ASP.NET WEB API

1. What is ASP.NET Web API

**In this video we will discuss** 

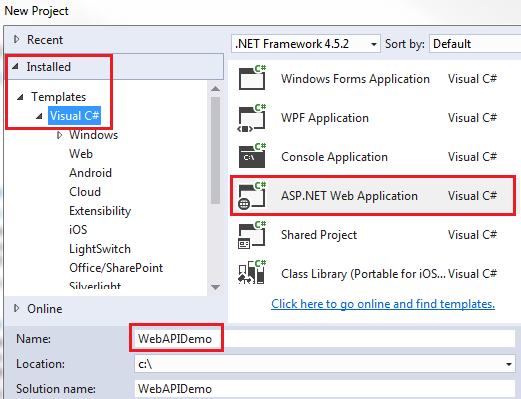
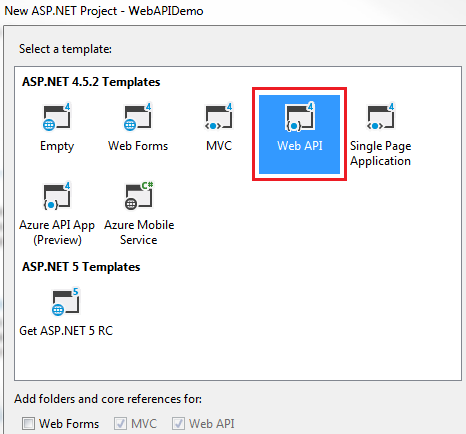
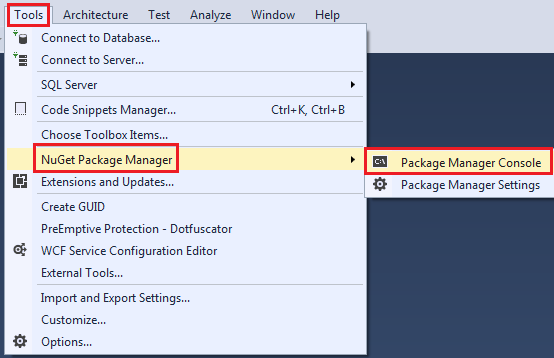
* What is ASP.NET Web API
* What are RESTful services
* Difference between WCF and Web API. When to use WCF over ASP.NET Web API and vice versa

**What is ASP.NET Web API ?**  
The term API stands for ‘Application Programming Interface’. ASP.NET Web API is a framework for building Web API’s, i.e. HTTP based services on top of the .NET Framework. The most common use case for using Web API is for building RESTful services. These services can then be consumed by a broad range of clients like  
1. Browsers  
2. Mobile applications  
3. Desktop applications  
4. IOTs  
  
**What are IOTs**  
The term IOTs stands for Internet Of Things. Internet Of Things are the objects or devices that have an IP address and can communicate over the internet with other internet enabled devices and objects. Examples for IoT include security systems, electronic appliances, thermostats, cars etc..., in addition to desktops, laptops, and smart phones.  
  
One important thing to keep in mind is that, though ASP.NET Web API framework is widely used to create RESTful services, it can also be used to create services that are not RESTful. In short, ASP.NET Web API framework does not dictate any specific architeture style for creating services. In this video, we will discuss creating RESTful services from scratch using ASP.NET Web API framework.  
  
**What are RESTful services**  
REST stands for Representational State Transfer. REST was first introduced in the year 2000 by Roy Fielding as part of his doctoral dissertation. REST is an architectural pattern for creating an API that uses HTTP as its underlying communication method. The REST architectural pattern specifies a set of constraints that a system should adhere to. Here are the REST constraints.  
  
**Client Server constraint** - This is the first constraint. Client sends a request and the server sends a response. This separation of concerns supports the independent evolution of the client-side logic and server-side logic.  
  
**Stateless constraint** - The next constraint is the stateless constraint. The communication between the client and the server must be stateless between requests. This means we should not be storing anything on the server related to the client. The request from the client should contain all the necessary information for the server to process that request. This ensures that each request can be treated independently by the server.  
  
**Cacheable constraint** - Some data provided by the server like list of products, or list of departments in a company does not change that often. This constraint says that let the client know how long this data is good for, so that the client does not have to come back to the server for that data over and over again.  
  
**Uniform Interface**- The uniform interface constraint defines the interface between the client and the server. To understand the uniform interface constraint, we need to understand what a resource is and the HTTP verbs - GET, PUT, POST & DELETE. In the context of a REST API, resources typically represent data entities. Product, Employee, Customer etc are all resources. The HTTP verb (GET, PUT, POST, DELETE) that is sent with each request tells the API what to do with the resource. Each resource is identified by a specific URI (Uniform Resource Identifier). The following table shows some typical requests that you see in an API

| **Resource** | **Verb** | **Outcome** |
| --- | --- | --- |
| /Employees | GET | Gets list of employees |
| /Employee/1 | GET | Gets employee with Id = 1 |
| /Employees | POST | Creates a new employee |
| /Employee/1 | PUT | Updates employee with Id = 1 |
| /Employee/1 | DELETE | Deletes employee with Id = 1 |

Another concept related to Uniform Interface is **HATEOAS**. HATEOAS stands for Hypermedia as the Engine of Application State. All this means is that in each request there will be set of hyperlinks that let's you know what other actions can be performed on the resource. If this is not clear at the moment, don't worry, we will discuss this in a later video.  
  
There are 2 other constraints as well  
Layered System  
Code on Demand (optional)  
  
Code on Demand constraint is optional. We will discuss these 2 constraints in a later video.  
  
**Difference between WCF and Web API. When to choose one over the other?**  
WCF (Windows Communication Foundation) - One of the choices available in .NET for creating RESTful services is WCF. The problem with WCF is that, a lot of configuration is required to turn a WCF service into a RESTful service. The more natural choice for creating RESTful services is ASP.NET Web API, which is specifically created for this purpose.  
  
WCF is more suited for building services that are transport/protocol independent. For example, you want to build a single service, that can be consumed by 2 different clients - Let's say, a Java client and .NET client. Java client expects transport protocol to be HTTP and message format to be XML for interoperability, where as the .NET client expects the protocol to be TCP and the message format to be binary for performance. For this scenario WCF is the right choice. What we do here is create a single WCF service, and then configure 2 end points one for each client (i.e one for the Java client and the other for the .NET client). If you are new to WCF, please watch our WCF video series. I will have the link available in the description of this video.  
  
There is nothing wrong to use WCF to create RESTful services. It's just that it's a bit more complex and configuration can be a headache. If you are stuck with .NET 3.5 or you have an existing SOAP service you must support but want to add REST to reach more clients, then use WCF.   
  
If you don't have the limitation of .NET 3.5 and you want to a create brand new restful service then use ASP.NET Web API.  
  
In our upcoming videos in this series, we will discuss creating RESTful services from scratch using ASP.NET Web API framework.

### Creating a Web API Project

1. **Creating a new ASP.NET Web API Project**  
   1. Open Visual Studio and select File - New - Project  
     
   2. In the "New Project" window  
       Select "Visual C#" under "Installed - Templates"  
       From the middle pane select, ASP.NET Web Application  
       Name the project "WebAPIDemo" and click "OK"   
     
      
     
   3. On the next window, select "Web API" and click "OK"   
     
      
     
   While creating the Web API project, you may get the following errors  
   Package Installation Error - Could not add all required packages to the project. The following packages failed to install from 'C:\Program Files (x86)\Microsoft ASP.NET\ASP.NET Web Stack 5\Packages'  
     
   Failed to initialize the PowerShell host.  
     
   **If you do then follow the below steps which may help resolve the issue**  
   1. Close all instances of Visual Studio  
     
   2. Open Windows Powershell as an Administrator and execute the following command  
   Set ExecutionPolicy AllSigned  
     
   3. Run Visual Studio 2015 as an Administrator  
     
   4. Open Package Manager Console window in Visual Studio. To do this click on Tools - NuGet Package Manager - Package Manager Console   
     
      
     
   5 In the Package Manager Console, type [R] for Run once and press the Enter key  
     
   At this point, you will be able to create a new Web API project.  
     
   **Now, let us explore and understand the Web API code auto-generated by Visual Studio**  
     
   1. If you have worked with ASP.NET MVC, then project folder structure should be familiar to you. Notice with in the **Controllers** folder we have **ValuesController** which inherits from ApiController class that is present in System.Web.Http namespace. This is different from the MVC controller. The MVC Controller class inherits from the Controllerclass that is present in System.Web.Mvc namespace. The HomeController class which is an MVC controller inherits from the Controller class.  
     
   2. Notice in the **ValuesController** class we have methods (Get, Put, Post & Delete) that map to the HTTP verbs (GET, PUT, POST, DELETE) respectively. We have 2 overloaded versions of Get() method - One without any parameters and the other with id parameter. Both of these methods respond to the **GET** http verb depending on whether the id parameter is specified in the URI or not.  
     
   3. Now let's look at the default route that is in place for our Web API project. We have the Application\_Start() event handler In Global.asax file. This event is raised when the application starts. In the Application\_Start() event handler method we have configuration for Filters, Bundles etc. The one that we are interested in is the configuration for our Web API project, which is in WebApiConfig.Register() method. Right click on WebApiConfig.Register and select "Go To Definition" from the context menu. This will take you to the Register() method in the WebApiConfig class. This class is in App\_Start folder.  
     
   4. In the Register() method we have the default route configured for our Web API project. Web API routes are different from the MVC routes. You can find the MVC routes in RouteConfig.cs file in App\_Start folder.  
     
   5. The default Web API route starts with the word api and then / and then the name of the controller and another / and an optiontion id parameter.  
   "api/{controller}/{id}"  
     
   6. At this point if we use the following URI in the browser, we get an error - Authorization has been denied for this request.  
   http://localhost/api/values  
     
   7. To get rid of this error, comment Authorize attribute on the **ValuesController** class. This is related to security which we will discuss in a later video.  
     
   8. Now if you visit, http://localhost/api/values, you should see the following XML as the result
2. <ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
4. <string>value1</string>
5. <string>value2</string>
6. </ArrayOfstring>
7. 9. Let us understand what is going on here. The name of the controller is values. So if we use a URI http://localhost:portnumber/api/values, then the web api is going to look for a controller with name **Values + the word controller** in your project. So if you have specified values in the URI it is going to look for **ValuesController**, if you specify **Products**, then it is going to look for **ProductsController**. This is all by convention and works this way out of the box.  
     
   10. In a real world application this might be the domain name, for example  
   http://pragimtech.com/api/values  
     
   11. The browser is issuing a GET request which maps to the Get() method in the **ValuesController** class. The GET() in the values controller is returning value1 and value2 which is what we see in the browser.  
     
   12. We have another overload of GET() method which takes Id parameter. Remember with the default route, the id parameter is optional. That is the reason we are able to call the GET method with or without the Id parameter. If the id parameter is specified in the URI, then the Get() method with the parameter in values controller is called   
     
   13. If a controller with the specified name is not found you will get an error. For example, in your project if you comment the ValuesController class in your project and then use the URI /api/values you will get the following error  
     
   No HTTP resource was found that matches the request URI 'http://localhost:15648/api/values'. No type was found that matches the controller named 'values'.  
     
   In our next video we will discuss how to perform the rest of the actions PUT, POST and DELETE.

### 3. HTTP GET PUT POST DELETE

In this video we will discuss HTTP PUT, POST and DELETE verbs. This is continuation to [Part 2](http://csharp-video-tutorials.blogspot.com/2016/08/creating-web-api-project.html). Please watch [Part 2](http://csharp-video-tutorials.blogspot.com/2016/08/creating-web-api-project.html) from [ASP.NET Web API tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhW7yoUOGNOfHurUE6bpOO2b) before proceeding.   
  
  
  
When we talk about a database table row, these are the following 4 actions that we can perform on the row  
C - Create a row  
R - Read a row  
U - Update a row  
D - Delete a row  
  
In the context of an ASP.NET Web API resource these 4 actions correspond to GET, POST, PUT and DELETE as shown in the table below

| **CRUD** | **HTTP Verb** |
| --- | --- |
| Create | POST |
| Read | GET |
| Update | PUT |
| Delete | DELETE |

Let us now understand some terms and concepts related to HTTP request and response system.  
  
**Request Verbs :** These HTTP verbs (GET, POST, PUT & DELETE) describe what should be done with the resource. For example do you want to create, read, update or delete an entity. GET, PUT, POST and DELETE http verbs are the most commonly used one's. For the complete list of the HTTP verbs, please check <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>  
  
**Request Header :** When a client sends request to the server, the request contains a header and a body. The request header contains additional information such as what type of response is required. For example, do you want the response to be in XML or JSON.  
  
**Request Body :** Request Body contains the data to send to the server. For example, a POST request contains the data for the new item that you want to create. The data format may be in XML or JSON.  
  
**Response Body :** The Response Body contains the data sent as response from the server. For example, if the request is for a specific product, the response body includes product details either in XML or JSON format.  
  
**Response Status codes :** These are the HTTP status codes, that give the client details on the status of the request. Some of the common status codes are 200/OK, 404/Not Found, 204/No Content. For the complete list of HTTP status codes and what they mean, please visit <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>  
  
We will use a tool called fiddler to perform POST, PUT & DELETE actions. You can download fiddler from the following link  
<https://www.telerik.com/download/fiddler>  
  
Modify the **ValuesController**as shown below, so that it can support POST, PUT and DELETE actions.

public class ValuesController : ApiController

{

    static List<string> strings = new List<string>()

    {

        "value0", "value1", "value2"

    };

    // GET api/values

    public IEnumerable<string> Get()

    {

        return strings;

    }

    // GET api/values/5

    public string Get(int id)

    {

        return strings[id];

    }

    // POST api/values

    public void Post([FromBody]string value)

    {

        strings.Add(value);

    }

    // PUT api/values/5

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

    {

        strings[id] = value;

    }

    // DELETE api/values/5

    public void Delete(int id)

    {

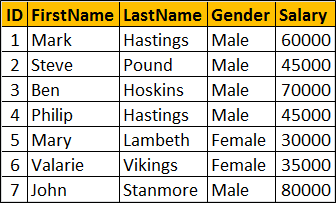
        strings.RemoveAt(id);

    }

}

Post(), Put() and Delete() methods in our **ValuesController**are returning void. That is the reason we are getting status code 204 No Content. An action that returns void will send status code 204 No Content. With ASP.NET Web API, we can control what status codes these methods return. We will discuss how to do that in a later video.

4. ASP .NET Web API and SQL Server

In this video we will discuss creating ASP.NET Web API service that returns data from a SQL Server database. We will be using this service as the basis for understanding many of the Web API concepts in our upcoming videos.   
  
  
  
The Web API **EmployeeService**that we will be building will retrieve and return the data from the following SQL Server Employees table. We will be using Entity Framework to retrieve data from the database. You can use any technology of your choice to retrieve data from the database. For example you can even use raw ADO.NET.   
   
  
Execute the following SQL Script using SQL Server Management studio. This script creates  
1. EmployeeDB database  
2. Creates the Employees table and populate it with sample data

Create Database EmployeeDB

Go

Use EmployeeDB

Go

Create table Employees

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50),

     Salary int

)

Go

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)

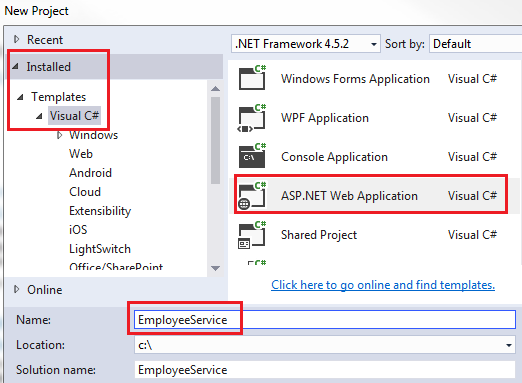
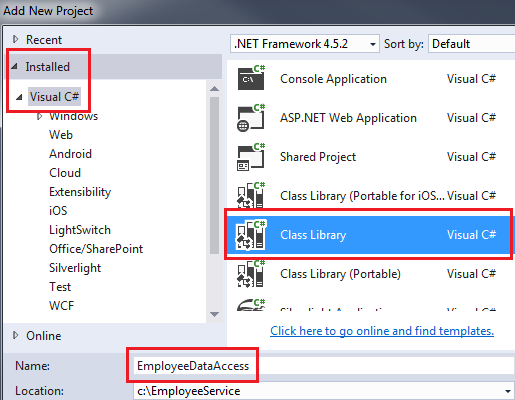
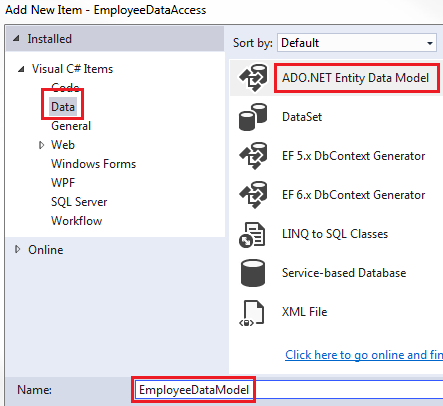
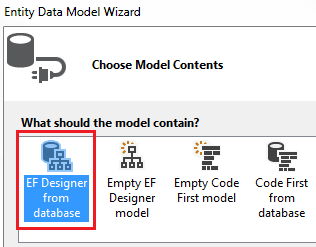
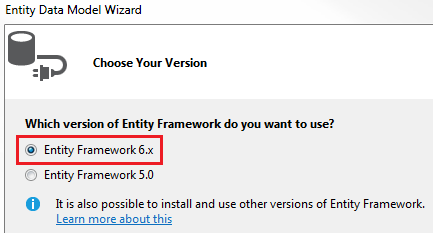
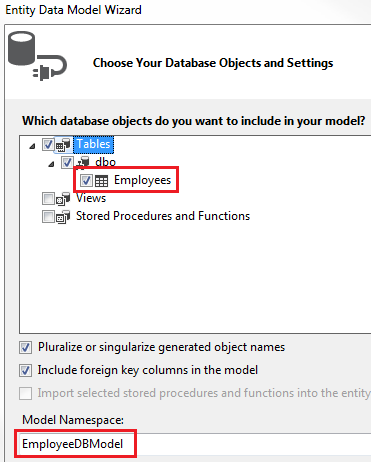
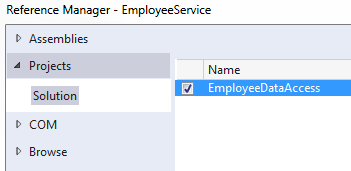
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000)

Go

**Creating a new ASP.NET Web API Project**  
1. Open Visual Studio and select File - New - Project  
  
2. In the "New Project" window  
    Select "Visual C#" under "Installed - Templates"  
    From the middle pane select, ASP.NET Web Application  
    Name the project "EmployeeService" and click "OK"   
   
  
3. On the next window, select "Web API" and click "OK"  
  
At this point you should have the Web API project created.  
  
**Adding ADO.NET Entity Data Model to retrieve data**  
1. We will have the Entity Model in a separate project.  
  
2. Right click on EmployeeService solution in the Solution Explorer and select Add - New Project  
  
3. In the Add New Project window  
    Select Visual C# from the left pane  
    Class Library Project from the Middle pane  
    Name the project EmployeeDataAccess and click OK   
   
  
4. Right click on EmployeeDataAccess project and select Add - New Item  
  
5. In the "Add New Item" window  
    Select "Data" from the left pane  
    Select ADO.NET Entity Data Model from the middle pane  
    In the Name text box, type EmployeeDataModel and click Add   
   
  
6. On the Entity Data Model Wizard, select "EF Designer from database" option and click next   
   
  
7. On the next screen, click "New Connection" button   
  
8. On "Connection Properties" window, set  
    Server Name = (local)  
    Authentication = Windows Authentication  
    Select or enter a database name = EmployeeDB  
    Click OK and then click Next  
  
9. On the nex screen, select Entity Framework 6.x   
   
  
10. On the nex screen, select "Employees" table and click Finish.   
   
  
**Using the Entity Data Model in EmployeeService project**  
1. Right click on the references folder in the EmployeeService project and select "Add Reference"  
  
2. On the "Reference Manager" screen select "EmployeeDataAccess" project and click OK.    
   
  
**Adding Web API Controller**  
1. Right click on the Controllers folder in EmployeeService project and select Add - Controller  
  
2. Select "Web API 2 Controller - Empty" and click "Add"  
  
3. On the next screen set the Controller Name = EmployeesController and click Add  
  
4. Copy and paste the following code in EmployeesController.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Web.Http;

using EmployeeDataAccess;

namespace EmployeeService.Controllers

{

    public class EmployeesController : ApiController

    {

        public IEnumerable<Employee> Get()

        {

            using(EmployeeDBEntities entities = new EmployeeDBEntities())

            {

                return entities.Employees.ToList();

            }

        }

        public Employee Get(int id)

        {

            using (EmployeeDBEntities entities = new EmployeeDBEntities())

            {

                return entities.Employees.FirstOrDefault(e => e.ID == id);

            }

        }

    }

}

5. At this point build the solution and navigate to /api/employees. You will get the following error.  
No connection string named 'EmployeeDBEntities' could be found in the application config file.  
  
6. This is because "Entity Framework" is looking for EmployeeDBEntities connection string in the web.config file of EmployeeService project. EmployeeDBEntities connection string is actually in App.config file of EmployeeDataAccess class library project. Include a copy of this connection string in web.config file.  
  
At this point when you navigate to /api/employees you should see all employees and when you navigate to /api/employees/1 you should see all the details of the employee whose Id=1  
  
In our next video we will discuss the concept of **Content Negotiation** in Web API

### 5. ASP.NET Web API Content Negotiation

One of the standards of the RESTful service is that, the client should have the ability to decide in which format they want the response - XML, JSON etc. A request that is sent to the server includes an Accept header. Using the Accept header the client can specify the format for the response. For example  
  
Accept: application/xml returns XML  
Accept: application/json returns JSON  
  
Depending on the Accept header value in the request, the server sends the response. This is called Content Negotiation.   
  
**So what does the Web API do when we request for data in a specific format**  
The Web API controller generates the data that we want to send to the client. For example, if you have asked for list of employees. The controller generates the list of employees, and hands the data to the Web API pipeline which then looks at the Accept header and depending on the format that the client has requested, Web API will choose the appropriate formatter. For example, if the client has requested for XML data, Web API uses XML formatter. If the client has requested for JSON data, Web API uses JSON formatter. These formatters are called Media type formatters.  
  
ASP.NET Web API is greatly extensible. This means we can also plugin our own formatters, for custom formatting the data.  
  
Multiple values can also be specified for the Accept header. In this case, the server picks the first formatter which is a JSON formatter and formats the data in JSON.  
Accept: application/xml,application/json  
  
You can also specify quality factor. In the example below, xml has higher quality factor than json, so the server uses XML formatter and formats the data in XML.  
application/xml;q=0.8,application/json;q=0.5  
  
If you don't specify the Accept header, by default the Web API returns JSON data.  
  
When the response is being sent to the client in the requested format, notice that the Content-Type header of the response is set to the appropriate value. For example, if the client has requested application/xml, the server send the data in XML format and also sets the Content-Type=application/xml.  
  
The formatters are used by the server for both request and response messages. When the client sends a request to the server, we set the Content-Type header to the appropriate value to let the server know the format of the data that we are sending. For example, if the client is sending JSON data, the Content-Type header is set to application/json. The server knows it is dealing with JSON data, so it uses JSON formatter to convert JSON data to .NET Type. Similarly when a response is being sent from the server to the client, depending on the Accept header value, the appropriate formatter is used to convert .NET type to JSON, XML etc.  
  
It's also very easy to change the serialization settings of these formatters. For example, if you want the JSON data to be properly indented and use camel case instead of pascal case for property names, all you have to do is modify the serialization settings of JSON formatters as shown below. With our example this code goes in WebApiConfig.cs file in App\_Start folder.

config.Formatters.JsonFormatter.SerializerSettings.Formatting =

                            Newtonsoft.Json.Formatting.Indented;

config.Formatters.JsonFormatter.SerializerSettings.ContractResolver =

    new CamelCasePropertyNamesContractResolver();

6. ASP.NET Web API MediaTypeFormatter

**In this video we will discuss**

* What is MediaTypeFormatter
* How to return only JSON from ASP.NET Web API Service irrespective of the Accept header value
* How to return only XML from ASP.NET Web API Service irrespective of the Accept header value
* How to return JSON instead of XML from ASP.NET Web API Service when a request is made from the browser
* Point you to an article that describes how to return CSV formatted data from ASP.NET Web API

**What is MediaTypeFormatter**  
MediaTypeFormatter is an abstract class from which JsonMediaTypeFormatter and XmlMediaTypeFormatter classes inherit from. JsonMediaTypeFormatter handles JSON and XmlMediaTypeFormatter handles XML.  
  
**How to return only JSON from ASP.NET Web API Service irrespective of the Accept header value**  
Include the following line in **Register**() method of WebApiConfig.cs file in **App\_Start**folder. This line of code completely removes XmlFormatter which forces ASP.NET Web API to always return JSON irrespective of the Accept header value in the client request. Use this technique when you want your service to support only JSON and not XML.   
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return JSON.

config.Formatters.Remove(config.Formatters.XmlFormatter);

**How to return only XML from ASP.NET Web API Service irrespective of the Accept header value**  
Include the following line in Register() method of WebApiConfig.cs file in App\_Start folder. This line of code completely removes JsonFormatter which forces ASP.NET Web API to always return XML irrespective of the Accept header value in the client request. Use this technique when you want your service to support only XML and not JSON.  
  
config.Formatters.Remove(config.Formatters.JsonFormatter);  
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return XML.  
  
**How to return JSON instead of XML from ASP.NET Web API Service when a request is made from the browser.**  
So here is what we want the service to do  
1. When a request is issued from the browser, the web API service should return JSON instead of XML.   
2. When a request is issued from a tool like fiddler the Accept header value should be respected. This means if the Accept header is set to application/xml the service should return XML and if it is set to application/json the service should return JSON.  
  
There are 2 ways to achieve this  
  
**Approach 1 :**Include the following line in Register() method of WebApiConfig.cs file in App\_Start folder. This tells ASP.NET Web API to use JsonFormatter when a request is made for text/html which is the default for most browsers. The problem with this approach is that Content-Type header of the response is set to text/html which is misleading.

config.Formatters.JsonFormatter.SupportedMediaTypes

    .Add(new MediaTypeHeaderValue("text/html"));

**Approach 2 :** Include the following class in WebApiConfig.cs file in App\_Start folder. 

public class CustomJsonFormatter : JsonMediaTypeFormatter

{

    public CustomJsonFormatter()

    {

        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));

    }

    public override void SetDefaultContentHeaders(Type type, HttpContentHeadersheaders, MediaTypeHeaderValue mediaType)

    {

        base.SetDefaultContentHeaders(type, headers, mediaType);

        headers.ContentType = new MediaTypeHeaderValue("application/json");

    }

}

**Register the formatter:** Place the following line in Register() method of WebApiConfig.cs file in App\_Start folder  
config.Formatters.Add(new CustomJsonFormatter());  
  
With these 2 changes, when a request is issued from the browser you will get JSON formatted data and the Content-Type header of the response is also set to application/json. If you are using tools like fiddler and if you set Accept header to application/xml you will still get XML formatted data.  
  
ASP.NET Web API is an extinsible framework. This means you can also plugin your own custom formatter. For example, if you want the response to be in CSV format, you can create custom CSVMediaTypeFormatter that inherits from the base abstract class MediaTypeFormatter . The following article describes how to do this.  
<http://www.tugberkugurlu.com/archive/creating-custom-csvmediatypeformatter-in-asp-net-web-api-for-comma-separated-values-csv-format>

### 7. Implementing post method in ASP.NET Web API

In this video we will discuss **implementing POST method in ASP.NET Web API**. Post allows us to create a new item.   
  
  
  
We want to add a new Employee to the Employees table. Include the following Post() method in EmployeesController. Notice the Employee object is being passed as parameter to the Post method. The Employee parameter is decorated with [FromBody] attribute. This tells Web API to get employee data from the request body.

public void Post([FromBody] Employee employee)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        entities.Employees.Add(employee);

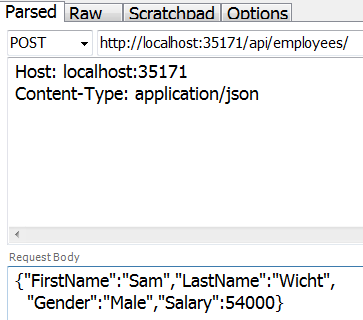
        entities.SaveChanges();

    }

}

**At this point build the solution. Fireup Fiddler and issue a Post request**

1. Set the HTTP verb to POST
2. Content-Type: application/json. This tells that we are sending JSON formatted data to the server
3. In the Request Body, include the employee object that you want to add to the database in JSON format
4. Finally click execute

   
  
This works fine and adds the employee to the database as expected. The problem here is that since the return type of the Post() method is void, we get status code of 204 No Content. When a new item is created, we should actually be returning status code 201 Item Created. With 201 status code we should also include the location i.e URI of the newly created item. To achieve this, change the implementation of the Post() method as shown below. 

public HttpResponseMessage Post([FromBody] Employee employee)

{

    try

    {

        using (EmployeeDBEntities entities = new EmployeeDBEntities())

        {

            entities.Employees.Add(employee);

            entities.SaveChanges();

            var message = Request.CreateResponse(HttpStatusCode.Created, employee);

            message.Headers.Location = new Uri(Request.RequestUri +

                employee.ID.ToString());

            return message;

        }

    }

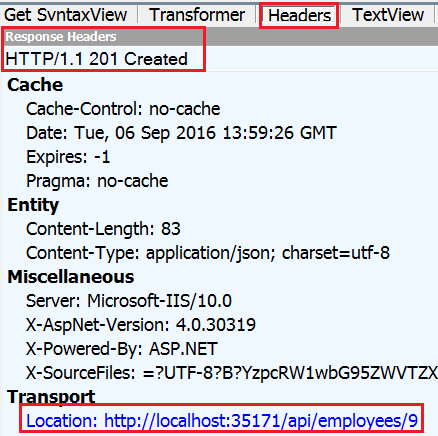
    catch (Exception ex)

    {

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);

    }

}

At this point, issue another Post request from fiddler. Notice in the response header we have status code 201 Created and also the location i.e the URI of the newly created item.   
   
  
Let's also modify Get(int id) method. At the moment when we issue a request for an employee id that does not exist, we get null back and the status code is 200 OK. According to the standards of REST when an item is not found we should be returning 204 Not Found. To achieve this modify the **Get(int id)** method as shown below. 

public HttpResponseMessage Get(int id)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

        if (entity != null)

        {

            return Request.CreateResponse(HttpStatusCode.OK, entity);

        }

        else

        {

            return Request.CreateErrorResponse(HttpStatusCode.NotFound,

                "Employee with Id " + id.ToString() + " not found");

        }

    }

}

At this point, when we issue a request for an employee with ID = 101 which does not exist we get a 404 along with the message "Employee with Id 101 not found"  
  
**Here are the important points to remember**

* If a method return type is void, by default status code 204 No Content is returned.
* When a new item is created, we should be returning status code 201 Item Created.
* With 201 status code we should also include the location i.e URI of the newly created item.
* When an item is not found, instead of returning NULL and status code 200 OK, return 404 Not Found status code along with a meaningful message such as "Employee with Id = 101 not found"