We cannot force the order of the messages, but we can design for handling messages out of order. In our scenario, the message with the order update can include all the information that was in the message to create the order. Since the order has not yet been created, there is no order number. So, the update message was sent knowing that it was an update to an order not yet created. Including the original information in the update message allows a consumer to create the order and apply the update in cases where the order does not exist. When the message to create the order is received, it is ignored as it now already exists.

Building the Examples

Since the microservice we will create in this chapter is meant to process invoices for the company, we need to understand the various states of an invoice. Those listed here are simple examples. You should get with your subject matter experts or domain experts to understand the applicable "real world" invoice states during processing.

- New
- Late
- Modified
- Paid
- Closed

The invoice microservice we will build in this chapter will first have the responsibility of creating an invoice. It may seem like the monolith would be responsible for creating the invoice. But since a microservice is responsible for persisting data, then it should own that data. So, this microservice will receive a request to create an invoice. And part of that process will persist it to a database.

Of course, the microservice we will build here is only an example of a fully featured microservice for processing invoices. This microservice will provide a starting point for you to build more business functionality as you deem necessary.

Building the Messaging Microservices

To demonstrate microservice messaging, we will create three projects: One for the Invoice Microservice. Another for a Payment Microservice. Then a test client will take the place of a monolith. The test client will act as a front-end service with information for the Invoice Microservice to create an invoice. Then the Invoice Microservice will publish a message about the newly created invoice. The Payment Microservice and the test client will both receive the message that the invoice was created. For the test client, it is confirmation the invoice was created. In a real-world scenario, it could display the invoice number to the user. The Payment Microservice serves as a quick example of a downstream microservice that reacts to the creation of the invoice.

Disclaimer The code examples are not meant to be production worthy. They are just enough to help demonstrate concepts.

Running RabbitMQ

You have some options when running RabbitMQ. It can run on a server, on your computer, or in a Docker container. If you would like to install RabbitMQ, go to `https://rabbitmq.com/download.html`. In our examples, we will run RabbitMQ from a Docker container. Docker Desktop is required to be installed. To install Docker Desktop, go to `https://docker.com/products/docker-desktop`.

I did note that we will be using MassTransit for messaging. We will be running MassTransit on top of RabbitMQ. MassTransit provides a layer of abstraction and makes coding easier. It can run on top of RabbitMQ, Azure Service Bus, ActiveMQ, and others.

With Docker Desktop installed, go to a command prompt and enter

```
docker run -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

If you prefer to run the RabbitMQ instance detached from the console:

```
docker run -d -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

You can look at the RabbitMQ Management site by going to `http://localhost:15672`. The default username and password for RabbitMQ are *guest* and *guest*.

With RabbitMQ running, now we will create the microservices. We will start with the microservice for invoices.

First Project

For the first project, we will create a class library on which the other projects will depend. This project will contain the interfaces and classes that become our messages.

In Visual Studio 2022, select the option to Create a New Project. Then select the Class Library option (see Figure 5-5).

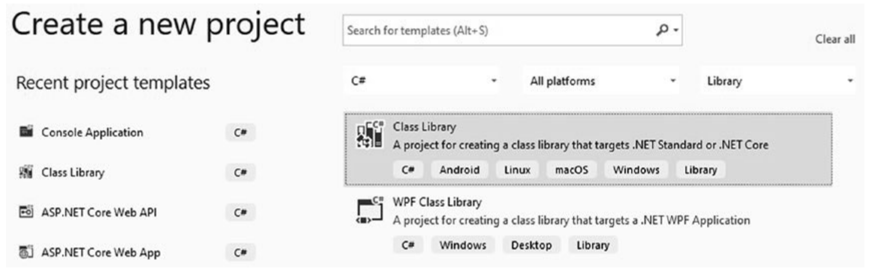
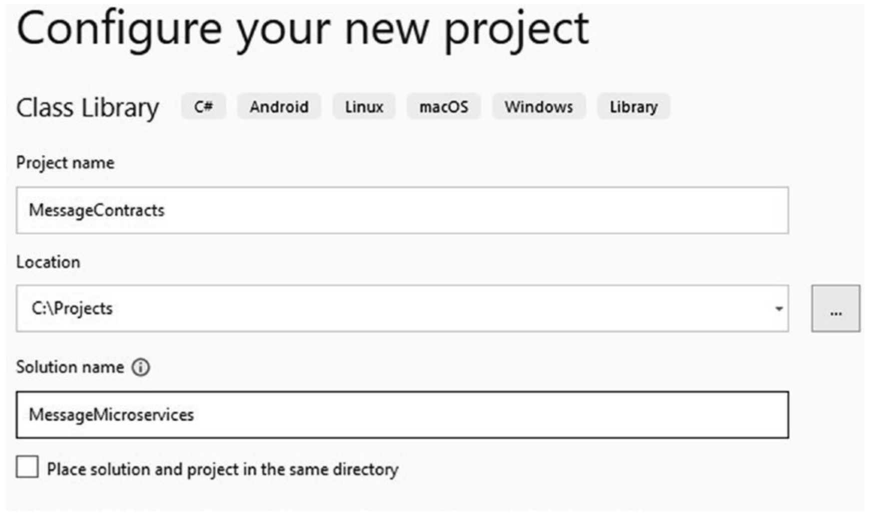

Create a new project
Search for templates (Alt+S)
Clear all
Recent project templates
C#
All platforms
Library
Console Application
Class Library
ASP.NET Core Web API
ASP.NET Core Web App
Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library
WPF Class Library
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library

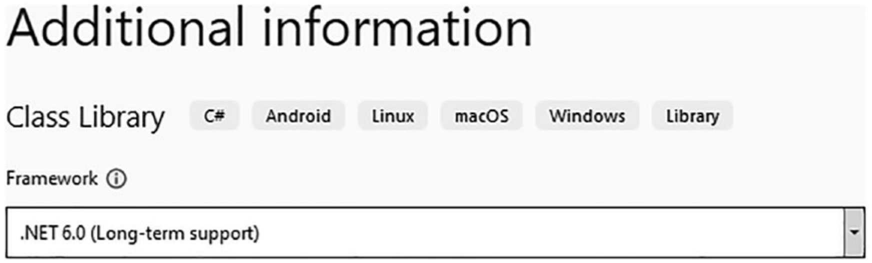
Figure 5-5. New class library


Configure your new project
Class Library C# Android Linux macOS Windows Library

Project name: MessageContracts
Location: C:\Projects
Solution name: MessageMicroservices
☐ Place solution and project in the same directory

Figure 5-6. Naming and selecting file location

Figure 5-6 shows the screen for the project name MessageContracts. Provide a location for your projects, and then for the solution name, use MessagingMicroservices. Make sure the checkbox is not selected to place the solution in the same directory as the project. When done, select the "Next" button.



Framework: .NET 6.0 (Long-term support)

Figure 5-7. Additional project options

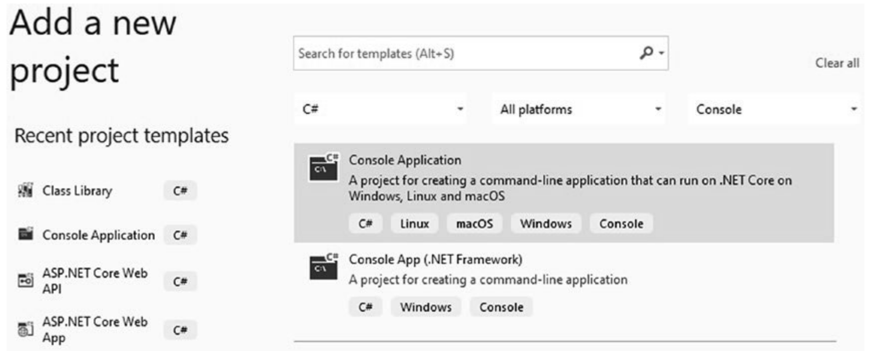
If not already selected, choose the framework for .NET 6 (see Figure 5-7). Then, select the "Create" button. After creating the project, rename the Class1.cs file to MessageContracts.cs. Then replace all the code in that file with the following code:

```
using System.Collections.Generic;

namespace MessageContracts
{
  public interface IInvoiceCreated
  {
    int InvoiceNumber { get; }
    IInvoiceToCreate InvoiceData { get; }
  }

  public interface IInvoiceToCreate
  {
    int CustomerNumber { get; set; }
    List<InvoiceItems> InvoiceItems { get; set; }
  }
  public class InvoiceItems
  {
    public string Description { get; set; }
    public double Price { get; set; }
    public double ActualMileage { get; set; }
    public double BaseRate { get; set; }
    public bool IsOversized { get; set; }
    public bool IsRefrigerated { get; set; }
    public bool IsHazardousMaterial { get; set; }
  }
}
```

Building the Invoice Microservice

Now we will create the first microservice. This microservice is for processing invoices. As an example of a microservice using messaging, it will receive a command to create an invoice. It will then publish an event about the new invoice once created.

Right-click the solution and select Add ➤ New Project.

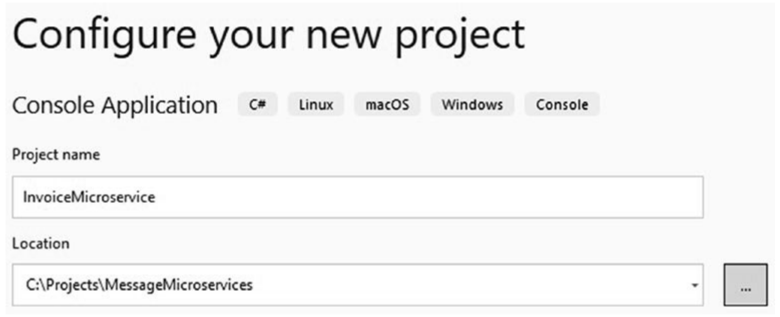


Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS

Console App (.NET Framework)
A project for creating a command-line application

Figure 5-8. *New console application*

Select the option for Console Application and then select the “Next” button (see Figure 5-8).



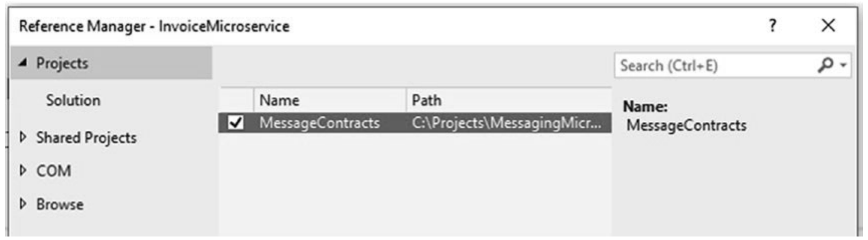
Project name
InvoiceMicroservice
Location
C:\Projects\MessageMicroservices

Figure 5-9. *Project name and location*

Now give the project the name of InvoiceMicroservice and then select the “Next” button (see Figure 5-9). Choose the .NET 6 framework and then select the “Create” button. This project also needs to have the MassTransit library with RabbitMQ installed. In the Package Manager Console, make sure the Default Project has the InvoiceMicroservice selected. Then at the prompt, enter

```
Install-Package MassTransit.RabbitMQ
```

We now need to connect this project to the MessageContract project. We will be writing code that is dependent on interfaces and classes in the MessageContract namespace. Right-click the InvoiceMicroservice project and select Add ➤ Project Reference. Figure 5-10 shows an example of selecting the project MessageContracts.



	Name	Path
✓	MessageContracts	C:\Projects\MessagingMicr...

Name:
MessageContracts

Figure 5-10. *Project dependency*

Select the checkbox left of MessageContracts and then select OK.

In the Program.cs file of InvoiceMicroservice project, replace all of that code with the following code:

```
using GreenPipes;
using MassTransit;
using MessageContracts;

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
  cfg.Host("localhost");
  cfg.ReceiveEndpoint("invoice-service", e =>
  {
    e.UseInMemoryOutbox();
    e.Consumer<EventConsumer>(c =>
      c.UseMessageRetry(m => m.Interval(5, new TimeSpan(0, 0, 10))));
  });
});

var source = new CancellationTokenSource(TimeSpan.FromSeconds(10));
await busControl.StartAsync(source.Token);

Console.WriteLine("Invoice Microservice Now Listening");

try
{
  while (true)
  {
    //sit in while loop listening for messages
    await Task.Delay(100);
  }
}
finally
{
  await busControl.StopAsync();
}
```

```
public class EventConsumer : IConsumer<IInvoiceToCreate>
{
  public async Task Consume(ConsumeContext<IInvoiceToCreate> context)
  {
    var newInvoiceNumber = new Random().Next(10000, 99999);

    Console.WriteLine($"Creating invoice {newInvoiceNumber} for customer:
    {context.Message.CustomerNumber}");


    context.Message.InvoiceItems.ForEach(i =>
    {
      Console.WriteLine($"With items: Price: {i.Price}, Desc:
      {i.Description}");
      Console.WriteLine($"Actual distance in miles: {i.ActualMileage}, Base
      Rate: {i.BaseRate}");
      Console.WriteLine($"Oversized: {i.IsOversized}, Refrigerated:
      {i.IsRefrigerated}, Haz Mat: {i.IsHazardousMaterial}");
    });

    await context.Publish<IInvoiceCreated>(new
    {
      InvoiceNumber = newInvoiceNumber,
      InvoiceData = new
      {
        context.Message.CustomerNumber,
        context.Message.InvoiceItems
      }
    });
  }
}
```

Building the Payment Microservice

Now we will create the PaymentMicroservice project. This project will serve as an example of a downstream microservice that reacts to creating an invoice. Right-click the solution and select Add ➤ New Project.

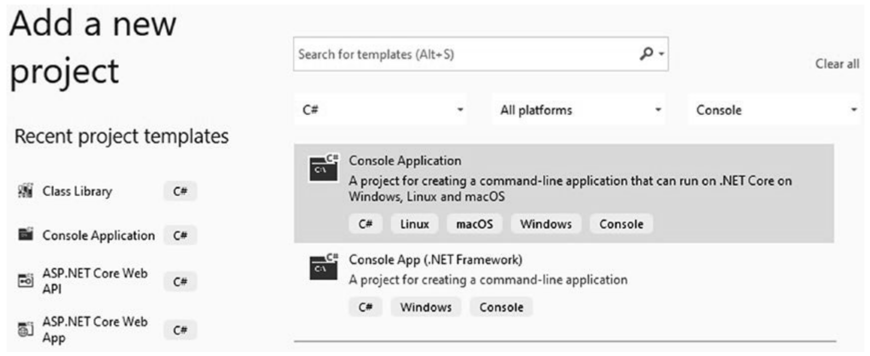
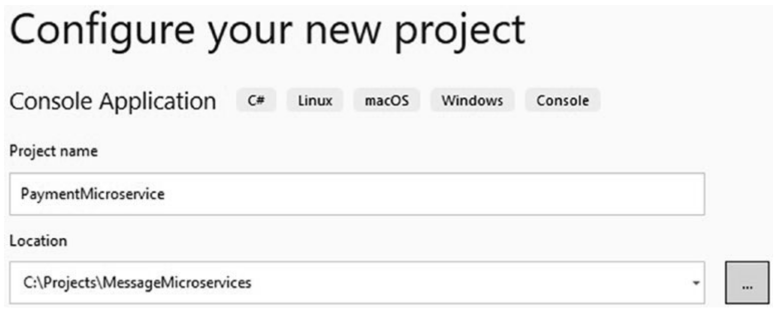

Add a new project
Search for templates (Alt+S)
Clear all
C#
All platforms
Console
Recent project templates
Class Library C#
Console Application C#
ASP.NET Core Web API C#
ASP.NET Core Web App C#
Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console
Console App (.NET Framework)
A project for creating a command-line application
C# Windows Console

Figure 5-11. *New console application*

Select the option for Console Application and then select the “Next” button (see Figure 5-11).


Configure your new project
Console Application C# Linux macOS Windows Console
...

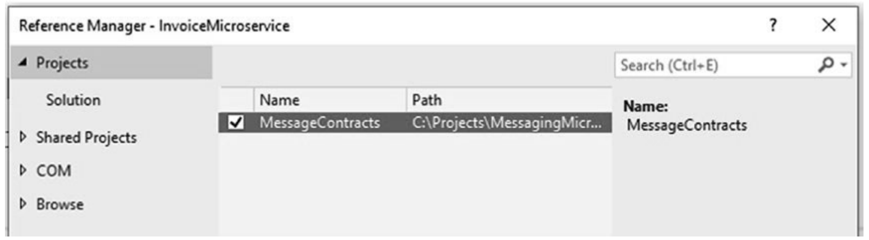
Project name
PaymentMicroservice
Location
C:\Projects\MessageMicroservices

Figure 5-12. *Project name and location*

For the project name, use PaymentMicroservice and then select the “Next” button (see Figure 5-12). Now select .NET 6 for the framework and select the “Create” button. This project also needs to have the MassTransit library with RabbitMQ installed. In the Package Manager Console, make sure the Default Project has the PaymentMicroservice selected. Then at the prompt, enter

```
Install-Package MassTransit.RabbitMQ
```

We now need to connect this project to the MessageContract project. We will be writing code that is dependent on interfaces and classes in the MessageContract namespace. Right-click the InvoiceMicroservice project and select Add ➤ Project Reference.



	Name	Path
✓	MessageContracts	C:\Projects\MessagingMicr...

Name:
MessageContracts

Figure 5-13. *Project dependency*

Select the checkbox left of MessageContracts and then select OK (see Figure 5-13). In the Program.cs file of PaymentMicroservice project, replace all of that code with the following code:

```
using GreenPipes;
using MassTransit;
using MessageContracts;

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
  cfg.Host("localhost");
  cfg.ReceiveEndpoint("payment-service", e =>
  {
    e.Consumer<InvoiceCreatedConsumer>(c =>
      c.UseMessageRetry(m => m.Interval(5, new TimeSpan(0, 0, 10))));
  });
});

var source = new CancellationTokenSource(TimeSpan.FromSeconds(10));
await busControl.StartAsync(source.Token);
Console.WriteLine("Payment Microservice Now Listening");
```

```
try
{
  while (true)
  {
    //sit in while loop listening for messages
    await Task.Delay(100);
  }
}
finally
{
  await busControl.StopAsync();
}

class InvoiceCreatedConsumer : IConsumer<IInvoiceCreated>
{
  public async Task Consume(ConsumeContext<IInvoiceCreated> context)
  {
    await Task.Run(() =>
      Console.WriteLine($"Received message for invoice number: {context.
      Message.InvoiceNumber}"));
  }
}
```

At this point, you do want to make sure the code will compile. But there is nothing to run just yet.

Building a Test Client

We need to create a project that will play the role of what could be in a monolithic application. For simplicity, we will just have the code send a request for a new invoice when the application starts. Once again, right-click the solution and select Add ➤ New Project.

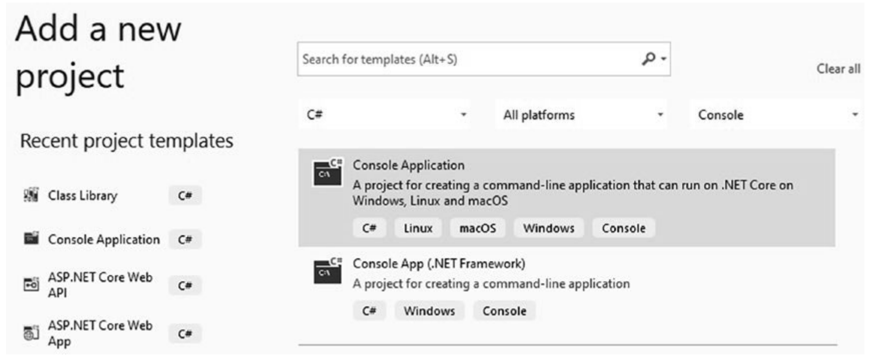

Add a new project
Search for templates (Alt+S)
Clear all
C#
All platforms
Console
Recent project templates
Class Library C#
Console Application C#
ASP.NET Core Web API C#
ASP.NET Core Web App C#
Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console
Console App (.NET Framework)
A project for creating a command-line application
C# Windows Console

Figure 5-14. *New console application*

Select the option for Console Application and then the "Next" button (see Figure 5-14).

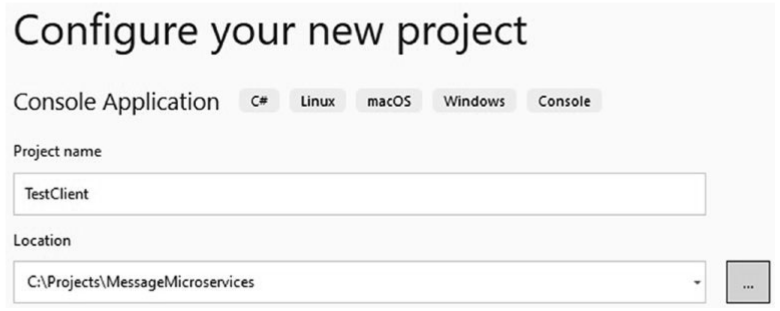
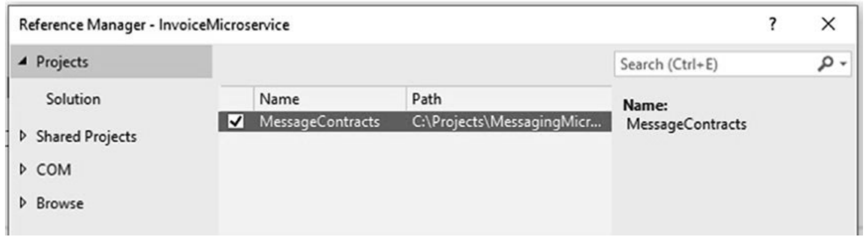

Configure your new project
Console Application C# Linux macOS Windows Console
Project name
TestClient
Location
C:\Projects\MessageMicroservices
...

Figure 5-15. *Project name and location*

For the project name, use TestClient and then select the "Next" button (see Figure 5-15). Select .NET 6 for the framework and then select the "Create" button. This project also needs to have the MassTransit library with RabbitMQ installed. In the Package Manager Console, make sure the Default Project has the PaymentMicroservice selected. Then at the prompt, enter

```
Install-Package MassTransit.RabbitMQ
```

We now need to connect this project to the MessageContract project. We will be writing code that is dependent on interfaces and classes in the MessageContract namespace. Right-click the InvoiceMicroservice project and select Add ➤ Project Reference.



Name	Path
✓ MessageContracts	C:\Projects\MessagingMicr...

Name:
MessageContracts

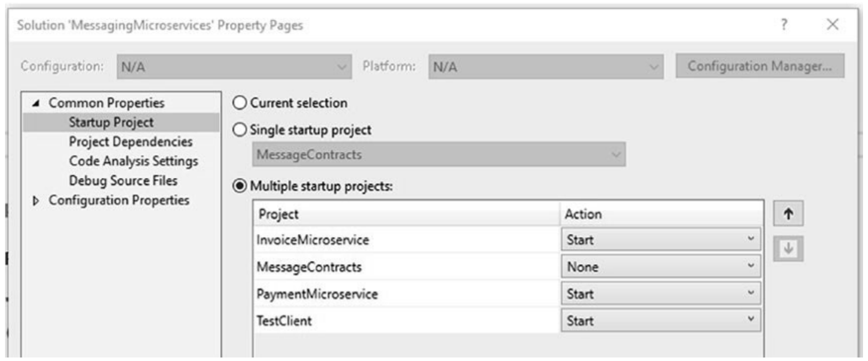
Figure 5-16. *Project dependency*

Select the checkbox left of MessageContracts and then select OK (see Figure 5-16). In the Program.cs file, replace all of the code with the following code:

```
using GreenPipes;
using MassTransit;
using MessageContracts;

Console.WriteLine("Waiting while consumers initialize.");
await Task.Delay(3000); //because the consumers need to start first

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
  cfg.Host("localhost");
  cfg.ReceiveEndpoint("invoice-service-created", e =>
  {
    e.UseInMemoryOutbox();
    e.Consumer<InvoiceCreatedConsumer>(c =>
      c.UseMessageRetry(m => m.Interval(5, new TimeSpan(0, 0, 10))));
  });
});
```

```
var source = new CancellationTokenSource(TimeSpan.FromSeconds(10));

await busControl.StartAsync(source.Token);
var keyCount = 0;
try
{
  Console.WriteLine("Enter any key to send an invoice request or Q to
  quit.");

  while (Console.ReadKey(true).Key != ConsoleKey.Q)
  {
    keyCount++;
    await SendRequestForInvoiceCreation(busControl);
    Console.WriteLine($"Enter any key to send an invoice request or Q to
    quit. {keyCount}");
  }
}
finally
{
  await busControl.StopAsync();
}

static async Task SendRequestForInvoiceCreation(IPublishEndpoint
publishEndpoint)
{
  var rnd = new Random();
  await publishEndpoint.Publish<IInvoiceToCreate>(new
  {
    CustomerNumber = rnd.Next(1000, 9999),
    InvoiceItems = new List<InvoiceItems>()
          {
            new InvoiceItems{Description="Tables", Price=Math.Round(rnd.
            NextDouble()*100,2), ActualMileage = 40, BaseRate = 12.50,
            IsHazardousMaterial = false, IsOversized = true, IsRefrigerated
            = false},
          new InvoiceItems{Description="Chairs", Price=Math.Round(rnd.
          NextDouble()*100,2), ActualMileage = 40, BaseRate =
          12.50, IsHazardousMaterial = false, IsOversized = false,
          IsRefrigerated = false}
        }
  });
}

public class InvoiceCreatedConsumer : IConsumer<IInvoiceCreated>
{
  public async Task Consume(ConsumeContext<IInvoiceCreated> context)
  {
    await Task.Run(() => Console.WriteLine($"Invoice with number: {context.
    Message.InvoiceNumber} was created."));
  }
}
```

We need multiple applications to start at the same time. Right-click the solution and select Set Startup Projects. In the dialog window, select the option for Multiple startup projects. Then change the action of each console application from None to Start. Be sure to leave the MessageContracts project action set to None. Then select OK (see Figure 5-17).


Solution 'MessagingMicroservices' Property Pages

Configuration: N/A   Platform: N/A   Configuration Manager...

- Common Properties
  - Startup Project
  - Project Dependencies
  - Code Analysis Settings
  - Debug Source Files
- Configuration Properties

Current selection
Single startup project: MessageContracts
[✓ Multiple startup projects:]

Project	Action
InvoiceMicroservice	Start
MessageContracts	None
PaymentMicroservice	Start
TestClient	Start

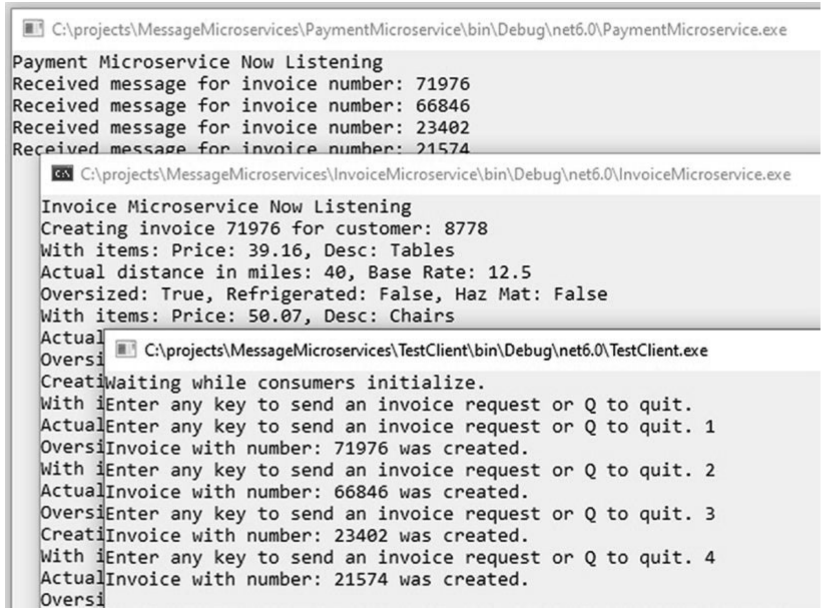
Figure 5-17. Setting start action on multiple projects

Testing What We Have

We need to test and prove that messages are being sent from the Test Client to the invoice Microservice. Then we need to make sure that the message that is sent from the Invoice Microservice is received by the Payment Microservice and the Test Client. Select either the F5 button or the menu option to start the applications (see Figure 5-18).

Figure 5-18. Start debug run option

With RabbitMQ running, start the applications in Visual Studio. This will launch the Test Client, Invoice Microservice, and Payment Microservice. Three windows will show on the screen. On the Test Client screen, press any key, and it will trigger a command message for the Invoice Microservice to create an invoice with the information in the message (see Figure 5-19).


C:\projects\MessageMicroservices\PaymentMicroservice\bin\Debug\net6.0\PaymentMicroservice.exe

```
Payment Microservice Now Listening
Received message for invoice number: 71976
Received message for invoice number: 66846
Received message for invoice number: 23402
Received message for invoice number: 21574
```

C:\projects\MessageMicroservices\InvoiceMicroservice\bin\Debug\net6.0\InvoiceMicroservice.exe

```
Invoice Microservice Now Listening
Creating invoice 71976 for customer: 8778
With items: Price: 39.16, Desc: Tables
Actual distance in miles: 40, Base Rate: 12.5
Oversized: True, Refrigerated: False, Haz Mat: False
With items: Price: 50.07, Desc: Chairs
Actual
Oversi
Creati
With i
Actual
Oversi
With i
Actual
Oversi
Creati
With i
Actual
Oversi
```

C:\projects\MessageMicroservices\TestClient\bin\Debug\net6.0\TestClient.exe

```
Waiting while consumers initialize.
Enter any key to send an invoice request or Q to quit.
Enter any key to send an invoice request or Q to quit. 1
Invoice with number: 71976 was created.
Enter any key to send an invoice request or Q to quit. 2
Invoice with number: 66846 was created.
Enter any key to send an invoice request or Q to quit. 3
Invoice with number: 23402 was created.
Enter any key to send an invoice request or Q to quit. 4
Invoice with number: 21574 was created.
```

Figure 5-19. Example of running the microservices and test client

Play-by-Play Explanation

Now that you have a working application, we will go over some details to better understand how the different pieces work. We will start with the TestClient application. This project intends to be a quick and not full-featured example to fulfill the purpose of a monolith. The monolith in our storyline is an existing application that needs to be modified to send information to the newly created invoice microservice. In this case, it will use messaging instead of a direct call with RPC.

Looking at the example code, you see that it creates an instance of a messaging bus with MassTransit using RabbitMQ. It sets the host to the address, “localhost,” where the RabbitMQ instance in Docker is running. Of course, hard-coding a value like this is generally bad. It is only hard-coded here for the sake of the example.

We then set up a Receive Endpoint. This is because our test client application will also listen for messages in the queue named “invoice-service-created.” Also defined is the InvoiceCreatedConsumer, which tells MassTransit that our code wants to listen to messages sent with the message type of InvoiceCreated. The InvoiceCreatedConsumer is a class defined at the bottom of that file. You can see that it responds to receiving the InvoiceCreated message by simply posting a message on the screen.

The SendRequestForInvoiceCreation method sends information to the Invoice Microservice. It creates some random data as interface type IInvoiceToCreate. The data is serialized, packaged, and sent by MassTransit to the endpoint in RabbitMQ called “invoice-service.” In our case, the Invoice Microservice defines a consumer which creates a queue in RabbitMQ. When the Invoice Microservice runs, it receives the messages from the “invoice-service” queue and processes them (see Figure 5-20).

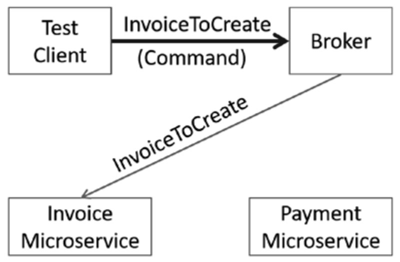


Figure 5-20. *Sending commands to create an invoice*

Looking in the Invoice Microservice code, you see the RabbitMQ host being defined and the “invoice-service” Receive Endpoint setting. The EventConsumer class is of type IConsumer that is of type IInvoiceToCreate. This is reacting to the messages with information for this service to create an invoice. Here you could save information to a database, logging, etc. Notice that when the microservice creates an invoice, it then publishes an event called IInvoiceCreated (see Figure 5-21).

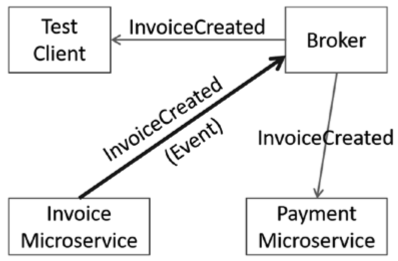


Figure 5-21. *Sending events as a result of invoice creation*

When the new invoice is published as an event, one or more subscribers could receive the same message. In this case, the test client that sent the original information and is a subscriber receives the event message. Also, the other example Payment Microservice is a subscriber. The idea here is that another microservice wants to know and react to the invoice creation event. Then it can do whatever processing is deemed necessary for payment processing based on the new invoice.

Drawbacks of Messaging

Although there are numerous reasons for implementing messaging, it is not without drawbacks. Messaging solutions require effort and time to understand the many pieces that must be decided. Expect to create several proofs of concept to try out the many designs. You will need to judge each design based on complexity, ease of implementation, and manageability variations.

After a messaging solution has been chosen, you then have the infrastructure to create and manage. The messaging product, RabbitMQ, for example, must run on a server someplace. Then for high availability, you must create a cluster on multiple servers. With the additional servers, you have more infrastructure to maintain.

Troubleshooting is also much harder. Since microservices can reside on numerous servers, there may be many log files to comb through when there is an error. You will have to understand if only one microservice failed to receive and process a message and which microservice instance failed.

Since messages can end up in a Dead Letter Queue, you may have to replay that message or decide to delete it. Then, decide if the timeout setting for the DLQ is sufficient. You will also need to verify the messaging system of choice is fully functioning.

Summary

We covered a lot in this chapter. There is so much more to learn with messaging, even if not used with microservices. This chapter provided a high-level overview of messaging with a couple of example microservices. Here you learned that RPC-style communication has many drawbacks.

You also learned about the reasons to use messaging. Messaging is a communication style that helps keep microservices loosely coupled. It also provides buffering, scaling, and independent processing of messages. Each microservice can stay as an independent application, written in the best language for the business need.

In this chapter, you also learned about message types. Commands are for when a microservice is being directed to execute a specific business operation. Queries are for retrieving data, and Events are for alerting subscribers about the fact something has occurred. In our code examples, a command was used to have the Invoice Microservice create an invoice. When that invoice was created, it published an Event. Subscribers like Payment Microservice and the Test Client received the published message and reacted independently with that information.

You also learned that message routing could be done by brokered or broker-less systems. They each have pros and cons that must be evaluated depending on the various business needs you will have. And note, depending on what business needs you are trying to solve, you may have a mix of solutions. Just because a broker-less system solves one aspect of your needs does not mean it will solve them all.

With messaging, you learned about different types of consumers. When you have a consumer subscribed to a queue and scale-out that consumer, multiple consumers compete for the message. Competing consumers help to ensure that only one microservice is processing a specific message. Using independent consumers means that differing microservices can receive the same message. You saw this with the Test Client