

Microservices in .NET Core

with examples in Nancy

Christian Horsdal Gammelgaard

SAMPLE CHAPTER





Microservices in .NET Core

by Christian Horsdal Gammelgaard

Chapter 7

Copyright 2017 Manning Publications

brief contents

PART 1 GETTING STARTED WITH MICROSERVICES1

- 1 ■ Microservices at a glance 3
- 2 ■ A basic shopping cart microservice 30

PART 2 BUILDING MICROSERVICES.....55

- 3 ■ Identifying and scoping microservices 57
- 4 ■ Microservice collaboration 79
- 5 ■ Data ownership and data storage 109
- 6 ■ Designing for robustness 134
- 7 ■ Writing tests for microservices 155

PART 3 HANDLING CROSS-CUTTING CONCERNS: BUILDING A REUSABLE MICROSERVICE PLATFORM 183

- 8 ■ Introducing OWIN: writing and testing OWIN
middleware 185
- 9 ■ Cross-cutting concerns: monitoring and logging 199

10	■	Securing microservice-to-microservice communication	223
11	■	Building a reusable microservice platform	248
PART 4		BUILDING APPLICATIONS	271
12	■	Creating applications over microservices	273



Writing *tests for microservices*

This chapter covers

- Writing good automated tests
- Understanding the test pyramid and how it applies to microservices
- Testing microservices from the outside
- Writing fast, in-process tests for endpoints
- Using Nancy.Testing for integration and unit tests

Up to this point, you've written a few microservices and set up collaborations between some of them. The implementations are fine, but you haven't written any tests for them. As you write more and more microservices, developing systems without good automated tests becomes unmanageable. In the first half of this chapter, I'll discuss what you need to test for each individual microservice. Then we'll dive into code, looking first at testing endpoints using the `Nancy.Testing` library, and then at testing a complete microservice as if you were sending it requests from another microservice.

7.1 What and how to test

In chapter 1, you saw three characteristics of a microservice that make it good for continuous delivery:

- *Individually deployable*—As soon as any small, safe change has been made to a microservice, the microservice can be deployed to production. But how do you know a change is safe? This is where testing and, particularly, test automation come into the picture. Several other activities, like code reviews, static code analysis, and designing public APIs for backward compatibility, also play into determining that a change is safe, but testing is where much of your confidence will come from.
- *Replaceable*—You should strive to be able to replace the implementation of a microservice with another functionally equivalent implementation within the normal pace of work. Again, tests play an important role, because a good set of tests lets you assess whether the new implementation really is equivalent to the old one.
- *Maintainable by a small team*—Microservices are sufficiently small and focused that a team can maintain several of them. This has the advantage that you can write tests that cover all parts of your microservices.

If you want to become confident about changes quickly and be able to replace a badly implemented microservice, testing has to be fast and repeatable. To make testing fast and repeatable, you must automate a significant part of it—and that’s the focus of this chapter.

7.1.1 The test pyramid: what to test in a microservices system

The *test pyramid* shown in figure 7.1 is a tool you can use to guide which kinds of tests you should write and how many you should have of each kind. You can find variations of the test pyramid in different writings; all of them put tests on different levels, where the levels at the top of the pyramid are broad in scope and the tests at the bottom are narrow. The test pyramid illustrates that you should aim for having many narrowly focused tests (the ones at the wide bottom of the pyramid) and only a few broadly scoped tests (the ones at the narrow top).

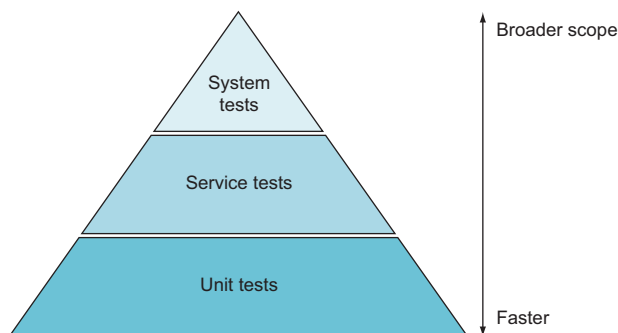


Figure 7.1 The test pyramid illustrates that you should have a few system-level tests, many service-level tests, and even more unit-level tests.

The version of the test pyramid that I use here has three levels:

- *System tests (top level)*—Tests that span the complete system of microservices and are usually implemented through the GUI.
- *Service tests (middle level)*—Tests that work against one, but only one, complete microservice.
- *Unit tests (bottom level)*—Tests that test one small piece of functionality in a microservice. Unit tests call code in the microservice under test in-process and usually involve only part of a microservice.

Note that when I use the term *unit test*, the word *unit* refers to a small piece of functionality. I define the scope of a unit test not in terms of any particular code construct, like a class or a method, but rather in terms of functionality. When we look at implementations of unit tests later, you'll see that unit tests can easily span all layers of a microservice: for example, from a Nancy module, through a domain object, down to a data access class.

Although the test pyramid tells you to have more tests as you move down the levels, exactly how many tests you should have on each level is situational. It depends on such factors as the size of the system, the complexity of the system, and the cost of failure.

7.1.2 System-level tests: testing a complete microservice system end-to-end

The tests at the top of the pyramid have a very broad scope and therefore cover a lot of code with just a few tests. Because they have such a broad scope, they're also imprecise. When a system-level test breaks, it isn't immediately clear where the problem lies. The test can potentially use the entire system, so the issue could be anywhere.

An example of a system-level test is one that uses the web UI of the point-of-sale system we talked about in earlier chapters to add a number of items to an invoice, apply a discount code, and pay using a test credit card. If that test passes, it gives you confidence that invoices are created, that discounts can be applied, and that you can receive credit card payments. During such a system test, you might assert that the amount due on the invoice is as expected. If that assertion fails, any number of things could have caused the problem: you might be using the wrong price for one or more items, you might have applied the discount incorrectly, or you might have misinterpreted the invoice data. In other words, such a failure could be caused by at least a handful of different microservices. To figure out which one is the culprit, you need to investigate.

The specific way a system-level test fails can give some hints as to where the problem lies, but there's usually a lot of code that could be at fault. From the system test alone, it won't even be clear which microservice caused the failure. On the other hand, when system-level tests pass, they give you a good deal of confidence.

The second downside to system-level tests is that they tend to be slow. This again is the flip side of them involving the complete system: real HTTP requests are made, things are written to real data stores, and real event feeds are polled.

Considering that system-level tests, when successful, can give you good confidence, but that they're both slow and imprecise, my advice is to *write system-level tests for the success path of the most important use cases*. This should give you coverage for the success paths of all the most important parts of the system. You can, optionally, supplement this with some tests for the most common and important failure scenarios. Exactly how many system tests this amounts to is, as mentioned earlier, entirely situational. This advice applies equally to microservices, traditional SOA, and monoliths. There's nothing microservice-specific about system-level tests. For this reason, I won't show implementations of any system-level tests in this chapter.

7.1.3 Service-level tests: testing a microservice from outside its process

The tests in the middle level of the test pyramid interact with one microservice as a whole and in isolation—the collaborators of the microservice under test are replaced with *microservice mocks*. Like system tests, these tests interact with the microservice under test from the outside. But unlike system-level tests, they interact directly with the public API of the microservice and make assertions about responses to the microservice as well as the interactions the microservice has with other microservices: for instance, about the commands the microservice under test sends to other microservices.

A microservice mock simulates a real microservice and records interactions

A *microservice mock* can be used in place of a real microservice in service-level tests. It implements the same endpoints as the real microservice, but instead of using real business logic to implement the endpoints, the mock has dumbed-down endpoint implementations; usually endpoints in a mock return hardcoded responses. Furthermore, a mock often records the requests made to the endpoints, so the test code can inspect the requests made during the test.

This is similar to the mock objects widely used in tests for object-oriented code. But where mock objects replace a real object, a microservice mock replaces a real microservice.

Like system-level tests, service-level tests test scenarios rather than single requests. That is, they make a sequence of requests that together form a meaningful scenario. The requests made from the microservice under test to its mocked collaborators are real HTTP requests, and the responses are real HTTP responses.

For examples, recall the Loyalty Program microservice from the example point-of-sale system. In chapter 4, you saw that it collaborated with a number of other microservices, as shown in figure 7.2, using all three collaboration styles: events, queries, and commands.

To test Loyalty Program in isolation, you can create mock versions of its collaborators. As shown in figure 7.3, when Loyalty Program interacts with a mocked collaborator, it gets back a hardcoded response.

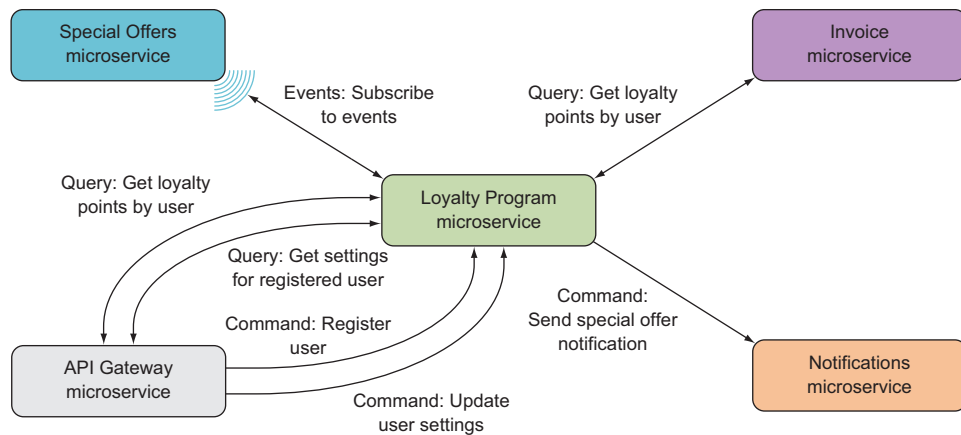


Figure 7.2 The Loyalty Program microservice collaborates with a number of other microservices through all three types of collaboration: events, queries, and commands.

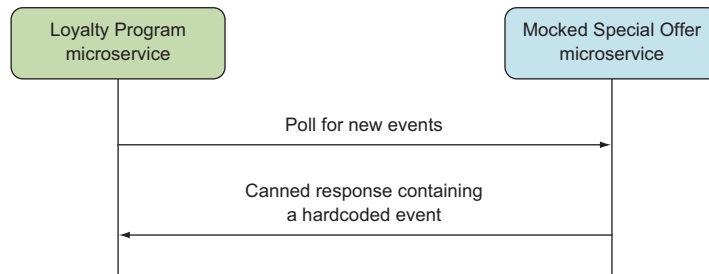


Figure 7.3 For service-level testing, the Loyalty Program microservice interacts with mocked versions of its collaborators. The mocked microservices respond to requests with hardcoded responses.

A service-level test for the Loyalty Program microservice could do the following:

- Send a command to create a user
- Wait for the Loyalty Program microservice to query a mock Special Offer microservice for events, and get back a hardcoded event about a new special offer
- Record any commands sent to the Notifications microservice, and assert that a command for a notification to the new user about the new special offer was sent

When a test like this passes, you can have confidence that important aspects of the Loyalty Program microservice work. When it fails, you know that the problem is within Loyalty Program itself.

Service-level tests are much more precise than system-level tests, because they cover only a single microservice: if such a test fails, the problem should lie within the microservice under test, assuming the test setup itself isn't buggy. Because microservices are small—they're replaceable, after all—knowing that a problem lies within a certain microservice is a lot more precise than what you get from system-level tests.

On the other hand, service-level tests are still slow, because they interact with the microservice under test over HTTP, because the microservice uses a real database, and because it interacts with its mocked collaborators over HTTP.

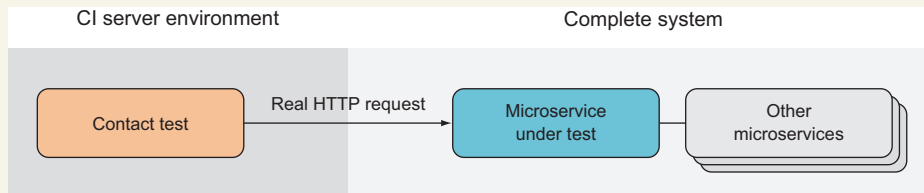
Contract tests

As you know by now, there's a lot of collaboration between microservices in a microservices system. You implement the collaborations as requests from one microservice to another. If you aren't careful, changes in an endpoint can break the microservices that call that endpoint. This is where contract tests come into the picture.

When any two microservices in the system collaborate, the one making requests to the other has some expectations about how the other microservice will behave. That is, given a collaboration, the calling microservice expects the called microservice to implement a certain contract. A *contract test* is a test with the purpose of determining whether the called microservice implements the contract expected by the calling microservice.

Contract tests are written from the point of view of the caller and are there for the sake of the calling microservice: as long as the contract test passes, the assumptions the caller makes about the contract are still valid. Consequently, the contract tests are part of the caller's code base. They aren't part of the same code base as the endpoints they test. Contract tests shouldn't have any knowledge of how the microservices they test are implemented. This is where contract tests differ from service-level tests. With service-level tests, you isolate the microservice under test by providing it with mocked microservices in place of its collaborators. You don't want to do that for contract tests, because the contract tests shouldn't know about the other collaborators of the microservice they test. In other words, contract tests run against the complete system.

Because contract tests are part of the code base of one microservice but test things in other microservices, and because they run against the complete system, it can be a good idea to run them against a QA or staging environment. Moreover, it's a good idea to have them run automatically every time the microservice under test is deployed. When a contract breaks, it's a strong indication that the collaboration between the microservice the contract test belongs to and the microservice under test is broken, too.



A contract test runs against the complete system. It may, for instance, run against a staging or QA environment, where the complete microservices system is deployed.

In terms of implementation, contract tests look a bit like the service-level tests you'll write later in this chapter. The difference is that contract tests are a slightly higher level in the test pyramid, between system-level tests and service-level tests. Contract tests don't set up mocked collaborators, whereas service-level tests do; but just like service-level tests, they work by making real HTTP requests to the microservice under test.

My recommendation regarding service-level tests is that you should write such tests for the success versions of all functionality the microservice under test offers. Such tests will naturally use all endpoints of the microservice as well as rely on any event subscriptions in the microservice. In other words, they will cover all success paths in the microservice. In general, I recommend writing service-level tests only for the most important failure scenarios. Again, the number of service-level tests needed and how many failure scenarios they should cover depends on the system and the cost of failure in that particular system.

7.1.4 Unit-level tests: testing endpoints from within the process

The tests at the bottom of the test pyramid also deal with a single microservice, but these tests don't work over HTTP and don't deal with the entire microservice. These unit tests interact with the parts of the microservice under test directly and in memory. To call the endpoints implemented in your Nancy modules, you'll use the `Nancy.Testing` library that comes as a companion library alongside Nancy. `Nancy.Testing` lets you write tests that make calls to Nancy endpoints in memory. The calls go through Nancy in exactly the same way HTTP requests would, but without going through the network stack. To the code in your Nancy modules, calls made with `Nancy.Testing` look exactly like real HTTP requests.

At the unit-test level, I'll show you two kinds of tests (see figure 7.4): one that uses a database and one that uses a mock in place of the database. I consider both to be unit tests, even though the first type uses a database. Two things make a test a unit test: its scope is a small piece of functionality, and the test code and the production code in the microservice run in the same process.

The narrow scope of a unit test makes it precise: when it fails, the problem lies in a small amount of code. A narrow scope also enables you to write tests that cover failure scenarios properly. Both types of unit tests are faster than service-level tests, but of course the tests that mock out databases are faster than those that use a database. Therefore, you can have both and will probably have more tests that mock the database than tests that don't.

Sometimes you may also have even narrower unit tests that test the business logic in the microservices directly by instantiating domain objects and testing them directly. I take a pragmatic approach to deciding how narrow the narrowest unit tests should be: I use a test-first workflow that starts from the outside, with tests that use `Nancy.Testing` making calls to endpoint handlers in Nancy modules. I start with tests that cover the broad strokes of what the endpoint should do, and then I progressively add tests for more details. Only when it becomes awkward to test a particular detail through the endpoint handler do I begin to write narrower unit tests. For instance, covering a particular case in the business logic with tests that call through the endpoint handler might require a

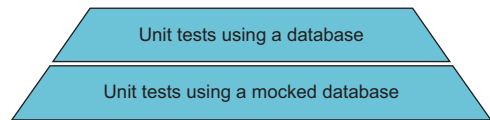


Figure 7.4 At the unit-test level, there are two kinds of tests: those that use a database and those that don't.

lot of setup code. That's a signal to switch down to a test that has a narrower scope: just those cases in the business logic. I'll write tests for those cases that work directly on the classes that should implement that particular part of the business logic.

For the Loyalty Program microservice, you need unit tests that test the endpoint that lets you create users with a number of different inputs covering both possible valid inputs and invalid inputs. Likewise, you need tests that try to read both existing and nonexistent users from the query endpoint that lets you read users. You need similar tests for the other endpoints in the microservice. Loyalty Program is sufficiently simple that you don't need to switch down to tests that are narrower than the microservice's endpoints. So, the units tests I'll show you later all work by calling endpoint handlers through `Nancy.Testing`.

7.2 **Testing libraries: *Nancy.Testing* and *xUnit***

In this chapter, you'll use two new libraries:

- `Nancy.Testing` (<https://github.com/NancyFx/Nancy/wiki/Testing-your-application>)
- `xUnit` (<https://xunit.github.io/>)

I'll give you a brief introduction to each, and then you'll implement tests for some of the microservices you wrote in earlier chapters.

7.2.1 **Meet *Nancy.Testing***

The `Nancy.Testing` library is a companion to `Nancy` that makes it easy to test endpoints implemented in `Nancy` modules. The main entry point into `Nancy.Testing` is the `Browser` type, which accepts method calls like `Get("/")`, `Post("/user")`, `Put("/user/42")`, and `Delete("/user/42")` that let tests call GET, POST, PUT, and DELETE endpoints in `Nancy` modules, respectively. When a test calls an endpoint through the `Browser` type, the call goes through the real `Nancy` pipeline. This means routes are resolved the same way as for real HTTP requests, the dependency injection container is set up and used as usual, and serialization and deserialization run as they normally do. In short, to the endpoint, the call looks exactly like a real HTTP request. The cool thing is that it's all done in process, so it's much faster than a real HTTP request would be. The return value of each method is a `NancyResponse` object and contains everything a real HTTP response would, including headers, status codes, and a body.

In addition to the `Browser` type, the `Nancy.Testing` library provides `ConfigurableBootstrapper`, which offers a nice API for creating ad hoc bootstrappers used in tests. Among other things, `ConfigurableBootstrapper` allows you to do the following:

- Create `Browser` objects that see only one `Nancy` module instead of all modules in the application
- Override registrations in the dependency injection container: for instance, to provide mock objects in place of real ones
- Add hooks to the `Nancy` pipeline, such as an error handler

Finally, *Nancy.Testing* comes with a bunch of convenience methods that make writing assertions against *NancyResponse* objects easy.

Nancy.Testing offers a wealth of functionality that makes it easier to write tests. Going through all of it is beyond the scope of this chapter, but you'll see some of its power. I find the APIs in the library to be quite discoverable, so I'm sure once you get going, you'll discover more of what *Nancy.Testing* has to offer.

You can find further information on *Nancy.Testing* in the *Nancy* documentation (<https://github.com/NancyFx/Nancy/wiki/Testing-your-application>), or you can jump right in and start using it. I think you'll find that the APIs are quite discoverable through IntelliSense.

7.2.2 Meet xUnit

xUnit (<http://xunit.github.io>) is a unit-test tool for .NET. It has a library part that allows you to write automated tests and a runner part that can run those tests. To write a test with *xUnit*, you create a method with a *Fact* attribute over it and put the code to perform the test there. The *xUnit* runner scans for methods with a *Fact* attribute and executes all of them. In addition, *xUnit* has an API for making assertions in tests. If an assertion fails, the *xUnit* runner picks up the failure and reports it back when it's finished running tests. The *xUnit* test runner can be run by *dotnet* and is therefore well suited for the projects you're building in this book.

Other .NET test tools similar to *xUnit*—*NUnit*, for instance—are available that you can also use. This book sticks with *xUnit* because it's used for the test projects that Yeoman and Visual Studio create. If you prefer another tool, feel free to use it, as long as it works with *dotnet*.

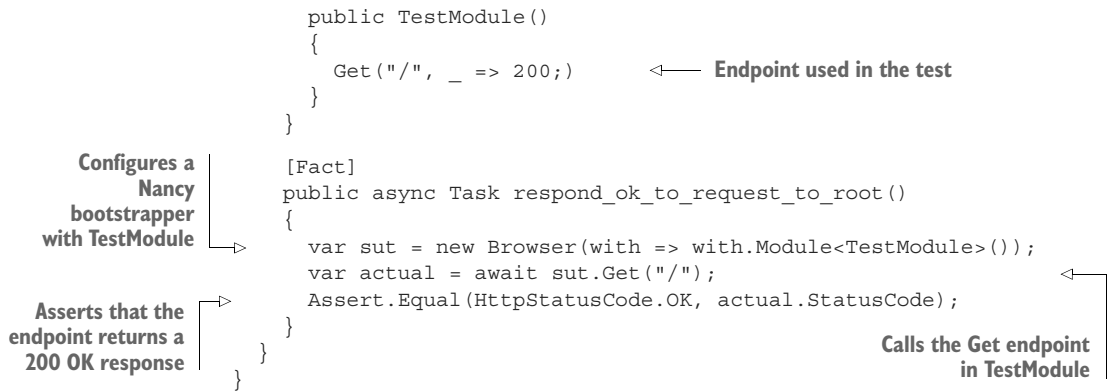
7.2.3 xUnit and Nancy.Testing working together

Putting *Nancy.Testing* and *xUnit* together, you can write succinct tests for endpoints implemented in *Nancy* modules. In section 7.3.1, you'll set up a project for these unit tests and run them with *dotnet*; but for now, I just want to give you a quick peek at how the tests will look. The following test calls the *Get* endpoint in *TestModule* and makes the assertion that the response status code is 200 OK.

Listing 7.1 Simple test using xUnit and Nancy.Testing

```
namespace LoyaltyProgramUnitTests
{
    using Nancy;
    using Nancy.Testing;
    using Xunit;

    public class TestModule_should
    {
        public class TestModule : NancyModule
        {
```



Naming conventions

My tests follow these naming conventions:

- My tests work on an object called *sut* for *system under test*. In the previous test, *sut* is a *Browser* object that I use to make a call to an endpoint.
- I name my test classes after the thing they test—*TestModule* in this example test—followed by *_should*.
- I name the *Fact* method after the scenario being tested and the expected result. I separate the words in *Fact* method names with underscores and try to make sure they form a sentence when combined with the name of the surrounding class. For instance, in this test, concatenating the class name and the *Fact* method name and replacing underscores with spaces, you get “TestModule should respond ok to requests to root.”

Whether you like these conventions is a matter of taste. I happen to like them, but they’re in no way essential to writing good tests.

You can run the previous test with *dotnet*; it will execute in-memory and give you good coverage because the call to *sut.Get("/")* executes the real Nancy pipeline, including the implementation of the endpoint in *TestModule*. The string argument *"/* is the relative URL to which the fake request is made. In section 7.3.1, we’ll look at setting up a project for these unit tests and how to run them with *dotnet*.

For the rest of this chapter, we’ll work at the code level and implement unit tests and service-level tests for the Loyalty Program microservice. When you implemented Loyalty Program in chapter 4, it didn’t have an event feed; but for these examples you’ll add an event feed that other microservices can subscribe to.

7.3 Writing unit tests using *Nancy.Testing*

In this section, you’ll implement some unit tests for the endpoints in the Loyalty Program microservice. In chapter 4, you saw that Loyalty Program has three command and query endpoints:

- An HTTP GET endpoint at URLs of the form `/users/{userId}` that responds with a representation of the user
- An HTTP POST endpoint to `/users/` that expects a representation of a user in the body of the request and then registers that user in the loyalty program
- An HTTP PUT endpoint at URLs of the form `/users/{userId}` that expects a representation of a user in the body of the request and then updates an already-registered user

Let's write tests for these endpoints. The Loyalty Program microservice has an event feed for which you'll also write a test. You won't write comprehensive tests for the endpoints and event feed in Loyalty Program—only enough to see how tests against Nancy endpoints are written.

In the following subsections, you'll do the following:

- Set up a test project to house unit tests for the Loyalty Program microservice.
- Write tests that use `Browser` from `Nancy.Testing` to test endpoints in Loyalty Program and that let the code in the microservice use the real database. You'll write three such tests, one for each of these pieces of functionality:
 - A test that tries to read a user that doesn't exist
 - A test that creates a user and reads it back out
 - A test that modifies a user and reads it back out
- Write tests that also use `Browser` to test an endpoint but are limited in scope by a mocked database injected in the endpoint under test. These tests test the event feed in the microservice.

When you're finished, you'll have learned to write unit tests for Nancy endpoints both with and without a real database.

7.3.1 Setting up a unit-test project

Before you can start writing tests, you need a project to house them. For that, create a new project next to the `LoyaltyProgram` project, and call it `LoyaltyProgramUnit-Tests`. If you create the project with Visual Studio, choose the Class Library (.NET Core) template from the dialog; and if you use you Yeoman, choose Unit Test Project (xUnit.net) from the menu.

Your solution should look similar to this:

```
C:.\
|--LoyaltyProgram
|   Bootstrapper.cs
|   project.json
|   README.md
|   Startup.cs
|   UsersModule.cs
|   YamlSerializerDeserializer.cs
|
|--LoyaltyProgram
```

```

|   └──EventFeed
|       Event.cs
|       EventsFeedModule.cs
|       EventStore.cs
|       IEventStore.cs
|
|──LoyaltyProgramEventConsumer
|   Program.cs
|   project.json
|
|──LoyaltyProgramUnitTests
|   project.json
|   Class1.cs

```

If you used Yeoman to create the new `LoyaltyProgramUnitTests` project, you're ready to run your first tests. But if you used the Visual Studio template, you need to edit the `Class1.cs` and `project.json` files a bit. The following listing shows how `Class1.cs` should look.

Listing 7.2 `Class1.cs` file

```

using Xunit;

namespace UnitTest
{
    // see example explanation on xUnit.net website:
    // https://xunit.github.io/docs/getting-started-dotnet-core.html
    public class Class1
    {
        [Fact]
        public void PassingTest()
        {
            Assert.Equal(4, Add(2, 2));
        }

        [Fact]
        public void FailingTest()
        {
            Assert.Equal(5, Add(2, 2));
        }

        int Add(int x, int y)
        {
            return x + y;
        }
    }
}

```

In the `project.json` file, add the following to set up a test command that refers to the `xunit` test runner:

```
"testRunner": "xunit",
```


The xunit test runner is added to the project via the NuGet package `dotnet-test-xunit`, and the `xUnit` package is installed. Here are all the dependencies:

```
"dependencies": {
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "xunit": "2.1.0"
},
```

You can now go to the `LoyaltyProgramUnitTests` folder in PowerShell and restore the NuGet packages as usual, using `dotnet`:

```
PS> dotnet restore
```

The `Class1.cs` file now contains two small `xUnit` tests: one that passes and one that fails. You run them with `dotnet` like this:

```
PS> dotnet test
```

Once you have the initial tests running, add a dependency on `Nancy.Testing` so you can use `Browser` and later `ConfigurableBootstrapper`. Also add a dependency on `LoyaltyProgram` so you can begin testing it. The dependencies now look like this:

```
"dependencies": {
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "xunit": "2.1.0",
  "Nancy.Testing": "2.0.0--barneyrubble",
  "LoyaltyProgram": {"target": "project"}
},
```

← **Project reference**

The last line is the reference to the `LoyaltyProgram` project. As you can see, the project references in `project.json` look almost like NuGet references. You don't specify a version for `LoyaltyProgram` because you want the test to run against the version of the `LoyaltyProgram` code that you have next to the `LoyaltyProgramUnitTests` project.

7.3.2 Using the Browser object to unit-test endpoints

Now that you have a test project set up, you can begin adding tests to it. The first test you'll add is very simple: given that there are no registered users in the `Loyalty Program` microservice, the test queries for a user and expects to get back a response with a 404 Not Found status code. Add a file called `userModule_should.cs` to the `LoyaltyProgramUnitTests` project, and put the following code in it.

Listing 7.3 First test for the users endpoint

```

namespace LoyaltyProgramUnitTests
{
    using LoyaltyProgram;
    using Nancy;
    using Nancy.Testing;
    using Xunit;

    public class UserModule_should
    {
        private Browser sut;

        public UserModule_should()
        {
            this.sut = new Browser(
                new Bootstrapper(),
                defaultsTo => defaultsTo.Accept("application/json"));
        }

        [Fact]
        public void respond_not_found_when_queried_for_unregistered_user()
        {
            var actual = await sut.Get("/users/1000");
            Assert.Equal(HttpStatusCode.NotFound, actual.StatusCode);
        }
    }
}

```

Remember that sut stands for "system under test."

Real LoyaltyProgram bootstrapper

All "requests" accept JSON

Requests a user that doesn't exist

The most interesting part of this test class is in the constructor, where you create a Browser object. When xUnit runs, it creates an instance of UserModule_should and then calls a method with the Fact attribute on that instance. Unlike most other .NET test frameworks, xUnit create a new, clean instance for each Fact method.

The Browser object in listing 7.3 is initialized with the real bootstrapper from LoyaltyProgram. This means the LoyaltyProgram application that the Browser calls into is wired up exactly the same way it is when it runs on top of a real web server and receives real HTTP requests. Furthermore, for convenience, you set a default Accept header on Browser. This header will be added to all requests made through the Browser object unless explicitly overridden. For instance, sut.Get("/users/1000") has the Accept header set.

Let's move on to a test that registers a new user and then queries it to check that it was registered as it should be. Add the following test to the UserModule_should class.

Listing 7.4 Test for registering a user through the users endpoint

```

[Fact]
public void allow_to_register_new_user()
{
    var expected =
        new LoyaltyProgramUser() { Name = "Chr" };
}

```

```

var registrationResponse = await
    sut.Post("/users", with => with.JsonBody(expected));
var newUser =
    registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();

var actual = await sut.Get($" /users/{newUser.Id}");

Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
Assert.Equal(
    expected.Name,
    actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
// more assertions on the response from the GET
    
```

Reads the new user from the body of the response from the POST

Registers a new user through the POST endpoint

Reads the new user through the GET endpoint

Checks that the response from the GET is correct

Here, you see another use of the Browser object. For instance, you add a body to the Post via the lambda in the second argument. In that lambda, you can do a variety of things to the request, such as adding headers, cookies, form values, a host name, or an identity, or choosing between HTTP and HTTPS. Here, you add a body to the request.

The last test you'll add registers a user and then modifies it via the PUT endpoint in the Loyalty Program microservice. Add it to `UserModule_should.cs`.

Listing 7.5 Test for modifying users through the users endpoint

```

[Fact]
public void allow_modifying_users()
{
    var expected = "jane";
    var user = new LoyaltyProgramUser() { Name = "Chr" };
    var registrationResponse = await
        sut.Post("/users", with => with.JsonBody(user));
    var newUser =
        registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();

    newUser.Name = expected;
    var actual = await
        sut.Put($" /users/{newUser.Id}", with => with.JsonBody(newUser));

    Assert.Equal(
        expected,
        actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
}
    
```

Registers a user

Updates the user

Asserts that the update was done

There's nothing new in this code compared to what you've seen in the two previous tests. But I wanted to include it because it's a good illustration of the kind of unit tests I think you should write for the endpoints in your microservices: unit tests that focus on the behavior the endpoints provide rather than on testing just one endpoint in isolation.


7.3.3 Using a configurable bootstrapper to inject mocks into endpoints

Now that you've tested the endpoints in `UserModule`, let's turn to testing the `LoyaltyProgram` event feed. The event feed is a Nancy module that depends on an `IEventStore` to store and read events. Here's the `IEventStore` interface.

Listing 7.6 `IEventStore` interface

```
using System.Collections.Generic;

namespace LoyaltyProgram.EventFeed
{
    public interface IEventStore
    {
        IEnumerable<Event> GetEvents(
            long firstEventSequenceNumber,
            long lastEventSequenceNumber);
        void Raise(string eventName, object content);
    }
}
```



Reads events from the event store

Stores events to the event store

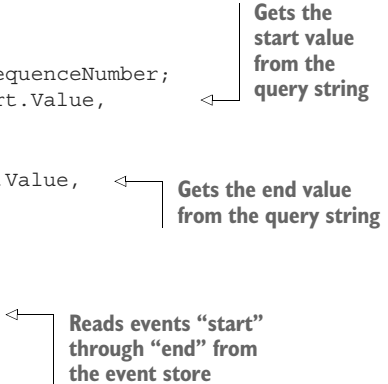
You saw an event feed in chapter 4, but I'll repeat it here, to remind you how it works.

Listing 7.7 Event feed

```
namespace LoyaltyProgram.EventFeed
{
    using Nancy;

    public class EventsFeedModule : NancyModule
    {
        public EventsFeedModule(IEventStore eventStore) : base("/events")
        {
            Get("/", _ =>
            {
                long firstEventSequenceNumber, lastEventSequenceNumber;
                if (!long.TryParse(this.Request.Query.start.Value,
                    out firstEventSequenceNumber))
                    firstEventSequenceNumber = 0;
                if (!long.TryParse(this.Request.Query.end.Value,
                    out lastEventSequenceNumber))
                    lastEventSequenceNumber = 50;

                return
                    eventStore.GetEvents(
                        firstEventSequenceNumber,
                        lastEventSequenceNumber);
            });
        }
    }
}
```



Gets the start value from the query string

Gets the end value from the query string

Reads events "start" through "end" from the event store

As you can see, the event feed is a Nancy module that responds to requests to `/events` with the events it reads from `IEventStore`. You want to write a test to check whether the event feed returns exactly the event from the `IEventFeed`. Toward that end, you want to control which events `IEventStore` returns. So, you'll create a fake implementation of `IEventStore` and use that in the test.

Listing 7.8 Fake IEventStore to use in tests

```
public class FakeEventStore : IEventStore
{
    public IEnumerable<Event> GetEvents(
        long firstEventSequenceNumber,
        long lastEventSequenceNumber)
    {
        if (firstEventSequenceNumber > 100)
            return Enumerable.Empty<Event>();
        else
            return
                Enumerable
                    .Range((int) firstEventSequenceNumber,
                        (int) (lastEventSequenceNumber - firstEventSequenceNumber))
                    .Select(i =>
                        new Event(
                            i,
                            DateTimeOffset.Now,
                            "some event",
                            new Object()));
    }

    public void Raise(string eventName, object content) {}
}
```

← Returns a list of fake events when firstEventSequenceNumber is less than 100

With this fake implementation of an event store, you know the event store will return a list of events only if the `firstEventSequenceNumber` argument is less than 100. Otherwise, `FakeEventStore` will return an empty list of events. If you inject this `IEventStore` implementation into `EventsFeedModule`, you'll know which events `EventsFeedModule` will get from the event store and therefore which events it should return.

You can use another feature of `Nancy.Testing` to inject the fake `IEventStore` implementation into `EventsFeedModule`: `ConfigurableBootstrapper`, which allows you to modify how the Nancy application under test is configured. Here, you'll use `ConfigurableBootstrapper` to set up `FakeEventStore` as the implementation of `IEventStore` when creating the `Browser` object. That is done with the following piece of code.

Listing 7.9 Using the fake event store while testing

```

this.sut = new Browser(
    with => with
        .Module<EventsFeedModule>()
        .Dependency<IEventStore>(typeof(FakeEventStore)),
    withDefault => withDefault.Accept("application/json"));

```

Registers FakeEventStore as the implementation of IEventStore

with has the type ConfigurableBootstrapper

Limits Browser to using EventsFeedModule only

Adds a JSON Accept header to all requests

With this code in the tests, constructor instances of `EventsFeedModule` will have `FakeEventStore` injected. You can use that to write two tests:

- A test that asserts that events are returned from the feed when the start number in the request is less than 100
- A test that asserts that no events are returned when the start number is greater than 100

Listing 7.10 Tests for the event feed, using the fake event store

```

using System;
using System.Collections.Generic;
using System.Linq;
using LoyaltyProgram.EventFeed;
using Nancy;
using Nancy.Testing;
using Xunit;

public class EventFeed_should
{
    private Browser sut;

    public EventFeed_should()
    {
        this.sut = new Browser(
            with => with
                .Module<EventsFeedModule>()
                .Dependency<IEventStore>(typeof(FakeEventStore)),
            withDefault => withDefault.Accept("application/json"));
    }

    [Fact]
    public void return_events_when_from_event_store()
    {
        var actual = await sut.Get("/events/", with =>
        {
            with.Query("start", "0");
            with.Query("end", "100");
        });
    }
}

```

Creates Browser configured to use FakeEventStore

Makes a request to /events with the query string "start=0&end=100"

```

    Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
    Assert.StartsWith("application/json", actual.ContentType);
    Assert.Equal(100,
        actual.Body.DeserializeJson<IEnumerable<Event>>().Count());
}

[Fact]
public void return_empty_response_when_there_are_no_more_events()
{
    var actual = wait sut.Get("/events/", with =>
    {
        with.Query("start", "200");
        with.Query("end", "300");
    });

    Assert.Empty(actual.Body.DeserializeJson<IEnumerable<Event>>());
}
}

```

← Makes a request to /events with the query string "start=200&end=300"

Now that you have some unit tests in place, you can run them with dotnet, as you saw earlier. When you do, xUnit will scan for classes with Fact methods and then execute each Fact method. The output from the tests shows a summary of how many tests ran, how many errors there were, how many tests failed, and how many were skipped:

```

PS > dotnet test
xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: LoyaltyProgramUnitTests
  Discovered:  LoyaltyProgramUnitTests
  Starting:    LoyaltyProgramUnitTests
  Finished:    LoyaltyProgramUnitTests
=== TEST EXECUTION SUMMARY ===
  LoyaltyProgramUnitTests  Total: 6, Errors: 0, Failed: 0, Skipped: 0, Time:
    2.375s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.

```

As you can see, six tests were run, and none of them failed. In other words, all tests passed.

Now that you have tests for EventsFeedModule and UsersModule, you're off to a good start writing unit tests for endpoints in your microservices. In real life, these tests aren't sufficient; I'd write more tests for edge cases and error scenarios. But now you know how to write those tests using Nancy.Testing.

7.4 Writing service-level tests

Let's move on to writing service-level tests for the entire Loyalty Program microservice. Service-level tests interact with a microservice from the outside and provide the microservice with mocked versions of its collaborators.

Loyalty Program makes requests to two collaborators: the event feed in the Special Offers microservice and the API of the Notifications microservice. The service-level tests for Loyalty Program go through these steps:

- 1 Set up two endpoints in the same process as the test:
 - One that works as a mocked special-offer event feed
 - One that works as a mocked notification endpoint
- 2 Start the Loyalty Program microservice in separate processes, and configure it to use the mocked endpoints in place of the real collaborators. This means whenever Loyalty Program needs to call one of its collaborators, it will call one of the mocked endpoints.
- 3 Execute a scenario against Loyalty Program as a sequence of HTTP requests.
- 4 Record any calls to the mocked endpoints.
- 5 Make assertions on the responses from Loyalty Program and on the requests made to the mocked endpoints.

Figure 7.5 shows the runtime setup for the service-level tests for the Loyalty Program microservice.

You'll follow these steps to create the test setup from figure 7.5:

- 1 Create a test project for the service-level tests.
- 2 Create the mocked endpoints for the special-offers event feed and the notification endpoint.
- 3 Start both processes of the Loyalty Program microservice: the Nancy application containing the HTTP API and the event consumer.
- 4 Write test code that executes a test scenario against Loyalty Program.

When that setup is in place, you'll write a test that uses it.

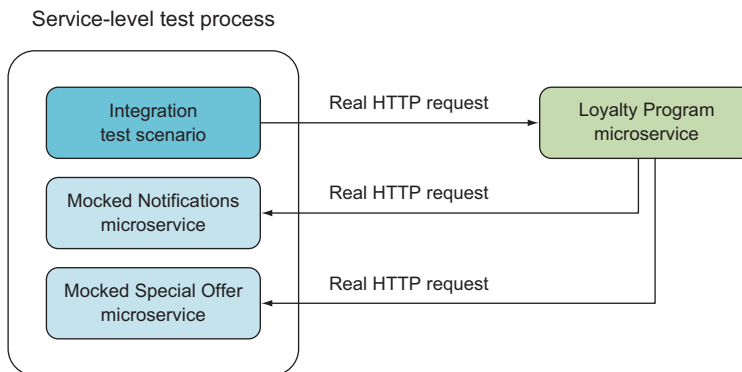


Figure 7.5 A service-level test executes a scenario against the API of the microservice under test but configures the microservice to use mocked endpoints running in the same process as the test, in place of real collaborators. When a service-level test runs, it makes real HTTP requests to the microservice under test, which makes real HTTP requests back to mocked endpoints as needed. The test can inspect the responses from the microservice under test as well as the calls it makes to the mocked endpoints.

7.4.1 Creating a service-level test project

For the service-level tests, you'll create a new test project exactly like the unit-test project you create earlier. That is, create a project based on either the ASP.NET Test Project Template in Visual Studio or the Unit Test project template in Yeoman, and call it `LoyaltyProgramIntegrationTest`. Just like the unit-test project, place this new project side by side with `LoyaltyProgram`. You now have four projects:

Mode	LastWriteTime		Length	Name
----	-----		-----	----
d-----	4/6/2016	8:53 PM		LoyaltyProgram
d-----	4/6/2016	8:53 PM		LoyaltyProgramEventConsumer
d-----	4/6/2016	8:53 PM		LoyaltyProgramIntegrationTest
d-----	8/6/2016	10:59 PM		LoyaltyProgramUnitTests

These are the two projects that make up the Loyalty Program microservice—the Nancy application and the event consumer—and the test projects that go along with the microservice.

7.4.2 Creating mocked endpoints

As shown in figure 7.5, you need to create mocked versions of the endpoints in the Special Offers microservice and the Notifications microservice that the Loyalty Program microservice uses. You'll do so by writing two simple Nancy modules, each of which implements an endpoint that returns a hardcoded response. Listing 7.11 shows the mocked special-offers event feed endpoint, and listing 7.12 shows the mocked notifications endpoint.

Listing 7.11 Mock event feed returning hardcoded events

```
public class MockEventFeed : NancyModule
{
    public static AutoResetEvent polled =
        new AutoResetEvent(initialState: false);

    public MockEventFeed()
    {
        this.Get("/events", _ =>
        {
            polled.Set();
            return new []
            {
                new
                {
                    SequenceNumber = 1,
                    Name= "baz",
                    Content = new
                    {

```

Signals to the test that
Loyalty Program has
been polled for events

Returns a hardcoded
response

```

        OfferName = "foo",
        Description = "bar",
        item = new { ProductName = "name" }
    }
}
};
});
}
}

```

Listing 7.12 Mock endpoint that records when it was called

```

public class MockNotifications : NancyModule
{
    public static AutoResetEvent notificationWasSent =
        new AutoResetEvent(initialState: false);

    public MockNotifications()
    {
        this.Get("/notify", _ =>
        {
            notificationWasSent.Set();
            return 200;
        });
    }
}

```

← Used later in the test to make assertions on

← Returns a hardcoded response

The plan is to run these two modules in the test process. To do that, you'll use Nancy on top of ASP.NET Core like you usually do. You need to add the `Microsoft.AspNetCore.Owin` NuGet packages and add Nancy and `LoyaltyProgram` as dependencies. The dependencies section in the `project.json` file now looks like this.

Listing 7.13 Integration project dependencies, including Nancy

```

"dependencies": {
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "Microsoft.NETCore.App": {
        "version": "1.0.0",
        "type": "platform"
    },
    "xunit": "2.1.0",
    "Microsoft.AspNetCore.Owin": "1.0.0",
    "Nancy": "2.0.0-barneyrubble",
    "LoyaltyProgram": { "target": "project" }
},

```

Next, add a file called `RegisterUserAndGetNotification.cs` containing the following code, which uses `Nancy.Hosting.Self` to start a Nancy application in the test process.

Listing 7.14 Starting up Nancy inside the test process

```

public class RegisterUserAndGetNotification : IDisposable
{
    private readonly NancyHost hostForMockEndpoints;

    public RegisterUserAndGetNotification()
    {
        StartFakeEndpoints();
    }

    private void StartFakeEndpoints()
    {
        this.hostForFakeEndpoints = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseStartup<FakeStartup>()
            .UseUrls("http://localhost:5001")
            .Build();

        new Thread(() => this.hostForFakeEndpoints.Run()).Start();
    }
}

public class FakeStartup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseOwin(buildFunc => buildFunc.UseNancy());
    }
}

```

Uses FakeStartup to bootstrap the ASP.NET Core application →

Creates an ASP.NET Core application ←

Lets the ASP.NET Core application listen on port 5001 ←

Adds Nancy to the ASP.NET Core application ←

Later, you'll add a `Fact` method to this class: then, when you run `xUnit`, it will find this class and instantiate it to execute `Fact`. The constructor starts up Nancy, which will automatically discover the `MockEventsFeed` and `MockUsersModule` modules and expose the endpoints defined in them. This is all you need to create mocked endpoints in the service-level test process.

7.4.3 Starting all the processes of the microservice under test

With the mocked endpoints running, you're ready to start up Loyalty Program. The microservice consists of two processes: a Nancy application and the event consumer. You add the code to start those to the setup in `RegisterUserAndGetNotification`. The following listing shows only new code—leave the existing code to start and stop Nancy.

Listing 7.15 Starting the microservice in a separate process

```

public class RegisterUserAndGetNotification : IDisposable
{
    ...
    private Process eventConsumer;
}

```

```

private Process web;

public RegisterUserAndGetNotification()
{
    StartLoyaltyProgram();
    ...
}

private void StartLoyaltyProgram()
{
    StartEventConsumer();
    StartLoyaltyProgramApi();
}

private void StartLoyaltyProgramApi()
{
    var apiInfo = new ProcessStartInfo("dotnet.exe")
    {
        Arguments = "run",
        WorkingDirectory = "../LoyaltyProgram"
    };
    this.api = Process.Start(apiInfo);
}

private void StartEventConsumer()
{
    var eventConsumerInfo = new ProcessStartInfo("dotnet.exe")
    {
        Arguments = "run localhost:5001",
        WorkingDirectory = "../LoyaltyProgramEventConsumer"
    };
    this.eventConsumer = Process.Start(eventConsumerInfo);
}

public void Dispose()
{
    this.eventConsumer.Dispose();
    this.api.Dispose();
}

```

Setup for running the command "dotnet run" in the LoyaltyProgram folder

Starts the LoyaltyProgram process

Setup for running the event consumer

Starts the event-consumer process

Closes the processes, and releases resources

This code spawns two dotnet processes, one for each process in the Loyalty Program microservice. This is like running dotnet from the command line, so running the Nancy application is the same as usual. Running the event consumer is different, and you need to solve these two problems:

- The event consumer expects to run as a Windows service. Now it also needs to be able to run like a simple process.
- In the following line from listing 7.15, the event consumer doesn't understand the command-line argument `localhost:5001`, which is the host name for the mocked endpoints you want the event consumer to use in place of the real collaborators:

```
Arguments = "run localhost:5001",
```

Both of these issues are easy to solve. You just change the Main method in the event consumer to the following.

Listing 7.16 Letting the consumer run as a Windows or normal process

```
public static void Main(string[] args) => new Program().Entry(args);

public void Entry(string[] args)
{
    this.subscriber = new EventSubscriber(args[0]);
    if (args.Length >= 2 && args[1].Equals("--service"))
        Run(this);
    else
    {
        OnStart(null);
        Console.ReadLine();
    }
}
```

Reads the host name from the command-line argument

Runs as a service if there's a --service in the command-line arguments

Runs the start method by hand

Now both processes of the Loyalty Program microservice are started from the test startup code. A nice side effect of the changes to the event consumer is that it's also easier to run by hand for testing reasons.

7.4.4 Executing the test scenario against the microservice under test

Finally, you're ready to write the test. It has three steps:

- 1 Make an HTTP request to register a user.
- 2 Wait for the Loyalty Program microservice to poll for events.
- 3 Assert that a request to the notifications endpoint was made.

In code, the test goes in the RegisterUserAndGetNotification file and is as follows.

Listing 7.17 Service-level test using an outside loyalty program

```
[Fact]
public void Scenario()
{
    RegisterNewUser();
    WaitForConsumerToReadSpecialOffersEvents();
    AssertNotificationWasSent();
}

private async Task RegisterNewUser()
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri("http://localhost:5000");
        var response = await
```

```

        httpClient.PostAsync(
            "/users/",
            new StringContent(
                JsonConvert.SerializeObject(new LoyaltyProgramUser()),
                Encoding.UTF8,
                "application/json")).ConfigureAwait(false);
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
        Console.WriteLine("registered users");
    }
}

private static void WaitForConsumerToReadSpecialOffersEvents()
{
    Assert.True(MockEventFeed.pollled.WaitOne(30000));
    Thread.Sleep(100);
}

private static void AssertNotificationWasSent()
{
    Assert.True(MockNotifications.NotificationWasSent);
}

```

You can run the test in PowerShell with dotnet:

```
PS> dotnet test
```

This will open two command windows: one with each of the processes in the Loyalty Program microservice. The test runs, and, when it finishes, the two windows are closed. The output from xUnit is as follows:

```

Discovering: LoyaltyProgramIntegrationTest
Discovered:  LoyaltyProgramIntegrationTest
Starting:    LoyaltyProgramIntegrationTest
  LoyaltyProgramIntegrationTests.RegisterUserAndGetNotification.Scenario
Finished:    LoyaltyProgramIntegrationTest
=== TEST EXECUTION SUMMARY ===
  LoyaltyProgramIntegrationTest  Total: 1, Errors: 0, Failed: 0, Skipped:
  ➡ 0, Time: 12.563s

```

This test is slow, and you had to do some setup before you were ready to write it. This is why such tests are higher on the test pyramid than the unit tests you wrote earlier. You should have only a few of this kind of test, whereas you can have many unit tests.

7.5 Summary

- The test pyramid tells you to have few system-level tests that test the complete system, several service-level tests for each microservice, and many unit tests for each microservice.
- System-level tests are likely to be slow and are very imprecise.

- You should write system-level tests for important success scenarios, to provide some test coverage for most of the system.
- Service-level tests are likely to be slow, but they're faster and more precise than system-level tests.
- You should write service-level tests for success scenarios and important failure scenarios for each microservice. This adds more test coverage to each microservice than just the system-level tests.
- You can use the process for writing service-level tests as the basis for writing contract tests that verify the assumption one microservice makes about the API and behavior of another microservice. In terms of the test pyramid, contract tests are between system-level tests and service-level tests.
- Unit tests are fast and should be kept fast. They're also precise, because they target a specific, narrow piece of functionality.
- You should write unit tests for success and failure scenarios alike. Use them to cover edge cases that are harder to cover with higher-level tests.
- I recommend working in an outside-in fashion with each microservice: write service-level tests first, and then begin writing unit tests when the service-level tests become awkward to work with.
- The `Nancy.Testing` library is a powerful companion to Nancy that makes it easy to test endpoints in Nancy modules.
- You use the `Browser` type in `Nancy.Testing` to test endpoints through a nice API that lets you simulate HTTP requests. Calls through the `Browser` object look exactly like real HTTP requests to the endpoint handlers in Nancy modules.
- You test endpoints through `Browser` both with real data stores and with mocked data stores.
- You can write service-level tests where you do the following:
 - Write mocked endpoints for the collaborators of the microservice under test, and use Nancy to host these in the test process.
 - Start up all the processes of the microservice under test, passing in the configuration through command-line arguments.
 - Write scenarios that interact with the microservice under test via HTTP requests.
 - Make assertions both on the response from the microservice under test and on the requests it makes to its collaborators.
- You can use the xUnit test framework to write and run your automated tests.
- xUnit can be run with `dotnet`.

Microservices in .NET Core

Christian Horsdal Gammelgaard



Microservice applications are built by connecting single-capability, autonomous components that communicate via APIs. These systems can be challenging to develop because they demand clearly defined interfaces and reliable infrastructure. Fortunately for .NET developers, OWIN (the Open Web Interface for .NET), and the Nancy web framework help minimize plumbing code and simplify the task of building microservice-based applications.

Microservices in .NET Core provides a complete guide to building microservice applications. After a crystal-clear introduction to the microservices architectural style, the book will teach you practical development skills in that style, using OWIN and Nancy. You'll design and build individual services in C# and learn how to compose them into a simple but functional application back end. Along the way, you'll address production and operations concerns like monitoring, logging, and security.

What's Inside

- Design robust and ops-friendly services
- Build HTTP APIs with Nancy
- Expose events via feeds with Nancy
- Use OWIN middleware for plumbing

This book is written for C# developers. No previous experience with microservices required.

Christian Horsdal Gammelgaard is a Nancy committer and a Microsoft MVP.

“A definite must-read for anyone who works in C#/.NET regularly.”

—Nick McGinness, Direct Supply

“Elegant and convincing. Developers will rethink their application architecture.”

—James McGinn
Bull Valley Software

“Brings together two modern technologies and delves deeply into the code.”

—Andy Kirsch
Concur Technologies

“An extremely approachable book that tackles a complex topic.”

—Shahid Iqbal
Head For Cloud

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/microservices-in-net-core

ISBN-13: 978-1-61729-337-5
ISBN-10: 1-61729-337-7



9 781617 293375