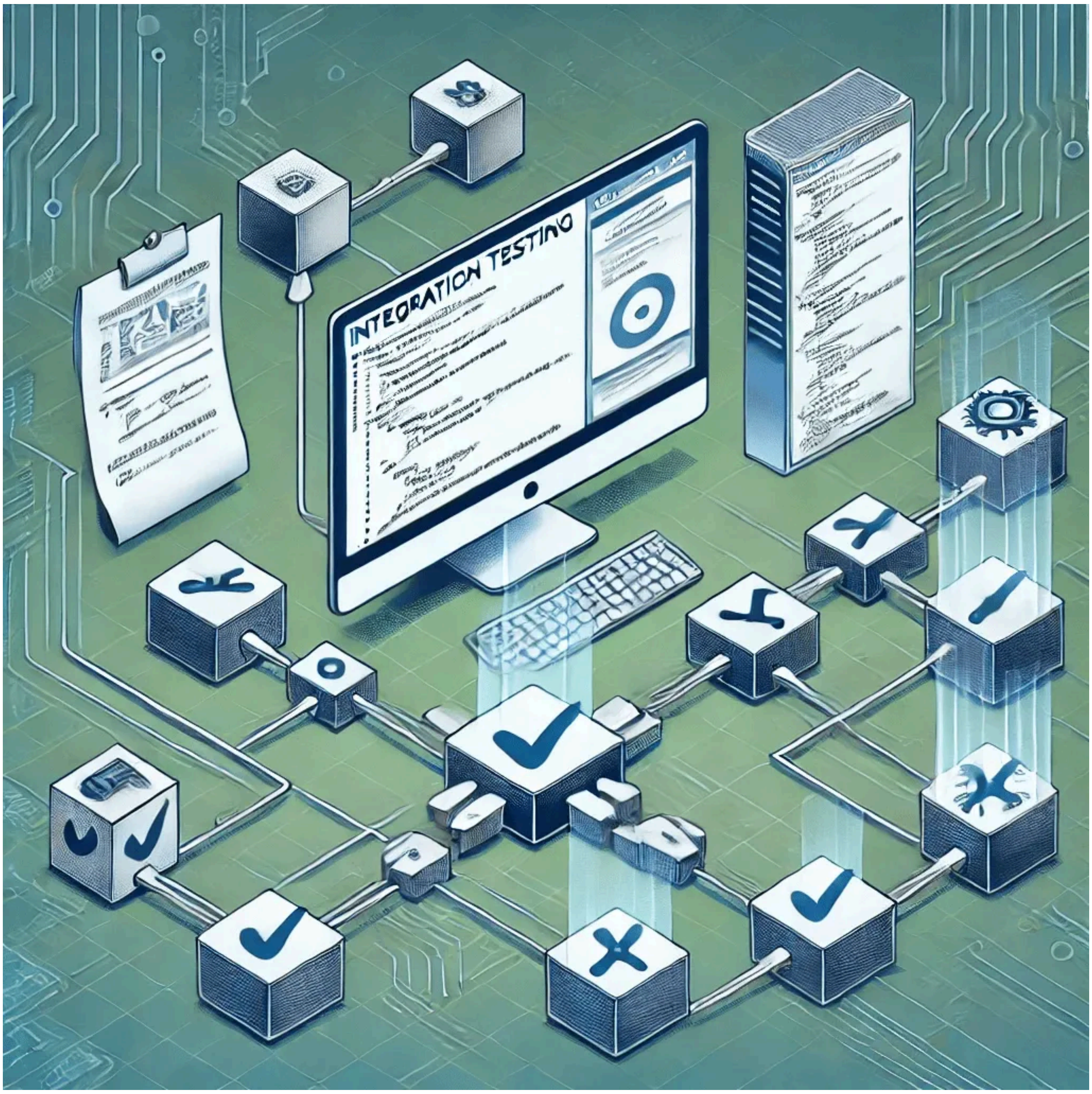
Integration testing for dotnet core APIs: Introduction

Ajay kumar · Follow

6 min read · Sep 29, 2024

Listen Share



This is going to be multi-part series on integration tests from introduction to advanced used cases.

What are integration tests in context of APIs?

In the context of **.NET Core APIs** for a blog, **integration tests** are automated tests that evaluate the functionality of various parts of your application working together as a whole. Specifically, these tests ensure that multiple components — such as controllers, database access, middleware, internal and external services — function correctly when integrated, as opposed to functioning correctly only in isolation (which would be covered by unit tests).

Here is a quick one liner summary on popular kinds on testing on API codebase

> ***Unit tests*** *validate individual components like controllers or services in isolation*
>
> ***Integration tests*** *verify the interaction between multiple components (e.g., API, database, middleware) as a cohesive system*
>
> ***Contract tests*** *ensure that API endpoints conform to agreed-upon interfaces or expectations between services*

Integration tests will help you in identifying possible bugs introduced due to any new changes in you code

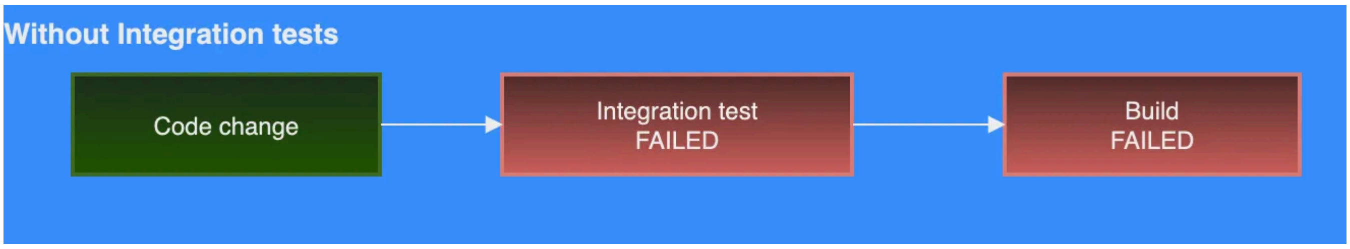


Image 1: Build process when integration tests fail

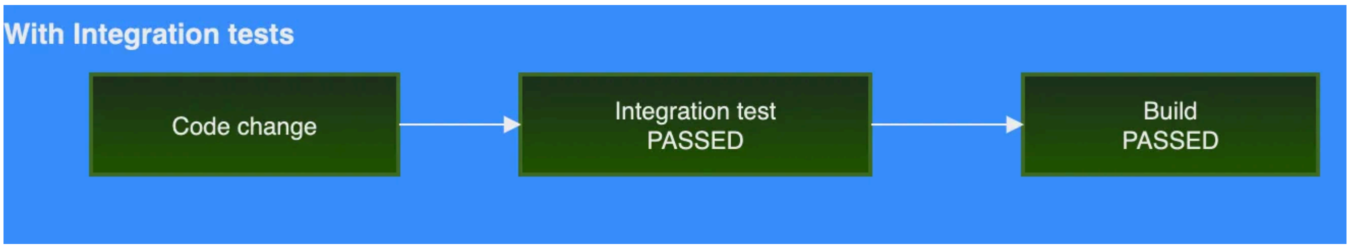


Image 2: Build process when integration tests Pass

Ways to write integration tests:

Now there are various ways to write integration tests

- Using Postman or any other API testing tool
- Deploying services to actual environment, either server or cloud service
- Using WebApplicationFactory in dotnet to run an in memory server locally

We will be discussing the **`WebApplicationFactory`** method, which is fast, flexible, configurable and does not require any additional tools or hosting environment.

Some of the advantages of this approach as below:Using **`WebApplicationFactory`** for integration testing in ASP.NET Core offers several advantages:

1. **Realistic Environment:** It provides a full-fledged, in-memory test server that mimics the real production environment, allowing you to test the entire request/response pipeline of your application without needing to host it on an actual web server.
2. **End-to-End Testing**: You can perform end-to-end testing, including routing, middleware, dependency injection, and database interactions, ensuring that all components work together as expected.
3. **Customizable Configuration**: You can override or customize the application's configuration (e.g., swapping real services or databases for test versions) to simulate different environments or conditions without affecting the production code.
4. **Easy HTTP Client Access**: `WebApplicationFactory` makes it simple to create an `HttpClient` for sending HTTP requests to your API, making it easy to test API endpoints and validate their responses.
5. **Automatic Startup**: It handles the application's startup process, so you don't have to manually configure or boot the application, saving time and reducing boilerplate code.
6. **Flexible Testing with Dependency Injection**: You can easily replace services, middleware, or database contexts using the test server's dependency injection, which allows for testing specific scenarios like using an in-memory database.
7. **Seamless Integration with Test Frameworks:** It integrates well with popular testing frameworks like xUnit, NUnit, or MSTest, and is optimized for use in .NET Core testing scenarios, reducing friction in writing and running tests.

Diving into the code

For a demo code, we will be referring a code repository, which stores the information for superhero personalities. I'll be sharing the github link of the code repo at the end of the article.

Our demo API has 2 APIs

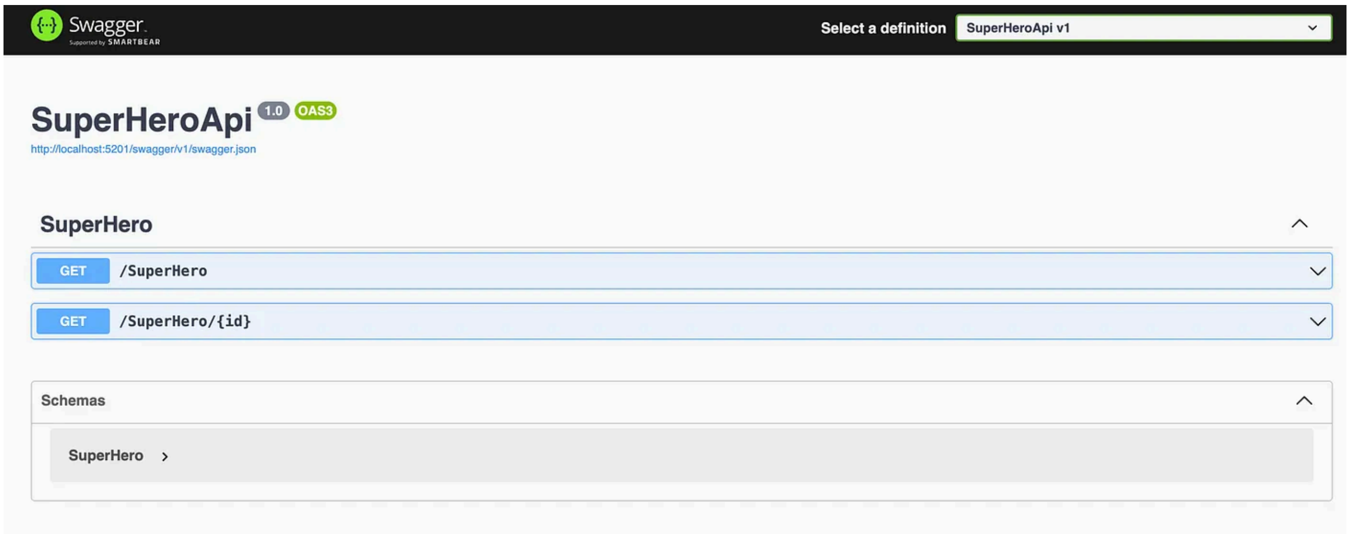


Image 3: Demo API swagger definition

A quick glimpse on our simple SuperHero controller

```
[ApiController]
[Route("[controller]")]
public class SuperHeroController(ISuperHeroRepository superHeroRepository)
    : ControllerBase
{
    [HttpGet("")]
    public async Task<IEnumerable<SuperHero>> Get()
    {
        return await superHeroRepository.GetAllSuperHeroes();
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetById(int id)
    {
        var superHero = await superHeroRepository.GetSuperHeroById(id);
        if (superHero == null)
        {
            return NotFound();
        }

        return Ok(superHero);
        }
    }
```

Since this is basic demo, our ISuperHeroRepository interacts with a Sqlite database which has some predefined SuperHero entries.

Integration test example

Here is an example for our integration test for Get SuperHero By Id scenario

```
        [Fact(DisplayName = "Get superhero by Id returns superhero")]
        public async Task Get_ById_SuperHero_Returns_SuperHero()
        {
            // Arrange
            var factory = new WebApplicationFactory<Program>();
            var htmlClient = factory.CreateClient();

            // Act
            var response = await htmlClient.GetAsync("/SuperHero/1");

            // Assert
            response.StatusCode.Should().Be(HttpStatusCode.OK);
            var superHeroes = await response.Content.ReadFromJsonAsync<SuperHero>()
            superHeroes.Should().NotBeNull();
            superHeroes!.Id.Should().Be(1);
            superHeroes!.SuperName.Should().Be("Batman");
        }
```

Lets understand whats all happening inside test

1. Test Definition and Setup

The test starts with the `[Fact]` attribute from **xUnit**, which defines the test and makes it executable within the test framework. The `DisplayName` parameter provides a readable description for the test, which is helpful for test reports.

```
    [Fact(DisplayName = "Get superhero by Id returns superhero")]
```

This test will ensure that the API correctly retrieves a superhero by ID when we hit the `/SuperHero/1` endpoint.

2. Arranging the Test Environment

In the **Arrange** phase, we use `WebApplicationFactory` to spin up a test server and create an `HttpClient` for sending requests to the API.

```
var factory = new WebApplicationFactory<Program>();
var httpClient = factory.CreateClient();
```

- **`WebApplicationFactory<Program>`** : This initializes a test instance of the API, emulating the real application.
- **`CreateClient()`** : This method returns an `HttpClient` that can be used to make requests to the in-memory test server.

By using `WebApplicationFactory`, we simulate the application environment, allowing us to test the full HTTP pipeline, including routing, middleware, and controllers.

3. Act: Sending the Request

In the **Act** phase, we make a `GET` request to the `/SuperHero/1` endpoint using the `httpClient` created earlier.

```
var response = await httpClient.GetAsync("/SuperHero/1");
```

This line sends an HTTP `GET` request to the API and awaits the response. The application processes the request as it would in production, returning the corresponding superhero (with ID 1) if it exists.

4. Assert: Validating the Response

Now we enter the **Assert** phase, where we check if the response from the API matches our expectations.

Checking the Status Code:

```
response.StatusCode.Should().Be(HttpStatusCode.OK);
```

- This assertion ensures the API returns a status code of **200 OK**, confirming that the request was successful. If the API returned any other status code (like `404 Not Found` or `500 Internal Server Error`), the test would fail.

Validating the Returned Data:
Next, we check the actual content of the response by deserializing the JSON response into a `SuperHero` object:

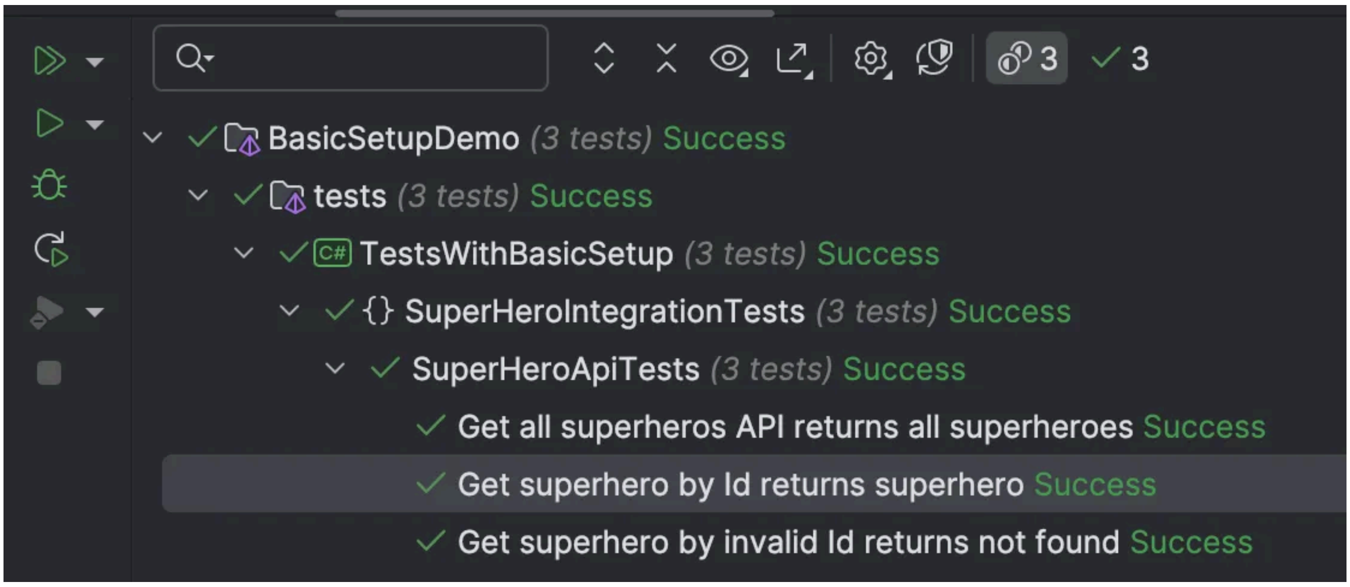
```
var superHero = await response.Content.ReadFromJsonAsync<SuperHero>();
superHero.Should().NotBeNull();
```

- **`ReadFromJsonAsync<SuperHero>()`**: This method reads the JSON response and converts it into a `SuperHero` object. If the response doesn't match the expected structure, or if the superhero doesn't exist, the test will fail.
- **`superHero.Should().NotBeNull()`**: This assertion checks that the API did, in fact, return a superhero. If the API returned `null` (i.e., no superhero was found), the test would fail here.

Verifying Specific Property Values:
Finally, we verify that the returned superhero has the correct properties, specifically checking that the ID is `1` and the superhero's name is "Batman":

```
superHero!.Id.Should().Be(1);
superHero!.SuperName.Should().Be("Batman");
```

When executed, if everything is good with our code, we get all green tests



```
BasicSetupDemo (3 tests) Success
  tests (3 tests) Success
    TestsWithBasicSetup (3 tests) Success
      {} SuperHeroIntegrationTests (3 tests) Success
        SuperHeroApiTests (3 tests) Success
          Get all superheros API returns all superheroes Success
          Get superhero by Id returns superhero Success
          Get superhero by invalid Id returns not found Success
```

Image 4: Success test result

Negative scenario

Lets deliberately try to break our code. The test scenario `*Get superhero by invalid Id returns not found*` expects a 404 status code, when superhero is not found.

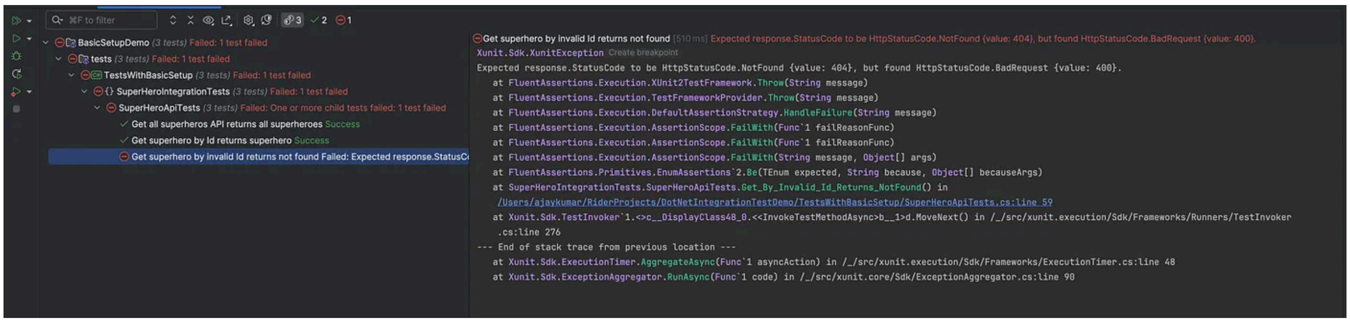
We will make a change in our code to return bad request (400) instead of not found (404).

```
        [HttpGet("{id}")]
        public async Task<IActionResult> GetById(int id)
        {
            var superHero = await superHeroRepository.GetSuperHeroById(id);
            if (superHero == null)
            {
                return BadRequest(); // Changed from NotFound()
            }

            return Ok(superHero);
        }
```

Lets see the test results now. We can see the result is as below

> *Expected response.StatusCode to be HttpStatusCode.NotFound {value: 404}, but found HttpStatusCode.BadRequest {value: 400}*



```
BasicSetupDemo (3 tests) Failed: 1 test failed
  tests (3 tests) Failed: 1 test failed
    TestsWithBasicSetup (3 tests) Failed: 1 test failed
      SuperHeroIntegrationTests (3 tests) Failed: 1 test failed
        SuperHeroApiTests (3 tests) Failed: One or more child tests failed: 1 test failed
          Get all superheros API returns all superheroes Success
          Get superhero by Id returns superhero Success
          Get superhero by invalid Id returns not found Failed: Expected response.StatusC
```

```
Get superhero by invalid Id returns not found [510 ms] Expected response.StatusCode to be HttpStatusCode.NotFound {value: 404}, but found HttpStatusCode.BadRequest {value: 400}.
Xunit.Sdk.XunitException Create breakpoint
Expected response.StatusCode to be HttpStatusCode.NotFound {value: 404}, but found HttpStatusCode.BadRequest {value: 400}.
   at FluentAssertions.Execution.XUnit2TestFramework.Throw(String message)
   at FluentAssertions.Execution.TestFrameworkProvider.Throw(String message)
   at FluentAssertions.Execution.DefaultAssertionStrategy.HandleFailure(String message)
   at FluentAssertions.Execution.AssertionScope.FailWith(Func`1 failReasonFunc)
   at FluentAssertions.Execution.AssertionScope.FailWith(Func`1 failReasonFunc)
   at FluentAssertions.Execution.AssertionScope.FailWith(String message, Object[] args)
   at FluentAssertions.Primitives.EnumAssertions`2.Be(TEnum expected, String because, Object[] becauseArgs)
   at SuperHeroIntegrationTests.SuperHeroApiTests.Get_By_Invalid_Id_Returns_NotFound() in
    /Users/ajaykumar/RiderProjects/DotNetIntegrationTestDemo/TestsWithBasicSetup/SuperHeroApiTests.cs:line 59
   at Xunit.Sdk.TestInvoker`1.<>c__DisplayClass48_0.<<InvokeTestMethodAsync>b__1>d.MoveNext() in /_/src/xunit.execution/Sdk/Frameworks/Runners/TestInvoker
    .cs:line 276
--- End of stack trace from previous location ---
   at Xunit.Sdk.ExecutionTimer.AggregateAsync(Func`1 asyncAction) in /_/src/xunit.execution/Sdk/Frameworks/ExecutionTimer.cs:line 48
   at Xunit.Sdk.ExceptionAggregator.RunAsync(Func`1 code) in /_/src/xunit.core/Sdk/ExceptionAggregator.cs:line 90
```

Image 5: Failure result status

This is it for the basic setup demo. I will be covering more in the future articles like, how to work with database, authentication, events etc.

You can find the code repo used in this article at https://github.com/ajaysskumar/SuperHeroSolution. I am also reachable at ajay.a338@gmail.com for any feedback and suggestions.

Dotnet Core   Integration Testing   C Sharp Programming   Microsoft

Follow

Written by Ajay kumar

6 Followers · 2 Following

Software enthusiast in architecture and testing. Movie and series lover. Welcome to my blog of professional and personal passions.

No responses yet

Write a response