Learning NServiceBus Sagas

Discover how to design, build, and test sagas and messaging with NServiceBus

Rich Helton
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 “Saga Development”
- A synopsis of the book’s content
- More information on Learning NServiceBus Sagas

About the Author

Rich Helton, as a principal software engineer, builds and reviews large-scale systems and trains hundreds of developers as well.

Rich has spent over 2 decades in designing and building systems. During this time, he has built, architected, and designed multiple systems, as well as managed many different technical teams. He has built many large-scale enterprise solutions using the most popular C# and Java frameworks and has expertise in the financial, aeronautical, and security domains.

Rich’s passion for designing and teaching HTML5, ESBs, ORM’s test-driven development, NoSQL, iOS, IoCs, and cloud and iPad development was discovered while training developers and architects. He freely shares some of the slides from these trainings on http://www.slideshare.net/rhelton_1.

I would like to thank my wife, Johennie, and my daughters, Ashley and Courtney, for their ongoing support.
Learning NServiceBus Sagas

NServiceBus (NSB) is the most popular Enterprise Service Bus (ESB) in C#. It complements many of the other C# frameworks by providing an end-to-end ESB framework solution to work with services and messaging. The website, http://particular.net/, has many tools to assist in building endpoints, services, and messaging in Visual Studio. Visit http://particular.net/downloads for more details. There are also production tools to check on heartbeats of running endpoints and provide deep insights into running endpoints, services, and messages. NSB provides rapid development to allow integration into many different endpoints and services, for instance, e-mail, Secure File Transfer Protocol (SFTP), and the Windows Communication Foundation (WCF) protocol for web services.

Let's discuss using NHibernate, an object-oriented mapper (ORM) that maps objects to SQL databases, such as MySQL and SQL Server. As a developer, you will need to provide the mapping interface, usually an hbm.xml interface. While creating endpoints and sending messages NSB takes care of the mapping interface. This includes the creation of tables, logging, message durability, message retries, encryption, and many more components that help ensure high quality of software with the use of NSB. NSB provides many components, unique to NSB, needed for automation. NSB provides the following advantages:

- **Separation of duties**: There is separation of duties from the frontend to the backend, allowing the frontend to fire a message to a service and continue in its processing, not worrying about the results until it needs an update. Also, the separation of workflow responsibility exists through separating out NSB services. One service could be used to send payments to a bank, and another service could be used to provide feedback of the current status of the payment to the MVC-EF database, so that a user may see their payment status.

- **Message durability**: Messages are saved to queues between services so that if services are stopped, it can start from the messages in the queues when it restarts, and the messages will persist until told otherwise.

- **Workflow retries**: Messages, or endpoints, can be told to retry a number of times until they completely fail and send an error. The error is automated to return to an error queue. For instance, a web service message can be sent to a bank, and it can be set to retry the web service every 5 minutes for 20 minutes before giving up completely. This is useful during any network or server issues.

- **Monitoring**: NSB ServicePulse can keep a finger on the pulse of its services. Other monitoring can easily be done on the NSB queues to report on the number of messages.

- **Encryption**: Messages between services and endpoints can be easily encrypted.
High availability: Multiple services or subscribers could be processing the same or similar messages from various services that are living on different servers. When one server, or service, goes down, others could be made available to take over those that are already running. Sagas are at the heart of the NServiceBus (NSB) workflow. Sagas save the message state in the form of saga data. They can retrieve the data as it is related to a message to update the message or data. This allows the NSB workflow to control the flow of data and messaging from end to end and correlate saved data to messages in the saga.

Messages are the means to transfer interaction and data between services in a service-oriented architecture (SOA). Sagas correlate, save, route, and manage processes that are started by these messages between services. Sagas even provide timeouts to ensure that messages do not live forever in a system.

Sagas provide decoupling between frontend websites and backend processes. It allows workflow to transfer so that a website can continue to do its work without having to wait for processes to return, such as the dreaded message "Please do not refresh this website as we bill your credit card".

NSB doesn't stop at development on Windows servers and desktops, but plays a big part in cloud development, for example with Microsoft Azure and the Microsoft cloud. NSB uses Service Bus as well as storage queues, SQL Server, and other storage containers. As the cloud is used more and more, NSB plays a key part in Software as a Service, as it is the premier framework for ESB in the C# clouds.

Even if your cloud solution doesn't end up being a C# compatible cloud, as many might hide the software running behind the cloud as being preparatory, NSB is a component for those that will do hybrid solutions, such as keeping data resident on-site. The connection to the cloud then will likely be via web services, and NSB sagas might likely provide the workflow to those web services.

From this book, you will discover the many features and characteristics of NSB, as follows:

- Sagas handle messages. A saga is started, or updated, by a message and passes it through its message handler. As messages are passed into the saga, the saga updates its sagas data from these messages through a message handler. The message logic doesn't normally end at the saga, but the saga creates a new one and passes it on to the next service. The saga may also respond back to the originating client. The saga routes messages while saving state and may be routing based on the previous state.
Sagas contain long-lived transactions (LLTs) that contain database information for the messages for relatively long periods of time. An LLT is used when conditions such as short-lived transactions are not adequate. A short-lived transaction occurs when a call to a database, or MSMQ, performs a straightforward rollback or commit. For queues, NServiceBus performs second-level retries (SLRs) to try to commit a message a number of times before performing a rollback. In LLTs, there can be multiple conditions and multiple actions that need to take place for a message to be fully completed, or else operations execute the message from the beginning.

The message is changed from one type of message into another, as one is handled by the saga, and the saga may create a new one with the same ID to pass to another service. Even though the message is different, it is a continuation of the flow of the original message that is considered a single transaction. The transaction is the accumulation of messages as they flow from one end to another end with the same message ID so that it represents the same transaction. The messages may be a different message type as it passes through different services. For instance, it may be an order message for an order service. The transaction can take seconds, days, hours, or longer, as the services take responsibility for acting upon it.

Sagas contain timeouts for timing out messages and states. Because messages can be long-lived, services are responsible for retrieving and moving them. Sagas can have timers set on messages and data so that it doesn't live forever or the timer could be part of the business logic; for example, a customer has 30 seconds to enter a pin for their debit card. Sagas contain state information. Sagas save saga data to the database based on the message's data. The saga data is initially started with a message, and it is also updated with messages that are passed in with the same identification information. When a message passes between different services, saving the state information before the next service is wise, as there might be a business requirement to revert to its original state.

NServiceBus is the C# platform of choice for those that require workflows. In sagas, high availability, high performance, monitoring, encryption, rapid deployment, and many more features can only be found in this framework when building C# solutions.
What This Book Covers

Chapter 1, *Introduction to Sagas*, discusses NServiceBus and a basic design pattern that it uses, known as sagas, which is used to save states of messages. We will discuss the benefits of sagas and what it brings to the table with regard to software design.

Chapter 2, *NServiceBus Saga Architecture*, expands on the uses of sagas for persistence, timeouts, message durability, and message handling. We will discuss various message exchange patterns through examples to include gateway and cluster managing. These are important concepts as they drive the high availability and high performance that NSB brings to the table.

Chapter 3, *The Particular Service Platform*, is an overview of the Particular website-associated tools for NServiceBus. We will discuss building sagas through the ServiceMatrix tool, which is a Visual Studio extension tool for visually designing NSB endpoints, messages, and services. The other tools that we will discuss at length as they apply to sagas are the ServicePulse to monitor the endpoint availability during production, and ServiceInsight to take a deep dive into the functionality and properties of endpoints, services, and messages as they execute.

Chapter 4, *Saga Development*, focuses on various useful constructions of sagas and message handlers. The purpose of sagas will be discussed as the discussion goes into the need for extending and coordinating transactional integrity using sagas. The chapter then morphs into a discussion of NServiceBus, using integrated pre-built WCF bridges. While some might consider it unusual to discuss WCF in a saga chapter, sagas become an intermediate for coordinating WCF and NServiceBus work. We can decouple the workflow from the frontend for interaction with backend processes through message handling. Sagas provide the means to persist the state information of the messages. This discussion will also handle other queuing sources as well to include RabbitMQ and ActiveMQ.
Chapter 5, Saga Snippets, discusses two primary saga examples, one using an e-mail, and one using the Secure File Transfer Protocol (SFTP). These samples still demonstrate the saga workflow and the use of timeouts more in depth. The saga code will be a mediator between a frontend Windows Presentation Framework (WPF) and a backend client executing either an e-mail or SFTP. Using a saga as a mediator between frontend and backend code that will interface into an external server, there will be many added benefits and features. The external server interface, such as an e-mail server or SFTP server, is usually beyond our control and is in the control of external operations or organizations, such as a bank. So, the interface into these servers is all that we have to work with, and as business, software and operational needs increase, we need a framework that is robust enough to meet these demands. Thus, we have NSB and sagas.

Chapter 6, Using NServiceBus in the Cloud, gives an introduction to the cloud with a deeper dive into the Microsoft Azure cloud. The Azure Storage containers and Service Bus will be discussed at length. An Azure Storage example will be discussed, which will work on-premise using the Azure SDK and Azure Storage Emulator. Another example will be given with NSB sagas as it works through Service Bus in the Azure cloud.
In this chapter, we will be focusing on the various useful constructions of sagas and message handlers. The purpose of sagas will be discussed when the need to extend and coordinate transactional integrity by using sagas is discussed. This chapter will then morph into a discussion of NServiceBus using integrated, pre-built WCF bridges. Some might consider it unusual to discuss WCF in a saga chapter, but sagas become an intermediary for coordinating WCF and NServiceBus workflows. We can decouple the workflow from the frontend for interaction to the backend processes through message handling. Sagas provide the means to persist the state information of the messages.

We will start with unit testing saga handlers and message handlers as we are constructing them, and how NServiceBus brings rules into testing them through Visual Studio. We will briefly discuss building our own tools and then move on to changing the transport mechanics from MSMQ into RabbitMQ. The goal is to know enough about developing sagas and message handlers so as to start building and testing our own sagas, and to have enough of an introduction to MVC, MSMQ, and EF at this point in order to start constructing and testing different business scenarios.

In this chapter, we will cover the following:

- A brief overview of MVC
- Sagas and web services
- Creating a WCF server
  - Messaging
  - Configuration tracing
Saga Development

- Creating a WCF client
  - Adding the service reference
  - Calling the reference
- Revisiting the design
- Adding the service reference
  - Calling the reference
- Adding NServiceBus to MVC
  - Message handler unit testing
  - Saga handler unit testing
- RabbitMQ for NSB
- ActiveMQ for NSB

A brief overview of ASP.NET MVC

Model-View-Controller (MVC) is the most common design pattern for implementing user interfaces. ASP.NET MVC is the Microsoft framework to implement the MVC software design pattern in ASP.NET. Developing by reusing known design patterns and frameworks that have been justified and tested by others brings a lot of reusability of known quantities into any application.

By breaking up the logic into controllers, which have the session, request, and response helper functions, while passing the models, which are the View Models which have the information which we want to present into the view, it moves most of the logic and exposure away from the browser where APIs can be exposed. We will also use Microsoft Entity Framework (EF) for many of the model objects.

Microsoft recommends using Language Integrated Query (LINQ) and EF to prevent traditional SQL injection attacks. The web page at https://msdn.microsoft.com/en-in/library/bb308959.aspx also discusses other security measures that can be done in EF. EF does not mitigate all injection attacks (such as EF injections) but, using a combination of EF and LINQ correctly, it will mitigate many common SQL injection attacks. The reasoning here is that the injection can now only occur through LINQ and the EF objects rather than any open SQL commands, thus narrowing the attack surface from a wide range of commands to an object, and through a collection that may only be accessed through a controller in MVC. There are many scripting tools running on the Internet, such as SQLNinja. Visit http://sqlninja.sourceforge.net/ to find any insecure SQL command.
However, most tools are not built to signal EF attack vectors. Please visit http://www.slideshare.net/rhelton_1/sql-injection-amp-entity-frameworks for more information on this.

There are many more reasons to use an **object-relationship mapper (ORM)**, such as EF, as we can see in http://karwin.blogspot.com/2009/01/why-should-you-use-orm.html. Some of these reasons include the following:

- **Generating boilerplate code**: EF generates objects from the SQL Server databases and tables, thus creating boilerplate code that can be used to create, update, read, and delete the fields in the tables.
- **Supporting OO**: EF objects support common object-oriented programming design and methodology that is easy as PIE (polymorphism-encapsulation-inheritance), and to include reusability.
- **Speeding development**: Generating code from a database and using it in an application can be significantly faster than creating custom code from scratch. We can see the interaction of the MVC components in the following diagram:

So, why is this discussion on MVC-EF in an NSB book? That's because most of the official NSB examples are using MVC. We will be looking at some examples that we have extended in MVC to include Microsoft's best practices of EF and WCF.

In order to view tables and queues related to tables, we use the Kendo grids. Packt publishing offers many books on Kendo, such as *Kendo UI Grid* (http://www.packtpub.com/kendo-ui-grid/book). Some of the Kendo examples are also extended from files available at http://www.codeproject.com/Articles/606682/Kendo-Grid-In-Action.
Sagas and web services

One of the many endpoints that NSB provide integration into is the Windows Communication Foundation (WCF) endpoint. Visit http://en.wikipedia.org/wiki/Windows_Communication_Foundation for more information.

WCF is part of the .NET Framework ecosystem which provides a runtime and a set of APIs for building connected, service-oriented applications. In the Simple Object Access Protocol (SOAP) binding, it makes use of Web Services Description Languages (WSDL), which defines the interface between a web service and a web service client. Based on a WSDL, the XML is created on the WSDL specification so that the client and server can exchange information irrespective of their programming languages and platforms. It is sent between the server and the client as the protocol is normally HTTP or HTTPS. There are multiple binding types; we will discuss mostly SOAP binding in this book, but for more binding types, you can visit http://msdn.microsoft.com/en-us/library/ms731092%28v=vs.110%29.aspx. For example, if using a WCF client and service that are both built-in C#, we may consider using a NetTcpBinding class as it provides high performance between .NET WCF applications. Visit http://msdn.microsoft.com/en-us/library/ms731092%28v=vs.110%29.aspx for more information.

For simplicity's sake, we are only going to work with a SOAP web service in WCF in this chapter.

WCF is a highly extensible framework and allows easy integration into NServiceBus. More information on NSB WCF sample integration can be found at https://github.com/Particular/NServiceBus/tree/develop/IntegrationTests/WcfIntegration and http://docs.particular.net/NServiceBus/how-do-i-expose-an-nservicebus-endpoint-as-a-web-wcf-service.

While Microsoft’s WCF is considered a framework for implementing pieces of service-oriented architecture (SOA) guidelines, it is not an ESB. This WCF does not contain all the features and design patterns (like sagas), persistence, or other out-of-the-box features to implement an end-to-end solution for SOA guidelines. On the other hand, using NServiceBus as an ESB with the web services of WCF brings a lot to the table that WCF and other web service frameworks do not offer.

The source code

The directory for the code is under the Payment_WCFService directory. The WCFService is used to test the client by being the WCF service.
The solution is then run in Visual Studio 2012 in Windows Server 2012 with MSMQ, DTC, RavenDB, NServiceBus Version 4.0 references, and SQL Server 2012 Express LocalDB installed.

The WCFService must first be running for the client, Payment_WCFService, to send it messages.

Creating a WCF server

We will start by creating a WCFServer project in Visual Studio, as the server needs to be running before it can communicate with the client. By adding the reference NServiceBus.Host from NuGet or Package Manager Console, several NSB default settings will be created in the App.config file for the project, an EndpointConfig.cs file with AsA_Server will be created, and the project will be set to run with an NServiceBus.Host.exe executable. We can see the creation of the files by the reference in the following screenshot:
When we add an `NServiceBus.Host` reference, through either NuGet or Package Manager, into the project, many other items will be added into the project as well. These other items are as follows:

- An `EndpointConfig.cs` file will be created in the project with the default settings to add endpoints.
- The project will be set to run as a DLL, being executed by an `NServiceBus.Host.exe` executable when run from the debugger.
- Several default settings for creating a generic `AsA_Server` endpoint will be added to the `App.config` file.

We can see from the following screenshot that when we add the `NServiceBus.Host` reference, this sets the project to run as a DLL using the `NServiceBus.Host.exe` executable:

We will use NSB WCF Integration to save a lot of work on our part. In order to use this integration, we will need a few pieces related to both NSB and WCF. These are as follows:

- **A web service**: This is used with the `NServiceBus.WcfService<TRequest, TResponse>` class where we define a request and response on the web service. We will show here that the service called `PayService` will receive the `PaymentMessage` message as the request and respond with `ErrorCodes`. This is shown in the following line of code:

  ```csharp
  public class PayService : WcfService<PaymentMessage, ErrorCodes>
  ```

- **A message handler**: This is required to handle the message from the web service and to process it in the integration. We will call this handler `PayHandler`.

- **The request and response message structures**: This is required for the `PaymentMessages` project.

- **The configuration for NSB with WCF**: This is required in the `App.config` file.
Using NSB with WCF integration is similar but different from using straight WCF. In WCF, there are four basic steps that can be extended. Visit http://www.c-sharpcorner.com/UploadFile/dhananjaycoder/four-steps-to-create-first-wcf-service-for-beginners/ for more information.

These steps to create a WCF server are as follows:

1. Create a service contract. This defines the available functions between the WCF client and the WCF server through interfaces.
2. Expose endpoints with metadata—through either App.config or Web.config. We expose the endpoints through the configuration to include how the exchange of data will occur.
3. Implement the service. That is, we add functionality and data objects to the interfaces.
4. Consume the service. Or, in other words, expose the service to be imported into a WCF client.

In using NSB with NHibernate, NSB takes care of the mapping of NHibernate with the messages, endpoints, and services. In WCF integration, NSB also takes care of many of the service contracts with the use of the message handler and the message format.

We can observe these features pieces in the following screenshot and we will discuss them further:

We will add the messages next.
Adding messages

We will now create a PaymentMessages project inside the solution, as shown in the following screenshot:

We will be using the PaymentMessage request message, which the request needs to be NServiceBus IMessage Interface having two classes, a Guid method for a unique ID per message, and a PaymentReq class:

```csharp
public class PaymentMessage : IMessage
{
    public Guid EventId { get; set; }
    public PaymentReq paymentReq { get; set; }
}
```
The `PaymentReq` class will have many fields which are necessary for a normal payment to a bank, including items like bank routing number, bank account number and many more items to identify a payment (as shown in the following diagram):

![PaymentReq Class Diagram](image)

The `ErrorCodes` are just an `Enum` class that either returns `None` (for no error) or `Fail`, as shown in the following screenshot:

![ErrorCodes Enum](image)

Now that we have a message, and we also have a WCF service (called `PayService`), let's start a message handler to handle the message by using the message to receive it from the client.

Just to give an overview, `PayService` will get a message from the web service client and, in turn, place the message in the MSMQ by default (that is, a queue called `wcfservice`) for the message handler to process it, and then respond back to the web service.
Adding the message handler

We will now create a message handler to handle the message (PaymentMessage). Here, we are just printing out the EventId instance to the console window. Further, we may add subscription processing for handling the payment as a bank may handle it. The service will return an ErrorCode with the value of None. We are conducting very simplistic tests at the moment. This is the WCF service project under BasicWCF1:

```csharp
namespace WCFServer.Handlers
{
    public class PayHandlers : IHandleMessages<PaymentMessage>
    {
        private readonly IBus bus;
        public PayHandlers(IBus bus)
        {
            this.bus = bus;
        }
        public void Handle(PaymentMessage message)
        {
            Console.WriteLine("======================================
            =================================");
            Console.WriteLine(message.EventId);
            Console.WriteLine("===================================
            =================================");
            bus.Return((int)ErrorCodes.None);
        }
    }
}
```

We will then create the WCF client for the web service call. The client will simply define the messages to send to the web service for handling the request and response. The PayMessage instance will be the request going from the client to the service. The reply from the service will be the ErrorCode. We will perform these steps after we explore more about the configuration and tracing of the WCF service.
Adding the configuration

The task of configuring the App.config file for the web service still remains. This step is similar to exposing the endpoint with metadata that we mentioned earlier.

The App.config file will define several characteristics of the web service, such as the listening port and URL, the security of the service, and the binding type for the service. The binding type defines the communication mechanism of the endpoint, be it basic HTTP, MSMQ, or any other. A list, as well as more information on WCF binding, can be found at http://msdn.microsoft.com/en-us/library/ms730879(v=vs.110).aspx.

We could configure the App.config file by manually editing it, or by using the WCF Service Configuration Editor that comes as part of Visual Studio.

More information on the Configuration Editor can be found here:


We will now open the App.config file through the Configuration Editor to do the following:

- Establish a server URL, including port number, to be available for the client WCF as the WCF endpoint
- Establish the binding parameters that will expose the WCF service endpoints in a variety of different ways
- Set up tracing and logging to review the transmissions and services as they happen
We start by opening the project's App.config file and setting up the URL and ports that the web service will be listening with to start configuring the endpoint—as shown in the following screenshot:
Next we set the binding types. We will be using the WCF server endpoint to be set at a particular port, as shown in the following example:

We used mex binding here, which is metadata exchange binding. This is useful if data needs to change over time and the update of client information through discovery. In other words, we are adding metadata to the endpoint to expose the metadata of the service so that the WCF client can easily create a proxy.
Please visit http://msdn.microsoft.com/en-us/library/ms731734(v=vs.110).aspx to review the WCF configuration schema. We can edit the binding through the service endpoint as shown here:

Adding tracing
There may be some debate as to whether this section is required or not, as this section is specific to WCF but not related to sagas or NSB. However, if your WCF communication is not working between a WCF client and a WCF server, the issue could be related to the message structure, the network, the WSDL, or any other factor. And in order to trace the issues in WCF, the tracing feature needs to be enabled and set up. This also allows us to see the reaction of the communication. However, if your WCF is functional and no troubleshooting is required, feel free to skip this section.
There are service tracing utilities within WCF to view graphs, messages, network calls, exceptions, and activities of WCF web services and web clients. Please refer to http://msdn.microsoft.com/en-us/library/ms732023(v=vs.110).aspx for more information on Microsoft Service Trace Viewer.

We can capture the messages between a client and a server by setting the diagnostic sections (given as `<diagnostics/>`) in the `App.config` file to capture these messages, and also by setting the listeners and sources, which will be associated with the libraries, to log events in the `App.config` file, as shown here:

```xml
<system.serviceModel>
  <diagnostics>
    <messageLogging logEntireMessage="true" logMalformedMessages="false" logMessagesAtServiceLevel="true" logMessagesAtTransportLevel="true" maxMessagesToLog="30000" maxSizeOfMessageToLog="30000" />
    <endToEndTracing activityTracing="true" messageFlowTracing="true" />
  </diagnostics>
  <behaviors>
    <serviceBehaviors>
      <behavior name="Default">
        <serviceMetadata httpGetEnabled="true" />
        <serviceDebug includeExceptionDetailInFaults="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service name="WCFServer.WebServices.PayService" behaviorConfiguration="Default">  
      <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
    </service>
    <baseAddresses>
      <add baseAddress="http://localhost:9009/services/paymessage" />
    </baseAddresses>
  </services>
</system.serviceModel>
```
The following screenshot represents how we can configure message logging and what kind of trace listener we are going to use to write the logs:

We can view the messages being sent between the client and the server by opening the *svclog* file in Service Trace Viewer. As shown in the following screenshot, we can see the SOAP message, including the header and the body, which is what the WCF server was receiving.
Not only can we see the SOAP information, but the HTTP information as well. By adding a System.Net listener, we can trace the network socket calls for opening and closing sockets as they happen, as well as the sessions in between. For additional information on how to configure network tracing for WCF, please refer to the following links:


So, why discuss web service tracing? In many web services, such as those connecting to banks to process credit cards, we may only see one side of the WCF service. Sometimes, the other side may end up being a Java web service or other types of services that a third party develops; it may even be using CICS legacy code. Consequently, we may have very little control over some of the web services that we will be integrating into. And sometimes, there is very little documentation as well. So, tracing becomes a necessity for many web services to debug and log functionality as messages go in between systems.
Viewing the web service

Running the web service in Visual Studio 2012, we can view the web service through the browser at http://localhost:9009/services/paymessage. It will provide some simple instructions for the WCF client, as we can see in the following screenshot:

Considerations when deploying

Please note that for most production systems, it is always recommended to install digital certificates when using web services to encrypt the pipeline and verify the identity of the client and the server.

For a real-life production system, all the communication between the WCF client and WCF server must be secure using SSL, and you should look into your web services using WS-Security-based applications.
Creating a WCF client

In this section, we will create an MVCApp solution that will be under the BasicPayClient directory. It will contain several projects as follows:

- **MVCApp**: This contains MVCs, Kendo grids, and the WCF client for a browser user interface to send messages to the WCF service. It will read five XML files to load as messages.
- **MyMessages**: This contains IMessages of NServiceBus for the building of messages.
- **WriteXMLFiles**: This is a utility to write five XML files to a `C:\temp\` directory for the MVCApp project to load. This application saves messages in the form of XML files, which in turn are loaded through MVCApp to be sent from the WCF client to the WCF service. These are for testing purposes only, but using files in the form of messages makes it quick to change the data for various tests in the communications and endpoints. The messages are read from a `C:\temp\` directory as five XML files saved in a format that works with the WCF messaging service. The files can be created in the `C:\temp\` directory by running the `C:\temp WriteXMLFiles` project. These files are simply test messages and are saved to a disk so that they can be modified and tested easily.

Next, we need to design the client. We will have an MVC ASP.NET web interface that reads the XML files and displays them. We can select an individual message and send it to the WCF service as a request to get a response.

As a further exercise in this chapter, we will add a saga and a message in-between the MVC application and the WCF client to show how the workflow will assist us.

We already have instructions in the display of the WCF service, as given here:

**To test this service, you will need to create a client and use it to call the service.**

You can do this using the svcutil.exe tool from the command line with the following syntax:

```
svcutil.exe http://localhost:9009/services/paymessage?wsdl
```

You can also access the service description as a single file:

```
http://localhost:9009/services/paymessage?singlewsdl
```
In order to verify if the service is up and running, just browse to your service at the following link:
http://localhost:9009/services/paymessage

Adding the service reference

First, we will run the WPayment WCF service from Visual Studio. We can check to see whether it is running by just seeing whether we can view the web service in a browser. This is a self-hosted solution since we can use NSB to deploy the service as it is contained in an NSB package. However, WCF can be IIS-hosted as well. To see a comparison of some of the Windows hosted solutions, visit http://msdn.microsoft.com/en-us/library/ms730158.aspx.

```
c#
class Test
{
    static void Main()
    {
        WcfServiceOf_PayMessage_ErrorCodesClient client = new WcfServiceOf_PayMessage_ErrorCodesClient();
        // Use the 'client' variable to call operations on the service.
        // Always close the client.
        client.Close();
    }
}
```
We will now add a service reference from the available WSDL to the web service client. Visit http://msdn.microsoft.com/en-us/library/bb628652.aspx for more information. The following screenshot shows how the Add Service Reference window appears:

![Add Service Reference Window](image_url)
We will use the advanced settings to reuse the MyMessages packages of messages that we are using in this chapter's projects. The following image shows how your screen would look:

![Image of MVCApp.ServiceReference1 - Service Reference Settings]

### Calling the service reference

We will create an index.html file on the Kendo grid which will be a link on View which, when clicked, will load the XML files. When a particular XML PayMessage instance is selected to be sent to the WCF service, it will call the service reference, which was imported as ServiceReference1, to create a client and pass the selected PayMessage into it to be processed and sent to the WCF service. We will create this code in the MVC controller function, SendWCF, and pass it the ID of the message that we are sending to the WCF service. The code is shown in the following screenshot:
This will be the Kendo grid in the browser that offers a selection to be sent to the WCF service. Upon browsing the page, the grid will populate with the following data and an option to send the message to the WCF service, as shown in the following table:

<table>
<thead>
<tr>
<th>Event Id</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b265223-6c9e-4789-a6df-69dc9f644ad7</td>
<td>Call Payment Service</td>
</tr>
<tr>
<td>3721ba5d-4733-4d98-a5e2-8e8afa3e61f4</td>
<td>Call Payment Service</td>
</tr>
<tr>
<td>fac188cc-4b2e-436c-b989-db88c05d6b1fa</td>
<td>Call Payment Service</td>
</tr>
<tr>
<td>9bf180fa-fbf4-4b2b-8f8c-cca73a4e2cab</td>
<td>Call Payment Service</td>
</tr>
<tr>
<td>ee256f7-6d42-4314-b95e-48256d294437</td>
<td>Call Payment Service</td>
</tr>
</tbody>
</table>

The Kendo grid scripting inside the SendWCFPay.cshtml file will look like the following:

```html
<h2>Send Payment Message to WCF</h2>
<script type="text/javascript">
$(document).ready(function () {
    var modelData = @Html.Raw(Json.Encode(Model))
    $("#grid").kendoGrid({
        pageable: true
    , sortable: true
    , selectable: true
```
If the PayMessage was processed successfully, it will have no errors when we view the details of that request, as shown here:

<table>
<thead>
<tr>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Event ID</td>
</tr>
<tr>
<td>8b265223-dc9e-4789-a6df-69d19f644ad7</td>
</tr>
<tr>
<td>Error</td>
</tr>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

**Revisiting the design**

In the code, as an example, we have built a Payment WCF Service solution as a WCF Integration solution into NSB. We have also built an MVC frontend to the WCF client solution that we can use for testing the Payment WCF Service.

The frontend is a basic MVC application with some basic WCF client interfaces for a PayMessage. PayMessage has a GUID, an address, and basic information for payment.
For simplicity, the MVC controller is just reading XML files (that were created in a folder in `C:\temp\`) to be displayed in the frontend and selected to be sent to the Payment WCF service. Using XML files in this method is not recommended for production, as normally these files would be filled in by a frontend payment interface to the customer for the customer information. We chose this method to solidify the interfaces and test them. And designing customer interfaces to populate the messages was not part of this effort. Using XML files in this type of testing can be advantageous, as we could extend this example to send hundreds of messages, both in order to check performance as well as to populate individual messages through a customer interface without relying on the frontend.

What we basically have is a very common design, where we have an MVC frontend calling a WCF client to interact with a WCF server. This design is also common to see at some ordering sites. For instance, at, let's say a pizza site, is a pop-up saying order being sent, do not refresh. Because there is no decoupling to the backend workflow in this example as well, if this were to become a solution without decoupling through NSB, we may likely have to add payment being sent, do not refresh. Many things can go wrong in the event of during the need for a page refresh, just because a customer may not be refreshing their page, it does not mean that something else isn't doing a page refresh.

Our current architecture is as follows:

![Diagram of MVC, WCF Client (encapsulated in MVC), and WCF Server interactions.](image-url)
Let's build some saga processing into the preceding solution to decouple the MVC frontend from the WCF client. NSB will act as a mediator for the interaction between these two components pieces.

Why a saga? It routes messages, performs timeouts, and persists a state.

The preceding screenshot represents the new workflow using the NServiceBus saga. It seems a bit overwhelming, but NServiceBus handles all the complexity behind the scenes. For the controller, it just fires and forgets. If it needs to know the state of the message, it just executes an EF query on the DB. But it still sends the message off, and allows NSB to handle it.

**The source code**

The directory for the code is under the SagaPaymentClient directory.

The MVCApp – WCF is used to send WCF messages as a client using a saga.

The solution was built using Visual Studio 2012 in Windows Server 2012, and also tested with VS 2012 running in Windows 8.1, with MSMQ, DTC, RavenDB, NServiceBus Version 4.0 references, and SQL Server 2012 Express LocalDB installed.

Here's what the new project will look like with the saga:

```
[ 118 ]
```
The interaction will look like the following:

Wow! It looks like a lot, but most of the diagram is similar to the previous MVC application, and we are familiar with most of the orchestration represented in the preceding diagram. However, all the new pieces of orchestration are handled by the NSB saga. The WCF client and server remain the same, but the WCF client is encapsulated in a message handler that receives and responds to the saga. Before refactoring with NSB, the WCF client updates the MVC application with the status of the WCF by updating a database about the fact that the message was processed successfully from the WCF server. Now, after NSB, the MVC application has a message handler. It will still update the database, but now the MVC application can be event driven from the saga as well, whenever the user needs to be notified immediately. Has the saga refactoring process increased the quality of software? If a message is interrupted, the power is shut off, the user refreshes their page, and a thousand other things could happen. However, the payment will not be lost. Also, a page refresh will not affect the payment either.

As part of this refactoring into NSB, the saga handles all the complexity around the service calls, and we have a nice separation of concerns between the controller and the orchestration.
Adding NServiceBus to MVC

We will next be extending our BasicPaymentClient folder and projects into a SagaPaymentClient folder and projects with the addition of the bus. The differences will be as follows:

- We will refactor the WCF client out of the MVC controller and move it into a new message handler.
- We will create a database that keeps track of the state of the message. The saga data will be saved in the nservicebus table. Ensure that it is created when running the saga code.
- We will create a new message handler and put the WCF client code in it. This code from the WCF client to the WCF server will be kept separate from the frontend code as far as possible, to keep the PayMessage class completely separate from the frontend.
- We will create a new message handler in the MVC that will update the DB with the message state as it receives the new state from the endpoints. We will look at the code for the message and saga handlers as we test them in the next section.
- NServiceBus.Testing offers testing by sending messages through message handlers and sagas. This includes anything that a message handler and saga can do, including header manipulation and dependency injection. Refer to http://docs.particular.net/NServiceBus/unit-testing for some basic examples. For the source code of NServiceBus.Testing, go to https://github.com/Particular/NServiceBus/tree/develop/src/.

The very basis of starting unit testing is to create a unit testing project in Visual Studio by adding a new unit testing project to an existing solution. Visit http://msdn.microsoft.com/en-us/library/hh598957.aspx and hyperlink it. The Add New Project window is as shown here:
We will add the NServiceBus.Testing project from NuGet (http://www.nuget.org/packages/NServiceBus.Testing/). Your screen should look similar to what is shown in the following screenshot:
Saga Development

In the NServiceBus.Testing projects, all the tests are initialized with the Test.Initialize() method. A test will originate with the Test.Handler<HandlerName>() or TestSaga<SagaName>(). This is shown here:

```csharp
using NServiceBus;
using NServiceBus.Testing;

namespace UnitTestHandlers
{
    [TestClass]
    public class UnitTestHandler
    {
        ///
        /// Test the message handler for MyWCFClient
        /// This will call the WCF Service for a completion
        ///
        [TestMethod]
        public void Run()
        {
            Test.Initialize();

            Test.Handler<MyHandler>()
                .ExpectReply<ResponseMessage>(m => m.String == "hello")
                .OnMessage<RequestMessage>(m => m.String = "hello");
        }
    }
}
```

When a test is built, we can run it or debug it. The test indicators will tell us if anything failed or succeeded. As part of following test-driven development (TDD), we must follow the AAA rule. These rules incorporate the Arrange-Act-Assert (AAA) pattern to verify whether a test fails or passes. Visit http://c2.com/cgi/wiki?ArrangeActAssert for more information.

We can also put in rules and assertions where, if the correct response does not happen, it will fail the test. This is a great feature of Visual Studio, and there are many samples at http://msdn.microsoft.com/en-us/library/ms243176.aspx and http://www.visualstudio.com/en-us/get-started/create-and-run-unit-tests-vs.aspx. There are extensions available as well at http://www.codeproject.com/Articles/22358/Visual-Studio-Unit-Testing-Extensions.
Message handler unit testing

The message handler code will be in the unit test itself. From our project, UnitTestHandlers, in which we have various unit tests, we will walk through EventMessageHandler. EventMessageHandler receives a SendCommand object from the MVCApp, via the saga, as shown here:

We will proceed with creating a UnitTestHandler2.cs file, and then add the header information and [TestMethod]. This will be under SagaPaymentClient in the UnitTestHandlers project.
After the base of the file is created, we will create a normal message, `SendCommand`, with a GUID and state where the message should be at before reaching the message handler, called `command`. The code is as follows:

```csharp
namespace UnitTestHandlers
{
    [TestClass]
    public class UnitTestHandler2
    {
        ///
        /// Test the message handler for MWAFCClient
        /// This will call the WCF Service for a completion
        ///
        [TestMethod]
        public void Run()
        {
            Test.Initialize();
            //
            // Create a Command message
            // used to look up an XML Message file
            // on disk, send to WCF Server
            //
            //
            SendCommand command = new SendCommand();
            command.RequestId = new Guid("4b205223-dcb8-4709-aeef-98d19f50d0ed");
            command.state = MyMessages.MessageParts.StateCodes.SendMyWCFClient;

            // The Test code
            Test.HandleEventMessageHandler("
                .ExpectReply<ResponseCommands>(m => m.state == MyMessages.MessageParts.StateCodes.CompleteMyWCFClient)
                .OnMessage<SendCommands>(command);
            
```
This allows us to design and debug the handler functionality in the unit test code through TDD. There are many rules that can be used when testing the handler or saga. For instance, `ExpectNotReply` is used to expect that the handler does not reply with a specific message.

To get information on what is available in `NServiceBus.Testing`, we can execute the following steps:

1. Try to enter something and hover over IntelliSense.
2. Read the documentation at http://www.nudoq.org/#!/Packages/NServiceBus.Testing/Handler%28T%29. The following screenshot shows the documentation:

3. Read the code in GitHub at https://github.com/Particular/NServiceBus/blob/develop/src/NServiceBus.Testing/Handler.cs:

So, there are many possibilities for testing to test the code. For the message handler, it will get the command with the GUID and state, read the XML files to get a matching message, and send it to the WCF service, which will respond back to the saga. The saga keeps track of the message routing and states, and will respond to the MVCApp. The MVCApp will consequently update its state in the table. There could normally be multiple Views that could read the state – maybe an admin utility to check on the state of the messages, the **customer service rep (CSR)** talking to the customer, telling them if the payment has been processed, or a confirmation form or e-mail to the customer telling them that the payment succeeded, or many other scenarios.
Besides a couple of functions to read the XML file for the message, which is just used for testing, there could be a number of scenarios added, but the majority of the code to do this is simply the following. Simple enough!

/****

* The message handler
* Matches an XML message GUID from a file and the command sent
* to it from MVC via the Saga
* If found, sends it to the WCF Server and responds
* with the state of what happened.
* The WCF Service must be running to complete.
*
* ****/

public class EventMessageHandler : IHandleMessages<SendCommand>
{
    public IBus Bus { get; set; }
    public void Handle(SendCommand message)
    {
        ServiceReference1.WcfServiceOf_PayMessage_ErrorCodesClient client1 =
            new ServiceReference1.WcfServiceOf_PayMessage_ErrorCodesClient();
        // Create the response message
        ResponseCommand command = new ResponseCommand();
        command.RequestId = message.RequestId;
        /****
        * Find a match from the GUID
        * ****/
        List<PayMessage> list =
            EventMessageHandler.GetMessages();
        PayMessage payMessage = null;
        foreach (var temp_message in list)
        {
            if (message.RequestId == temp_message.EventId)
            {
                payMessage = temp_message;
            }
        }
        ErrorCodes returnCode =
            client1.Process(payMessage);
    }
}
if (returnCode == ErrorCodes.None)
{
    command.state =
        StateCodes.CompleteMyWCFClient;
}
else
{
    command.state =
        StateCodes.MyWCFClientFail;
}

Bus.Reply(command);
Console.WriteLine("Success");

What happens if the XML file does not exist? The following is the code used then:

// if no XML, just fail
if (payMessage == null)
{
    command.state = StateCodes.MyWCFClientFailXML;
    Bus.Reply(command);
    Console.WriteLine("No XML Found");
}
else
{
    ... normal path ...
}

After testing this code, we could use the tested code to create a class into a new project, barring the unit testing, and start using it as a message handler. It saves time by developing the code in a unit test and putting the tested product into the application's project. The unit test project also serves as a backup for knowing what it looked like during a good test.

**Saga handler unit testing**

Let's start testing saga code from the previous section in the message handler:

// The Test code
Test.Handler<EventMessageHandler>()
    .ExpectReply<ResponseCommand>(m => m.state ==
        CompleteMyWCFClient)
    .OnMessage<SendCommand>(command);
As we can see, the NServiceBus testing API makes use of a Fluent API specification style testing pattern as opposed to the more traditional assertion style that most people would normally use that are part of NUnit or other xUnit type frameworks.

We will now start testing the saga code from the UnitTestHandlers project and the UnitTestSaga2.cs file.

One thing to note is that if a saga entity object is deleted in different function calls, with the MarkAsComplete() method, these should be tested separately. This is because once we delete the object, we cannot delete it again. For example, in our tests, we will use the following:

```csharp
[TestMethod]
public void Run()
{
    Test.Initialize();

    /**
     * State sent to Saga
     */
    var command = new SendCommand();
    command.RequestId = Guid("8b265223-dc9e-4789-a6df-69d19f644ad7");
    command.state = MyMessages.MessageParts.StateCodes.SendMyWCFClient;

    /**
     * Response from WCF and to MVCApp
     */
    var resp = new ResponseCommand();
    resp.RequestId = Guid("8b265223-dc9e-4789-a6df-69d19f644ad7");
    resp.state = MyMessages.MessageParts.StateCodes.CompleteMyWCFClient;

    TestSaga<MyTestSaga>()
        .ExpectReplyToOriginator<ResponseCommand>()
        .ExpectSend<SendCommand>()
        .When(s => s.Handle(command))
        .ExpectReplyToOriginator<ResponseCommand>()
        .When(s => s.Handle(resp))
        .AssertSagaCompletionIs(true);

    TestSaga<MyTestSaga>()
        .ExpectReplyToOriginator<ResponseCommand>()
        .ExpectSend<SendCommand>()
        .ExpectTimeoutToBeSetInSendCommand({state, span} => span == TimeSpan.FromHours(3))
        .When(s => s.Handle(command))
        .ExpectReplyToOriginator<ResponseCommand>()
        .WhenSagaTimesOut()
        .AssertSagaCompletionIs(true);

    }
```
In this snippet, we are testing the message handler with two separate conditions. The first test case is the normal condition of a saga start where we are testing the `IHandleMessages<ResponseCommand>` message handler.

The second test case is the timeout condition where we are testing the `IHandleTimeouts<SendCommand>` handler. These two test cases were used in the same file as they reused some of the same pieces.

The saga handler itself will act as a mediator between MVCApp and the WCF client. This is needed to act as a timeout after three hours in case there is no response from the WCF service.

### Integration tests with MVC

Normally, when putting a bus in MVC, we wish to create the bus only once, and then reuse it over and over again from different controllers to send messages. In order to do this, we will be putting the code in the `Global.asax.cs` file under `.\MVCApp – WCF\MVCAp`.

However, we will perform integration tests to ensure that all the pieces are working. Performing tests outside of the MVC application itself can assist in isolating databases and endpoints that are not deployed and are needed for the application to process. The integration test can be found in the `MVCToNSBTests.cs` file under `.\MVCApp – WCF\IntegrationTests\`. There are two items that we have added to the MVC application: an endpoint to send messages to the saga and an MVCApp database with Entity Framework connections to store the payment messages in the PayMessage table.

In order to create this table, the MVCApp database must be created; in this case, `.\SQLExpress`. After the database is created, we can create the table structure by clicking on the **Generate Database from Model...** option, and then executing the SQL script that was created from this execution, as shown here:
Now that the database portion has been created, we need to create the mysaga endpoint in MSMQ. This is done by simply executing the MySaga solution as follows:
Saga Development

We can view in the **MSMQ Commander** window that the endpoint was created, as shown here:

If the tests run successfully, then the **mysaga** endpoint is present and working. We can see the test program working as follows:

```java
@TestMethod
public void TestMethod()
{
    // Use a single queue
   Configure.Scenarios => MssSingleBrokenQueue();
   _startableBus = Configure.With()
   .DefaultBuilder()
   .AllDefaultEncryptionService()
   .UseTransport(pdnp())
   .UnicastBus()
   .CreateBus();

   Configure.Instance.ForInstallationKnowWindows().Install();

   /**
   * State sent to Saga
   *===
   * SendCommand command = new SendCommand();
   * command.RequestId = new Guid("02090221-dc9e-4789-a5ef-90913964440f");
   * command.state = PaymentMessages.MessageParts.StateCodes.SendBySaga;
   * _bus = _startableBus.start();
   * _bus.Send(command);
   *
   */
}
```
We can see the test program in **Test Explorer** working as follows:

```csharp
[TestMethod]
public void TestMethod1()
{
    // Use a single queue
    Configure.ScaleOut(s => s.UseSingleBrokerQueue());
    _startableBus = Configure.With()
        .DefaultBuilder()
        .RijndaelEncryptionService()
        .UseTransportPlumber()
        .UseBus()
        .CreateBus();

    Configure.Instance.ForInstallationOnWindows().Install();

    /**
     * State sent to Saga
     * @param key
     * @return
     */
    SendCommand command = new SendCommand();
    command.RequestId = new Guid("6b205223-dcbe-4789-a6df-09d19f04ad42");

    _bus = _startableBus.Start();
    _bus.Send(command);
}
```

Now, we can copy this code into the `Application_Start()` function in the `Global.asax` file to be called at the startup of the MVC and use it as the bus for the MVC.

We create a controller to use this bus to send the command by ID; it can be selected from the MVC view as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Credit ID</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8025223-4d9a-4789-a6df-09d19f04ad42</td>
<td>CallPaymentService</td>
</tr>
<tr>
<td>2</td>
<td>1234567-8901-4234-c2b3-ad6b7c54d567</td>
<td>CallPaymentService</td>
</tr>
<tr>
<td>3</td>
<td>901234-5678-9012-3456-7890-1234567890</td>
<td>CallPaymentService</td>
</tr>
<tr>
<td>4</td>
<td>567890-1234-5678-9012-3456-7890-1234567890</td>
<td>CallPaymentService</td>
</tr>
<tr>
<td>5</td>
<td>678901-2345-6789-0123-4567-8901-2345678901</td>
<td>CallPaymentService</td>
</tr>
</tbody>
</table>
We will send the message to the saga by the methods in the MVC controller. The controller will be selected by ID, and a lookup in the table for the correct message will be made from the PayMessage table. The lookup will be done in an Entity Framework connecting the data access layer (DAL) of the code. The sending to the saga appears as follows:

```csharp
public ActionResult SendSaga(int id)
{
    var user = new XElement().GetPayments().Where(p => p.Id == id).FirstOrDefault();
    var message = new XElement().GetMessages().Where(p => p.EventId == user.EventId).FirstOrDefault();

    // Create the send
    SendCommand command = new SendCommand();
    command.RequestId = message.EventId;
    command.state = StateCodes.SendMySaga;

    // Change the state in the EF DAL
    MVCAppDAL dal = new MVCAppDAL();
    dal.changeState(command);

    // Send the command on the Bus
    MvcApplication.Bus.Send(command);
}
```

The DAL will read the database table called PayMessage, which is configured as an entity model object from the Model1.edmx file that does the ORM mapping. The connection string pointing towards the table and database is defined in the Web.config file. The PayMessageModel.edmx file was generated from the database to provide the mapping to the objects. So first, we need to build a database table to contain the PayMessage table that looks like the following with GUIDs and state. This is used to update the state from the messages for the MVCApp.
The database table, <PayMessage>, has the following properties, as shown here:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Allow Nulls</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>EventId</td>
<td>uniqueidentifier</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>varchar(50)</td>
<td>✓</td>
</tr>
<tr>
<td>state</td>
<td>varchar(50)</td>
<td>✓</td>
</tr>
</tbody>
</table>

### RabbitMQ for NSB

RabbitMQ is a cross-platform messaging framework, like MSMQ, that can run on both Linux and Windows operating systems.

RabbitMQ has many more features than MSMQ, such as routing, virtual hosts, and a powerful admin toolset. It can run on Linux and scales out very well. For large enterprise systems (especially in heterogeneous network environments), you should really consider RabbitMQ. RabbitMQ can easily be changed for the MSMQ queuing system in the NSB configuration.

For administrating RabbitMQ, a web admin interface can easily be installed. However, if the command-line interface is required, then the Python language will need to be installed.

To revisit some of the references in the earlier chapters, take a look at the following:

- **Local host management site**: This can be found at http://localhost15672/#/.
- **Documentation**: This can be found at http://www.rabbitmq.com/documentation.html.
- **Windows installation**: This can be found at https://www.rabbitmq.com/install-windows.html.
- **NServiceBus samples**: These can be found at https://github.com/Particular/NServiceBus.RabbitMQ.Samples.
- **Development tools site for RabbitMQ**: This can be found at http://www.rabbitmq.com/devtools.html.
- **NServiceBus.RabbitMQ source code**: This can be found at https://github.com/Particular/NServiceBus.RabbitMQ.
- **NServiceBus RabbitMQ hands-on lab**: This can be found at http://particular.net/HandsOnLabs.
After installing RabbitMQ, we can set the management plugin using the `rabbitmq-plugins` to enable `rabbitmq_management`. This is to view the queues using the web management `http://localhost:15672/#/queues`, as shown in the following screenshot:

![RabbitMQ Management Screenshot](image)

We can get a list of queues by entering `rabbitmqctl list_queues` in the RabbitMQ command prompt, as shown here:

```
C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.2.3\bin> rabbitmqctl list_queues
Listing queues ...
  MyAudits 2
  MyErrors 0
  MySaga 0
  MySaga.Retransmits 0
  MySaga.Retransmits:Dispatcher 0
  QueueStore.Sales 0
  QueueStore.Sales.Retransmits 0
  QueueStore.Sales.Retransmits:Dispatcher 0
  audit 2
  error 0
...
done.
C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.2.3\bin>
```
We can delete all the queues by running the following:

```
rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl start_app
```

**The source code**

The directory for the code is under the RabbitMQ directory. There are two solutions, which are as follows:

- **MVCApp – WCF**: This is used to send WCF messages as a client using a saga. But instead of MSMQ, RabbitMQ is used for queuing.
- **WCFService**: This is used as the WCF service.

The solution was built in VS 2012 in several operating systems, including Windows Server 2012, Windows Server 2008, and Windows 8.1, with MSMQ, DTC, RavenDB, NServiceBus Version 4.0 references, and SQL Server 2012 Express LocalDB installed.

**Changing the endpoints**

There are going to be subtle differences in setting up the endpoint configurations. These are the three basic steps:

1. Add the `NServiceBus.RabbitMQ` reference.
2. Change the NServiceBus transport mechanism from `<MSMQ>` to `<RabbitMQ>`.
3. Set the RabbitMQ transport configuration in the `App.config` file.

In the `MySaga` project, we will be making changes to the `App.config` and `EndpointConfig.cs` files.
The NServiceBus.RabbitMQ package will have to be installed into each project to support RabbitMQ. It will be added via NuGet. Go to http://www.nuget.org/packages/NServiceBus.RabbitMQ/ for more information.

The .UseTransport<> method that is defaulted to MSMQ will have to be switched to RabbitMQ, as shown here:

```csharp
Configure.With()
    .DefaultBuilder() // Autofac Default Container
    .UseTransport<NServiceBus.RabbitMQ>()
    .InMemorySubscriptionStorage()
    .UseNHibernateSagaPersister()
    .UseNHibernateTimeoutPersister()
    .UnicastBus(); // Create the default unicast Bus
```

We will have to set the transport mechanism to the local host as this is where the RabbitMQ service is residing. This is shown in the screenshot here:

Beyond the changes discussed, there are only very little changes needed to move to different queuing systems. The saga and message handlers work in the same way; we are only changing the endpoint transportation mechanisms.

We can see that queues were created and run from this example in a previous screen from the queues screenshot, which is shown at http://localhost:15672/#/queues.
ActiveMQ in NSB

Apache Active Message Queue (ActiveMQ) is a JAVA open source framework from the Apache foundation based on the Java Message Service (JMS). Visit [http://en.wikipedia.org/wiki/Apache_ActiveMQ](http://en.wikipedia.org/wiki/Apache_ActiveMQ) and [https://activemq.apache.org](https://activemq.apache.org) for more information. It will run on a machine, be it Windows or Linux, in a Java Runtime Environment (JRE). JAVA has to be operational on the machine and have the environment path for JAVA_HOME configured to point at the root folder of the JRE. The installation instructions for ActiveMQ can be found at [https://activemq.apache.org/getting-started#GettingStarted%20-%20Download](https://activemq.apache.org/getting-started#GettingStarted%20-%20Download).

The source code

In this section, we will be using the ActiveMQ solution. This solution is similar to RabbitMQ, except ActiveMQ is used instead of RabbitMQ.

There will be three basic steps:

1. Add the NServiceBusActiveMQ reference.
2. Change the NServiceBus transport mechanism from `<MSMQ>` to `<ActiveMQ>`.
3. Set the ActiveMQ transport configuration in the `App.config` file.

Once downloaded on the Windows OS, we unzipped the Window's binary files into the `c:\activemq\` directory. Running the `activemq.bat` batch file from the command prompt, in `c:\activemq\bin\`, will display a series of commands to show that the ActiveMQ is running. This is shown in the following screenshot:
An alternative is to install ActiveMQ as a Windows service in which installation
scripts exist for both Win32 and Win64 machines. For a 64-bit Windows Server, we
can use InstallService.bat in c:\activemq\bin\win64\.

Ensure that RabbitMQ is not running as a Windows Service in the background, as
they utilize the same network ports. Also, Microsoft ServiceBus for Windows Servers
will share the same JMX ports as well, which will be port 5672. ActiveMQ's default
port is 61616. Checking the ports can be done with Microsoft's TcpView, which can be

To ensure that ActiveMQ is running, you may access the admin console in the
browser by using http://localhost:8161/admin. The default user ID and
password are admin and admin respectively. Please visit http://activemq.apache.
org/getting-started.html to have a look at the documentation. When accessing
the admin console, you should get something that looks like the screenshot here:
To use ActiveMQ for NServiceBus in Visual Studio projects, the NuGet version of NServiceBus.ActiveMQ has to be installed. Go to https://www.nuget.org/packages/NServiceBus.ActiveMQ/1.0.5 to look at installing PM> Install-Package NServiceBus.ActiveMQ into the projects. Ensure that the configuration for the IBus is set for ActiveMQ, as shown here:

```csharp
protected void Application_Start()
{
    Configure.ScaleOut(s => s.UseSingleBrokerQueue());
    _startableBus = Configure.With()
        .DefaultBuilder()
        .RijndaelEncryptionService()
        .UseTransport<ActiveMQ>()
        .UnicastBus()
        .CreateBus();
    Configure.Instance.ForInstallationOn<Windows>().Install();
    _bus = _startableBus.Start();
    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

Ensure that the App.Config or Web.Config file have the appropriate connection string for the NServiceBus/Transport string to point at the correct instance of the ActiveMQ queues:

```xml
<connectionStrings>
  <add name="NServiceBus/Transport" connectionString="ServerUrl=activemq:tcp://localhost:61616"/>
</connectionStrings>
```
Some of the `EndpointConfig.cs` files may not explicitly call the `IBus.Configure()` method. So make sure that the ActiveMQ using transport call is explicitly called in the `EndpointConfig` class:

```csharp
namespace SFTPClient
{
    using NServiceBus;

    ///
    /// This class configures this endpoint as a Server. More information about how to configure the NServiceBus host
    /// can be found here: [http://particular.net/articles/the-nservicebus-host](http://particular.net/articles/the-nservicebus-host)
    ///</

    public class EndpointConfig : IConfigureThisEndpoint, IAA_Publisher, UsingTransports<ActiveMQ>
    {
        //
    }
}
```

When we start the MVC application and associated saga code, we can see that the queues are created in ActiveMQ by looking through the admin console of ActiveMQ at `http://localhost:8161/admin/queues.jsp`. Notice that the queues were created matching these programs. When we execute this example, we should have queues created in the ActiveMQ administration tool as shown here:

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages Pending</th>
<th>Messages Consumed</th>
<th>Messages Expired</th>
<th>Messages Expired</th>
<th>Name</th>
<th>Explained</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSOrder</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
<tr>
<td>MSOrder parcels</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>Send To Queue 1</td>
</tr>
</tbody>
</table>

At this point, we see that the queues are working for ActiveMQ, and we have a program that we can now start extending to use ActiveMQ with an MVC frontend, using Entity Frameworks, into saga data and user tables. This example will be the ActiveMQ solution.
There were not many changes needed in the code to change it from the RabbitMQ queues to the ActiveMQ queues. We can see the queuing in the Windows consoles as they run in the screenshot here:

Summary
In this chapter, we have discussed about building a WCF server application. Then we looked at building an MVC application that interfaces into a WCF client to communicate with the WCF server.

We extended the MVC-WCF example by constructing sagas and message handlers in a unit testing environment. We then moved onto led into unit testing sagas and message handlers as we discussed testing using the NServiceBus.Testing framework.

We took a deep dive into our example after adding WCF, and later added the bus to decouple the browser from services like WCF so as to enable the user to continue in the browser. This chapter also took into account the errors that could occur in web services.
We then took the example and changed the transport mechanism from MSMQ to RabbitMQ with minor changes. We discussed in testing how we can change sagas and message handlers to be similar, and also discussed how we can change them enough to enhance their use. We also briefly discussed the many NServiceBus testing rules to build the handlers in the unit testing environment, without worrying about the endpoints until later.

We will continue in the upcoming chapters to go through more snippets and scenarios. We will also go into greater detail about using handlers for transactional and error handling needs. We will also talk further about using tools in these upcoming discussions.
Where to buy this book

You can buy Learning NServiceBus Sagas from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.