Web API Controller

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.

Web API Controller is similar to ASP.NET MVC controller. It handles incoming HTTP requests and send response 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.

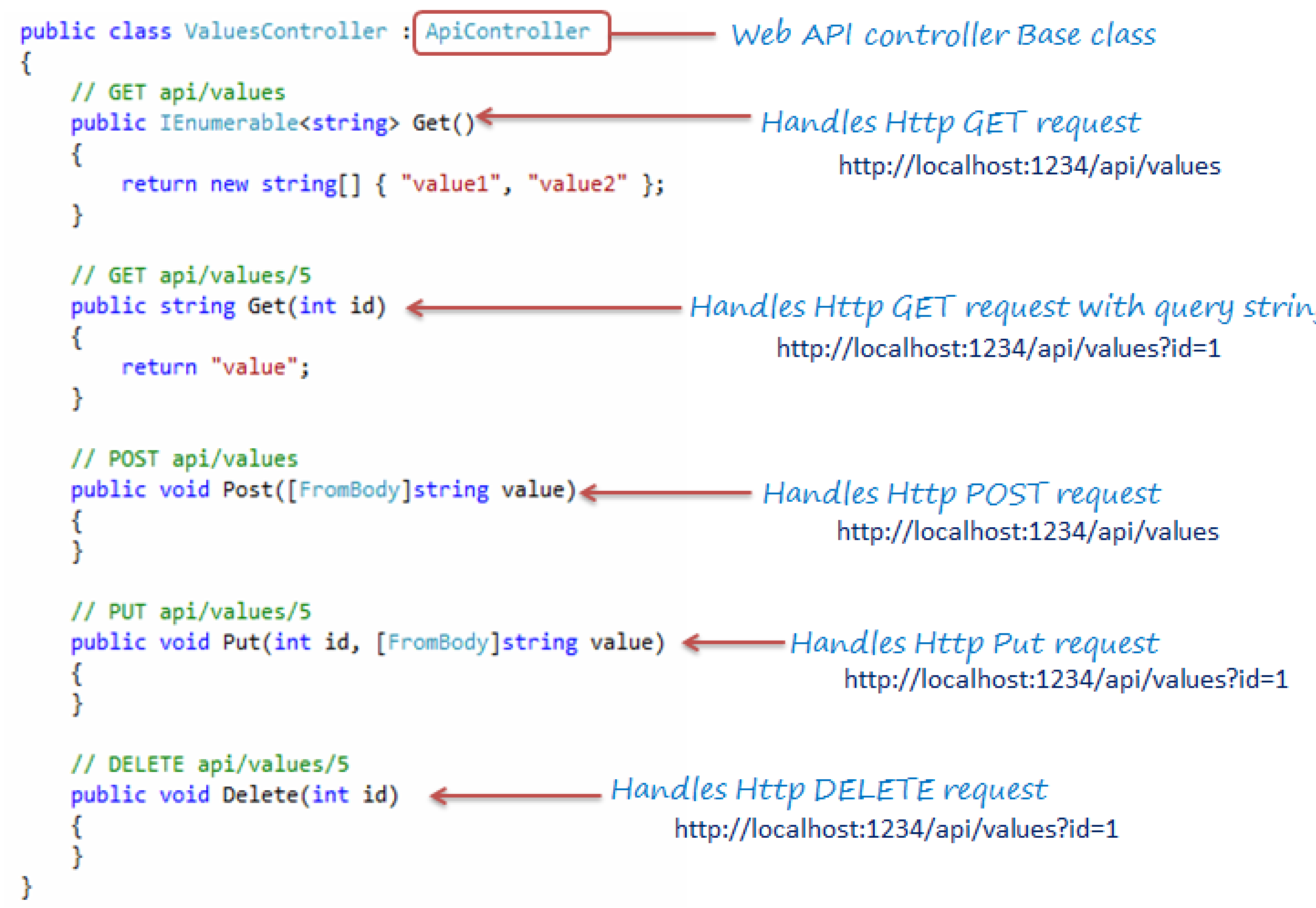
Example: Simple Web API Controller

```
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/student
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET: api/student/5
        public string Get(int id)
        {
            return "value";
        }

        // POST: api/student
        public void Post([FromBody]string value)
        {
        }

        // PUT: api/student/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE: api/student/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.



```
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)
    {
    }
}
```

Web API controller Base class
Handles Http GET request
http://localhost:1234/api/values
Handles Http GET request with query strin
http://localhost:1234/api/values?id=1
Handles Http POST request
http://localhost:1234/api/values
Handles Http Put request
http://localhost:1234/api/values?id=1
Handles Http DELETE request
http://localhost:1234/api/values?id=1

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. 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, 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

GET	Get()
	get()
	GET()
	GetAllStudent()
	*any name starting with Get *

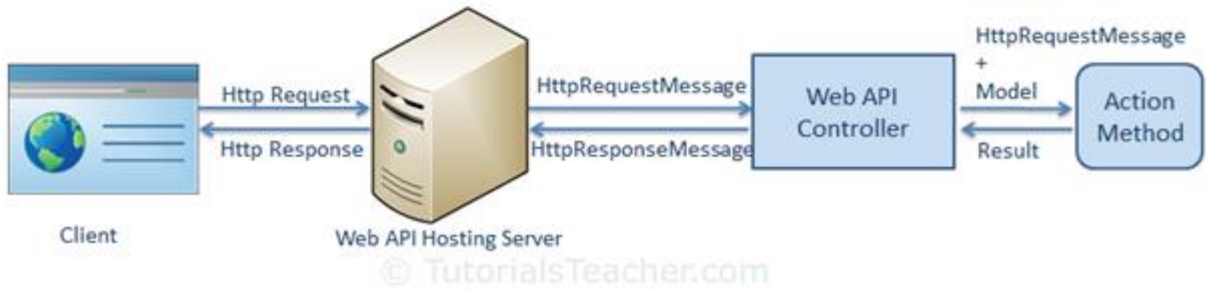
POST	Post()
	post()
	POST()
	PostNewStudent()
	any name starting with Post

PUT	Put()
	put()
	PUT()
	PutStudent()
	any name starting with Put

PATCH	Patch()
	patch()
	PATCH()
	PatchStudent()
	any name starting with Patch

DELETE	Delete()
	delete()
	DELETE()
	DeleteStudent()
	any name starting with Delete

The following figure illustrates the overall request/response pipeline.



Web API Request Pipeline

Visit Web API HTTP Message Life Cycle Poster for more details.

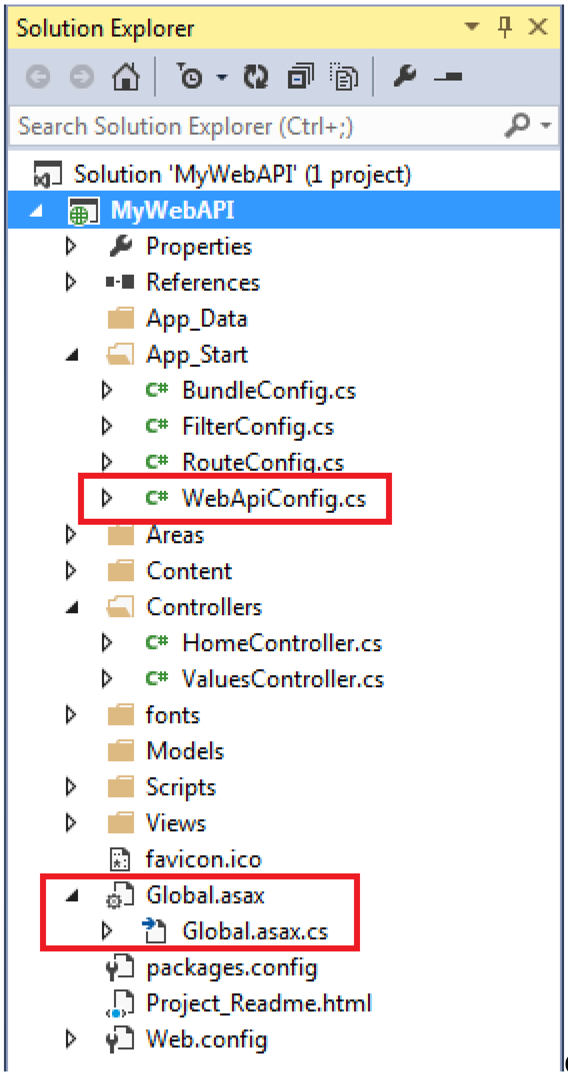
Difference between Web API and MVC controller

Web API Controller	
Derives from System.Web.Http.ApiController class	Derives from System.Web.M
Method name must start with Http verbs otherwise apply http verbs attribute.	Must apply appropriate Http
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
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 behaviour 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. 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 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 messa
MessageHandlers	Gets or sets the message handlers.
ParameterBindingRules	Gets the collection of rules for how parameters should be bound.

Property	Description
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

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 similar to 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 the 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 into route table in order 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. The "DefaultApi" route is added in the route table using MapHttpRoute() extension method. The `MapHttpRoute()` extension method internally creates a new instance of IHttpRoute 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.

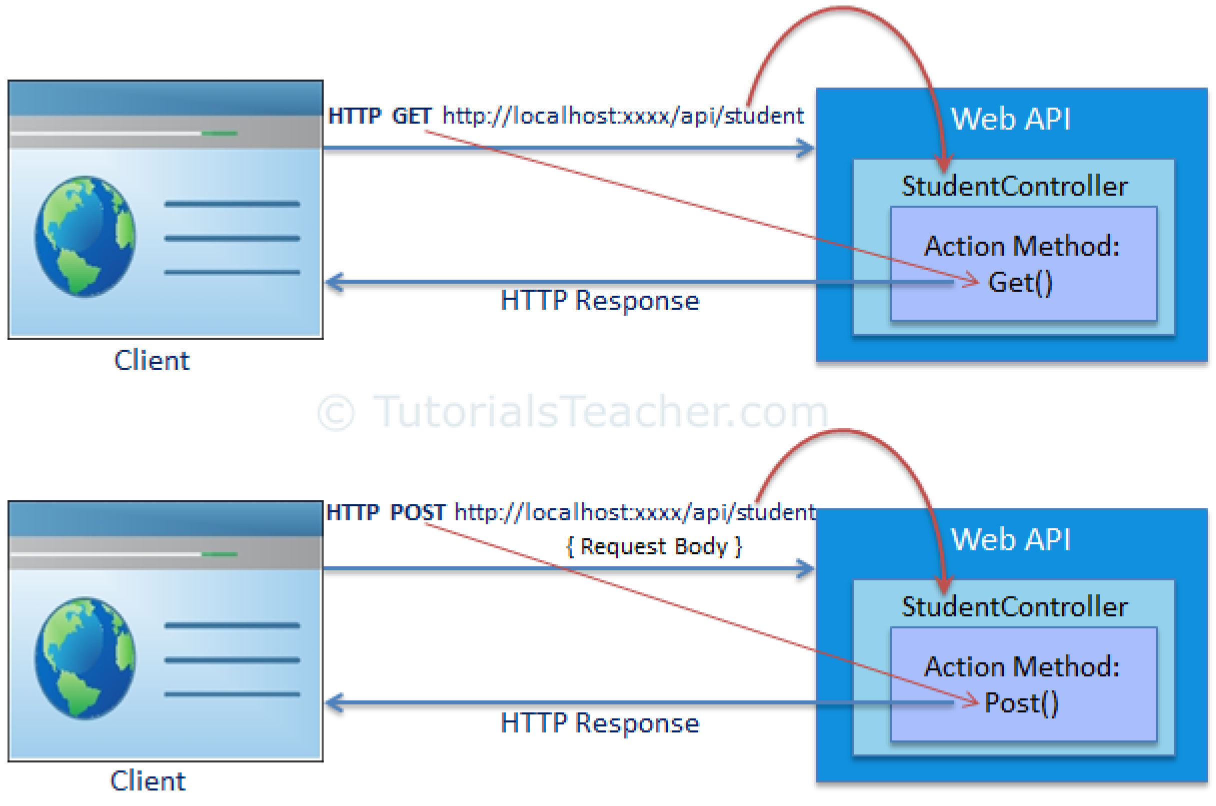
Sample HTTP GET Request

```
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. 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 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	
http://localhost:1234/api/course	GET	Ge
http://localhost:1234/api/product	POST	Pos
http://localhost:1234/api/teacher	PUT	Put

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 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.

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
GET	Primitive Type, Complex Type
POST	Primitive Type
PUT	Primitive Type
PATCH	Primitive Type
DELETE	Primitive Type, Complex Type

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 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, 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 resource. It can include request data into HTTP request body and also 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, 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 Post() method.

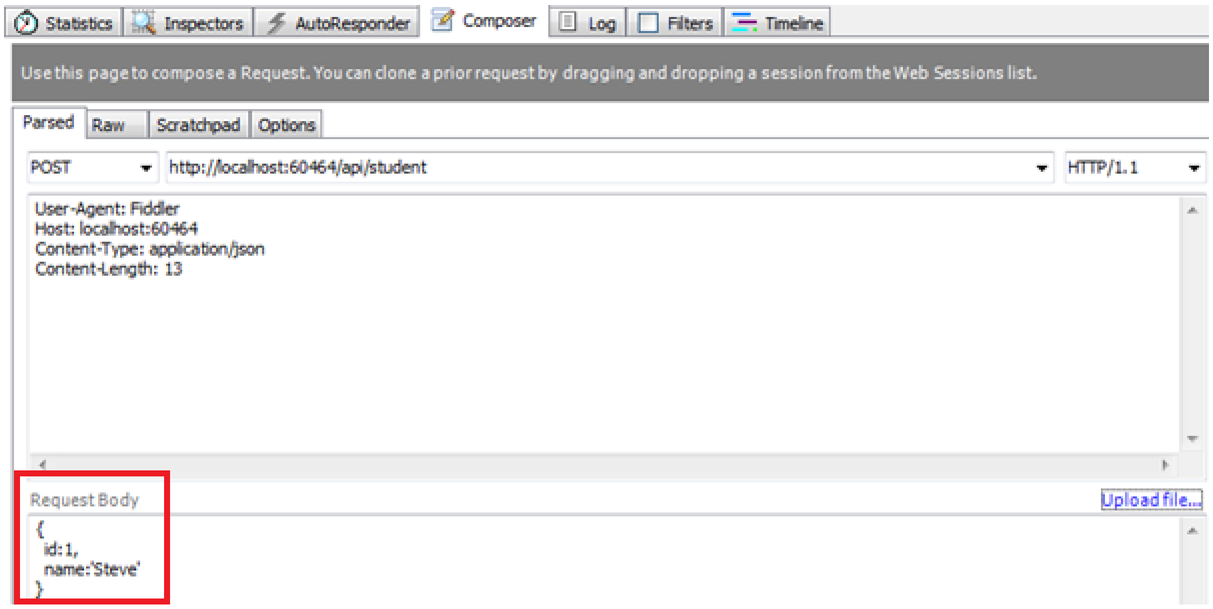
Now, consider the following Post() method with complex type parameter.

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.



Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the Web Sessions list.

```
POST    http://localhost:60464/api/student    HTTP/1.1
```

```
User-Agent: Fiddler
Host: localhost:60464
Content-Type: application/json
Content-Length: 13
```

```
{
 id:1,
 name:'Steve'
}
```

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 matches with the name of Student class properties (case-insensitive).

POST Method with Mixed Parameters

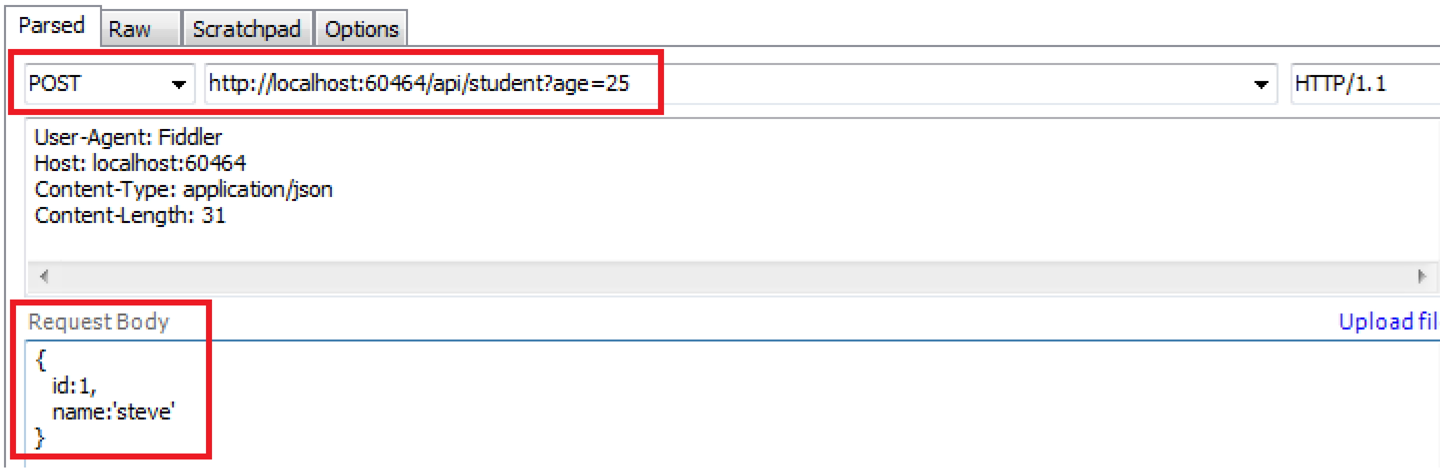
Post action method can include primitive and complex type parameters. Consider the following example.

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 parameter. 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.



```
POST http://localhost:60464/api/student?age=25 HTTP/1.1
User-Agent: Fiddler
Host: localhost:60464
Content-Type: application/json
Content-Length: 31

{
  id:1,
  name:'steve'
}
```

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 behaviour?

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, 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 Post method.

Example: FromUri

```
public class StudentController : ApiController
{
    public Student Post([FromUri]Student stud)
    {

    }
}
```

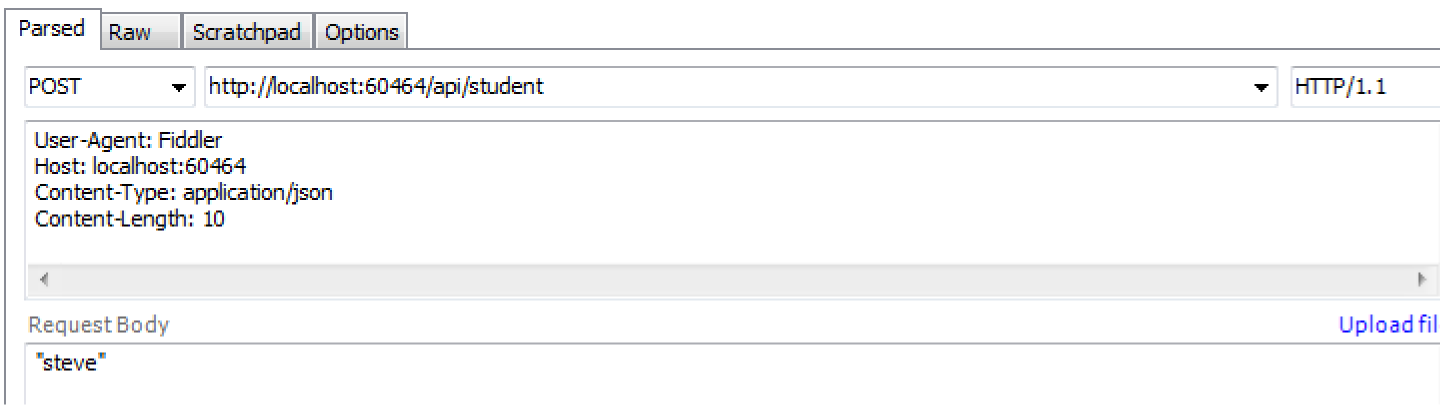
As you can see above, we have applied [FromUri] attribute with the Student parameter. Web API by default extracts the value of complex type from request body but here we have applied [FromUri] attribute. So now, Web API will extract the value of Student properties from the query string instead of request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



http://localhost:60464/api/student

```
User-Agent: Fiddler
Host: localhost:60464
Content-Type: application/json
Content-Length: 10
```

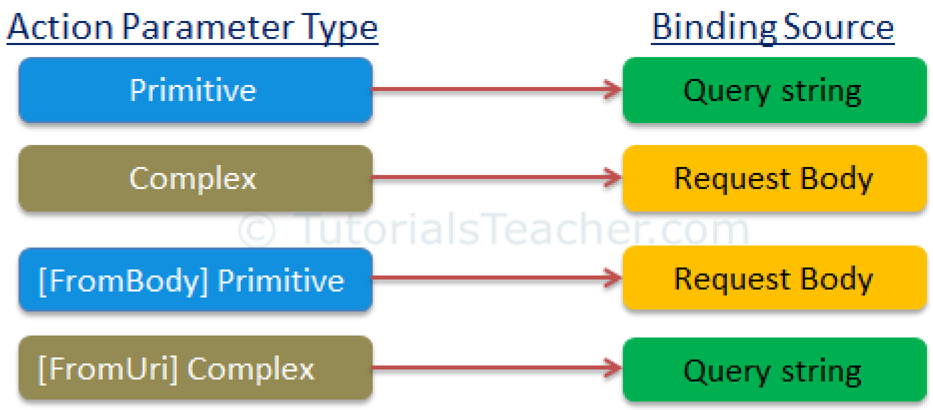
```
"steve"
```

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 following return types.

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

Void

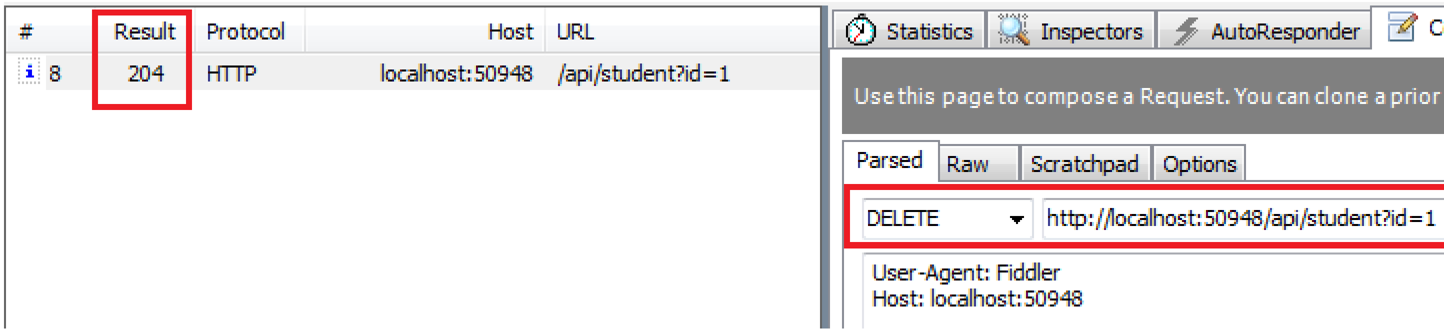
It's not necessary that all action methods must return something. It can have void return type.

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.



#	Result	Protocol	Host	URL
8	204	HTTP	localhost:50948	/api/student?id=1

DELETE http://localhost:50948/api/student?id=1

```
User-Agent: Fiddler
Host: localhost:50948
```

Void Response Status

Primitive or Complex Type

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

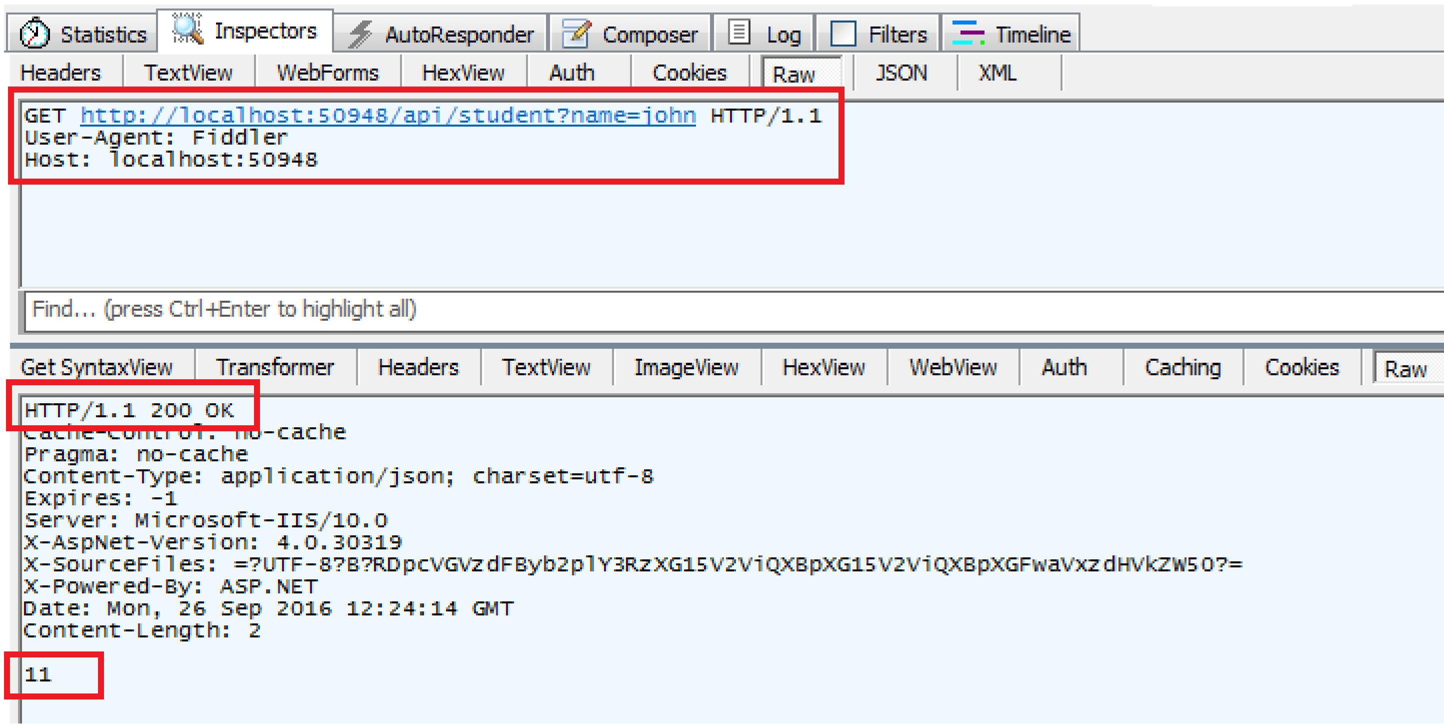
Consider the following Get action methods.

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.

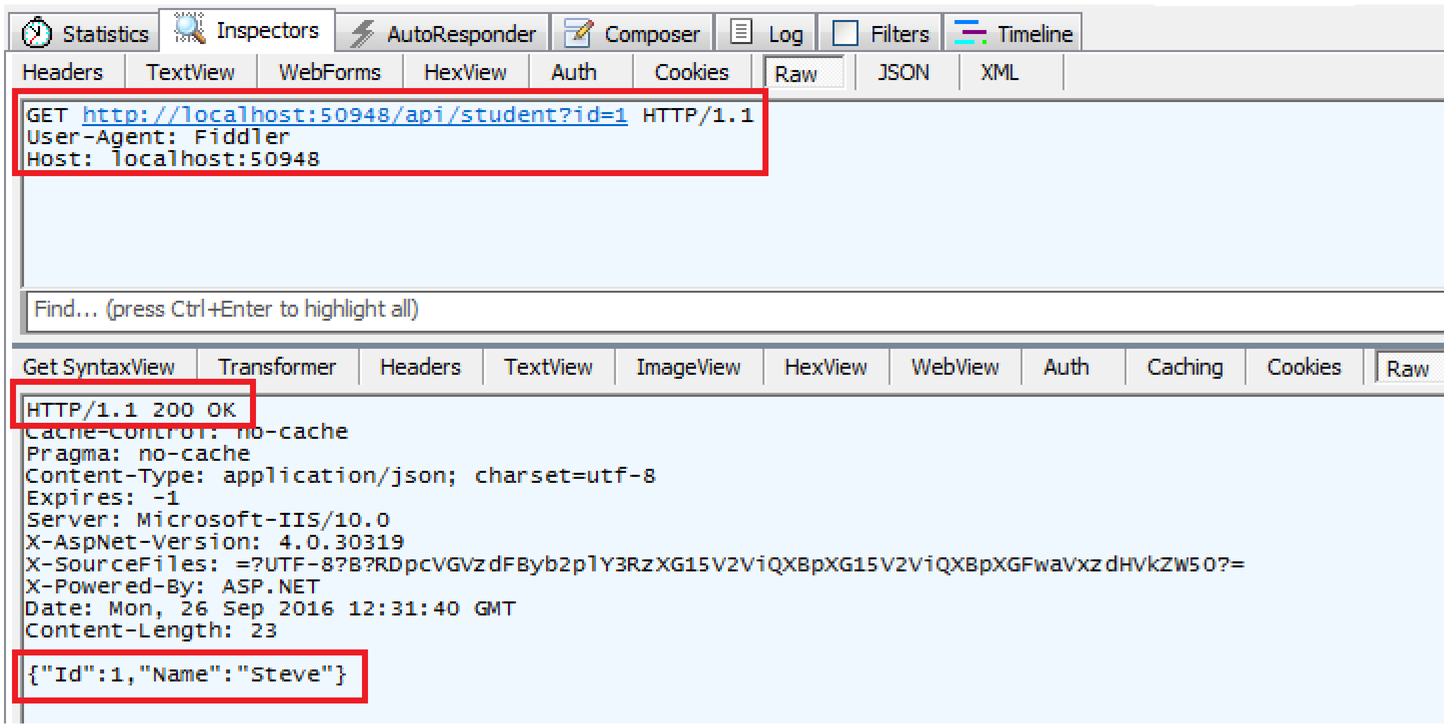


```
GET http://localhost:50948/api/student?name=john HTTP/1.1
User-Agent: Fiddler
Host: localhost:50948
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXG15V2ViQXBpXG15V2ViQXBpXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Mon, 26 Sep 2016 12:24:14 GMT
Content-Length: 2

11
```

Primitive Return Type in Response

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



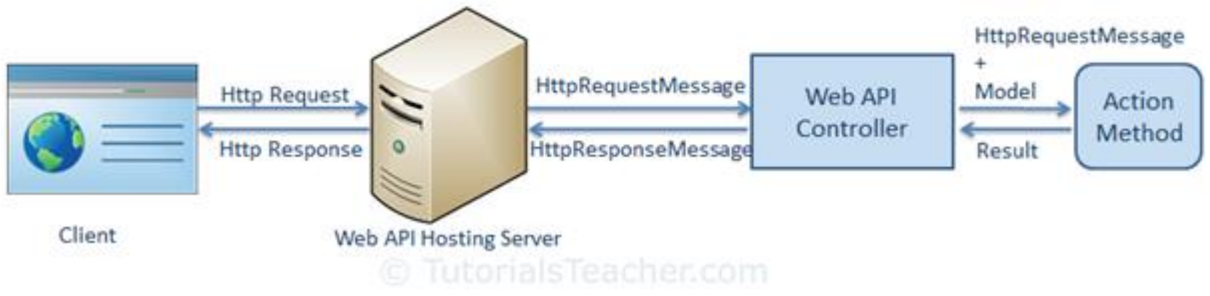
```
GET http://localhost:50948/api/student?id=1 HTTP/1.1
User-Agent: Fiddler
Host: localhost:50948
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXG15V2ViQXBpXG15V2ViQXBpXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Mon, 26 Sep 2016 12:31:40 GMT
Content-Length: 23

{"Id":1,"Name":"Steve"}
```

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 for more details.

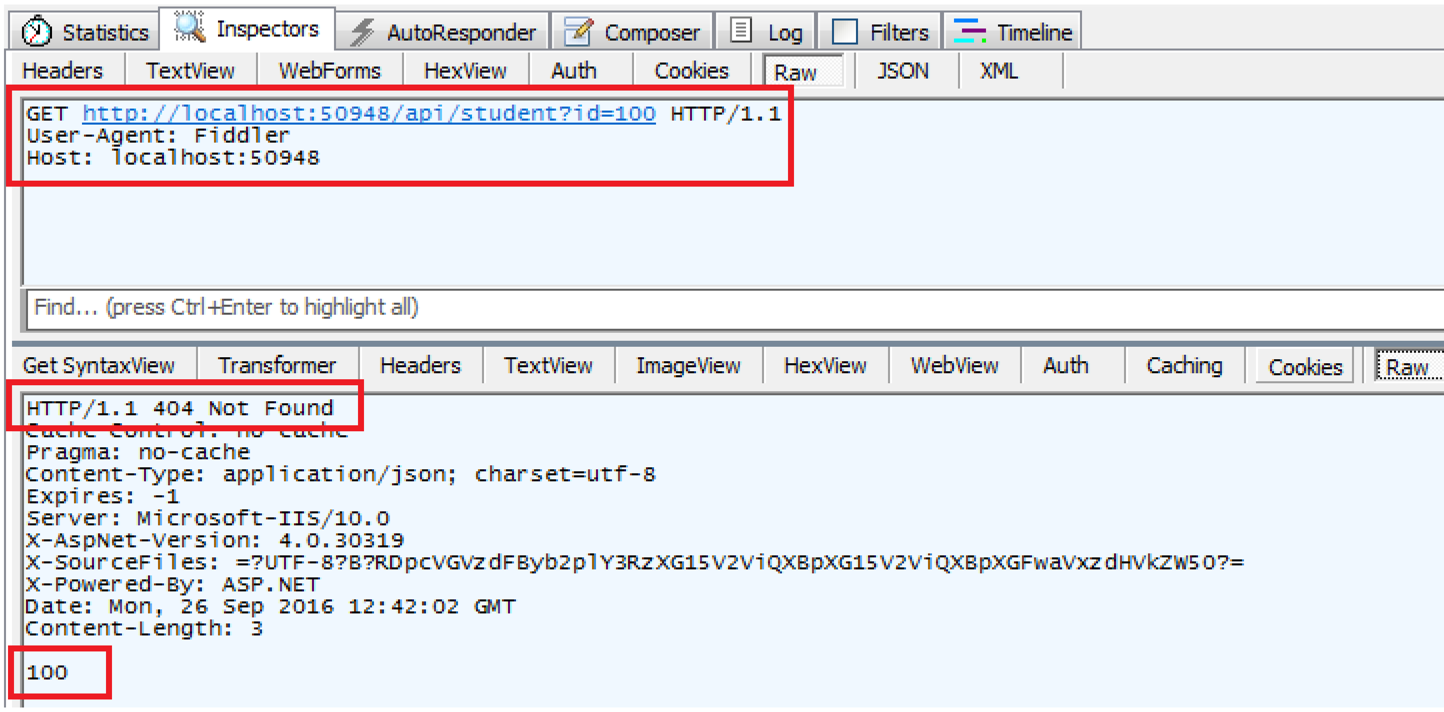
As you can see in the above figure, the Web API controller returns HttpResponseMessage object. You can also create and return an object of HttpResponseMessage directly from an action method.

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.

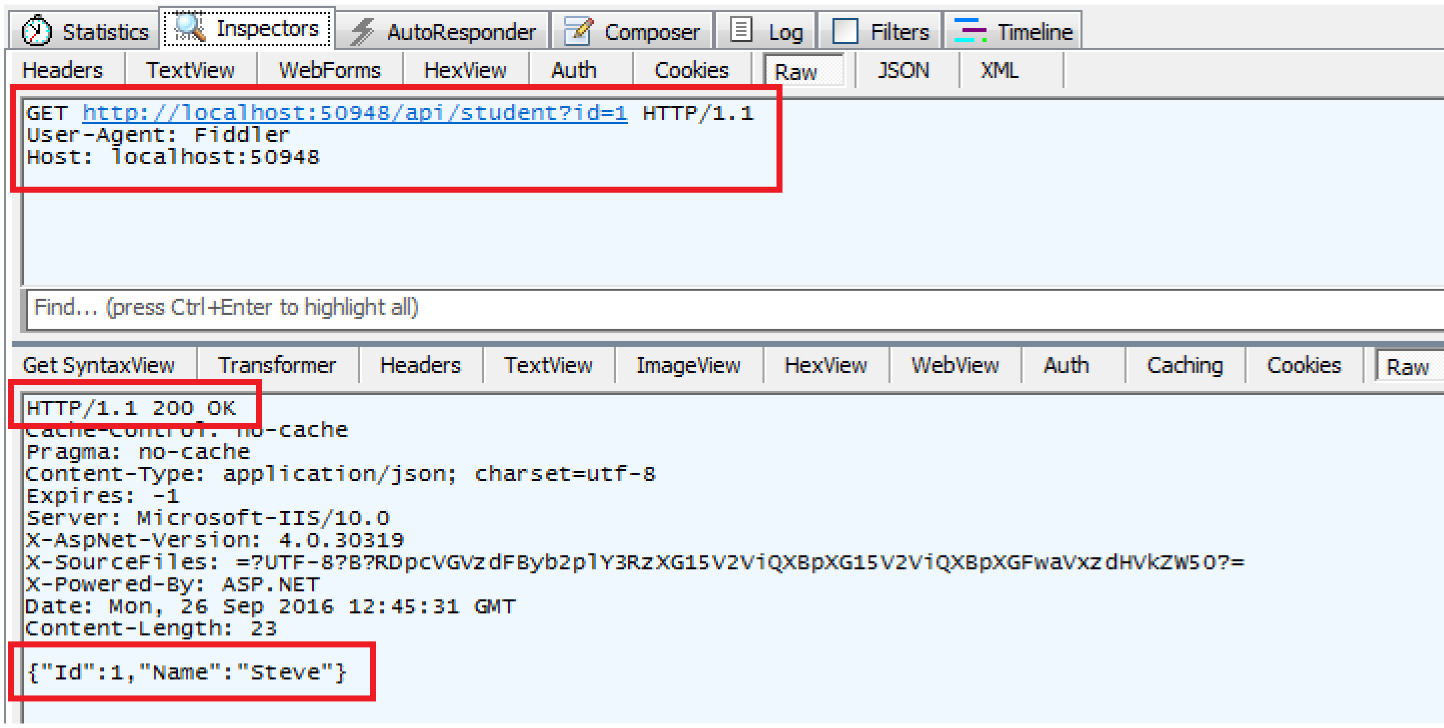
For example, an http GET request `http://localhost:xxxx/api/student?id=100` will get following response considering student with id=100 does not exists in the DB.



```
GET http://localhost:50948/api/student?id=100 HTTP/1.1
User-Agent: Fiddler
Host: localhost:50948
```

```
HTTP/1.1 404 Not Found
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXG15V2ViQXBpXG15V2ViQXBpXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Mon, 26 Sep 2016 12:42:02 GMT
Content-Length: 3

100
```
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 .



```
GET http://localhost:50948/api/student?id=1 HTTP/1.1
User-Agent: Fiddler
Host: localhost:50948
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXG15V2ViQXBpXG15V2ViQXBpXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Mon, 26 Sep 2016 12:45:31 GMT
Content-Length: 23

{"Id":1,"Name":"Steve"}
```

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 more or less similar to 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 implement 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 create
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.

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** attribute. 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 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'

}
```

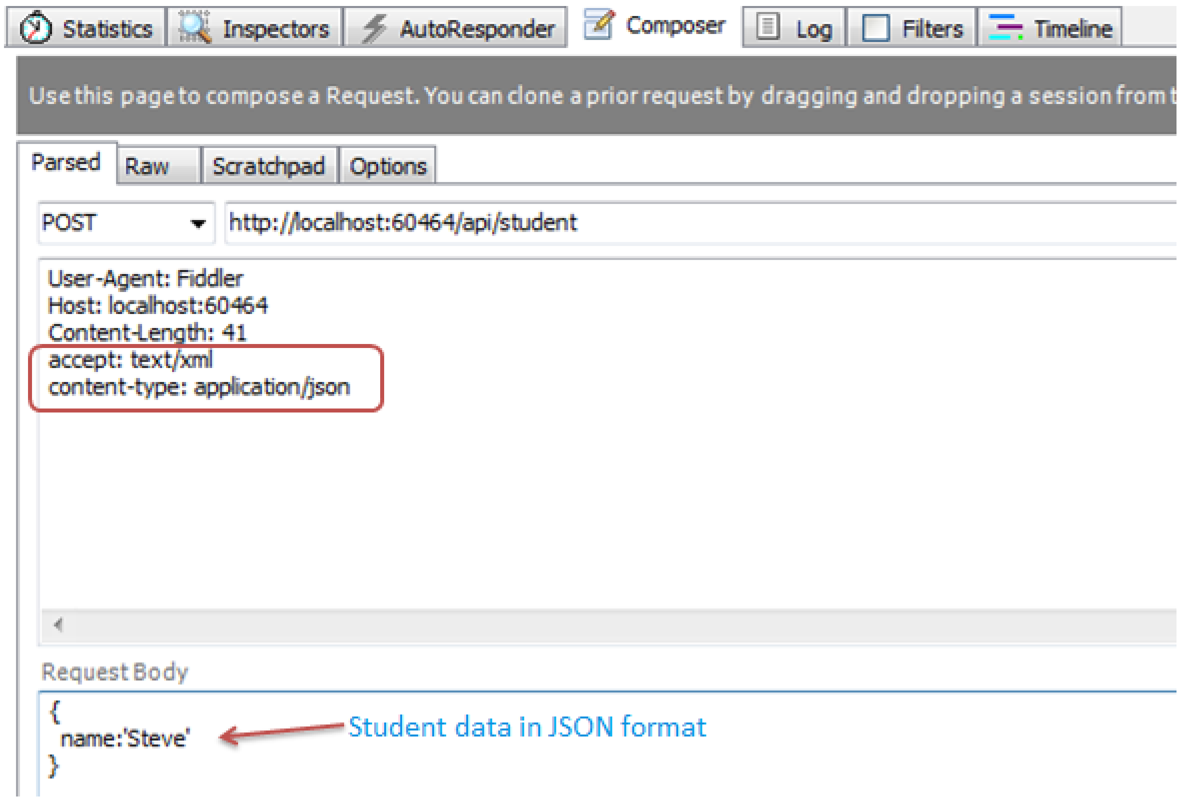
Web API converts request data into CLR object and also serialize CLR object into response data based on Accept and Content-Type headers. Web API includes built-in support for JSON, XML, BSON, and form-urlencoded data. It means it automatically converts request/response data into these formats OOB (out-of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

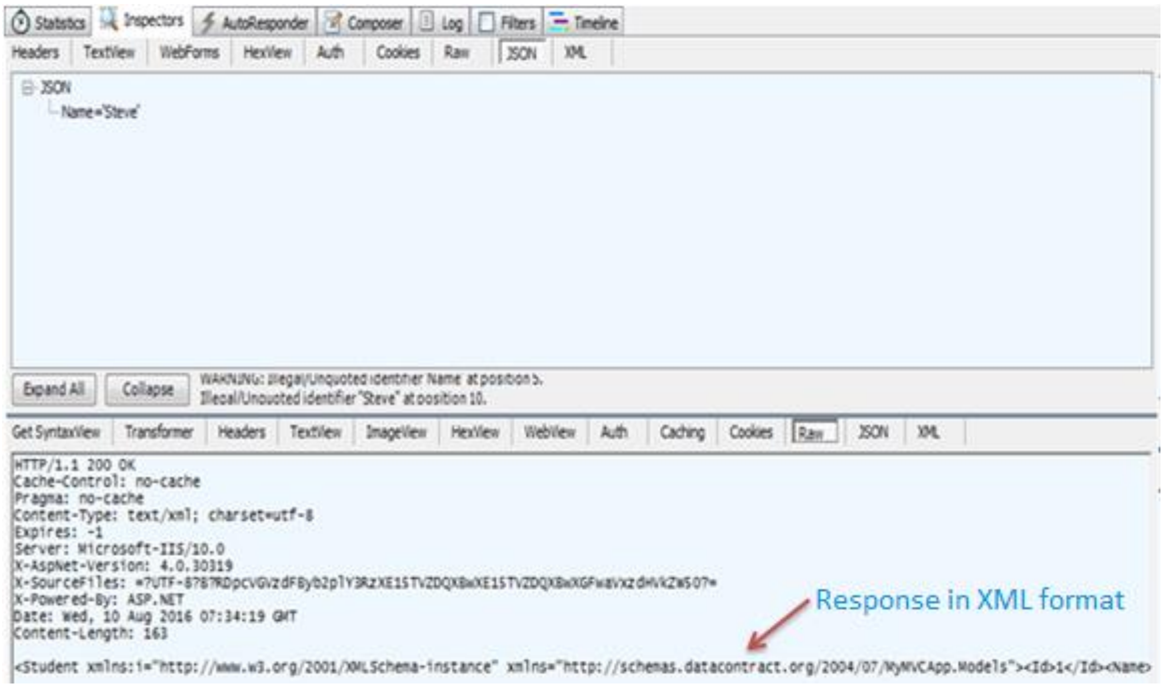

Statistics
Inspectors
AutoResponder
Composer
Log
Filters
Timeline
Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from t
Parsed
Raw
Scratchpad
Options
Request Body
Student data in JSON format

```
POST    http://localhost:60464/api/student

User-Agent: Fiddler
Host: localhost:60464
Content-Length: 41
accept: text/xml
content-type: application/json
```

```
{
 name:'Steve'
}
```

Request-Response Data Format

In the above figure, 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.



```
JSON
  Name='Steve'
```

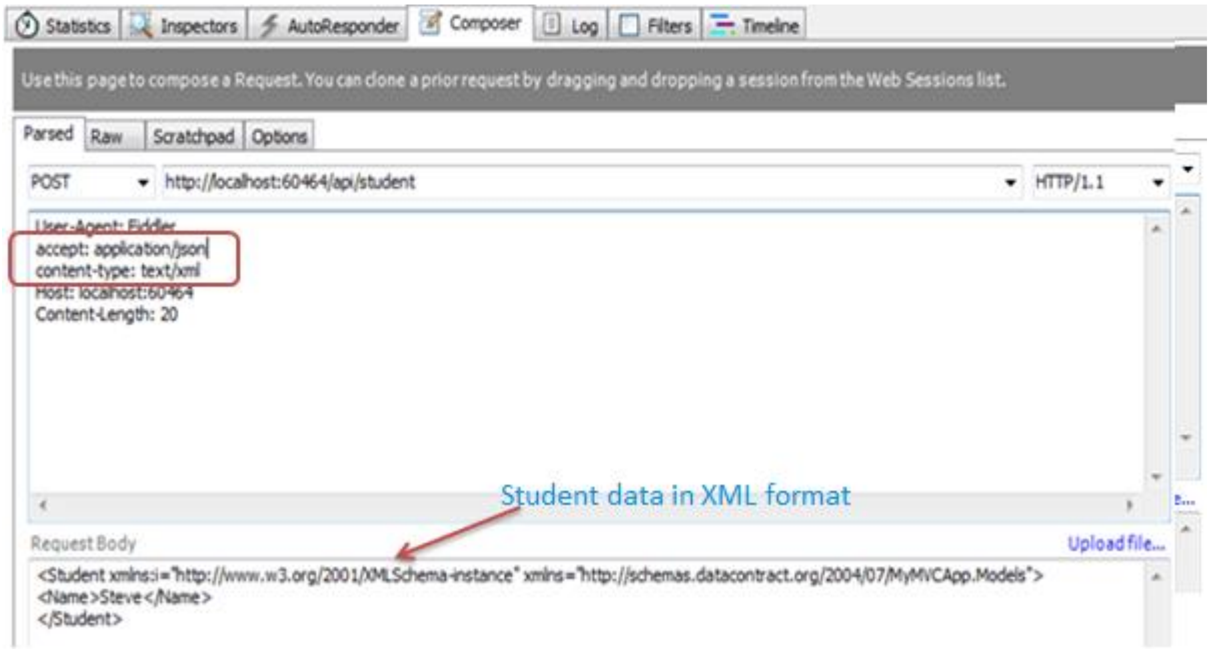
WARNING: Illegal/Unquoted identifier Name at position 5.
Illegal/Unquoted identifier "Steve" at position 10.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/xml; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXE15TVZDQXBwXE15TVZDQXBwXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Wed, 10 Aug 2016 07:34:19 GMT
Content-Length: 163

<Student xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/MyMVCApp.Models"><Id>1</Id><Name>
```

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.

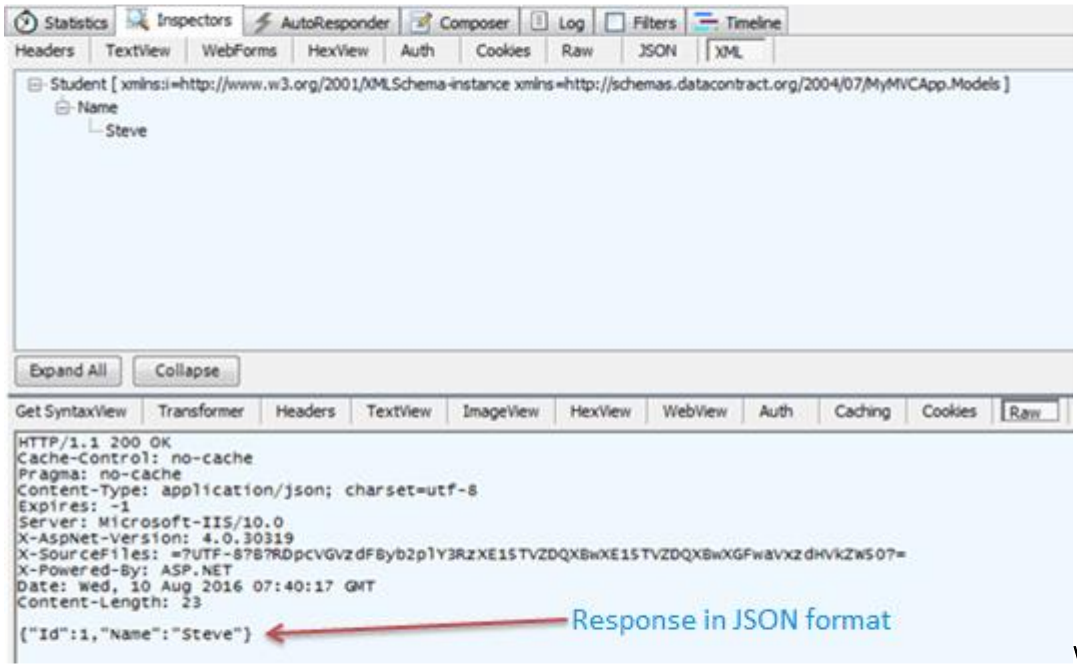

Statistics
Inspectors
AutoResponder
Composer
Log
Filters
Timeline
Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the Web Sessions list.
Parsed
Raw
Scratchpad
Options
Student data in XML format
Request Body
Upload file...

```
POST    http://localhost:60464/api/student    HTTP/1.1

User-Agent: Fiddler
accept: application/json
content-type: text/xml
Host: localhost:60464
Content-Length: 20
```

```
<Student xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/MyMVCApp.Models">
<Name>Steve</Name>
</Student>
```

Web API Request

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



```
Student [ xmlns:i=http://www.w3.org/2001/XMLSchema-instance xmlns=http://schemas.datacontract.org/2004/07/MyMVCApp.Models ]
  Name
    Steve
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?RDpcVGVzdFByb2plY3RzXE1STVZDQXBwXE1STVZDQXBwXGFwaVxzdHVkZW50?=
X-Powered-By: ASP.NET
Date: Wed, 10 Aug 2016 07:40:17 GMT
Content-Length: 23

{"Id":1,"Name":"Steve"}
```

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.

Web API Filters

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

Filters are actually 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 filter for specific purpose.

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

Filter Type	Interface	Class	
Simple Filter	IFilter	-	Defines the methods that are used in
Action Filter	IActionFilter	ActionFilterAttribute	Used to add extra logic before or afte
Authentication Filter	IAuthenticationFilter	-	Used to force users or clients to be au
Authorization Filter	IAuthorizationFilter	AuthorizationFilterAttribute	Used to restrict access to action meth
Exception Filter	IExceptionFilter	ExceptionFilterAttribute	Used to handle all unhandled excepti
Override Filter	IOverrideFilter	-	Used to customize the behaviour of o

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 than you must implement all the methods.

Visit MSDN to know all the classes and interfaces available in System.Web.Http.Filters.

Let's create 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
    }
}
```

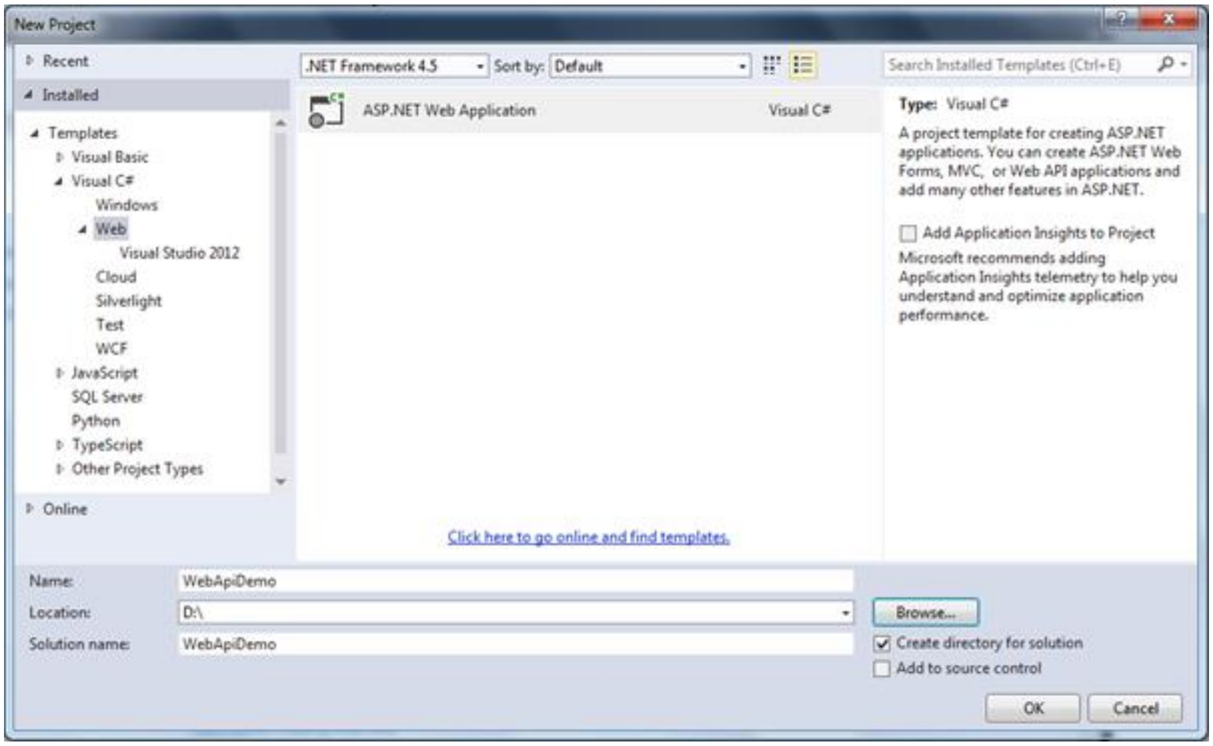
So now, it will log all the requests handled by above StudentController. Thus you can create filters for cross-cutting concerns.

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 2013 for Web express edition.

Open Visual Studio 2013 for Web and click on **File** menu -> **New Project..** This will open New Project popup as shown below.



.NET Framework 4.5 | Sort by: Default

ASP.NET Web Application — Visual C#

Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

☐ Add Application Insights to Project

Microsoft recommends adding Application Insights telemetry to help you understand and optimize application performance.

Name: WebApiDemo

Location: D:\

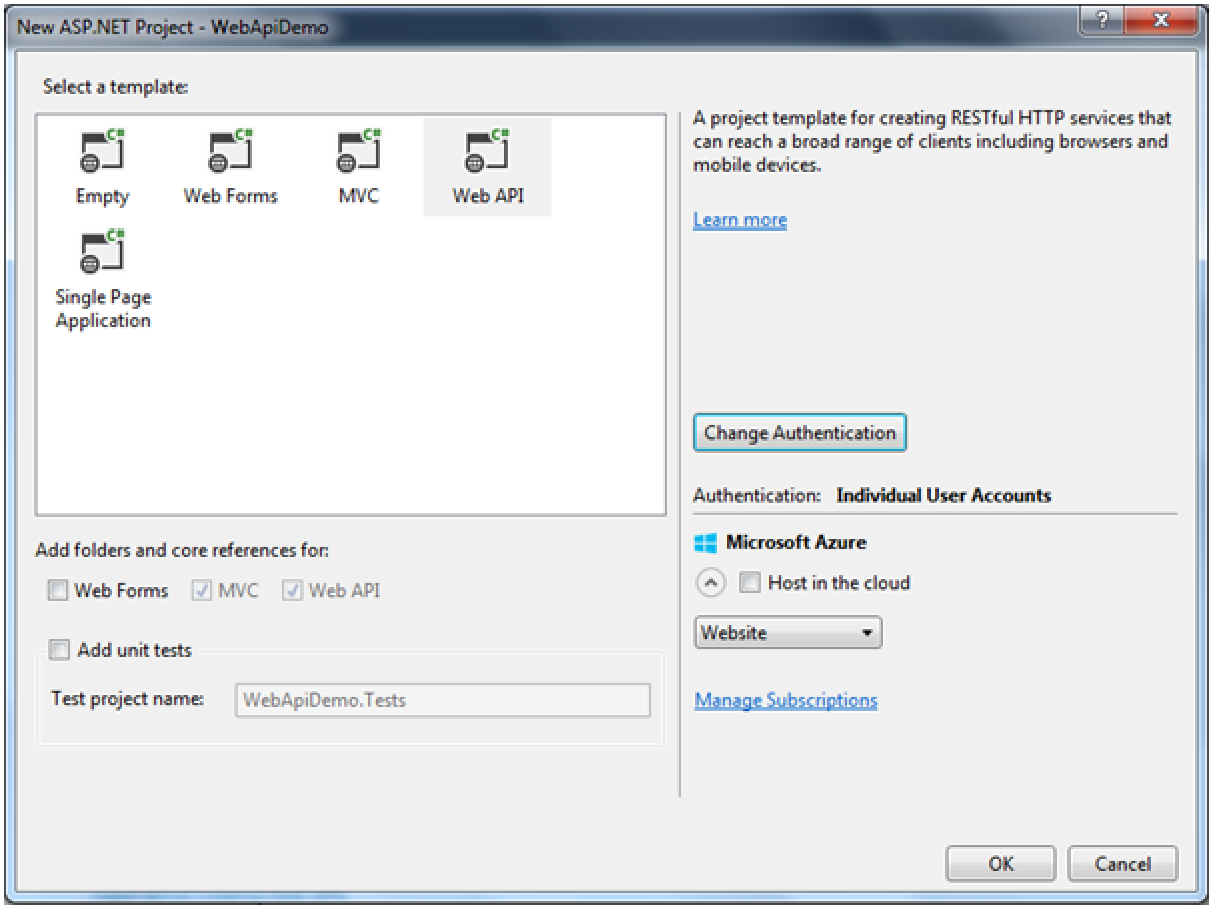
Solution name: WebApiDemo

☑ Create directory for solution

☐ Add to source control

Create Web API Project

In the New Project popup, select **Web** template under **Visual C#**. Enter 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.


New ASP.NET Project - WebApiDemo

Select a template:

A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Authentication: **Individual User Accounts**

Microsoft Azure

[ ] Host in the cloud

Website

Add folders and core references for:

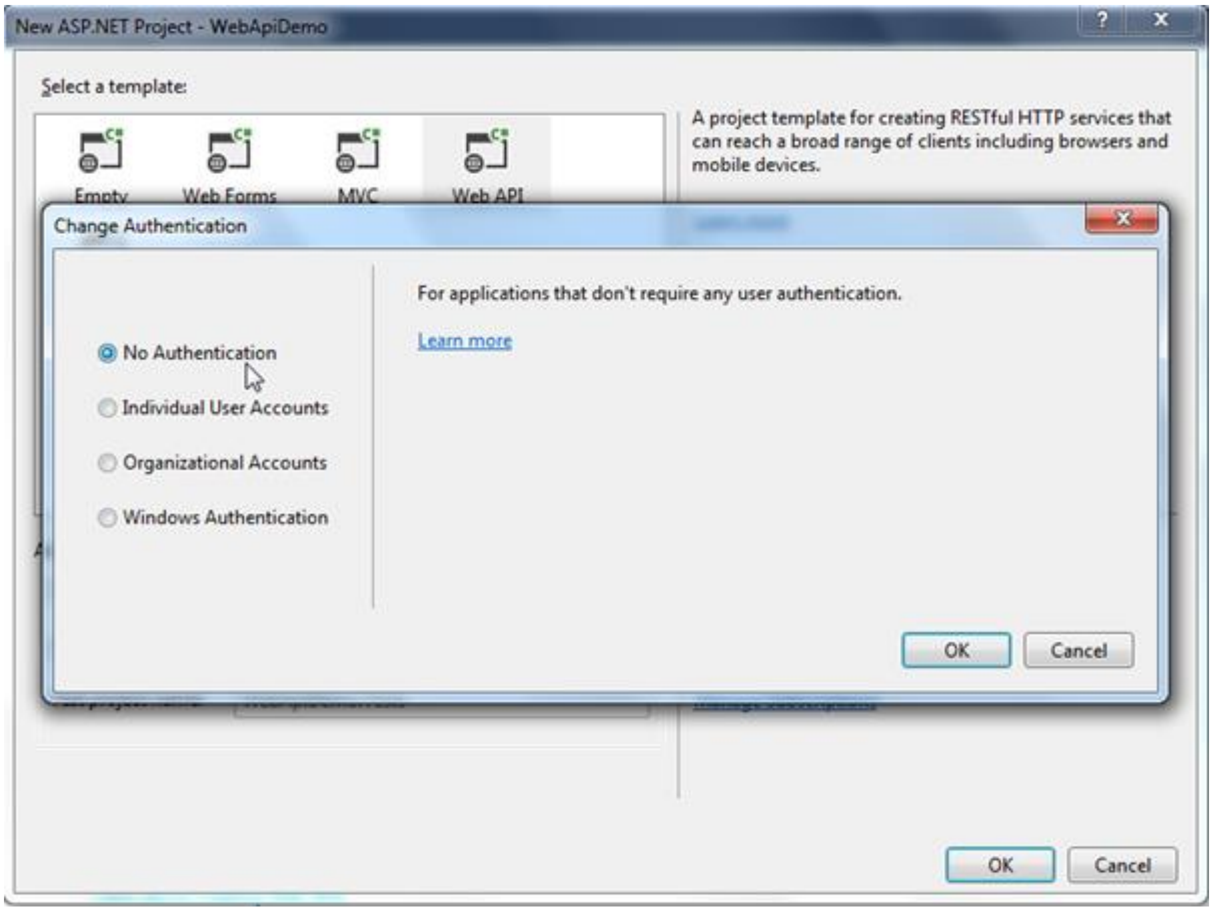
[ ] Web Forms [✓ MVC] [✓ Web API]

[ ] Add unit tests

Test project name: WebApiDemo.Tests

Select Web API Project Template

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



New ASP.NET Project - WebApiDemo

Select a template:

A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Change Authentication

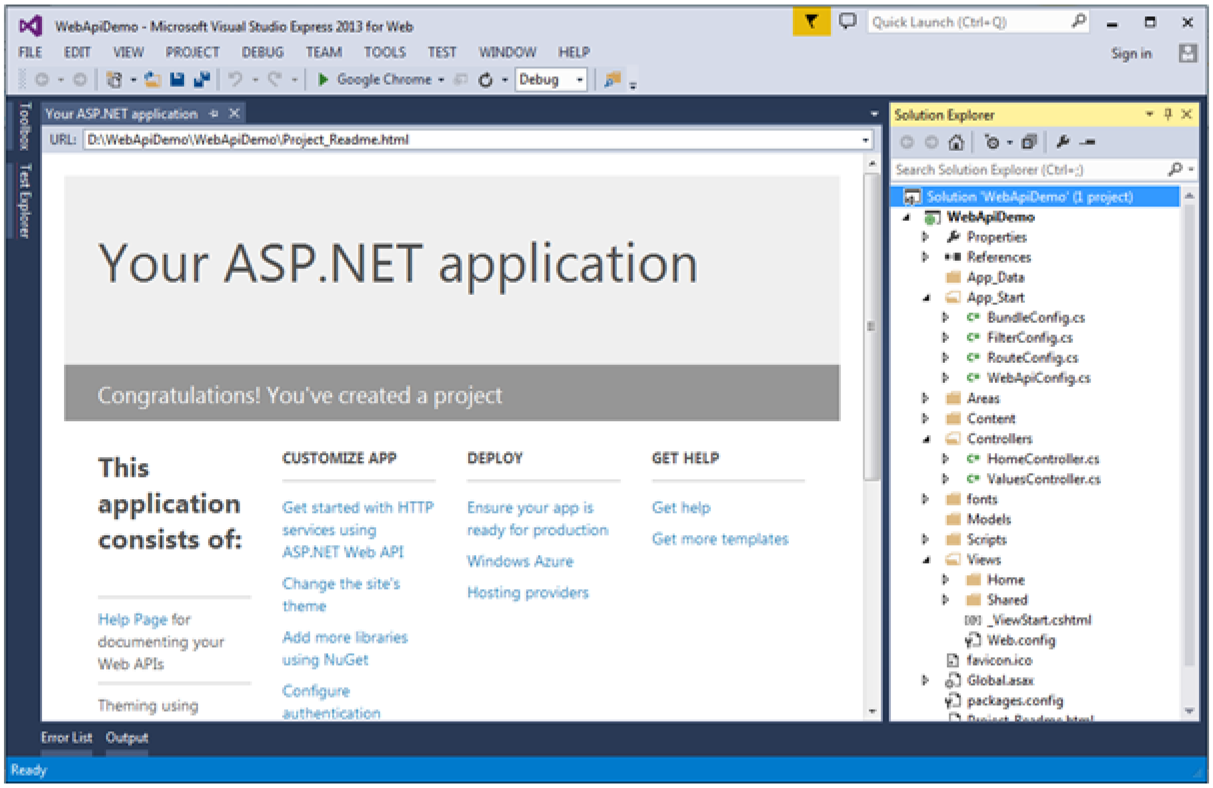
For applications that don't require any user authentication.

Learn more

[✓ No Authentication] Individual User Accounts Organizational Accounts Windows Authentication

Change Authentication

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



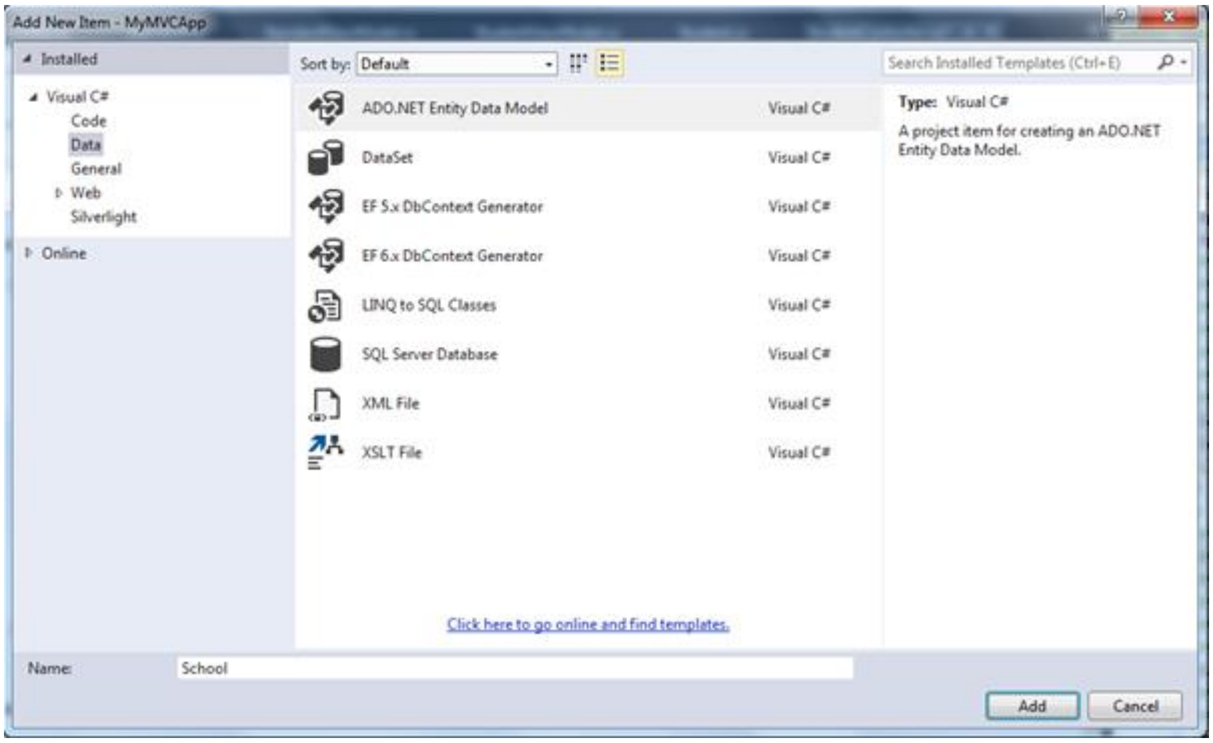
Web API Project

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.

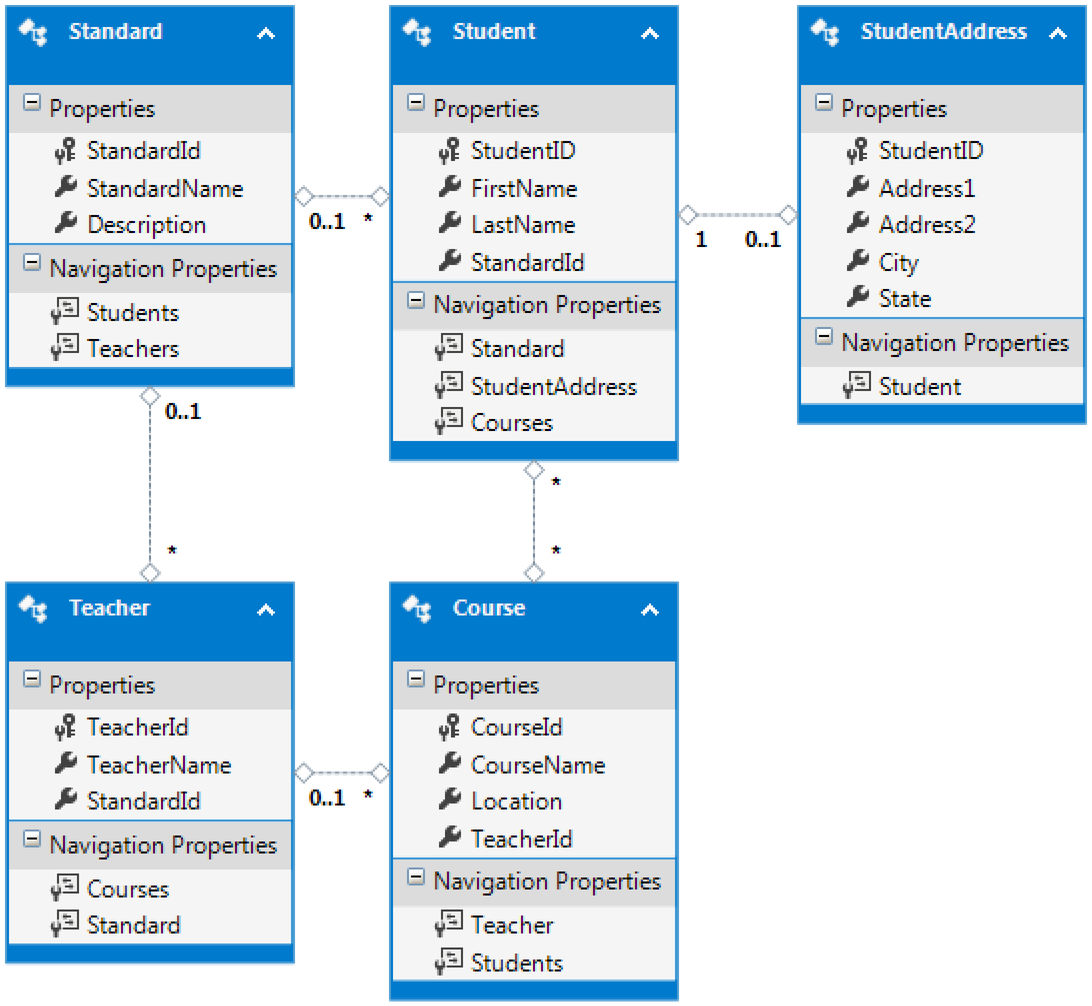


Name: School

Create Entity Data Model

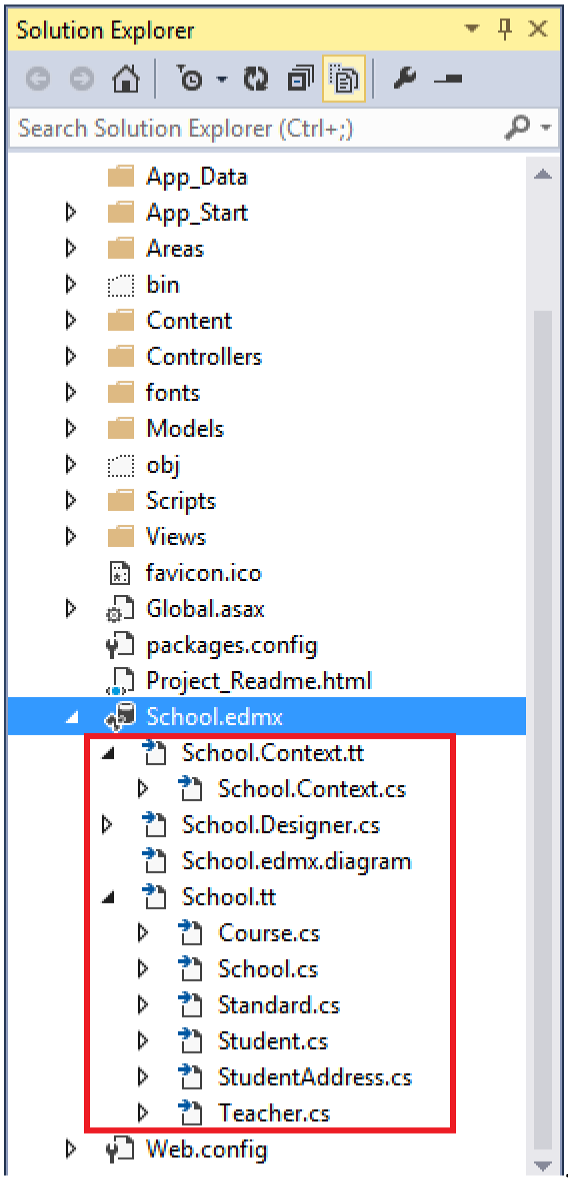
Select **Data** in the left pane and select **ADO.NET Entity Data Model** in the middle pane and enter the name of a data model and click **Add**. This will open Entity Data Model Wizard using which you can generate Entity Data Model for an existing School database. The scope of the topic is limited to Web API so we have not covered how to generate EDM. Learn about it here.

EntityFramework will generate 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.

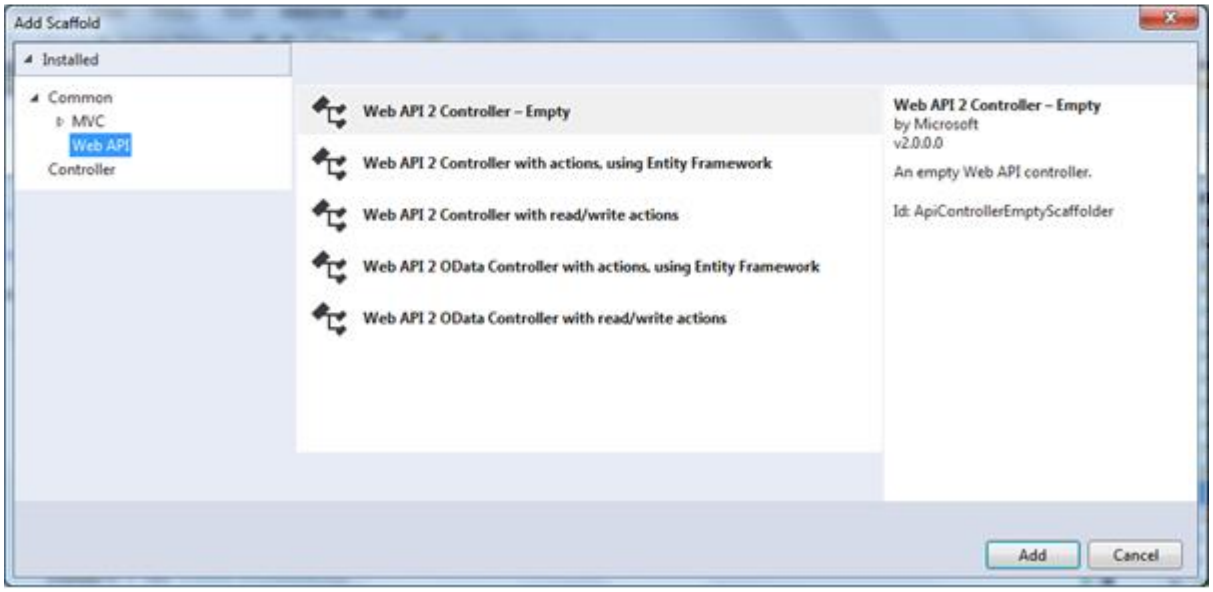


.edmx in the Project

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

Add Web API Controller

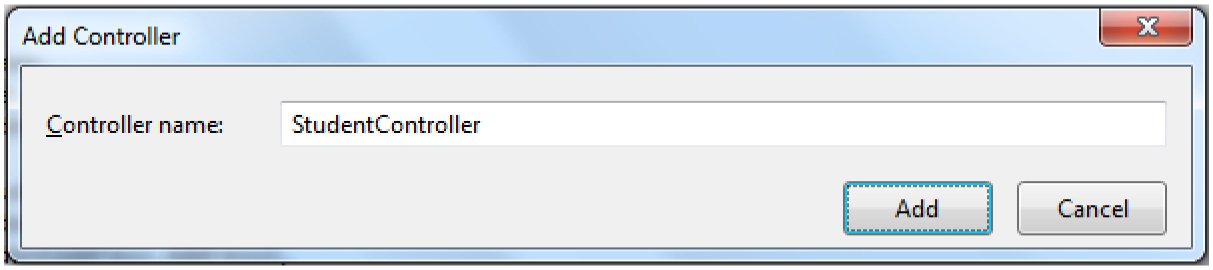
To add a Web API controller in your MVC project, right click on the **Controllers** folder or another folder where you want to add a Web API controller -> select **Add** -> select **Controller**. This will open **Add Scaffold** popup as shown below.


Add Scaffold
Installed
Common
MVC
Web API
Controller
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData Controller with actions, using Entity Framework
Web API 2 OData Controller with read/write actions
Web API 2 Controller – Empty
by Microsoft
v2.0.0.0
An empty Web API controller.
Id: ApiControllerEmptyScaffolder
Add
Cancel

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 **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.



Controller name: StudentController

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;
```

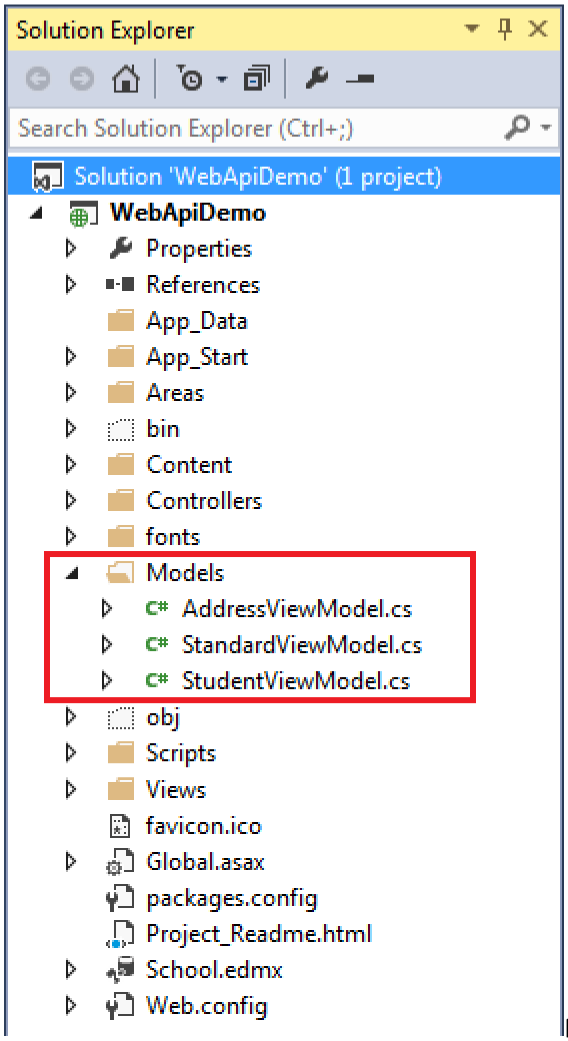
```
namespace MyMVCApp.Controllers
{
    public class StudentController : ApiController
    {

    }
}
```

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

Add Model

We will be accessing 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 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 followings 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")
```

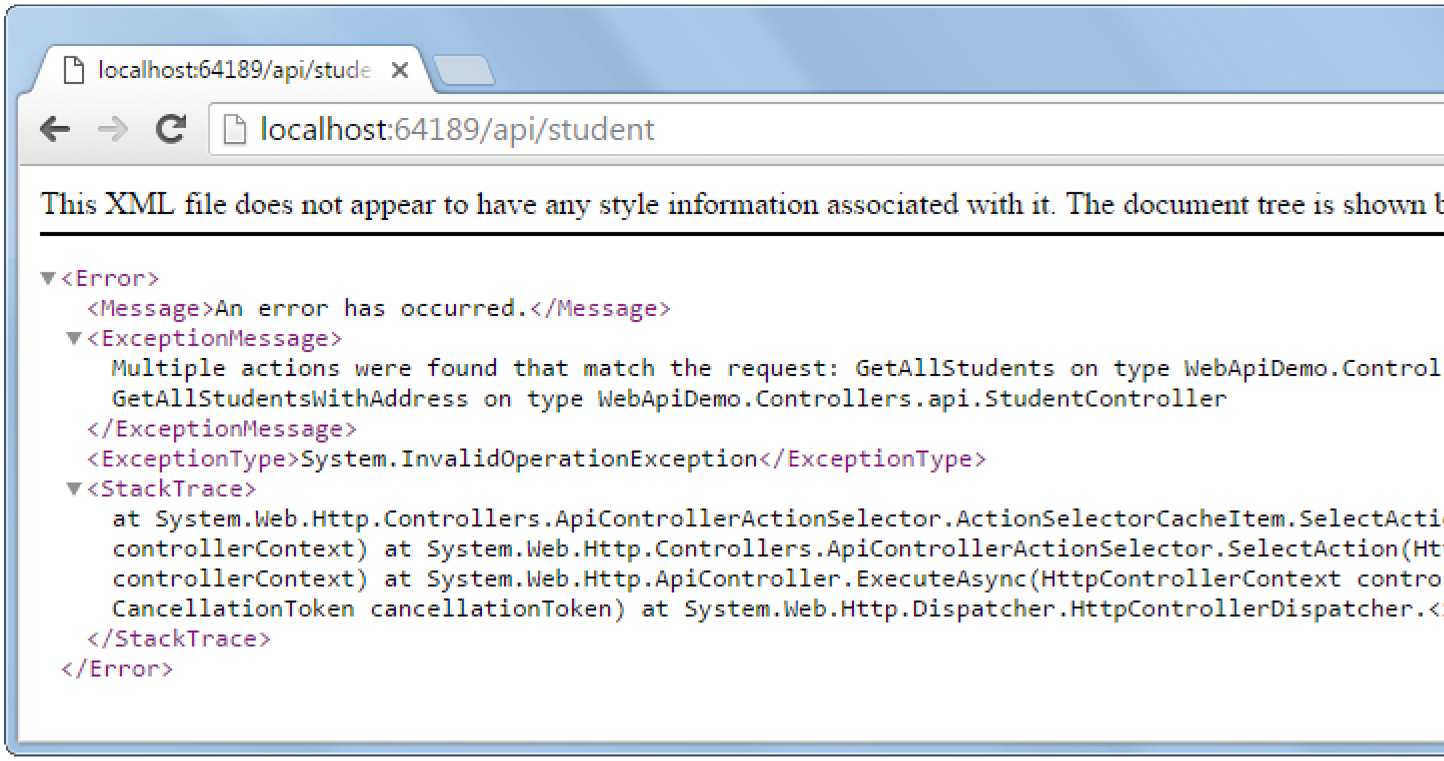
```
                        .Select(s => new StudentViewModel()
                        {
                            Id = s.StudentID,
                            FirstName = s.FirstName,
                            LastName = s.LastName
                        }).ToList<StudentViewModel>();
        }

        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.


localhost:64189/api/stude

localhost:64189/api/student

This XML file does not appear to have any style information associated with it. The document tree is shown b

```
▼<Error>
   <Message>An error has occurred.</Message>
 ▼<ExceptionMessage>
     Multiple actions were found that match the request: GetAllStudents on type WebApiDemo.Control
     GetAllStudentsWithAddress on type WebApiDemo.Controllers.api.StudentController
   </ExceptionMessage>
   <ExceptionType>System.InvalidOperationException</ExceptionType>
 ▼<StackTrace>
     at System.Web.Http.Controllers.ApiControllerActionSelector.ActionSelectorCacheItem.SelectActi
     controllerContext) at System.Web.Http.Controllers.ApiControllerActionSelector.SelectAction(Ht
     controllerContext) at System.Web.Http.ApiController.ExecuteAsync(HttpControllerContext contro
     CancellationToken cancellationToken) at System.Web.Http.Dispatcher.HttpControllerDispatcher.<
   </StackTrace>
 </Error>
```

Web API Error

This is because you cannot have multiple action methods with same number of parameters with 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`.

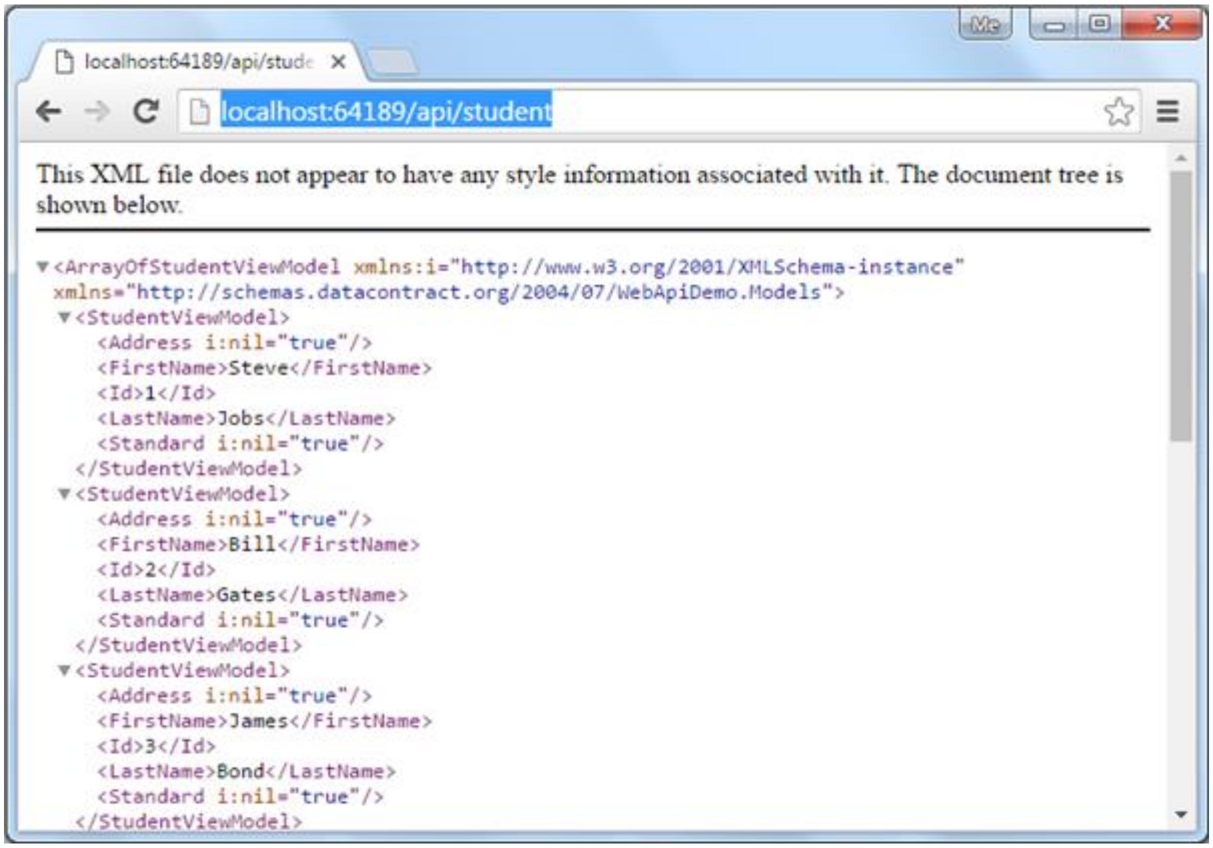
The following example illustrates how to handle this kind of scenario.

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.



localhost:64189/api/stude

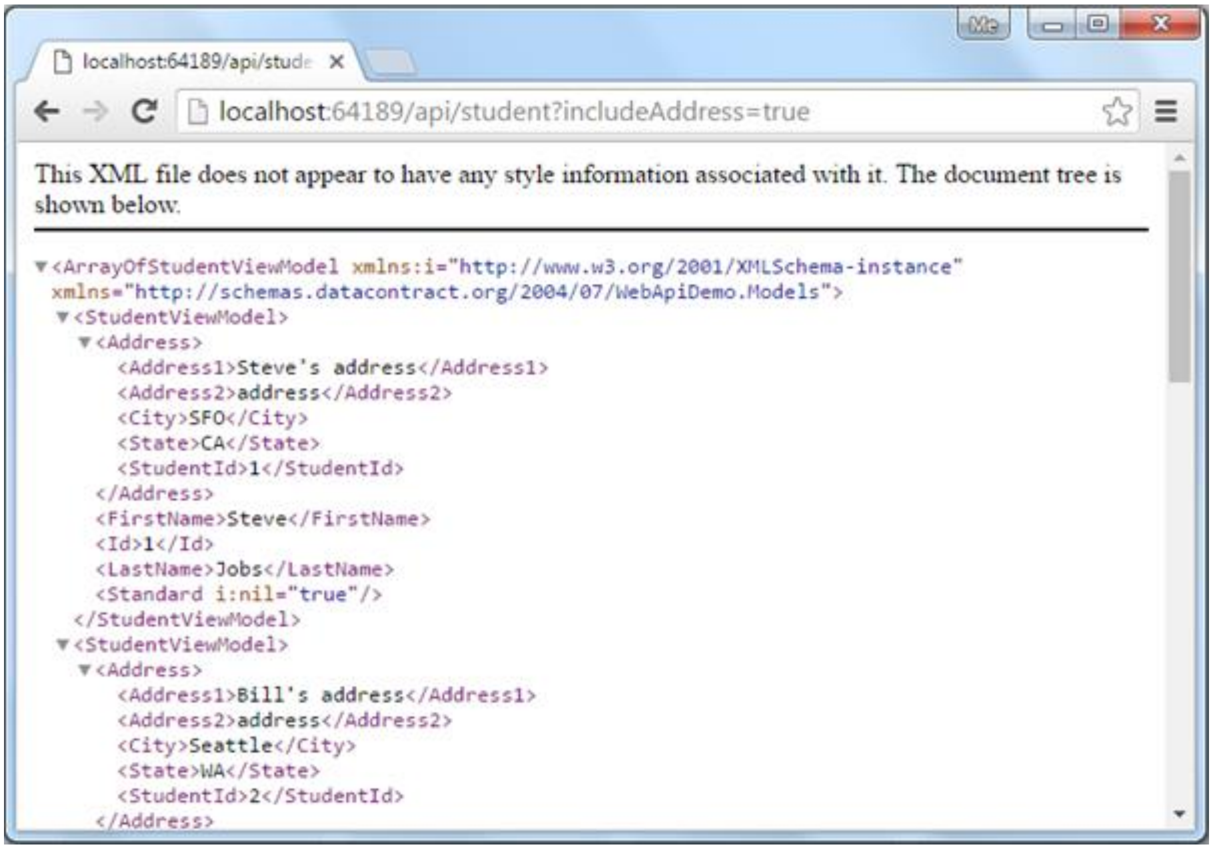
localhost:64189/api/student

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<ArrayOfStudentViewModel xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://schemas.datacontract.org/2004/07/WebApiDemo.Models">
  ▼<StudentViewModel>
     <Address i:nil="true"/>
     <FirstName>Steve</FirstName>
     <Id>1</Id>
     <LastName>Jobs</LastName>
     <Standard i:nil="true"/>
   </StudentViewModel>
  ▼<StudentViewModel>
     <Address i:nil="true"/>
     <FirstName>Bill</FirstName>
     <Id>2</Id>
     <LastName>Gates</LastName>
     <Standard i:nil="true"/>
   </StudentViewModel>
  ▼<StudentViewModel>
     <Address i:nil="true"/>
     <FirstName>James</FirstName>
     <Id>3</Id>
     <LastName>Bond</LastName>
     <Standard i:nil="true"/>
   </StudentViewModel>
```

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.


localhost:64189/api/stude
localhost:64189/api/student?includeAddress=true

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<ArrayOfStudentViewModel xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://schemas.datacontract.org/2004/07/WebApiDemo.Models">
  ▼<StudentViewModel>
    ▼<Address>
        <Address1>Steve's address</Address1>
        <Address2>address</Address2>
        <City>SFO</City>
        <State>CA</State>
        <StudentId>1</StudentId>
     </Address>
     <FirstName>Steve</FirstName>
     <Id>1</Id>
     <LastName>Jobs</LastName>
     <Standard i:nil="true"/>
   </StudentViewModel>
  ▼<StudentViewModel>
    ▼<Address>
        <Address1>Bill's address</Address1>
        <Address2>address</Address2>
        <City>Seattle</City>
        <State>WA</State>
        <StudentId>2</StudentId>
     </Address>
```

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 following action methods in StudentController to demonstrate how Web API handles multiple HTTP GET requests.

Action method	
GetStudentById(int id)	Returns student whose id matches with the specif
GetAllStudents(string name)	Returns list of students whose name matches with
GetAllStudentsInSameStandard(int standardId)	Returns list of students who are in the specified st

The following example implements the above action methods.

Example: Multiple Get Methods in Web API Controller

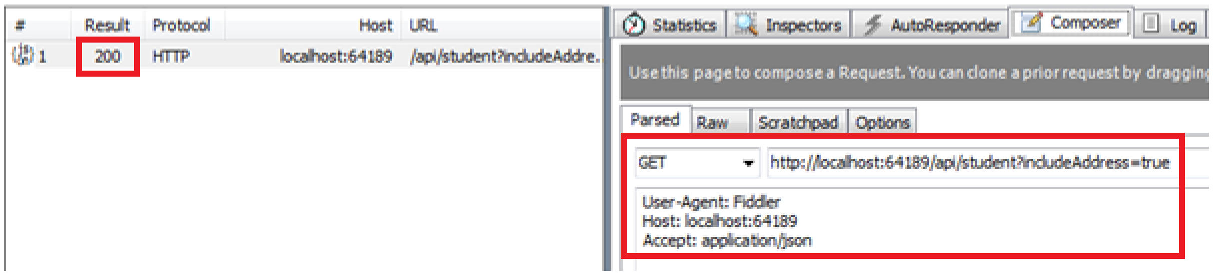
```
public class StudentController : ApiController
{
    public StudentController()
    {
    }

    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 following HTTP GET requests.

HTTP GET Request URL	
http://localhost:64189/api/student	Returns all students wi
http://localhost:64189/api/student?includeAddress=false	Returns all students wi
http://localhost:64189/api/student?includeAddress=true	Returns all students wi
http://localhost:64189/api/student?id=123	Returns student with t
http://localhost:64189/api/student?name=steve	Returns all students wi
http://localhost:64189/api/student?standardId=5	Returns all students wi

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.

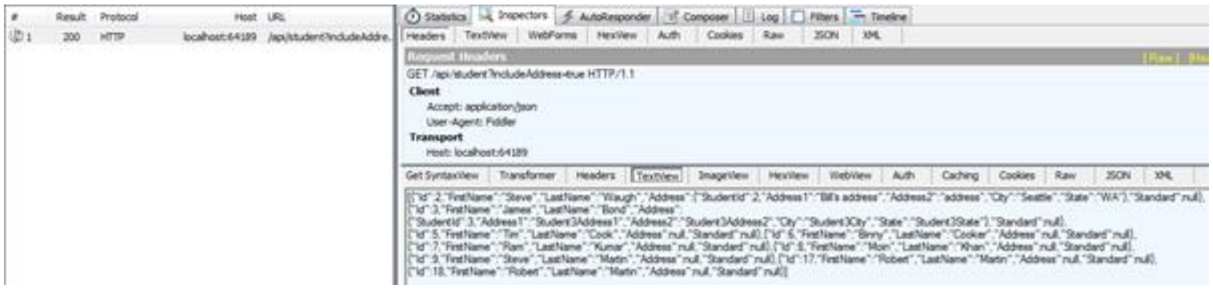


Use this page to compose a Request. You can clone a prior request by dragging

```
GET    http://localhost:64189/api/student?includeAddress=true

User-Agent: Fiddler
Host: localhost:64189
Accept: application/json
```

Http GET request in Fiddler

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



```
# Result Protocol Host URL
1 200 HTTP localhost:64189 /api/student?includeAddre...
```

```
GET /api/student?includeAddress=true HTTP/1.1
Client
    Accept: application/json
    User-Agent: Fiddler
Transport
    Host: localhost:64189
```

```
[{"Id":2,"FirstName":"Steve","LastName":"Waugh","Address":{"StudentId":2,"Address1":"Bill's address","Address2":"address","City":"Seattle","State":"WA"},"Standard":null},
{"Id":3,"FirstName":"James","LastName":"Bond","Address":
{"StudentId":3,"Address1":"Student3Address1","Address2":"Student3Address2","City":"Student3City","State":"Student3State"},"Standard":null},
{"Id":5,"FirstName":"Tim","LastName":"Cook","Address":null,"Standard":null},{"Id":6,"FirstName":"Binny","LastName":"Cooker","Address":null,"Standard":null},
{"Id":7,"FirstName":"Ram","LastName":"Kumar","Address":null,"Standard":null},{"Id":8,"FirstName":"Moin","LastName":"Khan","Address":null,"Standard":null},
{"Id":9,"FirstName":"Steve","LastName":"Martin","Address":null,"Standard":null},{"Id":17,"FirstName":"Robert","LastName":"Martin","Address":null,"Standard":null},
{"Id":18,"FirstName":"Robert","LastName":"Martin","Address":null,"Standard":null}]
```

Http GET response in Fiddler

Next, implement 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 also implemented GET methods. Here, we will implement 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 record in the database using Entity Framework.

The action method that will handle HTTP POST request must start with a 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 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();
    }
}
```

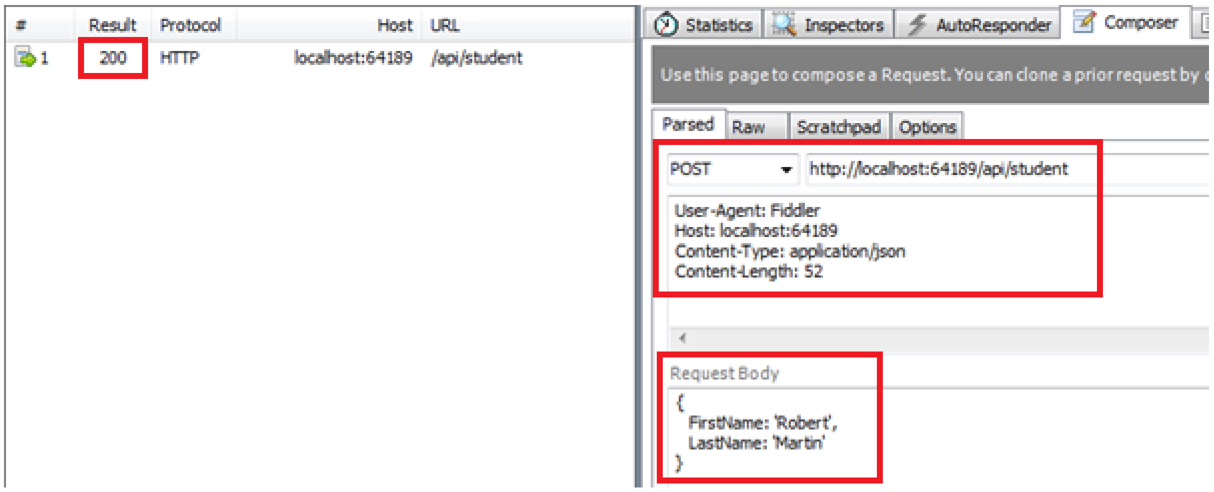
As you can see above, we named action method as PostNewStudent. You can give any name as per your requirement but it must start with the word "Post". The PostNewStudent() action method includes parameter of StudentViewModel type which includes all the information about new student.

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



```
1  200  HTTP  localhost:64189  /api/student
```

Use this page to compose a Request. You can clone a prior request by

```
POST  http://localhost:64189/api/student

User-Agent: Fiddler
Host: localhost:64189
Content-Type: application/json
Content-Length: 52

Request Body
{
  FirstName: 'Robert',
  LastName: 'Martin'
}
```

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 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 also implemented GET & POST methods. Here, we will implement 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 a 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 Put action method to handle HTTP PUT request.

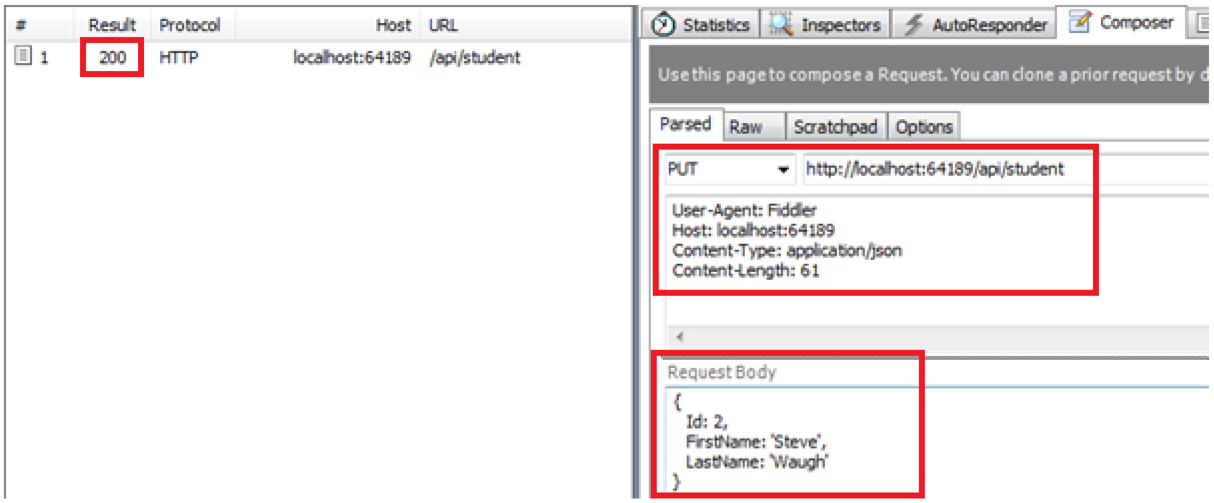
Example: Put Method in Web API Controller

```
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, Put action method includes a parameter of StudentViewModel. It then creates 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.



```
http://localhost:64189/api/student

User-Agent: Fiddler
Host: localhost:64189
Content-Type: application/json
Content-Length: 61
```

```
{
  Id: 2,
  FirstName: 'Steve',
  LastName: 'Waugh'
}
```

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 Delete action method to handle HTTP DELETE request in the Web A

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 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 Delete action method to handle HTTP DELETE request.

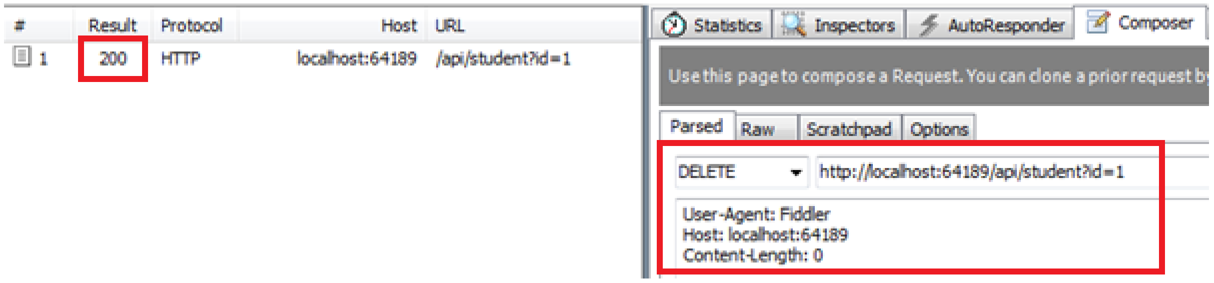
Example: Delete Method in Web API Controller

```
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, 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.



#	Result	Protocol	Host	URL
1	200	HTTP	localhost:64189	/api/student?id=1

Statistics
Inspectors
AutoResponder
Composer
Use this page to compose a Request. You can clone a prior request b
Parsed
Raw
Scratchpad
Options

DELETE http://localhost:64189/api/student?id=1

```
User-Agent: Fiddler
Host: localhost:64189
Content-Length: 0
```

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 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.

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.

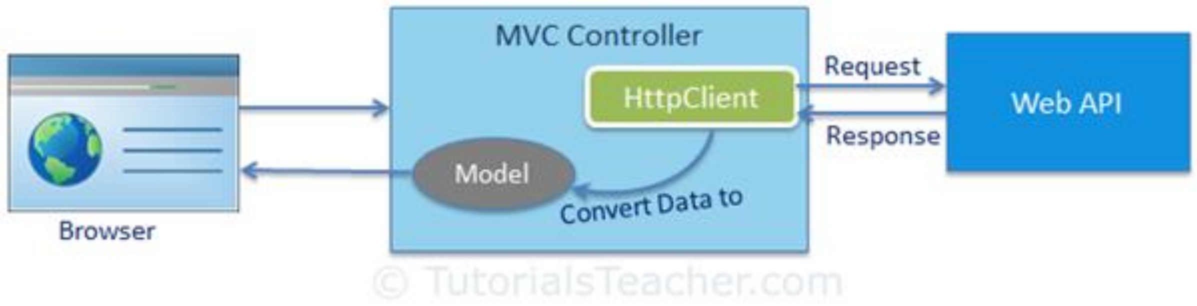
Here, we will consume our Web API (created in the previous section) in the following environments:

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.

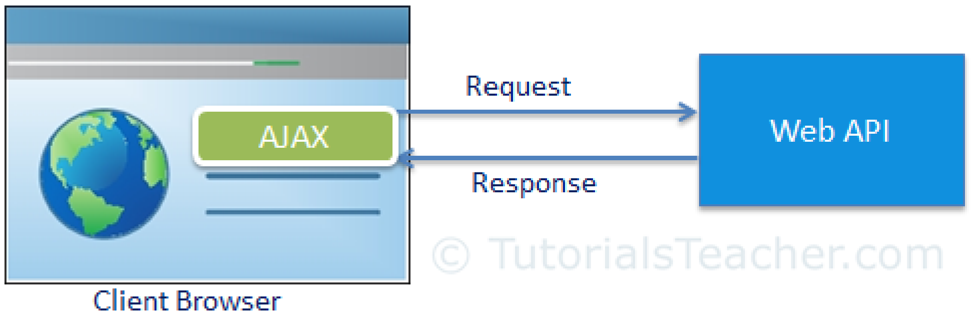


Consume Web API at Server side ASP.NET MVC
Note: AngularJS or any other JavaScript framework can be used in MVC view and can access Web API directly from the view using AJAX. We have taken ASP.NET MVC just to demonstrate how to access Web API from server side code in case you do not use any JavaScript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

- Share
- Tweet
- Share
- Whatsapp

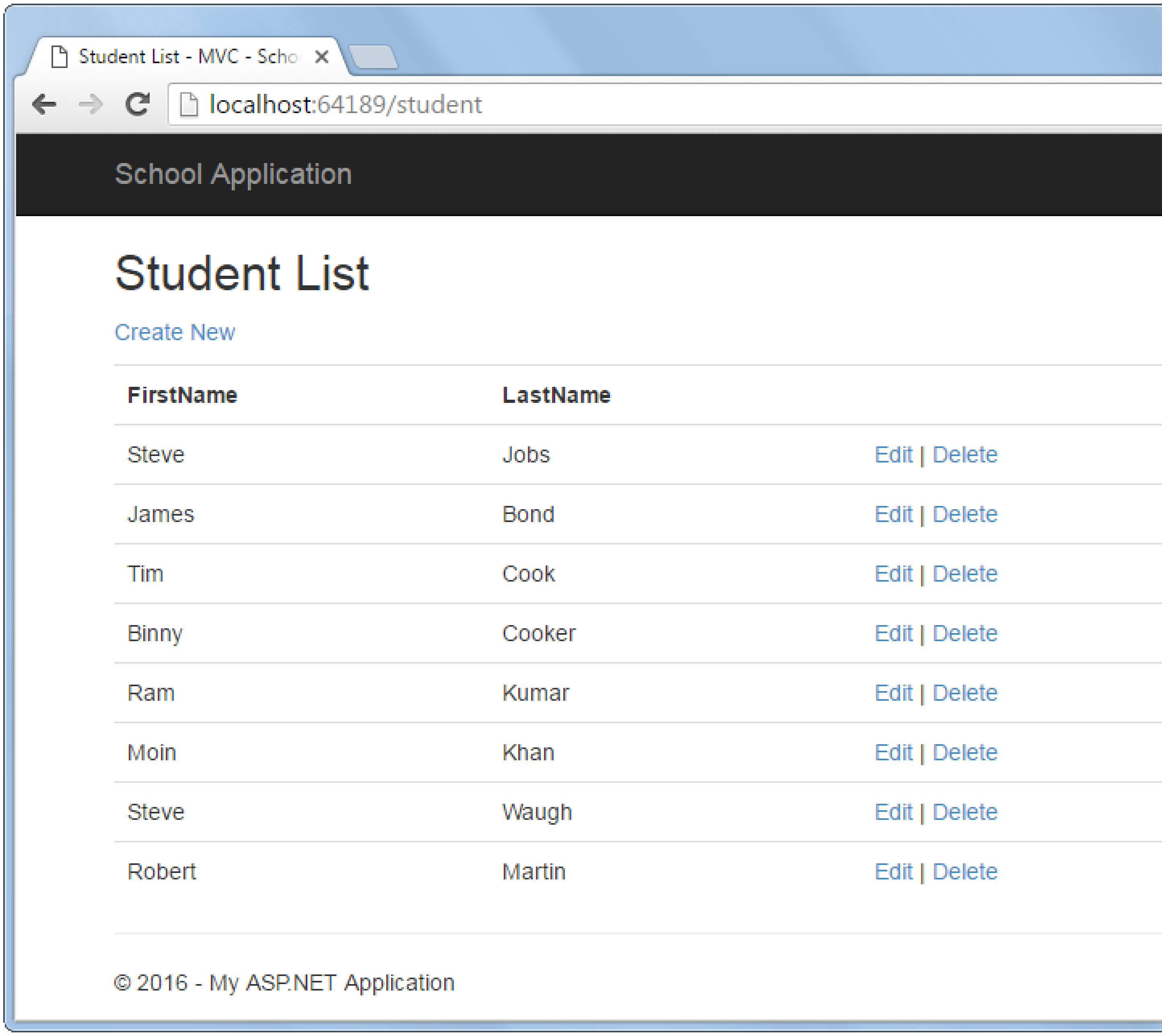
Consume Web API Get method in ASP.NET MVC

We created Web API and implemented various Get methods to handle different HTTP GET requests in the Implement Get Methodsection. 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 in 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 - MVC - Scho

localhost:64189/student

School Application

Student List

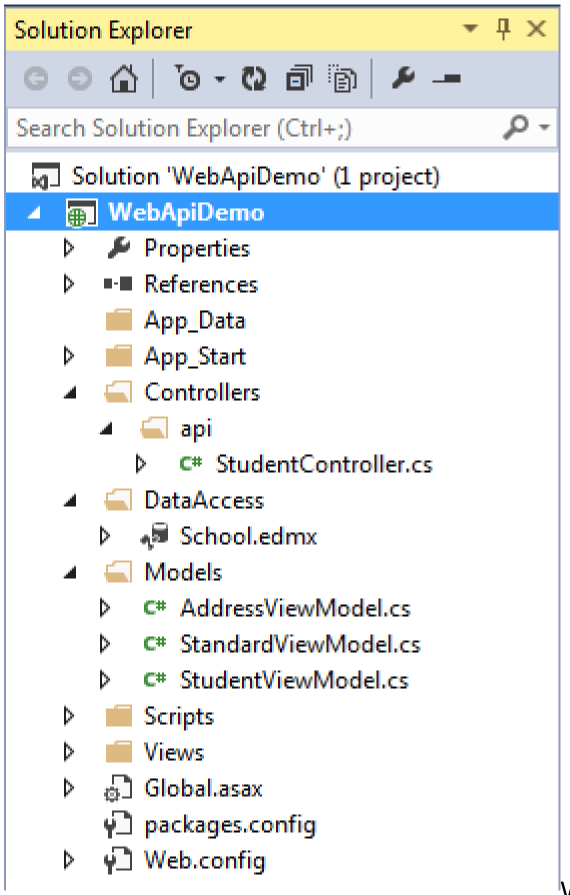
Create New

FirstName	LastName	
Steve	Jobs	Edit | Delete
James	Bond	Edit | Delete
Tim	Cook	Edit | Delete
Binny	Cooker	Edit | Delete
Ram	Kumar	Edit | Delete
Moin	Khan	Edit | Delete
Steve	Waugh	Edit | Delete
Robert	Martin	Edit | Delete

© 2016 - My ASP.NET Application

Student List View

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

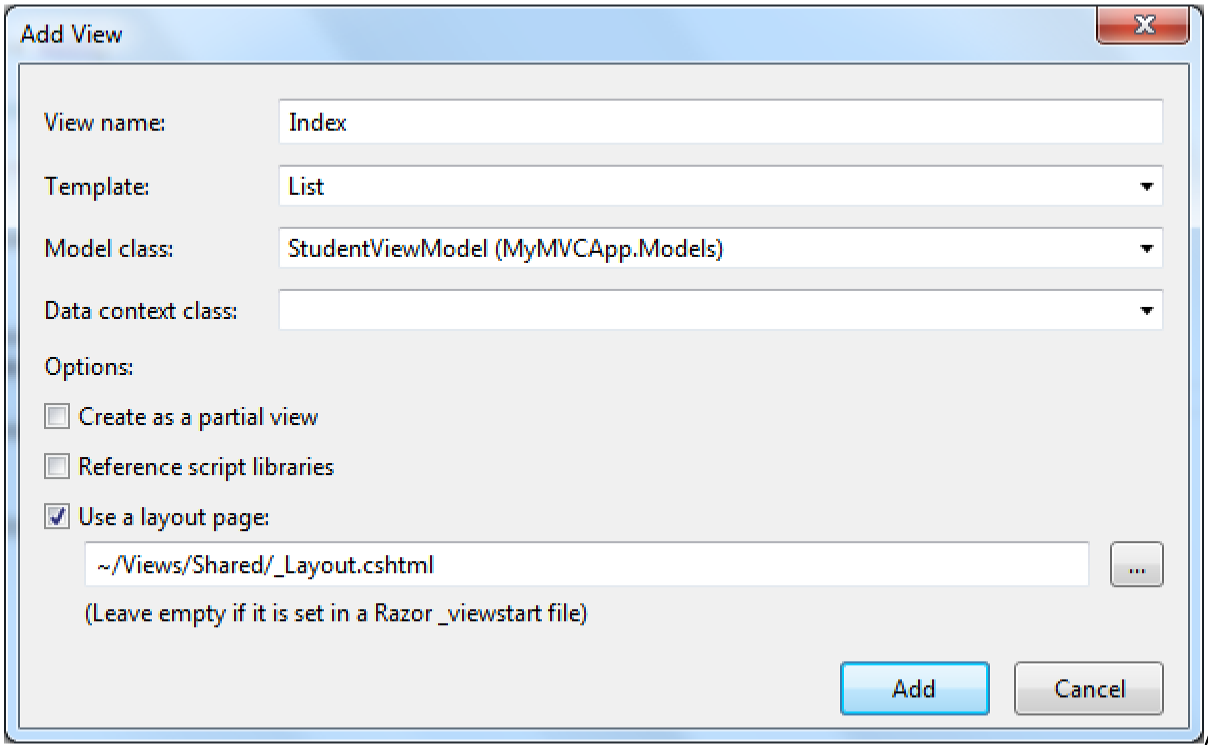
We need to access Web API in the Index() action method using HttpClient as shown below. Learn about HttpClient in detail here.

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 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

View name: Index

Template: List

Model class: StudentViewModel (MyMVCApp.Models)

Data context class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

Add View in ASP.NET MVC

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

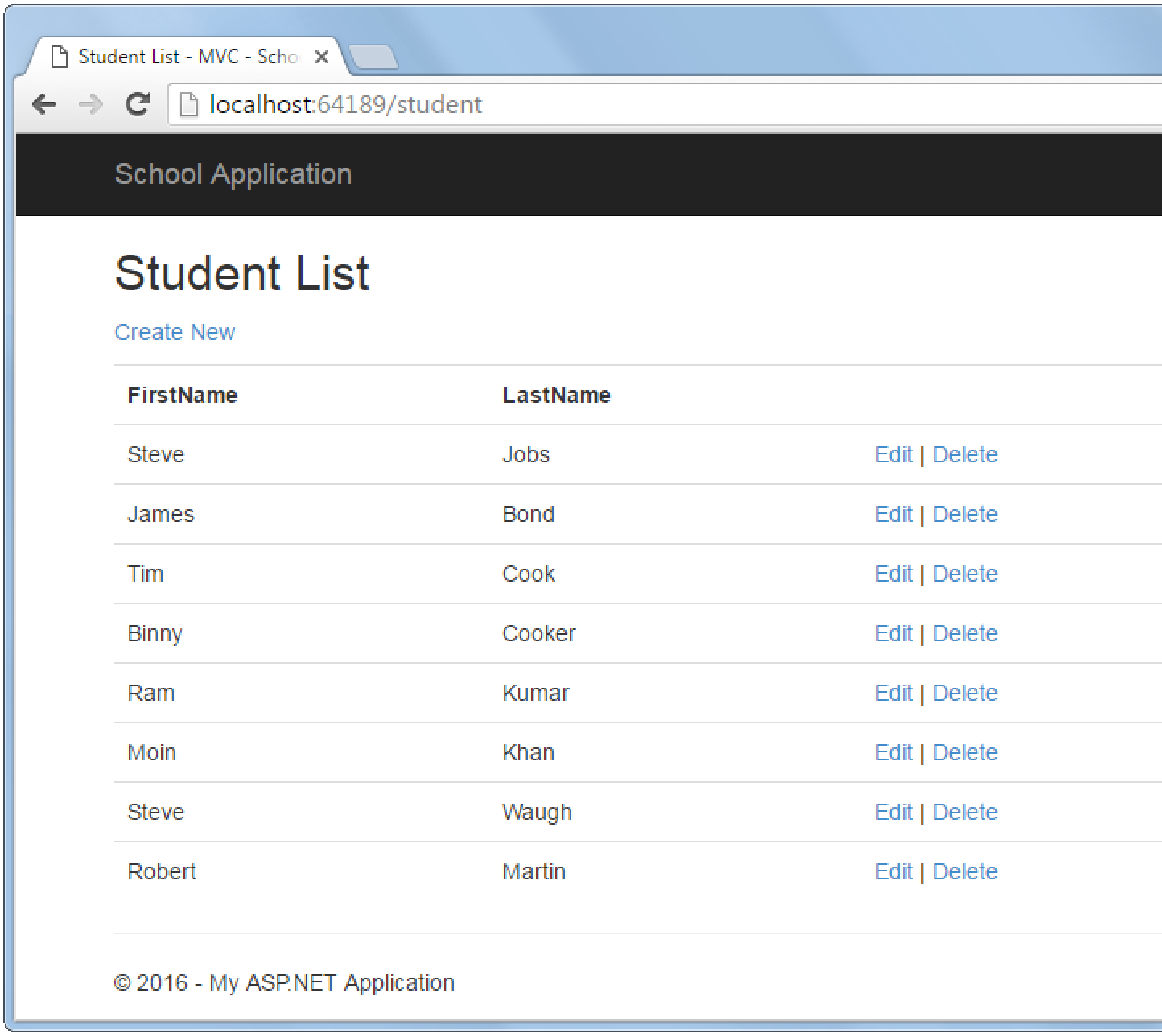
Index.cshtml

```
@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 - MVC - Scho

localhost:64189/student

School Application

Student List

Create New

FirstName	LastName	
Steve	Jobs	Edit | Delete
James	Bond	Edit | Delete
Tim	Cook	Edit | Delete
Binny	Cooker	Edit | Delete
Ram	Kumar	Edit | Delete
Moin	Khan	Edit | Delete
Steve	Waugh	Edit | Delete
Robert	Martin	Edit | Delete

© 2016 - My ASP.NET Application

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

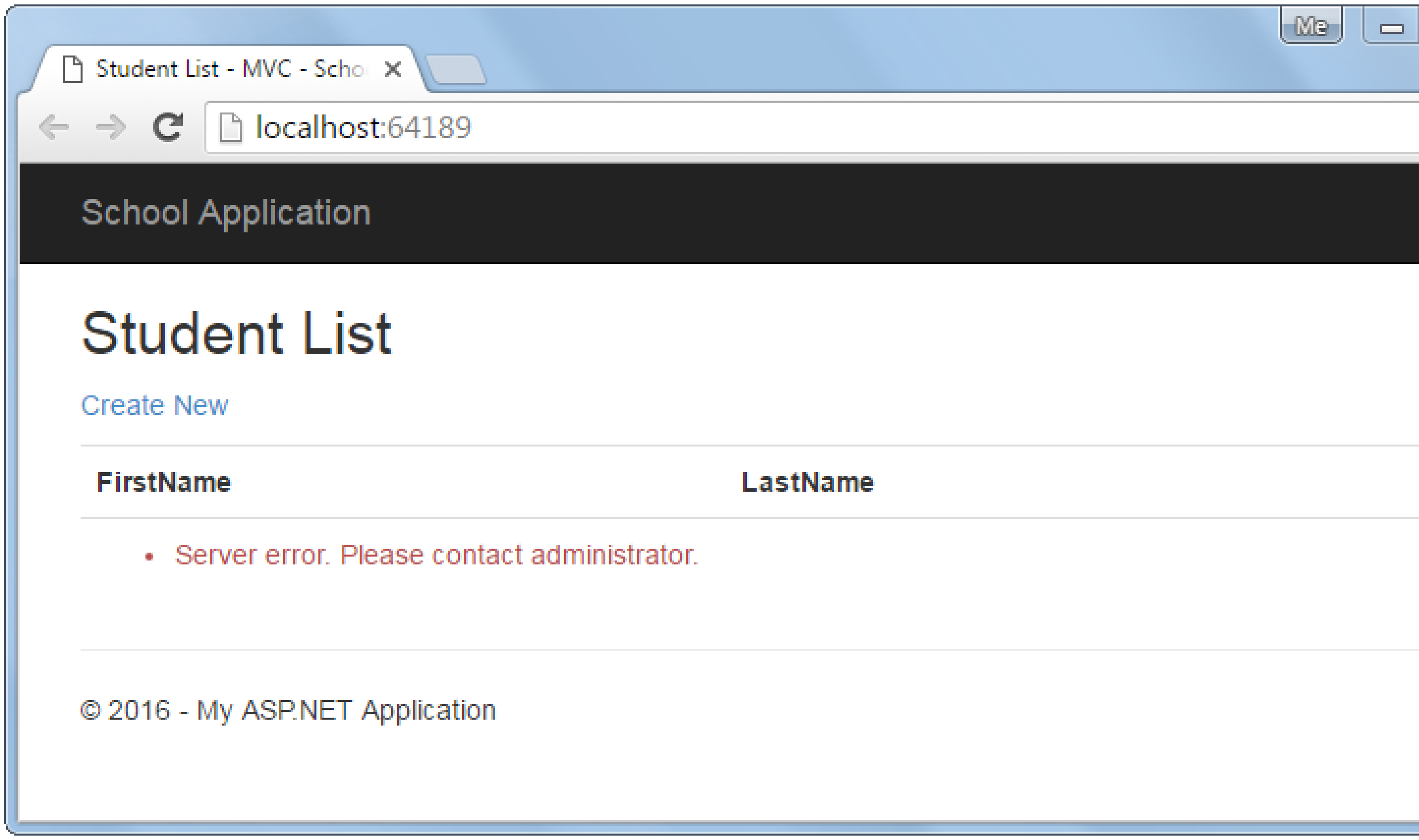
```
@model IEnumerable<WebAPIDemo.Models.StudentViewModel>
```

```
@{
    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 error message if Web API returns error response with the status other than 200 OK.

Please notice that we have added model error in the Index() action method in StudentController class created in the 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.



Student List

Create New

FirstName	LastName

- Server error. Please contact administrator.

© 2016 - My ASP.NET Application

Display Error Message

In the next section, we will consume 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 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.

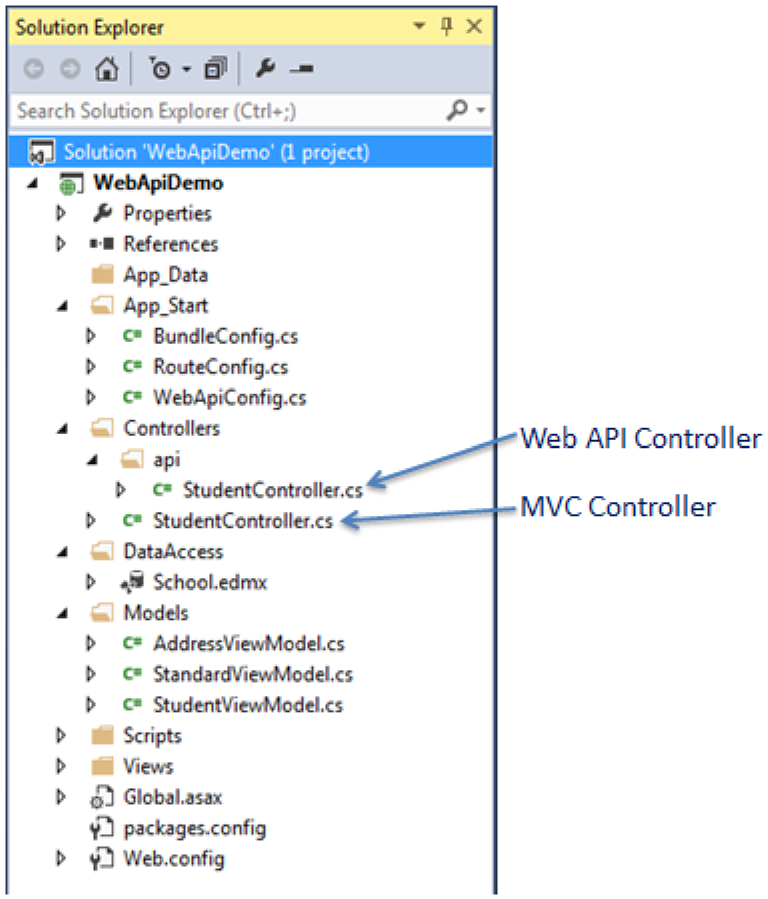
We already created Web API with Post method in the Implement Post Method section shown below.

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 new record in the database using Entity Framework and will return 200 OK response status.

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

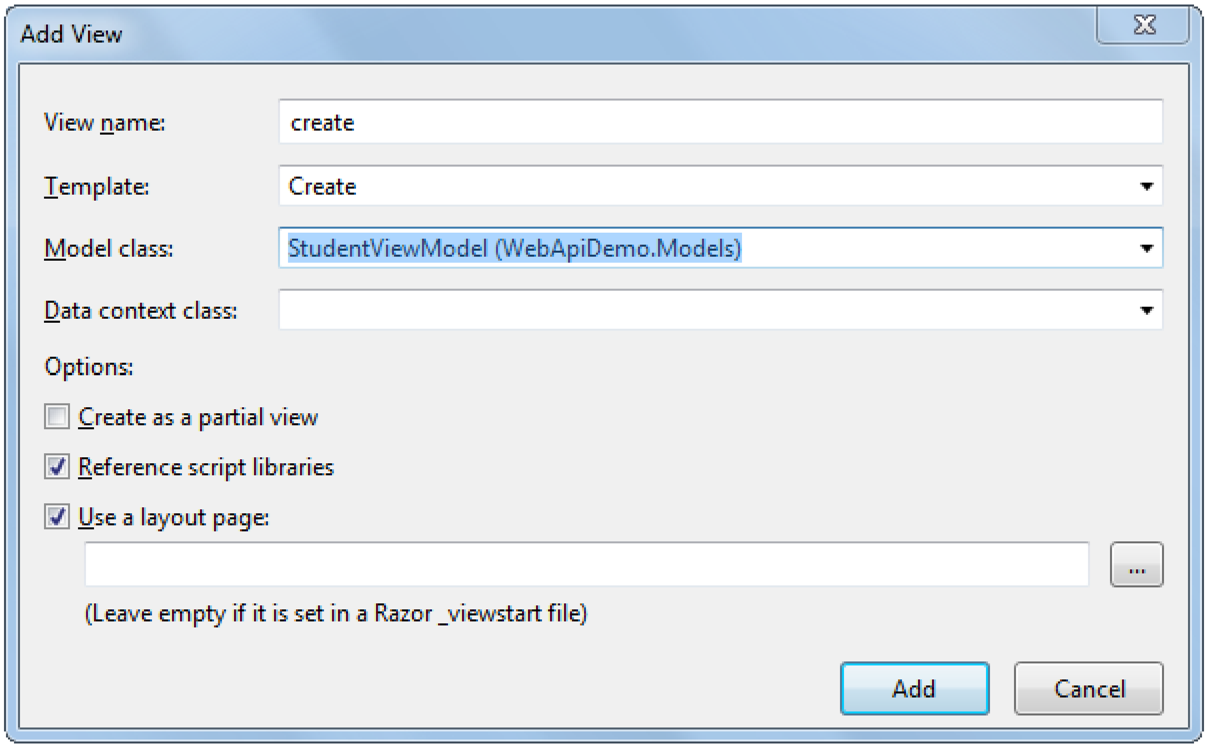
Step 1:

First, we need to add 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.



View name: create

Template: Create

Model class: StudentViewModel (WebApiDemo.Models)

Data context class:

Options:

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

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.

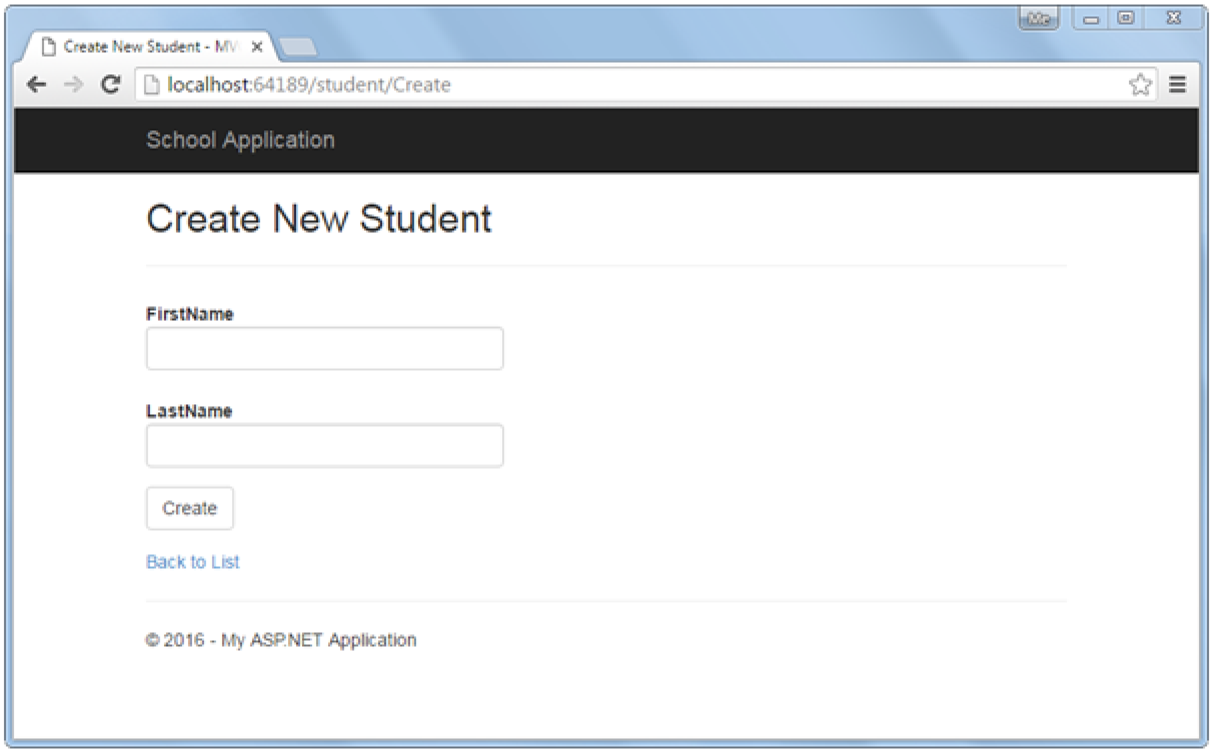
create.cshtml

```
@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.



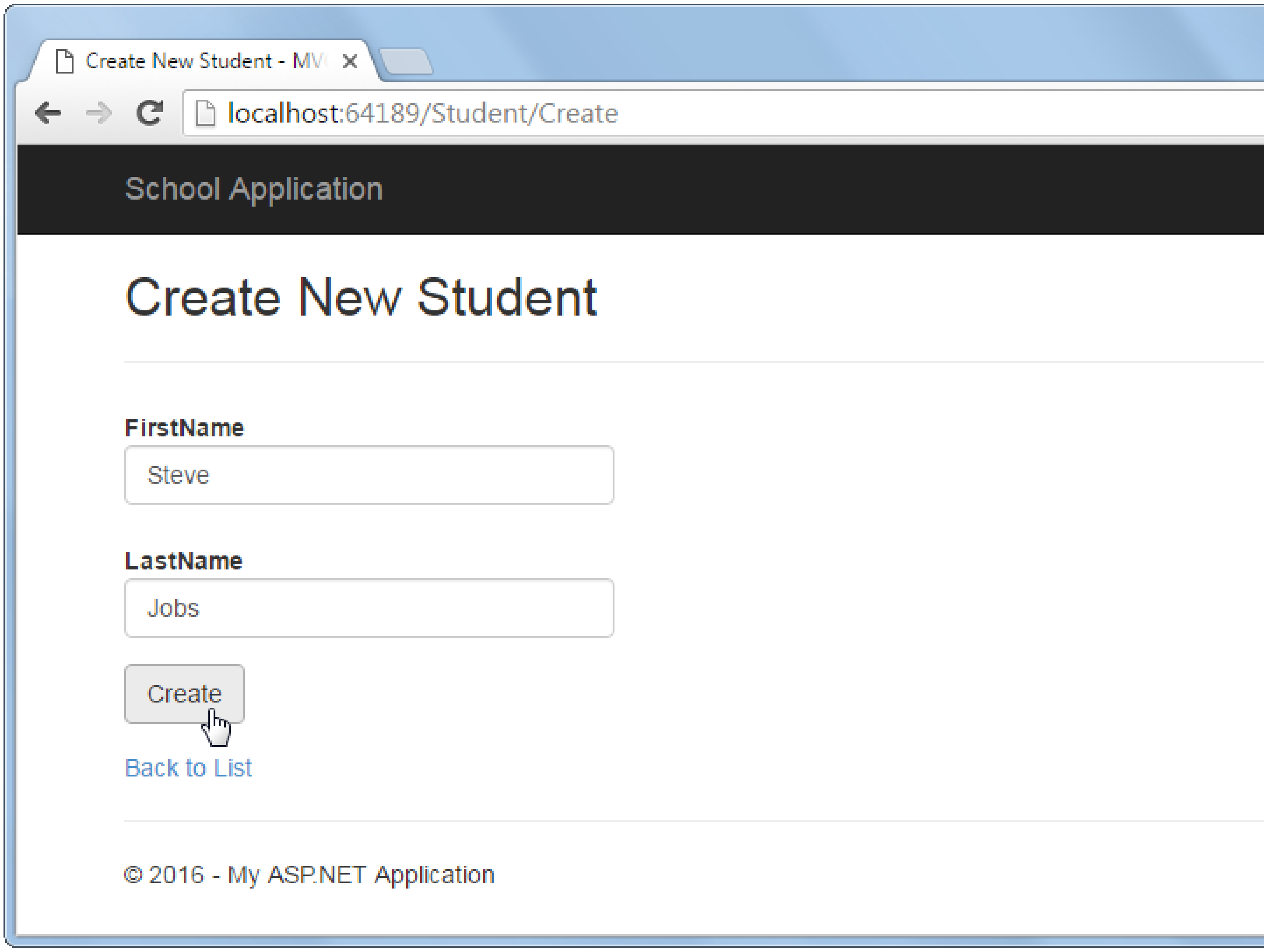
Create New Student View

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(), it uses `HttpClient` to send HTTP POST request to Web API with StudentViewModel object. If 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 New Student

FirstName

Steve

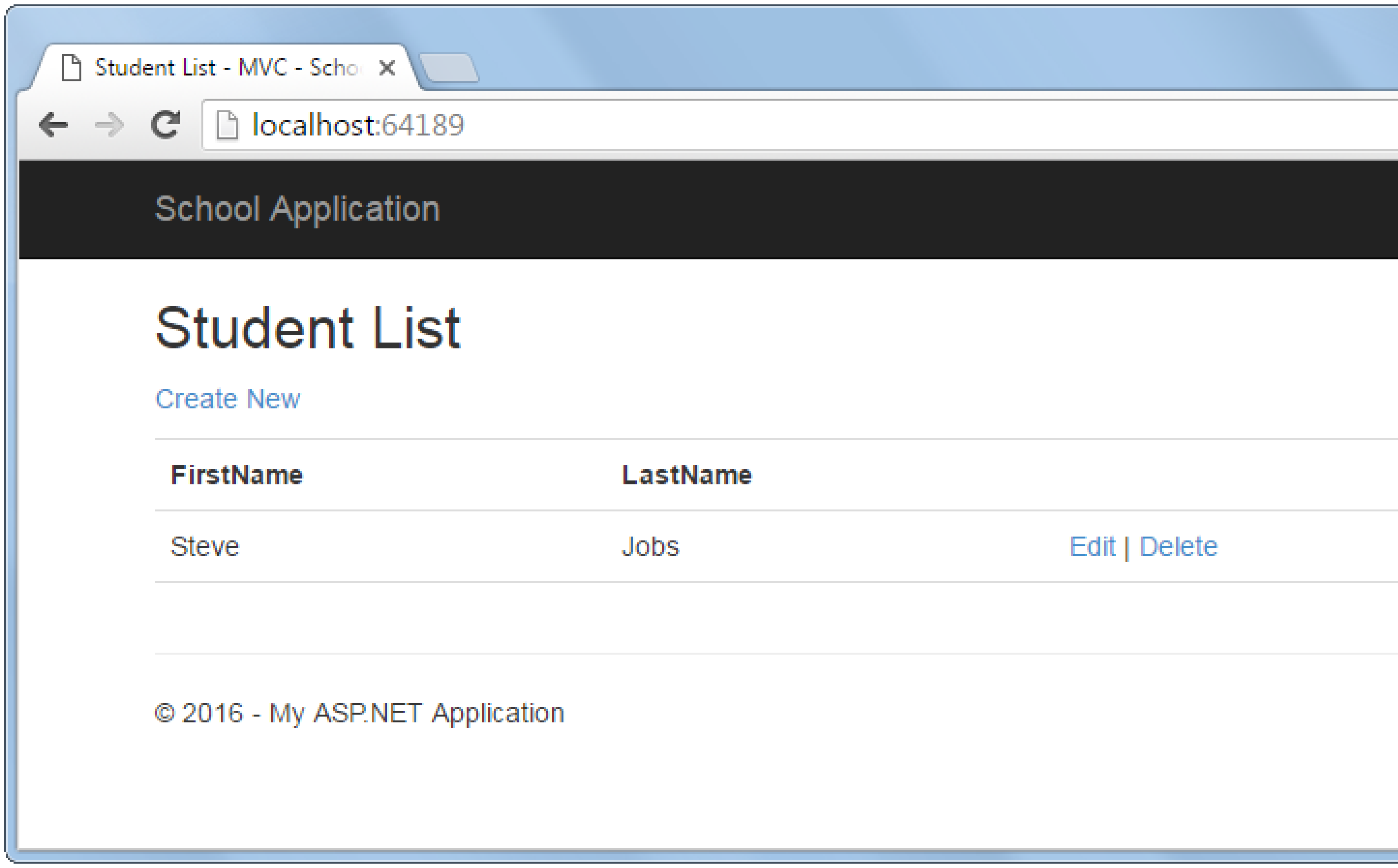
LastName

Jobs

© 2016 - My ASP.NET Application

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.


Student List - MVC - Scho

localhost:64189

School Application

Student List

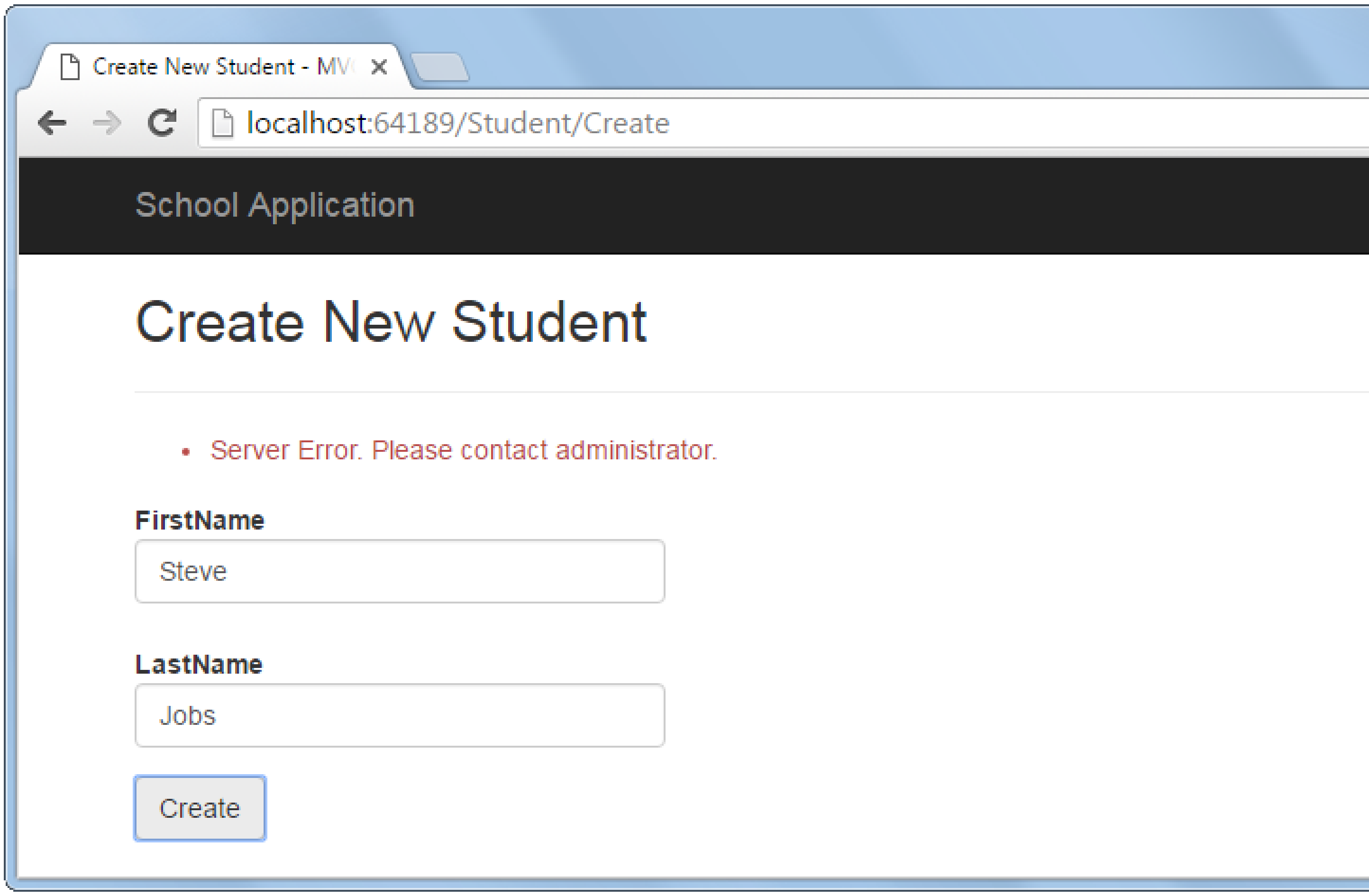
Create New

FirstName	LastName	
Steve	Jobs	Edit | Delete

© 2016 - My ASP.NET Application

Redirect to Student List View

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



Create New Student

- Server Error. Please contact administrator.

FirstName: Steve

LastName: Jobs

Display Error Message

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

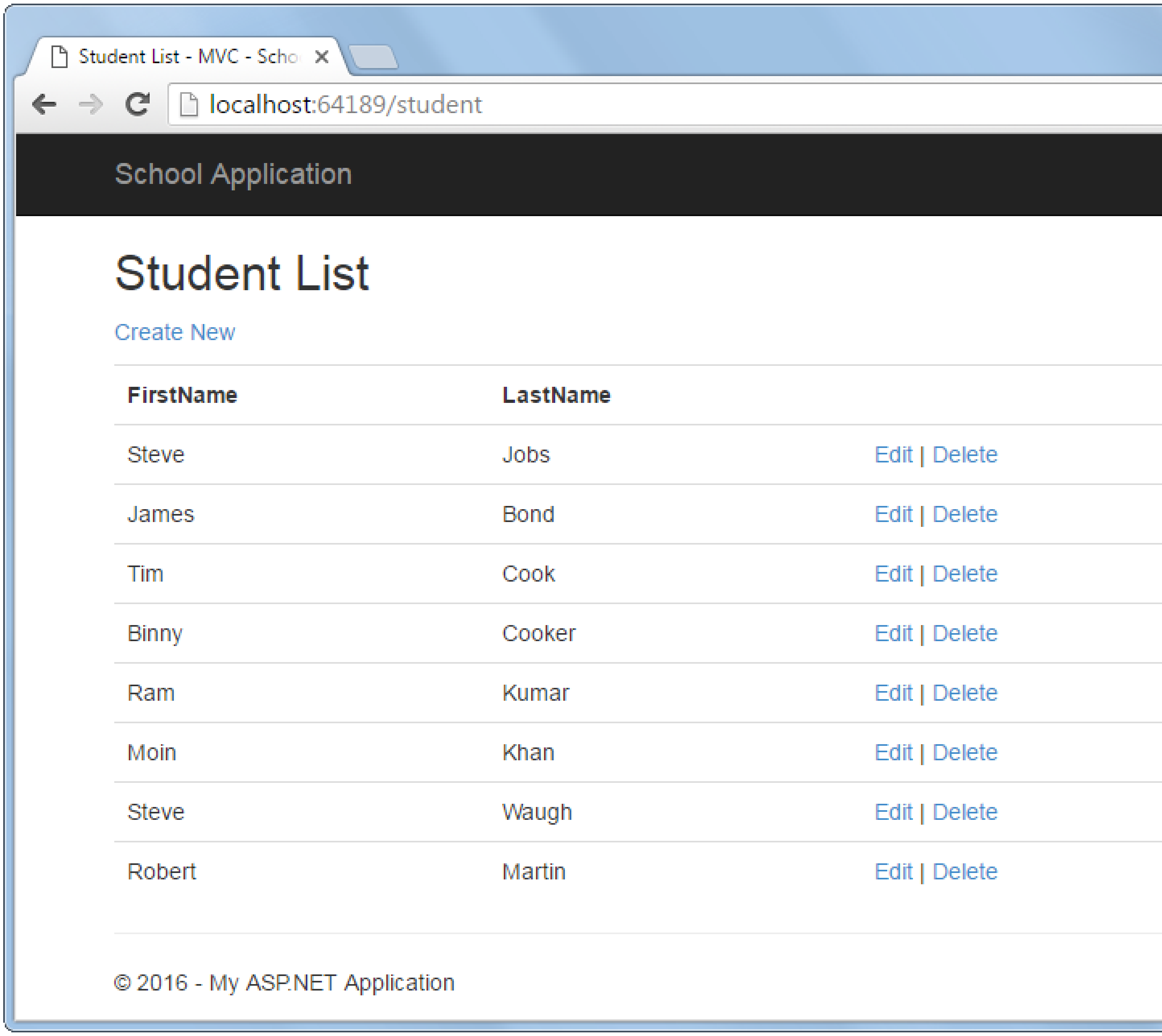
Consume Web API Put method in ASP.NET MVC

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 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 particular record. We will handle edit functionality in this section.



Student List - MVC - Scho

localhost:64189/student

School Application

Student List

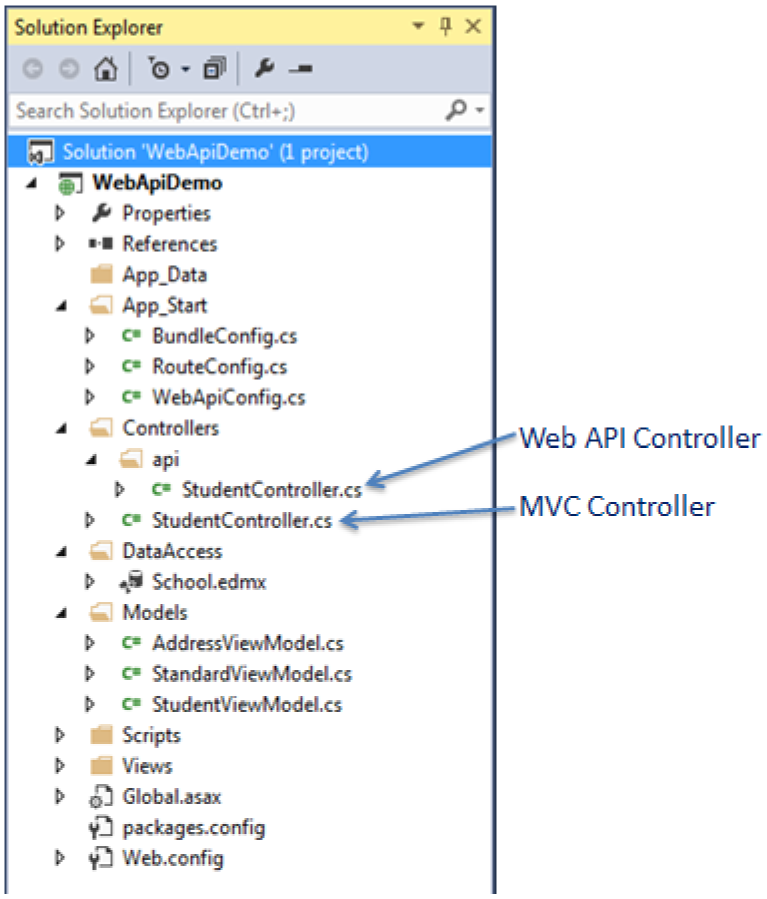
Create New

FirstName	LastName	
Steve	Jobs	Edit | Delete
James	Bond	Edit | Delete
Tim	Cook	Edit | Delete
Binny	Cooker	Edit | Delete
Ram	Kumar	Edit | Delete
Moin	Khan	Edit | Delete
Steve	Waugh	Edit | Delete
Robert	Martin	Edit | Delete

© 2016 - My ASP.NET Application

Student List View

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

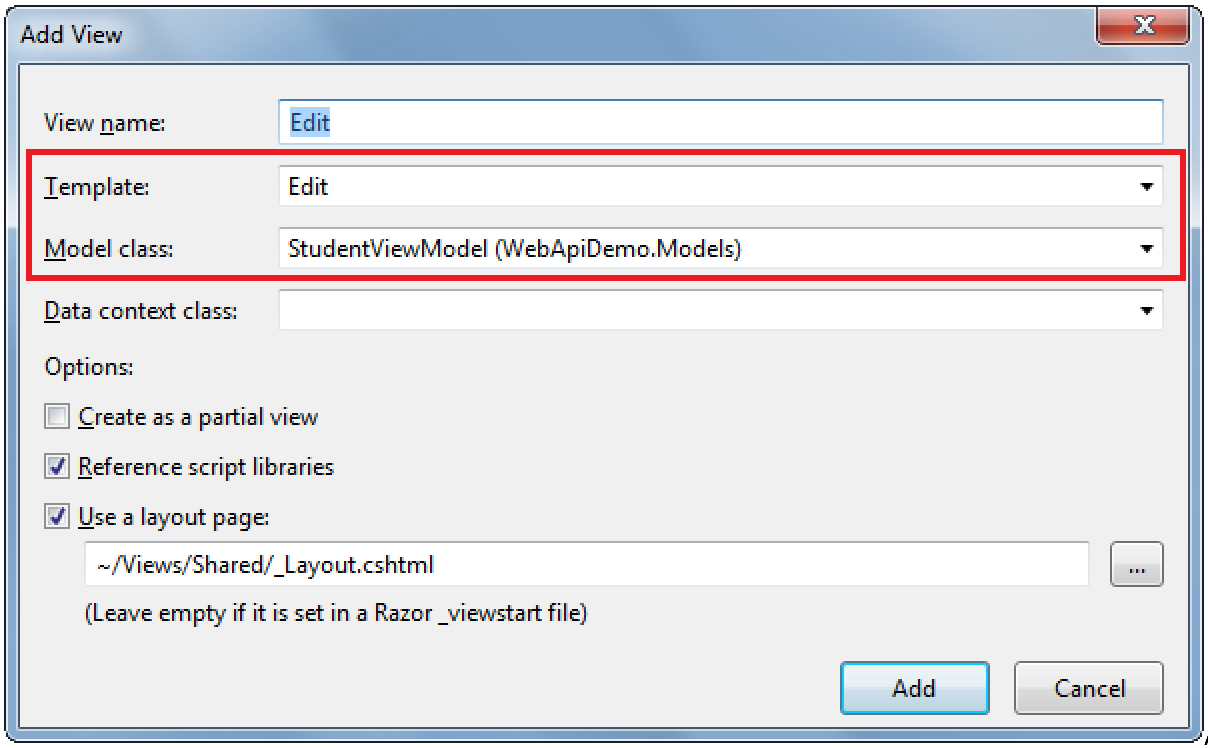
Example: Implement Edit Action Method

```
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, 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. Visit HttpClientsection to know more about it.

Step 2:

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


Add View

View name: Edit

Template: Edit

Model class: StudentViewModel (WebApiDemo.Models)

Data context class:

Options:

☐ Create as a partial view

☑ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

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.

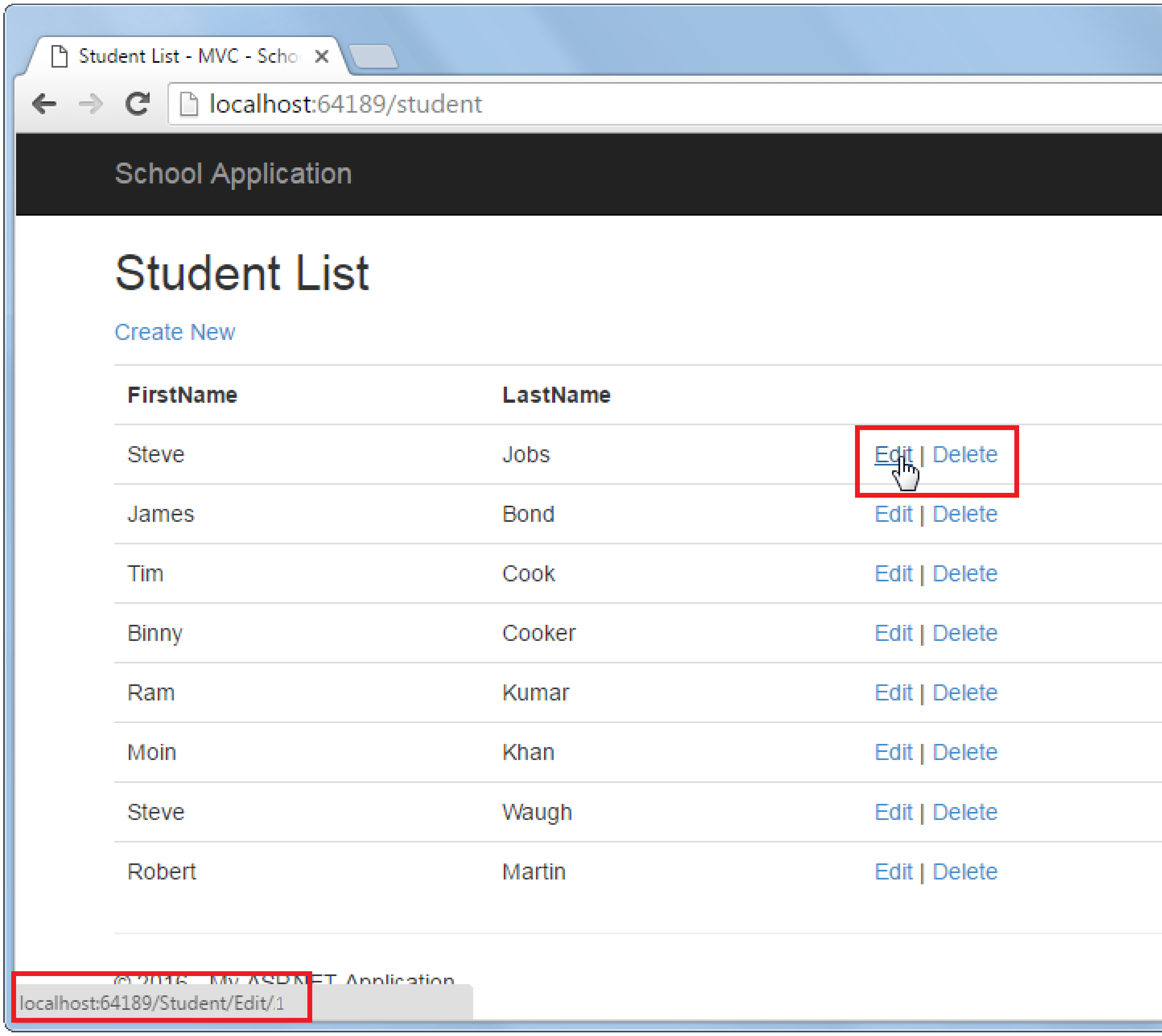
Edit.cshtml

```
@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 following Student List view when you run the project by pressing Ctrl + F5.



Student List - MVC - Scho

localhost:64189/student

School Application

Student List

Create New

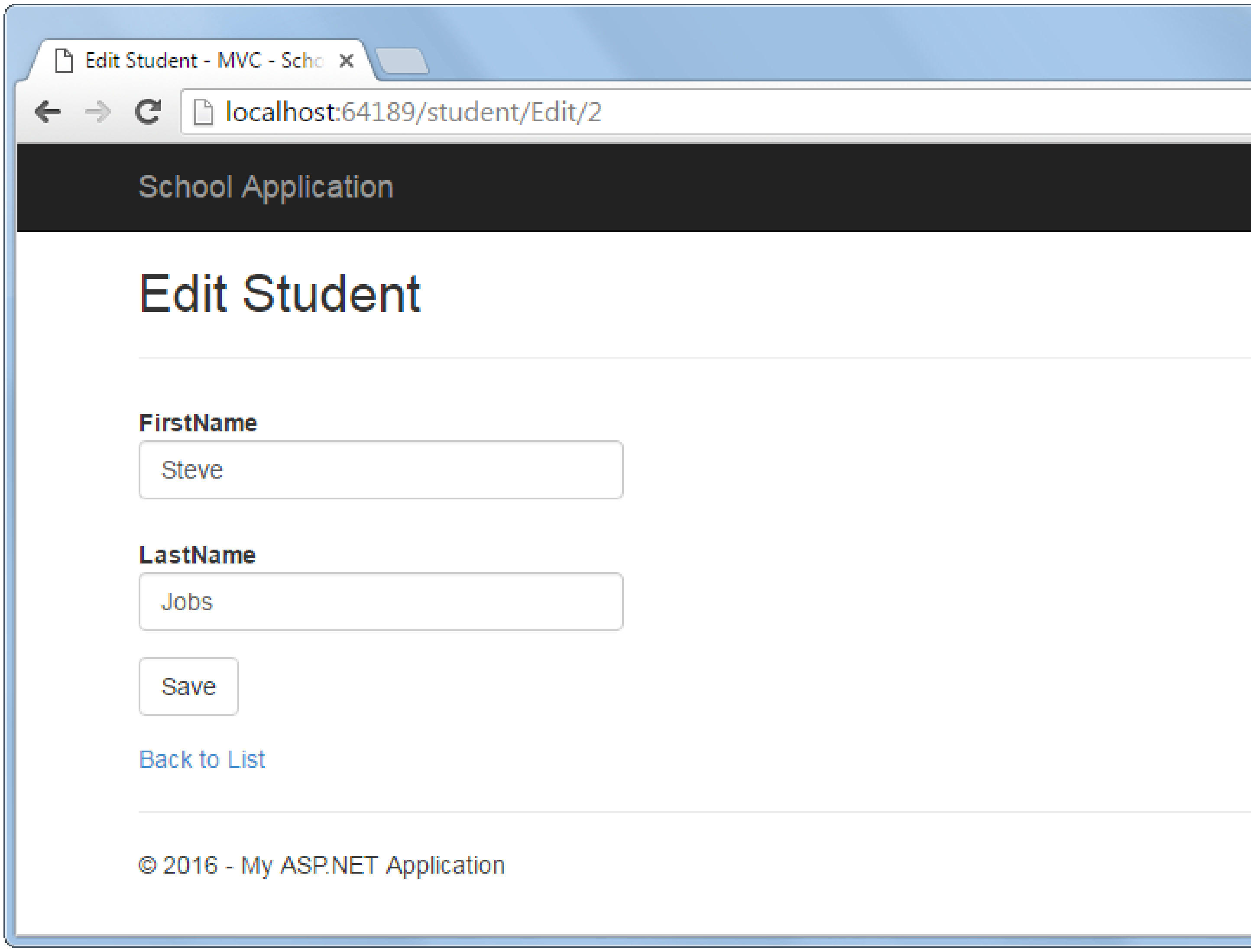
FirstName	LastName	
Steve	Jobs	Edit | Delete
James	Bond	Edit | Delete
Tim	Cook	Edit | Delete
Binny	Cooker	Edit | Delete
Ram	Kumar	Edit | Delete
Moin	Khan	Edit | Delete
Steve	Waugh	Edit | Delete
Robert	Martin	Edit | Delete

© 2016 - My ASP.NET Application

localhost:64189/Student/Edit/1

Student List View

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


Edit Student - MVC - Scho

localhost:64189/student/Edit/2

School Application

Edit Student

FirstName

Steve

LastName

Jobs

Save

Back to List

© 2016 - My ASP.NET Application

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

Consume Web API Delete Method in ASP.NET MVC

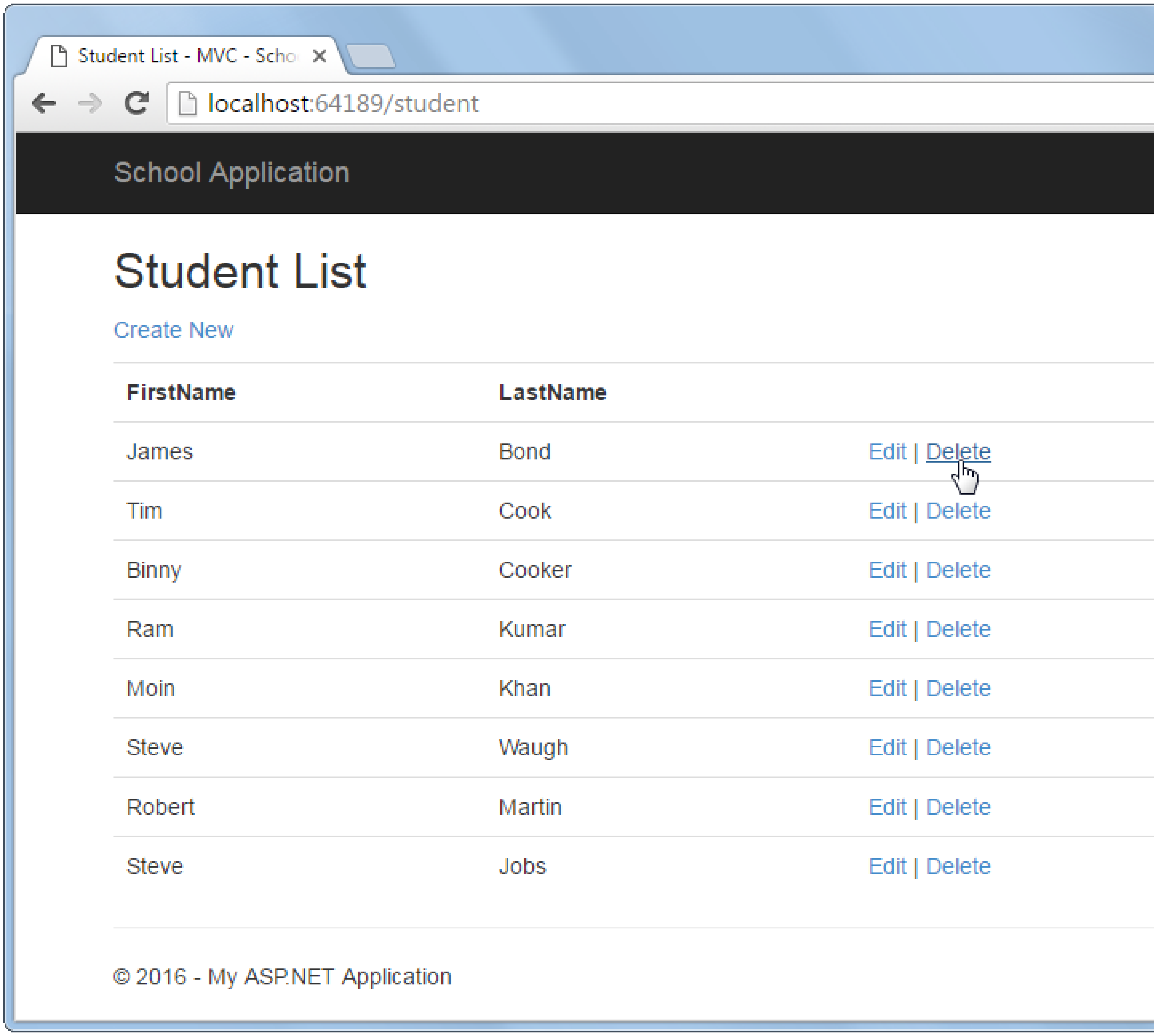
In the previous sections, we consumed Get, Post and Put methods of the Web API. Here, we will consume Delete method of Web API in ASP.NET MVC to delete a record.

We have already created 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 - MVC - Scho

localhost:64189/student

School Application

Student List

Create New

FirstName	LastName	
James	Bond	Edit | Delete
Tim	Cook	Edit | Delete
Binny	Cooker	Edit | Delete
Ram	Kumar	Edit | Delete
Moin	Khan	Edit | Delete
Steve	Waugh	Edit | Delete
Robert	Martin	Edit | Delete
Steve	Jobs	Edit | Delete

© 2016 - My ASP.NET Application

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, 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 section to learn more about it.

So, in this way you can consume 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 2013 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 an 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 completes, we get the result from Task using Task.result which is HttpResponseMessage. Now, you can check the status of an 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 send 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.

Method Name	Description
GetByteArrayAsync	Sends a GET request to the specified Uri and returns the response body as a byte array in an asynch
GetStreamAsync	Sends a GET request to the specified Uri and returns the response body as a stream in an asynchron
GetStringAsync	Sends a GET request to the specified Uri and returns the response body as a string in an asynchrono
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 serial
PostAsXmlAsync	Sends a POST request as an asynchronous operation to the specified Uri with the given value serial
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 serializ
PutAsXmlAsync	Sends a PUT request as an asynchronous operation to the specified Uri with the given value serializ
DeleteAsync	Sends a DELETE request to the specified Uri as an asynchronous operation.

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();
    }
}
```

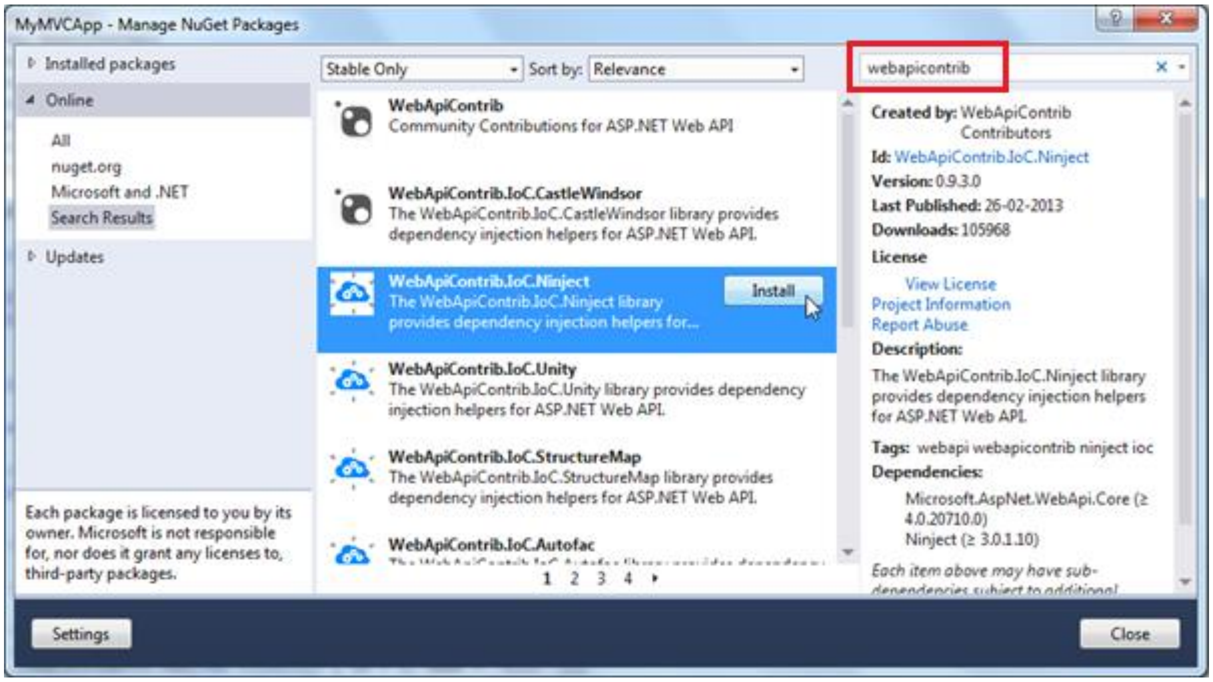
The following are IRepository and StudentRepository class.

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.



WebApiContrib
Community Contributions for ASP.NET Web API

WebApiContrib.IoC.CastleWindsor
The WebApiContrib.IoC.CastleWindsor library provides dependency injection helpers for ASP.NET Web API.

WebApiContrib.IoC.Ninject
The WebApiContrib.IoC.Ninject library provides dependency injection helpers for...

WebApiContrib.IoC.Unity
The WebApiContrib.IoC.Unity library provides dependency injection helpers for ASP.NET Web API.

WebApiContrib.IoC.StructureMap
The WebApiContrib.IoC.StructureMap library provides dependency injection helpers for ASP.NET Web API.

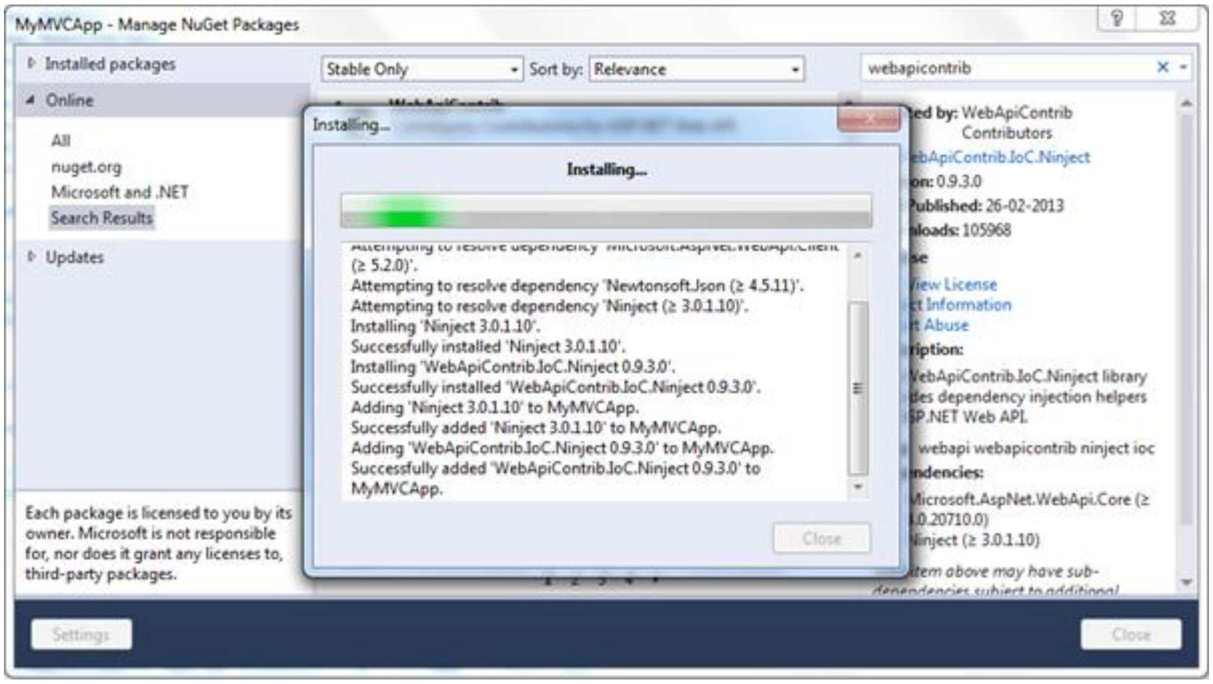
WebApiContrib.IoC.Autofac

Created by: WebApiContrib Contributors
Id: WebApiContrib.IoC.Ninject
Version: 0.9.3.0
Last Published: 26-02-2013
Downloads: 105968
License
View License
Project Information
Report Abuse
Description:
The WebApiContrib.IoC.Ninject library provides dependency injection helpers for ASP.NET Web API.
Tags: webapi webapicontrib ninject ioc
Dependencies:
Microsoft.AspNet.WebApi.Core (≥ 4.0.20710.0)
Ninject (≥ 3.0.1.10)
Each item above may have sub-dependencies subject to additional

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

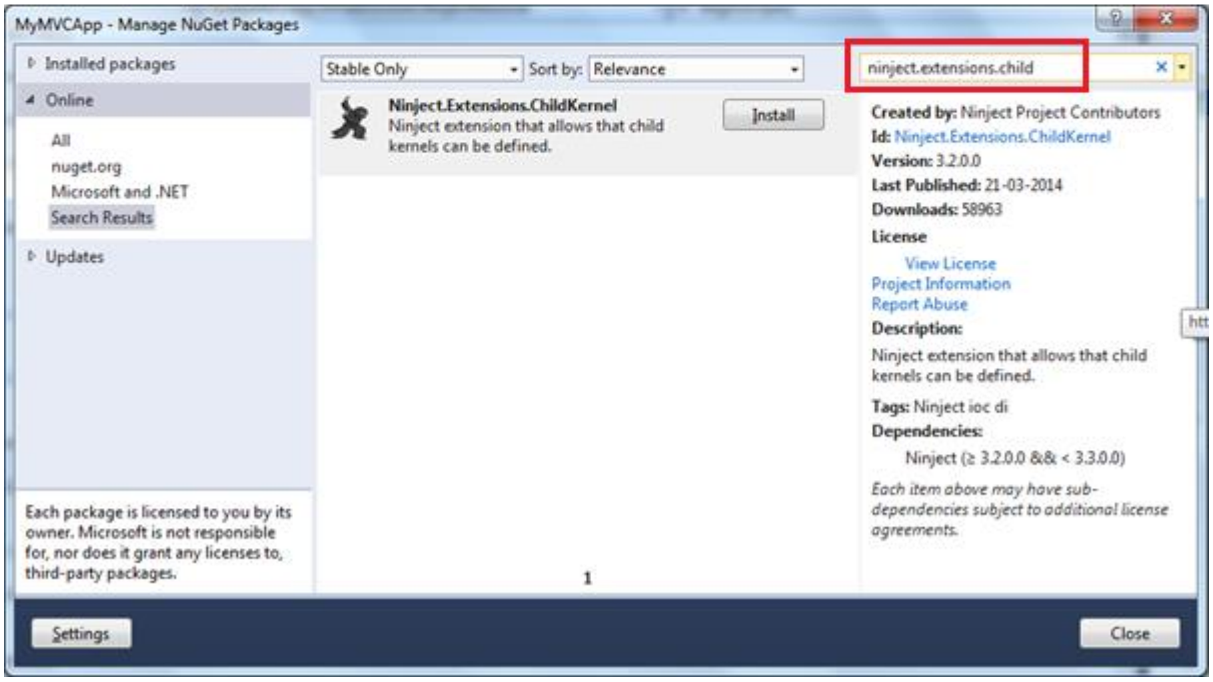
Web API Configuration

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


```
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Client
(≥ 5.2.0)'.
Attempting to resolve dependency 'Newtonsoft.Json (≥ 4.5.11)'.
Attempting to resolve dependency 'Ninject (≥ 3.0.1.10)'.
Installing 'Ninject 3.0.1.10'.
Successfully installed 'Ninject 3.0.1.10'.
Installing 'WebApiContrib.IoC.Ninject 0.9.3.0'.
Successfully installed 'WebApiContrib.IoC.Ninject 0.9.3.0'.
Adding 'Ninject 3.0.1.10' to MyMVCApp.
Successfully added 'Ninject 3.0.1.10' to MyMVCApp.
Adding 'WebApiContrib.IoC.Ninject 0.9.3.0' to MyMVCApp.
Successfully added 'WebApiContrib.IoC.Ninject 0.9.3.0' to
MyMVCApp.
```

Web API Configuration

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



Ninject.Extensions.ChildKernel
Ninject extension that allows that child kernels can be defined.

Created by: Ninject Project Contributors
Id: Ninject.Extensions.ChildKernel
Version: 3.2.0.0
Last Published: 21-03-2014
Downloads: 58963
Description: Ninject extension that allows that child kernels can be defined.
Tags: Ninject ioc di
Dependencies: Ninject (≥ 3.2.0.0 && < 3.3.0.0)

Web API Configuration

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

In order 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, 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 Web API that uses the same base address.

Self Hosting

You can host a Web API as 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.

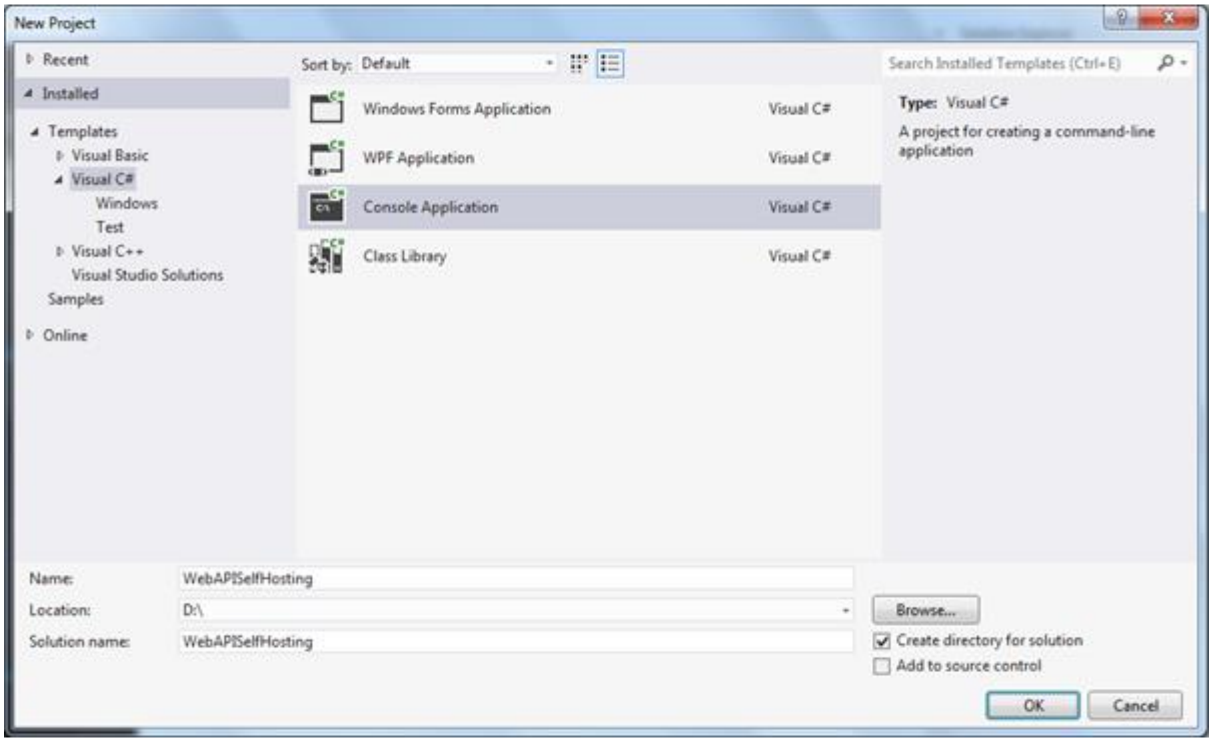
You need to do following steps in order to self-host a web API.

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 of all, create a console project in Visual Studio 2012 for Desktop (or latter version).

Note: You must open Visual Studio in Administration mode.



Type: Visual C#

A project for creating a command-line application

Name: WebAPISelfHosting

Location: D:\

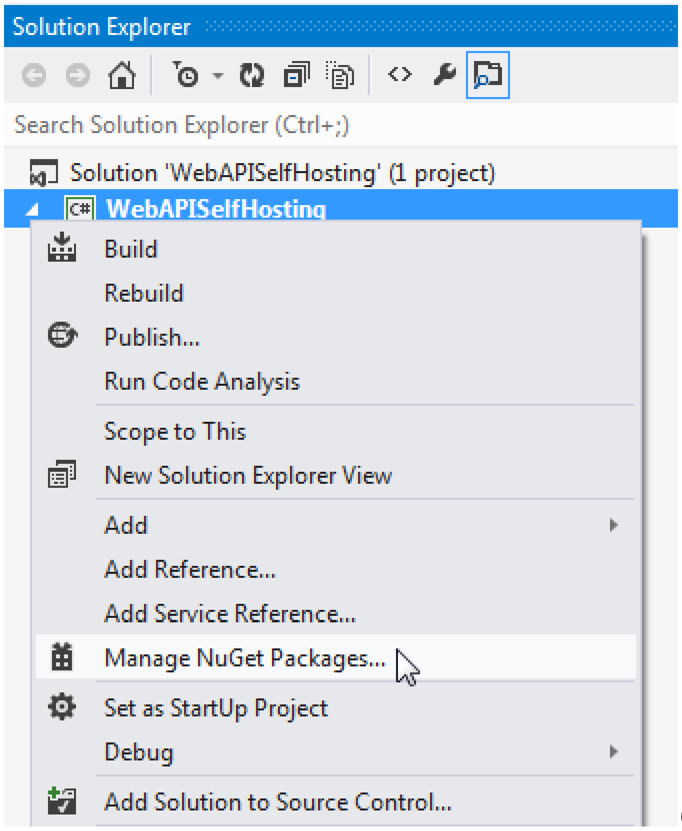
Solution name: WebAPISelfHosting

[✓] Create directory for solution

[ ] Add to source control

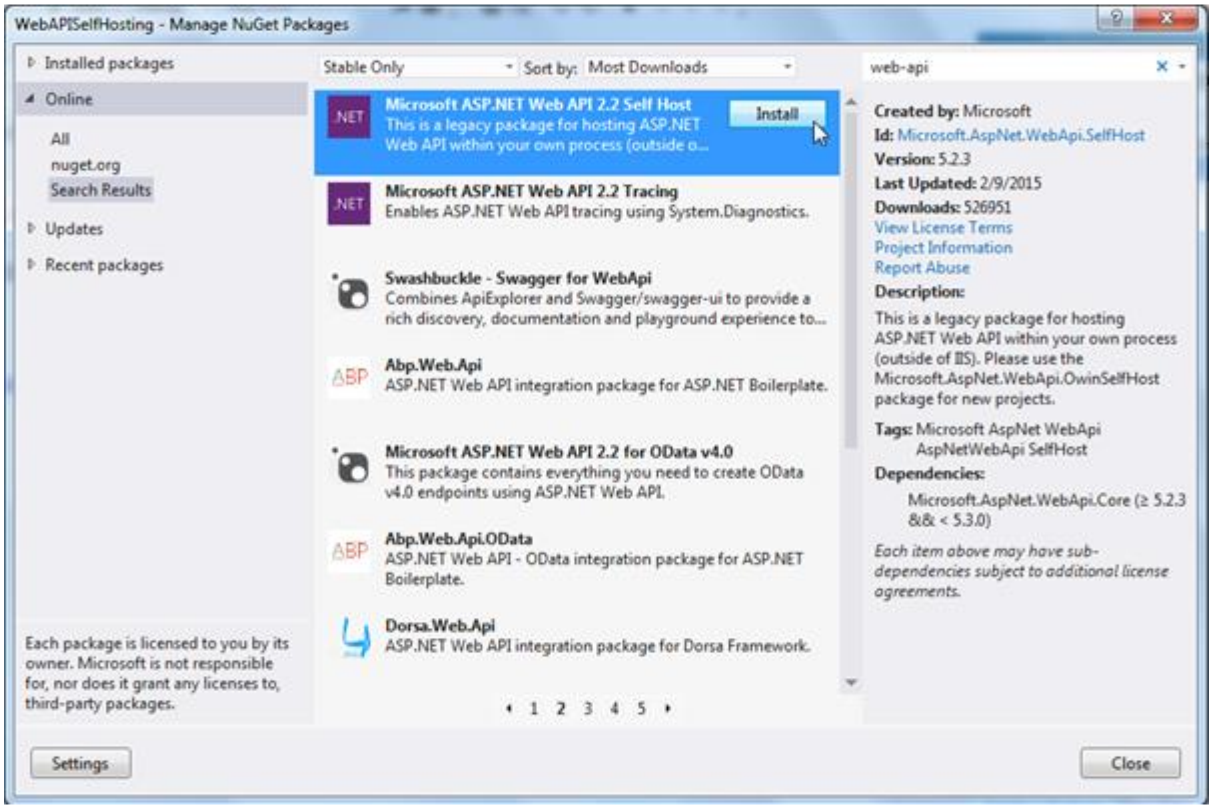
Create Console Application

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 Online option in left pan and search for web-api. This will list all the packages for Web API. Now, look for **Microsoft ASP.NET Web API 2.2 Self Host** package and click **Install**.



Microsoft ASP.NET Web API 2.2 Self Host
This is a legacy package for hosting ASP.NET Web API within your own process (outside o...

Microsoft ASP.NET Web API 2.2 Tracing
Enables ASP.NET Web API tracing using System.Diagnostics.

Swashbuckle - Swagger for WebApi
Combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery, documentation and playground experience to...

Abp.Web.Api
ASP.NET Web API integration package for ASP.NET Boilerplate.

Microsoft ASP.NET Web API 2.2 for OData v4.0
This package contains everything you need to create OData v4.0 endpoints using ASP.NET Web API.

Abp.Web.Api.OData
ASP.NET Web API - OData integration package for ASP.NET Boilerplate.

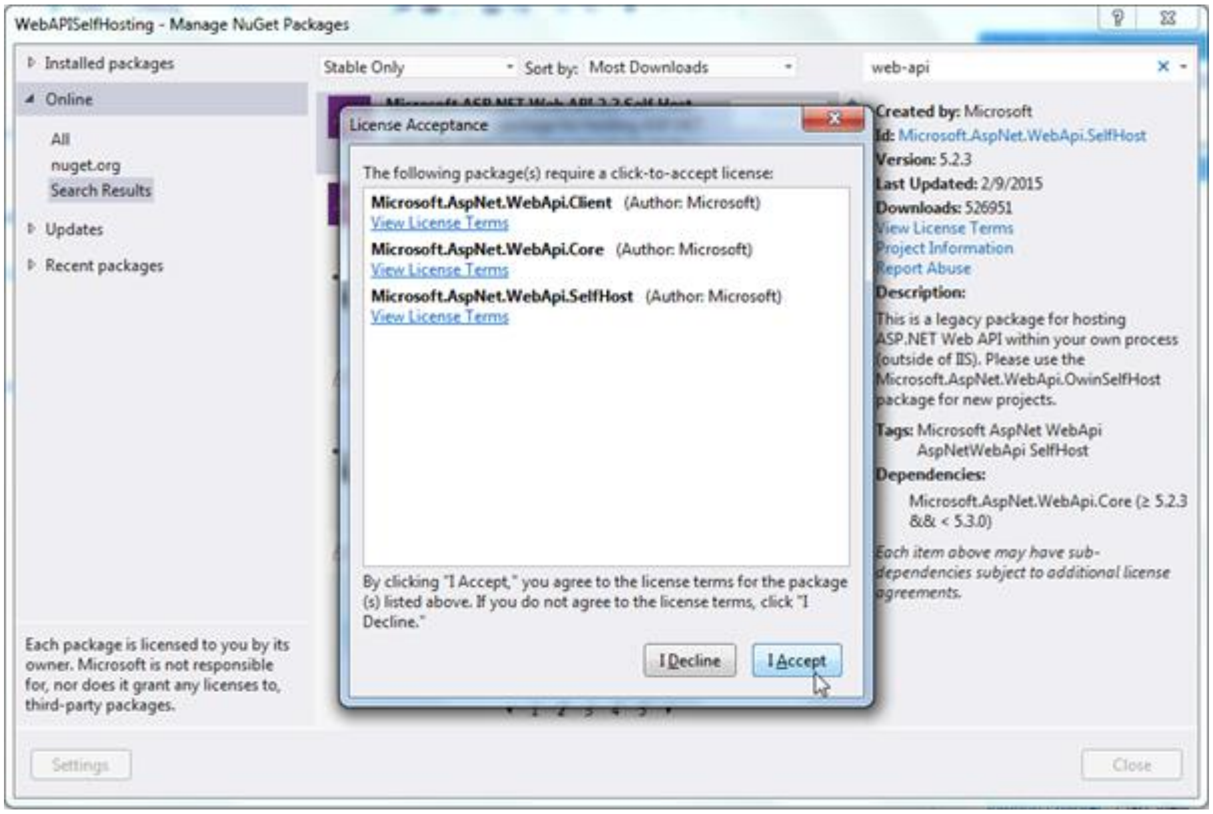
Dorsa.Web.Api
ASP.NET Web API integration package for Dorsa Framework.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Created by: Microsoft
Id: Microsoft.AspNet.WebApi.SelfHost
Version: 5.2.3
Last Updated: 2/9/2015
Downloads: 526951
View License Terms
Project Information
Report Abuse
Description:
This is a legacy package for hosting ASP.NET Web API within your own process (outside of IIS). Please use the Microsoft.AspNet.WebApi.OwinSelfHost package for new projects.
Tags: Microsoft AspNet WebApi AspNetWebApi SelfHost
Dependencies:
Microsoft.AspNet.WebApi.Core (≥ 5.2.3 && < 5.3.0)
Each item above may have sub-dependencies subject to additional license agreements.

Install Web API Self Host Package

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


WebAPISelfHosting - Manage NuGet Packages

License Acceptance

The following package(s) require a click-to-accept license:

Microsoft.AspNet.WebApi.Client (Author: Microsoft)
View License Terms

Microsoft.AspNet.WebApi.Core (Author: Microsoft)
View License Terms

Microsoft.AspNet.WebApi.SelfHost (Author: Microsoft)
View License Terms

By clicking "I Accept," you agree to the license terms for the package (s) listed above. If you do not agree to the license terms, click "I Decline."

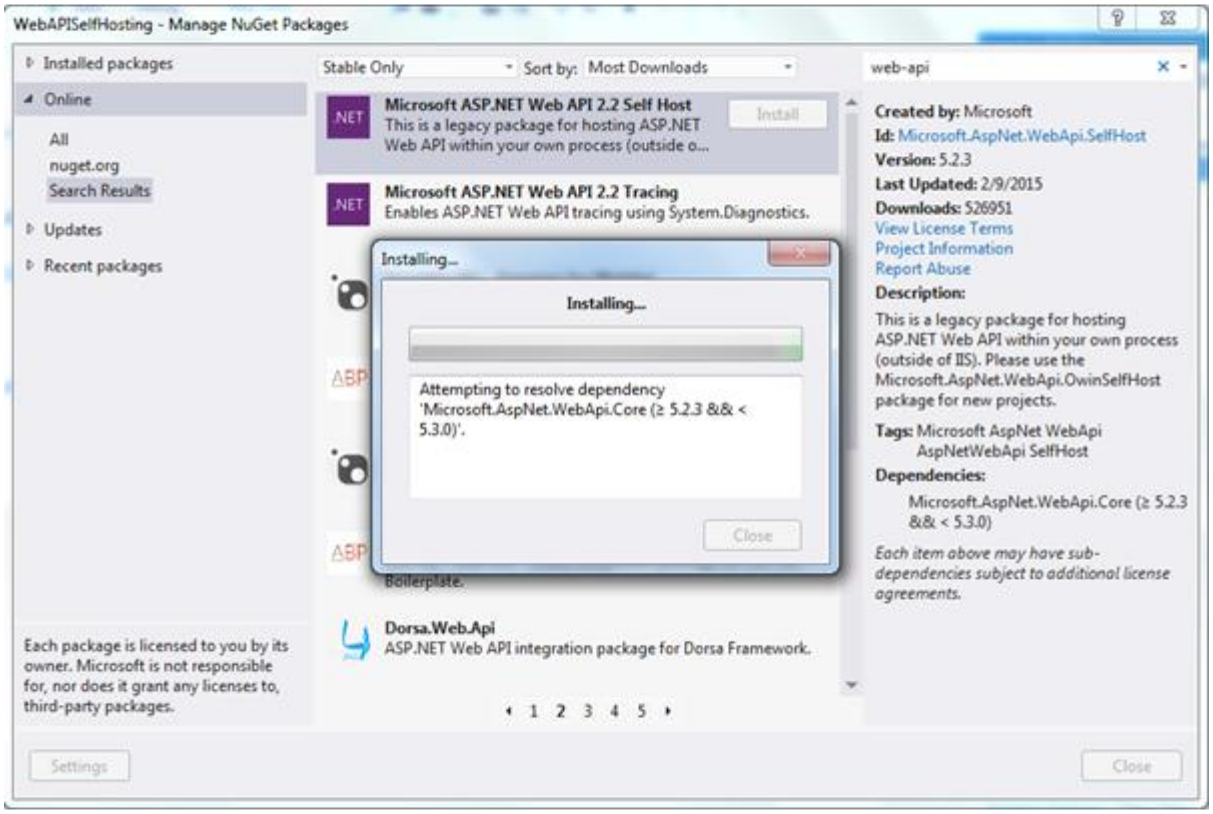
Created by: Microsoft
Id: Microsoft.AspNet.WebApi.SelfHost
Version: 5.2.3
Last Updated: 2/9/2015
Downloads: 526951
View License Terms
Project Information
Report Abuse
Description:
This is a legacy package for hosting ASP.NET Web API within your own process (outside of IIS). Please use the Microsoft.AspNet.WebApi.OwinSelfHost package for new projects.
Tags: Microsoft AspNet WebApi AspNetWebApi SelfHost
Dependencies:
Microsoft.AspNet.WebApi.Core (≥ 5.2.3 && < 5.3.0)

Each item above may have sub-dependencies subject to additional license agreements.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Accept License Agreement

This will install the package into your project.



Microsoft ASP.NET Web API 2.2 Self Host
This is a legacy package for hosting ASP.NET Web API within your own process (outside o...

Microsoft ASP.NET Web API 2.2 Tracing
Enables ASP.NET Web API tracing using System.Diagnostics.

Installing...

Installing...

Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Core (≥ 5.2.3 && < 5.3.0)'.

Boilerplate.

Dorsa.Web.Api
ASP.NET Web API integration package for Dorsa Framework.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Created by: Microsoft
Id: Microsoft.AspNet.WebApi.SelfHost
Version: 5.2.3
Last Updated: 2/9/2015
Downloads: 526951
View License Terms
Project Information
Report Abuse
Description:
This is a legacy package for hosting ASP.NET Web API within your own process (outside of IIS). Please use the Microsoft.AspNet.WebApi.OwinSelfHost package for new projects.
Tags: Microsoft AspNet WebApi AspNetWebApi SelfHost
Dependencies:
Microsoft.AspNet.WebApi.Core (≥ 5.2.3 && < 5.3.0)

Each item above may have sub-dependencies subject to additional license agreements.

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 request by calling server.OpenAsync() method. This will listen requests asynchronously, so it will return Task object.

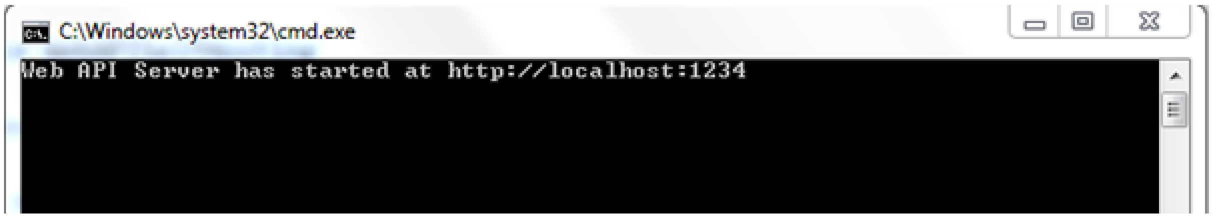
Create `MyWebAPIMessageHandler` class and write the following code.

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.

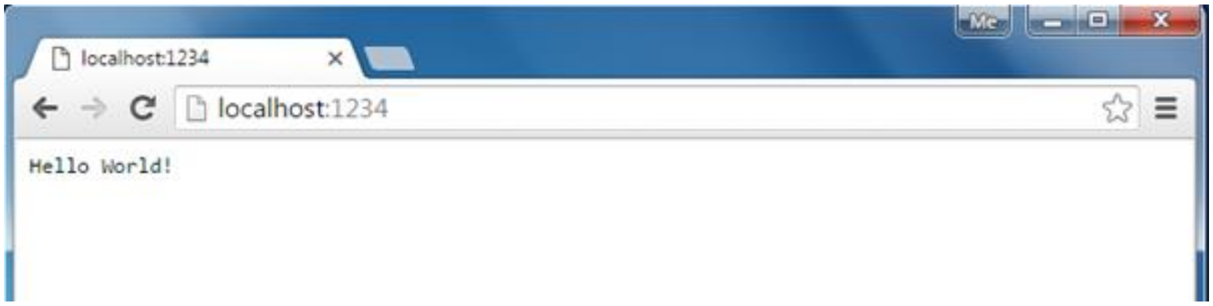
Run the console application using Ctrl + F5.



```
Web API Server has started at http://localhost:1234
```

Run Console Application

Open web browser and enter `http://localhost:1234/` and see the result.



```
Hello World!
```

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;
    }
}
```

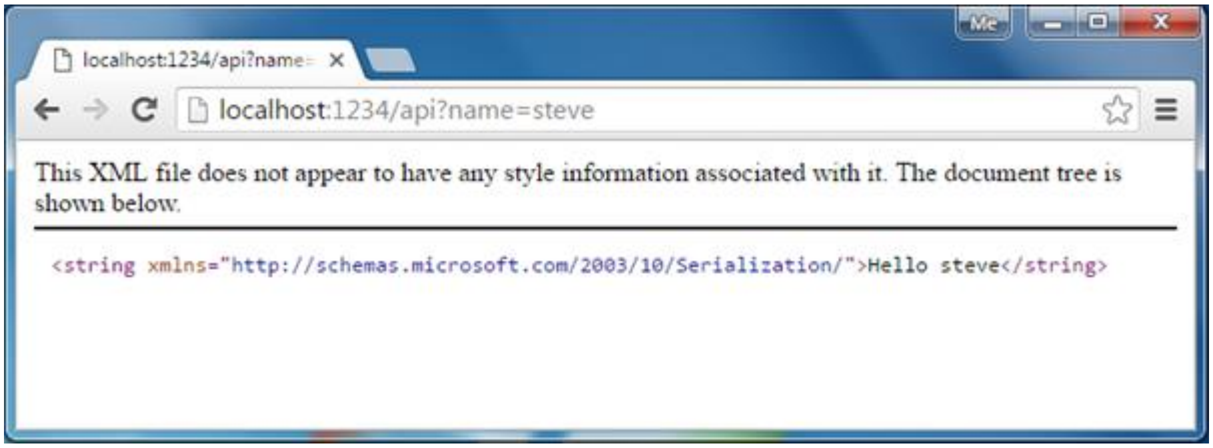
Now, in the Main() method, configure a default route using config object as shown below.

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 notice 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.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Hello steve</string>
```

Web API Response