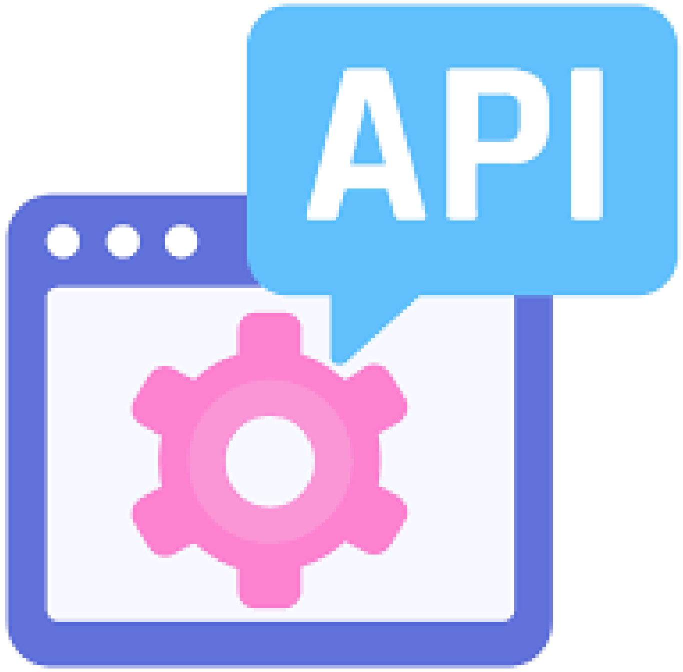
ASP.NET Web API

Agenda

- Introduction to Web API
- Working with HTTP
- Routing
- Formats and Model Bindings
- Security
- Tracing
- Extensibility



Introduction to ASP.NET Web API

What is ASP.NET Web API ?

- HTTP is not just for serving up web pages. It is also a powerful platform for building APIs that expose services and data.
- HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.
- ASP.NET Web API is a framework for building web APIs on top of the .NET Framework.
- A fully supported and extensible framework for building HTTP based endpoints. That can be consume by a broad range of clients including browsers, mobiles, iPhone and tablets.
- It is very similar to ASP.NET MVC since it contains the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection. But it is not a part of the MVC Framework. It is a part of the core ASP.NET platform and can be used with MVC and other types of Web applications like Asp. Net Web Forms.
- Released with ASP.NET MVC 4.0. It is not linked to MVC, we can use it with Asp.NET web Forms also. Web API are available via NuGet.

Web API Features

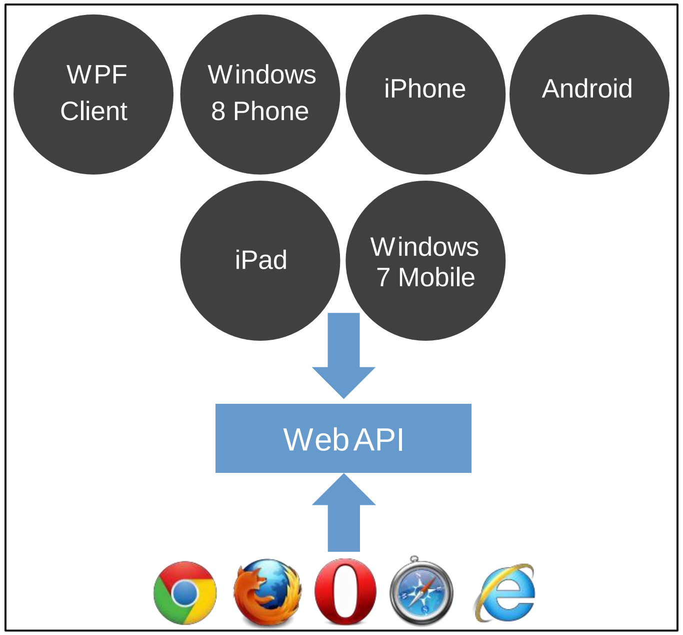
- It supports convention-based CRUD Actions since it works with HTTP verbs GET,POST,PUT and DELETE.
- Responses have an Accept header and HTTP status code.
- Responses are formatted by Web API's MediaTypeFormatter into JSON, XML or whatever format you want to add as a MediaTypeFormatter.
- It may accepts and generates the content which may not be object oriented like images, PDF files etc.
- It has automatic support for OData. Hence by placing the new [Queryable] attribute on a controller method that returns IQueryable, clients can use the method for OData query composition.
- It can be hosted with in the application or on IIS.
- It also supports the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection that makes it more simple and robust.

Why Asp. Net Web API (Web API) ?

- Today, a web-based application is not enough to reach it's customers. People are using iPhone, mobile, tablets etc. devices in its daily life. These devices also have a lot of apps for making the life easy. Actually, we are moving from the web towards apps world.
- If you like to expose your service data to the browsers and as well as all these modern devices apps in fast and simple way, you should have an API which is compatible with browsers and all these devices.
- Web API is the great framework for exposing your data and service to different-different devices. Moreover Web API is open source an ideal platform for building REST-ful services over the .NET Framework.
- Unlike WCF Rest service, it use the full features of HTTP (like URIs, request/response headers, caching, versioning, various content formats) and you don't need to define any extra config settings for different devices unlike WCF Rest service.

Why to choose Web API ?

- If we need a Web Service and don't need SOAP, then ASP. Net Web API is best choice.
- It is Used to build simple, non-SOAP-based HTTP Services on top of existing WCF message pipeline.
- It doesn't have tedious and extensive configuration like WCF REST service.
- Simple service creation with Web API. With WCF REST Services, service creation is difficult.
- It is only based on HTTP and easy to define, expose and consume in a REST-ful way.
- It is light weight architecture and good for devices which have limited bandwidth like smart phones.
- It is open source.



Is this REST ?

- The ASP.NET Web API doesn't dictate an architectural style.
- However RESTful services can be build on top of Web API. Web API doesn't get in way if we want to design using the REST architectural style.
- Web API is open source an ideal platform for building REST-ful services over the .NET Framework.
- Unlike WCF Rest service, it use the full features of HTTP (like URIs, request/response headers, caching, versioning, various content formats)

What is REST?

- REST is nothing but using the current features of the "Web" in a simple and effective way. If you see, some of the amazing features of the Web are:
    - 40 years old matured and widely accepted HTTP protocol.
    - Standard and Unified methods like POST, GET, PUT and DELETE.
    - Stateless nature of HTTP protocol.
    - Easy to use URI (Uniform resource identifier) format to locate any web resource.

- REST leverages these amazing features of the web with some constraints. There are 5 basic fundamentals of web which are leveraged to create REST services.

- Principle 1: Everything is a Resource
- Principle 2: Every Resource is Identified by a Unique Identifier
- Principle 3: Use Simple and Uniform Interfaces
- Principle 4: Communication is Done by Representation
- Principle 5: Be Stateless

WCF and ASP.NET Web API

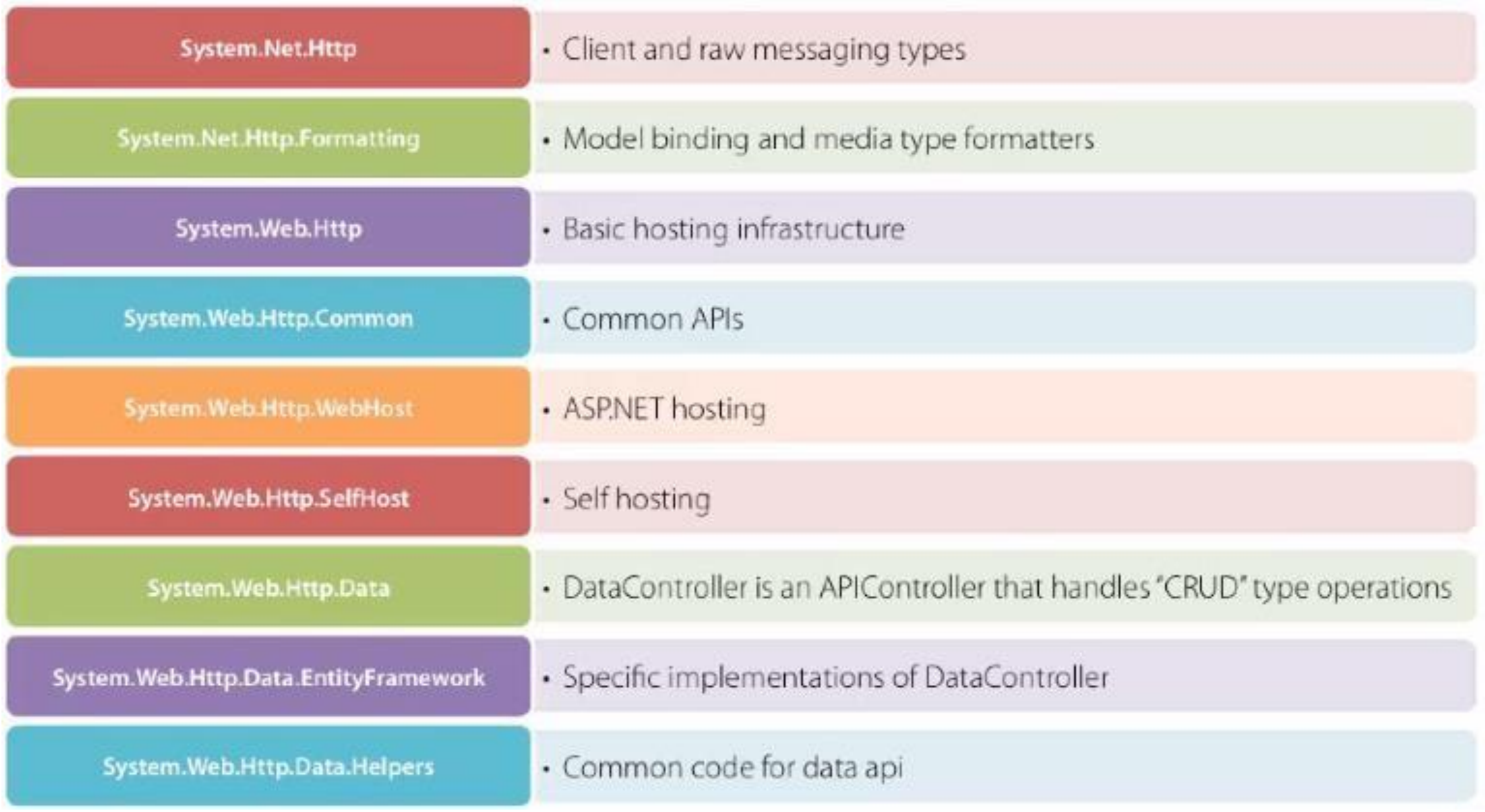
- WCF is Microsoft's unified programming model for building service-oriented applications.
- It enables developers to build secure, reliable, transacted solutions that integrate across platforms and interoperate with existing investments. (ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

Choosing which technology to use

- The following table describes the major features of each technology.

WCF	ASP.NET Web API
Enables building services that support multiple transport protocols (HTTP, TCP, UDP, and custom transports) and allows switching between them.	HTTP only. First-class programming model for HTTP. More suitable for access from various browsers, mobile devices etc enabling wide reach.
Enables building services that support multiple encodings (Text, MTOM, and Binary) of the same message type and allows switching between them.	Enables building Web APIs that support wide variety of media types including XML, JSON etc.
Supports building services with WS-* standards like Reliable Messaging, Transactions, Message Security.	Uses basic protocol and formats such as HTTP, WebSockets, SSL, JQuery, JSON, and XML. There is no support for higher level protocols such as Reliable Messaging or Transactions.
Supports Request-Reply, One Way, and Duplex message exchange patterns.	HTTP is request/response but additional patterns can be supported through SignalR and WebSockets integration.
WCF SOAP services can be described in WSDL allowing automated tools to generate client proxies even for services with complex schemas.	There is a variety of ways to describe a Web API ranging from auto-generated HTML help page describing snippets to structured metadata for OData integrated APIs.
Ships with the .NET framework.	Ships with .NET framework but is open-source and is also available out-of-band as independent download.

Assemblies



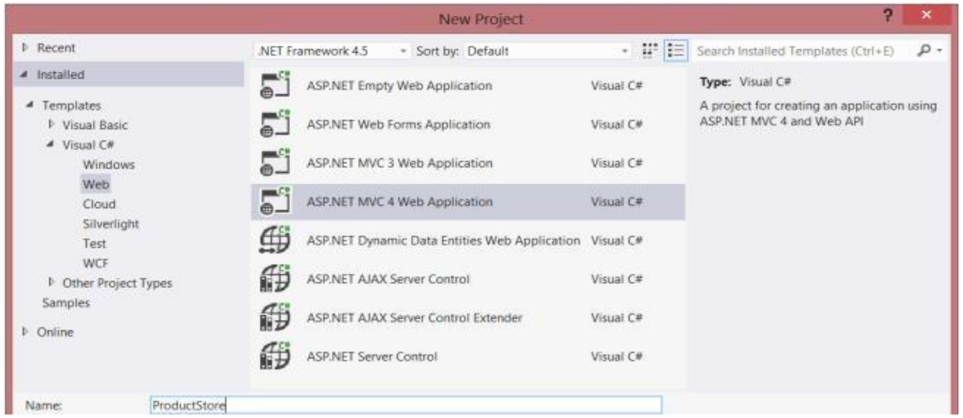
System.Net.Http	• Client and raw messaging types
System.Net.Http.Formatting	• Model binding and media type formatters
System.Web.Http	• Basic hosting infrastructure
System.Web.Http.Common	• Common APIs
System.Web.Http.WebHost	• ASP.NET hosting
System.Web.Http.SelfHost	• Self hosting
System.Web.Http.Data	• DataController is an APIController that handles "CRUD" type operations
System.Web.Http.Data.EntityFramework	• Specific implementations of DataController
System.Web.Http.Data.Helpers	• Common code for data api

Creating a Web API

- The four main HTTP methods (GET, PUT, POST, and DELETE) can be mapped to CRUD operations as follows:
- GET retrieves the representation of the resource at a specified URI. GET should have no side effects on the server.
- PUT updates a resource at a specified URI. PUT can also be used to create a new resource at a specified URI, if the server allows clients to specify new URIs. For this tutorial, the API will not support creation through PUT.
- POST creates a new resource. The server assigns the URI for the new object and returns this URI as part of the response message.
- DELETE deletes a resource at a specified URL.

Create a New Web API Project

- Start by running Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.
- In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.


New Project

- Recent
- Installed
  - Templates
    - Visual Basic
    - Visual C#
      - Windows
      - Web
      - Cloud
      - Silverlight
      - Test
      - WCF
    - Other Project Types
  - Samples
- Online

.NET Framework 4.5 · Sort by: Default · Search Installed Templates (Ctrl+E)

ASP.NET Empty Web Application	Visual C#
ASP.NET Web Forms Application	Visual C#
ASP.NET MVC 3 Web Application	Visual C#
ASP.NET MVC 4 Web Application	Visual C#
ASP.NET Dynamic Data Entities Web Application	Visual C#
ASP.NET AJAX Server Control	Visual C#
ASP.NET AJAX Server Control Extender	Visual C#
ASP.NET Server Control	Visual C#

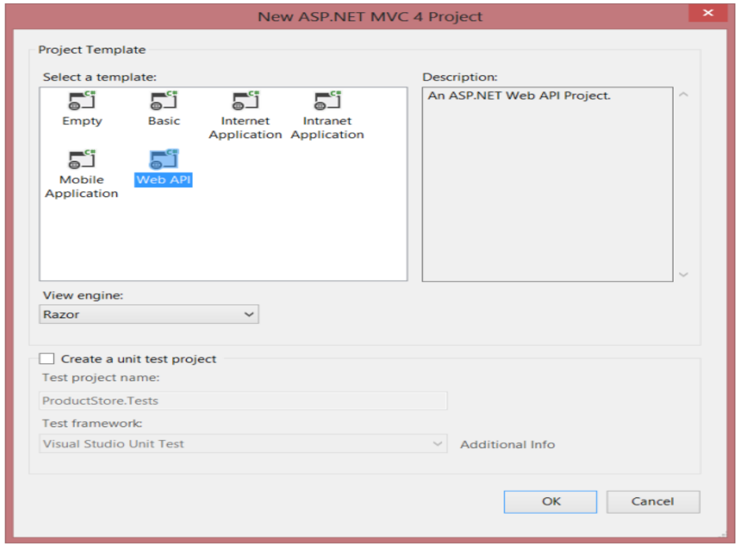
Type: Visual C#

A project for creating an application using ASP.NET MVC 4 and Web API

Name: ProductStore

Creating a Web API

In the **New ASP.NET MVC 4 Project** dialog, select **Web API** and click **OK**.



New ASP.NET MVC 4 Project

Project Template

Select a template: Empty, Basic, Internet Application, Intranet Application, Mobile Application, Web API

Description: An ASP.NET Web API Project.

View engine: Razor

Create a unit test project

Test project name: ProductStore.Tests

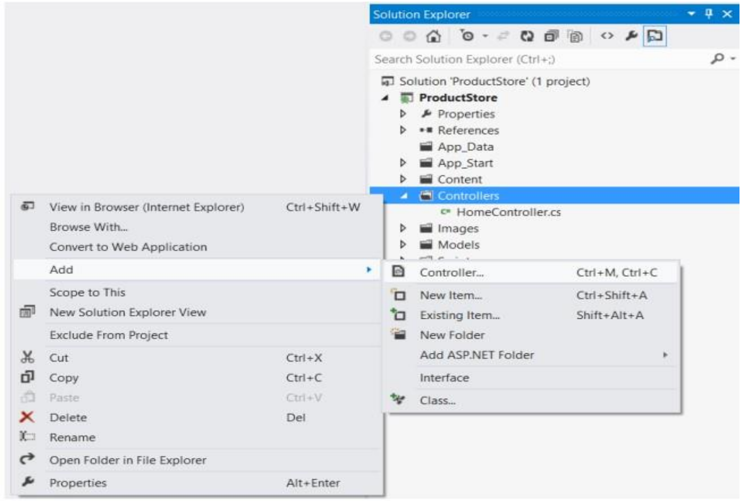
Test framework: Visual Studio Unit Test

Additional Info

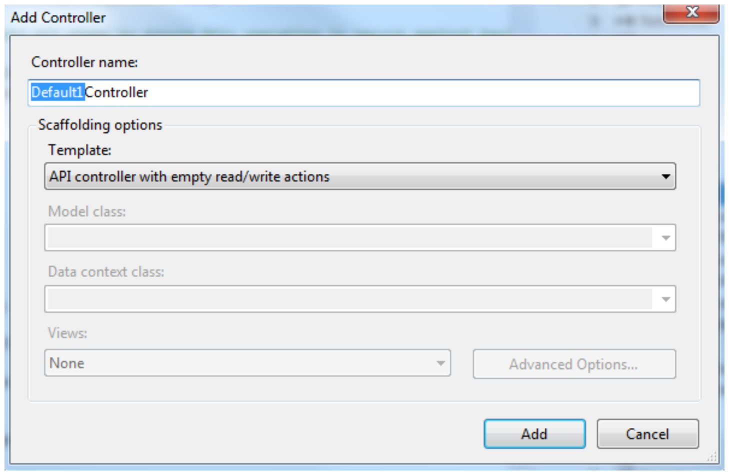
OK Cancel

Adding a Web API Controller

- If you have worked with ASP.NET MVC, then you are already familiar with controllers. In ASP.NET Web API, a *controller* is a class that handles HTTP requests from the client. The New Project wizard created two controllers for you when it created the project. To see them, expand the Controllers folder in Solution Explorer.
- HomeController is a traditional ASP.NET MVC controller. It is responsible for serving HTML pages for the site, and is not directly related to our web API.
- ValuesController is an example WebAPI controller.
- Go ahead and delete ValuesController, by right-clicking the file in Solution Explorer and selecting **Delete.** Now add a new controller, as shown.
- In the **Add Controller** wizard, name the controller "ProductsController". In the **Template** drop-down list, select **Empty API Controller**. Then click **Add**.
- It is not necessary to put your contollers into a folder named Controllers. The folder name is not important; it is simply a convenient way to organize your source files.

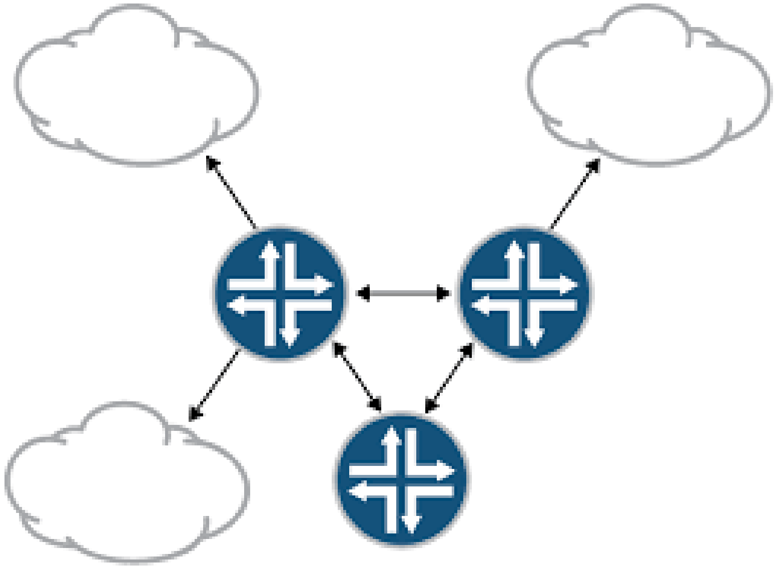


Creating a Web API



- The method name starts with "Get", so by convention it maps to GET requests. Also, because the method has no parameters, it maps to a URI that does not contain an *"id"* segment in the path.
- The method name starts with "Post...". To create a new product, the client sends an HTTP POST request.
- The method name starts with "Put...", so Web API matches it to PUT requests.
- To delete a resource, define a "Delete..." method. If a DELETE request succeeds, it can return status 200 (OK) with an entity-body that describes the status; status 202 (Accepted) if the deletion is still pending; or status 204 (No Content) with no entity body.

```
public class Default1Controller : ApiController
{
    // GET api/default1
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/default1/5
    public string Get(int id)
    {
        return "value";
    }

    // POST api/default1
    public void Post([FromBody]string value)
    {
    }

    // PUT api/default1/5
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE api/default1/5
    public void Delete(int id)
    {
    }
}
```



Routing

Routing

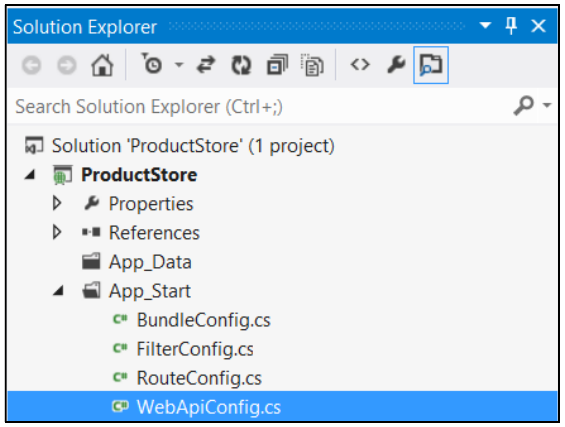
- ASP.NET Web API routes HTTP requests to controllers.
- In ASP.NET Web API, a controller is a class that handles HTTP requests.
- The public methods of the controller are called action methods or simply actions.
- When the Web API framework receives a request, it routes the request to an action.
- To determine which action to invoke, the framework uses a routing table.
- Template for Web API creates a default route

```
routes.MapHttpRoute(
    name: "API Default",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

- This route is defined in the WebApiConfig.cs file, which is placed in the App_Start directory

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

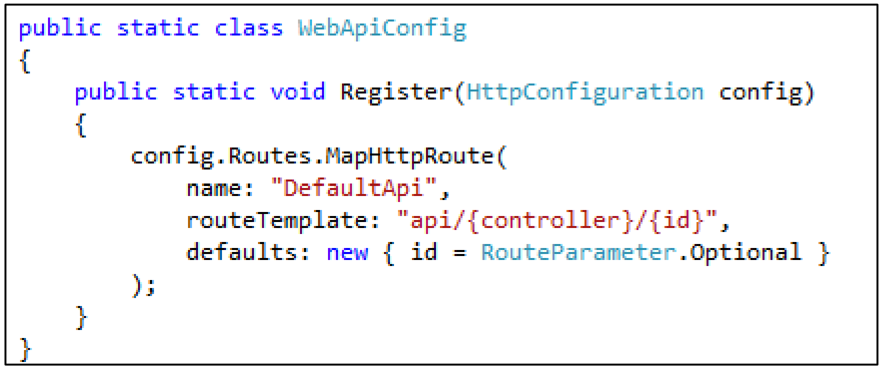
Global.aspx


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ProductStore' (1 project)
ProductStore
Properties
References
App_Data
App_Start
BundleConfig.cs
FilterConfig.cs
RouteConfig.cs
WebApiConfig.cs

WebApiConfig

Routing

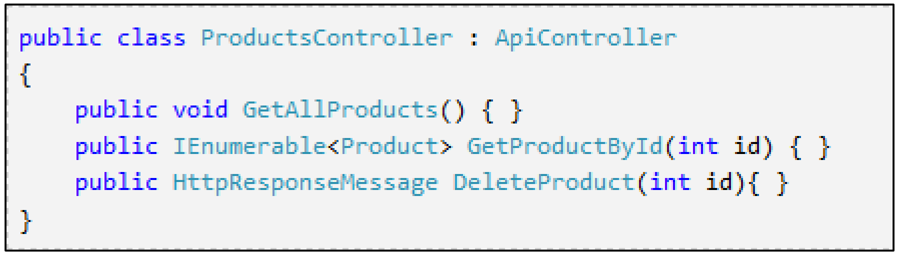
- Entry in the routing table contains a route template.
- The default route template for Web API is "api/{controller}/{id}".
- In this template, "api" is a literal path segment, and {controller} and {id} are placeholder variables.
- When the Web API framework receives an HTTP request, it tries to match the URI against one of the route templates in the routing table.
- If no route matches, the client receives a 404 error.
- The reason for using "api" in the route is to avoid collisions with ASP.NET MVC routing.



```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Once a matching route is found, Web API selects the controller and the action:

- To find the controller, Web API adds "Controller" to the value of the ***{controller}*** variable.
- To find the action, Web API looks at the HTTP method, and then looks for an action whose name begins with that HTTP method name. For example, with a GET request, Web API looks for an action that starts with "Get...", such as "GetContact" or "GetAllContacts". This convention applies only to GET, POST, PUT, and DELETE methods. You can enable other HTTP methods by using attributes on your controller.
- Other placeholder variables in the route template, such as ***{id}***, are mapped to action parameters.



```
public class ProductsController : ApiController
{
    public void GetAllProducts() { }
    public IEnumerable<Product> GetProductById(int id) { }
    public HttpResponseMessage DeleteProduct(int id){ }
}
```

Routing

- Here are some possible HTTP requests, along with the action that gets invoked for each:

HTTP Method	URI Path	Action	Parameter
GET	api/products	GetAllProducts	*(none)*
GET	api/products/4	GetProductById	4
DELETE	api/products/4	DeleteProduct	4
POST	api/products	*(no match)*	

- Notice that the *{id}* segment of the URI, if present, is mapped to the *id* parameter of the action. In this example, the controller defines two GET methods, one with an *id* parameter and one with no parameters.
- POST request will fail, because the controller does not define a "Post..." method.

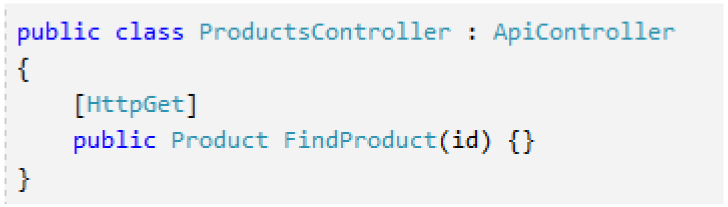
Routing Variations

We have seen basic routing mechanism for ASP.NET Web API. Let's describe some variations

- HTTP Methods
- Routing by Action Name
- Non-Actions

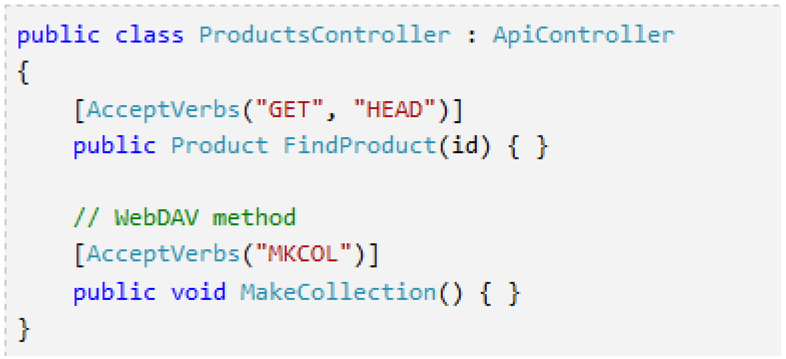
Routing Variations – HTTP Methods

- Instead of using the naming convention for HTTP methods, you can explicitly specify the HTTP method for an action by decorating the action method with the **HttpGet**, **HttpPut**, **HttpPost**, or **HttpDelete** attribute.



```
public class ProductsController : ApiController
{
    [HttpGet]
    public Product FindProduct(id) {}
}
```

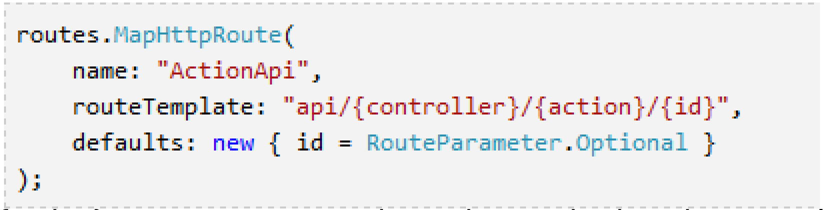
- To allow multiple HTTP methods for an action, or to allow HTTP methods other than GET, PUT, POST, and DELETE, use the **AcceptVerbs** attribute, which takes a list of HTTP methods.



```
public class ProductsController : ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Product FindProduct(id) { }

    // WebDAV method
    [AcceptVerbs("MKCOL")]
    public void MakeCollection() { }
}
```

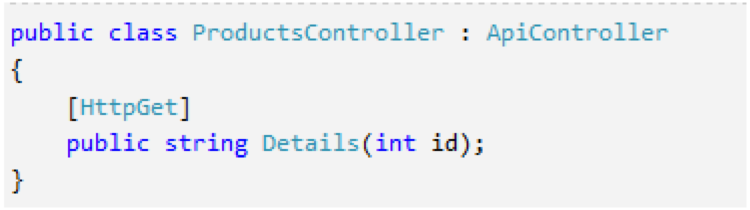
Routing Variations – Routing by Action Name

- With the default routing template, Web API uses the HTTP method to select the action. However, you can also create a route where the action name is included in the URI:



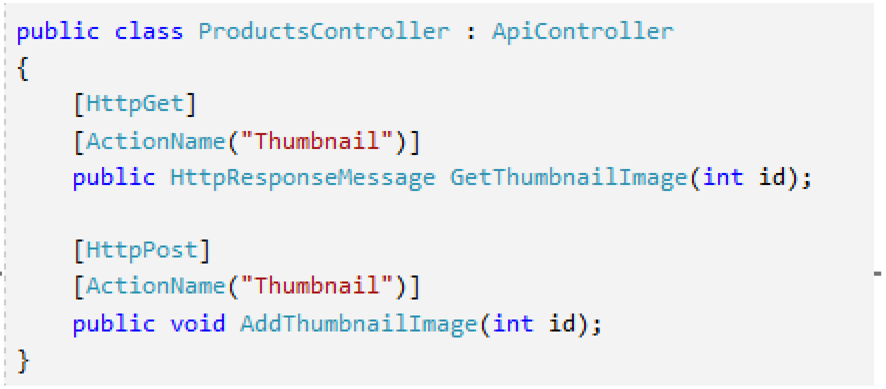
```
routes.MapHttpRoute(
    name: "ActionApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In this route template, the *{action}* parameter names the action method on the controller. With this style of routing, use attributes to specify the allowed HTTP methods. For example, suppose your controller has the following method:



```
public class ProductsController : ApiController
{
    [HttpGet]
    public string Details(int id);
}
```

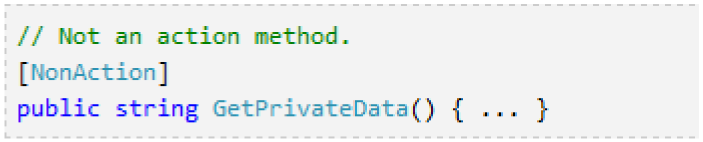
- In this case, a GET request for “api/products/details/1” would map to the Details method. This style of routing is similar to ASP.NET MVC, and may be appropriate for an RPC-style API.
- You can override the action name by using the **ActionName** attribute. In the following example, there are two actions that map to "api/products/thumbnail/*id*. One supports GET and the other supports POST:



```
public class ProductsController : ApiController
{
    [HttpGet]
    [ActionName("Thumbnail")]
    public HttpResponseMessage GetThumbnailImage(int id);

    [HttpPost]
    [ActionName("Thumbnail")]
    public void AddThumbnailImage(int id);
}
```

Routing Variations – Non-Action

To prevent a method from getting invoked as an action, use the **NonAction** attribute. This signals to the framework that the method is not an action, even if it would otherwise match the routing rules.



```
// Not an action method.
[NonAction]
public string GetPrivateData() { ... }
```

Exception Handling in ASP.NET Web API

Following are the errors and exception handling in ASP.NET Web API.

- HttpResponseException
- Exception Filters
- Registering Exception Filters
- HttpError

HttpResponseException

What happens if a Web API controller throws an uncaught exception? By default, most exceptions are translated into an HTTP response with status code 500, Internal Server Error. The **HttpResponseException** type is a special case. This exception returns any HTTP status code that you specify in the exception constructor. For example, the following method returns 404, Not Found, if the *id* parameter is not valid.

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

For more control over the response, you can also construct the entire response message and include it with the **HttpResponseException:**

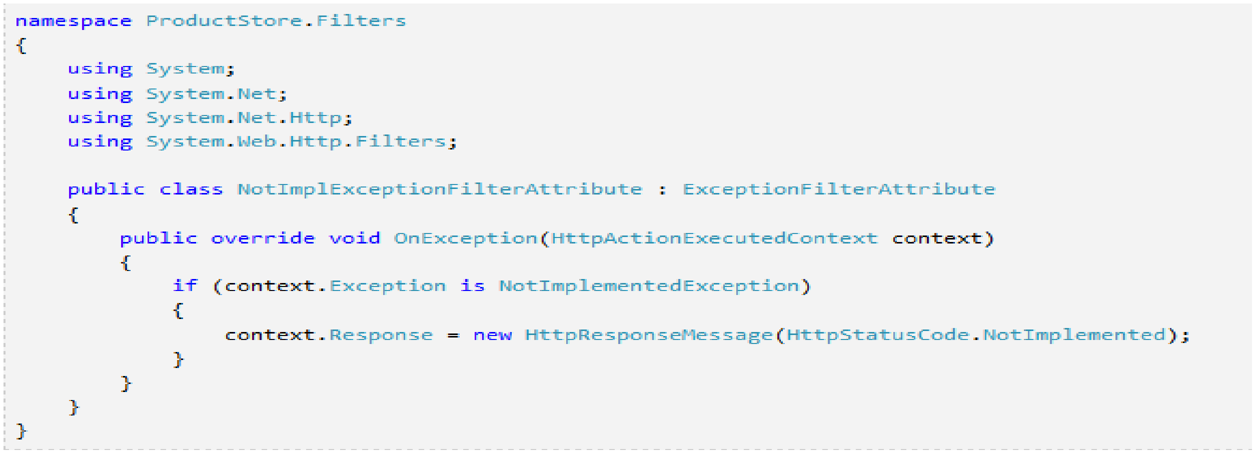
```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var resp = new HttpResponseMessage(HttpStatusCode.NotFound)
        {
            Content = new StringContent(string.Format("No product with ID = {0}", id)),
            ReasonPhrase = "Product ID Not Found"
        }
        throw new HttpResponseException(resp);
    }
    return item;
}
```

Exception Handling in ASP.NET Web API

Exception Filters

You can customize how Web API handles exceptions by writing an *exception filter*. An exception filter is executed when a controller method throws any unhandled exception that is *not* an **HttpResponseException** exception. The **HttpResponseException** type is a special case, because it is designed specifically for returning an HTTP response.

Exception filters implement the **System.Web.Http.Filters.IExceptionFilter** interface. The simplest way to write an exception filter is to derive from the **System.Web.Http.Filters.ExceptionFilterAttribute** class and override the **OnException** method. Here is a filter that converts **NotImplementedException** exceptions into HTTP status code 501, Not Implemented:



```
namespace ProductStore.Filters
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http.Filters;

    public class NotImplExceptionFilterAttribute : ExceptionFilterAttribute
    {
        public override void OnException(HttpActionExecutedContext context)
        {
            if (context.Exception is NotImplementedException)
            {
                context.Response = new HttpResponseMessage(HttpStatusCode.NotImplemented);
            }
        }
    }
}
```

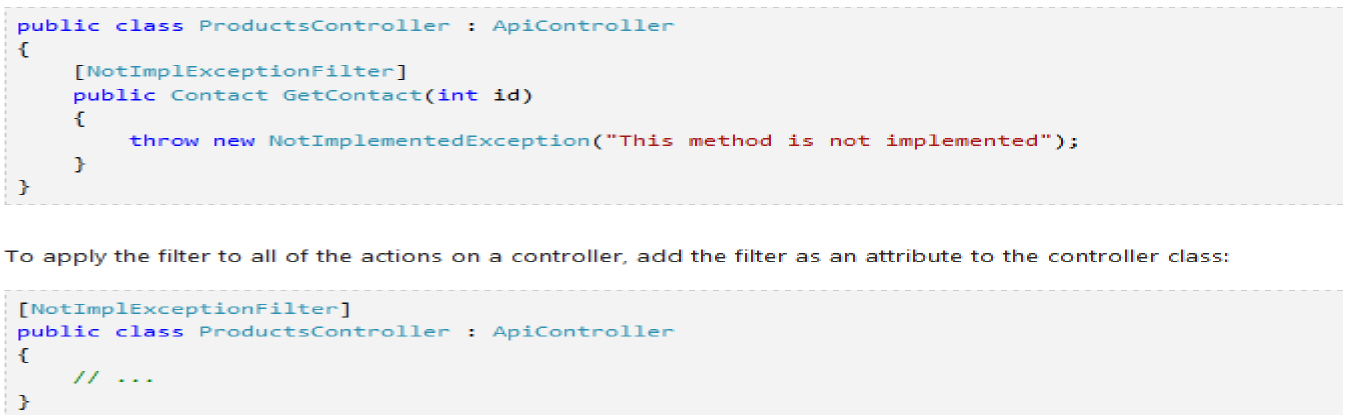
The **Response** property of the **HttpActionExecutedContext** object contains the HTTP response message that will be sent to the client.

Registering Exception Filters: There are several ways to register a Web API exception filter:

- By action
- By controller
- Globally

Exception Handling in ASP.NET Web API

To apply the filter to a specific action, add the filter as an attribute to the action:

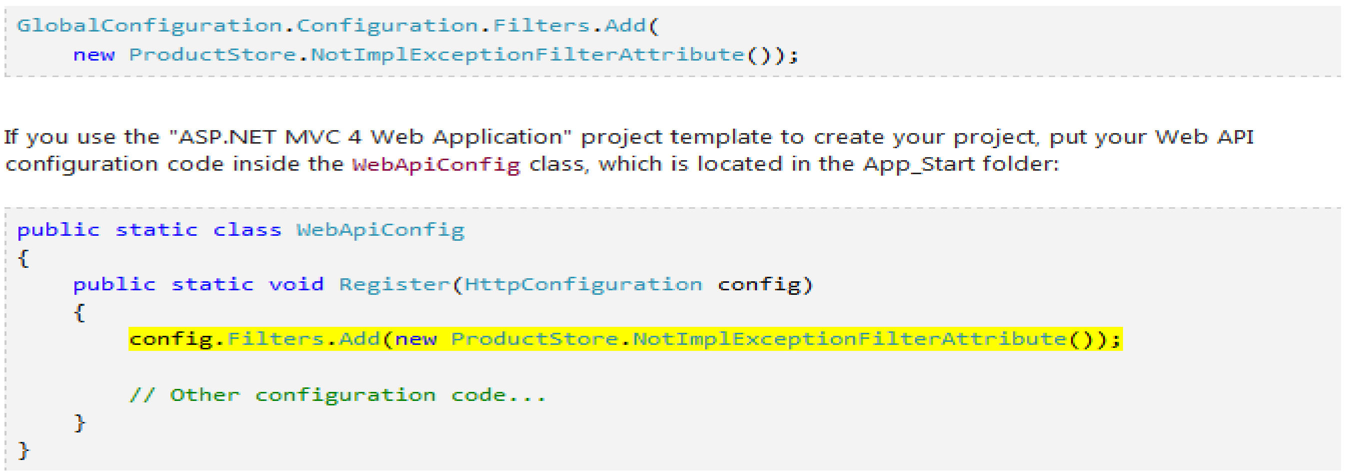


```
public class ProductsController : ApiController
{
    [NotImplExceptionFilter]
    public Contact GetContact(int id)
    {
        throw new NotImplementedException("This method is not implemented");
    }
}
```

To apply the filter to all of the actions on a controller, add the filter as an attribute to the controller class:

```
[NotImplExceptionFilter]
public class ProductsController : ApiController
{
    // ...
}
```

To apply the filter globally to all Web API controllers, add an instance of the filter to the **GlobalConfiguration.Configuration.Filters** collection. Exception filters in this collection apply to any Web API controller action.



```
GlobalConfiguration.Configuration.Filters.Add(
    new ProductStore.NotImplExceptionFilterAttribute());
```

If you use the "ASP.NET MVC 4 Web Application" project template to create your project, put your Web API configuration code inside the `WebApiConfig` class, which is located in the App_Start folder:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ProductStore.NotImplExceptionFilterAttribute());

        // Other configuration code...
    }
}
```

Exception Handling in ASP.NET Web API

HttpError

The **HttpError** object provides a consistent way to return error information in the response body. The following example shows how to return HTTP status code 404 (Not Found) with an **HttpError** in the response body:

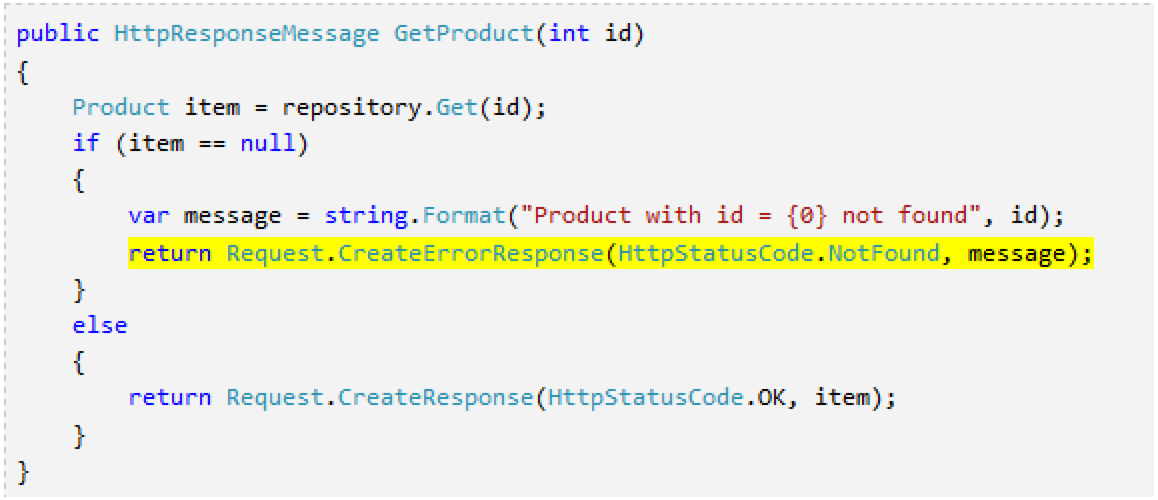
```
public HttpResponseMessage GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        HttpError err = new HttpError(message);
        return Request.CreateResponse(HttpStatusCode.NotFound, err);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK, item);
    }
}
```

In this example, if the method is successful, it returns the product in the HTTP response. But if the requested product is not found, the HTTP response contains an **HttpError** in the request body. The response might look like the following:

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Date: Thu, 09 Aug 2012 23:27:18 GMT
Content-Length: 51

{
  "Message": "Product with id = 12 not found"
}
```

Exception Handling in ASP.NET Web API

Notice that the **HttpError** was serialized to JSON in this example. One advantage of using **HttpError** is that it goes through the same content-negotiation and serialization process as any other strongly-typed model.
Instead of creating the **HttpError** object directly, you can use the **CreateErrorResponse** method:



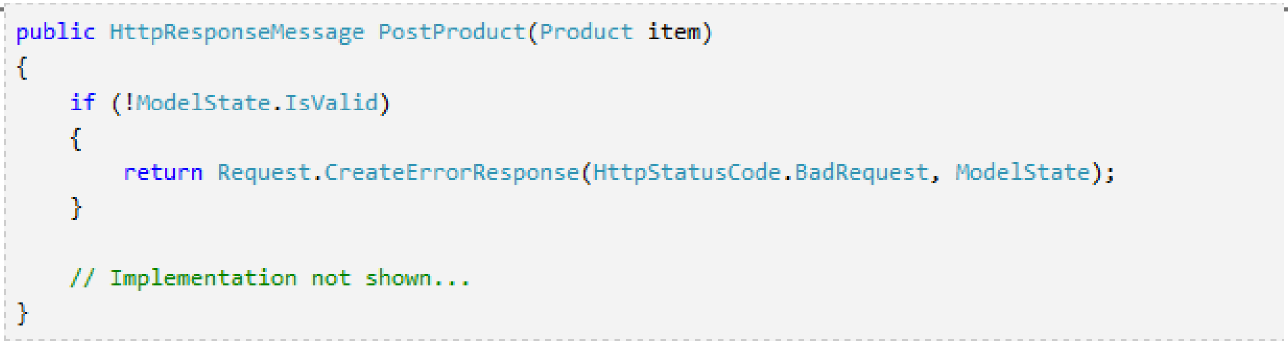
```
public HttpResponseMessage GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, message);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK, item);
    }
}
```

CreateErrorResponse is an extension method defined in the **System.Net.Http.HttpRequestMessageExtensions** class. Internally, **CreateErrorResponse** creates an **HttpError** instance and then creates an **HttpResponseMessage** that contains the **HttpError**.

HttpError and Model Validation

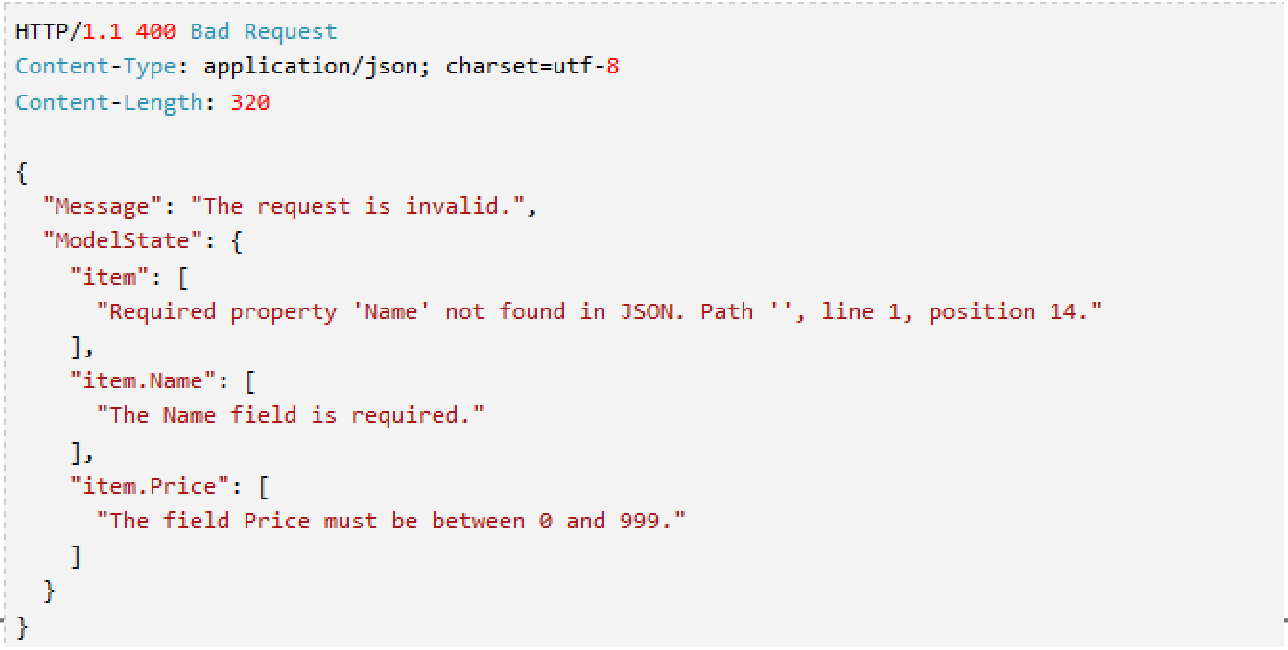
For model validation, you can pass the model state to **CreateErrorResponse**, to include the validation errors in the response:

Exception Handling in ASP.NET Web API



```
public HttpResponseMessage PostProduct(Product item)
{
    if (!ModelState.IsValid)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }

    // Implementation not shown...
}
```

This example might return the following response:

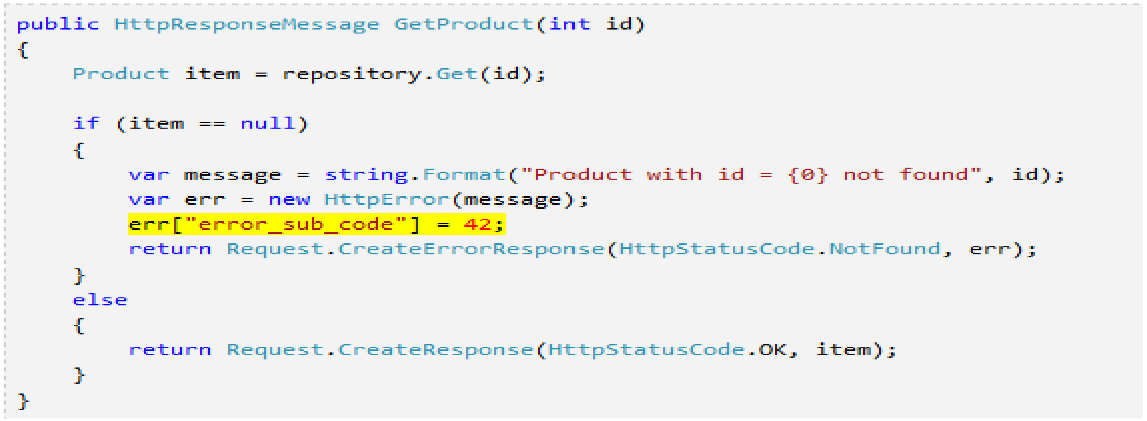


```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 320

{
  "Message": "The request is invalid.",
  "ModelState": {
    "item": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 14."
    ],
    "item.Name": [
      "The Name field is required."
    ],
    "item.Price": [
      "The field Price must be between 0 and 999."
    ]
  }
}
```

Exception Handling in ASP.NET Web API

Adding Custom Key-Values to HttpError

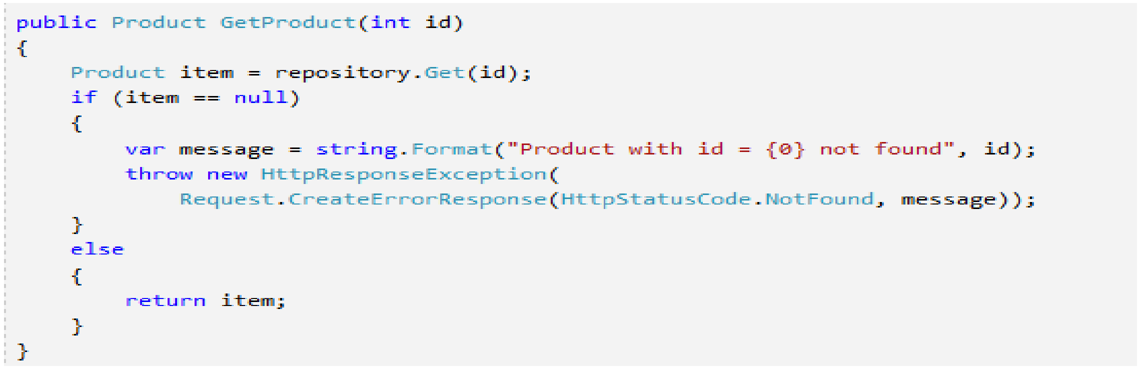
The **HttpError** class is actually a key-value collection (it derives from **Dictionary<string, object>**), so you can add your own key-value pairs:



```
public HttpResponseMessage GetProduct(int id)
{
    Product item = repository.Get(id);

    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        var err = new HttpError(message);
        err["error_sub_code"] = 42;
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, err);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK, item);
    }
}
```

Using HttpError with HttpResponseException

The previous examples return an **HttpResponseMessage** message from the controller action, but you can also use **HttpResponseException** to return an **HttpError**. This lets you return a strongly-typed model in the normal success case, while still returning **HttpError** if there is an error:



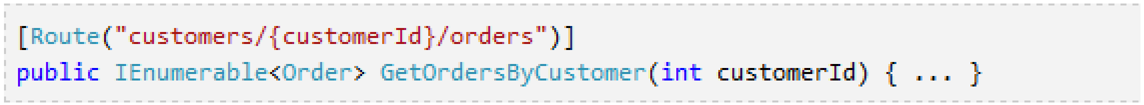
```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        throw new HttpResponseException(
            Request.CreateErrorResponse(HttpStatusCode.NotFound, message));
    }
    else
    {
        return item;
    }
}
```

Attribute Routing in Web API 2

- *Routing* is how Web API matches a URI to an action.
- Web API 2 supports a new type of routing, called *attribute routing*.
- As the name implies, attribute routing uses attributes to define routes.
- Attribute routing gives you more control over the URIs in your web API. For example, you can easily create URIs that describe hierarchies of resources.
- The earlier style of routing, called convention-based routing, is still fully supported. In fact, you can combine both techniques in the same project.

Why Attribute Routing?

- The first release of Web API used *convention-based* routing. In that type of routing, you define one or more route templates, which are basically parameterized strings.
- When the framework receives a request, it matches the URI against the route template.
- One advantage of convention-based routing is that templates are defined in a single place, and the routing rules are applied consistently across all controllers. Unfortunately, convention-based routing makes it hard to support certain URI patterns that are common in RESTful APIs. For example, resources often contain child resources: Customers have orders, movies have actors, books have authors, and so forth. It's natural to create URIs that reflect these relations: /customers/1/orders
- This type of URI is difficult to create using convention-based routing. Although it can be done, the results don't scale well if you have many controllers or resource types.
- With attribute routing, it's trivial to define a route for this URI. You simply add an attribute to the controller action:



```
[Route("customers/{customerId}/orders")]
public IEnumerable<Order> GetOrdersByCustomer(int customerId) { ... }
```

Here are some other patterns that attribute routing makes easy.

- **API versioning :** "/api/v1/products" would be routed to a different controller than "/api/v2/products".
- **Overloaded URI segments :** "1" is an order number, but "pending" maps to a collection. /Orders/1 , /Orders/pending
- **Multiple parameter types :**"1" is an order number, but "2013/06/16" specifies a date. /orders/1, /orders/2013/06/16

Enabling Attribute Routing in Web API 2

Enabling Attribute Routing

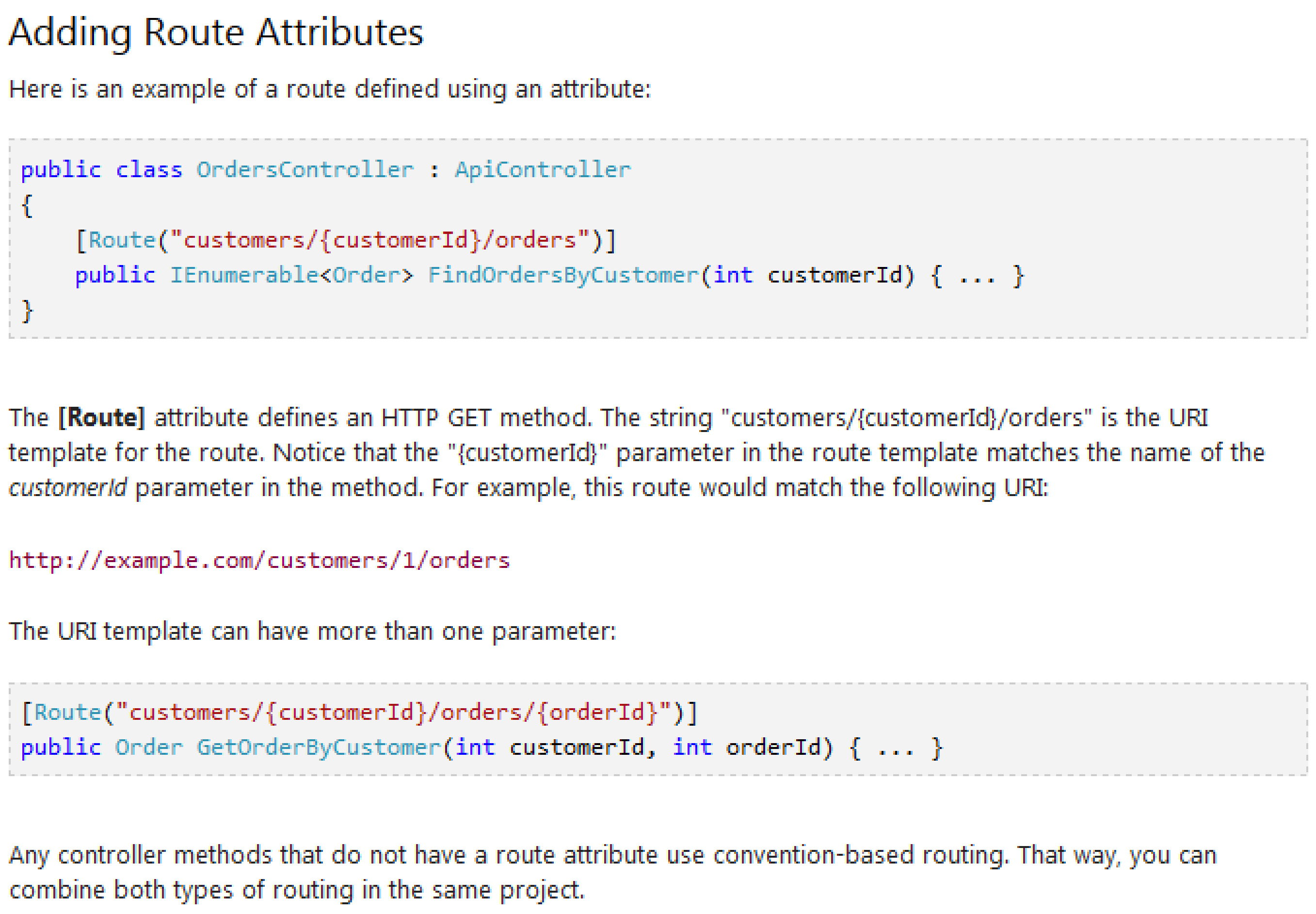
To enable attribute routing, call **MapHttpAttributeRoutes** during configuration. This extension method is defined in the **System.Web.Http.HttpConfigurationExtensions** class.

```
public static class \~ebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
    }
}
```

You can also combine attribute routing with convention-based routing. To define convention-based routes, call the **MapHttpRoute** method.

```
public static class \~ebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes()..:

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new {id = RouteParameter.Optional}
        )..:
    }
}
```

Adding Attribute Routing in Web API 2



Adding Route Attributes

Here is an example of a route defined using an attribute:

```
public class OrdersController : ApiController
{
    [Route("customers/{customerId}/orders")]
    public IEnumerable<Order> FindOrdersByCustomer(int customerId) { ... }
}
```

The **[Route]** attribute defines an HTTP GET method. The string "customers/{customerId}/orders" is the URI template for the route. Notice that the "{customerId}" parameter in the route template matches the name of the *customerId* parameter in the method. For example, this route would match the following URI:

```
http://example.com/customers/1/orders
```

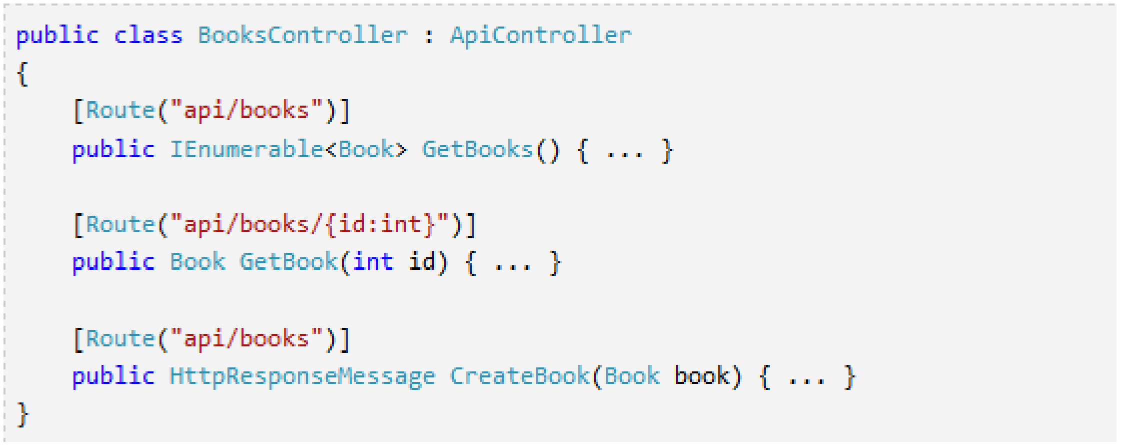
The URI template can have more than one parameter:

```
[Route("customers/{customerId}/orders/{orderId}")]
public Order GetOrderByCustomer(int customerId, int orderId) { ... }
```

Any controller methods that do not have a route attribute use convention-based routing. That way, you can combine both types of routing in the same project.

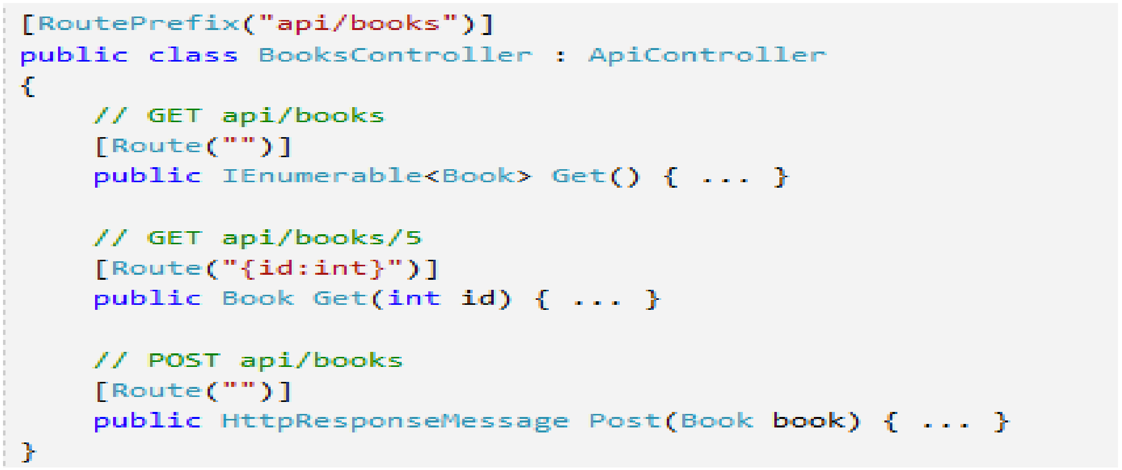
Route Prefixes

Often, the routes in a controller all start with the same prefix. For example:



```
public class BooksController : ApiController
{
    [Route("api/books")]
    public IEnumerable<Book> GetBooks() { ... }

    [Route("api/books/{id:int}")]
    public Book GetBook(int id) { ... }

    [Route("api/books")]
    public HttpResponseMessage CreateBook(Book book) { ... }
}
```

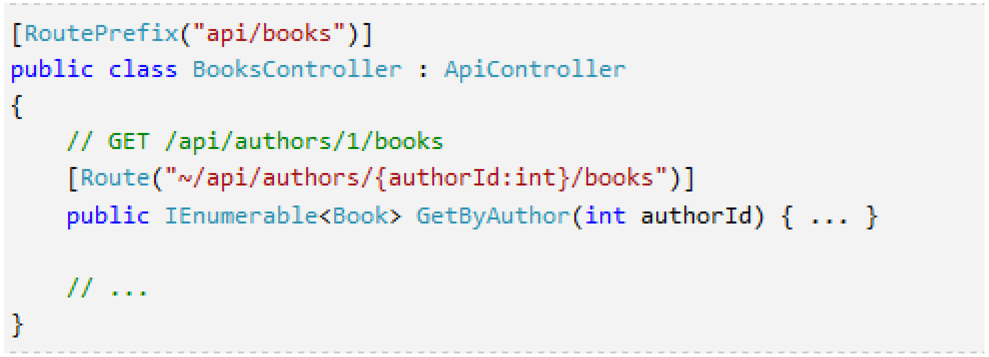
You can set a common prefix for an entire controller by using the **[RoutePrefix]** attribute:



```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET api/books
    [Route("")]
    public IEnumerable<Book> Get() { ... }

    // GET api/books/5
    [Route("{id:int}")]
    public Book Get(int id) { ... }

    // POST api/books
    [Route("")]
    public HttpResponseMessage Post(Book book) { ... }
}
```

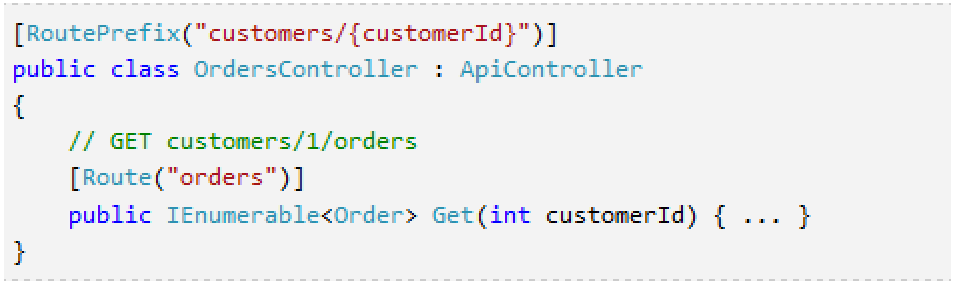
Route Prefixes

Use a tilde (~) on the method attribute to override the route prefix:



```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET /api/authors/1/books
    [Route("~/api/authors/{authorId:int}/books")]
    public IEnumerable<Book> GetByAuthor(int authorId) { ... }

    // ...
}
```

The route prefix can include parameters:



```
[RoutePrefix("customers/{customerId}")]
public class OrdersController : ApiController
{
    // GET customers/1/orders
    [Route("orders")]
    public IEnumerable<Order> Get(int customerId) { ... }
}
```

Route Constraints

Route constraints let you restrict how the parameters in the route template are matched. The general syntax is "{parameter:constraint}". For example:

```
[Route("users/{id:int}"]
public User GetUserById(int id) { ... }

[Route("users/{name}"]
public User GetUserByName(string name) { ... }
```

Here, the first route will only be selected if the "id" segment of the URI is an integer. Otherwise, the second route will be chosen. The following table lists the constraints that are supported.

Constraint	Description	Example
alpha	Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z)	{x:alpha}
bool	Matches a Boolean value.	{x:bool}
datetime	Matches a **DateTime** value.	{x:datetime}
decimal	Matches a decimal value.	{x:decimal}
double	Matches a 64-bit floating-point value.	{x:double}
float	Matches a 32-bit floating-point value.	{x:float}
guid	Matches a GUID value.	{x:guid}
int	Matches a 32-bit integer value.	{x:int}

length	Matches a string with the specified length or within a specified range of lengths.	{x:length(6)}
		{x:length(1,20)}

long	Matches a 64-bit integer value.	{x:long}
max	Matches an integer with a maximum value.	{x:max(10)}
maxlength	Matches a string with a maximum length.	{x:maxlength(10)}
min	Matches an integer with a minimum value.	{x:min(10)}
minlength	Matches a string with a minimum length.	{x:minlength(10)}
range	Matches an integer within a range of values.	{x:range(10,50)}
regex	Matches a regular expression.	{x:(^\d{3}-\d{3}-\d{4}$)}

Custom Route Constraints

- You can create custom route constraints by implementing the **IHttpRouteConstraint** interface. For example, the following constraint restricts a parameter to a non-zero integer value.

- Code below shows how to register the constraint.
- To apply the constraint in your routes

```
[Route("{id:nonzero}")]
public HttpResponseMessage GetNonZero(int id) { ... }
```

- You can also replace the entire **DefaultInlineConstraintResolver** class by implementing the **IInlineConstraintResolver** interface. Doing so will replace all of the built-in constraints, unless your implementation of **IInlineConstraintResolver** specifically adds them.

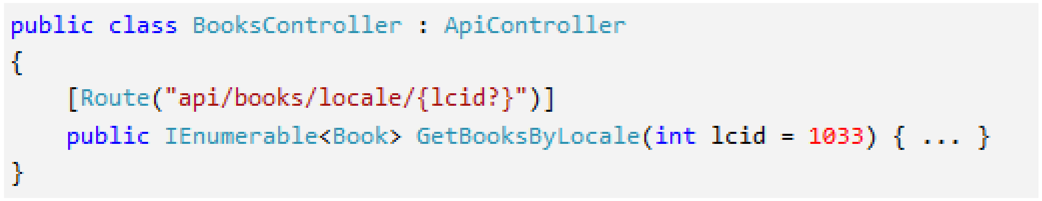
```
public class NonZeroConstraint : IHttpRouteConstraint
{
    public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,
        IDictionary<string, object> values, HttpRouteDirection routeDirection)
    {
        object value;
        if (values.TryGetValue(parameterName, out value) && value != null)
        {
            long longValue;
            if (value is long)
            {
                longValue = (long)value;
                return longValue != 0;
            }

            string valueString = Convert.ToString(value, CultureInfo.InvariantCulture);
            if (Int64.TryParse(valueString, NumberStyles.Integer,
                CultureInfo.InvariantCulture, out longValue))
            {
                return longValue != 0;
            }
        }
        return false;
    }
}
```

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var constraintResolver = new DefaultInlineConstraintResolver();
        constraintResolver.ConstraintMap.Add("nonzero", typeof(NonZeroConstraint));

        config.MapHttpAttributeRoutes(constraintResolver);
    }
}
```

Abhishek Sharma

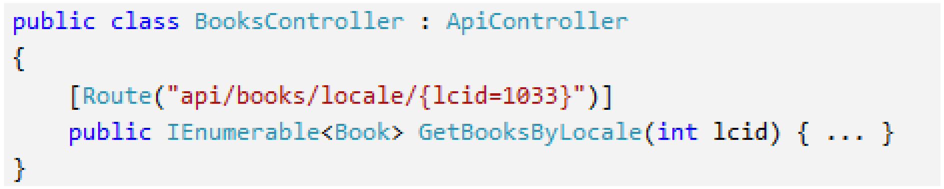
Optional URI Parameters and Default Values

- You can make a URI parameter optional by adding a question mark to the route parameter. If a route parameter is optional, you must define a default value for the method parameter.



```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid?}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid = 1033) { ... }
}
```

- In this example, /api/books/locale/1033 and /api/books/locale return the same resource.
- Alternatively, you can specifya default value inside the route template, as follows:



```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid=1033}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid) { ... }
}
```

This is almost the same as the previous example, but there is a slight difference of behavior when the default value is applied.

- In the first example ("{lcid?}"), the default value of 1033 is assigned directly to the method parameter, so the parameter will have this exact value.
- In the second example ("{lcid=1033}"), the default value of "1033" goes through the model-binding process. The default model-binder will convert "1033" to the numeric value 1033. However, you could plug in a custom model binder, which might do something different.

(In most cases, unless you have custom model binders in your pipeline, the two forms will be equivalent.)

Route Names

In Web API, every route has a name. Route names are useful for generating links, so that you can include a link in an HTTP response.
To specify the route name, set the **RouteName** property on the attribute. The following example shows how to set the route name, and also how to use the route name when generating a link.

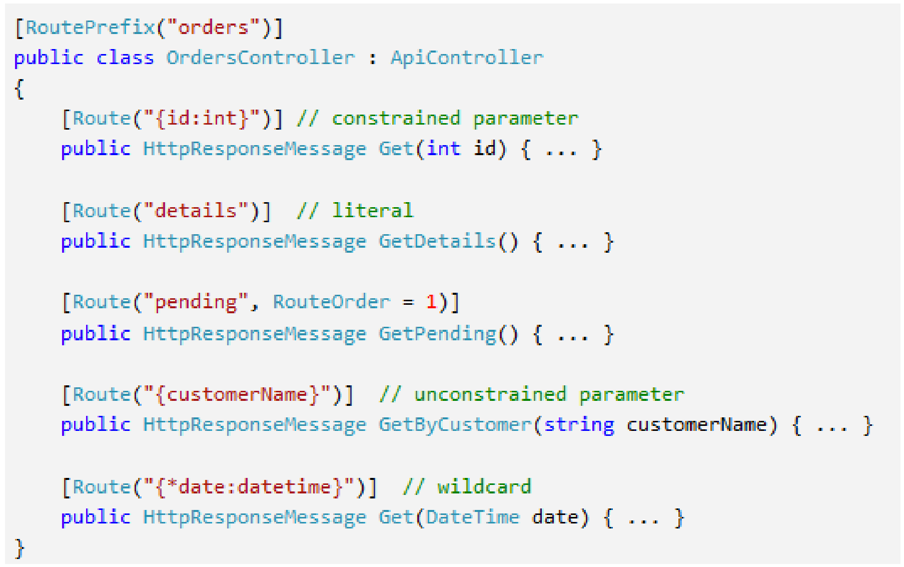
```
public class BooksController : ApiController
{
    [Route("api/books/{id}", RouteName="GetBookById")]
    public BookDto GetBook(int id)
    {
        // Implementation not shown...
    }

    [Route("api/books")]
    public HttpResponseMessage Post(Book book)
    {
        // Validate and add book to database (not shown)

        var response = Request.CreateResponse(HttpStatusCode.Created);

        // Generate a link to the new book and set the Location header in the response.
        string uri = Url.Link("GetBookById", new { id = book.BookId });
        response.Headers.Location = new Uri(uri);
        return response;
    }
}
```

If you don't set the **RouteName** property Web API generates the name. The default route name is "ControllerName.ActionName". In the previous example, the default route name would be "Books.GetBook" for the GetBook method. If the controller has multiple attribute routes with the same action name, a suffix is added; for example, "Books.GetBook1" and "Books.GetBook2".

Route Order

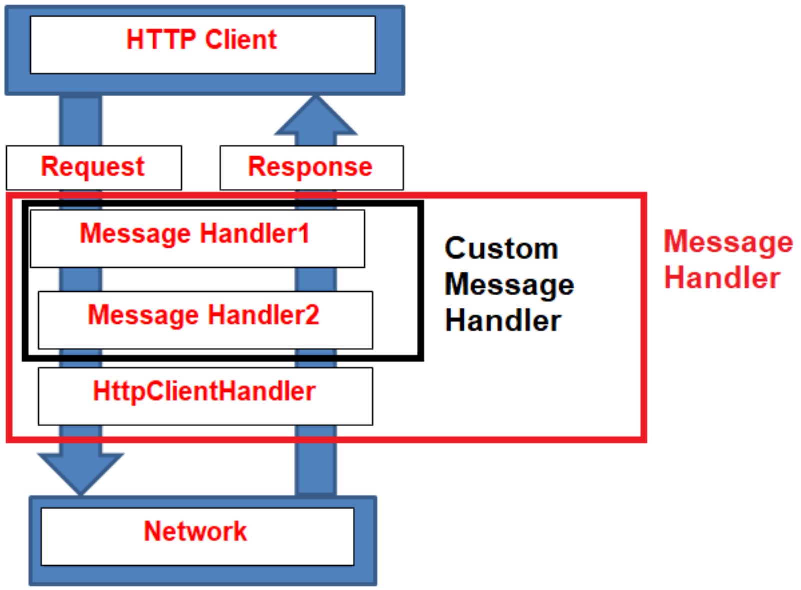
- When the framework tries to match a URI with a route, it evaluates the routes in a particular order. To specify the order, set the **RouteOrder** property on the route attribute. Lower values are evaluated first. The default order value is zero.

- Here is how the total ordering is determined:
    1. Compare the **RouteOrder** property of the route attribute.
    2. Look at each URI segment in the route template. For each segment, order as follows:
        - Literal segments.
        - Route parameters with constraints.
        - Route parameters without constraints.
        - Wildcard parameter segments with constraints.
        - Wildcard parameter segments without constraints.
    3. In the case of a tie, routes are ordered by a case-insensitive ordinal string comparison (OrdinalIgnoreCase) of the route template.



```
[RoutePrefix("orders")]
public class OrdersController : ApiController
{
    [Route("{id:int}")] // constrained parameter
    public HttpResponseMessage Get(int id) { ... }

    [Route("details")]  // literal
    public HttpResponseMessage GetDetails() { ... }

    [Route("pending", RouteOrder = 1)]
    public HttpResponseMessage GetPending() { ... }

    [Route("{customerName}")]  // unconstrained parameter
    public HttpResponseMessage GetByCustomer(string customerName) { ... }

    [Route("{*date:datetime}")]  // wildcard
    public HttpResponseMessage Get(DateTime date) { ... }
}
```

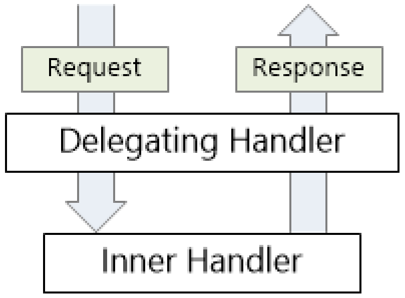
- Suppose you define the following controller , shown right
- These routes are ordered as follows.
    - orders/details
    - orders/{id}
    - orders/{customerName}
    - orders/{*date}
    - orders/pending



Working with HTTP

HTTP Message Handlers

- A message handler is a class that receives an HTTP request and returns an HTTP response. Message handlers derive from the abstract **HttpMessageHandler** class.
- Typically, a series of message handlers are chained together. The first handler receives an HTTP request, does some processing, and gives the request to the next handler. At some point, the response is created and goes back up the chain. This pattern is called a delegating handler.



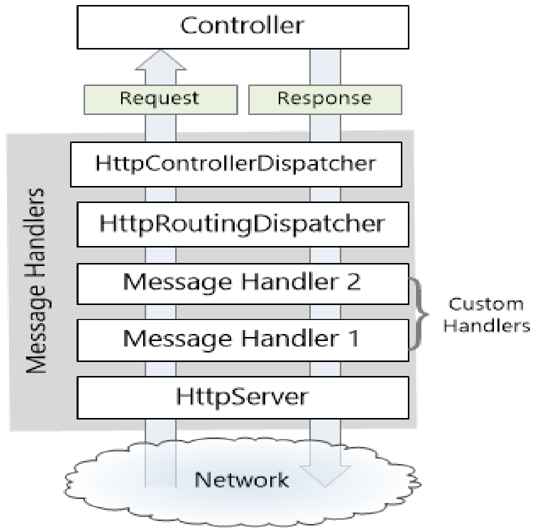
Server-Side Message Handlers

- **HttpServer** gets the request from the host.
- **HttpRoutingDispatcher** dispatches the request based on the route.
- **HttpControllerDispatcher** sends the request to a Web API controller.

You can add custom handlers to the pipeline. Message handlers are good for cross-cutting concerns that operate at the level of HTTP messages (rather than controller actions). For example, a message handler might:

- Read or modify request headers.
- Add a response header to responses.
- Validate requests before they reach the controller.

This diagram shows two custom handlers inserted into the pipeline

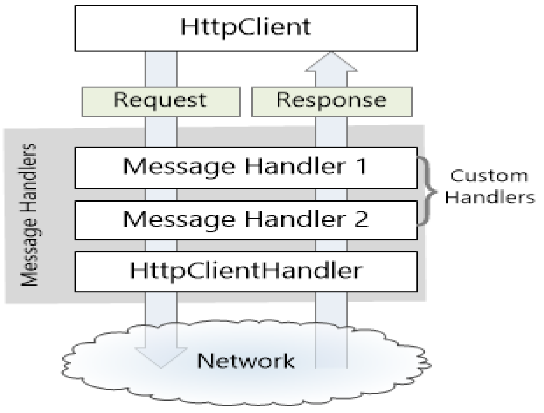


Server-Side Handlers

HTTP Message Handlers

Client – Side Handlers

On the client side, the **HttpClient** class uses a message handler to process requests. The default handler is **HttpClientHandler**, which sends the request over the network and gets the response from the server. You can insert custom message handlers into the client pipeline:



Client-Side Handlers

Custom Message Handlers

To write a custom message handler, derive from **System.Net.Http.DelegatingHandler** and override the **SendAsync** method. This method has the following signature:

```
Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken);
```

The method takes an **HttpRequestMessage** as input and asynchronously returns an **HttpResponseMessage**. A typical implementation does the following:

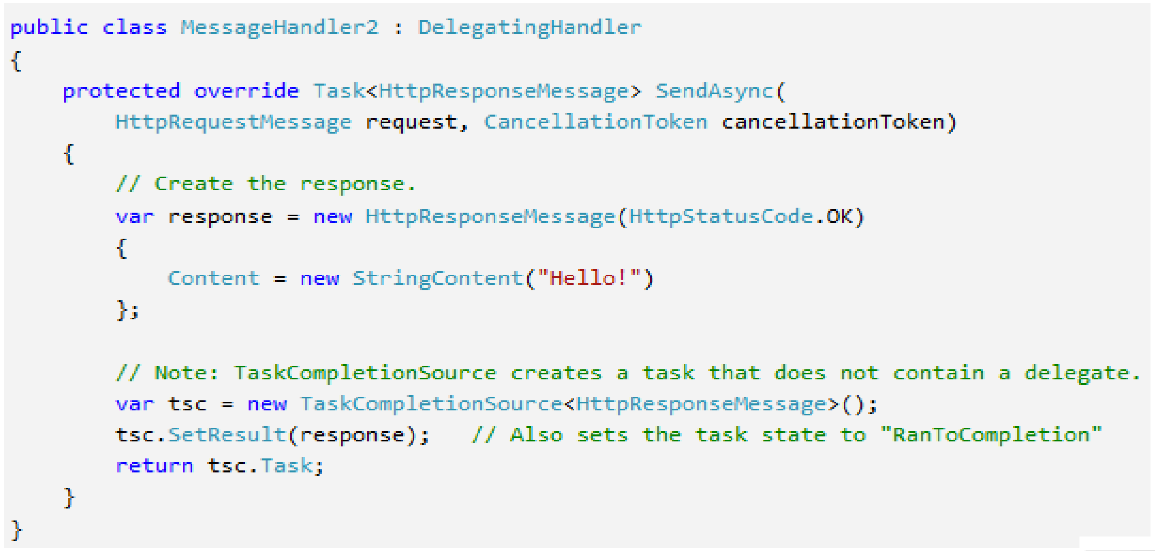
1.Process the request message.
2.Call base.SendAsync to send the request to the inner handler.
3.The inner handler returns a response message. (This step is asynchronous.)
4.Process the response and return it to he caller.

Here is a trivial example:

```
public class MessageHandler1 : DelegatingHandler
{
    protected async override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        Debug.WriteLine("Process request");
        // Call the inner handler.
        var response = await base.SendAsync(request, cancellationToken);
        Debug.WriteLine("Process response");
        return response;
    }
}
```

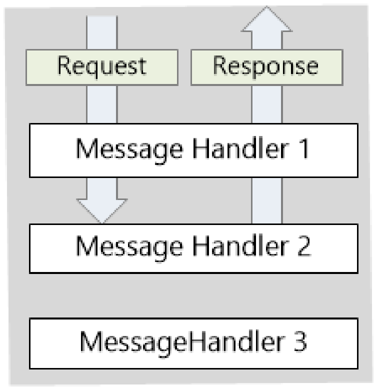
HTTP Message Handlers

- The call to baseSendAsync is asynchronous. If the handler does any work after this call, use the **await** keyword .
- A delegating handler can also skip the inner handler and directly create the response



```
public class MessageHandler2 : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        // Create the response.
        var response = new HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = new StringContent("Hello!")
        };

        // Note: TaskCompletionSource creates a task that does not contain a delegate.
        var tsc = new TaskCompletionSource<HttpResponseMessage>();
        tsc.SetResult(response);   // Also sets the task state to "RanToCompletion"
        return tsc.Task;
    }
}
```

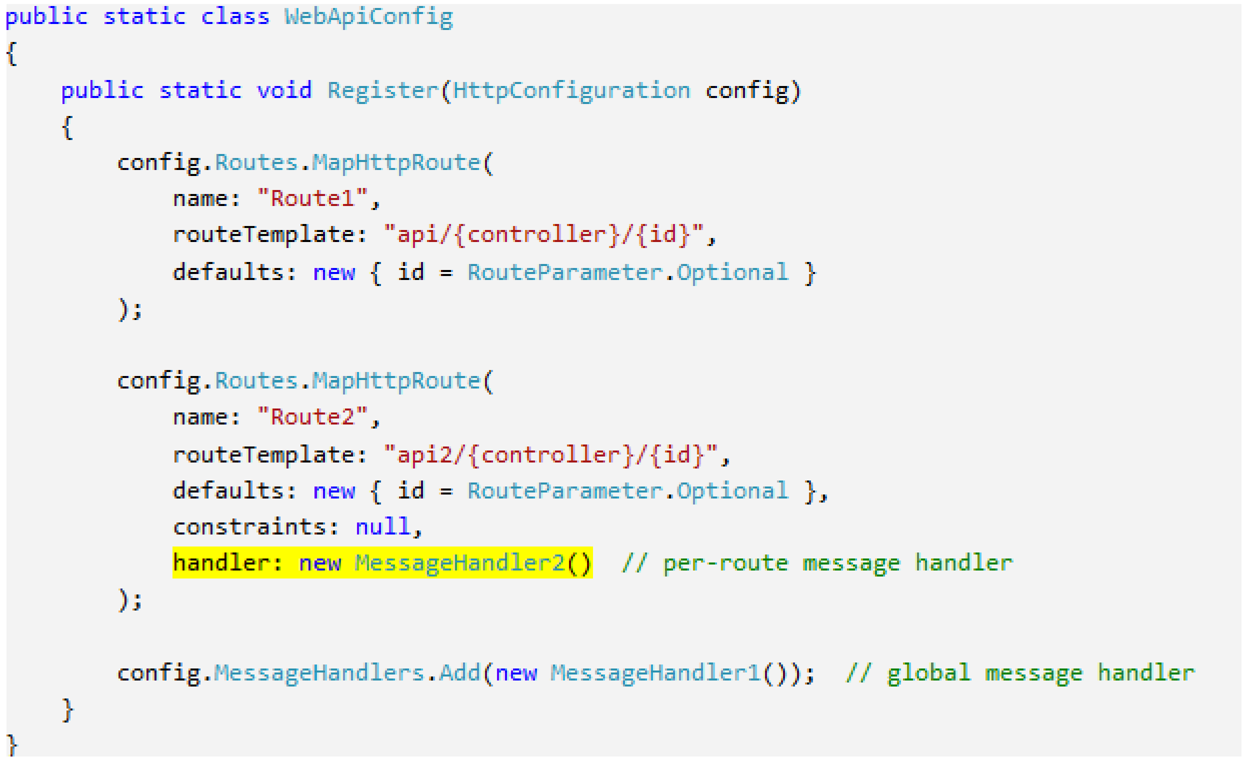
If a delegating handler creates the response without calling base.SendAsync, the request skips the rest of the pipeline. This can be useful for a handler that validates the request (creating an error response).



HTTP Message Handlers

Per-Route Message Handlers

Handlers in the **HttpConfiguration.MessageHandlers** collection apply globally. Alternatively, you can add a message handler to a specific route when you define the route:

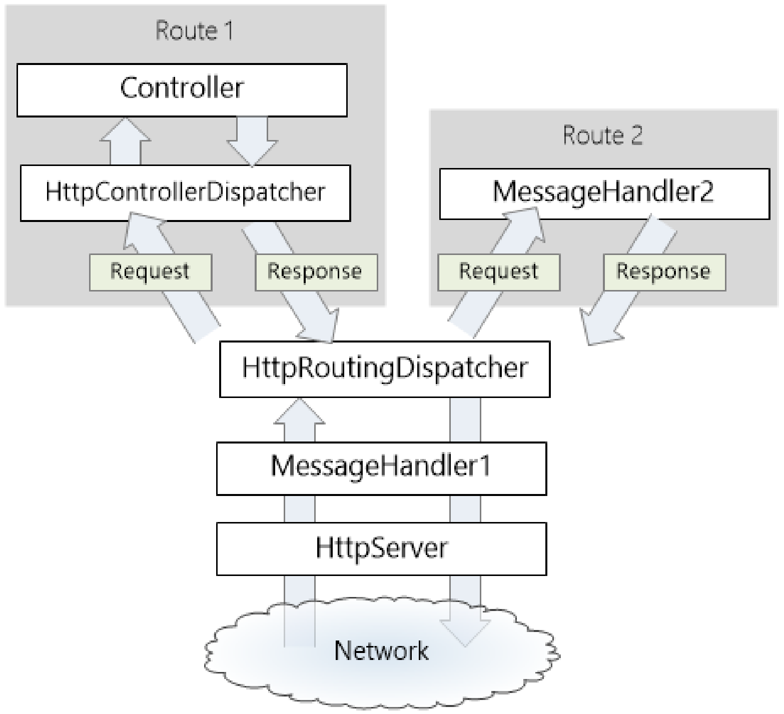


```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "Route1",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        config.Routes.MapHttpRoute(
            name: "Route2",
            routeTemplate: "api2/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional },
            constraints: null,
            handler: new MessageHandler2()  // per-route message handler
        );

        config.MessageHandlers.Add(new MessageHandler1());  // global message handler
    }
}
```

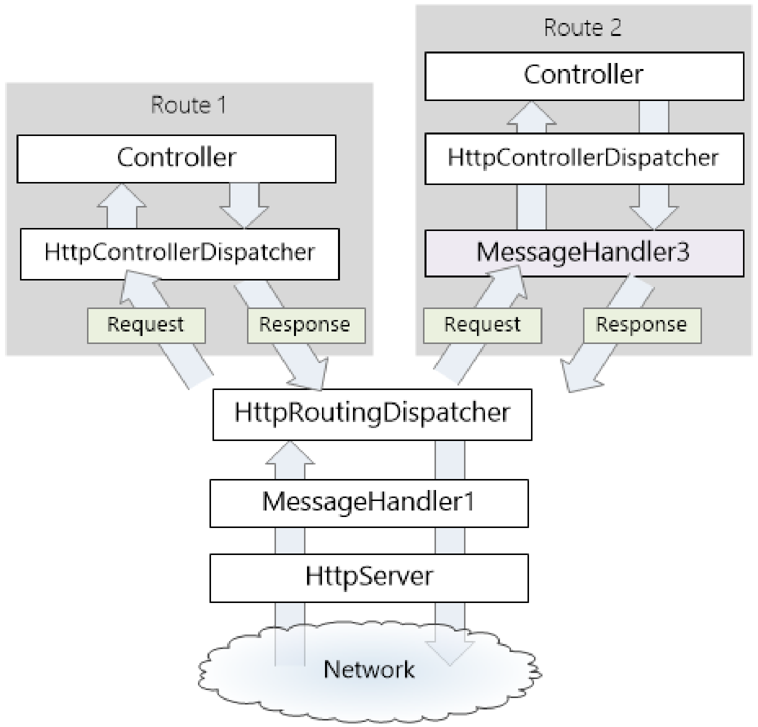
if the request URI matches "Rote2", the request is dispatched to MessageHandler2. The following diagram shows the pipeline for these two routes:

HTTP Message Handlers

MessageHandler2 replaces the default **HttpCntrollerDispatcher**. In this example, MessageHandler2 creates the response, and requests that match "Route2" never go to a controller. This lets you replace the entire Web API controller mechanism with your own custom endpoint.



Alternatively, a per-route message handler can delegate to **HttpControllerDispatcher**, which then dispatches to a controller.



HTTP Cookies

- A cookie is a piece of data that a server sends in the HTTP response.
- The client (optionally) stores the cookie and returns it on subsequent requests.
- This allows the client and server to share state.
- To set a cookie, the server includes a Set-Cookie header in the response.
- The format of a cookie is a name-value pair, with optional attributes.

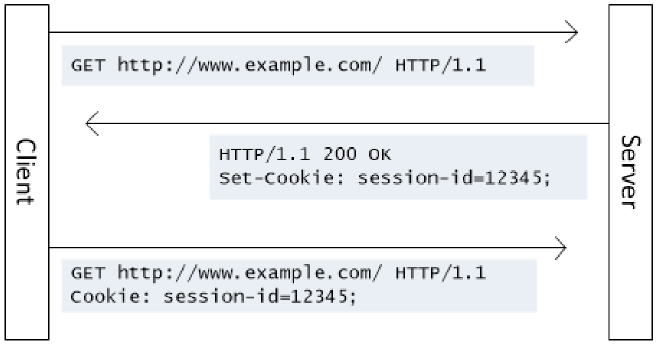
```
Set-Cookie: session-id=1234567
```

Here is an example with attributes:

```
Set-Cookie: session-id=1234567; max-age=86400; domain=example.com; path=/;
```

- To return a cookie to the server, the client includes a Cookie header in later requests.

```
Cookie: session-id=1234567
```



```
GET http://www.example.com/ HTTP/1.1
```

```
HTTP/1.1 200 OK
Set-Cookie: session-id=12345;
```

```
GET http://www.example.com/ HTTP/1.1
Cookie: session-id=12345;
```

- An HTTP response can include multiple Set-Cookie headers.

```
Set-Cookie: session-token=abcdef;
Set-Cookie: session-id=1234567;
```

- The client returns multiple cookies using a single Cookie header.

```
Cookie: session-id=1234567; session-token=abcdef;
```

HTTP Cookies

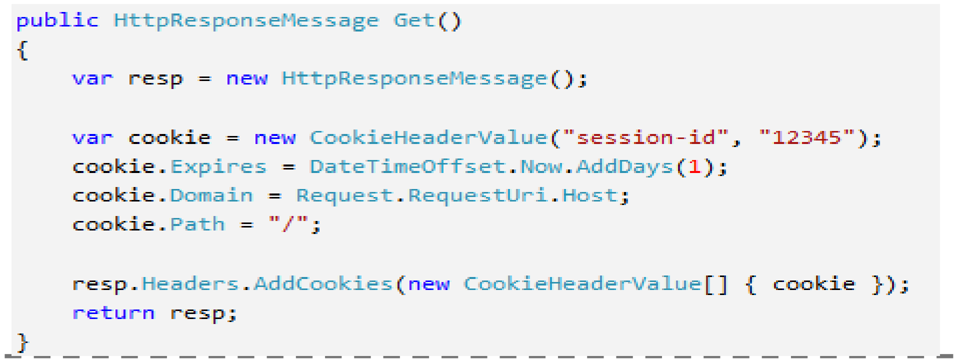
The scope and duration of a cookie are controlled by following attributes in the Set-Cookie header:

- **Domain**: Tells the client which domain should receive the cookie. For example, if the domain is "example.com", the client returns the cookie to every subdomain of example.com. If not specified, the domain is the origin server.
- **Path**: Restricts the cookie to the specified path within the domain. If not specified, the path of the request URI is used.
- **Expires**: Sets an expiration date for the cookie. The client deletes the cookie when it expires.
- **Max-Age**: Sets the maximum age for the cookie. The client deletes the cookie when it reaches the maximum age.

If both Expires and Max-Age are set, Max-Age takes precedence. If neither is set, the client deletes the cookie when the current session ends. (The exact meaning of "session" is determined by the user-agent.)
However, be aware that clients may ignore cookies. For example, a user might disable cookies for privacy reasons. Clients may delete cookies before they expire, or limit the number of cookies stored. For privacy reasons, clients often reject "third party" cookies, where the domain does not match the origin server. In short, the server should not rely on getting back the cookies that it sets.

Cookies in Web API

- To add a cookie to an HTTP response, create a **CookieHeaderValue** instance that represents the cookie.
- Then call the **AddCookies** extension method, which is defined in the **System.Net.Http. HttpResponseHeadersExtensions** class, to add the cookie, e.g. the following code adds a cookie within a controller action:



```
public HttpResponseMessage Get()
{
    var resp = new HttpResponseMessage();

    var cookie = new CookieHeaderValue("session-id", "12345");
    cookie.Expires = DateTimeOffset.Now.AddDays(1);
    cookie.Domain = Request.RequestUri.Host;
    cookie.Path = "/";

    resp.Headers.AddCookies(new CookieHeaderValue[] { cookie });
    return resp;
}
```

HTTP Cookies

AddCookies takes an array of **CookieHeaderValue** instances. To extract the cookies from a client request, call the **GetCookies** method:

```
string sessionId = "";

CookieHeaderValue cookie = Request.Headers.GetCookies("session-id").FirstOrDefault();
if (cookie != null)
{
    sessionId = cookie["session-id"].Value;
}
```

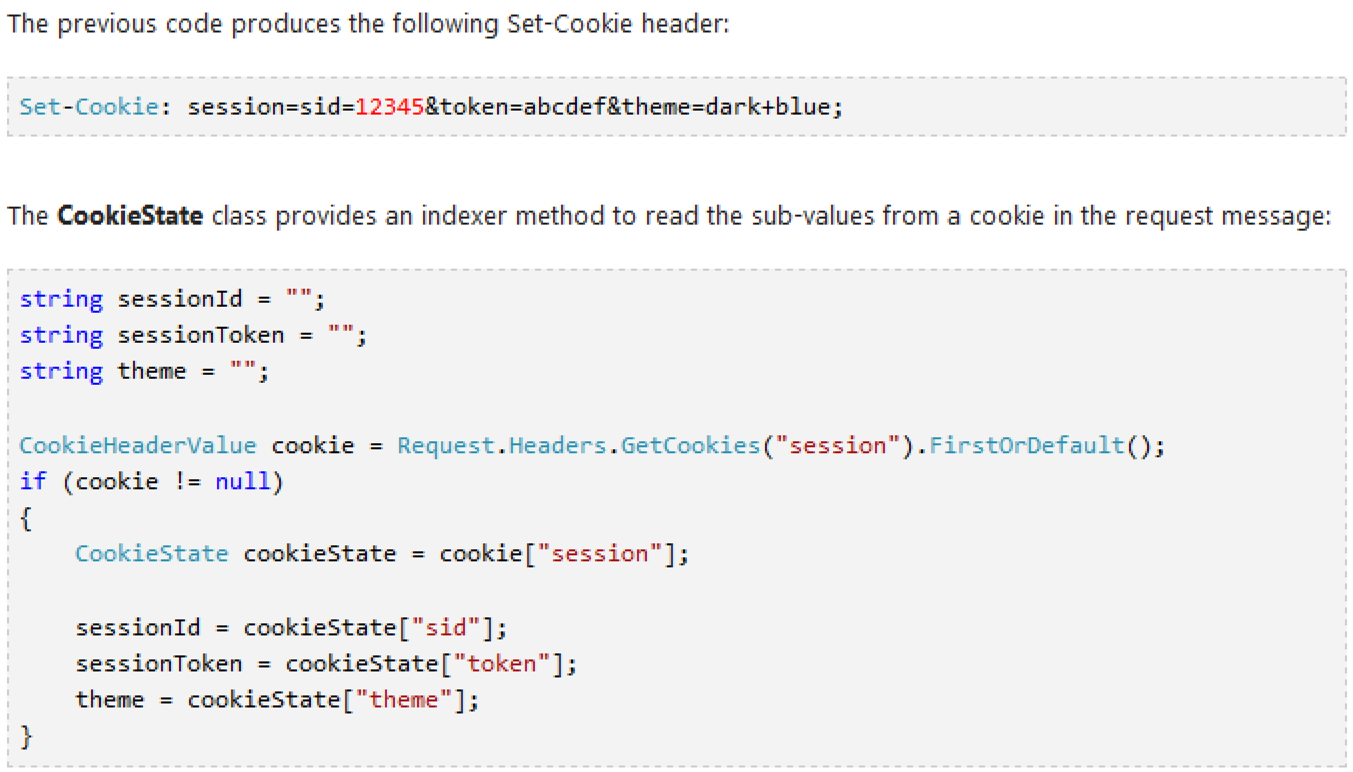
A **CookieHeaderValue** contains a collection of **CookieState** instances. Each **CookieState** represents one cookie. Use the indexer method to get a **CookieState** by name, as shown.

Structured Cookie Data

- Many browsers limit how many cookies they will store—both the total number, and the number per domain.
- It can be useful to put structured data into a single cookie, instead of setting multiple cookies.
- Using the **CookieHeaderValue** class, you can pass a list of name-value pairs for the cookie data. These name-value pairs are encoded as URL-encoded form data in the Set-Cookie header:

```
var resp = new HttpResponseMessage();

var nv = new NameValueCollection();
nv["sid"] = "12345";
nv["token"] = "abcdef";
nv["theme"] = "dark blue";
var cookie = new CookieHeaderValue("session", nv);

resp.Headers.AddCookies(new CookieHeaderValue[] { cookie });
```

HTTP Cookies



The previous code produces the following Set-Cookie header:

```
Set-Cookie: session=sid=12345&token=abcdef&theme=dark+blue;
```

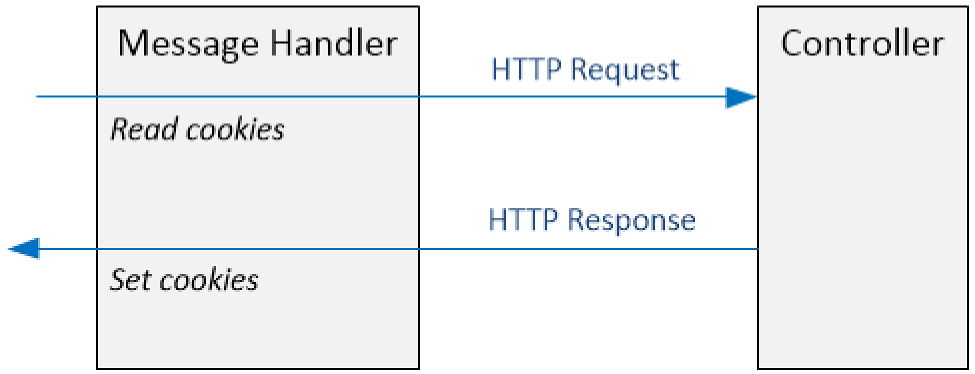
The **CookieState** class provides an indexer method to read the sub-values from a cookie in the request message:

```
string sessionId = "";
string sessionToken = "";
string theme = "";

CookieHeaderValue cookie = Request.Headers.GetCookies("session").FirstOrDefault();
if (cookie != null)
{
    CookieState cookieState = cookie["session"];

    sessionId = cookieState["sid"];
    sessionToken = cookieState["token"];
    theme = cookieState["theme"];
}
```

Set and Retrieve Cookies in a Message Handler

The previous examples showed how to use cookies from within a Web API controller. Another option is to use message handlers. Message handlers are invoked earlier in the pipeline than controllers. A message handler can read cookies from the request before the request reaches the controller, or add cookies to the response after the controller generates the response.

HTTP Cookies



The following code shows a message handler for creating session IDs. The session ID is stored in a cookie. The handler checks the request for the session cookie. If the request does not include the cookie, the handler generates a new session ID. In either case, the handler stores the session ID in the **HttpRequestMessage.Properties** property bag. It also adds the session cookie to the HTTP response.

Formats and Model Binding

Media Formatters

Internet Media Types

A media type, also called a MIME type, identifies the format of a piece of data. In HTTP, media types describe the format of the message body. A media type consists of two strings, a type and a subtype. For example:

- text/html
- image/png
- application/json

- When an HTTP message contains an entity-body, the Content-Type header specifies the format of the message body. This tells the receiver how to parse the contents of the message body. For example, if an HTTP response contains a PNG image, the response might have the following headers.

```
HTTP/1.1 200 OK
Content-Length: 95267
Content-Type: image/png
```

- When the client sends a request message, it can include an Accept header. The Accept header tells the server which media type(s) the client wants from the server. For example:

```
Accept: text/html,application/xhtml+xml,application/xml
```

- This header tells the server that the client wants either HTML, XHTML, or XML

In Web API, the media type determines how Web API serializes and deserializes the HTTP message body. There is built-in support for XML, JSON, and form-urlencoded data, and you can support additional media types by writing a *media formatter*.

To create a media formatter, derive from one of these classes:

- **MediaTypeFormatter**. This class uses asynchronous read and write methods.
- **BufferedMediaTypeFormatter**. This class derives from **MediaTypeFormatter** but wraps the asynchronous read/write methods inside sychronous methods. No asynchronous code, but it also means the calling thread can block during I/O.

Creating a Media Formatter

- The following example shows a media type formatter that can serialize a Product object to a comma-separated values (CSV) format. Here is the definition of the Product object:

```
namespace ProductStore.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

- To implement a CSV formatter, define a class that derives from **BufferedMediaTypeFormater**:

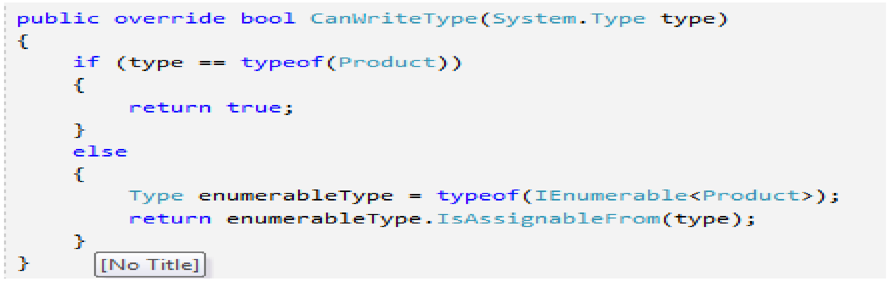
```
namespace ProductStore.Formatters
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Net.Http.Formatting;
    using System.Net.Http.Headers;
    using ProductStore.Models;

    public class ProductCsvFormatter : BufferedMediaTypeFormatter
    {
    }
}
```

- In the constructor, add the media types that the formatter supports. In this example, the formatter supports a single media type, "text/csv":

```
public ProductCsvFormatter()
{
    // Add the supported media type.
    SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
}
```

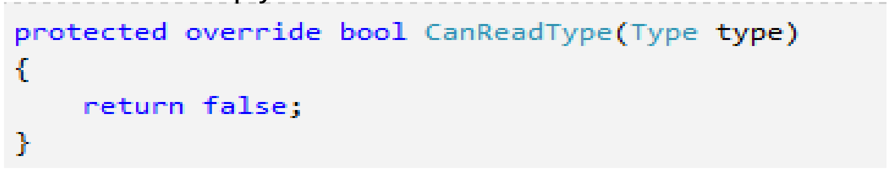
Creating a Media Formatter

- Override the **CanWriteType** method to indicate which types the formatter can serialize:



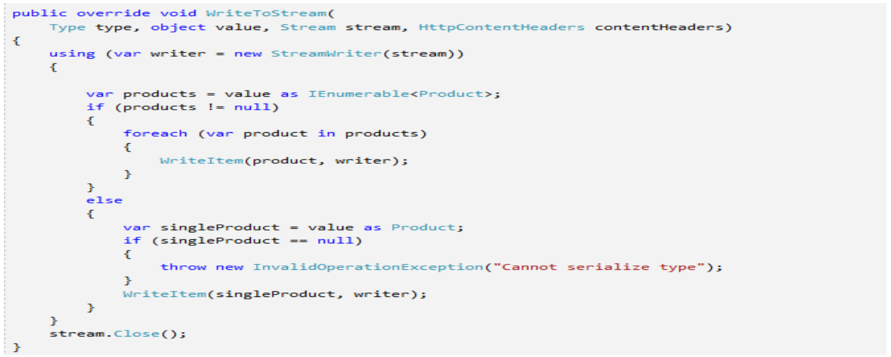
```
public override bool CanWriteType(System.Type type)
{
    if (type == typeof(Product))
    {
        return true;
    }
    else
    {
        Type enumerableType = typeof(IEnumerable<Product>);
        return enumerableType.IsAssignableFrom(type);
    }
}       [No Title]
```

- The formatter can serialize single Product objects as well as collections of Product objects. Similarly override the **CanReadType** method to indicate which types the formatter can deserialize. In this example, the formatter does not support deserialization, so the method simply returns **false**.



```
protected override bool CanReadType(Type type)
{
    return false;
}
```

- Override the **WriteToStream** method. This method serializes a type by writing it to a stream. If your formatter supports deserialization, also override the **ReadFromStream** method.



```
public override void WriteToStream(
    Type type, object value, Stream stream, HttpContentHeaders contentHeaders)
{
    using (var writer = new StreamWriter(stream))
    {

        var products = value as IEnumerable<Product>;
        if (products != null)
        {
            foreach (var product in products)
            {
                WriteItem(product, writer);
            }
        }
        else
        {
            var singleProduct = value as Product;
            if (singleProduct == null)
            {
                throw new InvalidOperationException("Cannot serialize type");
            }
            WriteItem(singleProduct, writer);
        }
    }
    stream.Close();
}
```

Adding a Media Formatter

- To add a media type formatter to the Web API pipeline, use the **Formatters** property on the **HttpConfiguration** object**.**

```
public static void ConfigureApis(HttpConfiguration config)
{
    config.Formatters.Add(new ProductCsvFormatter());
}
```

- For ASP.NET hosting, add this function to the Global.asax file and call it from the **Application_Start** method.

```
protected void Application_Start()
{
    ConfigureApis(GlobalConfiguration.Configuration);

    // ...
}
```

Now if a client specifies "text/csv" in the Accept header, the server will return the data in CSV format.
The following example uses **HttpClient** to get the CSV data and write it to a file:

```
HttpClient client = new HttpClient();

// Add the Accept header
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("text/csv"));

// Get the result and write it to a file.
// (Port 9000 is just an example port number.)
string result = client.GetStringAsync("http://localhost:9000/api/product/").Result;
System.IO.File.WriteAllText("products.csv", result);
```

JSON and XML Serialization in ASP.NET Web API

In ASP.NET Web API, a *media-type formatter* is an object that can:

- Read CLR objects from an HTTP message body
- Write CLR objects into an HTTP message body

Web API provides media-type formatters for both JSON and XML. The framework inserts these formatters into the pipeline by default. Clients can request either JSON or XML in the Accept header of the HTTP request.

JSON Media-Type Formatter

- JSON formatting is provided by the **JsonMediaTypeFormatter** class.
- By default, **JsonMediaTypeFormatter** uses the Json.NET library to perform serialization. Json.NET is a third-party open source project.
- You can configure the **JsonMediaTypeFormatter** class to use the **DataContractJsonSerializer** instead of Json.NET. To do so, set the **UseDataContractJsonSerializer** property to **true**:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.UseDataContractJsonSerializer = true;
```

JSON Serialization : What Gets Serialized?

By default, all public properties and fields are included in the serialized JSON. To omit a property or field, decorate it with the **JsonIgnore** attribute.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    [JsonIgnore]
    public int ProductCode { get; set; } // omitted
}
```

An "opt-in" approach, decorate the class with the **DataContract** attribute. If this attribute is present, members are ignored unless they have the **DataMember**. You can also use **DataMember** to serialize private members.

JSON and XML Serialization in ASP.NET Web API

```
[DataContract]
public class Product
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public decimal Price { get; set; }
    public int ProductCode { get; set; }  // omitted by default
}
```

Read-Only Properties

Read-only properties are serialized by default.

Dates

- By default, Json.NET writes dates in ISO 8601 format.
- Dates in UTC (Coordinated Universal Time) are written with a "Z" suffix.
- Dates in local time include a time-zone offset.

```
2012-07-27T18:51:45.53403Z         // UTC
2012-07-27T11:51:45.53403-07:00    // Local
```

- By default, Json.NET preserves the time zone. You can override this by setting the DateTimeZoneHandling property:

```
// Convert all dates to UTC
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Utc;
```

- If you prefer to use Microsoft JSON date format ("\/ate(*ticks*)\/") instead of ISO 8601, set the **DateFormatHandling** property on the serializer settings:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateFormatHandling
= Newtonsoft.Json.DateFormatHandling.MicrosoftDateFormat;
```

JSON and XML Serialization in ASP.NET Web API

Indenting

To write indented JSON, set the **Formatting** setting to **Formatting.Indented**:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
```

Camel Casing

To write JSON property names with camel casing, without changing your data model, set the **CamelCasePropertyNamesContractResolver** on the serializer :

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
```

Anonymous and Weakly-Typed Objects

An action method can return an anonymous object and serialize it to JSON. For example:

```
public object Get()
{
    return new {
        Name = "Alice",
        Age = 23,
        Pets = new List<string> { "Fido", "Polly", "Spot" }
    };
}
```

The response message body will contain the following JSON:

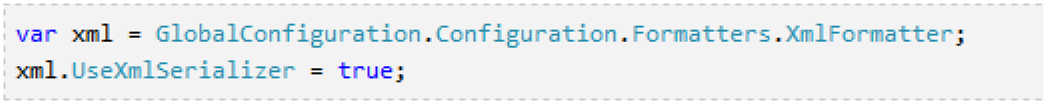
```
{"Name":"Alice","Age":23,"Pets":["Fido","Polly","Spot"]}
```

It is usually better to use strongly typed data objects. Then you don't need to parse the data yourself, and you get the benefits of model validation.

JSON and XML Serialization in ASP.NET Web API

XML Media-Type Formatter

- XML formatting is provided by the **XmlMediaTypeFormatter** class.
- By default, **XmlMediaTypeFormatter** uses the **DataContractSerializer** class to perform serialization.
- You can configure the **XmlMediaTypeFormatter** to use the **XmlSerializer** instead of the **DataContractSerializer**. To do so, set the **UseXmlSerializer** property to **true**:



```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.UseXmlSerializer = true;
```

- The **XmlSerializer** class supports a narrower set of types than **DataContractSerializer**, but gives more control over the resulting XML.
- Consider using **XmlSerializer** if you need to match an existing XML schema.

XML Serialization

The DataContractSerializer behaves as follows:

- All public read/write properties and fields are serialized. To omit a property or field, decorate it with the **IgnoreDataMember** attribute.
- Private and protected members are not serialized.
- Read-only properties are not serialized. (However, the contents of a read-only collection property are serialized.)
- Class and member names are written in the XML exactly as they appear in the class declaration.
- A default XML namespace is used.

If you need more control over the serialization, you can decorate the class with the **DataContract** attribute. When this attribute is present, the class is serialized as follows:

- "Opt in" approach: Properties and fields are not serialized by default. To serialize a property or field, decorate it with the **DataMember** attribute.
- To serialize a private or protected member, decorate it with the **DataMember** attribute.
- Read-only properties are not serialized.
- To change how the class name appears in the XML, set the *Name* parameter in the **DataContract** attribute.

JSON and XML Serialization in ASP.NET Web API

Read-Only Properties

Read-only properties are not serialized. If a read-only property has a backing private field, you can mark the private field with the **DataMember** attribute. This approach requires the **DataContract** attribute on the class.

```
[DataContract]
public class Product
{
    [DataMember]
    private int pcode;  // serialized

    // Not serialized (read-only)
    public int ProductCode { get { return pcode; } }
}
```

Dates

Dates are written in ISO 8601 format. For example, "2021-04-23T20:21:37.9116538Z".

Indenting

To write indented XML, set the **Indent** property to **true**:

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.Indent = true;
```

Setting Per-Type XML Serializers

You can set different XML serializers for different CLR types. For example, you might have a particular data object that requires **XmlSerializer** for backward compatibility. You can use **XmlSerializer** for this object and continue to use **DataContractSerializer** for other types. To set an XML serializer for a particular type, call **SetSerializer**.

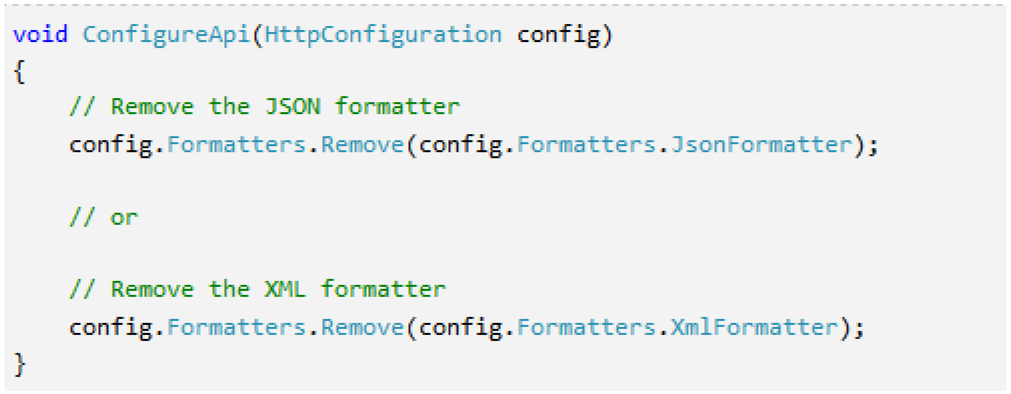
```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
// Use XmlSerializer for instances of type "Product":
xml.SetSerializer<Product>(new XmlSerializer(typeof(Product)));
```

Removing the JSON or XML Formatter

You can remove the JSON formatter or the XML formatter from the list of formatters, if you do not want to use them. The main reasons to do this are:

- To restrict your web API responses to a particular media type. For example, you might decide to support only JSON responses, and remove the XML formatter.
- To replace the default formatter with a custom formatter. For example, you could replace the JSON formatter with your own custom implementation of a JSON formatter.

The following code shows how to remove the default formatters. Call this from your **Application_Start** method, defined in Global.asax.



```
void ConfigureApi(HttpConfiguration config)
{
    // Remove the JSON formatter
    config.Formatters.Remove(config.Formatters.JsonFormatter);

    // or

    // Remove the XML formatter
    config.Formatters.Remove(config.Formatters.XmlFormatter);
}
```

Handling Circular Object References

JSON and XML formatters write all objects as values. If two properties refer to the same object, or if the same object appears twice in a collection, the formatter will serialize the object twice. This is a particular problem if your object graph contains cycles, because the serializer will throw an exception when it detects a loop in the graph.
Consider the following object models and controller.

```
public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
}

public class Department
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}

public class DepartmentsController : ApiController
{
    public Department Get(int id)
    {
        Department sales = new Department() { Name = "Sales" };
        Employee alice = new Employee() { Name = "Alice", Department = sales };
        sales.Manager = alice;
        return sales;
    }
}
```

Invoking this action will cause the formatter to thrown an exception, which translates to a status code 500 (Internal Server Error) response to the client. To preserve object references in JSON, add the following code to **Application_Start** method in the Global.asax file:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.PreserveReferencesHandling =
    Newtonsoft.Json.PreserveReferencesHandling.All;
```

Now the controller action will return JSON that looks like this:

```
{"$id":"1","Name":"Sales","Manager":{"$id":"2","Name":"Alice","Department":{"$ref":"1"}}}
```

Handling Circular Object References

- Serializer adds an "$id" property to both objects. Also, it detects that the Employee.Department property creates a loop, so it replaces the value with an object reference: {"$ref":"1"}.
- Object references are not standard in JSON. Before using this feature, consider whether your clients will be able to parse the results. It might be better simply to remove cycles from the graph. For example, the link from Employee back to Department is not really needed in this example.

To preserve object references in XML, you have two options. The simpler option is to add [DataContract(IsReference=true)] to your model class. The *IsReference* parameter enables object references. Remember that **DataContract** makes serialization opt-in, so you will also need to add **DataMember** attributes to the properties

```
[DataContract(IsReference=true)]
public class Department
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public Employee Manager { get; set; }
}
```

Now the formatter will produce XML similar to following:

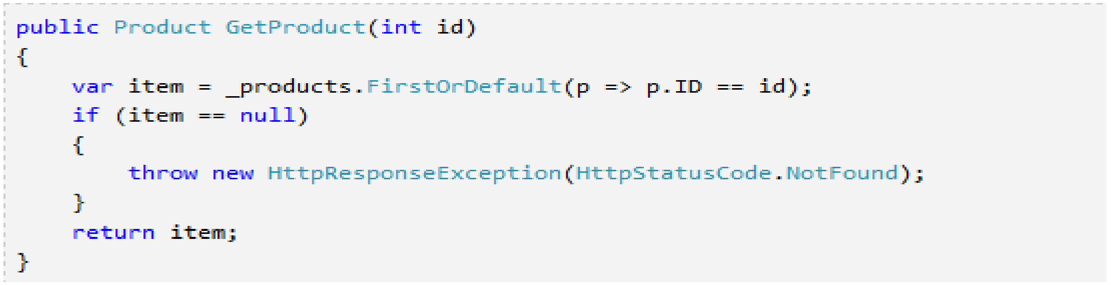
```
<Department xmlns:i="http://www.w3.org/2001/XMLSchema-instance" z:Id="i1"
            xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
            xmlns="http://schemas.datacontract.org/2004/07/Models">
  <Manager>
    <Department z:Ref="i1" />
    <Name>Alice</Name>
  </Manager>
  <Name>Sales</Name>
</Department>
```

Content Negotiation

- The HTTP specification (RFC 2616) defines content negotiation as “the process of selecting the best representation for a given response when there are multiple representations available.”
- The primary mechanism for content negotiation in HTTP are these request headers:
  - **Accept:** Which media types are acceptable for the response, such as “application/json,” “application/xml,” or a custom media type such as "application/vnd.example + xml"
  - **Accept-Charset:** Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
  - **Accept-Encoding:** Which content encodings are acceptable, such as gzip.
  - **Accept-Language:** The preferred natural language, such as “en-us”.
- The server can also look at other portions of the HTTP request. For example, if the request contains an X-Requested-With header, indicating an AJAX request, the server might default to JSON if there is no Accept header.

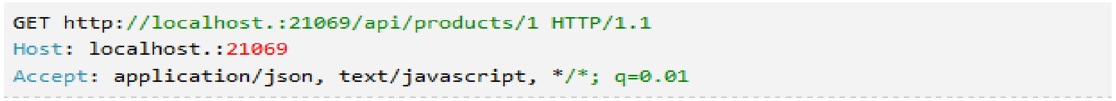
Serialization

If a Web API controller returns a resource as CLR type, the pipeline serializes the return value and writes it into the HTTP response body. For example, consider the following controller action:



```
public Product GetProduct(int id)
{
    var item = _products.FirstOrDefault(p => p.ID == id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

A client might send this HTTP request:



```
GET http://localhost.:21069/api/products/1 HTTP/1.1
Host: localhost.:21069
Accept: application/json, text/javascript, */*; q=0.01
```

Content Negotiation

In response, the server might send:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 57
Connection: Close

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

the client requested either JSON, Javascript or “anything” (*/*). The server responsed with a JSON representation of the Product object. Notice that the Content-Type header in the response is set to "application/json". A controller can also return an **HttpResponseMessage** object. To specify a CLR object for the response body, call the **CreateResponse** extension method:

```
public HttpResponseMessage GetProduct(int id)
{
    var item = _products.FirstOrDefault(p => p.ID == id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return Request.CreateResponse(HttpStatusCode.OK, product);
}
```

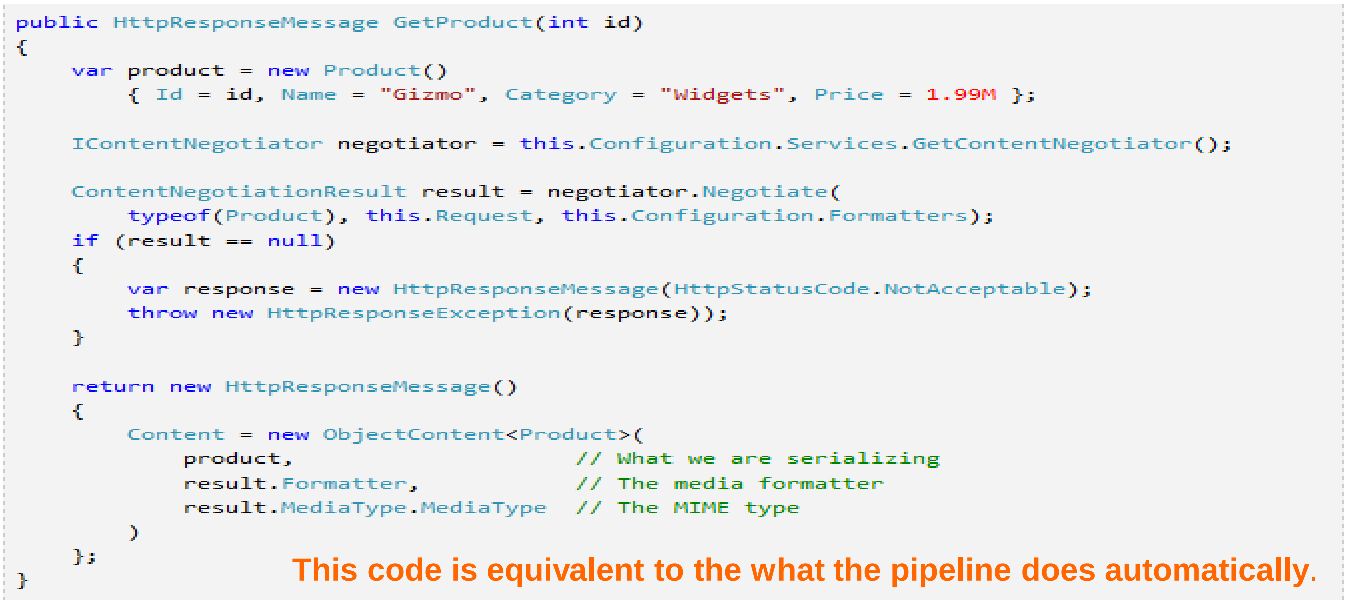
This option gives you more control over the details of the response. You can set the status code, add HTTP headers, and so forth.

The object that serializes the resource is called a *media formatter*. Media formatters derive from the **MediaTypeFormatter** class. Web API provides media formatters for XML and JSON, and you can create custom formatters to support other media types.

How Content Negotiation works?

- First, the pipeline gets the **IContentNegotiator** service from the **HttpConfiguration** object. It also gets the list of media formatters from the **HttpConfiguration.Formatters** collection.
- Next, the pipeline calls **IContentNegotiatior.Negotiate**, passing in:
  - The type of object to serialize
  - The collection of media formatters
  - The HTTP request
- The Negotiate method returns two pieces of information:
  - Which formatter to use
  - The media type for the response

If no formatter is found, the **Negotiate** method returns **null**, and the client recevies HTTP error 406 (Not Acceptable).
The following code shows how a controller can directly invoke content negotiation:



```
public HttpResponseMessage GetProduct(int id)
{
    var product = new Product()
        { Id = id, Name = "Gizmo", Category = "Widgets", Price = 1.99M };

    IContentNegotiator negotiator = this.Configuration.Services.GetContentNegotiator();

    ContentNegotiationResult result = negotiator.Negotiate(
        typeof(Product), this.Request, this.Configuration.Formatters);
    if (result == null)
    {
        var response = new HttpResponseMessage(HttpStatusCode.NotAcceptable);
        throw new HttpResponseException(response));
    }

    return new HttpResponseMessage()
    {
        Content = new ObjectContent<Product>(
            product,                    // What we are serializing
            result.Formatter,           // The media formatter
            result.MediaType.MediaType  // The MIME type
        )
    };
}
```

This code is equivalent to the what the pipeline does automatically.

Default Content Negotiator

The **DefaultContentNegotiator** class provides the default implementation of **IContentNegotiator**. It uses several criteria to select a formatter.

- First, the formatter must be able to serialize the type. This is verified by calling **MediaTypeFormatter.CanWriteType**.
- Next, the content negotiator looks at each formatter and evaluates how well it matches the HTTP request. To evaluate the match, the content negotiator looks at two things on the formatter:
  - The **SupportedMediaTypes** collection, which contains a list of supported media types. The content negotiator tries to match this list against the request Accept header. Note that the Accept header can include ranges. For example, "text/plain" is a match for text/* or */*.
  - The **MediaTypeMappings** collection, which contains a list of **MediaTypeMapping** objects. The **MediaTypeMapping** class provides a generic way to match HTTP requests with media types. For example, it could map a custom HTTP header to a particular media type.

If there are multiple matches, the match with the highest quality factor wins. For example:

```
Accept: application/json, application/xml; q=0.9, */*; q=0.1
```

- Application/json has an implied quality factor of 1.0, so it is preferred over application/xml.
- If no matches are found, the content negotiator tries to match on the media type of the request body, if any. For example, if the request contains JSON data, the content negotiator looks for a JSON formatter.
- If there are still no matches, the content negotiator simply picks the first formatter that can serialize the type.

Selecting a Character Encoding

After a formatter is selected, the content negotiator chooses the best character encoding. by looking at the **SupportedEncodings** property on the formatter, and matching it against the Accept-Charset header in the request (if any).

Model Validations

When a client sends data to your web API, often you want to validate the data before doing any processing. We have to annotate models, use the annotations for data validation, and handle validation errors in your web API.

Data Annotations

In ASP.NET Web API, you can use attributes from the System.ComponentModel.DataAnnotations namespace to set validation rules for properties on your model.

```
using System.ComponentModel.DataAnnotations;

namespace MyApi.Models
{
    public class Product
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public decimal Price { get; set; }
        [Range(0, 999)]
        public double Weight { get; set; }
    }
}
```

- You have used model validation in ASP.NET MVC, this should look familiar. The **Required** attribute says that the Name property must not be null. The **Range** attribute says that Weight must be between zero and 999.
- Suppose that a client sends a POST request with the following JSON representation:

```
{ "Id":4, "Price":2.99, "Weight":5 }
```

You can see that the client did not include the Name property, which is marked as required. When Web API converts the JSON into a Product instance, it validates the Product against the validation attributes. In your controller action, you can check whether the model is valid:

Model validation does not guarantee that client data is safe. Additional validation might be needed in other layers of the application.

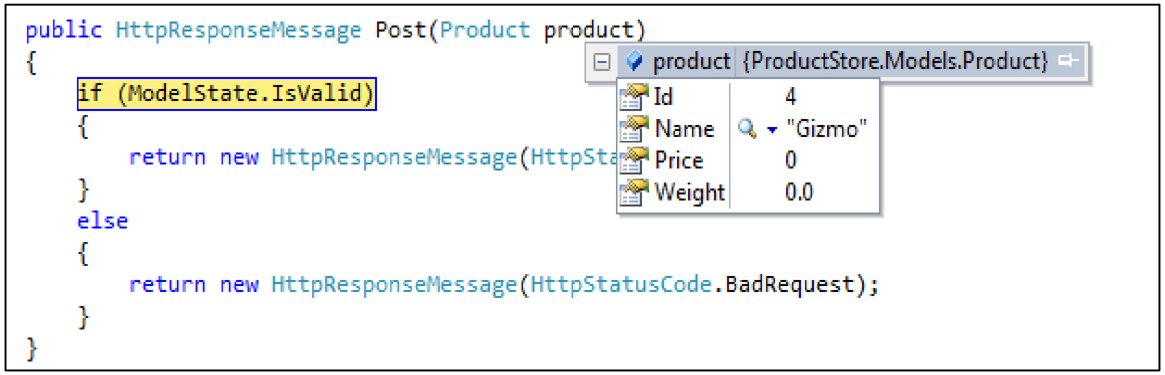
Model Validations

```
public class ProductsController : ApiController
{
    public HttpResponseMessage Post(Product product)
    {
        if (ModelState.IsValid)
        {
            // Do something with the product (not shown).

            return new HttpResponseMessage(HttpStatusCode.OK);
        }
        else
        {
            return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
        }
    }
}
```

"Under-Posting": Under-posting happens when the client leaves out some properties. For example, suppose the client sends the following:

```
{"Id":4, "Name":"Gizmo"}
```

Here, the client did not specify values for Price or Weight. The JSON formatter assigns a default value of zero to the missing properties.



```
public HttpResponseMessage Post(Product product)
{
    if (ModelState.IsValid)
    {
        return new HttpResponseMessage(HttpSta
    }
    else
    {
        return new HttpResponseMessage(HttpStatusCode.BadRequest);
    }
}
```

```
product {ProductStore.Models.Product}
  Id       4
  Name     "Gizmo"
  Price    0
  Weight   0.0
```

Model Validations

The model state is valid, because zero is a valid value for these properties. Whether this is a problem depends on your scenario. For example, in an update operation, you might want to distinguish between "zero" and "not set." To force clients to set a value, make the property Nullable and set the **Required** attribute:

```
[Required]
public decimal? Price { get; set; }
```

"Over-Posting": A client can also send *more* data than you expected. For example:

```
{"Id":4, "Name":"Gizmo", "Color":"Blue"}
```

Here, the JSON includes a property (Color") that does not exist in the Product model. In this case, the JSON formatter simply ignores this value. (The XML formatter does the same.) Over-posting causes problems if your model has properties that you intended to be read-only. For example:

```
public class UserProfile
{
    public string Name { get; set; }
    public Uri Blog { get; set; }
    public bool IsAdmin { get; set; }  // uh-oh!
}
```

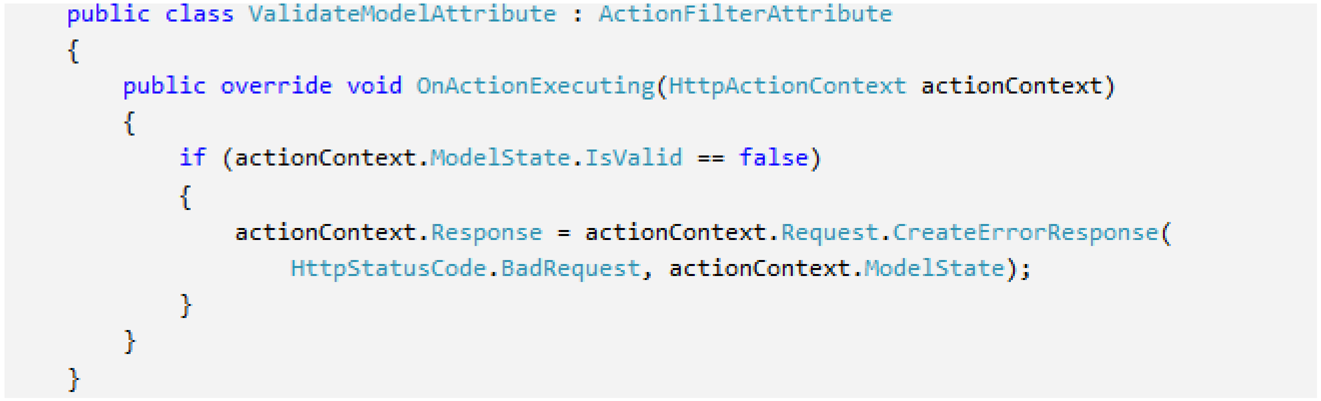
You don't want users to update the IsAdmin property and elevate themselves to administrators! The safest strategy is to use a model class that exactly matches what the client is allowed to send:

```
public class UserProfileDTO
{
    public string Name { get; set; }
    public Uri Blog { get; set; }
    // Leave out "IsAdmin"
}
```

Handling Validation Errors

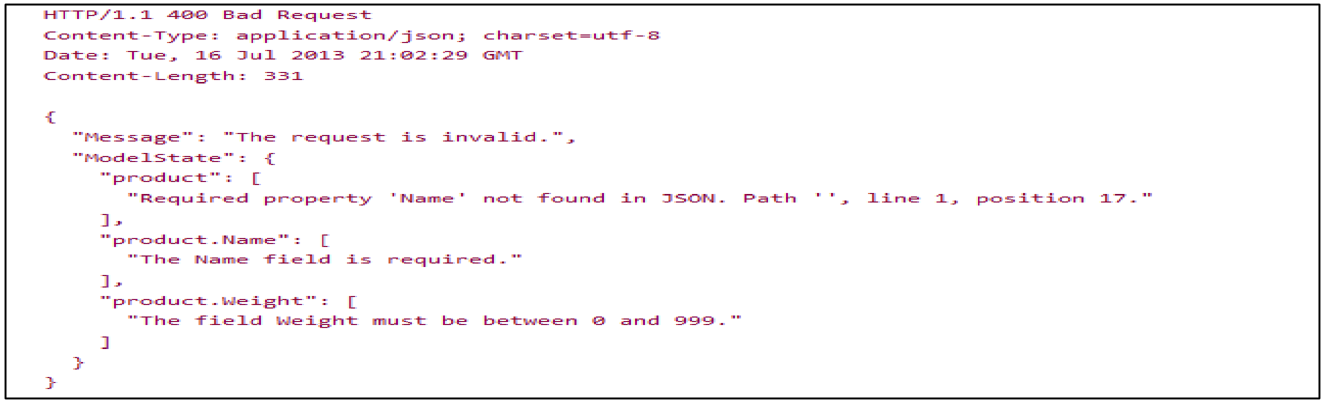
Web API does not automatically return an error to the client when validation fails. It is up to the controller action to check the model state and respond appropriately.

You can also create an action filter to check the model state before the controller action is invoked. The following code shows an example:



```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        if (actionContext.ModelState.IsValid == false)
        {
            actionContext.Response = actionContext.Request.CreateErrorResponse(
                HttpStatusCode.BadRequest, actionContext.ModelState);
        }
    }
}
```

If model validation fails, this filter returns an HTTP response that contains the validation errors. In that case, the controller action is not invoked.



```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Date: Tue, 16 Jul 2013 21:02:29 GMT
Content-Length: 331

{
  "Message": "The request is invalid.",
  "ModelState": {
    "product": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 17."
    ],
    "product.Name": [
      "The Name field is required."
    ],
    "product.Weight": [
      "The field Weight must be between 0 and 999."
    ]
  }
}
```

Handling Validation Errors

To apply this filter to all Web API controllers, add an instance of the filter to the **HttpConfiguration.Filters** collection during configuration:

```
public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.Filters.Add(new ValidateModelAttribute());

            // ...
        }
}
```

Another option is to set the filter as an attribute on individual controllers or controller actions:

```
public class ProductsController : ApiController
{
    [ValidateModel]
    public HttpResponseMessage Post(Product product)
    {
        // ...
    }
}
```

Parameter Binding in ASP.NET Web API

- When Web API calls a method on a controller, it must set values for the parameters, a process called *binding*.
- By default, Web API uses the following rules to bind parameters:

  - If the parameter is a “simple” type, Web API tries to get the value from the URI. Simple types include the .NET primitive types (**int**, **bool**, **double**, and so forth), plus **TimeSpan**, **DateTime**, **Guid**, **decimal**, and **string**, plus any type with a type converter that can convert from a string.
  - For complex types, Web API tries to read the value from the message body, using a media-type formatter.

For example, here is a typical Web API controller method:

```
HttpResponseMessage Put(int id, Product item) { ... }
```

- The *id* parameter is a "simple" type, so Web API tries to get the value from the request URI. The *item* parameter is a complex type, so Web API uses a media-type formatter to read the value from the request body.
- To get a value from the URI, Web API looks in the route data and the URI query string. The route data is populated when the routing system parses the URI and matches it to a route.

Customize the model binding process

For complex types, however, consider using media-type formatters whenever possible. A key principle of HTTP is that resources are sent in the message body, using content negotiation to specify the representation of the resource. Media-type formatters were designed for exactly this purpose.

Using [FromUri]

To force Web API to read a complex type from the URI, add the **[FromUri]** attribute to the parameter. The following example defines a GeoPoint type, along with a controller method that gets the GeoPoint from the URI.

Parameter Binding in ASP.NET Web API

```
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}

public ValuesController : ApiController
{
    public HttpResponseMessage Get([FromUri] GeoPoint location) { ... }
}
```

The client can put the Latitude and Longitude values in the query string and Web API will use them to construct a GeoPoint. For example: http://localhost/api/values/?Latitude=47.678558&Longitud=-122.130989

Using [FromBody]

To force Web API to read a simple type from the request body, add the **[FromBody]** attribute to the parameter:

```
public HttpResponseMessage Post([FromBody] string name) { ... }
```

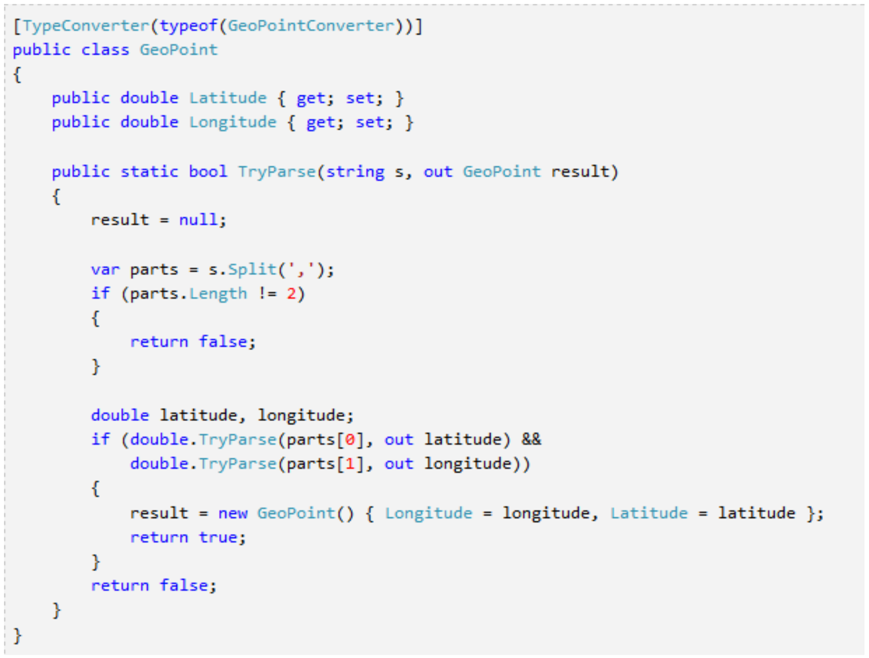
In this example, Web API will use a media-type formatter to read the value of *name* from the request body. Here is an example client request.

```
POST http://localhost:5076/api/values HTTP/1.1
User-Agent: Fiddler
Host: localhost:5076
Content-Type: application/json
Content-Length: 7

"Alice"
```

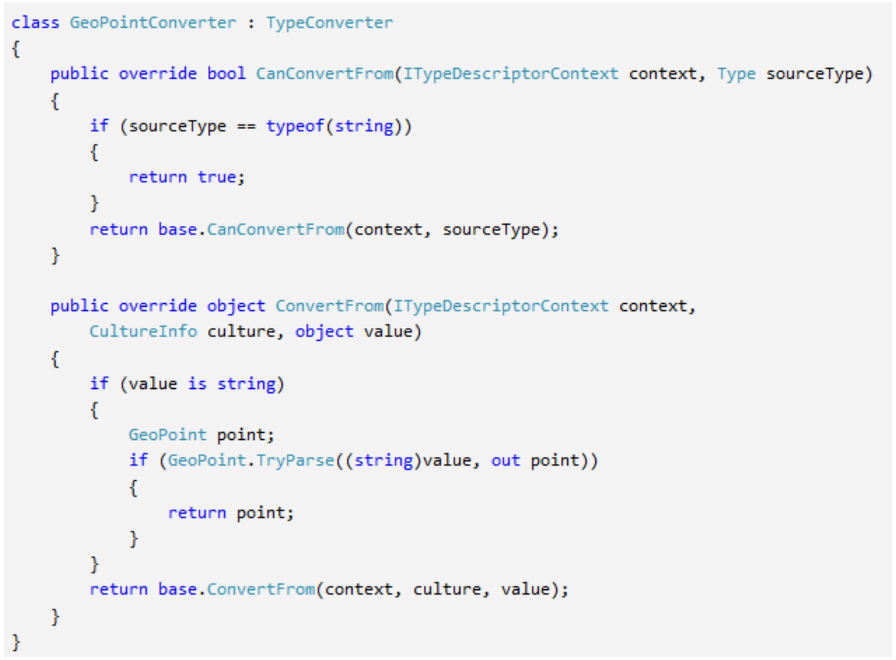
- When a parameter has [FromBody], Web API uses the Content-Type header to select a formatter. In this example, the content type is "application/json" and the request body is a raw JSON string (not a JSON object).

Type Converters

You can make Web API treat a class as a simple type (so that Web API will try to bind it from the URI) by creating a **TypeConverter** and providing a string conversion.
The following code shows a GeoPoint class that represents a geographical point, plus a **TypeConverter** that converts from strings to GeoPoint instances. The GeoPoint class is decorated with a **[TypeConverter]** attribute to specify the type converter.

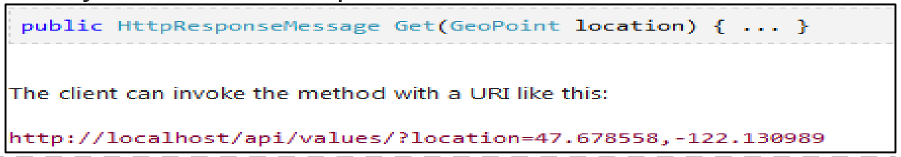


```
[TypeConverter(typeof(GeoPointConverter))]
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }

    public static bool TryParse(string s, out GeoPoint result)
    {
        result = null;

        var parts = s.Split(',');
        if (parts.Length != 2)
        {
            return false;
        }

        double latitude, longitude;
        if (double.TryParse(parts[0], out latitude) &&
            double.TryParse(parts[1], out longitude))
        {
            result = new GeoPoint() { Longitude = longitude, Latitude = latitude };
            return true;
        }
        return false;
    }
}
```



```
class GeoPointConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context, Type sourceType)
    {
        if (sourceType == typeof(string))
        {
            return true;
        }
        return base.CanConvertFrom(context, sourceType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
        CultureInfo culture, object value)
    {
        if (value is string)
        {
            GeoPoint point;
            if (GeoPoint.TryParse((string)value, out point))
            {
                return point;
            }
        }
        return base.ConvertFrom(context, culture, value);
    }
}
```

Now Web API will treat GeoPoint as a simple type, meaning it will try to ind GeoPoint parameters from the URI. You don't need to include **[FromUri]** on the parameter.



```
public HttpResponseMessage Get(GeoPoint location) { ... }
```

The client can invoke the method with a URI like this:

```
http://localhost/api/values/?location=47.678558,-122.130989
```

Security

Authentication and Authorization in ASP.NET Web API

- Once web API is created, we want to control access to it. Let’s look at some options for securing a web API from unauthorized users.
  - **Authentication**: It is knowing the identity of the user. E.g. Alice logs in with her username and password, and the server uses the password to authenticate Alice.
  - **Authorization:** It is deciding whether a user is allowed to perform an action. E.g. Alice has permission to get a resource but not create a resource.

Authentication

- Web API assumes that authentication happens in the host.
- For web-hosting, the host is IIS, which uses HTTP modules for authentication. You can configure your project to use any of the authentication modules built in to IIS or ASP.NET, or write your own HTTP module to perform custom authentication.
- When the host authenticates the user, it creates a *principal*, which is an IPrincipal object that represents the security context under which code is running. The host attaches the principal to the current thread by setting **Thread.CurrentPrincipal**.
- The principal contains an associated **Identity** object that contains information about the user. If the user is authenticated, the **Identity.IsAuthenticated** property returns **true**. For anonymous requests, **IsAuthenticated** returns **false.**

HTTP Message Handlers for Authentication

Instead of using the host for authentication, you can put authentication logic into an HTTP message handler. In that case, the message handler examines the HTTP request and sets the principal.

When should you use message handlers for authentication? Here are some tradeoffs:

- An HTTP module sees all requests that go through the ASP.NET pipeline. A message handler only sees requests that are routed to Web API.
- You can set per-route message handlers, which lets you apply an authentication scheme to a specific route.

Authentication and Authorization in ASP.NET Web API

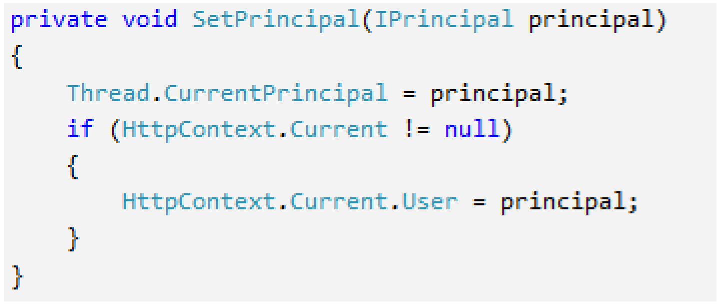
- HTTP modules are specific to IIS. Message handlers are host-agnostic, so they can be used with both web-hosting and self-hosting.
- HTTP modules participate in IIS logging, auditing, and so on.
- HTTP modules run earlier in the pipeline. If you handle authentication in a message handler, the principal does not get set until the handler runs. Moreover, the principal reverts back to the previous principal when the response leaves the message handler.

- Generally, if you don't need to support self-hosting, an HTTP module is a better option. If you need to support self-hosting, consider a message handler.

Setting the Principal

- If your application performs any custom authentication logic, you must set the principal on two places:
  - **Thread.CurrentPrincipal**. This property is the standard way to set the thread's principal in .NET.
  - **HttpContext.Current.User**. This property is specific to ASP.NET.

- For web-hosting, you must set the principal in both places.Otherwise the security context may become inconsistent.
- For self-hosting, however, **HttpContext.Current** is null. To ensure your code is host-agnostic, therefore, check for null before assigning to **HttpContext.Current**, as shown.

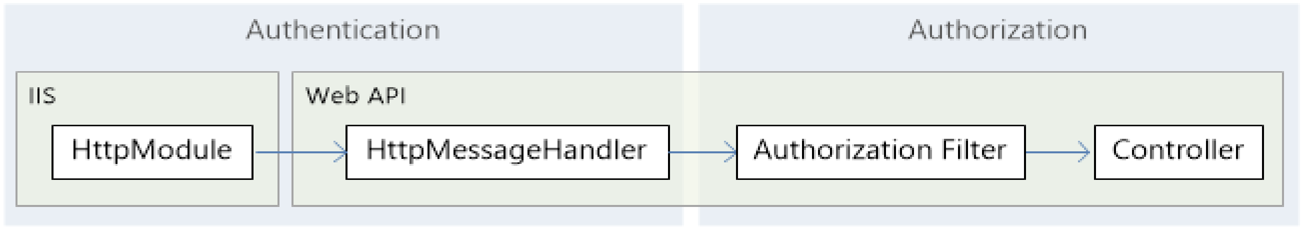


```
private void SetPrincipal(IPrincipal principal)
{
    Thread.CurrentPrincipal = principal;
    if (HttpContext.Current != null)
    {
        HttpContext.Current.User = principal;
    }
}
```

Authentication and Authorization in ASP.NET Web API

Authorization

- Authorization happens later in the pipeline, closer to the controller. That lets you make more granular choices when you grant access to resources.
- Authorization filters run before the controller action. If the request is not authorized, the filter returns an error response, and the action is not invoked.
- Within a controller action, you can get the current principal from the ApiController.User property. E.g. you might filter a list of resources based on the user name, returning only those resources that belong to that user.



Using the [Authorize] Attribute
Web API provides a built-in authorization filter, AuthorizeAttribute. This filter checks whether the user is authenticated. If not, it returns HTTP status code 401 (Unauthorized), without invoking the action. You can apply the filter globally, at the controller level, or at the level of individual actions.
Globally: To restrict access for every Web API controller, add the **AuthorizeAttribute** filter to the global filter list:

```
public static void Register(HttpConfiguration config)
{
    config.Filters.Add(new AuthorizeAttribute());
}
```

Controller: To restrict access for a specific controller, add the filter as an attribute to the controller:

Authentication and Authorization in ASP.NET Web API

```
[Authorize]
public class ValuesController : ApiController
{
    public HttpResponseMessage Get(int id) { ... }
    public HttpResponseMessage Post() { ... }
}
```

Action: To restrict access for specific actions, add the attribute to the action method:

```
public class ValuesController : ApiController
{
    public HttpResponseMessage Get() { ... }

    // Require authorization for a specific action.
    [Authorize]
    public HttpResponseMessage Post() { ... }
}
```

Alternatively, you can restrict the controller and then allow anonymous access to specific actions, by using the [**AllowAnonymous**] attribute. E.g. the Post method is restricted, but the Get method allows anonymous access.

```
[Authorize]
public class ValuesController : ApiController
{
    [AllowAnonymous]
    public HttpResponseMessage Get() { ... }

    public HttpResponseMessage Post() { ... }
}
```

Authentication and Authorization in ASP.NET Web API

In the previous example, the filter allows any authenticated user to access the restricted methods; only anonymous users are kept out. You can also limit access to specific users or to users in specific roles:

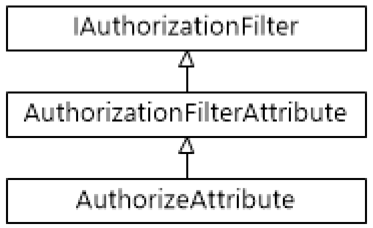
```
// Restrict by user:
[Authorize(Users="Alice,Bob")]
public class ValuesController : ApiController
{
}

// Restrict by role:
[Authorize(Roles="Administrators")]
public class ValuesController : ApiController
{
}
```

The **AuthorizeAttribute** filter for Web API controllers is located in the **System.Web.Http** namespace. There is a similar filter for MVC controllers in the **System.Web.Mvc** namespace, which is not compatible with Web API controllers.

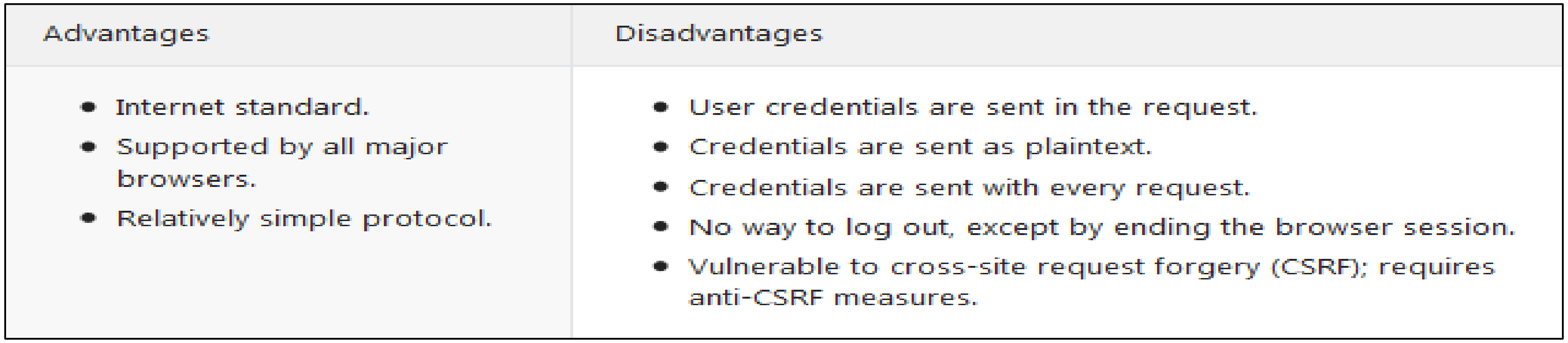
Custom Authorization Filters

To write a custom authorization filter, derive from one of these types:

- **AuthorizeAttribute**. Extend this class to perform authorization logic based on the current user and the user's roles.
- **AuthorizationFilterAttribute**. Extend this class to perform synchronous authorization logic that is not necessarily based on the current user or role.
- **IAuthorizationFilter**. Implement this interface to perform asynchronous authorization logic; for example, if your authorization logic makes asynchronous I/O or network calls. (If your authorization logic is CPU-bound, it is simpler to derive from **AuthorizationFilterAttribute**, because then you don't need to write an asynchronous method.)



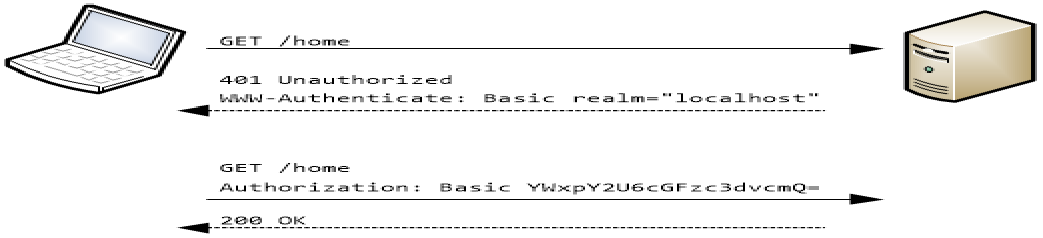
Basic Authentication



Advantages	Disadvantages
• Internet standard. • Supported by all major browsers. • Relatively simple protocol.	• User credentials are sent in the request. • Credentials are sent as plaintext. • Credentials are sent with every request. • No way to log out, except by ending the browser session. • Vulnerable to cross-site request forgery (CSRF); requires anti-CSRF measures.

Basic authentication works as follows:

- If a request requires authentication, the server returns 401 (Unauthorized). The response includes a WWW-Authenticate header, indicating the server supports Basic authentication.
- The client sends another request, with the client credentials in the Authorization header. The credentials are formatted as the string “name:password”, base64-encoded. The credentials are not encrypted.
- Basic authentication is performed within the context of a “realm.” The server includes the name of the realm in the WWW-Authenticate header. The user’s credentials are valid within that realm. The exact scope of a realm is defined by the server. For example, you might define several realms in order to partition resources.
- Because the credentials are sent unencrypted, Basic authentication is only secure over HTTPS.
- Basic authentication is also vulnerable to CSRF attacks. After the user enters credentials, the browser automatically sends them on subsequent requests to the same domain, for the duration of the session. This includes AJAX requests.



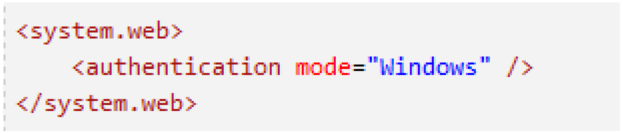
```
GET /home

401 Unauthorized
WWW-Authenticate: Basic realm="localhost"

GET /home
Authorization: Basic YWxpY2U6cGFzc3dvcmQ=

200 OK
```

Abhishek Sharma

Basic Authentication

Basic Authentication with IIS

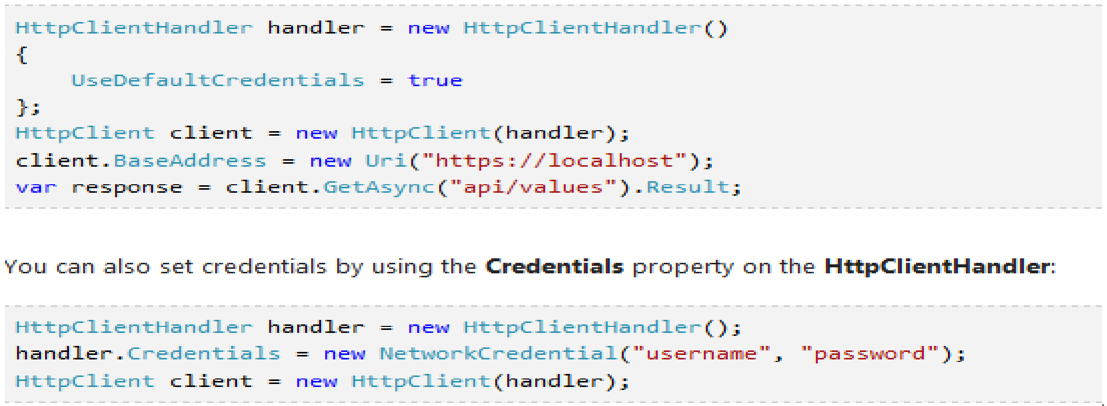
- IIS supports Basic authentication, but there is a caveat: The user is authenticated against their Windows credentials. That means the user must have an account on the server's domain. For a public-facing web site, you typically want to authenticate against an ASP.NET membership provider.
- To enable Basic authentication using IIS, set the authentication mode to "Windows" in the Web.config of your ASP.NET project:



```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

In this mode, IIS uses Windows credentials to authenticate. In addition, you must enable Basic authentication in IIS. In IIS Manager, go to Features View, select Authentication, and enable Basic authentication.

- In your Web API project, add the [Authorize] attribute for any controller actions that need authentication.
- A client authenticates itself by setting the Authorization header in the request. Browser clients perform this step automatically. Non-browser clients will need to set the header. The following C# example uses **HttpClient** to send the user's default credentials:



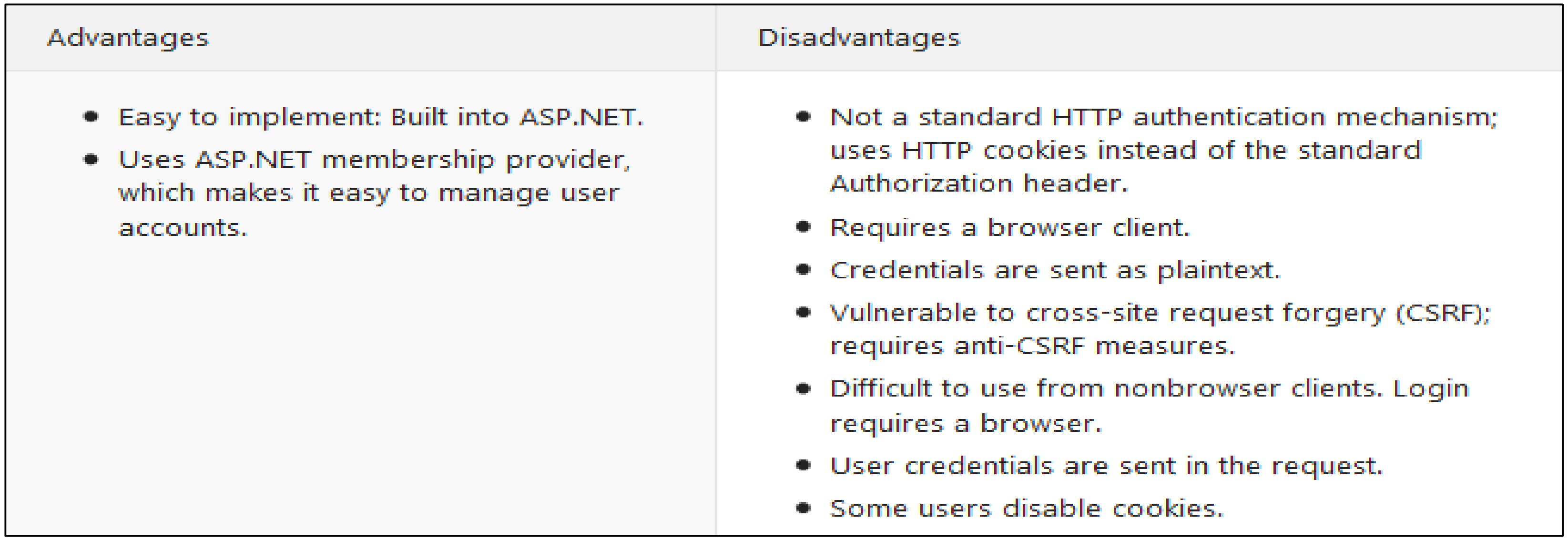
```
HttpClientHandler handler = new HttpClientHandler()
{
    UseDefaultCredentials = true
};
HttpClient client = new HttpClient(handler);
client.BaseAddress = new Uri("https://localhost");
var response = client.GetAsync("api/values").Result;
```

You can also set credentials by using the **Credentials** property on the **HttpClientHandler**:

```
HttpClientHandler handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential("username", "password");
HttpClient client = new HttpClient(handler);
```

Form Authentication

Forms authentication uses an HTML form to send the user's credentials to the server. It is not an Internet standard. Forms authentication is only appropriate for web APIs that are called from a web application, so that the user can interact with the HTML form.

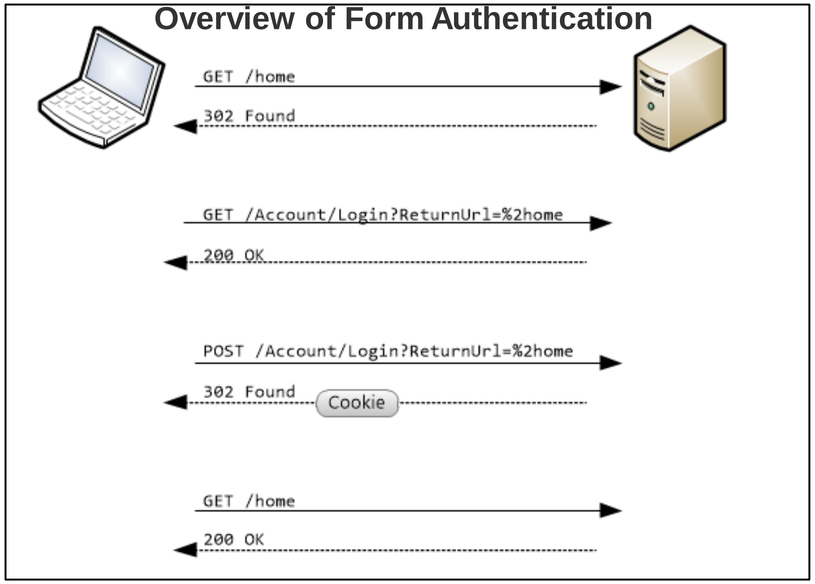


Advantages	Disadvantages
• Easy to implement: Built into ASP.NET. • Uses ASP.NET membership provider, which makes it easy to manage user accounts.	• Not a standard HTTP authentication mechanism; uses HTTP cookies instead of the standard Authorization header. • Requires a browser client. • Credentials are sent as plaintext. • Vulnerable to cross-site request forgery (CSRF); requires anti-CSRF measures. • Difficult to use from nonbrowser clients. Login requires a browser. • User credentials are sent in the request. • Some users disable cookies.

Briefly, forms authentication in ASP.NET works like this:

- The client requests a resource that requires authentication.
- If the user is not authenticated, the server returns HTTP 302 (Found) and redirects to a login page.
- The user enters credentials and submits the form.
- The server returns another HTTP 302 that redirects back to the original URI. This response includes an authentication cookie.
- The client requests the resource again. The request includes the authentication cookie, so the server grants the request.

Form Authentication

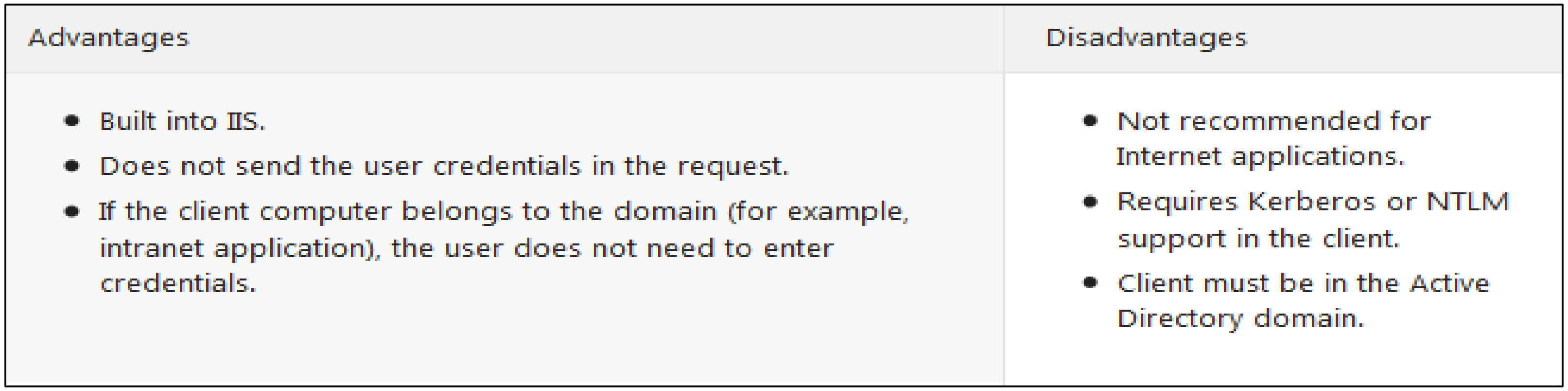

Overview of Form Authentication
GET /home
302 Found
GET /Account/Login?ReturnUrl=%2home
200 OK
POST /Account/Login?ReturnUrl=%2home
302 Found
Cookie
GET /home
200 OK

Using Forms Authentication with Web API

- To create an application that uses forms authentication, select the “Internet Application” template in the MVC 4 project wizard. This template creates MVC controllers for account management. You can also use the “Single Page Application” template, available in the ASP.NET Fall 2012 Update.
- In your web API controllers, you can restrict access by using the [Authorize] attribute, as described in Using the [Authorize] Attribute.
- Forms-authentication uses a session cookie to authenticate requests. Browsers automatically send all relevant cookies to the destination web site. This feature makes forms authentication potentially vulnerable to cross-site request forgery (CSRF) attacks .
- Forms authentication does not encrypt the user’s credentials. Therefore, forms authentication is not secure unless used with SSL.

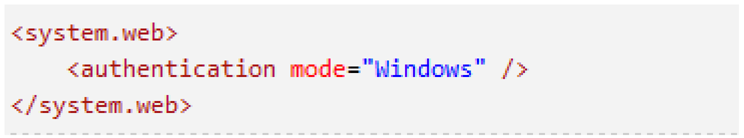
Integrated Windows Authentication

Integrated Windows authentication enables users to log in with their Windows credentials, using Kerberos or NTLM. The client sends credentials in the Authorization header. Windows authentication is best suited for an intranet environment.



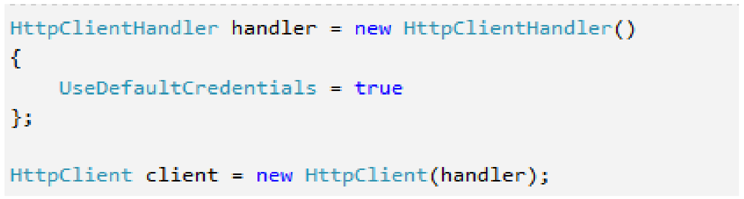
Advantages	Disadvantages
• Built into IIS. • Does not send the user credentials in the request. • If the client computer belongs to the domain (for example, intranet application), the user does not need to enter credentials.	• Not recommended for Internet applications. • Requires Kerberos or NTLM support in the client. • Client must be in the Active Directory domain.

To create an application that uses Integrated Windows authentication, select the "Intranet Application" template in the MVC 4 project wizard. This project template puts the following setting in the Web.config file:



```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

On the client side, Integrated Windows authentication works with any browser that supports the Negotiate authentication scheme, which includes most major browsers. For .NET client applications, the **HttpClient** class supports Windows authentication:



```
HttpClientHandler handler = new HttpClientHandler()
{
    UseDefaultCredentials = true
};

HttpClient client = new HttpClient(handler);
```

Preventing Cross – Site Request Forgery (CSRF) Attacks

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in. Here is an example of a CSRF attack:

- A user logs into www.example.com, using forms authentication.
- The server authenticates the user. The response from the server includes an authentication cookie.
- Without logging out, the user visits a malicious web site. This malicious site contains the following HTML form

```
<h1>You Are a Winner!</h1>
    <form action="http://example.com/api/account" method="post">
        <input type="hidden" name="Transaction" value="withdraw" />
        <input type="hidden" name="Amount" value="1000000" />
 <input type="submit" value="Click Me"/>
    </form>
```

Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

- The user clicks the submit button. The browser includes the authentication cookie with the request.
- The request runs on the serer with the user's authentication context, and can do anything that an authenticated user is allowed to do.

Although this example requires the user to click the form button, the malicious page could just as easily run a script that sends an AJAX request. Moreover, using SSL does not prevent a CSRF attack, because the malicious site can send an "https://" request.

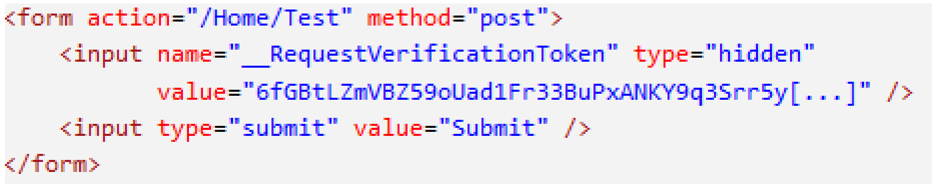
Typically, CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication. the browser automatically sends the credentials until the session ends.

Anti-Forgery Tokens

To help prevent CSRF attacks, ASP.NET MVC uses anti-forgery tokens, also called *request verification tokens*.

- The client requests an HTML page that contains a form.
- The server includes two tokens in the response. One token is sent as a cookie. The other is placed in a hidden form field. The tokens are generated randomly so that an adversary cannot guess the values.
- When the client submits the form, it must send both tokens back to the server. The client sends the cookie token as a cookie, and it sends the form token inside the form data. (A browser client automatically does this when the user submits the form.)
- If a request does not include both tokens, the server disallows the request.

Here is an example of an HTML form with a hidden form token:

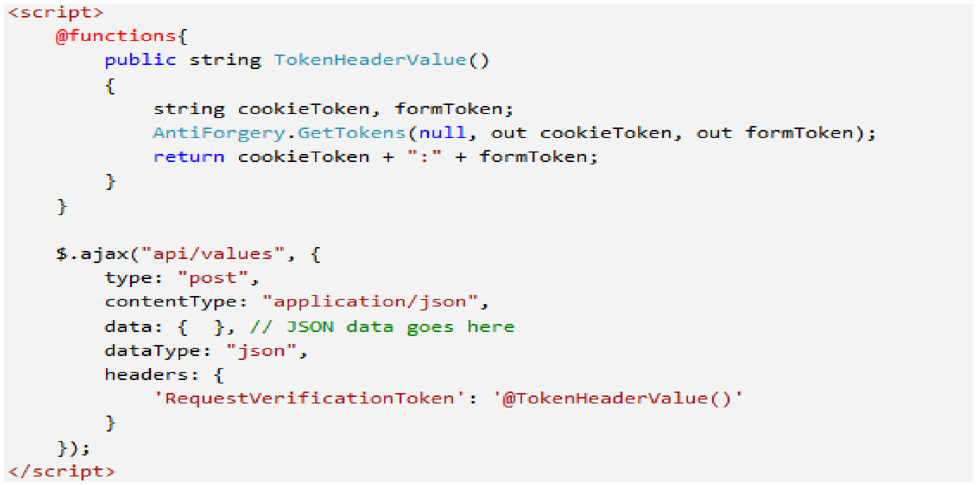


```
<form action="/Home/Test" method="post">
    <input name="__RequestVerificationToken" type="hidden"
           value="6fGBtLZmVBZ59oUad1Fr33BuPxANKY9q3Srr5y[...]" />
    <input type="submit" value="Submit" />
</form>
```

- Anti-forgery tokens work because the malicious page cannot read the user’s tokens, due to same-origin policies. (Same-orgin policies prevent documents hosted on two different sites from accessing each other’s content. So in the earlier example, the malicious page can send requests to example.com, but it cannot read the response.)
- To prevent CSRF attacks, use anti-forgery tokens with any authentication protocol where the browser silently sends credentials after the user logs in. This includes cookie-based authentication protocols, such as forms authentication, as well as protocols such as Basic and Digest authentication.
- You should require anti-forgery tokens for any non-safe methods (POST, PUT, DELETE). Also, make sure that safe methods (GET, HEAD) do not have any side effects. Moreover, if you enable cross-domain support, such as CORS or JSONP, then even safe methods like GET are potentially vulnerable to CSRF attacks, allowing the attacker to read potentially sensitive data.

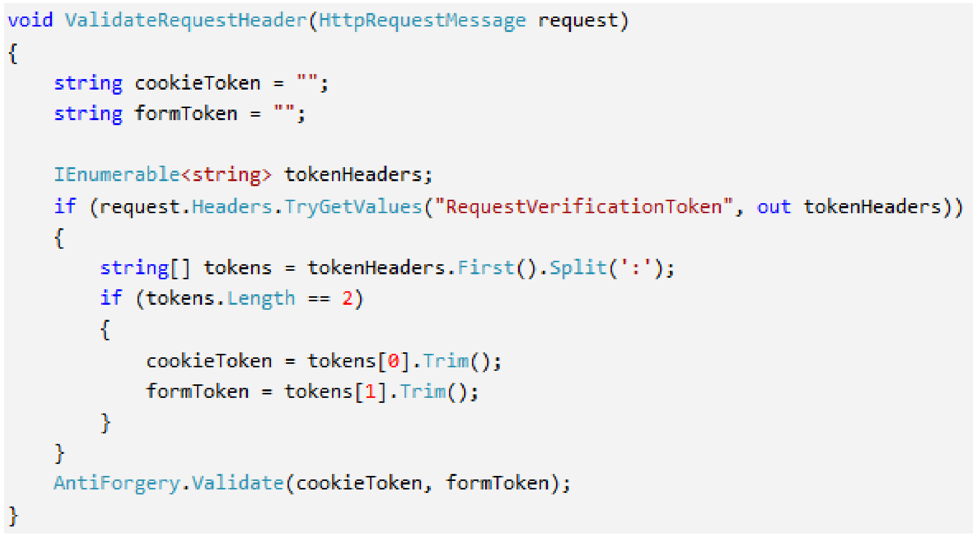
Anti-Forgery Tokens

The form token can be a problem for AJAX requests, because an AJAX request might send JSON data, not HTML form data. One solution is to send the tokens in a custom HTTP header. The following code uses Razor syntax to generate the tokens, and then adds the tokens to an AJAX request. The tokens are generated at the server by calling **AntiForgery.GetTokens**.



```
<script>
    @functions{
        public string TokenHeaderValue()
        {
            string cookieToken, formToken;
            AntiForgery.GetTokens(null, out cookieToken, out formToken);
            return cookieToken + ":" + formToken;
        }
    }

    $.ajax("api/values", {
        type: "post",
        contentType: "application/json",
        data: {  }, // JSON data goes here
        dataType: "json",
        headers: {
            'RequestVerificationToken': '@TokenHeaderValue()'
        }
    });
</script>
```

When you process the request, extract the tokens from the request header. Then call the **AntiForgery.Validate** method to validate the tokens. The **Validate** method throws an exception if the tokens are not valid.



```
void ValidateRequestHeader(HttpRequestMessage request)
{
    string cookieToken = "";
    string formToken = "";

    IEnumerable<string> tokenHeaders;
    if (request.Headers.TryGetValues("RequestVerificationToken", out tokenHeaders))
    {
        string[] tokens = tokenHeaders.First().Split(':');
        if (tokens.Length == 2)
        {
            cookieToken = tokens[0].Trim();
            formToken = tokens[1].Trim();
        }
    }
    AntiForgery.Validate(cookieToken, formToken);
}
```

Working with SSL in Web API

Several common authentication schemes are not secure over plain HTTP. In particular, Basic authentication and forms authentication send unencrypted credentials. To be secure, these authentication schemes *must* use SSL. In addition, SSL client certificates can be used to authenticate clients.

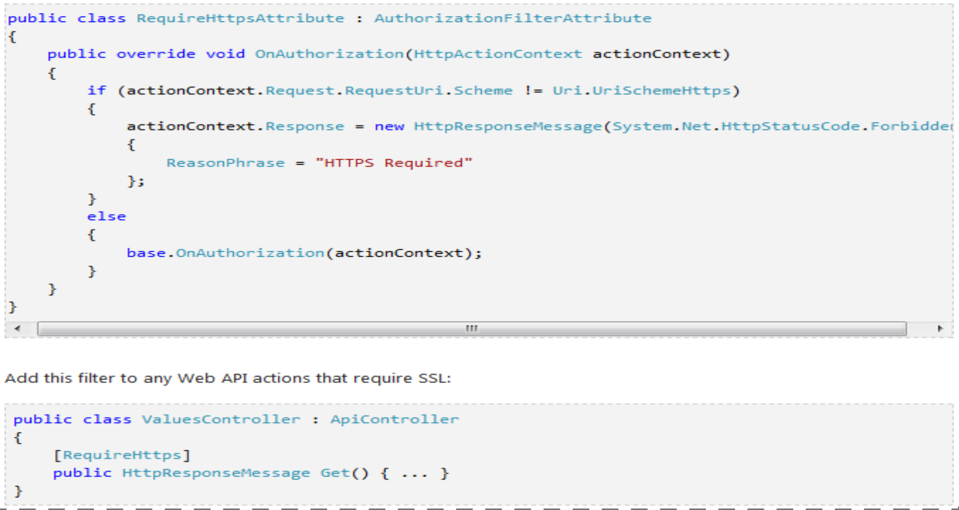
Enabling SSL on the Server

- Create or get a certificate. For testing, you can create a self-signed certificate.
- Add an HTTPS binding.

For local testing, you can enable SSL in IIS Express from Visual Studio. In the Properties window, set **SSL Enabled** to **True**. Note the value of **SSL URL**; use this URL for testing HTTPS connections

Enforcing SSL in a Web API Controller

If you have both an HTTPS and an HTTP binding, clients can still use HTTP to access the site. You might allow some resources to be available through HTTP, while other resources require SSL. In that case, use an action filter to require SSL for the protected resources. The following code shows a Web API authentication filter that checks for SSL:



```
public class RequireHttpsAttribute : AuthorizationFilterAttribute
{
    public override void OnAuthorization(HttpActionContext actionContext)
    {
        if (actionContext.Request.RequestUri.Scheme != Uri.UriSchemeHttps)
        {
            actionContext.Response = new HttpResponseMessage(System.Net.HttpStatusCode.Forbidde
            {
                ReasonPhrase = "HTTPS Required"
            };
        }
        else
        {
            base.OnAuthorization(actionContext);
        }
    }
}
```

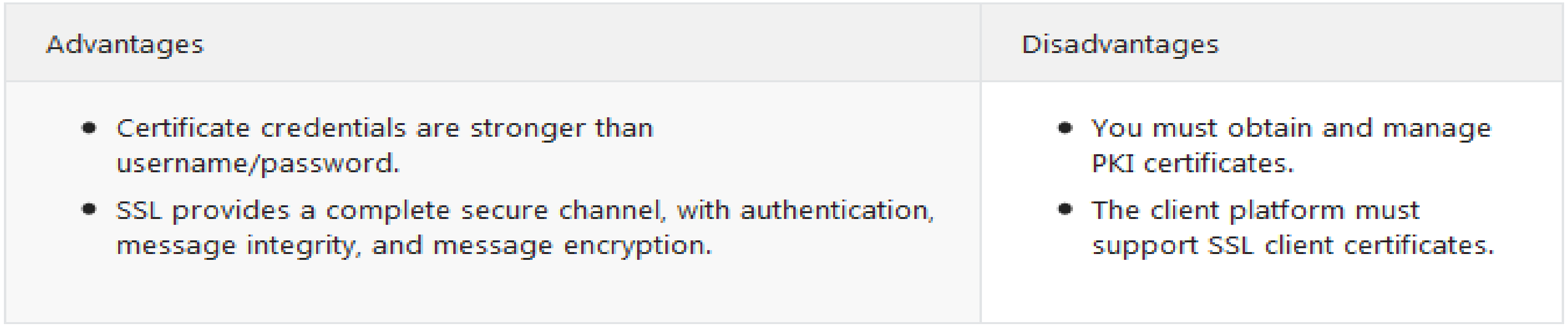
Add this filter to any Web API actions that require SSL:

```
public class ValuesController : ApiController
{
    [RequireHttps]
    public HttpResponseMessage Get() { ... }
}
```

Abhishek Sharma

SSL Client Certificates

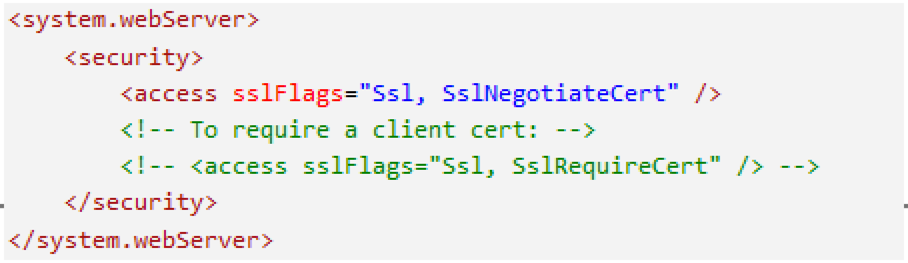
SSL provides authentication by using Public Key Infrastructure certificates. The server must provide a certificate that authenticates the server to the client. It is less common for the client to provide a certificate to the server, but this is one option for authenticating clients. To use client certificates with SSL, you need a way to distribute signed certificates to your users. For many application types, this will not be a good user experience, but in some environments (for example, enterprise) it may be feasible.



Advantages	Disadvantages
• Certificate credentials are stronger than username/password. • SSL provides a complete secure channel, with authentication, message integrity, and message encryption.	• You must obtain and manage PKI certificates. • The client platform must support SSL client certificates.

To configure IIS to accept client certificates, open IIS Manager and perform the following steps:

- Click the site node in the tree view.
- Double-click the **SSL Settings** feature in the middle pane.
- Under **Client Certificates**, select one of these options:
    - **Accept**: IIS will accept a certificate from the client, but does not require one.
    - **Require**: Require a client certificate. (To enable this option, you must also select "Require SSL")
- You can also set these options in the ApplicationHost.config file:



```
<system.webServer>
    <security>
        <access sslFlags="Ssl, SslNegotiateCert" />
        <!-- To require a client cert: -->
        <!-- <access sslFlags="Ssl, SslRequireCert" /> -->
    </security>
</system.webServer>
```

SSL Client Certificates

For testing purposes, you can use MakeCert.exe to create a client certificate. First, create a test root authority:

```
makecert.exe -n "CN=Development CA" -r -sv TempCA.pvk TempCA.cer
```

Makecert will prompt you to enter a password for the private key.
Next, add the certificate to the test server's "Trusted Root Certification Authorities" store, as follows:

- Open MMC.
- Under **File**, select **Add/Remove Snap-In**.
- Select **Computer Account**.
- Select **Local computer** and complete the wizard.
- Under the navigation pane, expand the "Trusted Root Certification Authorities" node.
- On the **Action** menu, point to **All Tasks**, and then click **Import** to start the Certificate Import Wizard.
- Browse to the certificate file, TempCA.cer.
- Click **Open**, then click **Next** and complete the wizard. (You will be prompted to re-enter the password.)
- Now create a client certificate that is signed by the first certificate:

```
makecert.exe -pe -ss My -sr CurrentUser -a sha1 -sky exchange -n "CN=name"
     -eku 1.3.6.1.5.5.7.3.2 -sk SignedByCA -ic TempCA.cer -iv TempCA.pvk
```

Using Client Certificates in Web API

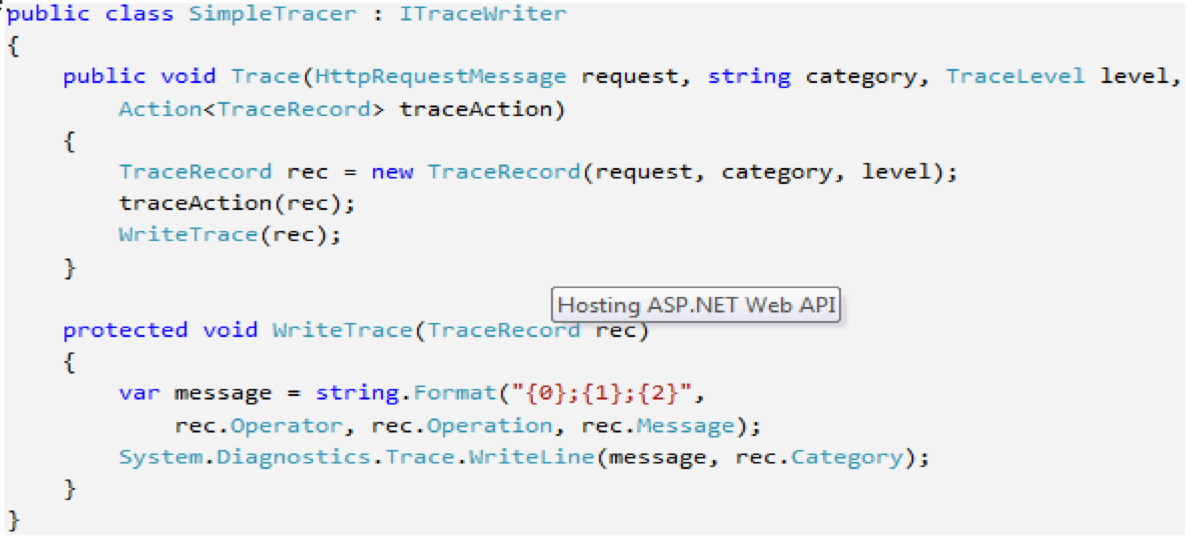
On the server side, you can get the client certificate by calling GetClientCertificate on the request message. The method returns null if there is no client certificate. Otherwise, it returns an **X509Certificate2** instance. Use this object to get information from the certificate, such as the issuer and subject. Then you can use this information for authentication and/or authorization.

```
X509Certificate2 cert = Request.GetClientCertificate();
string issuer = cert.Issuer;
string subject = cert.Subject;
```

Tracing

Tracing in ASP.NET Web API

- When you are trying to debug a web-based application, there is no substitute for a good set of trace logs.
- You can use this feature to trace what the Web API framework does before and after it invokes your controller. You can also use it to trace your own code.
- ASP.NET Web API is designed so that you can use your choice of tracing/logging library, whether that is ETW, NLog, log4net, or simply **System.Diagnostics.Trace**. To collect traces, implement the **ITraceWriter** interface. Here is a simple example:



```
public class SimpleTracer : ITraceWriter
{
    public void Trace(HttpRequestMessage request, string category, TraceLevel level,
        Action<TraceRecord> traceAction)
    {
        TraceRecord rec = new TraceRecord(request, category, level);
        traceAction(rec);
        WriteTrace(rec);
    }

    protected void WriteTrace(TraceRecord rec)
    {
        var message = string.Format("{0};{1};{2}",
            rec.Operator, rec.Operation, rec.Message);
        System.Diagnostics.Trace.WriteLine(message, rec.Category);
    }
}
```

The **ITraceWriter.Trace** method creates a trace. The caller specifies a category and trace level. The category can be any user-defined string. Your implementation of **Trace** should do the following:

- Create a new **TraceRecord**. Initialize it with the request, category, and trace level, as shown. These values are provided by the caller.
- Invoke the *traceAction* delegate. Inside this delegate, the caller is expected to fill in the rest of the **TraceRecord**.
- Write the **TraceRecord**, using any logging technique that you like. The example shown here simply calls into **System.Diagnostics.Trac**

Tracing in ASP.NET Web API

Setting the Trace Writer

To enable tracing, you must configure Web API to use your **ITraceWriter** implementation. You do this through the **HttpConfiguration** object, as shown in the following code:

```
public static void Register(HttpConfiguration config)
{
    config.Services.Replace(typeof(ITraceWriter), new SimpleTracer());
}
```

If you are hosting in ASP.NET, do this inside the **Application_Start** method, defined in Global.asax. Only one trace writer can be active. By default, Web API sets a "no-op" tracer that does nothing. Call **HttpConfiguration.Services.Replace** to replace this default tracer with your own. (The "no-op" tracer exists so that tracing code does not have to check whether the trace writer is **null** before writing a trace.)

Adding Traces to Your Code

Adding a trace writer gives you immediate access to the traces created by the Web API pipeline. You can also use the trace writer to trace your own code:

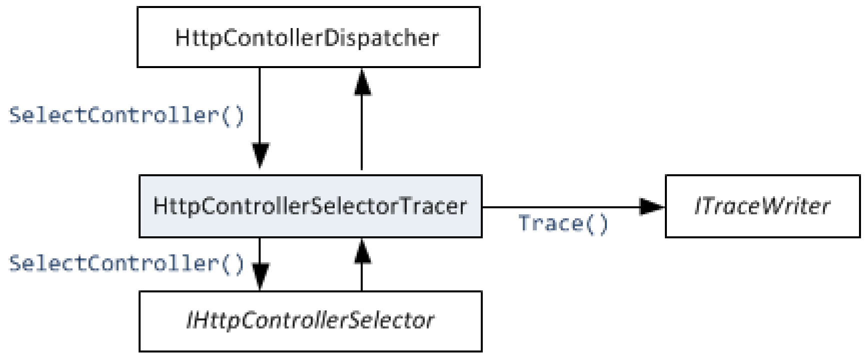
```
using System.Web.Http.Tracing;

public class ProductsController : ApiController
{
    public HttpResponseMessage GetAllProducts()
    {
        Configuration.Services.GetTraceWriter().Info(
            Request, "ProductsController", "Get the list of products.");

        // ...
    }
}
```

Get the trace writer by calling **HttpConfiguration.Services.GetTraceWriter**. From a controller, this method is accessible through the **ApiController.Configuration** property.

To write a trace, you can call the **ITraceWriter.Trace** method directly, but the **ITraceWriterExtensions** class defines some extension methods that are more friendly. For example, the **Info** method creates a trace with trace level **Info**.

How Web API Tracing Works ?

Tracing in Web API uses a in Web API uses a *facade* pattern: When tracing is enabled, Web API wraps various parts of the request pipeline with classes that perform trace calls. E.g. when selecting a controller, the pipeline uses the **IHttpControllerSelector** interface. With tracing enabled, the pipeline inserts a class that implements **IHttpControllerSelector** but calls through to the real implementation:



The benefits of this design include:

- If you do not add a trace writer, the tracing components are not instantiated and have no performance impact.
- If you replace default services such as **IHttpControllerSelector** with your own custom implementation, tracing is not affected, because tracing is done by the wrapper object.

You can also replace the entire Web API trace framework with your own custom framework, by replacing the default **ITraceManager** service:

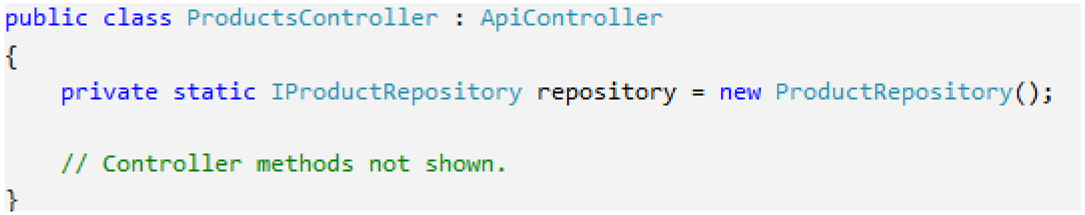
```
config.Services.Replace(typeof(ITraceManager), new MyTraceManager());
```

Implement **ITraceManager::Initialize** to initialize your tracing system. Be aware that this replaces the *entire* trace framework, including all of the tracing code that is built into Web API

Extensibility

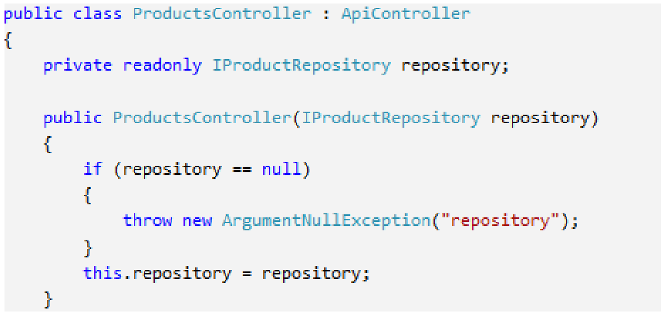
Using the Web API Dependency Resolver

- A dependency is an object or interface that another object requires. E.g. If we defined a ProductsController class that requires an IProductRepository instance. The implementation looked like this:



```
public class ProductsController : ApiController
{
    private static IProductRepository repository = new ProductRepository();

    // Controller methods not shown.
}
```

- This is not the best design, because the call to new the ProductRepository is hard-coded into the controller class. To use a different implementation of IProductRepository, we would need to change the code in ProductRepository. It is better if the ProductsController is not tied to any concrete instance of IProductRepository.
- Dependency injection addresses this problem. With dependency injection, an object is not responsible for creating its own dependencies. Instead, you inject the dependency when you create the object, through a constructor parameter or a setter method. Here is a revised implementation of ProductsController:



```
public class ProductsController : ApiController
{
    private readonly IProductRepository repository;

    public ProductsController(IProductRepository repository)
    {
        if (repository == null)
        {
            throw new ArgumentNullException("repository");
        }
        this.repository = repository;
    }
```

- This is better. Now can switch to another IProductRepository instance without touching the implementation of ProductsController.
- In ASP.NET Web API, you do not create the controller directly. Instead, the framework creates the controller for you, and the framework does not know anything about IProductRepository. The framework can only create your controller by calling a parameterless constructor.

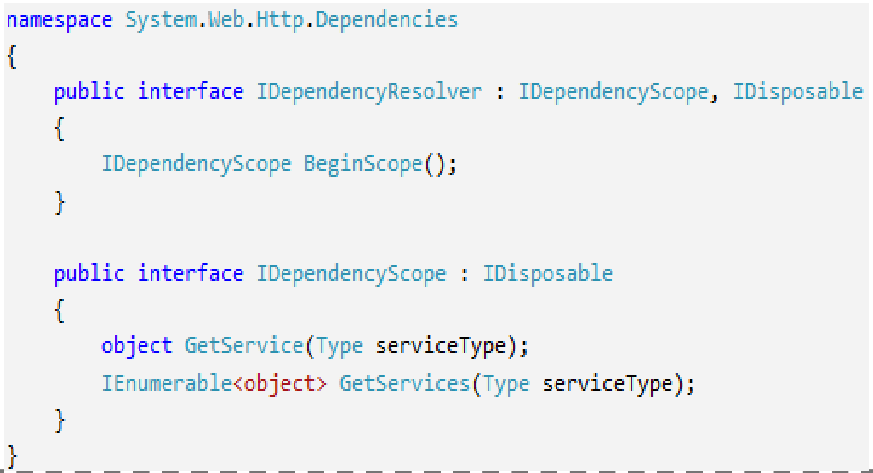
Using the Web API Dependency Resolver

This is where the dependency resolver comes in. The job of the dependency resolver is to create objects that the framework requires at run time, including controllers. By providing a custom dependency resolver, you can create controller instances on behalf of the framework.

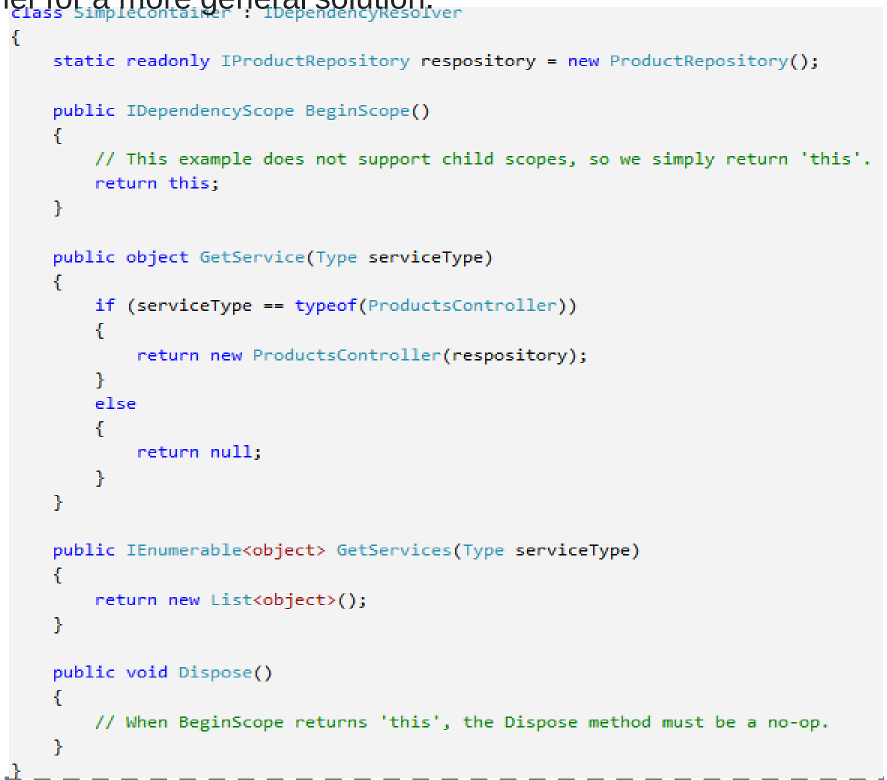
A Simple Dependency Resolver

The following code shows a simple dependency resolver. This code is mainly just to show how dependency injection works in Web API. Later, we'll see how to incorporate an IoC container for a more general solution.

A dependency resolver implements the **IDependencyResolver** interface. The **IDependencyResolver** interface inherits two other interfaces, **IDependencyScope** and **IDisposable**:



```
namespace System.Web.Http.Dependencies
{
    public interface IDependencyResolver : IDependencyScope, IDisposable
    {
        IDependencyScope BeginScope();
    }

    public interface IDependencyScope : IDisposable
    {
        object GetService(Type serviceType);
        IEnumerable<object> GetServices(Type serviceType);
    }
}
```



```
class SimpleContainer : IDependencyResolver
{
    static readonly IProductRepository respository = new ProductRepository();

    public IDependencyScope BeginScope()
    {
        // This example does not support child scopes, so we simply return 'this'.
        return this;
    }

    public object GetService(Type serviceType)
    {
        if (serviceType == typeof(ProductsController))
        {
            return new ProductsController(respository);
        }
        else
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return new List<object>();
    }

    public void Dispose()
    {
        // When BeginScope returns 'this', the Dispose method must be a no-op.
    }
}
```

Using the Web API Dependency Resolver

The **IDependencyScope** interface defines two methods:

- **GetService**: Creates one instance of a specified type.
- **GetServices**: Create a collection of objects of a specified type.

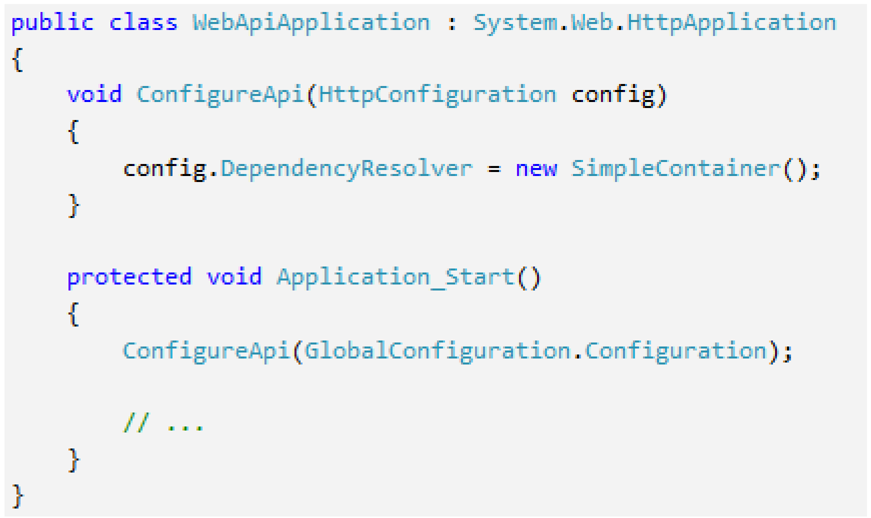
For controllers, the framework calls **GetService** to get a single instance of the controller. This is where our simple container creates the controller and injects the repository. For any types that your dependency resolver does not handle, **GetService** should return **null**, and **GetServices** should return an empty collection object. In particular, don't throw an exception for unknown types.

The **IDependencyResolver** interface inherits **IDependencyScope** and adds one method:
- **BeginScope**: Creates a nested scope.

Setting the Dependency Resolver

- In Solution Explorer, double-click Global.asax. Visual Studio will open the file named Global.asax.cs file, which is the code-behind file for Global.asax. This file contains code for handling application-level and session-level events in ASP.NET.

- Inside the **Application_Start** method, set **GlobalConfiguration.Configuration.DependencyResolver** to your dependency resolver:



```
public class WebApiApplication : System.Web.HttpApplication
{
    void ConfigureApi(HttpConfiguration config)
    {
        config.DependencyResolver = new SimpleContainer();
    }

    protected void Application_Start()
    {
        ConfigureApi(GlobalConfiguration.Configuration);

        // ...
    }
}
```

Using the Web API Dependency Resolver

Scope and Object Lifetime

- Controllers are created per request. To help manage object lifetimes, **IDependencyResolver** uses the **IDisposable** interface.
- The dependency resolver attached to the **HttpConfiguration** object has global scope.
- When the framework creates a new instance of a controller, it calls **IDependencyResolver.BeginScope**. This method returns an **IDependencyScope**. The framework calls **GetService** on the **IDependencyScope** instance to get the controller. When the framework is done processing the request, it calls **Dispose** on the child scope. You can use the **Dispose** method to dispose of the controller's dependencies.

Dependency Injection with IoC Containers

- An IoC container is a software component that is responsible for creating dependencies. IoC containers provide a general framework for dependency injection. If you use an IoC container, then you don't need to wire up objects directly in code. Several open-source .Net IoC containers are available, such as Autofac, Castle Windsor, Ninject, Spring.NET, StructureMap, and others.

API

References and Source

- MSDN
- Code Project
- Pluralsight