

API

<http://www.webdevelopmenthelp.net/2014/05/asp-net-web-api-interview-questions.html>

Difference between WCF and Web API and WCF REST and Web Service

Web Service

1. It is based on SOAP and return data in XML form.
2. It support only HTTP protocol.
3. It is not open source but can be consumed by any client that understands xml.
4. It can be hosted only on IIS.

WCF

1. It is also based on SOAP and return data in XML form.
2. It is the evolution of the web service(ASMX) and support various protocols like TCP, HTTP, HTTPS, Named Pipes, MSMQ.
3. The main issue with WCF is, its tedious and extensive configuration.
4. It is not open source but can be consumed by any client that understands xml.
5. It can be hosted with in the applicaion or on IIS or using window service.

WCF Rest

1. To use WCF as WCF [Rest service](#) you have to enable webHttpBindings.
2. It support HTTP GET and POST verbs by [WebGet] and [WebInvoke] attributes respectively.
3. To enable other HTTP verbs you have to do some configuration in IIS to accept request of that particular verb on .svc files

4. Passing data through parameters using a WebGet needs configuration. The UriTemplate must be specified
5. It support XML, JSON and ATOM data format.

Web API

1. This is the new framework for building HTTP services with easy and simple way.
2. Web API is open source an ideal platform for building REST-ful services over the .NET Framework.
3. Unlike WCF Rest service, it use the full feates of HTTP (like URIs, request/response headers, caching, versioning, various content formats)
4. It also supports the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection, unit testing that makes it more simple and robust.
5. It can be hosted with in the application or on IIS.
6. It is light weight architecture and good for devices which have limited bandwidth like smart phones.
7. Responses are formatted by Web API's MediaTypeFormatter into JSON, XML or whatever format you want to add as a MediaTypeFormatter.

To whom choose between WCF or WEB API

1. Choose WCF when you want to create a service that should support special scenarios such as one way messaging, message queues, duplex communication etc.
2. Choose WCF when you want to create a service that can use fast transport channels when available, such as TCP, Named Pipes, or maybe even UDP (in WCF 4.5), and you also want to support HTTP when all other transport channels are unavailable.
3. Choose Web API when you want to create a resource-oriented services over HTTP that can use the full features of HTTP (like URIs, request/response headers, caching, versioning, various content formats).
4. Choose Web API when you want to expose your service to a broad range of clients including browsers, mobiles, iphone and tablets.

1) What is Web API?

It is a framework which helps us to build/develop HTTP services. So there will be a client server communication using HTTP protocol.

2) What is Representational state transfer or REST?

REST is architectural style, which has defined guidelines for creating services which are scalable. REST used with HTTP protocol using its verbs GET, POST, PUT and DELETE.

3) Explain Web API Routing?

Routing is the mechanism of pattern matching as we have in MVC. These routes will get registered in Route Tables. Below is the sample route in Web API –

```
Routes.MapHttpRoute(  
    Name: "MyFirstWebAPIRoute",  
    routeTemplate: "api/{controller}/{id}"  
    defaults: new { id = RouteParameter.Optional }  
);
```

4) List out the differences between WCF and Web API?

WCF

- It is framework build for building or developing service oriented applications.
- WCF can be consumed by clients which can understand XML.
- WCF supports protocols like – HTTP, TCP, Named Pipes etc.

Web API

- It is a framework which helps us to build/develop HTTP services
- Web API is an open source platform.
- It supports most of the MVC features which keep Web API over WCF.

5) What are the advantages of using REST in Web API?

REST always used to make less data transfers between client and server which makes REST an ideal for using it in mobile apps. Web API supports HTTP protocol thereby it reintroduces the old way of HTTP verbs for communication.

6) Difference between WCF Rest and Web API?

WCF Rest

- “WebHttpBinding” to be enabled for WCF Rest.
- For each method there has to be attributes like – “WebGet” and “WebInvoke”
- For GET and POST verbs respectively.

Web API

- Unlike WCF Rest we can use full features of HTTP in Web API.
- Web API can be hosted in IIS or in application.

7) List out differences between MVC and Web API?

Below are some of the differences between MVC and Web API

MVC

- MVC is used to create a web app, in which we can build web pages.
- For JSON it will return JsonResult from action method.
- All requests are mapped to the respective action methods.

Web API

- This is used to create a service using HTTP verbs.
- This returns XML or JSON to client.
- All requests are mapped to actions using HTTP verbs.

8) What are the advantages of Web API?

Below are the list of support given by Web API –

- OData
- Filters

- Content Negotiation
- Self Hosting
- Routing
- Model Bindings

9) Can we unit test Web API?

Yes we can unit test Web API.

10) How to unit test Web API?

We can unit test the Web API using Fiddler tool. Below are the settings to be done in Fiddler –

Compose Tab -> Enter Request Headers -> Enter the Request Body and execute

11) Can we return view from Web API?

No. We cannot return view from Web API.

12) How we can restrict access to methods with specific HTTP verbs in Web API?

Attribute programming is used for this functionality. Web API will support to restrict access of calling methods with specific HTTP verbs. We can define HTTP verbs as attribute over method as shown below

```
[HttpPost]

public void UpdateTestCustomer(Customer c)

{

    TestCustomerRepository.AddCustomer(c);

}
```

13) Can we use Web API with ASP.NET Web Forms?

Yes. We can use Web API with ASP.NET Webforms.

14) List out the steps to be made for Web API to work in Web Forms?

Below are the steps to be followed –

- Creating new controller for Web API.

- Adding routing table to “Application_Start” method in Global.asax
- Make a AJAX call to Web API actions.

15) Explain how to give alias name for action methods in Web API?

Using attribute “ActionName” we can give alias name for Web API actions. Eg:

```
[HttpPost]
[ActionName("AliasTestAction")]
public void UpdateTestCustomer(Customer c)
{
    TestCustomerRepository.AddCustomer(c);
}
```

16) What is the difference between MVC Routing and Web API Routing?

There should be atleast one route defined for MVC and Web API to run MVC and Web API application respectively. In Web API pattern we can find “api/” at the beginning which makes it distinct from MVC routing. In Web API routing “action” parameter is not mandatory but it can be a part of routing.

17) Explain Exception Filters?

Exception filters will be executed whenever controller methods (actions) throws an exception which is unhandled. Exception filters will implement “IExceptionFilter” interface.

18) Explain about the new features added in Web API 2.0 version?

Below are the list of features introduced in Web API 2.0 –

- OWIN
- Attribute Routing
- External Authentication
- Web API OData

19) How can we pass multiple complex types in Web API?

Below are the methods to pass the complex types in Web API –

- Using ArrayList
- Newtonsoft JArray

20) Write a code snippet for passing arraylist in Web API?

Below is the code snippet for passing arraylist –

```
ArrayList paramList = new ArrayList();

Category c = new Category { CategoryId = 1, CategoryName = "SmartPhones"};
Product p = new Product { ProductId = 1, Name = "Iphone", Price = 500, CategoryID = 1 };

paramList.Add(c);
paramList.Add(p);
```

21) Give an example of Web API Routing?

Below is the sample code snippet to show Web API Routing –

```
config.Routes.MapHttpRoute(
    name: "MyRoute", //route name
    routeTemplate: "api/{controller}/{action}/{id}", //as you can see "api" is at the beginning.
    defaults: new { id = RouteParameter.Optional }
);
```

22) Give an example of MVC Routing?

Below is the sample code snippet to show MVC Routing –

```
routes.MapRoute(
    name: "MyRoute", //route name
    url: "{controller}/{action}/{id}", //route pattern
    defaults: new
```

```
{  
    controller = "a4academicsController",  
    action = "a4academicsAction",  
    id = UrlParameter.Optional  
}  
);
```

23) How we can handle errors in Web API?

Below are the list of classes which can be used for error handling -

- `HttpResponseException`
- `Exception Filters`
- `Registering Exception Filters`
- `HttpError`

24) Explain how we can handle error from “`HttpResponseException`”?

This returns the HTTP status code what you specify in the constructor. Eg :

```
public TestClass MyTestAction(int id)  
{  
    TestClass c = repository.Get(id);  
    if (c == null)  
    {  
        throw new HttpResponseException(HttpStatusCode.NotFound);  
    }  
    return c;  
}
```

25) How to register Web API exception filters?

Below are the options to register Web API exception filters –

- From Action
- From Controller
- Global registration

26) Write a code snippet to register exception filters from action?

Below is the code snippet for registering exception filters from action –

```
[NotImplementedExceptionFilter]

public TestCustomer GetMyTestCustomer(int custid)

{

    //Your code goes here

}
```

27) Write a code snippet to register exception filters from controller?

Below is the code snippet for registering exception filters from controller –

```
[NotImplementedExceptionFilter]

public class TestCustomerController : Controller

{

    //Your code goes here

}
```

28) Write a code snippet to register exception filters globally?

Below is the code snippet for registering exception filters globally –

```
GlobalConfiguration.Configuration.Filters.Add( new

MyTestCustomerStore.NotImplementedExceptionFilterAttribute());
```

29) How to handle error using HttpError?

HttpException will be used to throw the error info in response body. "CreateErrorResponse" method is used along with this, which is an extension method defined in "HttpRequestMessageExtensions".

30) Write a code snippet to show how we can return 404 error from HttpError?

Below is the code snippet for returning 404 error from HttpError –

```
string message = string.Format("TestCustomer id = {0} not found", customerid);  
return Request.CreateErrorResponse(HttpStatusCode.NotFound, message);
```

31) How to enable tracing in Web API?

To enable tracing place below code in –"Register" method of WebAPIConfig.cs file.

```
config.EnableSystemDiagnosticsTracing();
```

32) Explain how Web API tracing works?

Tracing in Web API done in façade pattern i.e, when tracing for Web API is enabled, Web API will wrap different parts of request pipeline with classes, which performs trace calls.

33) Can we unit test Web API?

Yes we can unit test Web API.

34) Explain Authentication in Web API?

Web API authentication will happen in host. In case of IIS it uses Http Modules for authentication or we can write custom Http Modules. When host is used for authentication it used to create principal, which represent security context of the application.

35) Explain ASP.NET Identity?

This is the new membership system for ASP.NET. This allows to add features of login in our application.

Below are the list of features supported by ASP.NET Identity in Web API –

- One ASP.NET Identity System
- Persistence Control

36) What are Authentication Filters in Web API?

Authentication Filter will let you set the authentication scheme for actions or controllers. So this way our application can support various authentication mechanisms.

37) How to set the Authentication filters in Web API?

Authentication filters can be applied at the controller or action level. Decorate attribute – "IdentityBasicAuthentication" over controller where we have to set the authentication filter.

38) Explain method – "AuthenticateAsync" in Web API?

"AuthenticateAsync" method will create "IPrincipal" and will set on request. Below is the sample code snippet for "AuthenticateAsync" –

```
Task AuthenticateAsync(  
    HttpContext mytestcontext,  
    CancellationToken mytestcancellationToken  
)
```

39) How to set the Error Result in Web API?

Below is the sample code to show how to set error result in Web API –

```
HttpResponseMessage myresponse = new  
    HttpResponseMessage(HttpStatusCode.Unauthorized);  
myresponse.RequestMessage = Request;  
myresponse.ReasonPhrase = ReasonPhrase;
```

40) Explain method – "ChallengeAsync" in Web API?

"ChallengeAsync" method is used to add authentication challenges to response. Below is the method signature –

```
Task ChallengeAsync(  
    HttpContext mytestcontext,  
    CancellationToken mytestcancellationToken  
)
```

41) What are media types?

It is also called MIME, which is used to identify the data . In Html, media types is used to describe message format in the body.

42) List out few media types of HTTP?

Below are the list of media types –

- Image/Png
- Text/HTML
- Application/Json

43) Explain Media Formatters in Web API?

Media Formatters in Web API can be used to read the CLR object from our HTTP body and Media formatters are also used for writing CLR objects of message body of HTTP.

44) How to serialize read-only properties?

Read-Only properties can be serialized in Web API by setting the value “true” to the property –

“SerializeReadOnlyTypes” of class – “DataContractSerializerSettings”.

45) How to get Microsoft JSON date format ?

Use “DateFormatHandling” property in serializer settings as below –

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

46) How to indent the JSON in web API?

Below is the code snippet to make JSON indenting –

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

47) How to JSON serialize anonymous and weakly types objects?

Using “Newtonsoft.Json.Linq.JObject” we can serialize and deserialize weakly typed objects.

48) What is the use of “IgnoreDataMember” in Web API?

By default if the properties are public then those can be serialized and deserialized, if we does not want to serialize the property then decorate the property with this attribute.

49) How to write indented XML in Web API?

To write the indented xml set “Indent” property to true.

50) How to set Per-Type xml serializer?

We can use method – “SetSerializer”. Below is the sample code snippet for using it –

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

51) What is “Under-Posting” and “Over-Posting” in Web API?

- “Under-Posting” - When client leaves out some of the properties while binding then it’s called under – posting.
- “Over-Posting” – If the client sends more data than expected in binding then it’s called over-posting.

52) How to handle validation errors in Web API?

Web API will not return error to client automatically on validation failure. So its controller’s duty to check the model state and response to that. We can create a custom action filter for handling the same.

53) Give an example of creating custom action filter in Web API?

Below is the sample code for creating custom action filter –

```
public class MyCustomModelAttribute : ActionFilterAttribute  
{  
    public override void OnActionExecuting(HttpContext actionContext)  
    {  
        if (actionContext.ModelState.IsValid == false)
```

```
{  
    //Code goes here  
}  
  
}  
  
}
```

In case validation fails here it returns HTTP response which contains validation errors.

54) How to apply custom action filter in WebAPI.config?

Add a new action filter in “Register” method as shown -

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

55) How to set the custom action filter in action methods in Web API?

Below is the sample code of action with custom action filter –

```
public class MyCustomerTestController : ApiController  
{  
    [MyCustomModelAttribute]  
    public HttpResponseMessage Post(MyTestCustomer customer)  
    {  
        // ...  
    }  
}
```

```
}  
  
}
```

56) What is BSON in Web API?

It's a binary serialization format. "BSON" stands for "Binary JSON". BSON serializes objects to key-value pair as in JSON. Its light weight and its fast in encode/decode.

57) How to enable BSON in server?

Add "BsonMediaTypeFormatter" in WebAPI.config as shown below

```
public static class WebApiConfig  
{  
    public static void Register(HttpConfiguration config)  
    {  
        config.Formatters.Add(new BsonMediaTypeFormatter());  
  
        // Other Web API configuration goes here  
    }  
}
```

58) How parameter binding works in Web API?

Below are the rules followed by WebAPI before binding parameters –

- If it is simple parameters like – bool,int, double etc. then value will be obtained from the URL.
- Value read from message body in case of complex types.

59) Why to use "FromUri" in Web API?

In Web API to read complex types from URL we will use "FromUri" attribute to the parameter in action method. Eg:

```
public MyValuesController : ApiController  
{
```

```
public HttpResponseMessage Get([FromUri] MyCustomer c) { ... }  
}
```

60) Why to use “FromBody” in Web API?

This attribute is used to force Web API to read the simple type from message body. “FromBody” attribute is along with parameter. Eg:

```
public HttpResponseMessage Post([FromBody] int customerid, [FromBody] string  
customername) { ... }
```

61) Why to use “IValueprovider” interface in Web API?

This interface is used to implement custom value provider.

ASP.NET Web API Interview Questions List

- [Foundation: HTTP and REST Concepts](#)
 - Resources and URIs
 - HTTP Methods
 - HTTP Status Codes
 - HTTP Content
 - Internet Media Types
 - REST
 - JSON & XML
- [What is ASP.NET Web API?](#)
- [What are the advantages of using ASP.NET Web API?](#)
- [What new features are introduced in ASP.NET Web API 2.0?](#)
- [WCF Vs ASP.NET Web API?](#)
- [WCF REST Vs ASP.NET Web API?](#)
- [Is it true that ASP.NET Web API has replaced WCF?](#)
- [MVC Vs ASP.NET Web API?](#)
- [How to return View from ASP.NET Web API method?](#)
- [How to restrict access to Web API method to specific HTTP Verb?](#)
- [Can we use Web API with ASP.NET Web Form?](#)

More Web API Interview Questions

- [How we can provide an alias name for ASP.NET Web API action?](#)
- [What are Exception Filters? What are different ways to register exception filters?](#)
- [How we can guarantee that our Web API returns JSON data only?](#)

Before we start Web API Tutorial, test your skill by answering simple question?

You are working with WebDevTutorial as a Developer and designing an ASP.NET Web API application. You need to select an HTTP verb to allow blog administrators to moderate comments posted on Web Development Tutorial blog. Which HTTP verb should you use?

- A. GET
- B. POST
- C. DELETE
- D. PUT

For a complete online test and Practice Exams on Web Technologies, [Click Here](#).

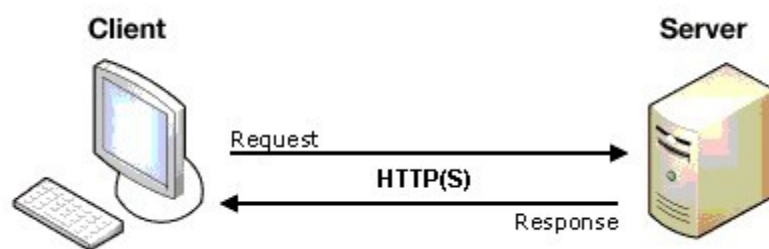
Correct Answer: D

Foundation: HTTP and REST Concept

- Resources and URIs
- HTTP Methods
- HTTP Status Codes
- HTTP Content
- Internet Media Types
- REST
- JSON & XML

HTTP (Hypertext Transfer Protocol) is an application level protocol that is used to deliver data such as html files, image files, query results, etc. on the World Wide Web. **HTTP is stateless protocol** means after one cycle of request and response the server forgets everything about the cycle, it considers another request from the same client as a new request from a new client.

Below is a simple Client-Server architecture of HTTP protocol.



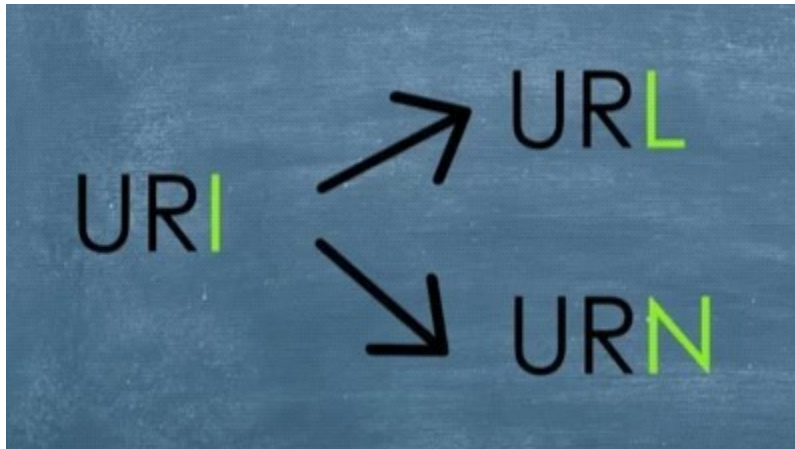
Resource and URIs:

URI stands for **Uniform Resource Identifier**. URI is a string of characters used to identify a resource on the internet either by location or by name, or by both.

URI is used to identify resources, For example, In real world assume there is a person named "Muhammad Ahmad" who lives in "2807, Badar Street, Dubai". We can find that person by name or by address or by both.

URI can be categorized into 2 as URL and URN

- **URL:** Uniform Resource Locator
- **URN:** Uniform Resource Name



URI is used to send requests to the server. It can be done with both URN and URL, but using URN is inefficient because there can be many resources with same name. So most commonly use method is URL and it consists of two required components; **The Protocol & The Domain.**

Look at the following URL:

<http://www.webdevelopmenthelp.net/asp-net-mvc>

Here the **red** colored part is the **protocol** and gold **color** part is the **domain**, other parts are optional. It may contain the port/path/query strings/fragments.

HTTP Methods:

There are multiple HTTP methods which can be used according to the user requirements. Those methods are case-sensitive and they must used in uppercase.

- **GET** – The GET method is used to retrieve information from the given server using a given URI.
- **HEAD** – Same as GET, but transfers the status line and header section only.
- **POST** – A POST request is used to send data to the server.
- **PUT** – Replaces all current resources with the uploaded content.
- **DELETE** – Removes all current resources given by a URI.
- **CONNECT** – Establishes a tunnel to the server identified by a given URI.
- **OPTIONS** – Describes the communication options for the target resource.
- **TRACE** – Performs a message loop-back test along the path to the target resource.

Among these methods GET/PUT/POST/DELETE are the most popular methods.

HTTP Status Codes:

HTTP Status Codes are the 3 digit integers which contain in server response. There is a meaning for each number. Followings are the set of status codes with its meaning:

- **1xx: Informational**-It means the request has been received and the process is continuing.
- **2xx: Success**-It means the action was successfully received, understood, and accepted.
- **3xx: Redirection**-It means further action must be taken in order to complete the request.
- **4xx: Client Error**-It means the request contains incorrect syntax or cannot be fulfilled
- **5xx: Server Error**-It means the server failed to fulfill an apparently valid request.

HTTP Content:

There are more contents in a HTTP header field. HTTP header fields are components of the header section of request or response messages in the Hypertext Transfer Protocol. It defines the operating parameters of an HTTP transaction.

For example, HTTP request to fetch ***webapi.htm*** page from the web server running on *webdevelopmenthelp.net*.



- 1 //Client request
- 2 GET /webapi.htm HTTP/1.1
- 3 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
- 4 Host: www.webdevelopmenthelp.net
- 5 Accept-Language: en-us
- 6 Accept-Encoding: gzip, deflate
- 7 Connection: Keep-Alive
- 8
- 9 //Server response

1
0

1
1 HTTP/1.1 200 OK

1 Date: Wed, 01 Feb 2017 12:28:53 GMT
2
Server: Apache/2.2.14 (Win32)
1
Last-Modified: Mon, 12 Feb 2017 19:15:56 GMT
3

Content-Length: 88
1
4 Content-Type: text/html

1 Connection: Closed
5

1
6

Internet Media Types:

It is a file identification mechanism on the MIME encoding system, formerly "MIME type", the Internet media type has become the de facto standard for identifying content on the Internet.

For example, an instance of receiving an email from a server with an attachment, the server embed the media type of the attachment in the message header. So the browser can launch appropriate helper application or plug-in.

REST:

REST (Representational State Transfer) is an architectural style for designing applications and it dictates to use HTTP for making calls for communications instead of complex mechanism like CORBA, RPC or SOAP.

There are few principles associated with REST architectural style:

- Everything is a resource i.e. File, Images, Video, WebPage etc.
- Every Resource is identified by a Unique Identifier.
- Use simple and Uniform Interfaces.

- Everything is done via representation (sending requests from client to server and receiving responses from server to client).
- Be Stateless- Every request should be an independent request.

JSON & XML:

XML stands for eXtensible Markup Language, it is a markup language like HTML and designed to store and transport data. XML doesn't do anything, but it store data in a specific format.

Following is a letter written from **Ahmad** to **Hamza** in XML format:



```
1 <Letter>
2     <to>Hamza</to>
3     <from>Ahmad</from>
4     <heading>Invitation</heading>
5     <body>I invite you to my birthday party in Dubai</body>
6 </Letter>
```

The above XML file represent the following message.



```
1 Letter
2 To: Hamza
3 From: Ahmad
4 Invitation
```

5 I invite you to my birthday party in Dubai

JSON (JavaScript Object Notation) also used to store and transport data. The above message can be transferred into JSON in following way:



```
//Invitation
1
2 {Letter:{to:'Hamza', from:'Ahmad', heading:'Invitation', body:'I invite you to my birthday party in
Dubai'}}}
```

Move on with ASP.NET Web API Interview Questions

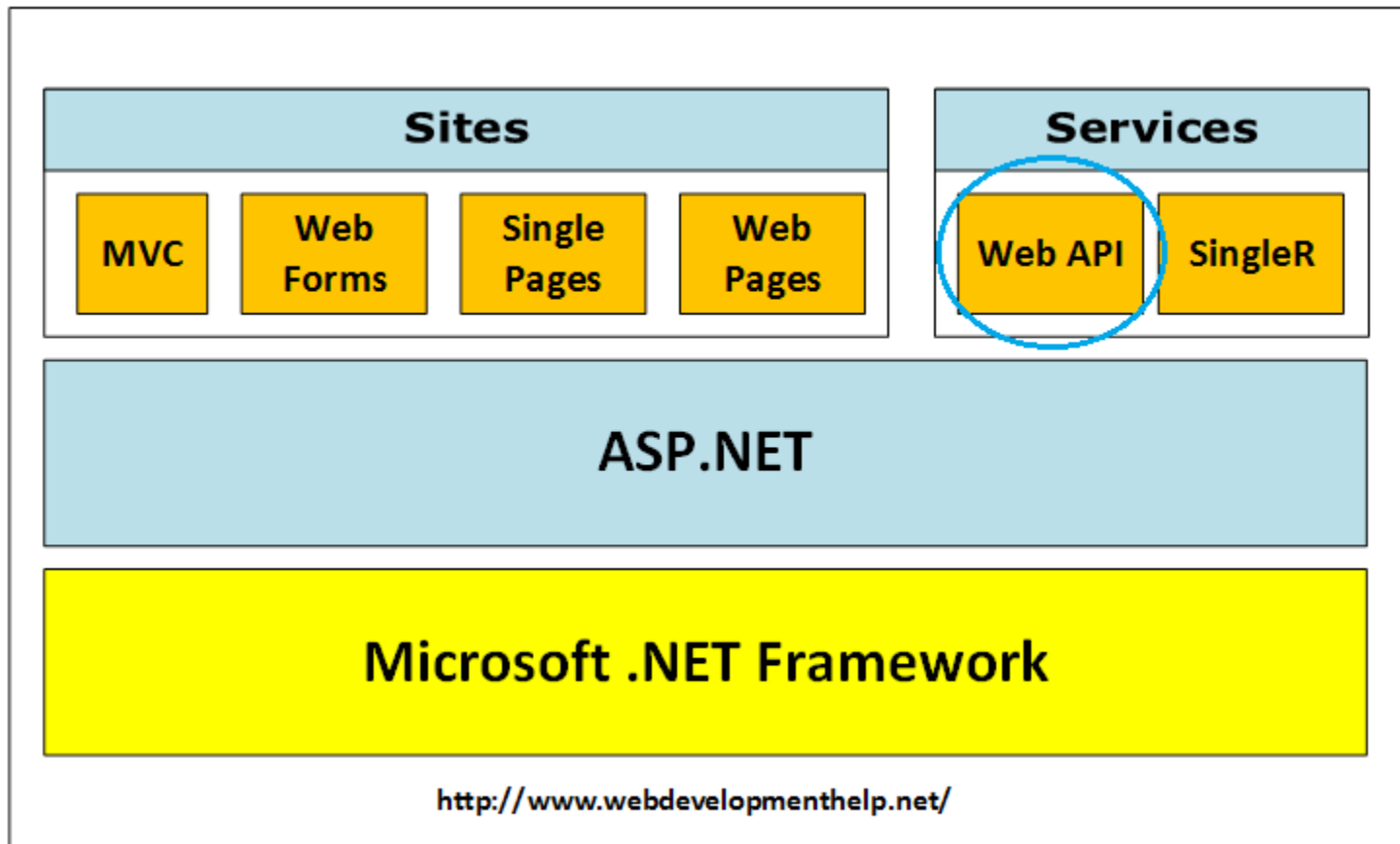
Now, we are well familiar with HTTP and REST Concepts, let's move on with ASP.NET Web API Interview Questions and Answers in details.

INTERVIEW QUESTIONS ASP.NET WEB API

What is ASP.NET Web API?

ASP.NET Web API is a framework that simplifies building HTTP services for broader range of clients (including browsers as well as mobile devices) on top of .NET Framework. Using ASP.NET Web API we can create non-SOAP based services like plain XML or JSON strings etc. with many other advantages including:

- Create resource-oriented services using the full features of HTTP.
- Exposing services to a variety of clients easily like browsers or mobile devices etc.



[Back to top](#)

What are the advantages of using ASP.NET Web API?

Using ASP.NET Web API has a number of advantages, but core of the advantages are:

- It works the HTTP way using standard HTTP verbs like GET, POST, PUT, DELETE etc for all CRUD operations.
- Complete support for routing.
- Response generated in JSON or XML format using MediaTypeFormatter.
- It has the ability to be hosted in IIS as well as self-host outside of IIS.
- Supports Model binding and Validation.
- Support for OData.
- and more....

For implementation on performing all CRUD operations using ASP.NET Web API, [click here](#).

Are you preparing for MCSD Certification Exam?

70-486: Developing ASP.NET MVC Web Applications

Take this professional exam to test your knowledge here...

[Back to top](#)

What new features are introduced in ASP.NET Web API 2.0?

More new features introduced in ASP.NET Web API framework v2.0 are as follows:

- Attribute Routing
- External Authentication
- CORS (Cross-Origin Resource Sharing)
- OWIN (Open Web Interface for .NET) Self Hosting
- IHttpActionResult
- Web API OData

You can follow a good Web API new feature details on [Top 5 New Features in ASP.NET Web API 2](#) here.

[Back to top](#)

Free Online Tests

- [C# Online Test](#)
- [ASP.NET Online Test](#)
- [ASP.NET MVC Online Test](#)
- [ASP.NET AJAX Online Test](#)
- [HTML5 Online Test](#)
- [jQuery Online Test](#)

WCF Vs ASP.NET Web API?

Actually, **Windows Communication Foundation** is designed to exchange standard SOAP-based messages using variety of transport protocols like HTTP, TCP, NamedPipes or MSMQ etc. On the other hand, **ASP.NET API** is a framework for building non-SOAP based services over HTTP only.

	SOAP-based WCF	Web API
Protocol	Multiple Protocol Support.	Limited to HTTP only.
Format	SOAP is the default format. SOAP message has a specific format.	Web API renders string (Plain XML or JSON string).
Size	Heavy weight as SOAP message have more than just data.	Light weight as only required information is passed.
Principles	Follows WS-* specification.	Follows REST principles.

[Back to top](#)

WCF RESTful Service Vs ASP.NET Web API?

Although both WCF REST and ASP.NET Web API follows the REST architecture but these have follow differences:

WCF REST

- Microsoft introduced "WebHttpBinding" to be used for creating WCF RESTful Services.
- HTTP Methods are mapped to attributes, for example, "WebGet" for GET method and "WebInvoke" for POST.

ASP.NET Web API

- As compared with WCF REST, Web API supports full features of HTTP.
- Its possible to host Web API in IIS as well as in an application.

WCF REST	ASP.NET Web API
<ul style="list-style-type: none"> • "WebHttpBinding" to be used for creating WCF RESTful Services. • HTTP Methods are mapped to attributes, for example, GET for "WebGet" and POST for "WebInvoke". 	<ul style="list-style-type: none"> • Web API supports full features of HTTP. • Its possible to host Web API in IIS as well as in an application.

[Back to top](#)

Is it true that ASP.NET Web API has replaced WCF?

It's a misconception that ASP.NET Web API has replaced WCF. It's another way of building non-SOAP based services, for example, plain XML or JSON string etc.

Yes, it has some added advantages like utilizing full features of HTTP and reaching more clients such as mobile devices etc.

But WCF is still a good choice for following scenarios:

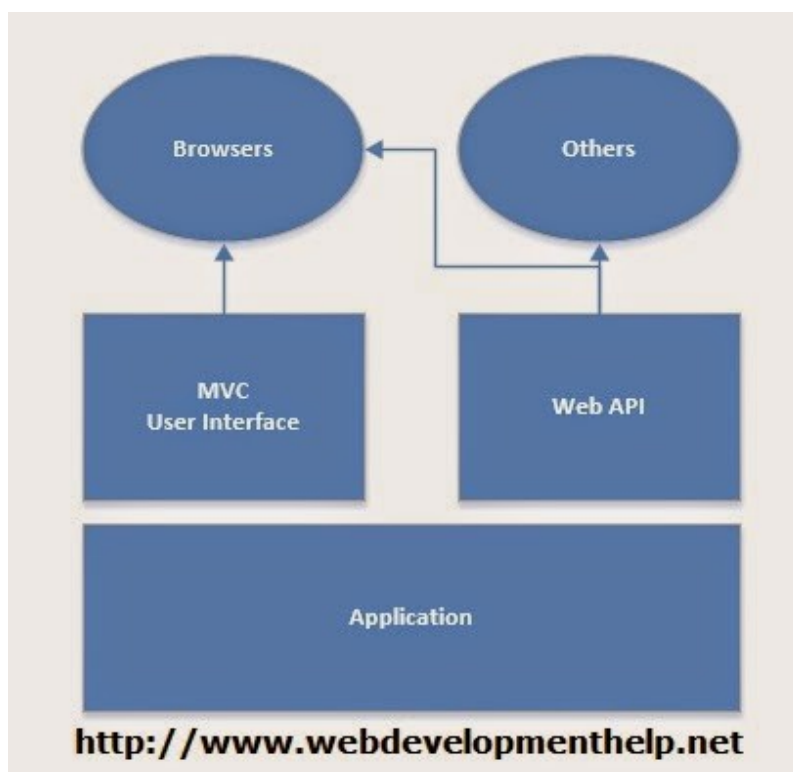
- If we intended to use transport other than HTTP e.g. TCP, UDP or Named Pipes.
- Message Queuing scenario using MSMQ.
- One-way communication or Duplex communication

A good understanding for WCF(Windows Communication Foundation), please follow [WCF Tutorial](#).

[Back to top](#)

MVC Vs ASP.NET Web API?

As in previous ASP.NET Web API Interview Questions, we discussed that purpose of Web API framework is to generate HTTP services that reaches more clients by generating data in raw format, for example, plain XML or JSON string. So, ASP.NET Web API creates simple HTTP services that renders raw data. On the other hand, ASP.NET MVC framework is used to develop web applications that generates Views as well as data. ASP.NET MVC facilitates in rendering HTML easy.



For **ASP.NET MVC Interview Questions**, [follow the link](#).

[Back to top](#)

Answer this simple Question to test your skill?

Please look into following piece of code:

```
[ActionName("GetById")]

public ActionResult GetEmployeeById()
{
    //Code logic here.

    return View();
}
```

What's true about the above code.

- A. "GetEmployeeById" action method with be identified and called by name "GetEmployeeById".
- B. "GetEmployeeById" action method with be identified and called by name "GetById".
- C. Above code will generate an error because of wrong return type.
- D. Action method can't be called because of duplicate action method names.

For a complete online test and Practice Exams on Web Technologies, [Click Here](#).

Correct Answer: B

How to return View from ASP.NET Web API method?

(A tricky Interview Question) No, we can't return view from ASP.NET Web API Method. As we discussed in earlier interview question about difference between ASP.NET MVC and Web API that ASP.NET Web API creates HTTP services that renders raw data. Although, it's quite possible in ASP.NET MVC application.

[Back to top](#)

How to restrict access to Web API method to specific HTTP Verb?

Attribute programming plays it's role here. We can easily restrict access to an ASP.NET Web API method to be called using a specific HTTP method. For example, we may required in a scenario to restrict access to a Web API method through HTTP POST only as follows:

```
[HttpPost]
public void UpdateStudent(Student aStudent)
{
    StudentRepository.AddStudent(aStudent);
}
```

[Back to top](#)

Can we use Web API with ASP.NET Web Form?

Yes, ASP.NET Web API is bundled with ASP.NET MVC framework but still it can be used with ASP.NET Web Form. It can be done in three simple steps as follows:

1. Create a Web API Controller.
2. Add a routing table to Application_Start method of Global.asax.
3. Make a jQuery AJAX Call to Web API method and get data.

[jQuery call to Web API](#) for all CRUD (Create, Retrieve, Update, Delete) operations can be [found here](#).

[Back to top](#)

How we can provide an alias name for ASP.NET Web API action?

We can provide an alias name for ASP.NET Web API action same as in case of ASP.NET MVC by using "ActionName" attribute as follows:

```
[HttpPost]
[ActionName("SaveStudentInfo")]
public void UpdateStudent(Student aStudent)
{
    StudentRepository.AddStudent(aStudent);
}
```

[ASP.NET Web API 2 Hands On](#)

Building RESTful Web Service using ASP.NET Web API2 taking from beginner level and learn how to use the new features of Web API2 i.e. Attribute Routing etc. [More Details](#)

[Back to top](#)

What are Exception Filters? What are different ways to register exception filters?

Exception Filter is basically a class that implements IExceptionHandler interface. While working with ASP.NET Web API, there can be scenarios where the code can generate unhandled exceptions. And for those unhandled exceptions, client will be receiving same generic error i.e. "Internal Server Error". In order to tackle such unhandled exceptions, Exception Filters can be used.

You can follow here for a detailed article on [Exception Handling in ASP.NET Web API](#) with following implementation:

- What are Exception Filters in Web API?
- How we can create a Custom Exception Filter?
- How we can register Custom Exception Filter at different levels?

We can register exception filters for ASP.NET Web API in following different levels:

- *Register Exception Filter from action*



```
1 [MyCustomExceptionFilter]
```

```
2 public Student Get(string id)
```

```
3 {
```

```
4     return StudentRepository.GetStudent(id);
```

```
5 }
```

- *Register Exception Filter from Controller*



```
1 [MyCustomExceptionFilter]
```

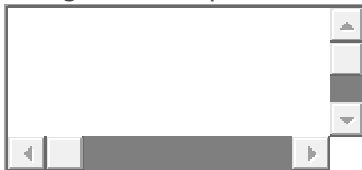
```
2 public class StudentsController : ApiController
```

```
3 {
```

```
4     //Controller detailed code.
```

```
5 }
```

- *Register Exception Filter globally*



```
1 CRUDWebAPI.MyCustomExceptionFilter ctrlr = new CRUDWebAPI.MyCustomExceptionFilter();
```

```
2 GlobalConfiguration.Configuration.Filters(ctrlr);
```

[Back to top](#)

How we can guarantee that our Web API returns JSON data only?

In order to ensure that our ASP.NET Web API serialize the returning object to JSON format and returns JSON data only, we have to add following piece of code in **WebApiConfig.cs** class of our MVC **Web API Project**:



```
1 //JsonFormatter
2
3 //MediaTypeHeaderValue
4
5 Config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
6   MediaTypeHeaderValue("application/json"));
```

[Back to top](#)

In this ASP.NET Web API Tutorial, we covered most important Interview Questions on ASP.NET Web API framework. Hopefully, it will be helpful for Web API developer Interview but along with these questions, do the practical implementation as much as you can. In [Practical guide to ASP.NET Web API](#), you can find a good step by step approach for understanding and implementing ASP.NET Web API services.

<http://www.webdevelopmenthelp.net/2014/05/asp-net-web-api-interview-questions.html>

What Are HTTP Methods?

Whenever a client submits a request to a server, part of that request is an HTTP method, which is what the client would like the server to do with the specified resource. HTTP methods [represent those requested actions](#). For example, some commonly-used HTTP methods will retrieve data from a server, submit data to a server for processing, delete an item from the server's data store, etc. For a more general overview of HTTP, see [Tutorials Point's article](#).

Selecting The Appropriate Method

A large portion of application functionality can be summed up in the [acronym CRUD](#), which stands for Create, Read, Update, Delete. There are four HTTP methods that correspond to these actions, one for each, like so:

C - Create - POST
R - Read - GET
U - Update - PUT
D - Delete - DELETE

So, in a given app, you might have the following action:

```
public IActionResult Add(string title)
{
    //Creates a Movie based on the Title
    return Ok();
}
```

We can tell from the name of the action (and, let's be real, the comment) that this action is supposed to create a movie. So we should use the POST verb on this action, like so:

```
[HttpPost]
public IActionResult Add(string title)
{
    //Creates a Movie based on the Title
    return Ok();
}
```

If you need a particular action to support more than one HTTP method, you can use the `[AcceptVerbs]` attribute:

```
[AcceptVerbs("POST", "PUT")]
public IActionResult Add(string title)
{
    //Creates a Movie based on the Title
}
```

```
return Ok();  
}
```

For the majority of applications, GET, POST, PUT, and DELETE should be all the HTTP methods you need to use. However, there are a few other methods we could utilize if the need arises.

- **HEAD:** This is identical to a GET request, but only returns the headers for the response, not the response body. Theoretically faster, commonly used for checking to see if a particular resource exists or can be accessed.
- **OPTIONS:** Returns the HTTP methods supported by the server for the specified URL.
- **PATCH:** Submits a partial modification to a resource. If you only need to update one field for the resource, you may want to use the PATCH method.

POST vs PUT

POST and PUT are very similar in that they both send data to the server that the server will need to store somewhere. Technically speaking, you could use either for the Create or Update scenarios, and in fact this is rather common. The difference [lies in the details](#).

PUT is [idempotent](#). What this means is that **if you make the same request twice using PUT, with the same parameters both times, the second request will have no effect**. This is why PUT is generally used for the Update scenario; calling Update more than once with the same parameters doesn't do anything more than the first call did.

By contrast, POST is not idempotent; making the same call using POST with same parameters each time will cause two different things to happen, hence why POST is commonly used for the Create scenario (submitting two identical items to a Create method should create two entries in the data store).

(It should be noted that, strictly speaking, HTTP does not force PUT to be idempotent, so you can implement your server to use PUT in a non-idempotent way. However, doing so is liable to cause a horde of angry server admins to show up at your desk and beat you with [ethernet cables](#). Don't say I didn't warn you.)

Default HTTP Methods

If we do not assign an explicit HTTP method to a controller action, what method(s) does that action accept? Let's imagine we have a Web API controller like so:

```
public class MovieController : ApiController  
{  
    /// <summary>  
    /// Returns all movies.
```



```

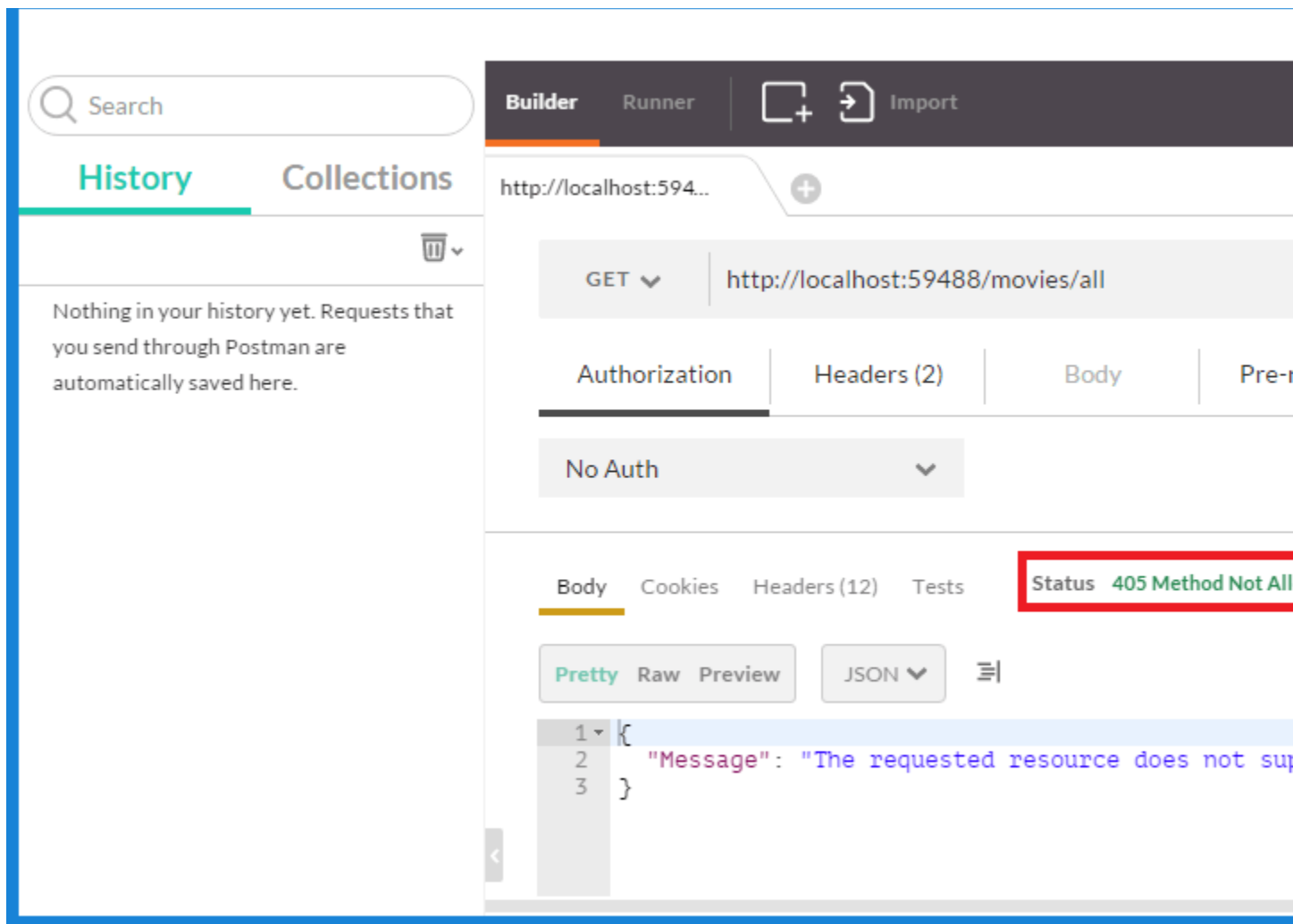
/// </summary>
/// <returns>A JSON list of all movies.</returns>
[Route("movies/all")]
public IHttpActionResult All()
{
    List<Movie> movies = new List<Movie>()
    {
        new Movie()
        {
            Id = 1,
            Title = "Up",
            ReleaseDate = new DateTime(2009, 5, 29),
            RunningTimeMinutes = 96
        },
        new Movie()
        {
            Id = 2,
            Title = "Toy Story",
            ReleaseDate = new DateTime(1995, 11, 19),
            RunningTimeMinutes = 81
        },
        new Movie()
        {
            Id = 3,
            Title = "Big Hero 6",
            ReleaseDate = new DateTime(2014, 11, 7),
            RunningTimeMinutes = 102
        }
    };

    return Ok(movies);
}
}

```

We can tell by looking at the code that this should be a GET action, since it is returning data. However, we're not explicitly saying that GET should be used (there's no `[HttpGet]` attribute). So, what method(s) will this action accept? Let's see what Postman can tell us.

It should be a GET action, so let's try to hit this action with a GET request.

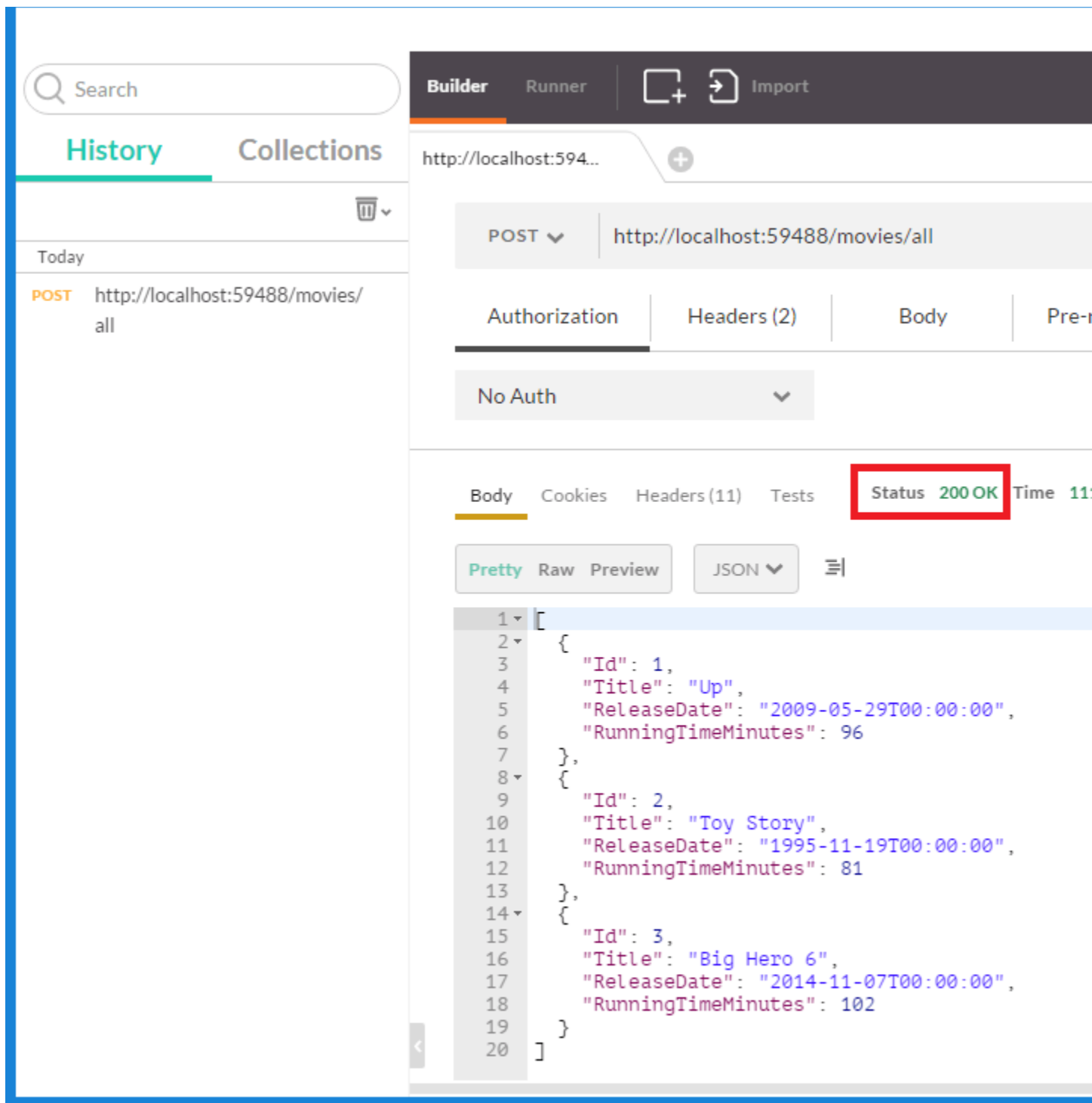


Well, that didn't work, we get back a 405 Method Not Allowed status. Why were we not able to use the GET method?

The [algorithm ASP.NET uses to calculate the "default" method](#) for a given action goes like this:

1. If there is an attribute applied (via `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[AcceptVerbs]`, etc), the action will accept the specified HTTP method(s).
2. If the name of the controller action starts the words "Get", "Post", "Put", "Delete", "Patch", "Options", or "Head", use the corresponding HTTP method.
3. Otherwise, the action supports the POST method.

We're falling in to the #3 condition here: the action name `All()` doesn't contain any of the key words and we didn't specify an action, so this action will only support POST. Sure enough, guess what Postman shows for a POST action?



Obviously, this is not what we want. We're getting data from the server using a POST method, and this (while not technologically prevented) is not what these HTTP methods were designed for.

We could solve this problem in two ways. The first would be to add the `[HttpGet]` attribute to the method. The second would be to rename the method to `GetAll()`; the existence of the word "Get" at the start of the method tells ASP.NET to accept a GET HTTP method on this action. My

personal preference is to **always explicitly state which HTTP method is accepted by any action**, like so:

```
public class MovieController : ApiController
{
    /// <summary>
    /// Returns all movies.
    /// </summary>
    /// <returns>A JSON list of all movies.</returns>
    [Route("movies/all")]
    [HttpGet] //Always explicitly state the accepted HTTP method
    public IHttpActionResult All()
    {
        //Get movies
        return Ok(movies);
    }
}
```

Summary

Always use the appropriate HTTP action in your Web API actions, as it establishes the kinds of communication your consumers can conduct with your app. Further, always explicitly state what HTTP method(s) each action can accept, and keep to the common definitions for each action (e.g. GET for data retrieval, POST for creating data, PUT for updating data, etc.).

For more information, check out [ASP.NET Web API 2: Building a Restful Service from Start to Finish](#) by [Jamie Kurtz](#), specifically Chapter 2, "What is RESTful?" and Chapter 5, "Up and Down the Stack with a POST".

Happy Coding!

<https://www.exceptionnotfound.net/using-http-methods-correctly-in-asp-net-web-api/>

How Routing Works for Web API

Unlike ASP.NET MVC, the Web API routing convention routes incoming requests to a specific controller, but by default simply matches the HTTP verb of the request to an action method whose name begins with that verb. Recall the default MVC route configuration:

The Default MVC Route Template:

Hide Copy Code

```
{controller}/{action}/{id}
```

In contrast, the default Web API route template looks like this:

The Default Web API Route Template:

Hide Copy Code

```
api/{controller}/{id}
```

The literal **api** at the beginning of the Web API route template above makes it distinct from the standard MVC route. Also, it is a good convention to include as part of an API route a segment that lets the consumer know they are accessing an API instead of a standard site.

Also notice that unlike the familiar MVC route template, the Web API template does not specify an **{action}** route parameter. This is because, as we mentioned earlier, the Web API framework will by default map incoming requests to the appropriate action based upon the HTTP verb of the request.

Consider the following typical Web API Controller:

WebApi Controller from Default Visual Studio WebApi Template

Hide Shrink ▲ Copy Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace StandardWebApiTemplateProject.Controllers
{
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

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

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

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

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

First, take note that Web API controllers do not inherit from `System.Web.Mvc.Controller`, but instead from `System.Web.Http.Controller`. Again, we are working with a familiar, but distinct library. Of particular note, however, are the default method names (the example above is what is created as part of the default VS 2012 project template). Again, without specifying a specific method using an `{action}` route parameter, the Web API framework determines the appropriate route based on the HTTP verb of the incoming request, and calls the proper method accordingly. The following incoming URL, included as part of an HTTP GET message, would map to the first `Get()` method as defined in the controller above:

Example URL With No ID Parameter

[Hide](#) [Copy Code](#)

```
http://mydomain/values/
```

Similarly, this URL, also as part of an HTTP GET request, would map to the second `Get(id)` method:

Example URL Including ID Parameter

[Hide](#) [Copy Code](#)

```
http://mydomain/values/5
```

We could, however, modify our controller thusly, and the routing would still work without modification:

Modifications to the ValuesController Class:

[Hide](#) [Copy Code](#)

```
// GET api/values
public IEnumerable<string> GetValues()
{
    return new string[] { "value1", "value2" };
}

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

// POST api/values
public void PostValue([FromBody]Book value)
{
}

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

// DELETE api/values/5
public void DeleteValue(int id)
{
}
```

Note that we retained the HTTP action verb as a prefix to all of our method names. Under these circumstances, the unmodified route parameter will still work just fine.

When per convention, no `{action}` route parameter is specified, the Web API framework again appends the "Controller" suffix to the value provided for the `{controller}` parameter, and then scans the project for a suitably named class (in this case, one which derives from `ApiController`).

Action Method Selection

Once the proper controller class is selected, the framework examines the HTTP action verb of the request, and searches the class for method names with a matching prefix.

In order to determine the proper method, the framework then examines the additional URL parameters and attempts to match them with method arguments by name (case insensitive). The method with the most matching arguments will be selected.

An item to note here. Unlike in MVC, in Web API complex types are not allowed as part of the URL. Complex types must be placed in the body of the HTTP message. Also, there can be one, and only one such complex type in the message body.

In evaluating parameter matches against method arguments, any complex types and URL query strings are disregarded when searching for a match.

Web API and Action Naming

We can modify our default Web API route to include an `{action}` route parameter, in which case selection will occur similar to that in MVC. However, action methods defined on our controller still need to include the proper HTTP action verb prefix, and incoming URLs must use the full method name.

If we modify our default route thusly, adding an `{action}` parameter:

The Modified `WebApiConfig` Class and `MapHttpRequest` Method:

[Hide](#) [Copy Code](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

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

To see how this works, consider the incoming URL:

```
http://mydomain/values/5
```

This will no longer work. If we wish to make a request of our modified `ValuesController` now, we would need to submit:

```
http://mydomain/values/GetValue/5
```

Review of the Basics for Web API Routing

- The routing convention for Web API is to route URLs to a controller, and then to the action which matches the HTTP verb of the request message. Action methods on the controller must either match the HTTP action verb, or at least include the action verb as a prefix for the method name.
- The default route template for a Web API Project is `{controller}/{id}` where the `{id}` parameter is optional.
- Web API route templates may optionally include an `{action}` parameter. However, the action methods defined on the controller must be named with the proper HTTP action verb as a prefix in order for the routing to work.
- In matching incoming HTTP messages to controllers, the Web API framework identifies the proper controller by appending the literal "Controller" to the value of the `{controller}` route parameter, then scans the project for a class matching that name.
- Actions are selected from the controller by considering the non-complex route parameters, and matching them by name to the arguments of each method which matches the HTTP verb of the request. The method which matches the most parameters is selected.
- Unlike MVC, URLs in Web API cannot contain complex types. Complex types must be placed in the HTTP message body. There may be one, and only one complex type in the body of an HTTP message.

<https://www.codeproject.com/Articles/624180/Routing-Basics-in-ASP-NET-Web-API>

FROMUri binding

ASP.NET Web API parameter binding part 1 – Understanding binding from URI

Today, let's kick off a series intended to look at different aspects of HTTP parameter binding in ASP.NET Web API. Why? Aside from the awesome series by [Mike Stall](#), there

isn't really that much material on the web on this particular subject. And developers coming from MVC background, often get surprised by differences in the model binding mechanism between MVC and Web API.

In this first post, let's have a brief overview of parameter binding in Web API and then specifically look at binding model from URI.

More after the jump.

Model binding with MVC

In MVC, the model binding process is done against both the body and the URI. In other words, the framework will look for model properties in both places, and try to stitch everything together. This is possible because MVC would buffer the body of the request and that would allow it to pick pieces of key value pairs (effectively, each value in the request is part of the name value collection) and try to compose models out of them.

Consider the simple example.

C#



```
1 public class Person
2 {
3     public string FirstName {get; set;}
4     public string LastName {get; set;}
5 }
```

In MVC you can send the following request:

C#



```
1 POST /person/index?FirstName=Filip
```

```
2
Content-Type: application/json
3
{"LastName":"W"}
```

This will correctly bind itself to an action with this signature:

C#



```
1 [HttpPost]
2 public JsonResult Index(Person p) {}
```

You can also easily bind the *whole* model just from the URI, which is particularly useful on GET requests where you might be passing in a complex sets of conditions. Conversely, you can bind the entire model just from the body too.

HttpParameterBinding

In Web API, the core concept behind binding parameters is *HttpParameterBinding*, which, out of the box, can execute either as a model binder, or through the use of a formatter.

In practice, this means that when Web API pipeline runs, a default implementation of *IActionValueBinder* will determine whether each specific parameter will be bound using the URI (model binding) or request's body (formatter). To be even more precise, this process will not happen on *every request*, because in Web API binding can be statically determined for each parameter. Therefore, it will really happen just once – through a call to *ApiControllerActionSelector*, which will cache the resulting *ActionDescriptor* inside a private *ActionSelectorCacheItem* class. The subsequent requests will then be fed from that cache.

We'll look at what happens when the formatter type of binding is selected in the next post. For now let's focus on what happens when Web API decides to use *ModelBinder* and bind from URI.

Web API will look for any *ValueProviderFactories* registered against the *Services* property of *HttpConfiguration*.

C#



```
1 public abstract class ValueProviderFactory
2 {
3     public abstract IValueProvider GetValueProvider(HttpContext actionContext);
4 }
```

A factory returns *IValueProvider*, which can be used to compose an object from the HTTP request using the most basic building block – a string key/value present in the HTTP request. The main role of *ValueProviders* is to abstract away the logic of retrieving the information from the HTTP request from *ModelBinders* and just feed them with individual bits used to compose more complex objects.

C#



```
1 public interface IValueProvider
2 {
3     bool ContainsPrefix(string prefix);
4     ValueProviderResult GetValue(string key);
5 }
```

By default, Web API ships with two registered factories that take part in the URI parameter bindings:

- – *QueryStringValueProviderFactory*
- – *RouteDataValueProviderFactory*

And their names & roles are rather self explanatory.

On a side note, it's worth mentioning, that if the split into model binders / formatters is not enough for you, you can also introduce your own versions of *HttpParameterBinding* – and we'll do that in the later posts in this series.

Understanding Web API parameter binding

So what happens if you just stick some parameters into your action? How does *IActionValueBinder* determine whether to use a formatter or a model binder?

The generic rules are:

- – simple, string-convertible parameters (value types, strings, Guids, DateTimes and so on) are by default read from URI
- – complex types are by default read from the body
- – collections of simple parameters are by default read from the body too
- – you cannot compose a single model based on input from both URI and request body, it has to be one or the other

While some of these default conventions may seem limiting and unintuitive, there are just enough customization hooks to allow you as a developer to override them.

Binding from URI

In order to bind a model (an action parameter), that would normally default to a formatter, from the URI you need to decorate it with either *[ModelBinder]* or *[FromUri]* attribute.

FromUriAttribute simply inherits from *ModelBinderAttribute*, providing you a shortcut directive to instruct Web API to grab specific parameters from the URI using the *ValueProviders* defined in the *IUriValueProviderFactory*. The attribute itself is sealed and cannot be extended any further, but you add as many custom *IUriValueProviderFactories* as you wish.

[Mike Stall](#) has a great post on creating a custom value provider for Web API.

Let's use a *ProductFilter* as an example:

C#



```
1 public class ProductFilter
```

```

2 {
3     public int? PageIndex {get; set;}
4     public int? PageSize {get; set;}
5     public string[] Sizes {get; set;}
6     public decimal? MinPrice {get; set;}
7     public decimal? MaxPrice {get; set;}
8 }

```

Just through the use of *[FromUri]* you are able to bind the following types entirely from the URI:

1) complex types, such as our *ProductFilter*

C#



```

1 public HttpResponseMessage Get([FromUri]Productfilter filter) {}
2
3 GET /products?pageindex=2&minprice=10&sizes=xl&sizes=xxl

```

This will bind correctly to the type we declared, leaving other properties as null (since we explicitly declared them as nullable).

2) collections, such as **List**

List

C#



```

1 GET /products?items=a&items=b&items=c

```

2

```
3 public HttpResponseMessage Get([FromUri]List<string> items) {}
```

3) key value pairs

C#



```
1 GET /product?key=mykey&value=myvalue
```

2

```
3 public HttpResponseMessage Get([FromUri]KeyValuePair<string, string> id) {}
```



4) apparently it also works with dictionaries, but I honestly don't know how

5) anything else, as long as you create a custom *ModelBinder* or a custom *ValueProvider* that would do the appropriate type coercion from the string

[ModelBinder], allows you to plug in your own logic of determining how a set of data fed from *ValueProviders* should be converted into a CLR type of your choice.

A good example might be:

C#



```
1 GET /products?sizes=L,XL,XXL
```

In this example you may want to bind *sizes* to an *IEnumerable* or *Array* of strings. There is nothing in the box in Web API for this particular scenario, so you'd need a custom *[ModelBinder]*.

In the above example it would be:

C#



```
1 public class CommaDelimitedCollectionModelBinder : IModelBinder
2 {
3     public bool BindModel(HttpContext actionContext, ModelBindingContext bindingContext)
4     {
5         var key = bindingContext.ModelName;
6         var val = bindingContext.ValueProvider.GetValue(key);
7         if (val != null)
8         {
9             var s = val.AttemptedValue;
10             if (s != null && s.IndexOf(",", System.StringComparison.Ordinal) > 0)
11             {
12                 var stringArray = s.Split(new[] { "," }, StringSplitOptions.None);
13                 bindingContext.Model = stringArray;
14             }
15             else
16             {
17                 bindingContext.Model = new[] { s };
18             }
19             return true;
20         }
21         return false;
22     }
23 }
```

0
2
1
2
2
2
3

And can now be used:

C#



```
1 public HttpResponseMessage  
1 Get([ModelBinder(typeof(CommaDelimitedCollectionModelBinder))]IEnumerable<string> sizes) { };
```

Notice we used the default *ValueProviders* and just split the string extracted by the provider in the *ModelBinder* itself. You could potentially split the string inside a custom value provider too.

<http://www.strathweb.com/2013/04/asp-net-web-api-parameter-binding-part-1-understanding-binding-from-uri/>

Parameter Binding in ASP.Net Web API



- [Mudita Rathore](#)

- Jul 17 2013

- [Article](#)

- 0
- 0
- 28.4k
-

- [-](#)
- [-](#)
- [-](#)
- [-](#)
- [-](#)
-

-
-
-
-

- [-](#)

Introduction

This article explains parameter binding in the ASP.NET Web API. Parameter binding is a type for catching values from the URI and from the message body by the Web API. These depend on the type of the parameter.

There are various rules for binding the parameters:

- **"Simple" type:** If the parameter is the simple type then it is a string convertible parameter that includes the preemptive data types such as Int, Double, Bool and so on with the Date Time, Decimal, string and so on. By default these are read from the URI.
- **"Complex" type:** If the parameter is a complex type then the Web API catches the value from the message body. It uses the media type formatters for catching the value from the body.

`HttpStatusCode` Get(int No, Item value)

{

....

}

The preceding is an example in which the parameter "No" is a simple type so the value of this parameter is caught from the URI. and the second parameter "value" that is a complex type so the value of this parameter is caught from the body.

From URI binding

In the From URI binding we use the [FromUri] attribute. The [FromUri] attribute is inherited from the [ModelBinder] attribute. Let's see an example:

```
public class Product
```

```
{  
    public int pageNo { get; set; }  
    public decimal Price { get; set; }  
    public int size { get; set; }  
}
```

We use the [FromUri] attribute that is bound in this example from the URI.

```
public class ValuesController : ApiController
```

```
{  
    HttpResponseMessage Get([FromUri] Product product)  
    {  
        .....  
    }  
}
```

The user puts the value of the pageNo, Price and size from the query string and the Web API reads these values from the URI and uses it for the "Product".

GET/products?pageNo=1&price=20&size=xxl

From Body binding

In the Body binding we use the [FromBody] attribute. That allows the Web API to catch the value of the parameters from the body.

```
public HttpResponseMessage Post([FromBody] string Address)
```

Model Binding

The model binding is used for both the URI and the message body. We can say that in the model binding the framework determines the properties of the model in both the URI and body and then it attempts to determine the result together after running it. It can be done because the body of the request is being loaded by the MVC that can determine the key value pairs and then produce the model.

When we create the model binder then we implement the class with the "IMoDelBinder" interface. This interface includes a method, "Bind Model".

```
bool BindModel(HttpContext context, ModelBindingContext contextb);
```

Lets see a sample of code using ModelBinder:

```
public class ProductBinder : IMoDelBinder
{
    private static ConcurrentDictionary<string, Product> products
    = new ConcurrentDictionary<string, Product>(StringComparer.OrdinalIgnoreCase);

    static ProductBinder()
    {
        products["redmond"] = new Product() { pageNo = 2, Price = 10, size = 20 };
        products["paris"] = new Product() { pageNo = 3, Price = 30, size = 40 };
    }

    public bool BindModel(HttpContext context, ModelBindingContext contextb)
    {
        ....
        ....
    }
}
```

We use the namespace "using System.Web.Http.ModelBinding" that has the reference of the "BindModel" method and "IMoDelBinder" interface. The "BindModel" method binds the model to a value using the specified controller context and binding context.

"ModelBindingContext" initializes the new instance of the ModelBindingContext class.

"HttpContext" initializes the instance of the HttpContext class.

It is possible to bind the entire model with the URI used in the GET request. In it you can provide the complex conditions. And you can also bind the Model with the message body.

<http://www.c-sharpcorner.com/UploadFile/2b481f/parameter-binding-in-Asp-Net-web-api/>

This article explains the "FromBody" and "FromUri" attributes and how to use them with an action parameter to consume data at the client end. So, let's learn by example.

Use of FromUri attribute to send data

Using the FormUri attribute, we can pass data through a URI/URL. The value should in the form of a key value pair. Here is one example to send data through a URI. In this example we are sending one string through the URL. The name of the parameter is "Name".

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="APICall.aspx.cs" Inherits="WebApplication1.APICall" %>
<head runat="server">
    <script src="jquery-1.7.1.js" type="text/javascript"></script>
    <script>
        $(document).ready(function () {
            $("#Save").click(function () {

                $.ajax({
                    url: 'http://localhost:3413/api/person?Name=Sourav',
                    type: 'POST',
                    dataType: 'json',
                    success: function (data, textStatus, xhr) {
                        console.log(data);
                    },
                    error: function (xhr, textStatus, errorThrown) {
                        console.log('Error in Operation');
                    }
                });
            });
        });
    </script>
</head>
<body>
    <form name="form1" id="form1">
        <input type="button" id="Save" value="Save Data" />
    </form>
</body>
</html>
```

It's obvious that there are certain limitations in this method. The length of the URL in some browsers is limited to 256 characters and there might be a security loophole.

Configure Web API to get data from URI

Now we need to configure the Web API to extract data from a URI. We have implemented one action named "PostAction" that will take one parameter and the parameter is specified with the FromUri attribute. This implies that the value of the Name parameter will pass through the URI. Here is the implementation of the working code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebApplication1.WebAPI
{
    public class personController : ApiController
    {
        [HttpPost]
        public String PostAction([FromUri] string Name)
        {
```

```

        return "Post Action";
    }
}
}

```


We have a halting application to check whether the data is coming and it's coming.

Note: We can pass multiple parameters in a single URI.

```

[HttpPost]
public String PostAction([FromBody] String Name)
{
    return "Post Action";
}

```



Get data using FromBody attribute

This is another way to get data in an ajax() call. In this example we will see how to get data using the FromBody attribute. Have a look at the following example.

Here is an Implementation of the ajax() function to send data. Have a look at the data parameter of the ajax() function. We are seeing that the data is being passed using a key value pair but the key portion is empty. When we are interested in receiving data using a FromBody attribute then we should keep the key section as empty.

```

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="APICall.aspx.cs" Inherits="WebApplication1.APICall" %>
<head runat="server">
    <script src="jquery-1.7.1.js" type="text/javascript"></script>
    <script>
        $(document).ready(function () {
            $("#Save").click(function () {

                $.ajax({
                    url: 'http://localhost:3413/api/person',
                    type: 'POST',
                    dataType: 'json',
                    data: {"":"Sourav Kayal"},
                    success: function (data, textStatus, xhr) {
                        console.log(data);
                    },
                    error: function (xhr, textStatus, errorThrown) {
                        console.log('Error in Operation');
                    }
                });

            });
        });
    </script>
</head>
<body>
    <input type="button" id="Save" value="Save Data" />
</body>

```

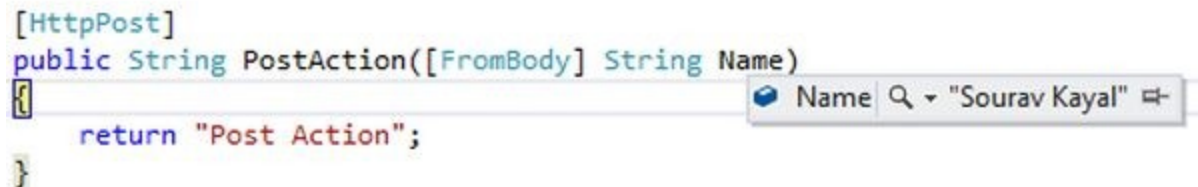
</html>

Now we need to configure the Web API action to receive the value using theFromBody attribute. See the "PostAction" action in the person controller. We will see that the Name parameter is specified with the FromBody attribute.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebApplication1.WebAPI
{
    public class person
    {
        public string name { get; set; }
        public string surname { get; set; }
    }

    public class personController : ApiController
    {
        [HttpPost]
        public String PostAction([FromBody] String Name)
        {
            return "Post Action";
        }
    }
}
```



The screenshot shows a code editor with the following code snippet:

```
[HttpPost]
public String PostAction([FromBody] String Name)
{
    return "Post Action";
}
```

A search bar is visible on the right side of the code editor, containing the text "Name" and a search icon. The search results show "Sourav Kayal" with a dropdown arrow.

We are seeing that the value is coming from the ajax() function at the time of the POST operation.

Is multiple FromBody attribute is allowed?

No, multiple formBodys is not allowed in a single action. In other words, we cannot specify multiple FromBody parameters in a single action. Here is an example of an incorrect implementation of an action:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
```

```

namespace WebApplication1.WebAPI
{
    public class personController : ApiController
    {
        [HttpPost]
        public String PostAction([FromBody]string name, [FromBody] string surname)
        {
            return "";
        }
    }
}

```

Conclusion

This article has explained the concept of a FromUri and FromBody to receive a value from the jQuery ajax() function. I hope you have understood it and you will implement it in your next AJAX call. In a future article we will explore a few more interesting facts.

Passing multiple complex type parameters to ASP.NET Web API

Asp.Net Web API introduces a new powerful REST API which can be consumed by a broad range of clients including browsers, mobiles, iPhone and tablets. It is focused on resource based solutions and HTTP verbs.

Asp.Net Web API has a limitation while sending data to a Web API controller. In Asp.Net Web API you can pass only single complex type as a parameter. But sometimes you may need to pass multiple complex types as parameters, how to achieve this?

You can also achieve this task by wrapping your Supplier and Product classes into a wrapper class and passing this wrapper class as a parameter, but using this approach you need to make a new wrapper class for each action which required complex types parameters. In this article, I am going to explain another simple approach using ArrayList.

Let's see how to achieve this task. Suppose you have two classes - Supplier and Product as shown below -

```

1.      public class Product
2.      {
3.          public int ProductId { get; set; }
4.          public string Name { get; set; }
5.          public string Category { get; set; }
6.          public decimal Price { get; set; }
7.      }
8.
9.      public class Supplier

```

```

10.     {
11.         public int SupplierId { get; set; }
12.         public string Name { get; set; }
13.         public string Address { get; set; }
14.     }

```

In your Asp.Net MVC controller you are calling your Web API and you need to pass both the classes objects to your Web API controller.

Method 1 : Using ArrayList

For passing multiple complex types to your Web API controller, add your complex types to ArrayList and pass it to your Web API actions as given below-

```

1.     public class HomeController : Controller
2.     {
3.         public ActionResult Index()
4.         {
5.             HttpClient client = new HttpClient();
6.             Uri baseAddress = new Uri("http://localhost:2939/");
7.             client.BaseAddress = baseAddress;
8.
9.             ArrayList paramList = new ArrayList();
10.            Product product = new Product { ProductId = 1, Name = "Book", Price =
11.            500, Category = "Soap" };
12.            Supplier supplier = new Supplier { SupplierId = 1, Name = "AK Singh",
13.            Address = "Delhi" };
14.            paramList.Add(product);
15.            paramList.Add(supplier);
16.
17.            HttpResponseMessage response =
18.            client.PostAsJsonAsync("api/product/SupplierAndProduct", paramList).Result;
19.            if (response.IsSuccessStatusCode)
20.            {
21.                return View();
22.            }
23.            else
24.            {
25.                return RedirectToAction("About");
26.            }
27.        }
28.        public ActionResult About()
29.        {
30.            return View();
31.        }
32.    }

```

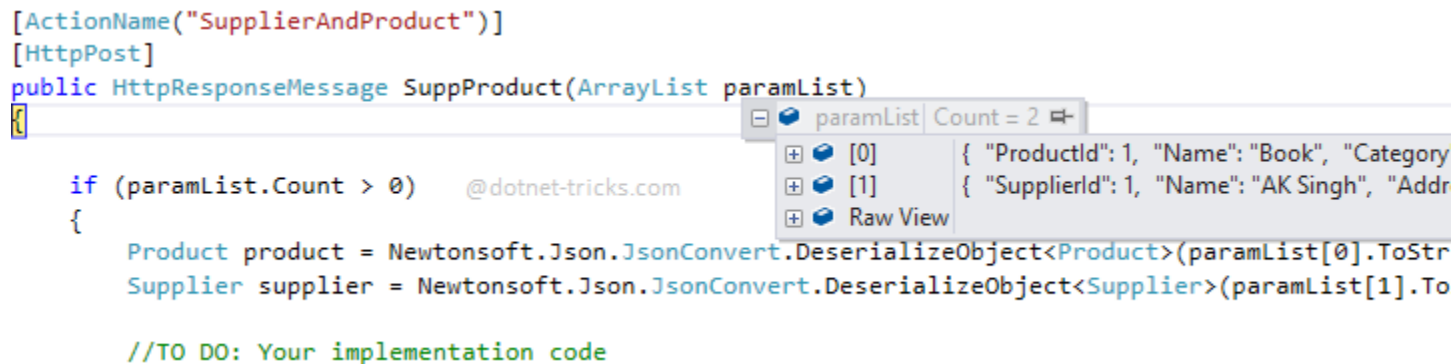
Now, on Web API controller side, you will get your complex types as shown below.


```

[ActionName("SupplierAndProduct")]
[HttpPost]
public HttpResponseMessage SuppProduct(ArrayList paramList)
{
    if (paramList.Count > 0)    @dotnet-tricks.com
    {
        Product product = Newtonsoft.Json.JsonConvert.DeserializeObject<Product>(paramList[0].ToString());
        Supplier supplier = Newtonsoft.Json.JsonConvert.DeserializeObject<Supplier>(paramList[1].ToString());

        //TO DO: Your implementation code
    }
}

```



Now deserialize your complex types one by one from ArrayList as given below-

```

1. public class ProductController : ApiController
2. {
3.     [ActionName("SupplierAndProduct")]
4.     [HttpPost]
5.     public HttpResponseMessage SuppProduct(ArrayList paramList)
6.     {
7.         if (paramList.Count > 0)
8.         {
9.             Product product =
Newtonsoft.Json.JsonConvert.DeserializeObject<Product>(paramList[0].ToString());
10.            Supplier supplier =
Newtonsoft.Json.JsonConvert.DeserializeObject<Supplier>(paramList[1].ToString());
11.
12.            //TO DO: Your implementation code
13.
14.            HttpResponseMessage response = new HttpResponseMessage {
15.                StatusCode = HttpStatusCode.Created };
16.            return response;
17.        }
18.        else
19.        {
20.            HttpResponseMessage response = new HttpResponseMessage {
21.                StatusCode = HttpStatusCode.InternalServerError };
22.            return response;
23.        }
24.    }
25. }

```

Method 2 : Using Newtonsoft JArray

For passing multiple complex types to your Web API controller, you can also add your complex types to JArray and pass it to your Web API actions as given below-

```

1. public class HomeController : Controller
2. {
3.     public ActionResult Index()
4.     {

```

```

5.     HttpClient client = new HttpClient();
6.     Uri baseAddress = new Uri("http://localhost:2939/");
7.     client.BaseAddress = baseAddress;
8.
9.     JArray paramList = new JArray();
10.    Product product = new Product { ProductId = 1, Name = "Book", Price =
11.    500, Category = "Soap" };
12.    Supplier supplier = new Supplier { SupplierId = 1, Name = "AK Singh",
13.    Address = "Delhi" };
14.
15.    paramList.Add(JsonConvert.SerializeObject(product));
16.    paramList.Add(JsonConvert.SerializeObject(supplier));
17.
18.    HttpResponseMessage response =
19.    client.PostAsJsonAsync("api/product/SupplierAndProduct", paramList).Result;
20.    if (response.IsSuccessStatusCode)
21.    {
22.        return View();
23.    }
24.    else
25.    {
26.        return RedirectToAction("About");
27.    }
28.    public ActionResult About()
29.    {
30.        return View();
31.    }

```

Note

Don't forget to add reference of Newtonsoft.Json.dll to your ASP.NET MVC project and WebAPI as well.

Now, on Web API controller side, you will get your complex types within JArray as shown below.

```

public HttpResponseMessage SuppProduct(JArray paramList)
{
    if (paramList.Count > 0)
    {
        Product product = JsonConvert.DeserializeObject<Product>(paramList[0].ToString());
        Supplier supplier = JsonConvert.DeserializeObject<Supplier>(paramList[1].ToString());
        //TO DO: Your implementation code
    }
}

```

Now deserialize your complex types one by one from JArray as given below-

```

1.    public class ProductController : ApiController
2.    {
3.        [ActionName("SupplierAndProduct")]

```

```

4.      [HttpPost]
5.      public HttpResponseMessage SuppProduct(JArray paramList)
6.      {
7.          if (paramList.Count > 0)
8.          {
9.              Product product = JsonConvert.DeserializeObject(paramList[0].ToString());
10.             Supplier supplier = JsonConvert.DeserializeObject(paramList[1].ToString());
11.
12.             //TO DO: Your implementation code
13.
14.             HttpResponseMessage response = new HttpResponseMessage {
15.                 StatusCode = HttpStatusCode.Created };
16.             return response;
17.         }
18.         else
19.         {
20.             HttpResponseMessage response = new HttpResponseMessage {
21.                 StatusCode = HttpStatusCode.InternalServerError };
22.             return response;
23.         }
24.     }

```

In this way, you can easily pass your complex types to your Web API. There are two solution, there may be another one as well.

What do you think?

I hope you will enjoy the tips while programming with Asp.Net Web API. I would like to have feedback from my blog readers. Your valuable feedback, question, or comments about this article are always welcome.

<http://www.dotnettricks.com/learn/webapi/passing-multiple-complex-type-parameters-to-aspnet-web-api>

How to pass multiple parameters to Web API controller methods

Learn how to pass multiple complex objects as parameters to Web API controller methods

InfoWorld | OCT 31, 2016

•

•

•

•

•

•

•

•

```
[HttpPost]
0 references
public HttpResponseMessage PostAuthor1(AuthorRequest request)
{
    var author = request.Author;
    var token = request.Token;
    //Usual code to store data in the database
    return Request.CreateResponse(HttpStatusCode.OK, "Success...");
}
```

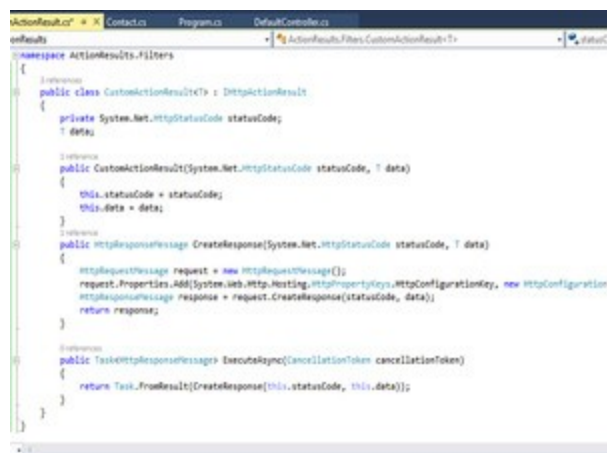
```
[HttpPost]
0 references
public HttpResponseMessage PostAuthor2(FormDataCollection form)
{
    var author = form.Get("Author");
    var token = form.Get("Token");

    //Usual code to store data in the database
    return Request.CreateResponse(HttpStatusCode.OK, "Success...");
}
```

Ads by [Kiosked](#)

MORE LIKE THIS

- [Parameter binding in Web API](#)



```
namespace ActionResults.Filters
{
    [reference]
    public class CustomActionResult<T> : IHttpActionResult
    {
        private System.Net.HttpStatusCode statusCode;
        T data;

        [reference]
        public CustomActionResult(System.Net.HttpStatusCode statusCode, T data)
        {
            this.statusCode = statusCode;
            this.data = data;
        }

        [reference]
        public HttpResponseMessage CreateResponse(System.Net.HttpStatusCode statusCode, T data)
        {
            HttpRequestMessage request = new HttpRequestMessage();
            request.Properties.Add(System.Web.Http.Hosting.HttpPropertyKeys.HttpConfigurationKey, new HttpConfiguration());
            HttpResponseMessage response = request.CreateResponse(statusCode, data);
            return response;
        }

        [reference]
        public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
        {
            return Task.FromResult(CreateResponse(this.statusCode, this.data));
        }
    }
}
```

[How to work with ActionResult in Web API](#)

Routing in ASP.NET Web API

```

Configure * X
-----
services.WebApiConfig
-----
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

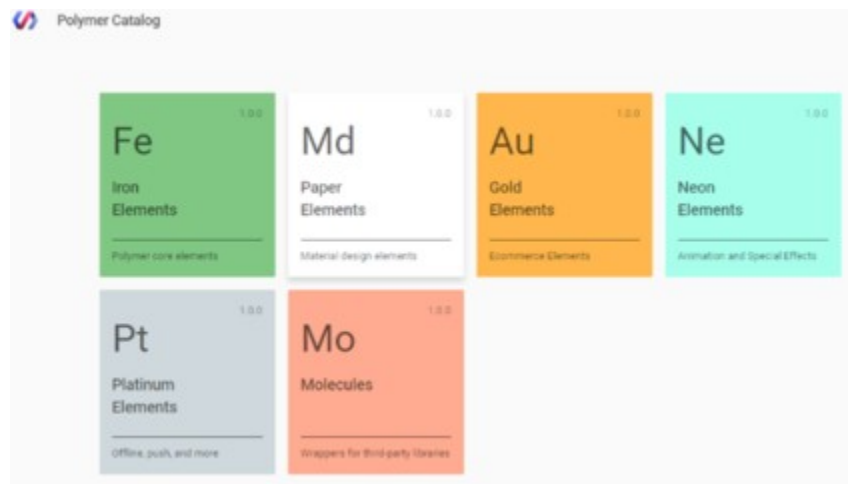
namespace D00Services
{
    [reference]
    public static class WebApiConfig
    {
        [reference]
        public static void Register(HttpConfiguration config)
        {
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            // Uncomment the following line of code to enable query support for actions with an IHttpQueryable or IQueryable<T> return
            // To avoid processing unexpected or malicious queries, use the validation settings on IQueryableAttribute to validate
            // For more information, visit http://msdn.microsoft.com/feedback/76062479732
            //config.EnableQuerySupport();

            // To disable tracing in your application, please comment out or remove the following line of code
            // For more information, refer to http://www.asp.net/web-api
            config.EnableSystemDiagnosticsTracing();
        }
    }
}

```

Exploring routing in Web API



VIDEO

Why you need to start using Polymer for your Web development



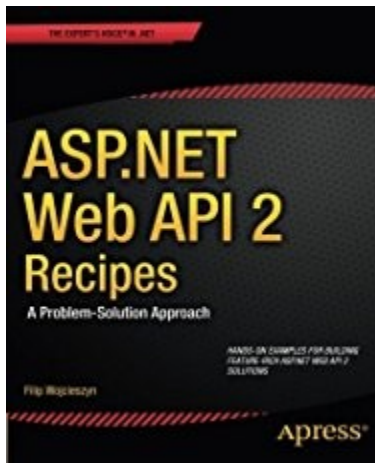
71% off Cambridge SoundWorks OontZ Angle 3 PLUS Wireless Bluetooth Speaker - ...



35% off Razor Hovertrax 2.0 Hoverboard Self-Balancing Smart Scooter - Deal...



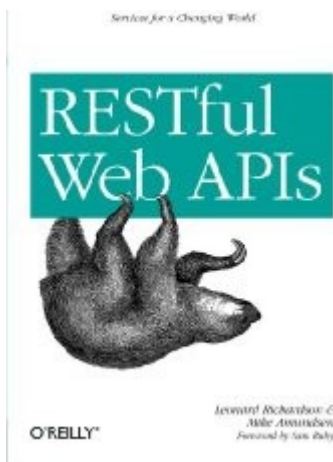
34% off TurboTax Deluxe 2016 Tax Software Federal & State - Deal Alert



[ASP.NET Web API 2 Recipes: A Probl...](#)

\$47.49

(10)



RELATED ARTICLES



[Create your own Slack bots -- and web APIs -- in R](#)



[Why you need a bug bounty program](#)



[The best Go language IDEs and editors](#)

[See all Insider](#)

In [an earlier post here](#) we explored Parameter Binding in Web API. In this post, we will learn how to pass multiple parameters to Web API controller methods.

Web API provides the necessary action methods for HTTP GET, POST, PUT, DELETE operations. You would typically pass a single object as a parameter to the PUT and POST action methods. Note that Web API doesn't support for passing multiple POST parameters to Web API controller methods by default. But what if you were to make a POST request with multiple objects passed as parameter to a Web API controller method?

Understanding the problem

Web API doesn't allow you to pass multiple complex objects in the method signature of a Web API controller method -- you can post only a single value to a Web API action method. This value in turn can even be a complex object. It is possible to pass multiple values though on a POST or a PUT operation by mapping one parameter to the actual content and the remaining ones via query strings.

[Download the InfoWorld megaguide: [The best Python frameworks and IDEs](#). | Keep up with hot topics in programming with InfoWorld's [App Dev Report newsletter](#).]

The following controller class contains a POST method named Save that accepts multiple parameters.

```
public class AuthorsController : ApiController
{
    [HttpPost]

    public HttpResponseMessage Save(int Id, string FirstName, string
LastName, string Address)
    {
        //Usual code

        return Request.CreateResponse(HttpStatusCode.OK, "Success...");
    }
}
```

Now suppose you attempt to call the Web API controller method from JQuery as shown below.

```
$.ajax({

    url: 'api/authors',

    type: 'POST',

    data: { Id: 1, FirstName: 'Joydip', LastName: 'Kanjilal', Address:
'Hyderabad' },

    dataType: 'json',
```

```
    success: function (data) {  
  
    alert(data);  
  
    }));
```

Unfortunately, this call will fail as this request cannot be processed by Web API. Similarly, if you have a Web API controller method that accepts multiple complex objects, you would not be able to invoke this method directly from a client in a straight forward manner.

```
[HttpPost]  
  
public HttpResponseMessage PostAuthor(Author author, string  
authenticationToken)  
  
{  
  
    //Usual code  
  
    return Request.CreateResponse(HttpStatusCode.OK, "Success...");  
  
}
```

You can pass parameters to Web API controller methods using either the [FromBody] or the [FromUri] attributes. Note that the [FromBody] attribute can be used only once in the parameter list of a method. To reiterate, you are allowed to pass only one value (simple or complex type) as parameter to a Web API controller method when using the [FromBody] attribute. You can pass any number of parameters using the [FromUri] attribute but that is not the ideal solution in our case.

And now, the solution

Now that we have understood what the problem is when passing parameters to a Web API controller method, let's explore the possible solutions. One way to achieve this is by passing the complex object as a [FromBody] attribute and the string parameter via the Uri as shown in the code snippet below.

```
$.ajax({  
  
    url: 'api/authors?authenticationToken=abcxyz',  
  
    type: 'POST',  
  
    data: JSON.stringify(author),  
  
    dataType: 'json',
```

```
success: function (data) {  
  
    alert(data);  
  
}});
```

You would need to modify your Web API controller method accordingly to parse the query string as shown below.

```
[HttpPost]  
  
public HttpResponseMessage PostAuthor(Author author)  
{  
  
    var data = Request.RequestUri.ParseQueryString();  
  
    string criteria = queryItems["authenticationToken"];  
  
    //Usual code to store data in the database  
  
    return Request.CreateResponse(HttpStatusCode.OK, "Success...");  
  
}
```

Well, but what if you have multiple complex objects to be passed as parameters to the Web API controller method? You can create a single object that wraps the multiple parameters. Refer to the AuthorRequest class given below.

```
public class AuthorRequest  
{  
  
    public Author Author { get; set; }  
  
    public string Token { get; set; }  
  
}
```

Basically, you can wrap multiple parameters in a single class and use this class as a parameter to your Web API controller method.

Here's how the updated Web API controller method would now look like.

```
[HttpPost]  
  
public HttpResponseMessage PostAuthor(AuthorRequest request)  
{
```

```

        var author = request.Author;

        var token = request.Token;

        //Usual code to store data in the database

        return Request.CreateResponse(HttpStatusCode.OK, "Success...");
    }

```

You can also use JObject to parse multiple parameter values from out of an object.

```

[HttpPost]

public HttpResponseMessage PostAuthor(JObject jsonData)
{
    dynamic json = jsonData;

    JObject jauthor = json.Author;

    string token = json.Token;

    var author = jauthor.ToObject<Author>();

    //Usual code to store data in the database

    return Request.CreateResponse(HttpStatusCode.OK, "Success...");
}

```

Another way to solve this is by using FormDataCollection. Incidentally, FormDataCollection is a key / value pair collection much like the FormCollection in MVC.

```

[HttpPost]

public HttpResponseMessage PostAuthor(FormDataCollection form)
{
    var author = form.Get("Author");

    var token = form.Get("Token");

    //Usual code to store data in the database

    return Request.CreateResponse(HttpStatusCode.OK, "Success...");
}

```

}

Thanks to Web API framework extensibility; you can also create your own custom parameter binder by extending the `HttpParameterBinding` class to provide support for multiple parameter binding.

This article is published as part of the IDG Contributor Network. [Want to Join?](#)

<http://www.infoworld.com/article/3136743/application-development/how-to-pass-multiple-parameters-to-web-api-controller-methods.html>

Passing multiple POST parameters to Web API Controller Methods

 May 08, 2012 - from Maui, Hawaii

 [34 comments](#)

[Tweet](#)



ASP.NET Web API introduces a new API for creating REST APIs and making AJAX callbacks to the server. This new API provides a host of new great functionality that unifies many of the features of many of the various AJAX/REST APIs that Microsoft created before it - ASP.NET AJAX, WCF REST specifically - and combines them into a whole more consistent API. Web API addresses many of the concerns that developers had with these older APIs, namely that it was very difficult to build consistent REST style resource APIs easily.

While Web API provides many new features and makes many scenarios much easier, a lot of the focus has been on making it

easier to build REST compliant APIs that are focused on resource based solutions and HTTP verbs. But RPC style calls that are common with AJAX callbacks in Web applications, have gotten a lot less focus and there are a few scenarios that are not that obvious, especially if you're expecting Web API to provide functionality similar to ASP.NET AJAX style AJAX callbacks.

RPC vs. 'Proper' REST

RPC style HTTP calls mimic calling a method with parameters and returning a result. Rather than mapping explicit server side resources or 'nouns' RPC calls tend simply map a server side operation, passing in parameters and receiving a typed result where parameters and result values are marshaled over HTTP. Typically RPC calls - like SOAP calls - tend to always be POST operations rather than following HTTP conventions and using the GET/POST/PUT/DELETE etc. verbs to implicitly determine what operation needs to be fired.

RPC might not be considered 'cool' anymore, but for typical private AJAX backend operations of a Web site I'd wager that a large percentage of use cases of Web API will fall towards RPC style calls rather than 'proper' REST style APIs. Web applications that have needs for things like live validation against data, filling data based on user inputs, handling small UI updates often don't lend themselves very well to limited HTTP verb usage. It might not be what the cool kids do, but I don't see RPC calls getting replaced by proper REST APIs any time soon. Proper REST has its place - for 'real' API scenarios that manage and publish/share resources, but for more transactional operations RPC seems a better choice and much easier to implement than trying to shoehorn a boatload of endpoint methods into a few HTTP verbs.

In any case Web API does a good job of providing both RPC abstraction as well as the HTTP Verb/REST abstraction. RPC works well out of the box, but there are some differences especially if you're coming from ASP.NET AJAX service or WCF Rest when it comes to multiple parameters.

Action Routing for RPC Style Calls

If you've looked at Web API demos you've probably seen a bunch of examples of how to create HTTP Verb based routing endpoints. Verb based routing essentially maps a controller and then uses HTTP verbs to map the methods that are called in response to HTTP requests. This works great for resource APIs but doesn't work so well when you have many operational methods in a single controller. HTTP Verb routing is limited to the few HTTP verbs available (plus separate method signatures) and - worse than that - you can't easily extend the controller with custom routes or action routing beyond that.

Thankfully Web API also supports Action based routing which allows you create RPC style endpoints fairly easily:

```
RouteTable.Routes.MapHttpRoute(  
    name: "AlbumRpcApiAction",  
    routeTemplate: "albums/{action}/{title}",  
    defaults: new  
    {  
        title = RouteParameter.Optional,  
        controller = "AlbumApi",  
        action = "GetAblums"  
    }  
);
```

This uses traditional MVC style {action} method routing which is different from the HTTP verb based routing you might have read a bunch about in conjunction with Web API. Action based routing like above lets you specify an end point method in a Web API controller either via the {action} parameter in the route string or via a default value for custom routes.

Using routing you can pass multiple parameters either on the route itself or pass parameters on the query string, via ModelBinding or content value binding. For most common scenarios this actually works very well. As long as you are passing either a single complex type via a POST operation, or multiple simple types via query string or POST buffer, there's no issue. But if you need to pass multiple

parameters as was easily done with WCF REST or ASP.NET AJAX things are not so obvious.

Web API has no issue allowing for single parameter like this:

```
[HttpPost]
public string PostAlbum(Album album)
{
    return String.Format("{0} {1:d}", album.AlbumName,
album.Entered);
}
```

There are actually two ways to call this endpoint:

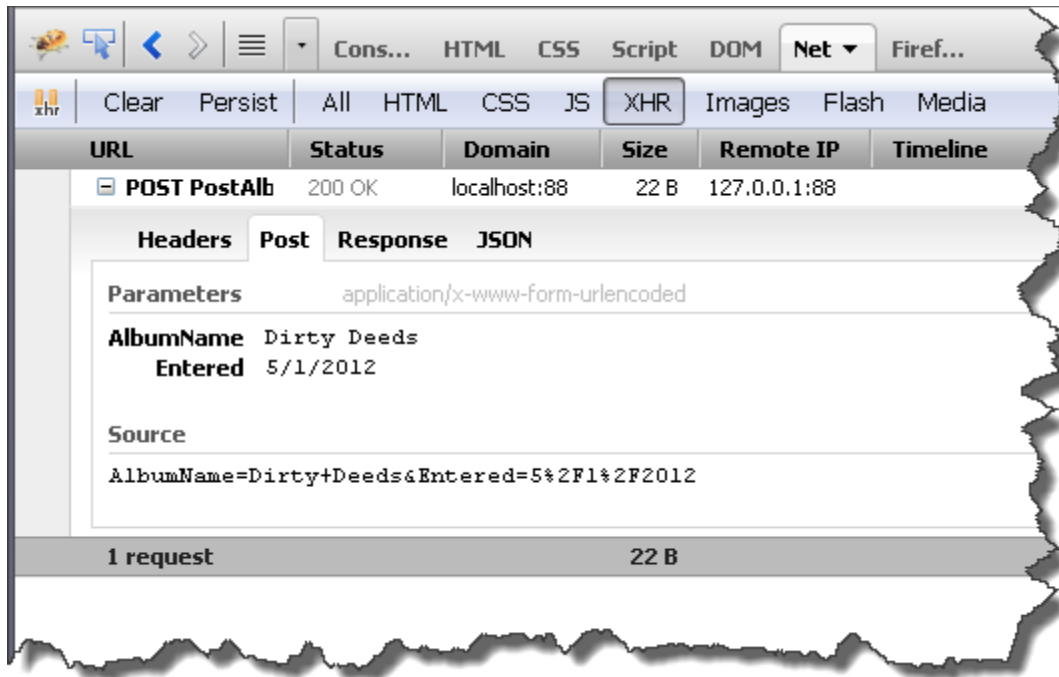
albums/PostAlbum

Using the Model Binder with plain POST values

In this mechanism you're sending plain urlencoded POST values to the server which the ModelBinder then maps the parameter. Each property value is matched to each matching POST value. This works similar to the way that MVC's ModelBinder works. Here's how you can POST using the ModelBinder and jQuery:

```
$.ajax(
{
    url: "albums/PostAlbum",
    type: "POST",
    data: { AlbumName: "Dirty Deeds", Entered: "5/1/2012" },
    success: function (result) {
        alert(result);
    },
    error: function (xhr, status, p3, p4) {
        var err = "Error " + " " + status + " " + p3;
        if (xhr.responseText && xhr.responseText[0] == "{")
            err = JSON.parse(xhr.responseText).message;
        alert(err);
    }
});
```

Here's what the POST data looks like for this request:



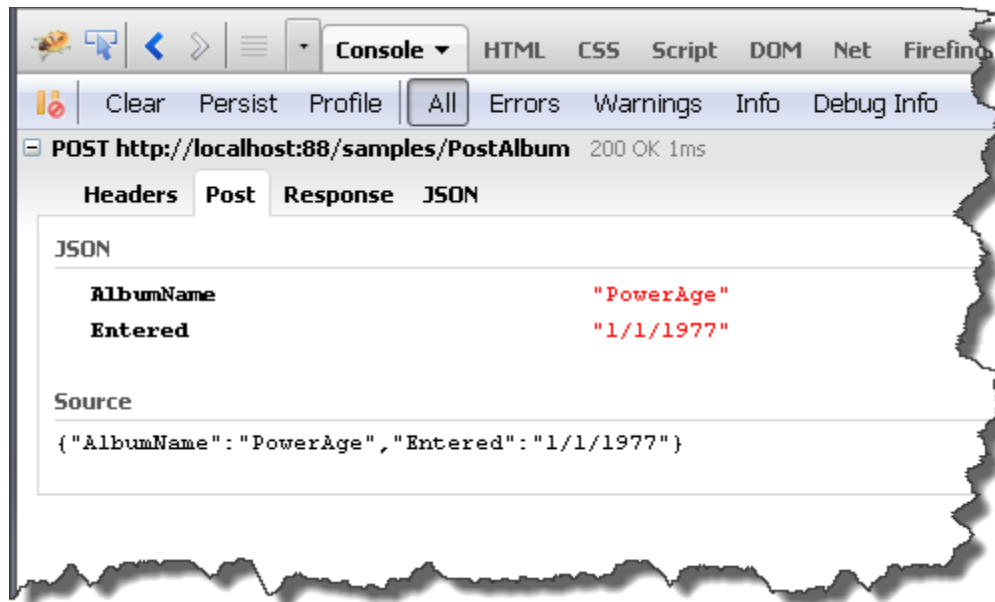
The model binder and its straight form based POST mechanism is great for posting data directly from HTML pages to model objects. It avoids having to do manual conversions for many operations and is a great boon for AJAX callback requests.

Using Web API JSON Formatter

The other option is to post data using a JSON string. The process for this is similar except that you create a JavaScript object and serialize it to JSON first.

```
album = {  
    AlbumName: "PowerAge",  
    Entered: new Date(1977,0,1)  
}  
$.ajax(  
    {  
        url: "albums/PostAlbum",  
        type: "POST",  
        contentType: "application/json",  
        data: JSON.stringify(album),  
        success: function (result) {  
            alert(result);  
        }  
    }  
});
```

Here the data is sent using a JSON object rather than form data and the data is JSON encoded over the wire.



The trace reveals that the data is sent using plain JSON (Source above), which is a little more efficient since there's no UrlEncoding that occurs.

BTW, notice that WebAPI automatically deals with the date. I provided the date as a plain string, rather than a JavaScript date value and the Formatter and ModelBinder both automatically map the date properly to the Entered DateTime property of the Album object.

Passing multiple Parameters to a Web API Controller

Single parameters work fine in either of these RPC scenarios and that's to be expected. ModelBinding always works against a single object because it maps a model. But what happens when you want to pass multiple parameters?

Consider an API Controller method that has a signature like the following:

```
[HttpPost]  
public string PostAlbum(Album album, string userToken)
```

Here I'm asking to pass two objects to an RPC method. Is that possible? This used to be fairly straight forward either with WCF REST and ASP.NET AJAX ASMX services, but as far as I can tell this is not directly possible using a POST operation with WebAPI.

There are a few workarounds that you can use to make this work:

Use both POST *and* QueryString Parameters in Conjunction

If you have both complex and simple parameters, you can pass simple parameters on the query string. The above would actually work with:

*/album/PostAlbum?**userToken=sekkritt***

but that's not always possible. In this example it might not be a good idea to pass a user token on the query string though. It also won't work if you need to pass multiple complex objects, since query string values do not support complex type mapping. They only work with simple types.

Use a single Object that wraps the two Parameters

If you go by service based architecture guidelines every service method should always pass and return a single value only. The input should wrap potentially multiple input parameters and the output should convey status as well as provide the result value. You typically have a xxxRequest and a xxxResponse class that wraps the inputs and outputs.

Here's what this method might look like:

```
public PostAlbumResponse PostAlbum(PostAlbumRequest request)
{
    var album = request.Album;
    var userToken = request.UserToken;

    return new PostAlbumResponse()
    {
        IsSuccess = true,
```

```
        Result = String.Format("{0} {1:d} {2}", album.AlbumName,
album.Entered,userToken)
    };
}
```

with these support types:

```
public class PostAlbumRequest
{
    public Album Album { get; set; }
    public User User { get; set; }
    public string UserToken { get; set; }
}

public class PostAlbumResponse
{
    public string Result { get; set; }
    public bool IsSuccess { get; set; }
    public string ErrorMessage { get; set; }
}
```

To call this method you now have to assemble these objects on the client and send it up as JSON:

```
var album = {
    AlbumName: "PowerAge",
    Entered: "1/1/1977"
}
var user = {
    Name: "Rick"
}
var userToken = "sekkritt";

$.ajax(
{
    url: "samples/PostAlbum",
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify({ Album: album, User: user, UserToken:
userToken } ),
    success: function (result) {
        alert(result.Result);
    }
}
```

```
}  
});
```

I assemble the individual types first and then combine them in the data: property of the \$.ajax() call into the actual object passed to the server, that mimics the structure of PostAlbumRequest server class that has Album, User and UserToken properties.

This works well enough but it gets tedious if you have to create Request and Response types for each method signature. If you have common parameters that are always passed (like you always pass an album or usertoken) you might be able to abstract this to use a single object that gets reused for all methods, but this gets confusing too: Overload a single 'parameter' too much and it becomes a nightmare to decipher what your method actual can use.

Use JObject to parse multiple Property Values out of an Object

If you recall, ASP.NET AJAX and WCF REST used a 'wrapper' object to make default AJAX calls. Rather than directly calling a service you always passed an object which contained properties for each parameter:

```
{ parm1: Value, parm2: Value2 }
```

WCF REST/ASP.NET AJAX would then parse this top level property values and map them to the parameters of the endpoint method.

This automatic type wrapping functionality is no longer available directly in Web API, but since Web API now uses [JSON.NET](#) for it's JSON serializer you can actually simulate that behavior with a little extra code. You can use the JObject class to receive a dynamic JSON result and then using the dynamic cast of JObject to walk through the child objects and even parse them into strongly typed objects.

Here's how to do this on the API Controller end:

```
[HttpPost]
```

```
public string PostAlbum(JObject jsonData)
{
    dynamic json = jsonData;
    JObject jalbum = json.Album;
    JObject juser = json.User;
    string token = json.UserToken;

    var album = jalbum.ToObject<Album>();
    var user = juser.ToObject<User>();

    return String.Format("{0} {1} {2}", album.AlbumName,
user.Name, token);
}
```

This is clearly not as nice as having the parameters passed directly, but it works to allow you to pass multiple parameters and access them using Web API.

JObject is JSON.NET's generic object container which sports a nice *dynamic* interface that allows you to walk through the object's properties using standard 'dot' object syntax. All you have to do is cast the object to *dynamic* to get access to the property interface of the JSON type.

Additionally JObject also allows you to parse JObject instances into strongly typed objects, which enables us here to retrieve the two objects passed as parameters from this jquery code:

```
var album = {
    AlbumName: "PowerAge",
    Entered: "1/1/1977"
}
var user = {
    Name: "Rick"
}
var userToken = "sekkritt";

$.ajax(
{
    url: "samples/PostAlbum",
    type: "POST",
    contentType: "application/json",
```

```
data: JSON.stringify({ Album: album, User: user, UserToken:
userToken })),
    success: function (result) {
        alert(result);
    }
});
```

Summary

ASP.NET Web API brings many new features and many advantages over the older Microsoft AJAX and REST APIs, but realize that some things like passing multiple strongly typed object parameters will work a bit differently. It's not insurmountable, but just knowing what options are available to simulate this behavior is good to know.

Now let me say here that it's probably not a good practice to pass a bunch of parameters to an API call. Ideally APIs should be closely factored to accept single parameters or a single content parameter at least along with some identifier parameters that can be passed on the querystring. But saying that doesn't mean that occasionally you don't run into a situation where you have the need to pass several objects to the server and all three of the options I mentioned might have merit in different situations.

For now I'm sure the question of how to pass multiple parameters will come up quite a bit from people migrating WCF REST or ASP.NET AJAX code to Web API. At least there are options available to make it work.

<https://weblog.west-wind.com/posts/2012/may/08/passing-multiple-post-parameters-to-web-api-controller-methods>