**ASP.NET Core Web API Controllers**

**https://docs.microsoft.com/en-us/learn/modules/build-web-api-aspnet-core/4-aspnet-controllers**

In the previous exercise, you created a web application that provides sample weather forecast data, and then interacted with it in the HTTP REPL.

Before you dive in to writing your own PizzaController class, let's look at the code in the WeatherController sample to understand how it works. In this unit, you'll see how WeatherController uses the ControllerBase base class and a few .NET attributes to build a functional web API in a few dozen lines of code. After you understand those concepts, you'll be ready to write your own PizzaController class.

using Microsoft.AspNetCore.Mvc;

namespace ContosoPizza.Controllers;

[ApiController]

[Route("[controller]")]

public class WeatherForecastController : ControllerBase

{

private static readonly string[] Summaries = new[]

{

"Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"

};

private readonly ILogger<WeatherForecastController> \_logger;

public WeatherForecastController(ILogger<WeatherForecastController> logger)

{

\_logger = logger;

}

[HttpGet(Name = "GetWeatherForecast")]

public IEnumerable<WeatherForecast> Get()

{

return Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = Random.Shared.Next(-20, 55),

Summary = Summaries[Random.Shared.Next(Summaries.Length)]

})

.ToArray();

}

}

**The base class: ControllerBase**

A controller is a public class with one or more public methods known as *actions*. By convention, a controller is placed in the project root's *Controllers* directory. The actions are exposed as HTTP endpoints via routing. So an HTTP GET request to https://localhost:{PORT}/weatherforecast causes the Get() method of the WeatherForecastController class to be executed.

The first thing to notice is that this class inherits from the ControllerBase base class. This base class provides a lot of standard functionality for handling HTTP requests, so you can focus on the specific business logic for your application.

**Note**

If you have experience with Razor Pages or model-view-controller (MVC) architecture development in ASP.NET Core, you've used the Controller class. Don't create a web API controller by deriving from the Controller class. Controller derives from ControllerBase and adds support for views, so it's for handling webpages, not web API requests.

**API controller class attributes**

Two important attributes are applied to WeatherForecastController, as shown in the following code:

[ApiController]

[Route(“[controller]”)]

public class WeatherForecastController : ControllerBase

[ApiController] enables [opinionated behaviors](https://docs.microsoft.com/en-us/aspnet/core/web-api/" \l "apicontroller-attribute) that make it easier to build web APIs. Some behaviors include [parameter source inference](https://docs.microsoft.com/en-us/aspnet/core/web-api/" \l "binding-source-parameter-inference), [attribute routing as a requirement](https://docs.microsoft.com/en-us/aspnet/core/web-api/" \l "attribute-routing-requirement), and [model validation error-handling enhancements](https://docs.microsoft.com/en-us/aspnet/core/web-api/" \l "automatic-http-400-responses).

[Route] defines the routing pattern [controller]. The [controller] token is replaced by the controller’s name (case-insensitive, without the *Controller* suffix). This controller handles requests to <https://localhost>:{PORT}/weatherforecast.

**Note**

The route might contain static strings, as in api/[controller]. In this example, this controller would handle a request to <https://localhost>:{PORT}/api/weatherforecast.

**Providing weather results with the Get() method**

WeatherForecastController includes a single controller action, designated by the [HttpGet(Name = “GetWeatherForecast”)] attribute. This attribute routes HTTP GET requests to the public Ienumerable<WeatherForecast> Get() method. That’s why, in the previous exercise, you saw that requests to <https://localhost>:{PORT}/weatherforecast caused the weather results to be returned.

As you’ll learn later in this module, other common actions are associated with a web API that performs CRUD operations (GET, PUT, POST, DELETE). But an API controller needs to implement only one controller action.

In this case, you’re getting the full isto f WeatherForecast returned. The GET operation also allows for retrieving a single item by passing in an identifier. In ASP.NET, you can achieve this by using the [HttpGet(“{id}”)] attribute. You’ll implement that attribute in the next exercise.

Now that you’ve learned the fundamental components of a web API controller, you’re ready to create your own PizzaController class.

**Check your knowledge**Top of Form

What is the purpose of the [ApiController] attribute?

[ApiController] enables opinionated behaviors that make it easier to build web APIs.

**Create a pizza model**

Run the following command to create a *Models* folder:

mkdir Models

Select the *Models* folder and add a new file called *Pizza.cs*.

namespace ContosoPizza.Models;

public class Pizza

{

public int Id { get; set; }

public string? Name { get; set; }

public bool IsGlutenFree { get; set; }

}

**Tip**

* Ctrl+v Paste
* Ctrl+ s Save

**Add a data service**

Run the following command to create a *Services* folder:

* Create folder Services
* Create file PizzaServices.cs

using ContosoPizza.Models;

namespace ContosoPizza.Services;

public static class PizzaService

{

static List<Pizza> Pizzas { get; }

static int nextId = 3;

static PizzaService()

{

Pizzas = new List<Pizza>

{

new Pizza { Id = 1, Name = "Classic Italian", IsGlutenFree = false },

new Pizza { Id = 2, Name = "Veggie", IsGlutenFree = true }

};

}

public static List<Pizza> GetAll() => Pizzas;

public static Pizza? Get(int id) => Pizzas.FirstOrDefault(p => p.Id == id);

public static void Add(Pizza pizza)

{

pizza.Id = nextId++;

Pizzas.Add(pizza);

}

public static void Delete(int id)

{

var pizza = Get(id);

if(pizza is null)

return;

Pizzas.Remove(pizza);

}

public static void Update(Pizza pizza)

{

var index = Pizzas.FindIndex(p => p.Id == pizza.Id);

if(index == -1)

return;

Pizzas[index] = pizza;

}

}

This service provides a simple in-memory data caching service with two pizzas by default.

dotnet build

A *controller* is a public class with one or more public methods known as *actions*. By convention, a controller is placed in the project root's *Controllers* directory. The actions are exposed as HTTP endpoints inside the web API controller.

**Create a controller**

1. Create folder Controllers
2. Create file *PizzaController.cs*

An empty class file named *PizzaController.cs* is created in the *Controllers* directory. The directory name *Controllers* is a convention. The directory name comes from the model-view-*controller* architecture that the web API uses.

**Note**

By convention, controller class names are suffixed with *Controller*.

Add the following code to *Controllers/PizzaController.cs*. Save your changes.

using ContosoPizza.Models;

using ContosoPizza.Services;

using Microsoft.AspNetCore.Mvc;

namespace ContosoPizza.Controllers;

[ApiController]

[Route("[controller]")]

public class PizzaController : ControllerBase

{

public PizzaController()

{

}

// GET all action

// GET by Id action

// POST action

// PUT action

// DELETE action

}

As you learned previously, this class derives from ControllerBase, the base class for working with HTTP requests in ASP.NET Core. It also includes the two standard attributes you've learned about, [ApiController] and [Route]. As before, the [Route] attribute defines a mapping to the [controller] token. Because this controller class is named PizzaController, this controller handles requests to

[https://localhost:{PORT}/pizza](https://localhost:%7bPORT%7d/pizza)

|  |  |
| --- | --- |
| Get all pizzas  [https://localhost:{PORT}/pizza](https://localhost:%7bPORT%7d/pizza) | [HttpGet]  public ActionResult<List<Pizza>> GetAll() =>  PizzaService.GetAll(); |
| Get one pizza  [https://localhost:{PORT}/pizza](https://localhost:%7bPORT%7d/pizza)/1 | [HttpGet("{id}")]  public ActionResult<Pizza> Get(int id)  {  var pizza = PizzaService.Get(id);  if(pizza == null)  return NotFound();  return pizza;  } |

Each ActionResult instance used in the preceding action is mapped to the corresponding HTTP status code in the following table:

| **TABLE 1** | | |
| --- | --- | --- |
| **ASP.NET Core action result** | **HTTP status code** | **Description** |
| Ok is implied | 200 | A product that matches the provided id parameter exists in the in-memory cache. The product is included in the response body in the media type, as defined in the accept HTTP request header (JSON by default). |
| NotFound | 404 | A product that matches the provided id parameter doesn't exist in the in-memory cache. |

Build and start the web API by running the following command:

dotnet run

Open the existing httprepl terminal, or open a new integrated terminal from Visual Studio Code by selecting **Terminal** > **New Terminal** from the main menu.

Connect to our web API by running the following command:

httprepl https://localhost:{PORT}

Alternatively, run the following command at any time while HttpRepl is running:

(Disconnected)> connect https://localhost:{PORT}

To see the newly available Pizza endpoint, run the following command:

ls

The preceding command detects all APIs available on the connected endpoint. It should display the following code:

https://localhost:{PORT}/> ls

. []

Pizza [GET]

WeatherForecast [GET]

1. Go to the Pizza endpoint by running the following command:

.NET CLICopy

cd Pizza

The preceding command shows an output of available APIs for the Pizza endpoint:

https://localhost:{PORT}/> cd Pizza

/Pizza [GET]

Make a GET request in HttpRepl by using the following command:

get

The preceding command returns a list of all pizzas in json:

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

Date: Fri, 02 Apr 2021 21:55:53 GMT

Server: Kestrel

Transfer-Encoding: chunked

[

{

"id": 1,

"name": "Classic Italian",

"isGlutenFree": false

},

{

"id": 2,

"name": "Veggie",

"isGlutenFree": true

}

]

To query for a single pizza, you can make another GET request, but pass in an id parameter by using the following command:

get 1

The preceding command returns Classic Italian with the following output:

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

Date: Fri, 02 Apr 2021 21:57:57 GMT

Server: Kestrel

Transfer-Encoding: chunked

{

"id": 1,

"name": "Classic Italian",

"isGlutenFree": false

}

Our API also handles situations where the item doesn't exist. Call the API again, but pass in an invalid pizza id parameter by using the following command:

get 5

The preceding command returns a 404 Not Found error with the following output:

HTTP/1.1 404 Not Found

Content-Type: application/problem+json; charset=utf-8

Date: Fri, 02 Apr 2021 22:03:06 GMT

Server: Kestrel

Transfer-Encoding: chunked

{

"type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",

"title": "Not Found",

"status": 404,

"traceId": "00-ec263e401ec554b6a2f3e216a1d1fac5-4b40b8023d56762c-00"

}

<https://docs.microsoft.com/en-us/learn/modules/build-web-api-aspnet-core/7-crud>

<https://docs.microsoft.com/en-us/learn/modules/build-web-api-aspnet-core/4-aspnet-controllers>

| **HTTP action verb** | **CRUD operation** | **ASP.NET Core attribute** |
| --- | --- | --- |
| GET | Read | [HttpGet] |
| POST | Create | [HttpPost] |
| PUT | Update | [HttpPut] |
| DELETE | Delete | [HttpDelete] |

|  |  |
| --- | --- |
| Add | [HttpPost]  public IActionResult Create(Pizza pizza)  {  PizzaService.Add(pizza);  return CreatedAtAction(nameof(Create), new { id = pizza.Id }, pizza);  } |
| Update | [HttpPut("{id}")]  public IActionResult Update(int id, Pizza pizza)  {  if (id != pizza.Id)  return BadRequest();    var existingPizza = PizzaService.Get(id);  if(existingPizza is null)  return NotFound();    PizzaService.Update(pizza);    return NoContent();  } |
| Delete | [HttpDelete("{id}")]  public IActionResult Delete(int id)  {  var pizza = PizzaService.Get(id);    if (pizza is null)  return NotFound();    PizzaService.Delete(id);    return NoContent();  } |

dotnet run

httprepl [https://localhost:{PORT}](https://localhost:%7bPORT%7d)

Alternatively, run the following command at any time while HttpRepl is running:

(Disconnected)> connect [https://localhost:{PORT}](https://localhost:%7bPORT%7d)

Go to the Pizza endpoint by running the following command:

cd Pizza

Run the following command to see the new actions on the Pizza API:

ls

The preceding command shows an output of available APIs for the Pizza endpoint:

.NET CLICopy

https://localhost:{PORT}/Pizza> ls

. [GET|POST]

.. []

{id} [GET|PUT|DELETE]

Make a POST request to add a new pizza in HttpRepl by using the following command:

post -c "{"name":"Hawaii", "isGlutenFree":false}"

The preceding command returns a list of all pizza:

HTTP/1.1 201 Created

Content-Type: application/json; charset=utf-8

Date: Fri, 02 Apr 2021 23:23:09 GMT

Location: https://localhost:{PORT}/Pizza?id=3

Server: Kestrel

Transfer-Encoding: chunked

{

"id": 3,

"name": "Hawaii",

"isGlutenFree": false

}

Update the new Hawaii pizza to a Hawaiian pizza with a PUT request by using the following command:

put 3 -c "{"id": 3, "name":"Hawaiian", "isGlutenFree":false}"

The preceding command returns the following output that indicates success:

.NET CLICopy

HTTP/1.1 204 No Content

Date: Fri, 02 Apr 2021 23:23:55 GMT

Server: Kestrel

To verify that the pizza was updated, rerun the GET action by using the following command:

get 3

The preceding command returns the newly updated pizza:

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

Date: Fri, 02 Apr 2021 23:27:37 GMT

Server: Kestrel

Transfer-Encoding: chunked

{

"id": 3,

"name": "Hawaiian",

"isGlutenFree": false

}

Our API can also delete the newly created pizza through the DELETE action if you run the following command:

delete 3

The preceding command returns a 204 No Content result for success:

.NET CLICopy

HTTP/1.1 204 No Content

Date: Fri, 02 Apr 2021 23:30:04 GMT

Server: Kestrel

To verify that the pizza was removed, rerun the GET action by using the following command:

get

The preceding command returns the original pizzas as results:

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

Date: Fri, 02 Apr 2021 23:31:15 GMT

Server: Kestrel

Transfer-Encoding: chunked

[

{

"id": 1,

"name": "Classic Italian",

"isGlutenFree": false

},

{

"id": 2,

"name": "Veggie",

"isGlutenFree": true

}

]

**Videos for learning more**

* [.NET 101](https://docs.microsoft.com/en-us/shows/NET-Core-101/?WT.mc_id=Educationaldotnet-c9-scottha)
* [ASP.NET Core Web API 101](https://docs.microsoft.com/en-us/shows/Beginners-Series-to-Web-APIs/)
* [ASP.NET Core 101](https://docs.microsoft.com/en-us/shows/ASPNET-Core-101/?WT.mc_id=Educationaspnet-c9-niner)

**Articles for learning more**

* [Tutorial: Create a web API with ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api)
* [Create web APIs with ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/web-api/)
* [Controller action return types in ASP.NET Core web API](https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types)