Unit Testing in ASP.NET Core Web API

Posted by **Code Maze** | Updated Date May 27, 2022 | **19**

Software maintenance is an inevitable part of the development process and one that could give developers the most trouble. We've all been there, whether we left our code in someone else's care, or we've inherited some legacy code.

This doesn't necessarily need to be a bad thing and there are ways to improve our code and make it more maintainable. **Unit testing** plays a very important role in making the software more maintainable.

Our intention in this post is to make an **intro to unit testing of the ASP.NET Core Web API application**.

> You can download the source code from **the repo on GitHub.**

Let's start

About Unit Testing in General

What is unit testing in the first place? It begins by defining what a „unit" is and although this is not strictly defined, the unit represents a unit of work – usually a single method in our code.

We test these units individually, making sure that each of them is doing exactly what it is written for.

Nothing more, nothing less.

What is important to understand is that we are not testing the behavior of the dependencies of that method. That is what the integration tests are for.

> We have a great series of articles dedicated to Testing ASP.NET Core Application. So, if you want to learn even more about the testing, we strongly recommend reading **ASP.NET Core MVC Testing Series**.

Preparing the Example Project

We will use Visual Studio to create our example project and it will be an **ASP.NET Core Web API** application.

So, let's start by creating a new ASP.NET Core Web Application.

When we create the **ASP.NET Core API** application initially, it comes with the default controller class.

Let's delete that one and create our own example controller named `ShoppingCartController`. Here we can define CRUD operations you would typically have on an entity based controller:

```
[Route("api/[controller]")]
[ApiController]
public class ShoppingCartController : ControllerBase
{
    private readonly IShoppingCartService _service;
    public ShoppingCartController(IShoppingCartService service)
    {
        _service = service;
    }

    [HttpGet]
    public IActionResult Get()
    {
        var items = _service.GetAllItems();
        return Ok(items);
    }

    [HttpGet("{id}")]
    public IActionResult Get(Guid id)
    {
        var item = _service.GetById(id);
        if (item == null)
        {
            return NotFound();
        }
        return Ok(item);
    }

    [HttpPost]
    public IActionResult Post([FromBody] ShoppingItem value)
    {
        if (!ModelState.IsValid)
        {
```

```
            return BadRequest(ModelState);
        }
        var item = _service.Add(value);
        return CreatedAtAction("Get", new { id = item.Id }, item);
    }

    [HttpDelete("{id}")]
    public IActionResult Remove(Guid id)
    {
        var existingItem = _service.GetById(id);
        if (existingItem == null)
        {
            return NotFound();
        }
        _service.Remove(id);
        return NoContent();
    }
}
```

Nothing special about the code here, we've got a simple example of a shopping cart controller where we have methods to get, add and remove items from the cart.

ShoppingCartService Explanation

To access the data source, we are using `ShoppingService` class that implements `IShoppingService` interface. This allows us to follow the dependency injection principle, which is used heavily for the purpose of unit testing.

Using dependency injection, we can inject whatever implementation of `IShoppingCart` interface we want into our test class.

> Wanna join Code Maze Team, help us produce more awesome .NET/C# content and **get paid? >> JOIN US! <<**

Please note that methods of the service are not implemented in the example project, because we are not focusing on the service implementation here, testing the controller is the main goal. In the real project, you would probably use some data access logic in your service methods:

```
public class ShoppingCartService : IShoppingCartService
{
    public ShoppingItem Add(ShoppingItem newItem) => throw new Not

    public IEnumerable<ShoppingItem> GetAllItems() => throw new No

    public ShoppingItem GetById(Guid id) => throw new NotImplement

    public void Remove(Guid id) => throw new NotImplementedExcepti
}
```

`IShoppingService` contains signatures of all the methods seen in the `ShoppingCartService`:

```
public interface IShoppingCartService
{
    IEnumerable<ShoppingItem> GetAllItems();
    ShoppingItem Add(ShoppingItem newItem);
    ShoppingItem GetById(Guid id);
    void Remove(Guid id);
}
```

`ShoppingItem` is our main (and only 🙄) entity with just a few fields:

```
public class ShoppingItem
{
    public Guid Id { get; set; }
    [Required]
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Manufacturer { get; set; }
}
```

As we are using dependency injection to create instances of our services, make sure not to forget to register the service in the `Startup` class:

```
services.AddScoped<IShoppingCartService, ShoppingCartService>();
```

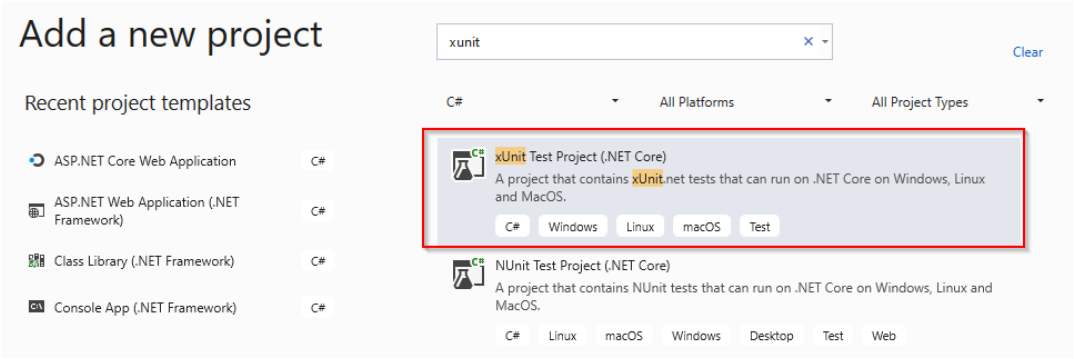
Of course, if you are using .NET 6, you have to use the `Program` class:

```
builder.Services.AddScoped<IShoppingCartService,
ShoppingCartService>();
```

Creating a Testing Project

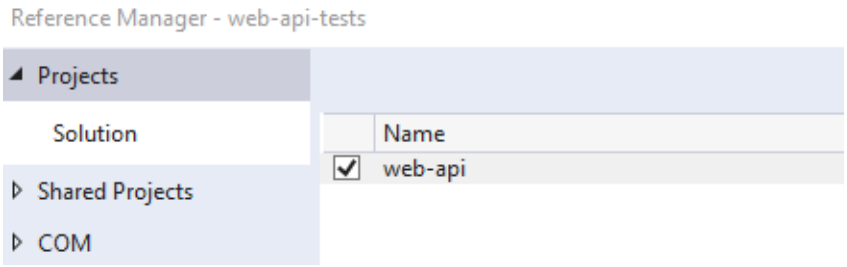
Finally, we come to the point when we need to create a new project where our tests are going to be. Conveniently for us, there is a `xUnit` testing project template out-of-the-box when using visual studio 2019, so we are going to make use of that.

The `xUnit` is an open-source unit testing tool for the .NET framework that simplifies the testing process and allows us to spend more time focusing on writing our tests:



We are going to name it `web-api-tests`.

Now we have a new project in our solution named `web-api-tests`. The next thing we should do is to add the reference to the project we are about to write tests for:



At this time we should create our fake implementation of the `IShoppingCartService` interface, which we are going to inject into our controller at the time of testing.

It has an in-memory collection which we are going to fill up with our dummy data:

```
public class ShoppingCartServiceFake: IShoppingCartService
{
```

```
        private readonly List<ShoppingItem> _shoppingCart;

        public ShoppingCartServiceFake()
        {
            _shoppingCart = new List<ShoppingItem>()
                {
                    new ShoppingItem() { Id = new Guid("ab2bd817-98cd-
                        Name = "Orange Juice", Manufacturer="Orange Tr
                    new ShoppingItem() { Id = new Guid("815accac-fd5b-
                        Name = "Diary Milk", Manufacturer="Cow", Price
                    new ShoppingItem() { Id = new Guid("33704c4a-5b87-
                        Name = "Frozen Pizza", Manufacturer="Uncle Mic
                };
        }

        public IEnumerable<ShoppingItem> GetAllItems()
        {
            return _shoppingCart;
        }

        public ShoppingItem Add(ShoppingItem newItem)
        {
            newItem.Id = Guid.NewGuid();
            _shoppingCart.Add(newItem);
            return newItem;
        }

        public ShoppingItem GetById(Guid id)
        {
            return _shoppingCart.Where(a => a.Id == id)
                .FirstOrDefault();
        }

        public void Remove(Guid id)
        {
            var existing = _shoppingCart.First(a => a.Id == id);
            _shoppingCart.Remove(existing);
        }
```

```
}
```

> Instead of creating a fake service manually, we could've used one of the many mocking frameworks available. One of those frameworks is called Moq. You can get more information about it in the **Testing MVC Controllers** article, where we use the Moq library to isolate our dependencies.

Let's write some unit tests!

Now we are all set and ready to write tests for our first unit of work – the `Get` method in our `ShoppingCartController`.

We will decorate test methods with the `[Fact]` attribute, which is used by the `xUnit` framework, marking them as the actual testing methods. Besides the test methods, we can have any number of helper methods in the test class as well.

When writing unit tests it is usually the practice to follow the AAA principle (Arrange, Act, and Assert):

Arrange – this is where you would typically prepare everything for the test, in other words, prepare the scene for testing (creating the objects and setting them up as necessary)

Act – this is where the method we are testing is executed

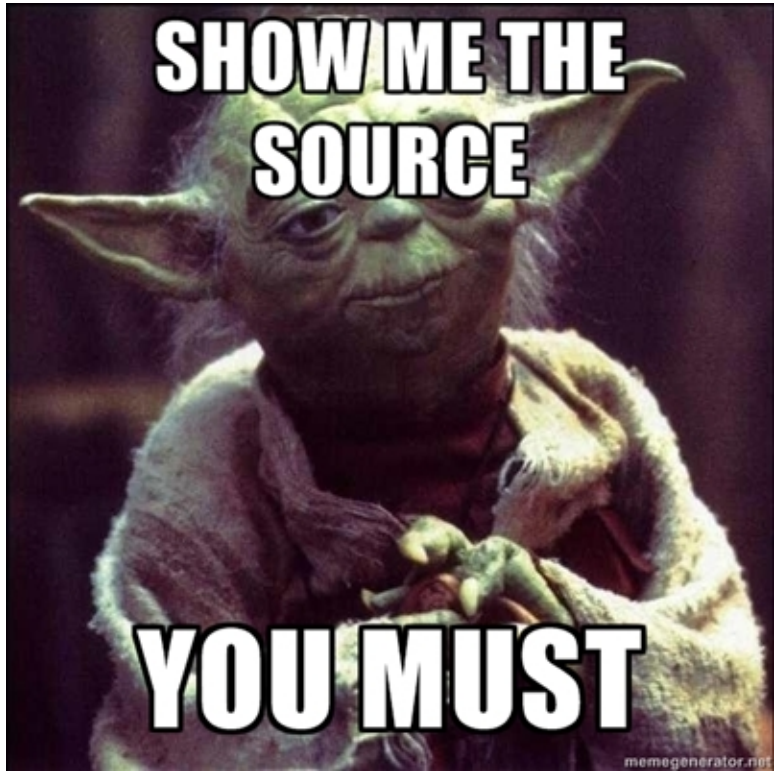
Assert – this is the final part of the test where we compare what we expect to happen with the actual result of the test method execution

Test method names should be as descriptive as possible. In most cases, it is possible to name the method so that it is not even necessary to read the actual code to understand what is being tested.

In the example we use the naming convention in which the first part represents the name of the method being tested, the second part tells us more about the testing scenario and the last part is the expected result.

Generally, the logic inside our controllers should be minimal and not so focused on business logic or infrastructure (eg. data access). We want to test the controller logic and not the frameworks we are using.

We need to test how the controller behaves based on the validity of the inputs and controller responses based on the result of the operation it performs.



Testing Our Actions

The first method we are testing is the `Get` method and there we will want to verify the following:

- Whether the method returns the `OkObjectResult` which represents 200 HTTP code response
- Whether returned object contains our list of `ShoppingItems` and all of our items

Testing the Get Method

So let's see how we go about testing our method:

```
public class ShoppingCartControllerTest
{
    private readonly ShoppingCartController _controller;
    private readonly IShoppingCartService _service;

    public ShoppingCartControllerTest()
    {
        _service = new ShoppingCartServiceFake();
        _controller = new ShoppingCartController(_service);
    }

    [Fact]
    public void Get_WhenCalled_ReturnsOkResult()
    {
        // Act
        var okResult = _controller.Get();

        // Assert
        Assert.IsType<OkObjectResult>(okResult as OkObjectResult);
    }

    [Fact]
```

```
    public void Get_WhenCalled_ReturnsAllItems()
    {
        // Act
        var okResult = _controller.Get() as OkObjectResult;

        // Assert
        var items = Assert.IsType<List<ShoppingItem>>(okResult.Val
        Assert.Equal(3, items.Count);
    }
}
```

We create an instance of the `ShoppingCartController` object in the test class and that is the class we want to test. It is important to note here that this constructor is called before each test method, meaning that we are always resetting the controller state and performing the test on the fresh object.

This is important because the test methods should not be dependent on one another and we should get the same testing results, no matter how many times we run the tests and in which order we run them.

Testing the GetById method

Now let's see how we can test the `GetById` method:

```
[Fact]
public void GetById_UnknownGuidPassed_ReturnsNotFoundResult()
{
    // Act
    var notFoundResult = _controller.Get(Guid.NewGuid());

    // Assert
    Assert.IsType<NotFoundResult>(notFoundResult);
}
```

```
    [Fact]
    public void GetById_ExistingGuidPassed_ReturnsOkResult()
    {
        // Arrange
        var testGuid = new Guid("ab2bd817-98cd-4cf3-a80a-53ea0cd9c200"

        // Act
        var okResult = _controller.Get(testGuid);

        // Assert
        Assert.IsType<OkObjectResult>(okResult as OkObjectResult);
    }

    [Fact]
    public void GetById_ExistingGuidPassed_ReturnsRightItem()
    {
        // Arrange
        var testGuid = new Guid("ab2bd817-98cd-4cf3-a80a-53ea0cd9c200"

        // Act
        var okResult = _controller.Get(testGuid) as OkObjectResult;

        // Assert
        Assert.IsType<ShoppingItem>(okResult.Value);
        Assert.Equal(testGuid, (okResult.Value as ShoppingItem).Id);
    }
```

Firstly we verify that the controller will return a 404 status code (Not Found) if someone asks for the non-existing `ShoppingItem`. Secondly, we test if 200 code is returned when the existing object is asked for and lastly we check if the right object is returned.

Testing the Add Method

Let's see how we can deal with the Add method:

```
[Fact]
public void Add_InvalidObjectPassed_ReturnsBadRequest()
{
    // Arrange
    var nameMissingItem = new ShoppingItem()
    {
        Manufacturer = "Guinness",
        Price = 12.00M
    };
    _controller.ModelState.AddModelError("Name", "Required");

    // Act
    var badResponse = _controller.Post(nameMissingItem);
```

```
        // Assert
        Assert.IsType<BadRequestObjectResult>(badResponse);
    }

    [Fact]
    public void Add_ValidObjectPassed_ReturnsCreatedResponse()
    {
        // Arrange
        ShoppingItem testItem = new ShoppingItem()
        {
            Name = "Guinness Original 6 Pack",
            Manufacturer = "Guinness",
            Price = 12.00M
        };

        // Act
        var createdResponse = _controller.Post(testItem);

        // Assert
        Assert.IsType<CreatedAtActionResult>(createdResponse);
    }

    [Fact]
    public void Add_ValidObjectPassed_ReturnedResponseHasCreatedItem()
    {
        // Arrange
        var testItem = new ShoppingItem()
        {
            Name = "Guinness Original 6 Pack",
            Manufacturer = "Guinness",
            Price = 12.00M
        };

        // Act
        var createdResponse = _controller.Post(testItem) as CreatedAtA
        var item = createdResponse.Value as ShoppingItem;

        // Assert
```

```
        Assert.IsType<ShoppingItem>(item);
        Assert.Equal("Guinness Original 6 Pack", item.Name);
    }
```

Once again we are testing that the right objects are returned when someone calls the method, but there is something important to note here.

Among other things, we are testing if `ModelState` is validated and the proper response is returned in the case that the model is not valid. But to achieve this, it is not enough to just pass the invalid object to the Add method. That wouldn't work anyway since model state validation is only triggered during runtime. It is up to integration tests to check if the model binding works properly.

What we are going to do here instead is add the `ModelError` object explicitly to the `ModelState` and then assert on the response of the called method.

Remove method

Testing the remove method is pretty straightforward:

```
[Fact]
public void Remove_NotExistingGuidPassed_ReturnsNotFoundResponse()
{
    // Arrange
    var notExistingGuid = Guid.NewGuid();

    // Act
    var badResponse = _controller.Remove(notExistingGuid);

    // Assert
    Assert.IsType<NotFoundResult>(badResponse);
}
```

```
[Fact]
public void Remove_ExistingGuidPassed_ReturnsNoContentResult()
{
    // Arrange
    var existingGuid = new Guid("ab2bd817-98cd-4cf3-a80a-53ea0cd9c

    // Act
    var noContentResponse = _controller.Remove(existingGuid);

    // Assert
    Assert.IsType<NoContentResult>(noContentResponse);
}

[Fact]
public void Remove_ExistingGuidPassed_RemovesOneItem()
{
    // Arrange
    var existingGuid = new Guid("ab2bd817-98cd-4cf3-a80a-53ea0cd9c

    // Act
    var okResponse = _controller.Remove(existingGuid);

    // Assert
    Assert.Equal(2, _service.GetAllItems().Count());
}
```

Remove method tests take care that a valid response is returned and that object is indeed removed from the list.

Summary

This concludes the tests scenarios for our `ShoppingCartController` and we just want to summarize the general advice about unit testing. There are a few guidelines or best practices you should strive for when writing unit tests.

Respecting these practices will certainly make your (and the life of your fellow developer) easier.

Unit tests should be readable

No one wants to spend time trying to figure out what is that your test does. Ideally, this should be clear just by looking at the test name.

Unit tests should be maintainable

We should try to write our tests in a way that minor changes to the code shouldn't make us change all of our tests. The DRY (don't repeat yourself) principle applies here, and we should treat our test code the same as the production code. This lowers the possibility that one day someone gets to the point where he/she needs to comment out all of our tests because it has become too difficult to maintain them.

Unit sets should be fast

If tests are taking too long to execute, it is probable that people will run them less often. That is certainly a bad thing and no one wishes to wait too long for tests to execute.

Unit tests should not have any dependencies

It is important that anyone who is working on the project can execute tests without the need to provide access to some external system or database. Tests need to run in full isolation.

Make tests trustworthy rather than just aiming for the code coverage

Good tests should provide us with the confidence that we will be able to detect errors before they reach production. It is easy to write tests that don't assert the right things just to make them pass and to increase code coverage. But there is no point in doing that. We should try to test the right things to be able to rely on them when the time comes to make some changes to the code.

Conclusion

In this post, we've learned what unit testing is and how to set up the unit testing project with xUnit.

We've also learned the basic scenarios of testing the controller logic on some CRUD operations.

These examples should give you a great starting point for writing your own unit tests, and testing the more complex projects.

Thanks for reading and hopefully this article will help you grasp the unit concepts and unit testing in ASP.NET Core a little bit better.

Want to build **great APIs?** Or become **even better** at it? Check our **Ultimate ASP.NET Core Web API program** and learn how to create a full production-ready ASP.NET Core API using only the latest .NET technologies. Bonus materials (Security book, Docker book, and other bonus files) are included in the Premium package!

SHARE:

Subscribe Login

Join the discussion

19 COMMENTS Newest

TAFF

2 years ago

Highly informative post and explained in a precise manner that anyone can easily understand. Thanks for sharing.

Privacy

0 Reply

Marinko Spasojevic

Reply to TAFF 2 years ago

Thank you too, Taff. I'm glad you like the article.

+ 0 – Reply

Alessandro Dos Santos

3 years ago

Isn't it much more like integration testing rather than unit testing?
I'm structuring a new project and trying to apply testing on it so I'd like to understand better those concepts.

+ 0 – Reply

Marinko

Reply to Alessandro Dos Santos 3 years ago

We are testing here only our actions in the controller, so they are isolated parts withuot any dependency inside our tests, so I would say they are all unit tests. But my opinion is, as long as you understand how tests work, naming is not that important. Eventually you are going to write all kinds of tests.

+ 0 – Reply

Alessandro Dos Santos

Reply to Marinko 3 years ago

Thanks for replying, it's quite clear for me now, I was focusing on service consuming instead of methods interactions when trying to contextualize

integration testing, makes sense now thinking this way.
Agreed, naming is not that important,

+ 0 − Reply

Boomer

3 years ago

I do not understand: if you “add the Model Error object explicitly to the ModelState” and then test the model state to see if the error is there.. of CORSE it’s going to be there. You just added it! What is the value of this “test”?

+ 0 − Reply

Marinko

Reply to Boomer 3 years ago

Hello Boomer. Let me try explaining this to you. As you said, you didn’t understand it. In order to test invalid Post action, you have to add a model error to the model. As you can see, we are calling the Post method form the controller and pass an invalid model. So we test whether our POST action returns a bad result. If it does, than we’ve created our POST method in a right manner. So, basically, we have to provide an invalid model and send it to the POST action in order to test how our POST action handles invalid models. We have an entire series dedicated to testing: https://test2.code-maze.com/asp-net-core-mvc-testing/ You can find a lot more information there for sure. All the best.

+ 0 − Reply

Jahongir Murtazayev

4 years ago

Privacy

Thank you Milos Davidovich for this post. I realized that it is as not difficult as I imagine

0 Reply

Altiano Gerung

4 years ago

Great job. When should we expect the next part, kinda like intermediate test article?

0 Reply

Bernard Bos

4 years ago

Thanks a million for this post. Note: instead of Assert.IsType I had to use Assert.That(response, Is.InstanceOf()); That took me a minute but I figured it out.

0 Reply

Mathavan

4 years ago

Thank you very much for this post. How to perform Unit Test for Authorized API methods using moq.

0 Reply

Milos Davidovic

Reply to Mathavan

4 years ago

Hi Mathavan and thank you for reading the post.
You would perform unit test on the authorized Web API methods in the same way as on the non-authorized, by testing only the code of your method under test. This is because authorization happens as a part of a request lifecycle and to test that you would have to do an actual http request against you method (endpoint). But if you'd do that it would not be unit test anymore, but the integration test (not covered in this article). If that is indeed what you need, I suggest you search for integration testing articles to help you get started.

+ 0 − Reply

Oxygen

4 years ago

Thank you very much. I was able to learn a lot about xUnit by watching your course very easily.

+ 0 − Reply

Milos Davidovic

Reply to Oxygen 4 years ago

No problem, I'm very glad you liked the article 🙂

+ 0 − Reply

Preston

5 years ago

IsType and Equal do not exist and are not in the documentation for Assert: https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=mstest-net-1.2.0.

I had to use: Assert.IsInstanceOfType(object, type); and change Assert.Equal to Assert.Equals. Where did you get IsType from? Is that from an external library or an extension method? Also, you never created the class: ShoppingCartServiceFake. If you search the entire page there is only 1 result. What exactly is ShoppingCartServiceFake?

I have learned a lot. Thanks for making this!

0 Reply

Milos Davidovic

Reply to Preston 5 years ago

Hi Preston, I am happy liked the post.
If you take a look at the section 'Creating a Testing Project' , you will see that we are using xUnit testing tools, and library provided has all the methods you were missing.
In the same section there is a typo. Class named ShoppingCartService should be named ShoppingCartServiceFake, and you can see this if you take a look at the code from a git repo:
https://github.com/CodeMazeBlog/unit-testing-aspnetcore-webapi
Thanks for you comments, we will fix that in the post soon.

0 Reply

Preston

Reply to Milos Davidovic 5 years ago

My apologies, you're right I did start an MSTest Unit Test project instead. Though the functionality was basically the same, I for whatever reason didn't connect the dots.

0 Reply

barabasishe

5 years ago

Milos, thanks for the article.

Unfortunately, the base set of HTTP methods covers Update (PUT) method as well, but it isn't considered in the article.

0 Reply

Milos Davidovic

Reply to barabasishe 5 years ago

Thank you barabasishe, I am glad you liked the article.

You are right as the PUT request was not covered in the post, and the logic would be very similar as for the testing of the POST request.
The difference is that PUT request returns 204 HTTP response for the happy path, so I would assert that appropriate NoContentResult object was returned if the request was valid. If you are building a public API, might be a good idea to also verify that BadRequest response was returned when trying to update the non-existing item.

Hope this helps,
Milos

0 Reply