**How to Create Web APIs in ASP.NET Core [.NET 7.0 RESTful pattern]**

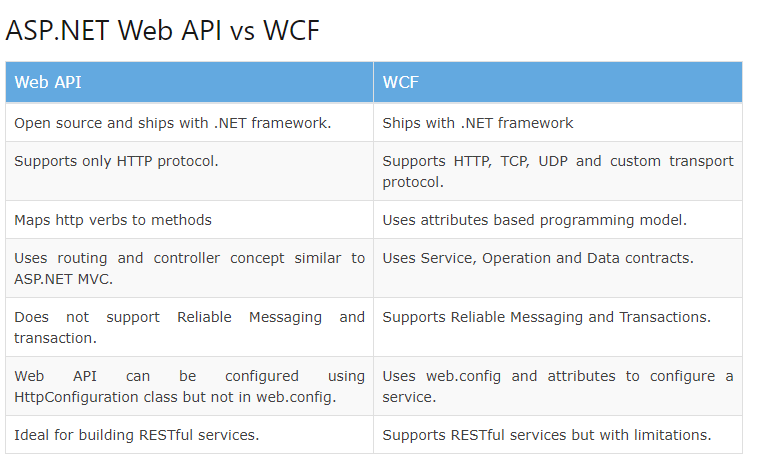
## ASP.NET Web API Characteristics

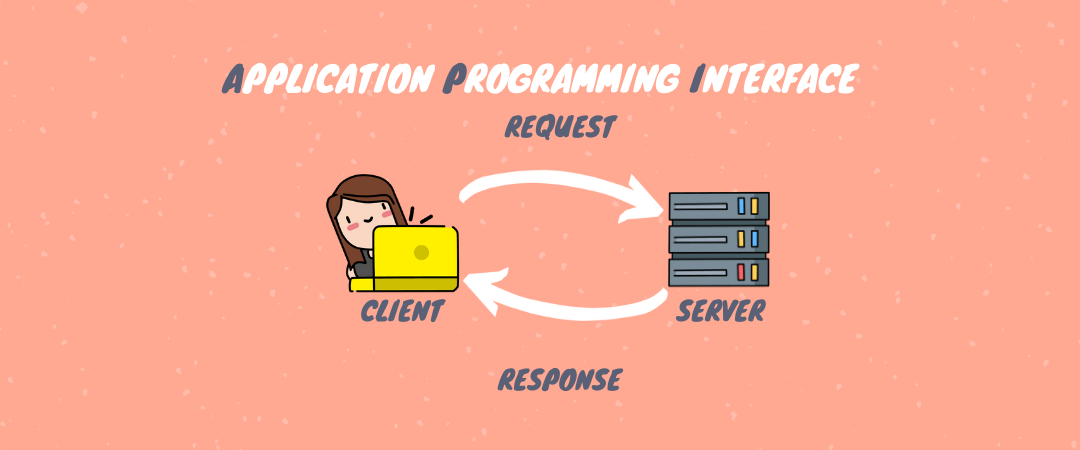
1. ASP.NET Web API is an ideal platform for building RESTful services.
2. ASP.NET Web API is built on top of ASP.NET and supports ASP.NET request/response pipeline
3. ASP.NET Web API maps HTTP verbs to method names.
4. ASP.NET Web API supports different formats of response data. Built-in support for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

## What are some API examples?

The most popular API are the ones which the people use daily, 3 of such APIs are:

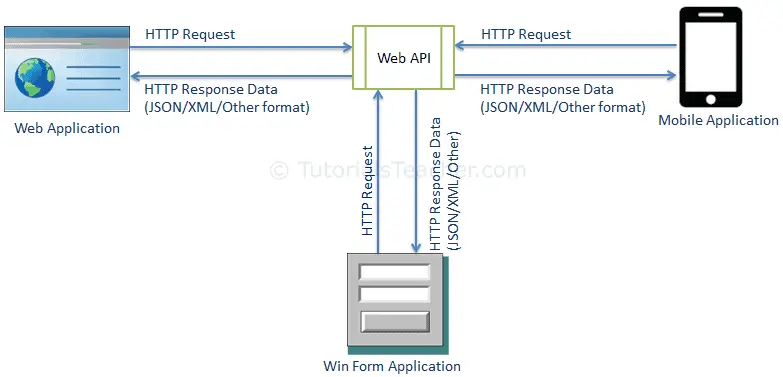
* 1. Skyscanner which is a travel site which lets people research and book travel options for their trips. On their website, when you perform a search, the website calls a bunch of external APIs to fetch the current state of flights, hotels, etc. This is then displayed to you on the browser.
* 2. OpenWeatherMap which provides global weather data via API.
* 3. Yahoo Finance provides current stock prices and market news.

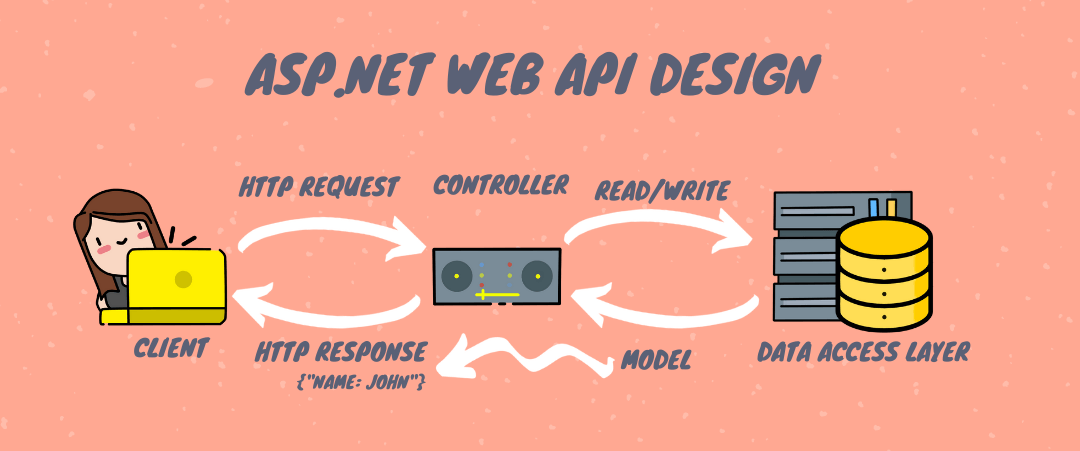
****



## How Do ASP.NET Web APIs Work?

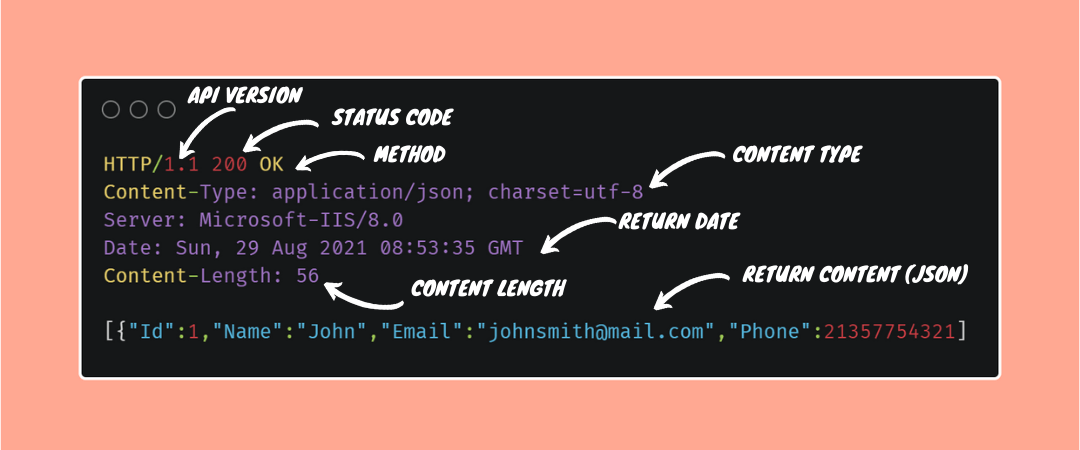
An ASP.NET Core Web API basically consists of one or more controller class that derives from ControllerBase. The ControllerBase class provides many methods and properties that are useful for working with HTTP requests.





As you can see in the image above, a “client” makes an “HTTP request” to the API, which—through the “controller”—identifies the call and makes the Read or Write in the “data access layer.” The “Model” is returned, in this case, a JSON object “Contact: Name” in the “HTTP response.” In simple terms, the API is bridging the “client” and the “data” in a simple and safe way.

**Example console API response**



Now that we’ve seen the basics of Web APIs, let’s create an API and see in practice how it works.

**Web APIs in ASP.NET Core** is very straightforward. You create controllers that have 3 things:

* 1. They should have **[ApiController]** attribute on them. This attribute tells that the controller will server **HTTP API Responses**.
* 2. They should derive from ControllerBase class instead of Controller class.
* 3. They should have attribute routing applied on them like [Route("someUrl/[controller]")].

The controller of a Web API looks like:

[ApiController]

[Route("someURL/[controller]")]

public class ExampleController : ControllerBase

This ASP.NET Core Web API tutorial series is based on .NET 7.0 version and consits of 4 articles.

* 1. *How to Create Web APIs in ASP.NET Core [RESTful pattern]*
* 2. [How to Call Web API in ASP.NET Core [with Codes]](https://www.yogihosting.com/aspnet-core-consume-api/)
* 3. [How to Call Web API from jQuery](https://www.yogihosting.com/aspnet-core-consume-api-jquery/)
* 4. [Call Web API from JavaScript with XMLHttpRequest (XHR)](https://www.yogihosting.com/aspnet-core-web-api-javascript/)

**Page Contents**

* [What is Web API?](https://www.yogihosting.com/aspnet-core-api-controllers/#what-is-web-api)
* [“ControllerBase” vs “Controller” class](https://www.yogihosting.com/aspnet-core-api-controllers/#controllerbase)
* [What is an ApiController attribute](https://www.yogihosting.com/aspnet-core-api-controllers/#what-is-apicontroller)
* [Create the Example Project](https://www.yogihosting.com/aspnet-core-api-controllers/#create)
* [Configuration in the Startup Class](https://www.yogihosting.com/aspnet-core-api-controllers/#startup)
* [Controller for Web API](https://www.yogihosting.com/aspnet-core-api-controllers/#controller)
* [Web API Action Methods](https://www.yogihosting.com/aspnet-core-api-controllers/#actions)
  + [HTTP Attributes](https://www.yogihosting.com/aspnet-core-api-controllers/#http-attributes)
* [[HttpGet] Action Methods](https://www.yogihosting.com/aspnet-core-api-controllers/#httpget)
* [[HttpPost] Action](https://www.yogihosting.com/aspnet-core-api-controllers/#httppost)
* [[HttpPut] Action](https://www.yogihosting.com/aspnet-core-api-controllers/#httpput)
* [[HttpDelete] Action](https://www.yogihosting.com/aspnet-core-api-controllers/#httpdelete)
* [[HttpPatch] Action](https://www.yogihosting.com/aspnet-core-api-controllers/#httppatch)
* [Download Source Codes](https://www.yogihosting.com/aspnet-core-api-controllers/#download)

What is Web API?

**A Web API is an API which can be accessed over the web using HTTP Protocol. For example, we can get the current stock value of any organization right in our browser by calling any Stock Market Web API. A Web API can be built in any technology like ASP.NET Core, JAVA, Python, etc. It can also be consumed in any technology. This makes Web APIs a perfect choice for apps, build on different technologies, to communicate with one another.**

The work of the Web API is to transfer data over the internet. The data is transferred with the HTTP Protocol’s Request methods which are commonly known as HTTP Verbs. The mostly used HTTP Verbs are GET, POST, PUT, PATCH and DELETE. JSON and XML file formats are used by Web APIS to transmit data over the internet.

“ControllerBase” vs “Controller” class

You must not create a **Web API controller** by deriving from the Controller class. This is because the Controller class derives from ControllerBase class and adds support for views, so it’s for handling web pages, not web API requests. There’s an exception to this rule: if you plan to use the same controller for both views and web APIs, then only derive it from Controller class. Check the skeleton of an **ASP.NET Core Web API which derives from ControllerBase class**.

[ApiController]

[Route("api/[controller]")]

public class ReservationController : ControllerBase

{

...

}

The ControllerBase class provides many properties and methods that are useful for handling HTTP requests. Some of these are:



| **Name** | **Description** |
| --- | --- |
| BadRequest | Returns 400 status code. |
| Ok | Returns a 200 status code along with the result object. |
| NotFound | Returns 404 status code. |
| PhysicalFile | Returns a file. |

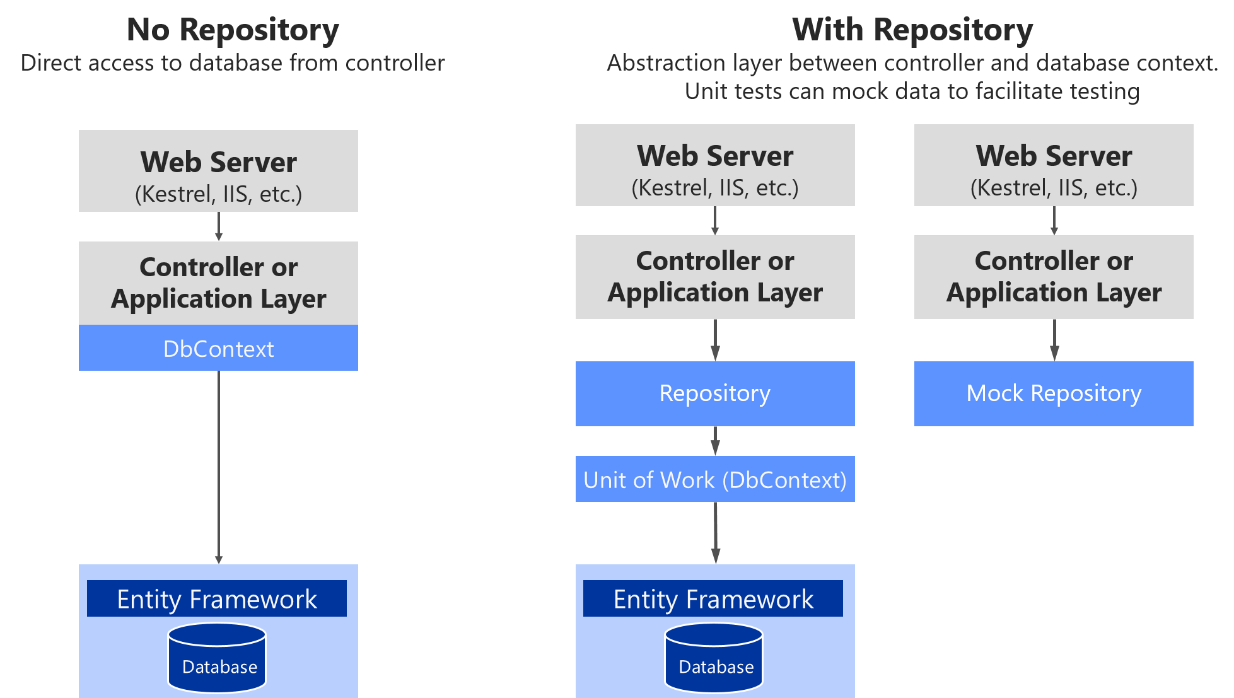
What is an ApiController attribute?

The Web API controller must be applied with **[ApiController]** attribute so that they can perform API specific behaviors. These behaviors are:

* 1. Respond to Attribute Routing.
* 2. Automatically triggers an HTTP 400 response when resource is not found on the server.
* 3. Defines the location at which an action method’s parameter value is found. We use **[FromBody], [FromForm], [FromHeader], [FromQuery], [FromRoute]** attributes to define these locations.
* 4. When action method’s parameter is annotated with the [FromForm] attribute then the multipart/form-data request content type is inferred.
* 5. Enables Web API returned error to be based on the RFC 7807 specification. This enables Web APIs made in different technologies like Java, Ruby, ASP.NET Core etc to communicate with one another without any problem.

Create the Example Project

Create a new project in Visual Studio, choose ASP.NET Core Web APP (MVC) template and name it APIControllers. Select the latest version of the DOT NET framework which is .NET 7.0. I have shown this in the below image.



Create a new class in the DataAccessEFCore Project and name it **Repositories/GenericRepository**

**public abstract *class* RepositoryBase<T> : IRepositoryBase<T> where T : *class***

**{**

**protected readonly ApplicationContext \_context;**

**public RepositoryBase(ApplicationContext context)**

**{**

**\_context = context;**

**}**

**public async Task<IEnumerable<T>> GetAll() => \_context.Set<T>().AsNoTracking();**

**public async Task<T> GetById(int id) => await \_context.Set<T>().FindAsync(id);**

**public void Create(T entity) => \_context.Set<T>().Add(entity);**

**public void Update(T entity) => \_context.Set<T>().Update(entity);**

**public void Delete(T entity) => \_context.Set<T>().Remove(entity);**

**}**

Model & Repository

public interface IRepositoryBase<T> where T : *class*

{

Task<IEnumerable<T>> GetAll();

Task<T> GetById(int id);

void Create(T entity);

void Update(T entity);

void Delete(T entity);

}

This is the Generic Interface, that expose the functions.

* Task<IEnumerable<T>> GetAll() – Get all the lists.
* Task<T> GetById(int id) – Get the list by Id
* void Create(T entity) – Adds new record into context
* void Update(T entity) – Update the record into context
* void Delete(T entity) – Removes the record from the context

Inside the Models folder, add a class called Reservation.cs to it. The class code is given below:

|  |  |
| --- | --- |
|  | namespace APIControllers.Models  {      public class Reservations      {          public int Id { get; set; }          public string Name { get; set; }          public string StartLocation { get; set; }          public string EndLocation { get; set; }      }  } |

Next add a class file called IRepository.cs to the Models folder and used it to define an interface as shown in the below code.

|  |  |
| --- | --- |
|  | namespace APIControllers.Models  {      public interface IRepository      {          IEnumerable<Reservation> Reservations { get; }          Reservation this[int id] { get; }          Reservation AddReservation(Reservation reservation);          Reservation UpdateReservation(Reservation reservation);          void DeleteReservation(int id);      }  } |

Finally add a class file called Repository.cs to the Models folder and used it to define a non-persistent store of reservations. It inherits the IRepository interface we defined earlier.

namespace APIControllers.Models

{

public class Repository : IRepository

{

private Dictionary<int, Reservation> items;

public Repository()

{

items = new Dictionary<int, Reservation>();

new List<Reservation> {

new Reservation {Id=1, Name = "Ankit", StartLocation = "New York", EndLocation="Beijing" },

new Reservation {Id=2, Name = "Bobby", StartLocation = "New Jersey", EndLocation="Boston" },

new Reservation {Id=3, Name = "Jacky", StartLocation = "London", EndLocation="Paris" }

}.ForEach(r => AddReservation(r));

}

public Reservation this[int id] => items.ContainsKey(id) ? items[id] : null;

public IEnumerable<Reservation> Reservations => items.Values;

public Reservation AddReservation(Reservation reservation)

{

if (reservation.Id == 0)

{

int key = items.Count;

while (items.ContainsKey(key)) { key++; };

reservation.Id = key;

}

items[reservation.Id] = reservation;

return reservation;

}

public void DeleteReservation(int id) => items.Remove(id);

public Reservation UpdateReservation(Reservation reservation) => AddReservation(reservation);

}

}

The Repository class creates 3 set of reservation objects when it is instantiated, and since there is no persistent storage, any changes will be lost when the application is stopped or restarted.

The Class contains methods and properties to do **CRUD Operations** and all the reservations are stored in a Dictionary<int, Reservation> type object. These methods and properties are:

* AddReservation – method is used for creating new reservations.
* Reservations – property for reading reservation.
* UpdateReservation – method is used for updating reservations.
* DeleteReservation – method to delete reservations.

CRUD stand for CREATE, READ, UPDATE and DELETE of objects. Normally in a class each of these operations are done by a specific function.

Configuration in the Program class

We use [AddSingleton](https://www.yogihosting.com/aspnet-core-dependency-injection/" \l "addsingleton" \t "_blank) method to set up the service mapping for the reservation repository. We do this by adding the below given code line to the Program.cs class.

builder.Services.AddSingleton<IRepository, Repository>();

Controller for Web API

Now comes the most important part of creating a **Controller for the Web API**. Remember that this Controller is just a normal Controller, that allows data in the model to be retrieved or modified, and then deliver it to the client. It does this without having to use the actions provided by the regular controllers.

The *data delivery* is done by following a pattern known by name as **REST**. REST Stands for **REpresentational State Transfer** pattern, which contains 2 things:

* 1. Action Methods which do specific operations and then deliver some data to the client. These methods are decorated with attributes that makes them to be invoked only by **HTTP requests**.
* 2. URLs which defines operational tasks. These operations can be – sending full or part of a data, adding, deleting or updating records. In fact it can be anything.

First add the package called Microsoft.AspNetCore.JsonPatch from NuGet. This is needed to support **JSON Patch**. I have shown this package in the below image.

Next to the Controllers folder of the project, add a new Controller called ReservationController.cs. Add the following code to it.

|  |  |
| --- | --- |
|  | using APIControllers.Models;  using Microsoft.AspNetCore.JsonPatch;  using Microsoft.AspNetCore.Mvc;    namespace APIControllers.Controllers  {      [ApiController]      [Route("api/[controller]")]      public class ReservationController : ControllerBase      {          private IRepository repository;          public ReservationController(IRepository repo) => repository = repo;            [HttpGet]          public IEnumerable<Reservation> Get() => repository.Reservations;            [HttpGet("{id}")]          public ActionResult<Reservation> Get(int id)          {              if (id == 0)                  return BadRequest("Value must be passed in the request body.");              return Ok(repository[id]);          }            [HttpPost]          public Reservation Post([FromBody] Reservation res) =>          repository.AddReservation(new Reservation          {              Name = res.Name,              StartLocation = res.StartLocation,              EndLocation = res.EndLocation          });            [HttpPut]          public Reservation Put([FromForm] Reservation res) => repository.UpdateReservation(res);            [HttpPatch("{id}")]          public StatusCodeResult Patch(int id, [FromBody] JsonPatchDocument<Reservation> patch)          {              var res = (Reservation)((OkObjectResult)Get(id).Result).Value;              if (res != null)              {                  patch.ApplyTo(res);                  return Ok();              }              return NotFound();          }            [HttpDelete("{id}")]          public void Delete(int id) => repository.DeleteReservation(id);      }  } |

Explanation

Notice the Controller derives from ControllerBase class and has an attribute called [ApiController] applied to it. The Controller gets the Reservation class object in the constructor through the Dependency Injection feature.

Route of the API Controller

The route by which this controller can be reached is defined by Attribute Routes. You can check my tutorial called [Learn Attribute Routing in ASP.NET Core](https://www.yogihosting.com/aspnet-core-attribute-routing/) to know it in full details.

This **Web API Controller** is reached through the URL – https://localhost:44324/api/Reservation. Here “44324” is the port number.

|  |
| --- |
| [ApiController]  [Route("api/[controller]")]  public class ReservationController : ControllerBase  {  ...  } |

Now test this thing by running the application and then opening the URL – https://localhost:44324/api/Reservation on the browser. You will see the JSON of the 3 reservations as shown in the below image.

The URL calls the Get method of the Reservation controller. This method is shown below:

[HttpGet]

public IEnumerable<Reservation> Get() => repository.Reservations;

As you can see this method returns all the reservations so you get the **JSON of the reservations** on the browser.

Web API Action Methods

A Web API can have action methods of type GET/POST/PUT/PATCH/DELETE/HEAD. Based on the incoming request the Web API decides which action method to execute. Example:

* HTTP GET type request will be handled by GET Type action method.
* HTTP POST type request will be handled by POST Type action method.
* and so on.

By default ASP.NET CORE does the following things:

* 1. Sends the data to the client as string if the action method returns a string. It also sets the Content-Type header of the response as text/plain.
* 2. Sends the data to the client as JSON if the action method return type is anything but not string like int, datetime, object, simple type, complex type, etc. It also sets the Content-Type header of the response as application/json.

The Web API Controller action methods have been applied some **HTTP attributes**. So they are invoked only by the specific HTTP method known as VERBS.

Examples of VERBS are – **GET, POST, PUT, PATCH, DELETE and HEAD**.

In short these HTTP Attributes correspond to the VERBS. So a specific VERB can only invoke an action that has a corresponding HTTP Attribute.

HTTP Attributes are defined in the below table:

| **Name** | **Description** |
| --- | --- |
| HttpGet | It specifies that the action method can be invoked only by HTTP requests that use the GET verb. |
| HttpPost | It specifies that the action method can be invoked only by HTTP requests that use the POST verb. |
| HttpDelete | It specifies that the action method can be invoked only by HTTP requests that use the DELETE verb. |
| HttpPut | It specifies that the action method can be invoked only by HTTP requests that use the PUT verb. |
| HttpPatch | It specifies that the action method can be invoked only by HTTP requests that use the PATCH verb. |
| HttpHead | It specifies that the action method can be invoked only by HTTP requests that use the HEAD verb. |

Note – for action methods accepting multiple verbs use the C# attribute called **[AcceptVerbs]**.

Don’t forget to secure your REST Web APIs. I have covered this topic at [How to secure APIs with JWT in ASP.NET Core [with source codes]](https://www.yogihosting.com/jwt-api-aspnet-core/)

[HttpGet] Action Methods

There are two [HttpGet] methods that will be invoked on HTTP request of type GET. These methods work in this way.

* 1. The first action method delivers all the reservation records to the client in **JSON**. The default return type of a Controller’s action is JSON so when a class object is delivered then it is done by **JSON**. The HttpGet attribute does not contain a routing segment so the URL to invoke this action method is – https://localhost:44324/api/Reservation. This action method is shown below.

[HttpGet]

public IEnumerable<Reservation> Get() => repository.Reservations;

* 2. The second action method contains the id routing segment as the argument. It then delivers the reservation record for that particular id only, in JSON format. I have used Ok() method which returns the status code 200 along with the reservation record to the client. The return type of this method is ActionResult<Reservation>. The URLs to invoke this Action method are:

https://localhost:44324/api/Reservation/1

https://localhost:44324/api/Reservation/2

https://localhost:44324/api/Reservation/3

etc

This action method is shown below.

|  |  |
| --- | --- |
|  | [HttpGet("{id}")]  public ActionResult<Reservation> Get(int id)  {      if (id == 0)          return BadRequest("Value must be passed in the request body.");      return Ok(repository[id]);  } |

If the client does not sends an id in the request then in that case the id gets the default value of 0. So I am simply returning BadRequest() for the response.

The **HTTP GET** requests can be made directly from the browser. Run your application and go to the URL https://localhost:44324/api/Reservation, where you will see a JSON containing all the reservations, as shown in the image below:

Similarly if you go to the URL – https://localhost:44324/api/Reservation/1 in your browser, then you will get the **JSON** for the first Reservation, as shown by the image below:

[HttpPost] Action

The **HttpPost Action** method is used to create a new Reservation. It receives the Reservation object in it’s argument. The [FromBody] attribute applied to it’s argument ensures the body content send from the client will be decoded, and put to this argument,

The URL to invoke this Action method is – https://localhost:44324/api/Reservation. Note that although it’s URL is same as that of the GET Action, the presence of [HttpPost] attribute ensures that it is invoked only for HTTP request of type POST.

This action returns the newly added Reservation object in JSON format. The reservation object also contains the value of the created Id field. This method is given below.

|  |  |
| --- | --- |
|  | [HttpPost]  public Reservation Post([FromBody] Reservation res) =>  repository.AddReservation(new Reservation  {      Name = res.Name,      StartLocation = res.StartLocation,      EndLocation = res.EndLocation  }); |

[HttpPut] Action

The **HttpPut Action** is used for doing the update of a Reservation object. It will be invoked when Http request of type PUT is made to the URL – https://localhost:44324/api/Reservation. This method is given below.

[HttpPut]

public Reservation Put([FromForm] Reservation res) => repository.UpdateReservation(res);

The [FromForm] attribute on the argument ensure that the form data sent by the client will be used to bind this Reservation object using Model Binding.

This action method returns the **Updated Reservation object in JSON** format.

You can also use [FromBody] attribute instead of [FromForm] attribute on the argument. The only difference is in sending the data from the client side. Data has to be send in JSON format if you use FromBody attribute, else for FromForm attribute it has to be send in Form data.

[HttpDelete] Action

The HttpDelete action **deletes a reservation** from the repository. This method is called when Http request of type DELETE is initiated on the URLs given below –

https://localhost:44324/api/Reseration/1

https://localhost:44324/api/Reseration/2

https://localhost:44324/api/Reseration/3

etc

Note: The id of the reservation to be deleted is passed as the 3rd segment of the URL. This method is shown below.

|  |  |
| --- | --- |
| 1  2 | [HttpDelete("{id}")]  public void Delete(int id) => repository.DeleteReservation(id); |

[HttpPatch] Action

The **HttpPatch** Action can do multiple operations, like Adding, Removing, Updating, Copying, etc, of a Reservation object which is sent by the client. Here the client only sends a specific set of Reservation properties instead of the whole reservation object to the API in **JSON format**.

This JSON format looks like:

|  |  |
| --- | --- |
| 1  2  3  4 | [      { "op": "replace", "path": "Name", "value": "Ram"},      { "op": "replace", "path": "StartLocation", "value": "Moscow"}  ] |

The JSON has op property which specifies the type of the operation, and a path property which specifies where the operation will be applied. The value property specifies it’s new value.

ASP.NET Core will automatically process the JSON data and sends it to the action method as a JsonPatchDocument<T> object, where T is the type of the model object to be modified (here it is the Reservation object).

The JsonPatchDocument object is then used to modify an object from the repository using the ApplyTo() method. See it’s code below.

|  |  |
| --- | --- |
|  | [HttpPatch("{id}")]  public StatusCodeResult Patch(int id, [FromBody]JsonPatchDocument<Reservation> patch)  {      var res = (Reservation)((OkObjectResult)Get(id).Result).Value;      if (res != null)      {          patch.ApplyTo(res);          return Ok();      }      return NotFound();  } |

The reservation is fetched by calling Get(id) method and casting the result to OkObjectResult. One more casting is perfomed afterwards to get the value in Reservation type object – var res = (Reservation)((OkObjectResult)Get(id).Result).Value.

Important for Patch

If you are creating the **Web API in ASP.NET Core version 3 and above** then you must know that AddNewtonsoftJson replaces the System.Text.Json-based input and output formatters used for formatting all JSON content. Therefore you have to enable JSON Patch support in your API project.

* 1. Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package.
* 2. Update the Program.cs to call AddNewtonsoftJson() method as shown in the below code.

builder.Services.AddControllersWithViews().AddNewtonsoftJson();

The table given below summarizes the working details for each of these action methods:

| **HTTP Request Type** | **URL** | **Data from Client** | **Returns** |
| --- | --- | --- | --- |
| GET | /api/Reservation | No data | Returns all the reservations in JSON |
| GET | /api/Reservation/1, /api/Reservation/2, etc | No data | Returns the reservation data of the id which is sent to its parameter in JSON |
| POST | /api/Reservation | The Reservation object in JSON. | Returns the newly created Reservation object in JSON |
| PUT | /api/Reservation | The Reservation object in JSON. | Returns the newly updated Reservation object in JSON |
| DELETE | /api/Reservation/1, /api/Reservation/2, etc | No data | None |
| PATCH | /api/Reservation/1, /api/Reservation/2, etc | A JSON that contains set of modifications to be applied. | Returns confirmation that the changes have been applied. |

The link to download the full source code of this tutorial is given below:

# est Web API

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

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

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

## Fiddler

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

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

**Step 1:**

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

**Step 2:**

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

Fiddler

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

Hide All Processes in Fiddler

**Step 3:**

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

Select HTTP Method

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

Enter URL and Execute

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

Response in Fiddler

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

Fiddler Request & Response

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

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

Fiddler Request & Response in Raw Format

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

## Postman

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

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

Test Web API in Postman

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

Response in Postman

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