Service Testing

At this level, you are testing the interaction to a microservice. The components are now talking to each other without mocks or stubs. Automating tests here is possible but hard to accomplish. If you are the developer of both the calling and the receiving sides, then automating is easier. However, with microservices developed by other teams, testing is usually manual. These tests are also more expensive in time and money because they involve people doing more test preparations. With service testing, you are verifying conditions like

- Use of network
- API interaction
- Sending or receiving messages
- Failure from network or CPU/memory exhaustion

End-to-End Testing

End-to-end testing is the most expensive level. It requires people to test the system as a whole. Here administrators might create users for the system and perhaps making new customer accounts, load requests, invoices, etc. End-to-end testing is about exercising the business logic from a user's perspective. The users do not care about what the system does to process an invoice. They just need to have the confidence that it does and that the system handles not only when correct data is used but also with invalid data. Does your new microservice handle network issues like partitioning and latency and incorrect data by a user?

Consumer-Driven Contract Testing Deep Dive

In this section, we will create examples of consumer-driven contract testing. We will create two services, one being the consumer and the other as the provider. You will see how to use the PactNet library to generate an output file. The provider microservice uses this file to confirm it has not broken the contract. For more information on the Pact framework, check out `https://pact.io`.

Consumer Project

In Visual Studio 2022, we are going to create the first microservice. The first step is to create a new project. Select "ASP.NET Core Web API" from the list of options (see Figure 7-8).

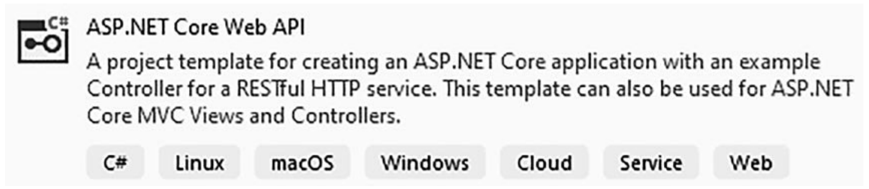

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Figure 7-8. *Project type ASP.NET Core Web API*

Select Next and configure the project name and location (see Figure 7-9). This first project is named OrderSvc-Consumer. It is not required to have "consumer" or "provider" in the name. Those terms are used with the project name to help keep the purpose of the projects obvious.

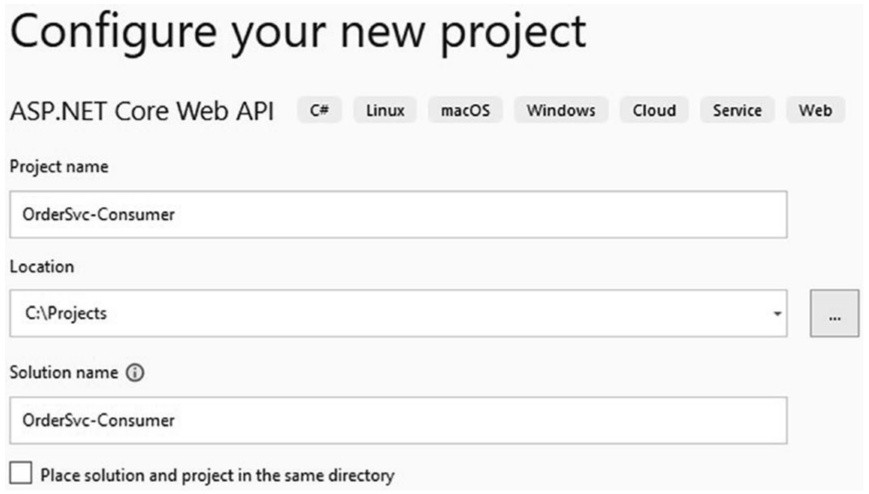
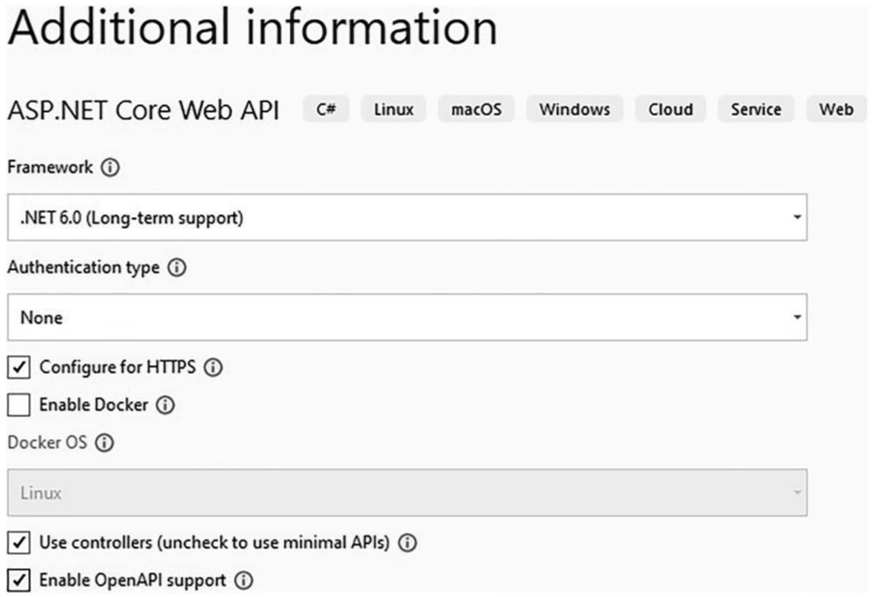

Configure your new project
ASP.NET Core Web API
Project name: OrderSvc-Consumer
Location: C:\Projects
Solution name: OrderSvc-Consumer
[ ] Place solution and project in the same directory

Figure 7-9. *Project name and location*

Select the Framework option for .NET 6 and then the “Create” button (see Figure 7-10).



Additional information

ASP.NET Core Web API

Framework: .NET 6.0 (Long-term support)

Authentication type: None

[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[✓] Use controllers (uncheck to use minimal APIs)

[✓] Enable OpenAPI support

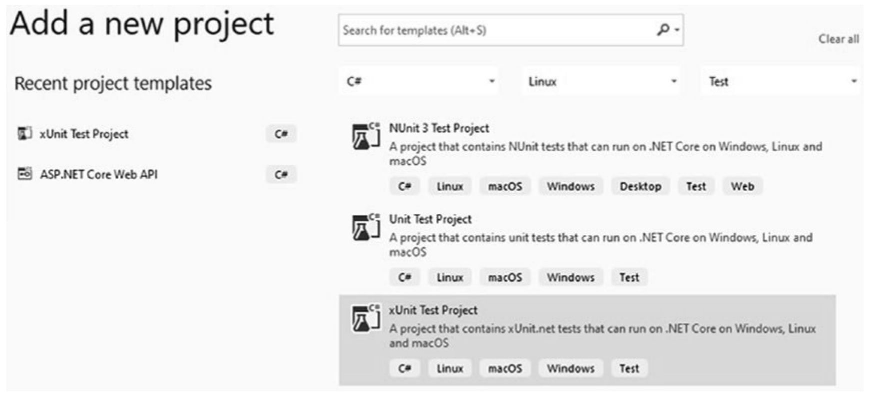
Figure 7-10. *Additional project options*

This service does not need any real code except for one model. The test code here and the mock in the next section will use the DiscountModel class. Create a new class called DiscountModel.

```
public class DiscountModel
{
  public double CustomerRating { get; set; }
  public double AmountToDiscount { get; set; }
}
```

Consumer Test Project

Now we will add a test project. In this example, we are using the xUnit test framework. The PactNet library is not dependent on a specific testing framework. So, you are welcome to pick the testing framework with which you are the most comfortable. Begin by right-clicking the solution and selecting Add ➤ New Project. Now select the project type xUnit Test Project and then the “Next” button (see Figure 7-11).



Add a new project

Recent project templates

- xUnit Test Project (C#)
- ASP.NET Core Web API (C#)

NUnit 3 Test Project
A project that contains NUnit tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Desktop Test Web

Unit Test Project
A project that contains unit tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Test

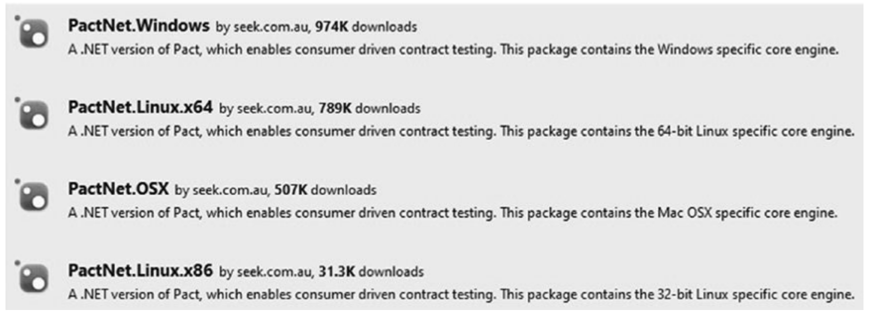
xUnit Test Project
A project that contains xUnit.net tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Test

Figure 7-11. *Selecting the xUnit Test Project type*

Now provide a Project Name and a Location of where to create the new project. You will then choose an appropriate Target Framework. Select the version that matches the main project, which for this book should be .NET 6.0. Now select the “Create” button.

After you create the test project, make a project reference to the project you are testing. In this example, the project reference uses the OrderSvc-Consumer.

You will need to add a NuGet package for the contract testing. First, you need to know which package you choose is based on the OS you are running the tests on (see Figure 7-12). If you are running on Windows, then select the PactNet.Windows library. If running on Linux, select the PactNet.Linux library based on if running on 32 bit (x86) or 64 bit (x64). There is also a library option for OSX.


PactNet.Windows by seek.com.au, 974K downloads
A .NET version of Pact, which enables consumer driven contract testing. This package contains the Windows specific core engine.

PactNet.Linux.x64 by seek.com.au, 789K downloads
A .NET version of Pact, which enables consumer driven contract testing. This package contains the 64-bit Linux specific core engine.

PactNet.OSX by seek.com.au, 507K downloads
A .NET version of Pact, which enables consumer driven contract testing. This package contains the Mac OSX specific core engine.

PactNet.Linux.x86 by seek.com.au, 31.3K downloads
A .NET version of Pact, which enables consumer driven contract testing. This package contains the 32-bit Linux specific core engine.

Figure 7-12. *PactNet library options*

The integration test calls a mock service instead of calling the real discount microservice. In your test project, create a new class called DiscountSvcMock. Then apply the following code:

```
public class DiscountSvcMock : IDisposable
{
  private readonly IPactBuilder _pactBuilder;
  private readonly int _servicePort = 9222;
  private bool _disposed = false;
  public IMockProviderService MockProviderService { get; }
  public string ServiceUri => $"http://localhost:{_servicePort}";

  public DiscountSvcMock()
  {
    var pactConfig = new PactConfig
    {
      SpecificationVersion = "2.0.0",
      PactDir = @"c:\temp\pact\OrderSvcConsumer",
      LogDir = @"c:\temp\pact\OrderSvcConsumer\logs"
    };

    _pactBuilder = new PactBuilder(pactConfig)
        .ServiceConsumer("Orders")
        .HasPactWith("Discounts");
```

```
    MockProviderService = _pactBuilder.MockService(_servicePort,
      new JsonSerializerSettings
        {
          ContractResolver = new CamelCasePropertyNamesContractResolver(),
          NullValueHandling = NullValueHandling.Ignore
        });
    }

  protected virtual void Dispose(bool disposing)
  {
    if (!_disposed)
    {
      if (disposing)
        {
        _pactBuilder.Build();
      }
      _disposed = true;
    }
  }

  public void Dispose()
  {
    Dispose(true);
  }
}
```

In your test class, modify it to the following code. It creates a test that is reliant on the mock service you just created. Because of the mock service, you do not need to run the service itself. Running the test will leverage PactNet and the mock service.

```
public class DiscountSvcTests : IClassFixture<DiscountSvcMock>
{
  private readonly IMockProviderService _mockProviderService;
  private readonly string _serviceUri;

  public DiscountSvcTests(DiscountSvcMock discountSvcMock)
  {
    _mockProviderService = discountSvcMock.MockProviderService;
    _serviceUri = discountSvcMock.ServiceUri;
    _mockProviderService.ClearInteractions();
  }

  [Fact]
  public async Task GetDiscountAdjustmentAmount()
  {
    var discountModel = new DiscountModel { CustomerRating = 4.1 };

    _mockProviderService
      .Given("Rate")
      .UponReceiving("Given a customer rating" +
        ", an adjustment discount amount will be returned.")
      .With(new ProviderServiceRequest
        {
          Method = HttpVerb.Post,
          Path = "/discount",
          Body = discountModel,
          Headers = new Dictionary<string, object>
          {
            {"Content-Type","application/json; charset=utf-8"}
          }
        })
      .WillRespondWith(new ProviderServiceResponse
        {
          Status = 200,
          Headers = new Dictionary<string, object>
          {
            {"Content-Type","application/json; charset=utf-8"}
          },
        Body = new DiscountModel
          {
            CustomerRating = 4.1,
            AmountToDiscount = .41
          }
        });
```

```
    var httpClient = new HttpClient();
    var response = await httpClient
            .PostAsJsonAsync($"{_serviceUri}/discount"
            ,discountModel);
    var discountModelReturned =
      await response.Content.ReadFromJsonAsync<DiscountModel>();

    Assert.Equal(
      discountModel.CustomerRating
      ,discountModelReturned.CustomerRating
    );
  }
}
```

The output of the executed test has two main items. There is an assert statement in the test. But there is also a file created in the folder `C:\temp\pact\OrderSvcConsumer`. You can change the location for the files; however, you will need to provide the same path when creating the provider microservice later.

Looking at the output file that PactNet has created, we can see the details that PactNet required us to set up and the information used by the provider when that service uses this same file.

```
{
  "consumer": {
    "name": "Orders"
  },
  "provider": {
    "name": "Discounts"
  },
  "interactions": [
    {
      "description": "Given a customer rating, an adjustment discount
      amount will be returned.",
      "providerState": "Rate",
      "request": {
        "method": "post",
        "path": "/discount",
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": {
          "customerRating": 4.1,
          "amountToDiscount": 0.0
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": {
          "customerRating": 4.1,
          "amountToDiscount": 0.41
        }
      }
    }
  ],
  "metadata": {
    "pactSpecification": {
      "version": "2.0.0"
    }
  }
}
```

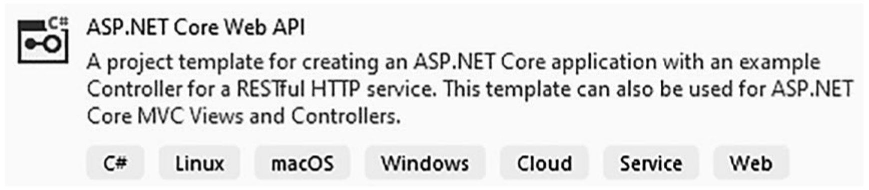
You can see the request is making a POST to the path “/discount.” It uses “application/json” as the content type as well as the “charset=utf-8.” Also noted is the body of the request containing the details of the DiscountModel type. The property “amountToDiscount” shows 0.0, default value of a double since a value was not given.

Also, in the file is the expected response. It includes an HTTP status code of 200, matching headers, and a body. The body includes the supplied customer rating as well as the expected result of calling the real service. Remember, it called the mock service, so all of these details are the setup of expectations. When the provider microservice uses this file, the test library compares the value to the value in the defined response object in this file.

Provider Project

Now that we have made a consumer-driven contract integration test with PactNet, we will create a provider microservice to use the output file. Begin by creating a new ASP.NET Core Web API project (see Figure 7-13).

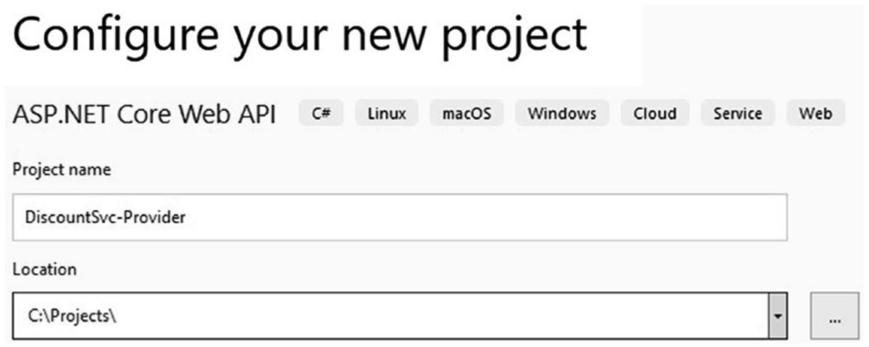
This example will be a fully working microservice. However, the business logic will be just enough to prove the point and help with testing.



ASP.NET Core Web API

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Figure 7-13. *Project type ASP.NET Core Web API*



Configure your new project

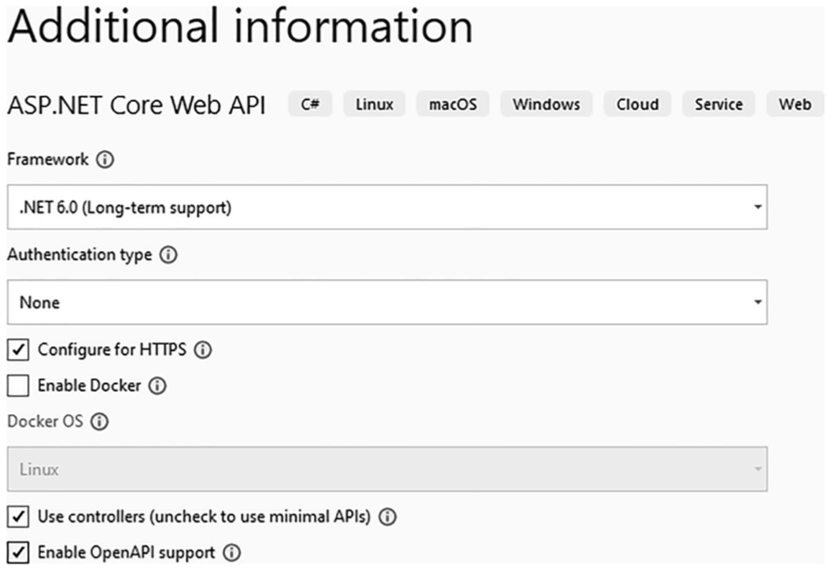
ASP.NET Core Web API

Project name: DiscountSvc-Provider

Location: C:\Projects\

Figure 7-14. *Project name and location*

Provide a project name and location and then select the “Next” button (see Figure 7-14). Now, select the framework .NET 6 and then the “Create” button (see Figure 7-15).



Additional information

ASP.NET Core Web API

Framework: .NET 6.0 (Long-term support)

Authentication type: None

[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[✓] Use controllers (uncheck to use minimal APIs)

[✓] Enable OpenAPI support

Figure 7-15. *Additional project options*

From the project, create two folders, one called Models and the other called Services.

In the Models folder, create a new class called DiscountModel. It has the same properties as the class you made in the consumer project.

```
public class DiscountModel
{
  public double CustomerRating { get; set; }
  public double AmountToDiscount { get; set; }
}
```

In the Services folder, create a new class called DiscountService.

```
public class DiscountService
{
  public double GetDiscountAmount(double customerRating)
  {
    return customerRating / 10;
  }
}
```

The code is just enough to provide a working example. In Program.cs file, replace existing code with this:

```
using DiscountSvc_Provider.Models;
using DiscountSvc_Provider.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddTransient<DiscountService>();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
app.UseRouting();

app.UseEndpoints(endpoints =>
{
  var svc = endpoints
      .ServiceProvider
  .GetRequiredService<DiscountService>();

  endpoints.MapPost("/discount", async context =>
  {
    var model = await context.Request
  .ReadFromJsonAsync<DiscountModel>();

    var amount = svc.GetDiscountAmount(model.CustomerRating);

    await context.Response
      .WriteAsJsonAsync(
        new DiscountModel
        {
          CustomerRating = model.CustomerRating,
          AmountToDiscount = amount
        });
  });
});

app.Run();
```

In the Properties folder, edit the file launchSettings.json. Here we will modify the profile called DiscountSvc_Provider. Because this is a microservice and no real need for anything to be shown to a browser, the setting “launchBrowser” value is false. We are also changing the port used in the “applicationUrl” setting to be something other than the default 5000. This is only required if you run multiple Web API projects for web apps that attempt to open and use the same port.

```
"DiscountSvc_Provider": {
  "commandName": "Project",
  "dotnetRunMessages": "true",
  "launchBrowser": false,
  "applicationUrl": "http://localhost:8080",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Provider Test Project

Now create the test project. The instructions here are the same as what you did for the consumer project. Right-click the solution and select Add ➤ New Project.

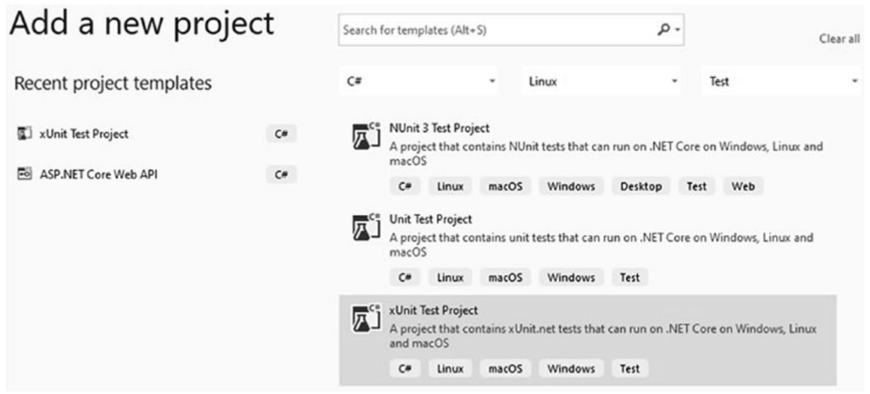
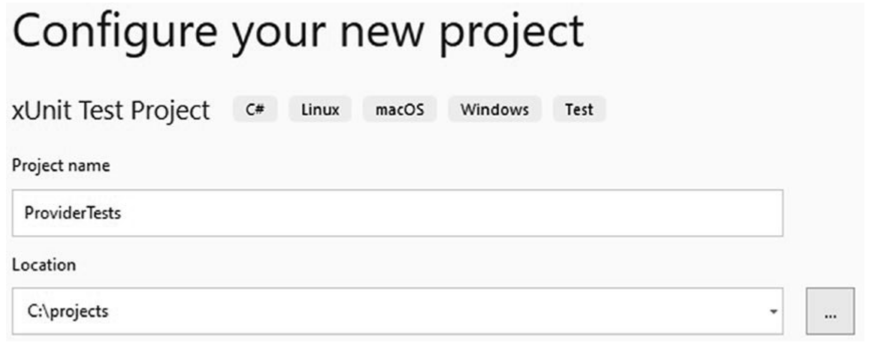

Add a new project
Search for templates (Alt+S)
Clear all
Recent project templates
C#
Linux
Test
xUnit Test Project
ASP.NET Core Web API
NUnit 3 Test Project
A project that contains NUnit tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Desktop Test Web
Unit Test Project
A project that contains unit tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Test
xUnit Test Project
A project that contains xUnit.net tests that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Test

Figure 7-16. *Adding the xUnit test project*

Select the project type for xUnit Test Project and then select the “Next” button (see Figure 7-16).

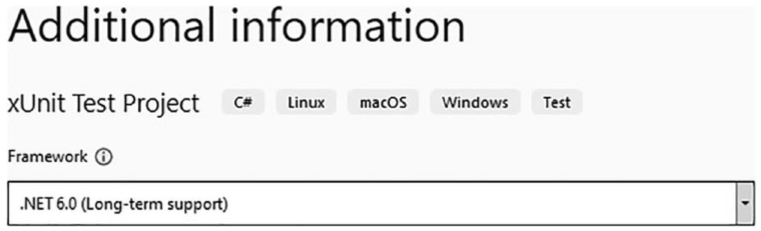


Project name: ProviderTests

Location: C:\projects

Figure 7-17. *Project name and location*

Provide the project name ProviderTests and location (see Figure 7-17). Now, select the framework .NET 6 and then the “Create” button (see Figure 7-18).



.NET 6.0 (Long-term support)

Figure 7-18. *Additional project options*

Add library reference to PactNet for Windows or Linux based on what OS you are running the tests on. Create a class called DiscountServiceTests and then apply the following code:

```
public class DiscountServiceTests : IDisposable
{
  private readonly ITestOutputHelper _output;
  private bool _disposed = false;
  private readonly string _serviceUri;

  public DiscountServiceTests(ITestOutputHelper output)
  {
    _output = output;
    _serviceUri = "http://localhost:8080";
  }

  [Fact]
  public void PactWithOrderSvcShouldBeVerified()
  {
    var config = new PactVerifierConfig
    {
      Verbose = true,
      ProviderVersion = "2.0.0",
      CustomHeaders = new Dictionary<string, string>
      {
        {"Content-Type", "application/json; charset=utf-8"}
      },
```

```
      Outputters = new List<IOutput>
      {
        new XUnitOutput(_output)
      }
    };

    new PactVerifier(config)
      .ServiceProvider("Discounts", _serviceUri)
      .HonoursPactWith("Orders")
      .PactUri(@"c:\temp\pact\OrderSvcConsumer\orders-discounts.json")
      .Verify();
  }
  protected virtual void Dispose(bool disposing)
  {
    if (!_disposed)
    {
      if (disposing)
      {
        //
      }
         _disposed = true;
    }
  }

  public void Dispose()
  {
    Dispose(true);
  }
}
```

Now create a new class called XUnitOutput and apply the following code:

```
using Microsoft.VisualStudio.TestPlatform.Utilities;
using Xunit.Abstractions;
using IOutput = PactNet.Infrastructure.Outputters.IOutput;
```

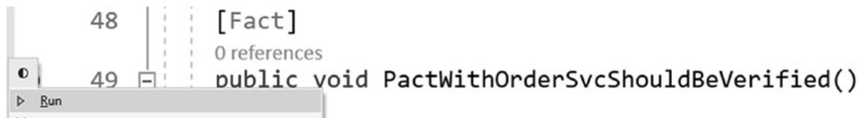
```
public class XUnitOutput : IOutput
{
  private readonly ITestOutputHelper _output;

  public XUnitOutput(ITestOutputHelper output)
  {
    _output = output;
  }

  public void Write(string message, OutputLevel level)
  {
    _output.WriteLine(message);
  }

  public void WriteLine(string line)
  {
    _output.WriteLine(line);
  }

  public void WriteLine(string message, OutputLevel level)
  {
    _output.WriteLine(message);
  }
}
```

There are a few specific items in that code worth mentioning. First, the service URI is the location of the provider project. The test will not succeed unless that project is running. We will need to start that application before running the tests. Second, you must supply the values for "ServiceProvider" and "HonoursPactWith." The values applied here match those in the output Pact contract file generated from the consumer test. Lastly, notice the file path to the file just mentioned. The file path can be an absolute path or a relative path.

With everything compiling, start the Discount Service provider application. You can do this by opening a console window or terminal and traversing to the folder location of the project file and using the .NET CLI.

```
dotnet run
```

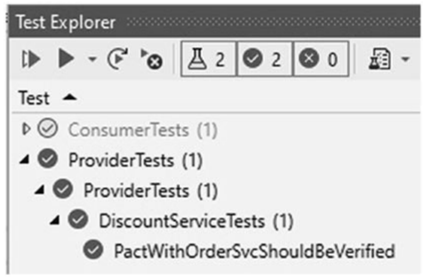
With the provider running, now execute the test that will run against the Pact contract file. Right-click the test and select Run (see Figure 7-19).



```
[Fact]
0 references
public void PactWithOrderSvcShouldBeVerified()
```

Figure 7-19. *xUnit test to execute*

After a small amount of time, you should see a successful test run (see Figure 7-20). The test uses the information in the Pact file to call the Discount microservice at that port location.



```
ConsumerTests (1)
ProviderTests (1)
  ProviderTests (1)
    DiscountServiceTests (1)
      PactWithOrderSvcShouldBeVerified
```

Figure 7-20. *Successful test run*

The conclusion is that you can test for breaking changes when the contract changes. Because many things are affected when a contract changes, there must be communication between the developers of each microservice. The caveat to this is when working with 3rd party or external services. The Google API service, for example, does not have to adhere to requests or contracts of consumers.

Testing Messaging-Based Microservices

In the last section, you learned how to test microservices that use REST-based communications. The challenge now is testing microservices that communicate using messaging. In Chapter 5, you built two microservices that used MassTransit to send messages to each other.

In the code you wrote in Chapter 5, the Payment Microservice receives and reacts to the “InvoiceCreated” message sent by the Invoice Microservice. We will now write different tests – one for the consumer and another for the producer.

Testing Consumer Messaging

Since the code in Chapter 5 is all-in-one solution, we will add two new test projects. In reality, the microservices are not likely to be in one solution. Begin by adding a new xUnit test project (see Figure 7-21).

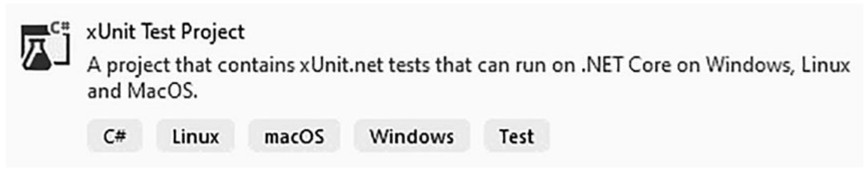

xUnit Test Project
A project that contains xUnit.net tests that can run on .NET Core on Windows, Linux and MacOS.

Figure 7-21. *Selecting the xUnit Test Project type*

Provide a project name and location (see Figure 7-22).

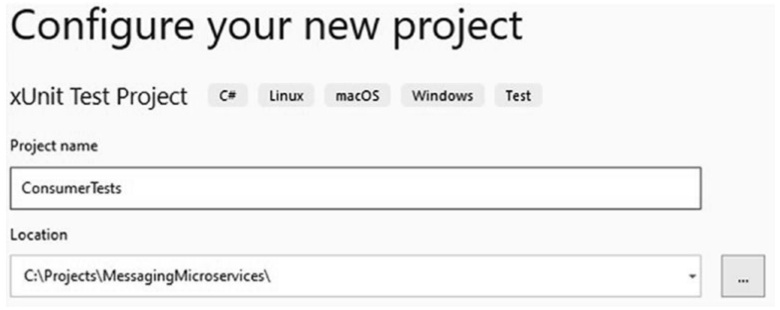

Configure your new project
xUnit Test Project
Project name: ConsumerTests
Location: C:\Projects\MessagingMicroservices\

Figure 7-22. *Project name and location*

Now, select the framework .NET 6 and then the “Create” button (see Figure 7-23). The choice of .NET is not critical but best to use the same version as the other projects in the solution.

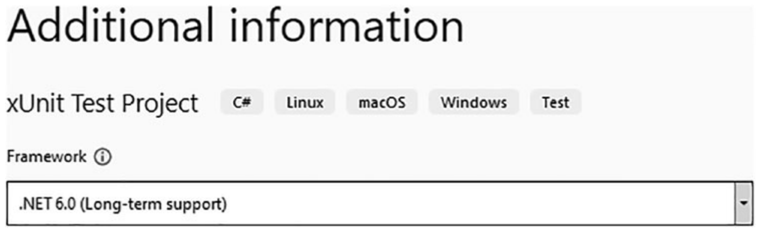
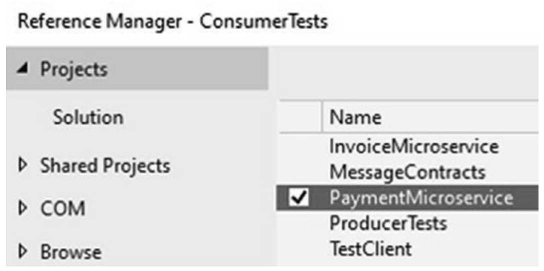

Additional information
xUnit Test Project
Framework: .NET 6.0 (Long-term support)

Figure 7-23. *Additional project options*

Now make a project reference to the Payment Microservice project (see Figure 7-24).


Reference Manager - ConsumerTests

InvoiceMicroservice
MessageContracts
✓ PaymentMicroservice
ProducerTests
TestClient

Figure 7-24. *Setting project reference*

You also need NuGet dependencies MassTransit.TestFramework and FluentAssertions. Get the latest available version of each (see Figure 7-25).

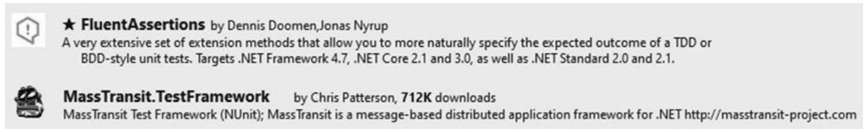

★ FluentAssertions by Dennis Doomen,Jonas Nyrup
A very extensive set of extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style unit tests. Targets .NET Framework 4.7, .NET Core 2.1 and 3.0, as well as .NET Standard 2.0 and 2.1.
MassTransit.TestFramework by Chris Patterson, 712K downloads
MassTransit Test Framework (NUnit); MassTransit is a message-based distributed application framework for .NET http://masstransit-project.com

Figure 7-25. *Selecting MassTransit.TestFramework*

Now we will create a test that will set up a test harness, send a message, and verify the InvoiceCreatedConsumer consumed it. The point here is to test that the consumer of the IInvoiceCreated message receives and reacts to the message. In the test project, open UnitTest1.cs file and add the following code:

```
[Fact]
public async Task Verify_InvoiceCreatedMessage_Consumed()
{
  var harness = new InMemoryTestHarness();

  var consumerHarness = harness.Consumer<InvoiceCreatedConsumer>();
  await harness.Start();

  try
  {
    await harness.Bus.Publish<IInvoiceCreated>(
        new {InvoiceNumber = InVar.Id });

    //verify endpoint consumed the message
    Assert.True(await harness.Consumed.Any<IInvoiceCreated>());

    //verify the real consumer consumed the message
    Assert.True(await consumerHarness.Consumed.Any<IInvoiceCreated>());

    //verify there was only one message published
    harness.Published.Select<IInvoiceCreated>().Count().Should().Be(1);
  }
  finally
  {
    await harness.Stop();
  }
}
```

In this code, you can see the use of the In-Memory Test Harness by MassTransit. The test harness allows us to use a fake bus for sending messages. The message IInvoiceCreated is created and published. There is also a test to verify that the test harness consumed the message and the specific InvoiceCreatedConsumer class consumes the message.

Open the Test Explorer window by selecting Test from the top menu and then selecting Test Explorer (see Figure 7-26). After the test shows in the list, right-click the test and select Run (see Figure 7-27).

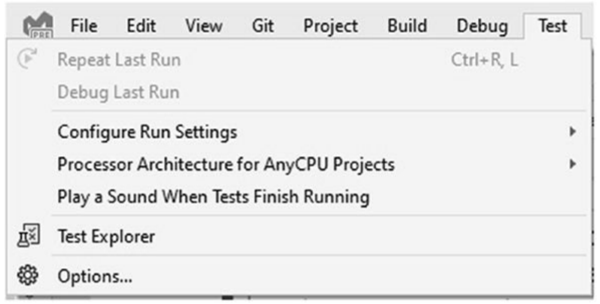

File
Edit
View
Git
Project
Build
Debug
Test
Repeat Last Run Ctrl+R, L
Debug Last Run
Configure Run Settings
Processor Architecture for AnyCPU Projects
Play a Sound When Tests Finish Running
Test Explorer
Options...

Figure 7-26. Open Test Explorer window

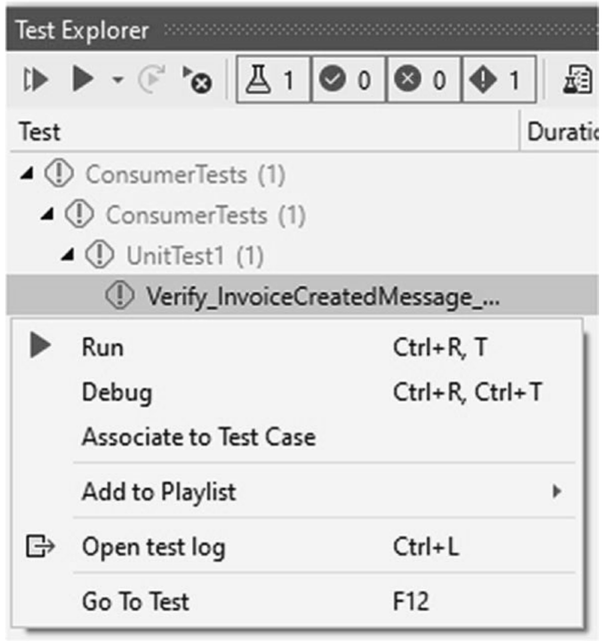


Figure 7-27. Run tests

You should see the results in the Test Explorer (see Figure 7-28).

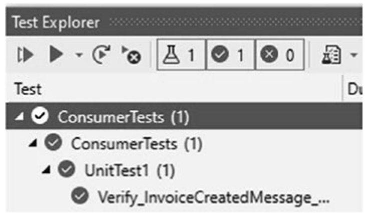


Figure 7-28. *Result of test run*

Testing Producer Messaging

Now we need to write a test for a provider of a message. Just like the last section, right-click the solution and add a new project. Select the project type xUnit Test Project (see Figure 7-29).

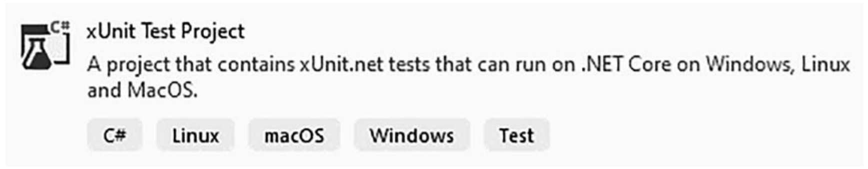
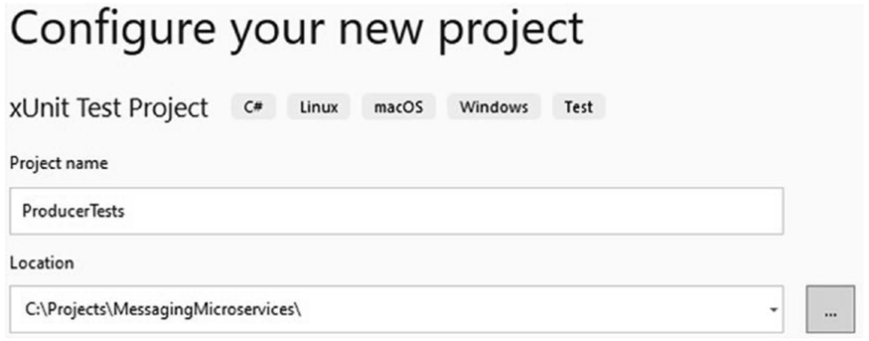

xUnit Test Project
A project that contains xUnit.net tests that can run on .NET Core on Windows, Linux and MacOS.

Figure 7-29. *Selecting the xUnit Test Project type*

Now give the project the name ProducerTests and provide a location for the project files (see Figure 7-30).


Configure your new project

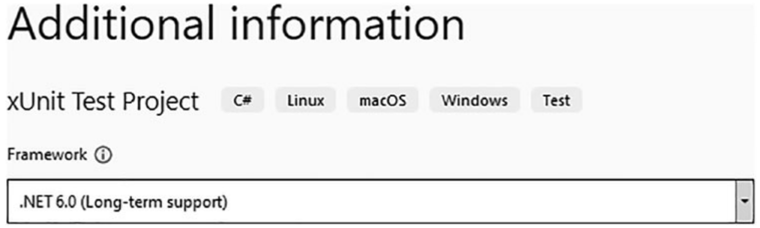
xUnit Test Project

Project name: ProducerTests

Location: C:\Projects\MessagingMicroservices\

Figure 7-30. *Project name and location*

Now select .NET 6 for the Target Framework (see Figure 7-31).


Additional information

xUnit Test Project

Framework: .NET 6.0 (Long-term support)

Figure 7-31. *Additional project options*

The test project needs a project reference connection to the Invoice Microservice project (see Figure 7-32).

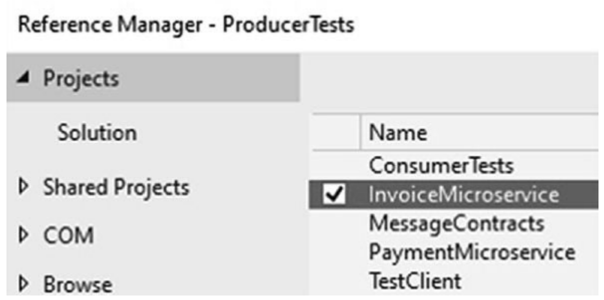
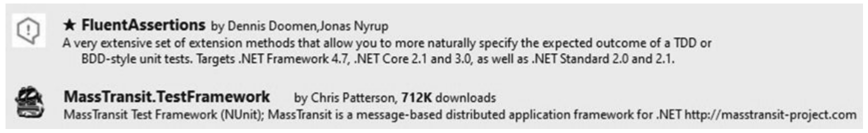


Figure 7-32. *Setting project reference*

Now add NuGet dependencies MassTransit.TestFramework and FluentAssertions. The latest versions are fine (see Figure 7-33).


★ **FluentAssertions** by Dennis Doomen,Jonas Nyrup
A very extensive set of extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style unit tests. Targets .NET Framework 4.7, .NET Core 2.1 and 3.0, as well as .NET Standard 2.0 and 2.1.

MassTransit.TestFramework by Chris Patterson, **712K** downloads
MassTransit Test Framework (NUnit); MassTransit is a message-based distributed application framework for .NET http://masstransit-project.com

Figure 7-33. *Selecting MassTransit.TestFramework*

The objective of this test is to provide a fake message and send it directly to the consumer. The test harness supplies a bus for the fake message and allows us to test the reception of the message. The EventConsumer class in the Invoice Microservice receives and reacts to the IInvoiceToCreate message.

In the UnitTest1.cs file, add the following code:

```
[Fact]
public async Task Verify_InvoiceToCreateCommand_Consumed()
{
  //Verify that we are receiving and reacting
  //to a command to create an invoice
  var harness = new InMemoryTestHarness();
  var consumerHarness = harness.Consumer<EventConsumer>();

  await harness.Start();
```

```
  try
  {
    await harness.InputQueueSendEndpoint.Send<IInvoiceToCreate>(
      new {
        CustomerNumber = 19282,
        InvoiceItems = new List<InvoiceItems>()
        {
          new InvoiceItems
          {
            Description="Tables",
            Price=Math.Round(1020.99),
            ActualMileage = 40,
            BaseRate = 12.50,
            IsHazardousMaterial = false,
            IsOversized = true,
            IsRefrigerated = false
          }
        }
      });

    //verify endpoint consumed the message
    Assert.True(await harness.Consumed.Any<IInvoiceToCreate>());

    //verify the real consumer consumed the message
    Assert.True(await consumerHarness.Consumed.Any<IInvoiceToCreate>());

    //verify that a new message was published
    //because of the new invoice being created
    harness.Published.Select<IInvoiceCreated>().Count().Should().Be(1);
  }
  finally
  {
    await harness.Stop();
  }
}
```

After sending the fake message to the InputQueueSendEndpoint, verify the test harness consumed the message. Then the EventConsumer is checked to verify it has also consumed the message. The reaction of receiving the IInvoiceToCreate message is to send another message. We then verify the publication of the IInvoiceCreated message. You know that message as it is the same as the consumer messaging tests you wrote in the last section.

Using the Test Explorer window, right-click the new test in the list and select Run (see Figure 7-34).

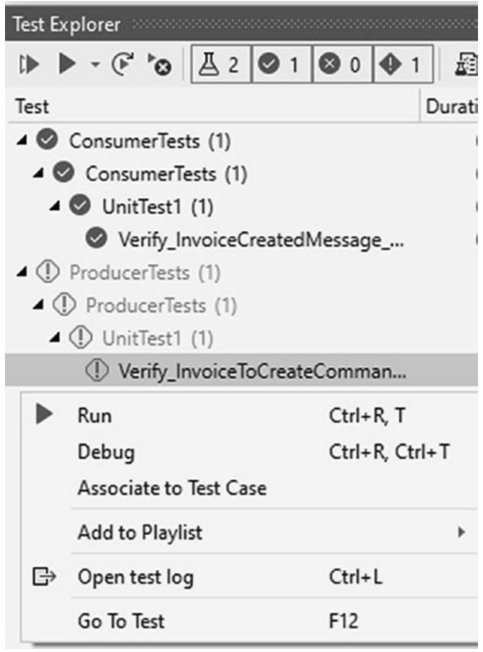


Figure 7-34. *Run test*

You should see the results in the Test Explorer (see Figure 7-35).

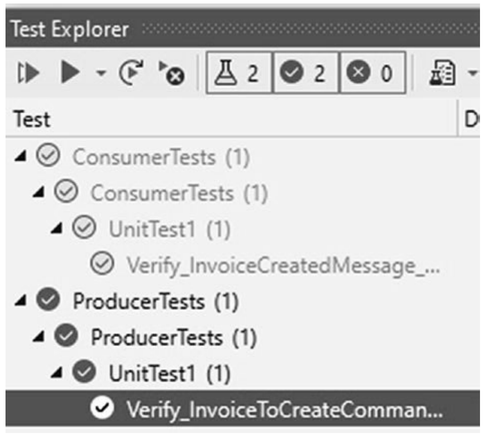


Figure 7-35. *Result of test run*

Summary

In this chapter, we went over how testing plays a critical role in developing any software application. You learned that the cost of errors rises as the project continues development and certainly after deployment to production. So, testing must be considered in the early stages of development and reviewed constantly. Methodologies like Test-Driven Development should be considered as it helps reduce opportunities in a lot of cases. Also, tests should evolve just as the code and architecture change.

You also learned that knowing what to test is as important as knowing how to test them. And knowing what not to test saves time and keeps you focused on error or system failure scenario handling.

We went over the testing pyramid by Mike Cohn and the modified one as it applies to microservices architecture. The modified pyramid shows different types of tests based on what you are testing and why.

Then you learned how to do contract testing for microservices that work together via REST communication. You also created code that uses the Pact library to test the contract between microservices. To finish this chapter, you learned how to implement testing on microservices that work together using messaging communication.