit/{id} to the MVC controller. So, we need to add HttpGet action method "Edit" in the StudentController to render an edit view as shown below.

Example: Implement Edit Action Method

**public class** StudentController : Controller

**public** ActionResult Index(){

//consume Web API Get method here..

**return** View();

}

**public** ActionResult Edit(**int** id){

StudentViewModel student = **null**;

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/");

//HTTP GET

**var** responseTask = client.GetAsync("student?id=" + id.ToString());

responseTask.Wait();

**var** result = responseTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<StudentViewModel>();

readTask.Wait();

student = readTask.Result;

}

}

**return** View(student);

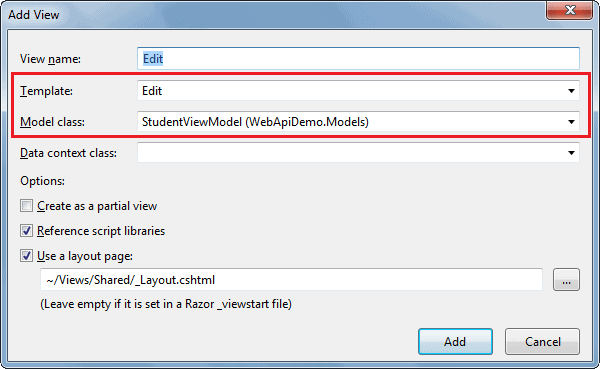
}

}

As you can see above, the Edit action method includes id parameter. This id parameter will be bound to the query string id parameter. We use this id to get a student record from the database using HttpClient and pass the student record in the edit view.

**Step 2:**

Create edit view by right clicking in the above Edit action method and select **Add View..** This will open the Add View popup as shown below.

 Add View in ASP.NET MVC

In the Add View popup, select Edit template and StudentViewModel as a model class as shown above. Click Add button to generate Edit.cshtml view in the Views > Student folder as shown below.

Edit.cshtml

@model WebApiDemo.Models.StudentViewModel

@{

ViewBag.Title = "Edit Student - MVC";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Edit Student</h2>

@**using** (Html.BeginForm()) {

@Html.AntiForgeryToken()

<div **class**="form-horizontal">

<hr />

@Html.ValidationSummary(**true**, "", **new** { @class = "text-danger" })

@Html.HiddenFor(model => model.Id)

<div **class**="form-group">

@Html.LabelFor(model => model.FirstName, htmlAttributes:

**new** { @class = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.FirstName,

**new** { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.FirstName, "",

**new** { @class = "text-danger" })

</div>

</div>

<div **class**="form-group">

@Html.LabelFor(model => model.LastName, htmlAttributes:

**new** { @class = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.LastName,

**new** { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.LastName, "",

**new** { @class = "text-danger" })

</div>

</div>

<div **class**="form-group">

<div **class**="col-md-offset-2 col-md-10">

<input type="submit" **value**="Save" **class**="btn btn-default" />

</div>

</div>

</div>

}

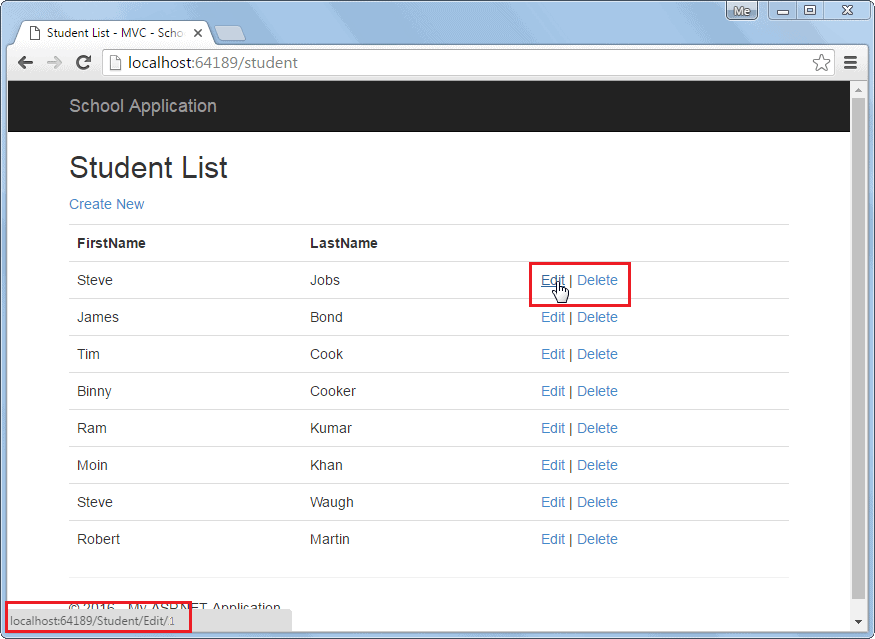
<div>

@Html.ActionLink("Back to List", "Index")

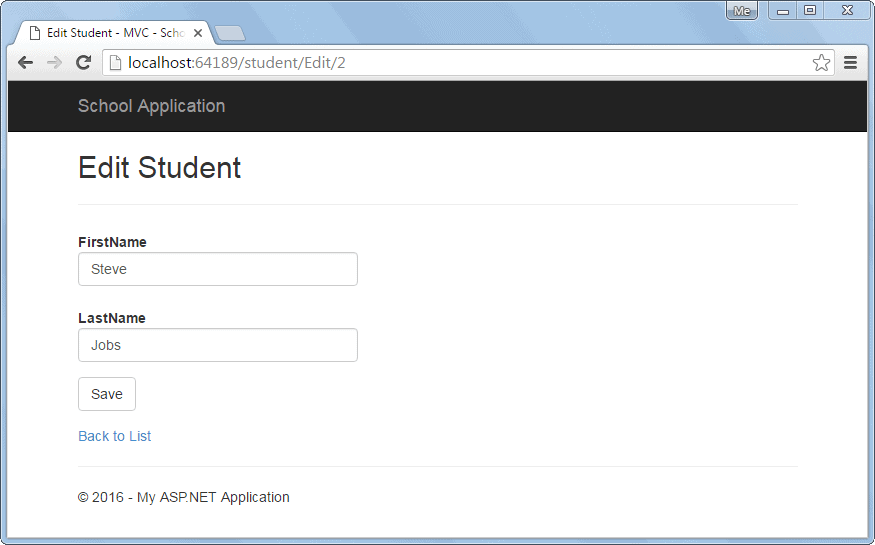
</div>

In the above view, Html.BeginForm() generates HTML form tag <form> action="/Student/edit" method="post" </form> which will send post request when user clicks on the save button.

Now, it will display the following Student List view when you run the project by pressing Ctrl + F5.

 Student List View

It will display the following edit view when you click on the Edit link in the above view.

 Edit View

Now, implement HttpPost Edit action method which will be executed when user clicks on the **Save** button above.

**Step 3:**

Add HttpPost action method in StudentController of MVC which will send HTTP PUT request to Web API to update current record.

Example: Implement HttpPost Action Method

**public class** StudentController : Controller{

**public** ActionResult Edit(**int** id){

StudentViewModel student = **null**;

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/");

//HTTP GET

**var** responseTask = client.GetAsync("student?id=" + id.ToString());

responseTask.Wait();

**var** result = responseTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<StudentViewModel>();

readTask.Wait();

student = readTask.Result;

}

}

**return** View(student);

}

[HttpPost]

**public** ActionResult Edit(StudentViewModel student){

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/student");

//HTTP POST

**var** putTask = client.PutAsJsonAsync<StudentViewModel>(

"student", student);

putTask.Wait();

**var** result = putTask.Result;

**if** (result.IsSuccessStatusCode)**return** RedirectToAction("Index");

}

**return** View(student);

}

}

As you can see above, HttpPost Edit action method uses HttpClient to send HTTP PUT request to the Web API with updated student record. Visit HttpClient section to learn more about it.

So, in this way we can consume Put method of Web API to execute HTTP PUT request to edit an existing record.

Next, consume the Delete method of Web API to delete a record in the data source.

# **Consume Web API Delete Method in ASP.NET MVC**

In the previous sections, we consumed Get, Post and Put methods of the Web API. Here, we will use the Delete method of Web API in ASP.NET MVC to delete a record.

We have already created a Web API with Delete method that handles HTTP DELETE request in the Implement Delete Method section as below.

Sample Web API with Delete Method

**public class** StudentController : ApiController{

**public** StudentController(){ }

**public** IHttpActionResult Delete(**int** id){

**if** (id <= 0) **return** BadRequest("Not a valid student id");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** student = ctx.Students

.Where(s => s.StudentID == id)

.FirstOrDefault();

ctx.Entry(student).State = System.Data.Entity.EntityState.Deleted;

ctx.SaveChanges();

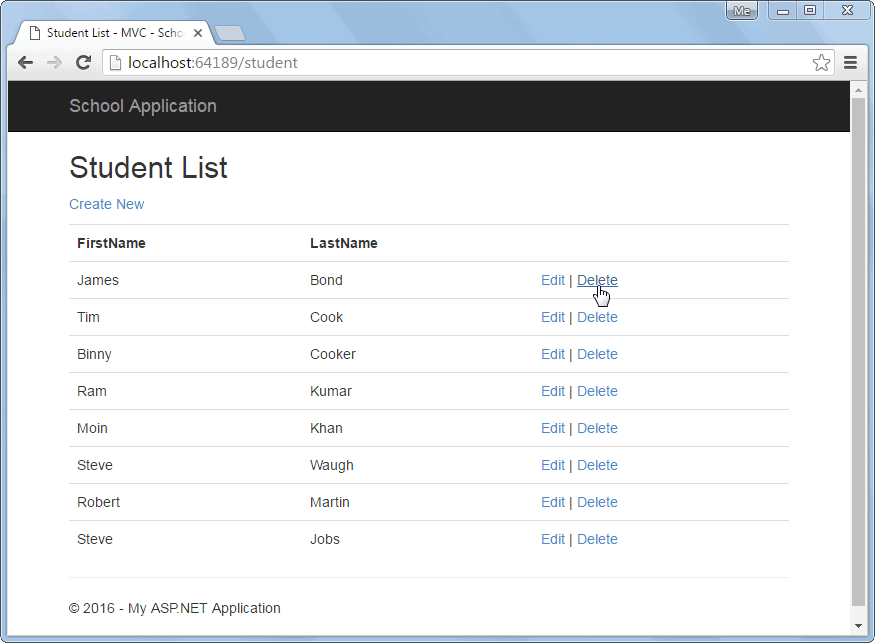
}

**return** Ok();

}

}

The following is a Student list view created in the Consuming get method in MVC section. Here, we will implement delete functionality when user clicks on the Delete link in the following UI.

 Student List View

When user clicks on the Delete link in the above UI, it sends HTTP Get request http://localhost:64189/student/delete/{id} to the Student controller with the current id parameter. So, let's implement delete functionality by consuming Web API Delete method.

**Step 1:**

Create HttpGet action method Delete with id parameter in the MVC StudentController as shown below.

Example: Implement HttpGet Delete method

**public class** StudentController : Controller{

// GET: Student

**public** ActionResult Index(){

IList<StudentViewModel> students = **null**;

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/student");

//HTTP GET

**var** responseTask = client.GetAsync("student");

responseTask.Wait();

**var** result = responseTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<IList<StudentViewModel>>();

readTask.Wait();

students = readTask.Result;

}

}

**return** View(students);

}

**public** ActionResult Delete(**int** id){

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:64189/api/");

//HTTP DELETE

**var** deleteTask = client.DeleteAsync("student/" + id.ToString());

deleteTask.Wait();

**var** result = deleteTask.Result;

**if** (result.IsSuccessStatusCode) **return** RedirectToAction("Index");

}

**return** RedirectToAction("Index");

}

}

As you can see, the Delete action method above uses HttpClient to send HTTP DELETE request with the current id parameter. The Web API controller shown in the first code example, will handle this DELETE request and delete the record from the data source. Visit [HttpClient](http://localhost:56670/webapi/consuming-web-api-in-dotnet-using-httpclient) section to learn more about it.

So, in this way you can consume the Delete method of Web API in ASP.NET MVC.

# **Consume Web API in .NET using HttpClient**

The .NET 2.0 included WebClient class to communicate with web server using HTTP protocol. However, WebClient class had some limitations. The .NET 4.5 includes HttpClient class to overcome the limitation of WebClient. Here, we will use HttpClient class in console application to send data to and receive data from Web API which is hosted on local IIS web server. You may use HttpClient in other .NET applications also such as MVC Web Application, windows form application, windows service application etc.

Let's see how to consume Web API using HttpClient in the console application.

We will consume the following Web API created in the previous section.

Example: Web API Controller

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Web.Http;

**namespace** MyWebAPI.Controller{

**public class** StudentController : ApiController{

**public** IHttpActionResult GetAllStudents(**bool** includeAddress = **false**){

IList<StudentViewModel> students = **null**;

**using** (**var** ctx = **new** SchoolDBEntities()){

students = ctx.Students.Include("StudentAddress").Select(

s => **new** StudentViewModel(){

Id = s.StudentID,

FirstName = s.FirstName,

LastName = s.LastName,

Address = s.StudentAddress == **null** ||

includeAddress == **false** ? **null** :

**new** AddressViewModel(){

StudentId = s.StudentAddress.StudentID,

Address1 = s.StudentAddress.Address1,

Address2 = s.StudentAddress.Address2,

City = s.StudentAddress.City,

State = s.StudentAddress.State

}

}).ToList<StudentViewModel>();

}

**if** (students.Count == 0)**return** NotFound();

**return** Ok(students);

}

**public** IHttpActionResult PostNewStudent(StudentViewModel student){

**if** (!ModelState.IsValid) **return** BadRequest("Not a valid data");

**using** (**var** ctx = **new** SchoolDBEntities()){

ctx.Students.Add(**new** Student(){

StudentID = student.Id,

FirstName = student.FirstName,

LastName = student.LastName

});

ctx.SaveChanges();

}

**return** Ok();

}

**public** IHttpActionResult Put(StudentViewModel student){

**if** (!ModelState.IsValid)**return** BadRequest("Not a valid data");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** existingStudent = ctx.Students.Where(

s => s.StudentID == student.Id).FirstOrDefault<Student>();

**if** (existingStudent != **null**){

existingStudent.FirstName = student.FirstName;

existingStudent.LastName = student.LastName;

ctx.SaveChanges();

}

**else** **return** NotFound();

}

**return** Ok();

}

**public** IHttpActionResult Delete(**int** id){

**if** (id <= 0)**return** BadRequest("Not a valid studet id");

**using** (**var** ctx = **new** SchoolDBEntities()){

**var** student = ctx.Students

.Where(s => s.StudentID == id)

.FirstOrDefault();

ctx.Entry(student).State = System.Data.Entity.EntityState.Deleted;

ctx.SaveChanges();

}

**return** Ok();

}

}

}

**Step 1:**

First, create a console application in Visual Studio for Desktop.

**Step 2:**

Open NuGet Package Manager console from **TOOLS** -> **NuGet Package Manager** -> **Package Manager Console** and execute following command.

Install-Package Microsoft.AspNet.WebApi.Client

**Step 3:**

Now, create a Student model class because we will send and receive Student object to our Web API.

Example: Model Class

**public class** Student{

**public int** Id { get; set; }

**public string** Name { get; set; }

}

## **Send GET Request**

The following example sends an HTTP GET request to Student Web API and displays the result in the console.

Example: Send HTTP GET Request using HttpClient

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**using** System.Net.Http;

**using** System.Net.Http.Headers;

**namespace** HttpClientDemo{

**class** Program{

**static void** Main(**string**[] args){

**using** (**var** client = **new** HttpClient()){

client.BaseAddress = **new** Uri("http://localhost:60464/api/");

//HTTP GET

**var** responseTask = client.GetAsync("student");

responseTask.Wait();

**var** result = responseTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<Student[]>();

readTask.Wait();

**var** students = readTask.Result;

**foreach** (**var** student **in** students){

Console.WriteLine(student.Name);

}

}

}

Console.ReadLine();

}

}

}

Let's understand the above example step by step.

First, we have created an object of HttpClient and assigned the base address of our Web API. The GetAsync() method sends a http GET request to the specified url. The GetAsync() method is asynchronous and returns a Task. Task.wait() suspends the execution until GetAsync() method completes the execution and returns a result.

Once the execution is complete, we get the result from Task using Task.result which is HttpResponseMessage. Now, you can check the status of a http response using IsSuccessStatusCode. Read the content of the result using ReadAsAsync() method.

Thus, you can send http GET request using HttpClient object and process the result.

## **Send POST Request**

Similarly, you can send HTTP POST request using PostAsAsync() method of HttpClient and process the result the same way as GET request.

The following example sends http POST request to our Web API. It posts Student object as json and gets the response.

Example: Send HTTP POST Request using HttpClient

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**using** System.Net.Http;

**using** System.Net.Http.Headers;

**namespace** HttpClientDemo{

**class** Program{

**static void** Main(**string**[] args){

**var** student = **new** Student() { Name = "Steve" };

**var** postTask = client.PostAsJsonAsync<Student>("student", student);

postTask.Wait();

**var** result = postTask.Result;

**if** (result.IsSuccessStatusCode){

**var** readTask = result.Content.ReadAsAsync<Student>();

readTask.Wait();

**var** insertedStudent = readTask.Result;

Console.WriteLine("Student {0} inserted with id: {1}",

insertedStudent.Name, insertedStudent.Id);

}

**else** Console.WriteLine(result.StatusCode);

}

}

}

The following table lists all the methods of HttpClient to send different HTTP requests.

| Method Name | Description |
| --- | --- |
| GetAsync | Sends a GET request to the specified Uri as an asynchronous operation. |
| GetByteArrayAsync | Sends a GET request to the specified Uri and returns the response body as a byte array in an asynchronous operation. |
| GetStreamAsync | Sends a GET request to the specified Uri and returns the response body as a stream in an asynchronous operation. |
| GetStringAsync | Sends a GET request to the specified Uri and returns the response body as a string in an asynchronous operation. |
| PostAsync | Sends a POST request to the specified Uri as an asynchronous operation. |
| PostAsJsonAsync | Sends a POST request as an asynchronous operation to the specified Uri with the given value serialized as JSON. |
| PostAsXmlAsync | Sends a POST request as an asynchronous operation to the specified Uri with the given value serialized as XML. |
| PutAsync | Sends a PUT request to the specified Uri as an asynchronous operation. |
| PutAsJsonAsync | Sends a PUT request as an asynchronous operation to the specified Uri with the given value serialized as JSON. |
| PutAsXmlAsync | Sends a PUT request as an asynchronous operation to the specified Uri with the given value serialized as XML. |
| DeleteAsync | Sends a DELETE request to the specified Uri as an asynchronous operation. |

Visit MSDN to know all the members of [HttpClient](https://msdn.microsoft.com/en-us/library/system.net.http.httpclient(v=vs.118).aspx) and [HttpClientExtension](https://msdn.microsoft.com/en-us/library/system.net.http.httpclientextensions(v=vs.118).aspx" \t "_blank).

# **Configure Dependency Injection with Web API**

Here you will learn how to configure and use IoC container for dependency injection with Web API.

There are many IoC containers available for dependency injection such as Ninject, Unity, castleWidsor, structuremap etc. Here we will use Ninject for dependency injection.

The following is our sample Web API that uses instance of a class that implements IRepository.

Example: Simple Web API Controller

**public class** StudentController : ApiController{

**private** IRepository \_repo = **null**;

**public** StudentController(IRepository repo){

\_repo = repo;

}

**public** IList<Student> Get(){

**return** \_repo.GetAll();

}

}

The following are IRepository and StudentRepository classes.

Example: Repository

**public interface** IRepository{

IList<Student> GetAll();

}

**public class** StudentRepository : IRepository{

**public** IList<Student> GetAll(){

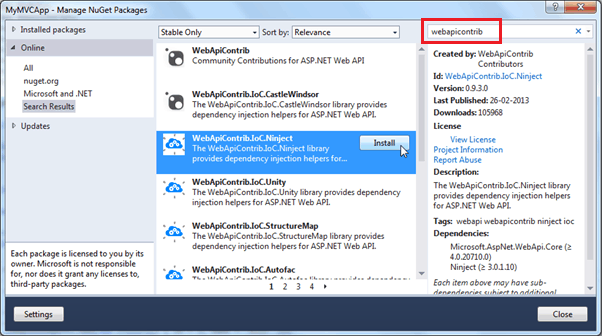
//return students from db here

}

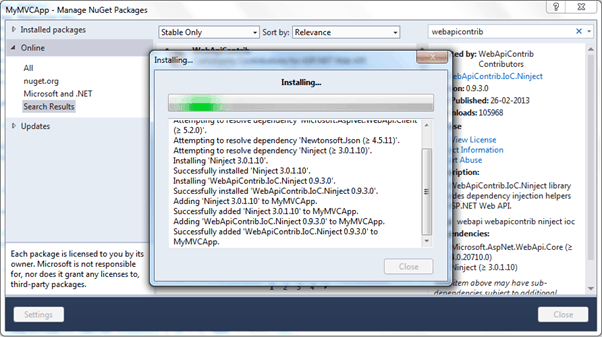
}

Now, let's use Ninject which will inject StudentRepository class in StudentController.

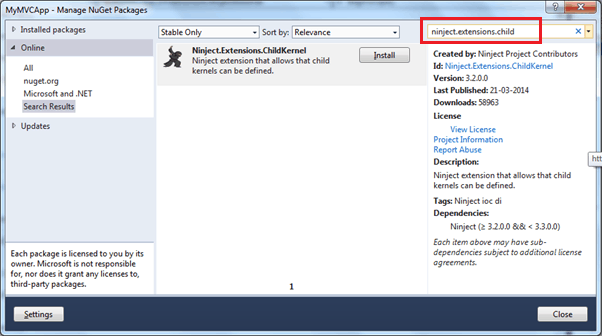
First of all, you need to install Ninject library for Web API using NuGet. To do this, right click on your project in the solution explorer -> click on **Manage NuGet packages..**. This will open NuGet popup. Now search for webapicontrib in the search box as shown below.

Web API Configuration

As you can see, this will list all the IoC containers for Web API. Select WebApiContrib.IoC.Ninject and click **Install**.

Web API Configuration

Now, you need to install Ninject.Extensions.ChildKernel. Search for it and install it.

Web API Configuration

So now after installing necessary packages for Ninject, we need to configure it.

To use dependency injection with Web API, we need to create a resolver class that implements IDependencyResolver interface. Here, we have created NinjectResolver class in Infrastructure folder in our Web API project as shown below.

Example: DI Resolver

**public class** NinjectResolver : IDependencyResolver{

**private** IKernel kernel;

**public** NinjectResolver() : **this**(**new** StandardKernel()){

}

**public** NinjectResolver(IKernel ninjectKernel, **bool** scope = **false**){

kernel = ninjectKernel;

**if** (!scope) AddBindings(kernel);

}

**public** IDependencyScope BeginScope(){

**return new** NinjectResolver(AddRequestBindings(

**new** ChildKernel(kernel)), **true**);

}

**public object** GetService(Type serviceType){

**return** kernel.TryGet(serviceType);

}

**public** IEnumerable<**object**> GetServices(Type serviceType){

**return** kernel.GetAll(serviceType);

}

**public void** Dispose(){

}

**private void** AddBindings(IKernel kernel){

// singleton and transient bindings go here

}

**private** IKernel AddRequestBindings(IKernel kernel){

kernel.Bind<IRepository>().To<StudentRepository>().InSingletonScope();

**return** kernel;

}

}

Now, we need to configure NijectResolver with Web API in the WebApiConfig class as shown below.

Example: Set DI Resolver

**public static class** WebApiConfig{

**public static void** Register(HttpConfiguration config){

config.DependencyResolver = **new** NinjectResolver();

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

**name**: "DefaultApi",

**routeTemplate**: "api/{controller}/{id}",

**defaults**: **new** { id = RouteParameter.Optional }

);

}

}

As you can see above, HttpConfiguration.DependencyResolver is set to NinjectResolver. So now, the Web API will use NinjectResolver class to create the objects it needs.

Thus, you can configure IoC container with Web API.

# **Web API Hosting**

In this section, you will learn how to host a Web API.

The Web API application can be hosted in two ways.

* IIS Hosting
* Self-Hosting

## **IIS Hosting**

Web API can be hosted under IIS, in the same way as a web application. You have learned to create a Web API in the previous section. As you have seen there, a Web API is created with ASP.NET MVC project by default. So, when you host your MVC web application under IIS it will also host a Web API that uses the same base address.

## **Self-Hosting**

You can host a Web API as a separate process than ASP.NET. It means you can host a Web API in console application or windows service or OWIN or any other process that is managed by .NET framework.

You need to do the following steps to self-host a web API.

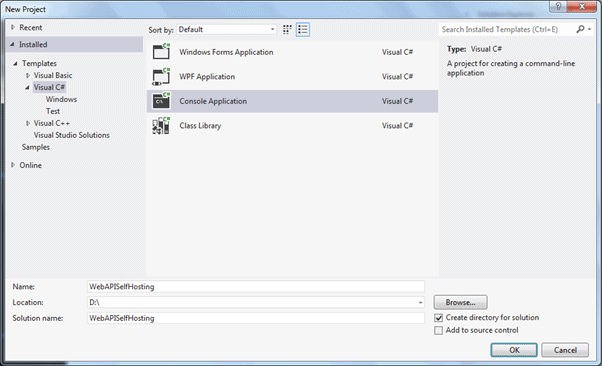
1. Use HttpConfiguration to configure a Web API
2. Create HttpServer and start listening to incoming http requests

Let's see how to host a simple Web API in console application.

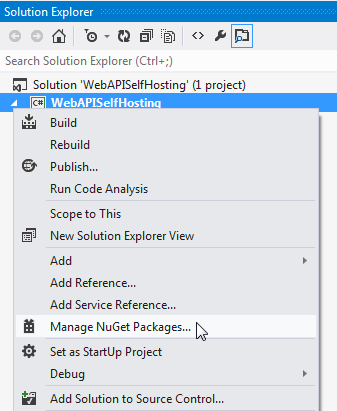
First, create a console project in Visual Studio for Desktop.

 Note:

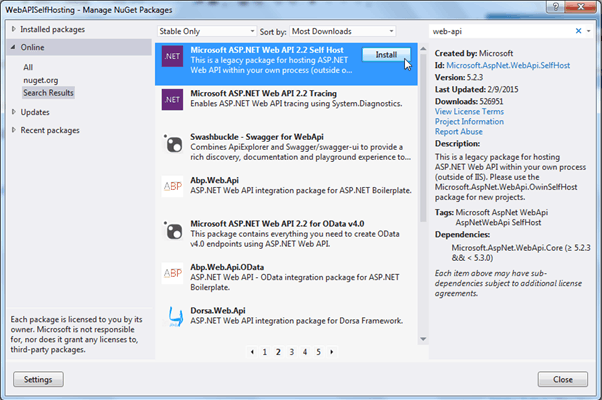
You must open Visual Studio in Administration mode.

 Create Console Application

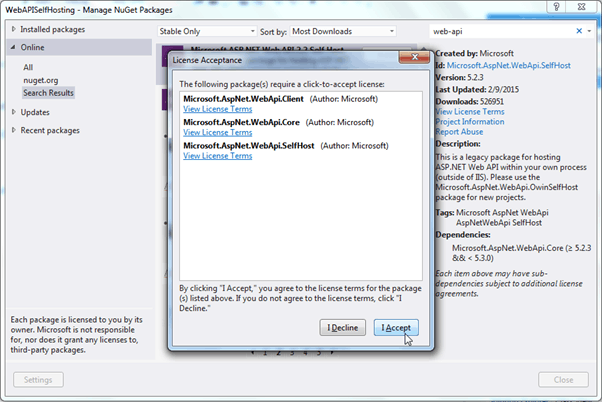
Now, you need to add Microsoft ASP.NET Web API 2.x Self Host package using NuGet. Right click on project in the Solution Explorer window and select **Manage NuGet Packges..**

Open NuGet Manager

In the Manage NuGet Packages window, select the Online option in the left pan and search for web-api. This will list all the packages for Web API. Now, look for **Microsoft.AspNet.WebApi.SelfHost** package and click **Install**.

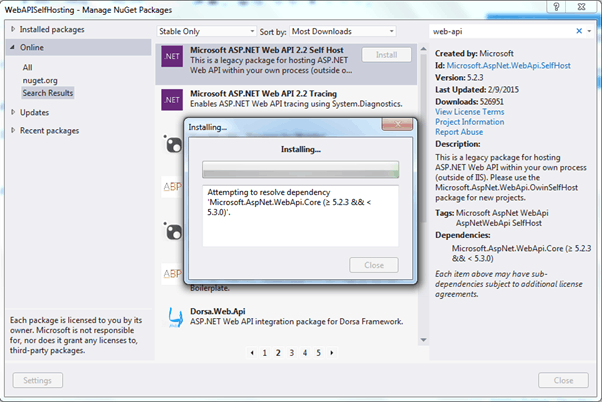
 Install Web API Self Host Package

Click on **the Accept** button in the License Acceptance window.



Accept License Agreement

This will install the package into your project.



Install Web API Self Hosting Package

Now write the following code in the Main method of Program class.

Example: Self-Hosting Web API

**class** Program{

**static void** Main(**string**[] args){

**var** config = **new** HttpSelfHostConfiguration("http://localhost:1234");

**var** server = **new** HttpSelfHostServer(config,**new** MyWebAPIMessageHandler());

**var** task = server.OpenAsync();

task.Wait();

Console.WriteLine("Web API Server has started at http://localhost:1234");

Console.ReadLine();

}

}

In the above code, first we created an object of HttpSelfHostConfiguration class by passing uri location. Then, we created an object of HttpSelfHostServer by passing config and HttpMessageHandler object. And then we started listening for incoming requests by calling server.OpenAsync() method. This will listen requests asynchronously, so it will return Task object.

Create MyWebAPIMessageHandler class and write the following code.

Example: MessageHandler

**class** MyWebAPIMessageHandler : HttpMessageHandler{

**protected override** Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

System.Threading.CancellationToken cancellationToken){

**var** task = **new** Task<HttpResponseMessage>(() => {

**var** resMsg = **new** HttpResponseMessage();

resMsg.Content = **new** StringContent("Hello World!");

**return** resMsg;

});

task.Start();

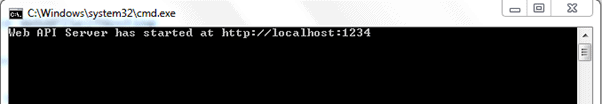
**return** task;

}

}

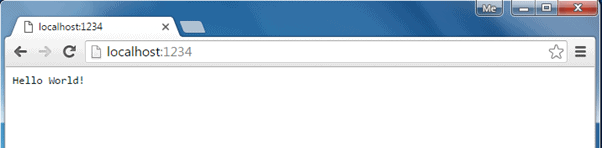
Thus, you can create simple console application and host simple Web API that returns "Hello World!" to every request.

Run the console application using Ctrl + F5.



Run Console Application

Open web browser and enter http://localhost:1234/ and see the result.

 Response in Browser

## **Hosting Controller Infrastructure**

You can use the same ASP.NET routing and ApiController capabilities of ASP.NET Hosting in self-hosting.

In the same self-hosting console application, create simple HomeController class as shown below.

Example: Web API Controller

**public class** HomeController : ApiController{

**public string** Get(){

**return** "Hello World!";

}

**public string** Get(**string** name){

**return** "Hello " + name;

}

}

Now, in the Main() method, configure a default route using config object as shown below.

Example: Self Hosting Web API

**static void** Main(**string**[] args){

**var** config = **new** HttpSelfHostConfiguration("http://localhost:1234");

config.Routes.MapHttpRoute("default",

"api/{controller}/{id}",

**new** { controller = "Home",

id = RouteParameter.Optional });

**var** server = **new** HttpSelfHostServer(config);

**var** task = server.OpenAsync();

task.Wait();

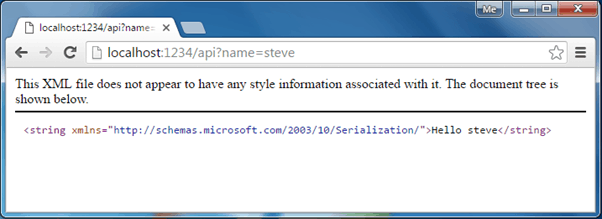
Console.WriteLine("Web API Server has started at http://localhost:1234");

Console.ReadLine();

}

Please note that we removed an object of MessageHandler when creating an object of HttpSelfHostServer.

Now, run the console application by pressing *Ctrl + F5*. Open the browser and enter http://localhost:1234/api or http://localhost:1234/api?name=steve and see the result as shown below.

Web API Response