



Community Experience Distilled

# Learning NServiceBus

Build reliable and scalable distributed software systems using the industry leading .NET Enterprise Service Bus

David Boike

**[PACKT]** open source\*  
PUBLISHING community experience distilled

[www.allitebooks.com](http://www.allitebooks.com)

# Learning NServiceBus

Build reliable and scalable distributed software systems using the industry leading .NET Enterprise Service Bus

**David Boike**

**[PACKT]** open source   
PUBLISHING community experience distilled  
BIRMINGHAM - MUMBAI

# Learning NServiceBus

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1200813

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-78216-634-4

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Suresh Mogre ([suresh.mogre.99@gmail.com](mailto:suresh.mogre.99@gmail.com))

# Credits

**Author**

David Boike

**Project Coordinator**

Suraj Bist

**Reviewers**

Hadi Eskandari

Johannes Gustafsson

Andreas Öhlund

**Proofreader**

Maria Gould

**Indexer**

Priya Subramani

**Acquisition Editor**

Andrew Duckworth

**Production Coordinator**

Zahid Shaikh

**Commissioning Editor**

Shreerang Deshpande

**Cover Work**

Zahid Shaikh

**Technical Editors**

Anita Nayak

Vrinda Nitesh Bhosale

# About the Author

**David Boike** is a Principal Consultant with ILM Professional Services with more than a decade of development experience in ASP.NET and related technologies and has been an avid proponent of NServiceBus since Version 2.0 in 2010. He is also an alumnus of Udi Dahan's Advanced Distributed Systems Design course. David resides in the Twin Cities with his wife and daughter. He can be found on Twitter at @DavidBoike and on his blog at <http://www.make-awesome.com>.

---

First and foremost, I would like to thank my wife for being patient with me while I was writing this book. Secondly, I would like to thank everyone at NServiceBus and all of the NServiceBus Champions for their willingness to answer questions and offer feedback throughout the writing process. Lastly, I would like to thank Sally Stewart, who gave me a little push at exactly the right moment.

---

# About the Reviewers

**Hadi Eskandari** was born in Tehran, Iran and has always been an early adapter of new technologies and tools on Microsoft Platform. Currently he lives in Sydney, Australia, and works as a Senior Software Developer for Readify. He likes to get involved in OSS and has contributed to open source projects such as NHibernate, Rhino Licensing, and NServiceBus.

---

I want to thank my family for their support and belief in me.  
In particular, I want to thank my wife for supporting me through the hard times: I couldn't have made it without your love, support, and inspiration!

---

**Johannes Gustafsson** is 34 years old and lives with his family in Skövde, a midsized town in the middle of Sweden.

He has 15 years of experience as a professional developer and has been working with NServiceBus since version 1.8. He is also an NServiceBus Champ and committer.

He is currently employed as Lead Developer at InExchange Factorum AB.

**Andreas Öhlund**, the technical director for NServiceBus, is an enterprise development expert with thorough experience of messaging-based solutions. Andreas is a passionate developer, speaker, and trainer. You can read his blog over at <http://andreasohlund.net> or follow him on Twitter using @andreasohlund.

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

### Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

### Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Getting on the IBus</b>	<b>5</b>
<b>Why NServiceBus?</b>	<b>5</b>
<b>Getting the code</b>	<b>7</b>
NServiceBus NuGet packages	8
<b>Creating a message assembly</b>	<b>9</b>
<b>Creating a service endpoint</b>	<b>10</b>
<b>Creating a message handler</b>	<b>11</b>
<b>Sending a message from an MVC application</b>	<b>13</b>
Creating the MVC website	13
<b>Running the solution</b>	<b>17</b>
<b>Summary</b>	<b>20</b>
<b>Chapter 2: Messaging Patterns</b>	<b>21</b>
<b>Commands versus events</b>	<b>21</b>
Eventual consistency	22
Achieving consistency with messaging	23
<b>Events</b>	<b>24</b>
Publishing an event	25
Subscribing to an event	27
<b>Message routing</b>	<b>30</b>
<b>Summary</b>	<b>32</b>
<b>Chapter 3: Preparing for Failure</b>	<b>33</b>
<b>Fault tolerance and transactional processing</b>	<b>34</b>
<b>Error queues and replay</b>	<b>35</b>
Automatic retries	36
Replaying errors	37
Second level retries	38
RetryDemo	40



<b>Express messaging</b>	<b>40</b>
<b>Messages that expire</b>	<b>41</b>
<b>Auditing messages</b>	<b>42</b>
<b>Web service integration and idempotence</b>	<b>43</b>
<b>Summary</b>	<b>46</b>
<b>Chapter 4: Self-Hosting</b>	<b>47</b>
<b>Web app and custom hosting</b>	<b>47</b>
Assembly scanning	48
Choosing an endpoint name	49
Dependency injection	49
Message transport	50
Why use a different transport?	50
ActiveMQ	51
RabbitMQ	51
SQL server	52
Windows Azure	52
Purging the queue on startup	52
Bus options	53
Startup	53
Send-only endpoints	54
<b>Summary</b>	<b>55</b>
<b>Chapter 5: Advanced Messaging</b>	<b>57</b>
<b>Modifying the NServiceBus host</b>	<b>57</b>
General extension points	57
Dependency injection	59
<b>Additional bus settings</b>	<b>61</b>
Message serializer	61
Transaction settings	62
<b>The unobtrusive mode</b>	<b>62</b>
<b>Message versioning</b>	<b>64</b>
Polymorphic dispatch	64
Events as interfaces	66
<b>Specifying the handler order</b>	<b>67</b>
<b>Message actions</b>	<b>68</b>
Stopping a message	68
Deferring a message	69
Forwarding messages	69
Message headers	69
<b>Unit of work</b>	<b>70</b>
<b>Message mutators</b>	<b>71</b>
<b>Property encryption</b>	<b>72</b>
<b>Transporting large payloads</b>	<b>73</b>

---

<b>Exposing web services</b>	<b>76</b>
<b>Summary</b>	<b>77</b>
<b>Chapter 6: Sagas</b>	<b>79</b>
<hr/>	
<b>Long-running processes</b>	<b>79</b>
<b>Defining a saga</b>	<b>80</b>
<b>Finding saga data</b>	<b>82</b>
<b>Ending a saga</b>	<b>83</b>
<b>Dealing with time</b>	<b>85</b>
<b>Design guidelines</b>	<b>87</b>
Business logic only	87
Saga lifetime	88
Saga patterns	89
Retraining the business	90
<b>Unit testing</b>	<b>90</b>
<b>Scheduling</b>	<b>93</b>
<b>Summary</b>	<b>94</b>
<b>Chapter 7: Administration</b>	<b>95</b>
<hr/>	
<b>Service installation</b>	<b>95</b>
<b>Profiles</b>	<b>97</b>
Environmental profiles	97
Feature profiles	98
Customizing profiles	99
Logging profiles	101
Customizing the log level	101
<b>Managing configurations</b>	<b>102</b>
<b>Monitoring</b>	<b>103</b>
<b>Scalability</b>	<b>104</b>
Scaling up	104
Scaling out	105
Decommissioning a worker	107
<b>Multiple sites</b>	<b>108</b>
<b>Managing RavenDB</b>	<b>109</b>
<b>Virtualization</b>	<b>110</b>
Message storage	110
<b>Summary</b>	<b>111</b>
<b>Chapter 8: Where to Go from Here?</b>	<b>113</b>
<hr/>	
<b>What we've learned</b>	<b>113</b>
<b>What next?</b>	<b>115</b>
<b>Index</b>	<b>117</b>

---



# Preface

Today's distributed applications need to be built on the principles of asynchronous messaging in order to be successful. While you could try to build this infrastructure yourself, to do so would be folly. NServiceBus is a framework that can give you a proven asynchronous messaging API and so much more.

This book will be your guide to NServiceBus. From sending a simple message, to publishing events, to implementing complex long-running processes and deploying a system to production, you'll learn everything you need to know to start building complex distributed systems in no time.

## What this book covers

*Chapter 1, Getting on the IBus*, introduces NServiceBus and get us started using the framework. We will learn how to get the framework and send our first message with it.

*Chapter 2, Messaging Patterns*, discusses asynchronous messaging theory, and introduces the concept of Publish/Subscribe, showing how we can achieve decoupling by publishing events.

*Chapter 3, Preparing for Failure*, introduces concepts like automatic retry that give us the ability to build a system that can deal with failure.

*Chapter 4, Self-Hosting*, gives us a glimpse into the many ways NServiceBus can be configured by analyzing the options for self-hosting the Bus.

*Chapter 5, Advanced Messaging*, delves into the advanced topics that will allow us to take full control over the NServiceBus message pipeline.

*Chapter 6, Sagas*, introduces the long-running business process known as a saga and explains how they are built and tested.

*Chapter 7, Administration*, shows how to build, deploy, monitor, and scale a successful NServiceBus system in a production environment.

*Chapter 8, Where to Go from Here?*, summarizes what we have learned in the book and lists additional sources of information.

## What you need for this book

This book covers NServiceBus 4.0, and as such, the requirements for this book are the same as for the software it covers:

- Microsoft .NET Framework 4.0
- Visual Studio 2010 or later

Additionally, the code samples use ASP.NET MVC 3 for web projects.

## Who this book is for

This book is for senior developers and software architects who need to build distributed software systems and software for the enterprise. It is assumed that you are familiar with the .NET Framework 4.0. A passing understanding of the fundamentals of ASP.NET MVC will also be helpful when discussing web-based projects.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the interface definition, implement the `IEvent` interface."

A block of code is set as follows:

```
public interface IUserCreatedEvent : IEvent
{
    Guid UserId { get; set; }
    string EmailAddress { get; set; }
    string Name { get; set; }
}
```



When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:



```
public interface IUserCreatedEvent : IEvent
{
    Guid UserId { get; set; }
    string EmailAddress { get; set; }
    string Name { get; set; }
}
```

Any command-line input or output is written as follows:

```
PM> Install-Package NServiceBus.Host -ProjectName UserService
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1

## Getting on the IBus

In this chapter, we'll explore the basics of NServiceBus by downloading the NServiceBus code and using it to build a simple solution to send a message from an MVC website to a backend service for processing.

### Why NServiceBus?

Before diving in, we should take a moment to consider why NServiceBus might be a tool worth adding to your repertoire. If you're eager to get started, feel free to skip this section and come back later.

So what is NServiceBus? It's a powerful, extensible framework that will help you to leverage the principles of **Service-oriented architecture (SOA)** to create distributed systems that are more reliable, more extensible, more scalable, and easier to update.

That's all well and good, but if you're just picking up this book for the first time, why should you care? What problems does it solve? How will it make your life better?

Ask yourself whether any of the following situations describe you:

- My code updates values in several tables in a transaction, which acquires locks on those tables, so it frequently runs into deadlocks under load. I've optimized all the queries that I can. The transaction keeps the database consistent but the user gets an ugly exception and has to retry what they were doing, which doesn't make them very happy.
- Our order processing system sometimes fails on the third of three database calls. The transaction rolls back and we log the error, but we're losing money because the end user doesn't know if their order went through or not, and they're not willing to retry for fear of being double charged, so we're losing business to our competitor.



- We built a system to process images for our clients. It worked fine for a while but now we've become a victim of our own success. We designed it to be multithreaded (which was no small feat!) but we already maxed out the original server it was running on, and at the rate we're adding clients it's only a matter of time until we max out this one too. We need to scale it out to run on multiple servers but have no idea how to do it.
- We have a solution that is integrating with a third-party web service, but when we call the web service we also need to update data in a local database. Sometimes the web service times out, so our database transaction rolls back, but sometimes the web service call does actually complete at the remote end, so now our local data and our third-party provider's data are out of sync.
- We're sending emails as part of a complex business process. It is designed to be retried in the event of a failure, but now customers are complaining that they're receiving duplicate emails, sometimes dozens of them. A failure occurs after the email is sent, the process is retried, and the emails is sent over and over until the failure no longer occurs.
- I have a long-running process that gets kicked off from a web application. The website sits on an interstitial page while the backend process runs, similar to what you would see on a travel site when you search for plane tickets. This process is difficult to set up and fairly brittle. Sometimes the backend process fails to start and the web page just spins forever.
- We added latitude and longitude to our customer database, but now it is a nightmare to try to keep that information up-to-date. When a customer's address changes, there is nothing to make sure the location information is also recalculated. There are dozens of procedures that update the customer address, and not all of them are under our department's control.

If any of these situations has you nodding your head in agreement, I invite you to read on.

NServiceBus will help you to make multiple transactional updates utilizing the principle of eventual consistency so that you do not encounter deadlocks. It will ensure that valuable customer order data is not lost in the deep dark depths of a multi-megabyte log file.

By the end of the book, you'll be able to build systems that can easily scale out, as well as up. You'll be able to reliably perform non-transactional tasks such as calling web services and sending emails. You will be able to easily start up long-running processes in an application server layer, leaving your web application free to process incoming requests, and you'll be able to unravel your spaghetti codebases into a logical system of commands, events, and handlers that will enable you to more easily add new features and version the existing ones.

You could try to do this all on your own by rolling your own messaging infrastructure and carefully applying the principles of service-oriented architecture, but that would be really time consuming. NServiceBus is the easiest solution to solve the aforementioned problems without having to expend too much effort to get it right, allowing you to put your focus on your business concerns, where it belongs.

So if you're ready, let's get started creating an NServiceBus solution.

## Getting the code

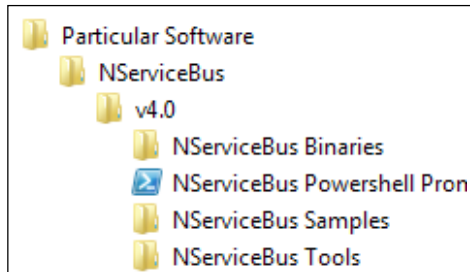
We will be covering a lot of information very quickly in this chapter, so if you see something that doesn't immediately make sense, don't panic! Once we have the basic example in place, we will loop back and explain some of the finer points more completely.

There are two main ways to get the NServiceBus code integrated with your project, by downloading the Windows Installer package, and via NuGet. I recommend you use Windows Installer the first time to ensure that your machine is set up properly to run NServiceBus, and then use NuGet to actually include the assemblies in your project.

Windows Installer automates quite a bit of setup for you, all of which can be controlled through the advanced installation options:

- NServiceBus binaries, tools, and sample code are installed.
- The NServiceBus Management Service is installed to enable integration with ServiceInsight. We'll talk more about this in *Chapter 3, Preparing for Failure*.
- **Microsoft Message Queuing (MSMQ)** is installed on your system if it isn't already. MSMQ provides the durable, transactional messaging that is at the core of NServiceBus.
- The **Distributed Transaction Coordinator (DTC)** is configured on your system. This will allow you to receive MSMQ messages and coordinate data access within a transactional context.
- RavenDB is installed, which provides the default persistence mechanism for NServiceBus subscriptions, timeouts, and saga data.
- NServiceBus performance counters are added to help you monitor NServiceBus performance.

Download the installer from <http://particular.net/downloads> and install it on your machine. After the install is complete, everything will be accessible from your **Start** Menu. Navigate to **All Programs | Particular Software | NServiceBus** as shown in the following screenshot:



The install package includes several samples that cover all the basics as well as several advanced features. The **Video Store** sample is a good starting point. Multiple versions of it are available for different message transports that are supported by NServiceBus. If you don't know which one to use, take a look at **VideoStore.Msmq**. We will learn more about message transports in *Chapter 4, Self-Hosting*.

I encourage you to work through all of the samples, but for now we are going to roll our own solution by pulling in the NServiceBus NuGet packages.

## NServiceBus NuGet packages

Once your computer has been prepared for the first time, the most direct way to include NServiceBus within an application is to use the NuGet packages.

There are four core NServiceBus NuGet packages:

- `NServiceBus.Interfaces`: This package contains only interfaces and abstractions, but not actual code or logic. This is the package that we will use for message assemblies, however in *Chapter 5, Advanced Messaging*, we will learn how to create message assemblies with zero dependencies using Unobtrusive Mode.
- `NServiceBus`: This package contains the core assembly with most of the code that drives NServiceBus except for the hosting capability. This is the package we will reference when we host NServiceBus within our own process, such as in a web application.
- `NServiceBus.Host`: This package contains the service host executable. With the host we can run an NServiceBus service endpoint from the command line during development, and then install it as a Windows service for production use.

- `NServiceBus.Testing`: This package contains a framework for unit testing `NServiceBus` endpoints and sagas. We will cover this in more detail in *Chapter 6, Sagas*.

The NuGet packages will also attempt to verify that your system is properly prepared through PowerShell cmdlets that ship as part of the package. However, if you are not running Visual Studio as an Administrator, this can be problematic as the tasks they perform sometimes require elevated privileges. For this reason it's best to run Windows Installer before getting started.

## Creating a message assembly

The first step to creating an `NServiceBus` system is to create a messages assembly. Messages in `NServiceBus` are simply plain old C# classes. Like the WSDL document of a web service, your message classes form a contract by which services communicate with each other.

For this example, let's pretend we're creating a website like many on the Internet, where users can join and become a member. We will construct our project so that the user is created in a backend service and not in the main code of the website.

Follow these steps to create your solution:

1. In Visual Studio, create a new class library project. Name the project `UserService.Messages` and the solution simply `Example`. This first project will be your messages assembly.
2. Delete the `Class1.cs` file that came with the class project.
3. From the NuGet Package Manager Console, run this command to install the `NServiceBus.Interfaces` package, which will add the reference to `NServiceBus.dll`.  

```
PM> Install-Package NServiceBus.Interfaces -ProjectName  
      UserService.Messages
```
4. Add a new folder to the project called `Commands`.
5. Add a new class to the `Commands` folder called `CreateNewUserCmd.cs`.
6. Add `using NServiceBus;` to the `using` block of the class file. It is very helpful to do this first so that you can see all of the options available with IntelliSense.
7. Mark the class as `public` and implement `ICommand`. This is a marker interface so there is nothing you need to implement.
8. Add the `public` properties for `EmailAddress` and `Name`.

When you're done, your class should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NServiceBus;

namespace UserService.Messages.Commands
{
    public class CreateNewUserCmd : ICommand
    {
        public string EmailAddress { get; set; }
        public string Name { get; set; }
    }
}
```

#### Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

Congratulations! You've created a message! This will form the communication contract between the message sender and receiver. Unfortunately, we don't have enough to run yet, so let's keep moving.

## Creating a service endpoint

Now we're going to create a service endpoint that will handle our command message.

1. Add a new class library project to your solution. Name the project `UserService`.
2. Delete the `Class1.cs` file that came with the class project.
3. From the NuGet **Package Manager Console** window, run this command to install the `NServiceBus.Host` package:

```
PM> Install-Package NServiceBus.Host -ProjectName UserService
```

4. Take a look at what the host package has added to your class library. Don't worry; we'll cover this in more detail later.
  - References to `NServiceBus.Host.exe`, `NServiceBus.Core.dll`, and `NServiceBus.dll`
  - An `App.config` file
  - A class named `EndpointConfig.cs`
5. In the service project, add a reference to the **UserService.Messages** project you created before.
6. Right-click on the project file and click on **Properties**, then in the property pages, navigate to the **Debug** tab and enter `NServiceBus.Lite` under **Command line arguments**. This tells NServiceBus not to run the service in production mode while we're just testing. This may seem obvious, but this is part of the NServiceBus promise to be safe by default, meaning you won't be able to mess up when you go to install your service in production.

## Creating a message handler

Now we will create a message handler within our service.

1. Add a new class to the service called `UserCreator.cs`.
2. Add three namespaces to the using block of the class file:

```
using NServiceBus;
using NServiceBus.Logging;
using UserService.Messages.Commands;
```
3. Mark the class as `public`.
4. Implement `IHandleMessages<CreateNewUserCmd>`.
5. Implement the interface using Visual Studio's tools. This will generate a `Handle(CreateNewUserCmd message)` stub method.

Normally we would want to create the user here with calls to a database, but we don't have time for that! We're on a mission, so let's just demonstrate what would be happening by logging a message.

It is worth mentioning that a new feature in NServiceBus 4.0 is the ability to use any logging framework you like, without being dependent upon that framework. NServiceBus can automatically hook up to log4net or NLog—just add a reference to either assembly, and NServiceBus will find it and use it. You can even roll your own logging implementation if you wish.

However, it is not required to pick a logging framework at all. NServiceBus internalizes log4net, which it will use (via the `NServiceBus.Logging` namespace) if you don't explicitly include a logging library. This is what we will be doing in our example.

Now let's finish our fake implementation for the handler:

1. Above the `Handle` method, add an instance of a logger:

```
private static readonly ILog log =
    LogManager.GetLogger(typeof(UserCreator));
```

2. To handle the command, remove `NotImplementedException` and replace it with:

```
log.InfoFormat("Creating user '{0}' with email '{1}'",
    message.Name,
    message.EmailAddress);
```

When you're done, your class should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UserService.Messages.Commands;
using NServiceBus;

namespace UserService
{
    public class UserCreator : IHandleMessages<CreateNewUserCmd>
    {
        private static readonly ILog log =
            LogManager.GetLogger(typeof(UserCreator));

        public void Handle(CreateNewUserCmd message)
        {
            log.InfoFormat("Creating user '{0}' with email '{1}'",
                message.Name,
                message.EmailAddress);
        }
    }
}
```

Now we have a command message and a service endpoint to handle it. It's OK if you don't understand quite how it all connects quite yet. Next we need to create a way to send the command.

## Sending a message from an MVC application

An ASP.NET MVC web application will be the user interface for our system. It will be sending a command to create a new user to the service layer, which will be in charge of processing it. Normally this would be from a user registration form, but in order to keep the example to the point, we'll take a shortcut and enter the information as query string parameters, and return data as JSON.



Because we will be viewing JSON data directly within a browser, it would be a good idea to make sure your browser supports displaying JSON directly instead of downloading it.

Firefox and Chrome natively display JSON data as plain text, which is readable but not very useful. Both browsers have an extension available called JSONView (although they are unrelated) which allows you to view the data in a more readable, indented format. Either of these options will work fine, so you can use whichever browser you prefer.

Beware that Internet Explorer will try to download JSON data to a file, which makes it cumbersome to view the output.

## Creating the MVC website

First, follow these directions to get the MVC website set up. You can use either MVC 3 or MVC 4, but for the example we will be using MVC 3.

1. Add a new ASP.NET MVC project to your solution and name it `ExampleWeb`. Select the **Empty** template and the Razor view engine.
2. From the NuGet **Package Manager Console**, run this command to install the NServiceBus package:  

```
PM> Install-Package NServiceBus -ProjectName ExampleWeb
```
3. Add a reference to the **UserService.Messages** project you created before.

Because the MVC project isn't fully controlled by NServiceBus, it is a little more involved to set up.

First, create a class file within the root of your MVC application and name it `ServiceBus.cs`, then fill it with this code. For the moment, don't worry about what it does.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using NServiceBus;
using NServiceBus.Installation.Environments;

namespace ExampleWeb
{
    public static class ServiceBus
    {
        public static IBus Bus { get; private set; }

        public static void Init()
        {
            if (Bus != null)
                return;

            lock (typeof(ServiceBus))
            {
                if (Bus != null)
                    return;

                Bus = Configure.With()
                    .DefineEndpointName("ExampleWeb")
                    .DefaultBuilder()
                    .UseTransport<Msmq>()
                    .PurgeOnStartup(true)
                    .UnicastBus()
                    .CreateBus()
                    .Start(() => Configure.Instance
                        .ForInstallationOn<Windows>()
                        .Install());
            }
        }
    }
}
```

That was certainly a mouthful! Don't worry about remembering all this; it's part of a fluent API that makes it pretty easy to discover things you need to configure through IntelliSense. We will come back to this in *Chapter 4, Self-Hosting*, and explain everything that's going on.

For now, suffice it to say that this is the code that initializes the service bus within our MVC application, and provides access to a single static instance of the `IBus` interface that we can use to access the service bus. If we were to compare the service bus to Ethernet (which is a fairly apt comparison) we have just detailed how to turn on the Ethernet card.

Now we need to call the `Init()` method from our `Global.asax.cs` file so that the `Bus` property is initialized when the application starts up.

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);

    ServiceBus.Init();
}
```

Now `NServiceBus` has been set up to run in the web application, so we can send our message. Create a `HomeController` class and add these methods to it:

```
public ActionResult Index()
{
    return Json(new { text = "Hello world." });
}

public ActionResult CreateUser(string name, string email)
{
    var cmd = new CreateNewUserCmd
    {
        Name = name,
        EmailAddress = email
    };

    ServiceBus.Bus.Send(cmd);

    return Json(new { sent = cmd });
}

protected override JsonResult JsonResult(object data,
    string contentType,
    System.Text.Encoding contentEncoding,
    JsonRequestBehavior behavior)
{
```

```
        return base.Json(data, contentType, contentEncoding,
            JsonRequestBehavior.AllowGet);
    }
```

The first and last methods aren't too important. The first returns some static JSON for the `/Home/Index` action because we aren't going to bother adding a view for it. The last one is for convenience to make it easier to return JSON data as a result of an HTTP `GET` request.

The highlighted method is the important one – this is where we create an instance of our command class and send it on the bus via the static instance `ServiceBus.Bus`. Lastly we return the command to the browser as JSON data so that we can see what we created.

The last step is to add some `NServiceBus` configuration to the MVC application's `Web.config` file. We need to add two configuration sections. We already saw `MessageForwardingInCaseOfFaultConfig` in the `app.config` file that NuGet added to the service project, so we can copy it from there. However we need to add a new section called `UnicastBusConfig` anyway, so the XML for both is included here for convenience:

```
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
      type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig,
        NServiceBus.Core" />
    <section name="UnicastBusConfig"
      type="NServiceBus.Config.UnicastBusConfig,
        NServiceBus.Core" />
  </configSections>

  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="UserService.Messages" Endpoint="UserService"
        />
    </MessageEndpointMappings>
  </UnicastBusConfig>

  <!-- Rest of Web.config -->

</configuration>
```

The first highlighted line determines what happens to a message that fails. This will be covered in more depth in *Chapter 3, Preparing for Failure*. The second highlighted line determines routing for messages. This will be covered in more depth in the Publish/Subscribe section *Chapter 2, Messaging Patterns*, but for now it is sufficient to say that it means that all messages found in the `UserService.Messages` assembly will be sent to the `UserService` endpoint, which is our service project.



NServiceBus also includes PowerShell cmdlets that make it a lot easier to add these configuration blocks. You could generate these sections using the `Add-NServiceBusMessageForwardingInCaseOfFaultConfig` cmdlet and the `Add-NServiceBusUnicastBusConfig` cmdlet.

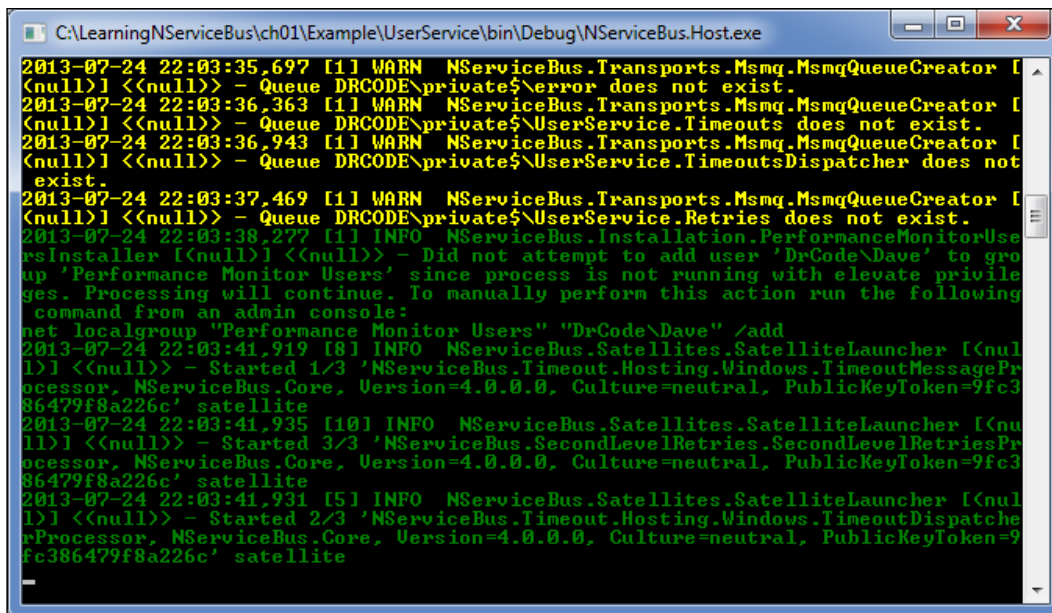
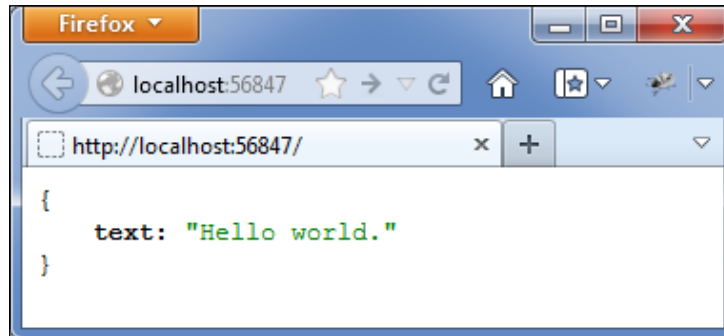
## Running the solution

One thing that will be useful when developing NServiceBus solutions is being able to specify multiple startup projects for a solution.

1. In the **Solution Explorer**, right-click on the solution file and click on **Properties**.
2. On the left, navigate to **Common Properties | Startup Project**.
3. Select the **Multiple startup projects** radio button.
4. Set the Action for the service project and the MVC project to **Start** and order them so that the MVC project starts last.
5. Click on **OK**.

Now, build the solution if you haven't already, and assuming there are no compilation errors, click on the **Start Debugging** button or press *F5*.

So what happens now? Let's take a look.



When you run the solution, both the MVC website and a console window should appear as shown in the preceding screenshots. As we can see, the browser window isn't terribly exciting right now; it's just showing the JSON results of the /Home/Index action. The console window is far more interesting.

If you remember, we never created a console application; our service endpoint was a class project. When we included the NServiceBus.Host NuGet package, a reference to NServiceBus.Host.exe was added to the class project (remember a .NET executable is also an assembly that can be referenced) and the project was set to run that executable when you debug it.

While it might not be easy to see in the screenshot, NServiceBus uses different colors to log messages of different levels of severity. In the screenshot, INFO messages are logged in green, and WARN messages are displayed in yellow. In addition, there can be DEBUG messages displayed in white, or ERROR and FATAL messages which are both logged in red. By default, the INFO log level is used for display, which is filtering out all the DEBUG messages here, and luckily we don't have any ERROR or FATAL messages!

The entire output is too much to show in a screenshot. It's worth reading through, even though you may not understand everything that's going on quite yet. Here are some of the important points:

- NServiceBus reports how many total message types it has found. In my example, four messages were found. Only one of those is ours; the rest are administrative messages used internally by NServiceBus. If this had said zero messages were found, that would have been distressing! We will revisit this message in *Chapter 5, Advanced Messaging*.
- The License Manager checks for a valid license. You can get a free developer license that allows unrestricted non-production use for 90 days. At the end of that, you can get a new one for another 90 days. For all licensing concerns, go to <http://particular.net/licensing>.
- The status of several features is listed for debugging purposes.
- NServiceBus checks for the existence of several queues, and creates them if they do not exist. In fact, if we go to the Message Queuing manager, we will see that the following private queues have now been created:
  - `audit`
  - `error`
  - `exampleweb`
  - `exampleweb.retries`
  - `exampleweb.timeouts`
  - `exampleweb.timeoutdispatcher`
  - `userservice`
  - `userservice.retries`
  - `userservice.timeouts`
  - `userservice.timeoutsdispatcher`

That's a lot of plumbing that NServiceBus takes care of for us! But this just gets the endpoint ready to go. We still need to send a message!

Visual Studio will likely give you a different port number for your MVC project than in the example, so change the URL in your browser to the following, keeping the host and port the same. Feel free to use your own name and email address:

```
/Home/CreateUser?name=David&email=david@example.com
```

Look at what happens in your service window:

```
INFO UserService.UserCreator [(null)] <(null)> - Creating user  
'David' with email 'david@example.com'
```

This might seem simple, but consider what had to happen for us to see this message. First, in the MVC website, an instance of our message class was serialized to XML, and then that payload was added to an MSMQ message with enough metadata to describe where it came from and where it needed to go. The message was sent to an input queue for our background service, where it waited to be processed until the service was ready for it. The service pulled it off the queue within a transaction, deserialized the XML payload, and was able to determine a handler that could process the message. Finally, our message handler was invoked, which resulted in the message being output to the log.

This is a great start, but there is a great deal more to discover.

## Summary

In this chapter, we created an MVC web application and an NServiceBus hosted service endpoint. Through the web application, we sent a command to the service layer to create a user, where we just logged the fact that the command was received, but in real life we would likely perform database work to actually create the user. For our example, our service was running on the same computer, but our command could just as easily be sent to a different server, enabling us to offload work from our web server.

In the next chapter we will take the code we developed here and extend it using Publish/Subscribe to enable decoupling services from each other. Then we will start to discover the true power that NServiceBus has to offer.

# 2

## Messaging Patterns

Sending messages is powerful, but still assumes coupling between the sender and the receiver. In this chapter we'll first delve deeper into the concept of asynchronous messaging, and then explore the Publish/Subscribe model, and discover how publishing events allows us to decouple services from one another.

By the end of the chapter, we will have updated our solution from the previous chapter to publish an event once the user has been created, and then we will show how we can create multiple subscribers to add functionality to a system without requiring changes to the original publisher.

### Commands versus events

In the previous chapter, the MVC website sent a command to the NServiceBus endpoint, commanding it to perform an action on its behalf. This is similar to a web service or any other **remote procedure call (RPC)** style of communication. The message sender must necessarily know not only how to communicate with the receiver, but also know what it expects the server to do once it receives the message.



A **command** is a message that can be sent from one or more logical senders, and is processed by a single logical receiver.

The main difference between sending an NServiceBus command and an RPC request is that an RPC call will block the client until the server sends a response back. There is no way that the client can continue without the response. Sending an NServiceBus command, however, is completely asynchronous. Although it's possible to emulate an RPC call using messaging, doing so is an anti-pattern and should be avoided whenever possible.



## Eventual consistency

This brings to the forefront the concept of **eventual consistency**, the notion that in a system with disconnected, one-way only communication, state may not be entirely consistent at every single moment, but will eventually be consistent assuming that all messages are eventually processed successfully.

But doesn't this fly in the face of what we've been taught to think about transactional processes? After all, in the acronym **ACID (Atomicity, Consistency, Isolation, Durability)** consistency is featured very prominently, and is not ACID the yardstick by which all database systems are measured?

The problem lies in building distributed systems, due to the **Fallacies of Distributed Computing** coined in 1994 by Peter Deutsch and his colleagues at Sun Microsystems:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

If we think critically about any of these statements we know them to be false, but a lot of times, we seem to look the other way when writing code. Network switches go up in smoke. Servers lose power. Someone trips over the network cord. These are all things that have happened and will happen again, *maybe even to you*.

When there are network issues, RPC communication falls flat on its face. The communication to the server cannot complete and the calling code cannot continue without the server's response. In the best case, data is lost. In the worst case, threads pile up waiting for responses that never come. In this case it is only a matter of time until the process becomes unresponsive.

In fact, this is even mathematically proven! The CAP theorem states, in a nutshell, that in any distributed system you can only have any two of the **Consistency**, **Availability**, and **Partition tolerance**, which is a communication failure between two nodes. The theorem proves it is impossible to have all three simultaneously. A complete discussion on CAP is beyond the scope of this book (many great resources exist on the Internet) but what is important is that this leads us to a new consistency style called **BASE (Basically Available, Soft state, Eventual consistency)** as opposed to ACID. We opt for availability and partition tolerance and in doing so sacrifice consistency for a short time. Then we make up for this lack of immediate consistency by sending messages.

## Achieving consistency with messaging

Consider a situation where you integrate with third-party providers via web services. For instance, let's say you are building an application that allows users to subscribe to different types of notifications via email or SMS. In order to subscribe a user, you must first update a local database, then call a web service for the email component, and then call another service for the SMS component.

What happens if the database transaction succeeds, and the email service call succeeds, but then the SMS service call fails? If you roll back the database transaction, it will appear as if the user has not subscribed, yet they will be receiving emails from the third-party provider. If you commit the database transaction, the user will appear subscribed but they will not receive the SMS notifications they wanted.

What happens if one of the third-party web services becomes extremely slow? It's entirely possible that the code calling the web service will time out, rolling back the database transaction, but on the third-party server the call actually succeeds, albeit very slowly.

By sending commands with `IServiceBus`, all these problems are removed by only taking on as much work as can be successfully accomplished within a transaction. Each `IServiceBus` message handler automatically creates an ambient transaction, and we do a finite amount of work within that context. To see this in action, inspect the value of `System.Transactions.Transaction.Current` within a message handler.

This is how our third-party web service scenario works with commands:



1. The website sends a command to enroll the user in alerts.
2. A message handler transactionally receives the command message, and within that same transaction, modifies the database, and sends two additional commands to an email service and SMS service. If an error occurs, the command is automatically retried.
3. The email service transactionally receives a command to contact the third-party email subscription service. A web service cannot be enrolled in a transaction, but if an exception occurs, the command is returned to the queue to be retried until the web service call completes.
4. The SMS service processes its message in the same way as the email service. In fact because the messages are asynchronous, this processing can happen at the same time as the email service, perhaps on a completely different server.

Technically, the entire system is inconsistent when the database updates have been made but the web services have not yet been contacted. But this is only a temporary condition. The assumption is that all commands will eventually succeed and the system will once again be completely consistent.

By using asynchronous commands, the focus shifts from trying to guarantee consistency that was impossible in the first place to ensuring that the commands do eventually succeed through automatic retries. We will return to this discussion in *Chapter 3, Preparing for Failure*.

## Events

If you compare this definition with that of a command from earlier in the chapter, they follow the same general premise, but the differences are very important.

 An **event** is a message that is published from a single logical sender, and is processed by zero or more logical receivers. 

A command can be sent by any number of different senders. An event is only published by one logical sender. Sending a command sends one copy of a message to one receiver, and that receiver is the only entity that can process that command. In contrast, when an event is published, a copy of that message may be sent to dozens or even hundreds of subscribers, or maybe none if there are no subscribers.

This has even broader implications. While a command is an order to please do something in the future, an event is an announcement that something has *already happened*. This is why commands are often named in the imperative, such as `DoSomethingNowPleaseCmd`, while events are often named in the past tense, such as `SomethingAlreadyHappenedEvent`.



Some SOA experts would argue against using the `-Cmd` and `-Event` suffixes when naming message types, preferring instead to rely on the imperative tense for commands and past tense for events. I won't take sides; you should do whatever works best for you and your team. In this book, however, we will use the suffixes just so it is absolutely clear what we are talking about.

While commands bring you the power of eventual consistency between components that must know about each other, events give you the power to decouple components that need not know much about each other. The importance of this cannot be understated.

## Publishing an event

Now that we've covered the fundamentals of messaging theory, let's add the concept of events to our code from *Chapter 1, Getting on the IBus*, and see first-hand what it can do for us.



Now that we've gotten our feet wet with `NServiceBus`, I won't be quite so verbose with the instructions. In particular, I will omit instructions to add using declarations for `NServiceBus` namespaces, since Visual Studio should be more than capable of resolving these references for you.

1. In the `UserService.Messages` assembly, add a folder called `Events`.
2. In this folder, create an interface called `IUserCreatedEvent` and mark it as `public`.
3. In the interface definition, implement the `IEvent` interface. Like `ICommand`, there is no implementation for this; it is just another marker interface.
4. Add a `Guid` property named `UserId`, and `string` properties for `Name` and `EmailAddress`.

Your code should look like this when you're done:

```
namespace UserService.Messages.Events
{
    public interface IUserCreatedEvent : IEvent
    {
        Guid String UserId{ get; set; }
        string EmailAddress { get; set; }
        string Name { get; set; }
    }
}
```

At this point you may have noticed a few things.

First, this bears a striking resemblance in structure to the `CreateNewUserCmd` class we created in the last chapter, both in name and in structure. Thanks to the dichotomy between commands and events, it is quite common to have a related command and event: a command to request an action to be done, and a corresponding event to announce that it has been done.

Second, you may have noticed that in situations where the command we created was a class, for the event we created an interface. While it is not required for an event to be an interface (it works just fine as a class) it helps to facilitate with event versioning, and allows us to take advantage of multiple inheritance. Because of these benefits, using interfaces for your event definitions is recommended as a best practice. We will learn more about message versioning in *Chapter 5, Advanced Messaging*.

But how do we instantiate an interface without a concrete class? `NServiceBus` provides this capability for us.

1. In the `UserService` project from the last chapter, open the `EndpointConfig` class, and change the `AsA_Server` interface to `AsA_Publisher`. This notifies `NServiceBus` that we intend to publish events from this endpoint, so it will automatically take care of things such as setting up storage for subscriptions. Some transports have native support for Pub/Sub and, as a result, do not need this. The default transport (MSMQ) does not and so we must include it.
2. Now open the `UserCreator` class.
3. Add a public property to the class:

```
public IBus Bus { get; set; }
```

4. Modify the `Handle` method as follows:

```
public void Handle(CreateNewUserCmd message)
{
    log.InfoFormat("Creating user '{0}' with email '{1}'",
```

```
        message.Name,
        message.EmailAddress);

    // This is where the user would be added to the database.
    // The database command would auto-enlist in the ambient
    // transaction and either succeed or fail along with
    // the message being processed.

    Bus.Publish<IUserCreatedEvent>(evt =>
    {
        evt.UserId = Guid.NewGuid();
        evt.Name = message.Name;
        evt.EmailAddress = message.EmailAddress;
    });
}
```

We added the `Bus` instance, which `NServiceBus` automatically fills by dependency injection, which we will learn about more in *Chapter 5, Advanced Messaging*. Then we publish our message, and we see that the `Bus` property gives us a way to utilize an interface without needing a concrete type to implement it. Under the covers, a concrete class is generated to do that work for us—all we have to do is supply an `Action<IUserCreatedEvent>` lambda where we set the properties of the event we are publishing.

So now that we've published the event, what happens? Well as it turns out, not much. Internally a lot of stuff is going on, but we won't really see any big differences in the system's behavior because the event has no subscribers. What is really happening is that `NServiceBus` queries the subscription storage to ask "Who is interested in hearing about this event?" and finds that the answer is nobody, so no messages are sent. Even though no new event messages are sent, however, the `CreateNewUserCmd` message still completes successfully, along with any transactional work we may have performed while under the message handler's transaction scope. Any subscriber that we do create will get a separate transaction in which to do its own work.

So now, let's add a subscriber to our system and see Pub/Sub in action.

## Subscribing to an event

Our system so far is like any other website out there that allows users to register. A common component of these systems is an email that is sent to the new user welcoming them to the site. Now we will see how we can add that kind of functionality without modifying any of the existing components.

1. Create a new class project named `WelcomeEmailService` and delete the `Class1.cs` file.
2. From the **Package Manager Console**, install the `NServiceBus.Host` package.  
**PM> Install-Package NServiceBus.Host -ProjectName WelcomeEmailservice**
3. Add a reference to the `UserService.Messages` assembly.
4. Modify the project properties to add **NServiceBus.Lite** in the debug command line arguments, so that we invoke the Lite profile.
5. Create a class named `EmailSender` and implement `IHandleMessages<IUserCreatedEvent>`, then add logging to simulate sending the welcome email. Your class should look something like this:

```
public class EmailSender: IHandleMessages<IUserCreatedEvent>
{
    private static readonly ILog log =
        LogManager.GetLogger(typeof(EmailSender));

    public void Handle(IUserCreatedEvent message)
    {
        log.InfoFormat("Sending welcome email to {0}",
            message.EmailAddress);
    }
}
```

6. Open the `App.config` file for the `WelcomeEmailService` project and add the `UnicastBusConfig` section. You can copy and paste it directly from the MVC project's `Web.config` file, but be sure to get both the `configSection/section` element as well as the `UnicastBusConfig` section itself. It may be initially confusing why this section would be exactly the same, but don't worry; we'll explain this in detail in the next section.



You could also generate the `UnicastBusConfig` section using the PowerShell `Add-NServiceBusUnicastBusConfig` cmdlet.

7. Modify your solution startup project settings so that this new service also starts along with the web project and user creation service.

Now when you start the project, two console windows will appear. In the new one, you will see similar messages as we did in the last chapter, where `NServiceBus` creates queues for us and gets the endpoint ready to roll.

One big addition should stand out, however. In the **WelcomeEmailService** window, you should see:

```
Subscribing to UserService.Messages.Events.IUserCreatedEvent,  
UserService.Messages, Version=1.0.0.0, Culture=neutral,  
PublicKeyToken=null at publisher queue userservice@computername
```

Then in the **UserService** window, you should see:

```
Subscribing welcomeemailservice@computername to message type UserService.  
Messages.Events.IUserCreatedEvent, UserService.Messages, Version=1.0.0.0,  
Culture=neutral, PublicKeyToken=null
```

This pair of messages announces that NServiceBus has automatically subscribed the **WelcomeEmailService** to **IUserCreatedEvent** messages from the **UserService** endpoint.

Now create a user like we did in the last chapter, by pointing your browser to:

```
/Home/CreateUser?name=David&email=david@example.com
```

Just as before, the **UserService** endpoint announces that it created the user, and in addition, the **WelcomeEmailService** endpoint announces that it has sent the welcome email to the user. Neither of these things are really happening because we are just logging messages to the console, but you can start to see how powerful this Publish/Subscribe concept is. The **UserService** endpoint didn't need to know anything about the welcome email; it just published the fact that something (the user creation) had happened. The **WelcomeEmailService** endpoint was completely responsible for sending the email, and both services can be maintained independently. All they share is the contract of the published message!

It's important to remember that multiple subscribers can respond to a published event. Even a web application can be a subscriber! As an exercise, try adding a subscribing message handler to the MVC website, and use it to create a feature common on some forum websites where the last five new users are displayed on the homepage. You should have everything you need at this point save for the following tips:

- Use a static `Queue<string>` collection in the `HomeController` class to track the recently created users. Don't worry about thread safety at this point.
- Create a folder named `MessageHandlers` in the MVC website and put your message handling classes there.

If you get stuck, check out the implementation in the downloadable code.



## Message routing

Now that we've spoken about commands and events, it's a good time to discuss how `NServiceBus` routes messages. Message routing is handled completely by configuration, within the `UnicastBusConfig` section that we have seen a little bit of so far. By storing the mappings in the configuration, this allows us to test our system all on one machine, and then modify the configuration for a production scenario that uses multiple machines for processing.

Messages are routed by type, and the `UnicastBusConfig` section maps message types to the queues and machines that must process them. When defining types, we may specify an entire assembly name, which is what we have done so far. In that case, all messages within that assembly are associated with the same endpoint. We may also specify particular message classes by using their type name and assembly name together, but in a sufficiently complex system this quickly starts to become very difficult to manage! New in `NServiceBus 4.0` is the ability to specify routing based on the combination of assembly and namespace.

You may have noticed in our examples so far that the MVC website and `WelcomeEmailService` endpoint contain exactly the same routing configuration. This may seem somewhat confusing. The MVC site is sending a command to the `UserCreator` service and so the configuration associates the message assembly with the `UserCreator` queue. This seems straightforward so far.


What makes it confusing is that the `WelcomeEmailService` endpoint contains exactly the same configuration, but it doesn't send any messages to the `UserCreator` service! Or does it?

In fact it does, because the service that subscribes to an event sends a subscription request message to the publishing service. After that, the publisher stores the subscriber's information in its subscription storage, so that the subscriber will receive a copy of the event message when it is published.

When `NServiceBus` starts up, it scans through all of the endpoint's types and reads the routing configuration. If the routing configuration contains a message type that is an event, and the code contains a message handler for that event, then `NServiceBus` will automatically subscribe to that event by sending the subscription event to the endpoint in the routing configuration.

If this still seems confusing, try to remember the following:

[



]

For commands, the message endpoint mappings specify where the message should be sent to.

For events, the message endpoint mappings specify where the event is published from, and thus, where the subscription request is sent to.

Let's look at a few examples. First of all, the message endpoint mappings always look like this:

```
<UnicastBusConfig>
  <MessageEndpointMappings>
    <!-- Mappings are defined by "add" elements here -->
  </MessageEndpointMappings>
</UnicastBusConfig>
```

Then we define individual mappings, each with an `<add />` element. To register all the messages in an assembly, use one of these:

```
<add Messages="assembly" Endpoint="destination" />
<add Assembly="assembly" Endpoint="destination" />
```

Or for a fully qualified message type, use one of these:

```
<add Messages="namespace.type, assembly" endpoint="destination" />
<add Assembly="assembly" Type="namespace.type"
  Endpoint="destination" />
```

You can even filter by namespaces:

```
<add Assembly="assembly" Namespace="MyMessages.Other"
  Endpoint="destination" />
```

In these examples, `destination` is either a simple queue name such as `Endpoint="MyQueue"` for the local server or `Endpoint="MyQueue@OtherServer"` to address a remote server.

## Summary

In this chapter, we started by learning about messaging theory. We learned about the Fallacies of Distributed Computing and the CAP Theorem, and realized that although we cannot achieve full consistency in a distributed system, we can leverage the power of asynchronous messaging to achieve eventual consistency.

We then learned about the distinction between commands and events, how commands are sent by multiple senders but processed by a single logical receiver, and how events are published by only one logical publisher, but received by zero or more logical subscribers. We learned how we can use the Publish/Subscribe model to decouple business processes by announcing that some business event has happened, and allowing subscribers to respond to it in whatever way they wish.

We then demonstrated this knowledge by publishing an event with NServiceBus, and then creating multiple subscribers for that event. We learned how to configure the message endpoint mappings so that NServiceBus knows where to send our messages.

Now that we have learned how to decouple our business processes through messaging and Publish/Subscribe, we must learn how to ensure that our messages are processed successfully. In the next chapter, we will explore how to ensure that our messaging processes are reliable and can withstand failure.

# 3

## Preparing for Failure

*Alfred: Why do we fall sir? So that we can learn to pick ourselves up.*

*Bruce: You still haven't given up on me?*

*Alfred: Never.*

*-Batman Begins (Warner Bros., 2005)*

I'm sure that many readers are familiar with this scene from Christopher Nolan's Batman reboot. In fact, if you're like me, you can't read the words without hearing them in your head delivered by Michael Caine's distinguished British accent.

At this point in the movie, Bruce and Alfred have narrowly escaped a blazing fire set by the Bad Guys that is burning Wayne Manor to the ground. Bruce had taken up the mantle of the Dark Knight to rid Gotham City of evil, and instead it seems as if evil has won, with the legacy of everything his family had built turning to ashes all around him.

It is at this moment of failure that Alfred insists he will never give up on Bruce. I don't want to spoil the movie if, by chance, you haven't seen it, but let's just say some bad guys get what's coming to them.

This quote has been on my mind from the past few months as my daughter has been learning to walk. Invariably she would fall and I would think of Alfred. I realized that this short exchange between Alfred and Bruce is a fitting analogy for the design philosophy of NServiceBus.

Software fails. Software engineering is an imperfect discipline, and despite our best efforts, errors will happen. Some of us have surely felt like Bruce, when an unexpected error makes it seem as if the software we built is turning to so much ash around us.

But like Alfred, NServiceBus will not give up. If we apply the tools that

NServiceBus gives us, even in the face of failure, we will not lose consistency, we will not lose data, we can correct the error, and we will make it through.

And then the bad guys will get what's coming to them.

In this chapter we will explore the tools that NServiceBus gives us to stare failure in the face and laugh.

## Fault tolerance and transactional processing

In order to understand the fault tolerance we gain from using NServiceBus, let us first consider what happens without it.

Let's order something from a fictional website and watch what might happen to process that order. On our fictional website we add *Batman Begins* to our shopping cart and then click on the **Checkout** button. While our cursor is spinning...

1. Our web request is transmitted to the web server.
2. The web application knows it needs to make several database calls, so it creates a new transaction scope.
3. Database Call 1 of 3: The shopping cart information is retrieved from the database.
4. Database Call 2 of 3: An Order record is inserted.
5. Database Call 3 of 3: We attempt to insert OrderLine records, but instead get the following error message:

*Transaction (Process ID 54) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.*

6. The exception causes the transaction to roll back.

Ugh! If you're using SQL Server and you've never seen this, you haven't been coding long enough. It never happens during development; there just isn't enough load. And it's even possible it won't occur during load testing. It will likely occur during heavy load at the worst possible time, like right after your big launch.

So obviously we should log the error, right? But then what happens to the order? Well that's gone, and your boss may not be happy about losing that revenue. And what about our user? They will likely get a nasty error message. We won't want to divulge the actual exception message so they will get something like "An unknown

---

error has occurred. The system administrator has been notified. Please try again later." However, the likelihood that they want to trust their credit card information to a website that has already blown up in their face once is quite low.

So how can we do better? Here's how this scenario could have happened with NServiceBus:

1. The web request is transmitted to the web server.
2. We add the shopping cart identifier to an NServiceBus command and send it on the Bus.
3. We redirect the user to a new page that displays the receipt, even though the order has not yet been processed.

Elsewhere, an Order Service is ready to start processing a new message:

1. The service creates a new transaction scope.
2. The service transactionally receives the message.
3. Database Call 1 of 3: The shopping cart information is retrieved from the database.
4. Database Call 2 of 3: An Order record is inserted.
5. Database Call 3 of 3: *Deadlock!*
6. The exception causes the database transaction to roll back.
7. The transaction controlling the message also rolls back.
8. The order is back in the queue!

This is great news! The message is back in the queue, and by default, NServiceBus will automatically retry this message a few times. Generally, deadlocks are a temporary condition and simply trying again is all that is needed. After all, it says right there, in the SQL Server exception: "Rerun the transaction".

Meanwhile, the user has no idea there was ever a problem. It will just take a little while longer (in the order of milliseconds or seconds) to process the order.

## Error queues and replay

Any time you talk about automatic retry in a messaging environment, you must invariably consider **poison messages**. A poison message is a message that cannot be retried because it will result in an error every single time.

A **deadlock** is a transient error. We can reasonably expect deadlocks and other transient errors to resolve by themselves without any intervention.

Poison messages, on the other hand, cannot resolve themselves. Sometimes this is because of an extended outage. Other times it is purely our fault – an exception we didn't catch, or an input condition we didn't foresee.

## Automatic retries

If we retry poison messages in perpetuity, they will create a blockage in the message processing pipeline. They will retry over and over, and valid messages will get stuck behind them, unable to make it through.

For this reason, we must set a reasonable limit on retries, and after failing too many times, poison messages must be removed from the processing queue and stored someplace else.

NServiceBus handles all this for us, and then some. By default, NServiceBus will try to process a message five times, after which it will move the message into an error queue. This is the queue that we have seen named in the examples in the previous chapter, configured by the `MessageForwardingInCaseOfFaultConfig` configuration section:

```
<MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
```

It is in this error queue that messages will wait for administrative intervention. In fact, you can even specify a different server to collect these messages, which allows you to configure one central point in a system where you watch for and deal with all failures:

```
<MessageForwardingInCaseOfFaultConfig ErrorQueue="error@SERVER" />
```

As mentioned previously, five failed attempts is the default metric for a failed message, but this is configurable via the `TransportConfig` configuration section:

```
<section name="TransportConfig" type="NServiceBus.Config.  
TransportConfig, NServiceBus.Core" />  
...  
<TransportConfig MaxRetries="3" />
```



You could also generate the `TransportConfig` section using the PowerShell `Add-NServiceBusTransportConfig` cmdlet.

---

Keep in mind two things:

1. Depending upon how you read it, `MaxRetries` can be a somewhat confusing name. What it really means is the total number of tries, which has the odd side effect that `MaxRetries="0"` is the same as `MaxRetries="1"` — in both instances the message would be attempted once.
2. During development, you may want to limit retries to `MaxRetries="1"` so that a single error doesn't cause a nausea-inducing wall of red that flushes your console window's buffer, leaving you unable to scroll up to see what came before! You can then enable retries in production by deploying the endpoint with a different config.

## Replaying errors

What happens to those messages unlucky enough to fail so many times that they are unceremoniously dumped in an error queue? "I thought you said that Alfred would never give up on us!" you cry.

As it turns out, this is just a temporary holding pattern that enables the rest of the system to continue functioning while the errant messages await human intervention. Let's say our message handler divides two numbers from the incoming message, and we forget to account for the possibility that one of those numbers might be zero and that dividing by zero is frowned upon.

At this point, we need to fix the error somehow. Exactly what we do will depend upon your business requirements:

- If the messages were sent in an error, we can fix the code that was sending them. In this case, the messages in the error queue are junk and can be discarded.
- We can check the inputs on the message handler and detect the divide by a zero condition, and make compensating actions. This may mean returning from the message handler, effectively discarding any divide by zero messages that are processed, or it may mean doing new work or sending new messages. In this case we may want to replay the error messages after we have deployed the new code.
- We may want to fix both the sending and receiving side!

If we decide that we want to replay the messages after fixing the code, `NServiceBus` includes a tool to do this called **ReturnToSourceQueue.exe**. Assuming you installed `NServiceBus` with the Windows Installer, you can find this tool under the **NServiceBus Tools** folder in the **Start** menu.



Using the tool is pretty simple. Just run the program by double-clicking on it. It will ask for the name of the error queue to use, and then ask for the ID of the message you would like to replay, or type `all` to loop through all the messages in the error queue and return them all to their respective sources.

A new tool called ServiceInsight (currently in Beta at the time of this writing) is now available with NServiceBus 4.0 that gives you visibility into error queues across multiple systems, with return to source capability, testing, and MSMQ management. ServiceInsight can be downloaded from <http://particular.net/>.

## Second level retries

Automatically retrying error messages and sending repeated errors to an error queue is a pretty good strategy for managing both transient errors, such as deadlocks and poison messages like an all-out exception case. But as it turns out, there is a gray area in between best referred to as "semi-transient" errors. These include things like a web service being down for a few seconds, or a database being temporarily offline. Even with a SQL Server Failover Cluster, the failover procedure can take upwards of a minute depending on its size and traffic levels.

During a time like this, the automatic retries will be executed immediately and great swathes of messages might go to the error queue, requiring an administrator to take notice and return them to their source queues. But is this really necessary?

As it turns out, it is not. NServiceBus contains a feature called **Second Level Retries** (or **SLR** for short) that will retry the message additional times after a wait. By default, the SLR will retry three additional times, with an additional wait of 10 seconds each time.

Let's track a message's full path to complete failure:

- Attempt to process the message five times, then wait for 10 seconds
- Attempt to process the message five times, then wait for 20 seconds
- Attempt to process the message five times, then wait for 30 seconds
- Attempt to process the message five times, and then send the message to the error queue

Remember that using five retries, NServiceBus attempts to process the message five times on every pass.

Using second level retries, almost every message should be able to be processed unless it is definitely a poison message.

Be warned, however, that using SLR has its downsides too. The first is ignorance of transient errors. If an error never makes it to an error queue, and we never manually check the error logs, there's a chance we might miss it completely. For this reason, it is smart to always keep an eye on error logs. A random deadlock now and then is not a big deal, but if they happen all the time it is probably still worth some work to improve the code so that the deadlock is not as frequent.

An additional risk lies in the time to process a true poison message through all the retry levels. Not accounting for any time taken to process the message itself 20 times or to wait for other messages in the queue, the use of second level retries with the default settings includes an entire minute of waiting before you will ever see it in an error queue. If your business stakeholders require the message to either succeed or fail in 30 seconds, then you cannot possibly meet those requirements.

Luckily, second level retries are completely configurable. The configuration element is shown here with the default settings:

```
<section name="SecondLevelRetriesConfig"
  type="NServiceBus.Config.SecondLevelRetriesConfig,
  NServiceBus.Core"/>
...
<SecondLevelRetriesConfigEnabled="true" TimeIncrease="00:00:10"
  NumberOfRetries="3" />
```



You could also generate the `SecondLevelRetriesConfig` section using the PowerShell `Add-NServiceBusSecondLevelRetriesConfig` cmdlet.

Like first-level retries, keep in mind that you may want to disable second-level retries during development for convenience, and then enable them in production.

## RetryDemo

The sample solution, **RetryDemo**, included with this chapter demonstrates the basics of first and second level retries. Just type `GoBoom` into the console window when prompted and a message will be sent that will result in an error, and you can watch the retries as they happen, as shown in the following screenshot:

```
C:\Learning\NServiceBus\ch03\RetryDemo\RetryService\bin\Debug\NServiceBus.Host.exe
Type GoBoom to send a message that will cause an error.
GoBoom
Phase: First Level Processing, Try #1
Phase: First Level Processing, Try #2
Phase: First Level Processing, Try #3
Phase: First Level Processing, Try #4
Phase: First Level Processing, Try #5
Beginning SLR Round #1, 16 seconds since last attempt.
Phase: Second Level Retries Round #1, Try #1
Phase: Second Level Retries Round #1, Try #2
Phase: Second Level Retries Round #1, Try #3
Phase: Second Level Retries Round #1, Try #4
Phase: Second Level Retries Round #1, Try #5
Beginning SLR Round #2, 23 seconds since last attempt.
Phase: Second Level Retries Round #2, Try #1
Phase: Second Level Retries Round #2, Try #2
Phase: Second Level Retries Round #2, Try #3
Phase: Second Level Retries Round #2, Try #4
Phase: Second Level Retries Round #2, Try #5
Beginning SLR Round #3, 32 seconds since last attempt.
Phase: Second Level Retries Round #3, Try #1
Phase: Second Level Retries Round #3, Try #2
Phase: Second Level Retries Round #3, Try #3
Phase: Second Level Retries Round #3, Try #4
Phase: Second Level Retries Round #3, Try #5
2013-03-21 21:35:12.957 INServiceBus Dequeue Worker Thread for [retryservice.retries@rcode] - 181 ERROR NServiceBus.Management.Retries.SecondLevelRetries [(null)] <<null>> - SLR has failed to resolve the issue with message e81b6892-a195-44b6-b179-a18801635a4d and will be forwarded to the error queue at error@rcode
```

Play around with the settings in the service project's `App.config` file to see how they affect the output. Note that although the time delays between SLR rounds increase by 10 seconds by default, the actual observed delay is not guaranteed to be exactly accurate.

## Express messaging

One of the architectural principles that NServiceBus uses to guarantee fault tolerance is the concept of **store and forward**. This means that when a message is sent to a destination, it is first stored locally, and then the message queuing framework will attempt to send it to the destination queue until it is successfully delivered. This is what allows us to send a message to another server while that server is rebooting; it is stored on the sending server until the recipient is back online and ready to receive.

NServiceBus takes store and forward pretty seriously, using **Recoverable** messages in MSMQ by default to guarantee delivery. This means that the system does not consider the message as sent until it is physically saved to the disk and not in any disk cache.

However, there may be situations where the transient nature of a message's intent may make it advantageous to skip some of these safeguards. NServiceBus makes this possible through the use of the `ExpressAttribute`.

```
[Express]
public interface ISomeRapidlyChangingValueHasChanged : IEvent
{
    public double SomeValue { get; set; }
}
```

Decorating any message with this attribute causes MSMQ to use lazy writes to persist messages to disk and return immediately, which means they may be delivered faster, but it's also possible that they will never be persisted successfully to disk if there is a crash soon after! This will make your messages vulnerable to server failures; indeed, this is the point! If the message is no longer valuable unless it is delivered in a timely manner, then it makes no sense to forward potentially hundreds of messages once a server comes back online.

Generally, you will not need to use this attribute in most cases unless you have very specific business requirements. Do not be tempted to overuse this attribute for the promise of additional speed or throughput. The fault tolerance given to us by the store and forward pattern is much more valuable than speed, and as we will learn in *Chapter 7, Administration*, we have much better strategies for increasing message throughput.

## Messages that expire

Another consideration when considering potential failures are messages that lose their business value after a specific amount of time.

Consider a weather reporting system that reports the current temperature every few minutes. How long is that data meaningful? Nobody seems to care what the temperature was 2 hours ago, they want to know what the temperature is now!

NServiceBus provides a method to have messages automatically expire after a given amount of time. Unlike storing this information in a database, you don't have to run any batch jobs or take any other administrative action to ensure that old data is discarded. You simply mark the message with an expiration date and when that time arrives, the message simply evaporates into thin air.

```
[TimeToBeReceived("01:00:00")]
public class RecordCurrentTemperatureCmd : ICommand
{
    public double Temperature { get; set; }
}
```

This example shows that the message must be received within one hour of being sent, or it simply disappears. If a message with an expiration date is processed but then sent to an error queue, it will disappear out of the error queue as well! This can be very confusing when you observe logs that warn of an error, but cannot find the message that caused it.

Another valuable use for this attribute is for high-volume message types where a communication failure between servers or extended downtime could cause a huge backlog of messages to pile up either at the sending or receiving side. Running out of disk space to store messages is a show-stopper for most message queuing systems, and the `TimeToBeReceived` attribute is the way to guard against it.

## Auditing messages

At times, it can be difficult to debug a distributed system. Commands and events are sent all around, but after they are processed they go away. We may be able to tell what will happen to a system in the future by examining queued messages, but how can we analyze what happened in the past?

For this reason `NServiceBus` contains an auditing function that will enable an endpoint to send a copy of every message it successfully processes to a secondary location, a queue that is generally hosted on a separate server.

This is accomplished by adding an attribute or two to the `UnicastBusConfig` section of an endpoint's configuration:

```
<UnicastBusConfig ForwardReceivedMessagesTo="audit@SecondaryServer"
    TimeToBeReceivedOnForwardedMessages="1.00:00:00">
  <MessageEndpointMappings>
    <!-- Mappings go here -->
  </MessageEndpointMappings>
</UnicastBusConfig>
```

In this example, the endpoint will forward a copy of all successfully processed messages to a queue named `audit` on a server named `SecondaryServer`, and those messages will expire after one day.

While it is not required to use the `TimeToBeReceivedOnForwardedMessages` parameter, it is highly recommended. Otherwise it is possible (even likely) that messages will build up in your audit queue until MSMQ runs out of available storage, and unfortunately when that happens MSMQ basically shuts down and becomes unusable. We would really like to avoid that. The exact time limit you use is dependent upon the volume of messages in your system and how much storage MSMQ has available.

The best use for these audit messages is to fuel the new NServiceBus reporting system available in the ServiceInsight tool. It includes a backend service that consumes messages from your audit and error queues and then makes the data available via an API over HTTP. From there, the ServiceInsight tool can give you valuable insights into what is happening in your system.

## Web service integration and idempotence

When talking about managing failure, it's important to spend a few minutes discussing web services because they are such a special case; they are just too good at failing.

In the previous chapter we discussed only doing as much work within a message handler as you can reliably perform within the scope of one transaction. For database operations this limitation is obvious. For some other non-transactional operations (sending email comes to mind) it is easy to isolate that operation within its own message handler. When the message is processed the email will either be sent or it won't; there really aren't any in-between cases.



In reality, when sending an email it is technically possible that we could call the SMTP server, successfully send an email, and then the server could fail before we are able to finish marking the message as processed. However, in practice, this chance is so infinitesimal that we generally assume it to be zero. Even if it is not zero, we can generally assume, in most cases, that sending a user a duplicate email one time in a few million won't be the end of the world.

Web services are another story. There are just so many ways a web service can fail!

- A DNS or network failure may prevent us from contacting the remote web server at all
- The server may receive our request, but then throw an error before any state is modified on the server

- The server may receive our request and successfully process it, but a communication problem prevents us from receiving the 200 OK response
- The connection times out, thus ignoring any response the server may have been about to send us

For this reason, it makes our lives a lot easier if all the web services we ever have to deal with are **idempotent**, which means a process that can be invoked multiple times with no adverse effects.

Any service that queries data without modifying it is inherently idempotent. We don't have to worry about how many times we call a service if doing so doesn't change any data. Where we start to get into trouble is when we begin mutating state.

Sometimes we can modify state safely. Consider an example used previously regarding registering for alert notifications. Let's assume that on the first try, the third-party service technically succeeds in registering our user for alerts, but it took too long to do so and we receive a timeout error. When we retry, we ask to subscribe the email address to alerts again, and the web service call succeeds. What's the net effect? Either way, the user is subscribed for alerts. This web service satisfies idempotence.

The classic example of a non-idempotent web service is a credit card transaction processor. If the first attempt to authorize a credit card succeeded on the server and we retry, we may double charge our customer! This is not an acceptable business case and you will quickly find many people angry with you.

In these cases we need to do a little work ourselves, because unfortunately it's impossible for NServiceBus to know whether your web service is idempotent or not.

Generally this takes the form of recording each step we perform to durable storage as we perform it, and then query that storage to see which steps have been attempted.

In our credit processing example, the "straight through" approach would look like this:

1. Record our intent to make a web service call to durable storage.
2. Make the actual web service call.
3. Record the results of the web service call to durable storage.
4. Send commands or publish events with the results of the web service call.

---

Now, if the message is retried, we can inspect the durable storage and decide what step to jump to, and if any compensating actions need to be taken first.

If we have recorded our intent to call the web service but do not see any evidence of a response, we can query the credit card processor based on an order or transaction identifier. Then we will know if we need to retry the authorization or just get the results of the already complete authorization.

If we see that we have already made the web service call and received results, then we know that the web service call was successful but some exception happened before the resulting messages could be sent. In response, we can just take the results and send the messages, without requiring any further web service invocations.

The choice of durable storage to use for this process is up to you. If you choose to use a database, however, you must remember to exempt it from the message handler's ambient transaction, or those changes will also get rolled back if and when the handler fails!

In order to escape the transaction to write to durable storage, use a new `TransactionScope` to suppress the transaction like this:

```
public void Handle(CallNonIdempotentWebServiceCmd cmd)
{
    // Under control of ambient transaction

    using (TransactionScope ts = new TransactionScope(TransactionScopeOp
tion.Suppress))
    {
        // Not under transaction control
        // Write updates to durable storage here
        ts.Complete();
    }

    // Back under control of ambient transaction
}
```



## Summary

In this chapter, we considered the inevitable failure of our software, and how NServiceBus can help us to be prepared for it. We learned how NServiceBus promises fault tolerance within every message handler, so that messages are never dropped or forgotten, but instead, they are retried and then held in an error queue if they cannot be successfully processed. Once we fix the error, or take other administrative action, we can then replay those messages.

We learned that sometimes we must apply special conditions to messages in a system so that the system can recover from failure more gracefully. Specifically, we learned that messages carrying data that changes quickly can be marked as Express, which protects us from a backup of worthless transient messages during a failure, and how to expire messages that lose their business value after a specific amount of time.

Finally, we learned how to build auditing into a system by forwarding a copy of all messages for later inspection, and how to properly deal with the challenges involved in calling external web services.

In this chapter we dealt exclusively with NServiceBus endpoints hosted by the NServiceBus Host process. In the next chapter we will explore in detail how we can host NServiceBus in our own processes.

# 4

## Self-Hosting

We have already seen how simple NServiceBus makes it to turn a normal class library into a runnable messaging endpoint using the NServiceBus.Host process. The host process gives us many shortcuts that simplify setting up a messaging endpoint, and as we will see in *Chapter 7, Administration*, it even allows us to easily install an endpoint as a Windows service.

However, there are other host processes that we must be able to support. Chief among these are web applications, although it is possible to host NServiceBus within Windows Forms apps as well. We must be able to configure and host our messaging infrastructure within all these environments.

In this chapter we will learn how to use the NServiceBus fluent configuration to host a message bus within any application we want. This effort comes with a bonus: the fluent configuration API is the basis for customizations that we can make in endpoints hosted with the NServiceBus host process.

### Web app and custom hosting

NServiceBus draws a distinction between configuration that is managed by the system's developer, and the configuration that is managed by the system's administrator. The developer configuration options are exposed as a system of chained method calls and is compiled into the application. This leaves just a few configuration settings for the system administrator to determine in the `App.config` or `Web.config` files, such as message routing, error handling, and log levels, which allows these settings to be modified between development, integration, and production environments.

These chained method calls, referred to as **fluent configuration**, are what we use to configure NServiceBus within a custom host, such as a web or Windows Forms app.

We first touched on the fluent configuration methods in *Chapter 1, Getting on the IBus*, when we configured an MVC web application to send a message to a background service. At the time we took it on faith, but now we should take a look at it again in detail.

Here is the block of code we used to configure NServiceBus in our web application in *Chapter 1, Getting on the IBus*, and *Chapter 2, Messaging Patterns*, along with some comments to divide it up into some basic steps:

```
IBus Bus = Configure.With()           // Assembly Scanning
    .DefineEndpointName("ExampleWeb") // Endpoint Name
    .DefaultBuilder()                 // Dependency Injection
    .UseTransport<Msmq>()              // Message Transport
    .PurgeOnStartup(true)
    .UnicastBus()                     // Bus Options
    .CreateBus()                       // Startup
    .Start(() => Configure.Instance
        .ForInstallationOn<Windows>()
        .Install());
```

Now, let's take a look at these steps one by one.

## Assembly scanning

The `Configure.With()` declaration starts the process of scanning an application's assemblies. In order to do the things we have seen so far, such as automatically wiring up message handlers based on marker interfaces, NServiceBus needs to inspect all of the assemblies in the application to find those types. The empty `.With()` method is the default, and scans all of the application's assemblies. For a web application, this means the `bin` directory; otherwise, the application's base directory is used for scanning.

Sometimes we will run into an assembly that does not respond well to assembly scanning. Generally this is the case with a third-party assembly that references other assemblies that are not provided, making it impossible to fully scan types that have dependencies upon missing assemblies. In these cases it is necessary to ignore these assemblies, and this is easy to do in the fluent configuration:

```
Configure.With(AllAssemblies.Except("TroublesomeAssembly")
    .And("AnotherDifficultAssembly"))
```

Using this pattern we can use the `.And()` method to chain together as many difficult assemblies as we need to.

## Choosing an endpoint name

The endpoint name is very important, as it drives the names of all the queues that `NServiceBus` creates. In an endpoint hosted by the `NServiceBus.Host` process, the endpoint name is automatically determined by locating the `EndpointConfig` class (that is, the class that implements `IConfigureThisEndpoint`, which is automatically created for you when you include the `NServiceBus` NuGet package) and using the namespace of that class as the endpoint name. This is almost always the same as the Visual Studio project name, unless you have gone to lengths to change it.

When we are hosting `NServiceBus` on our own, there is no `EndpointConfig`, so instead `NServiceBus` will default to the namespace of the class calling `Configure.With()`.

If we really want to, however, we can take control and explicitly set the endpoint name. We have two options for doing so:

```
// Simple Method
.DefineEndpointName(string name)

// Lambda Method
.DefineEndpointName(Func<string> definesEndpointName)
```

The lambda method allows us a little more flexibility if we need it. For example, we could pull in an `appSetting`, which would allow us to host Test and QA versions of the same website on the same server, but with different endpoint names.

In the examples so far, we really haven't had the need to specifically control the endpoint name with this method. It is just included in the code samples to demonstrate its proper place very early in the fluent configuration process.

## Dependency injection

**Dependency injection** is a pattern that allows us to avoid hard-coded dependencies and define them at runtime. `NServiceBus` uses dependency injection to manage a host of dependencies, but the most visible one is the `IBus` dependency that we take in our message handlers. We just declare a public instance of an interface as a property, and at runtime `NServiceBus` will inject the runtime value that provides the implementation for that interface.

By default, `NServiceBus` will use the `Autofac` container for its own needs. When self-hosting, this is configured by the `.DefaultBuilder()` method. In most cases this is just fine, but if you already use a DI container and would like to integrate with it directly, you can do so using one of the following NuGet packages:

- NServiceBus.Autofac
- NServiceBus.CastleWindsor
- NServiceBus.Ninject
- NServiceBus.Spring
- NServiceBus.StructureMap
- NServiceBus.Unity

Each package will contain an extension method to allow you to use it, such as `.NinjectBuilder()` in the **NServiceBus.Ninject** package.

We will cover more topics related to dependency injection in *Chapter 5, Advanced Messaging*.

## Message transport

One of the big improvements in NServiceBus 4.0 is the addition of multiple message transport options. **Microsoft Message Queuing (MSMQ)** is the default transport and is used throughout this book, but several other transports are currently supported.

Declaring any of the transports when self-hosting follows the same pattern of using the `.UseTransport<TransportType>()` extension method, where `TransportType` is the transport you would like to use. MSMQ is built in to the NServiceBus core and is the default. To use MSMQ, therefore, we don't need to do anything, but if we would like to be explicit about it we can use `.UseTransport<Msmq>()`.

Using any of the other transports requires an additional NuGet package. Full details for the use of each alternative transport are beyond the scope for this book, but we will touch on them briefly. For more information, take a look at the samples that are included with the NServiceBus installer. There is a sample for each transport illustrating its use.

## Why use a different transport?

MSMQ has been part of NServiceBus since the beginning, as it is a solid technology that provides a lot of safeguards such as transactional store-and-forward and distributed transaction support. So why wouldn't we want to use it?

One consideration is platform independence and interoperability. As good as MSMQ is, it is a Microsoft technology that only runs on Windows servers. If we want to interoperate with a Java platform, for example, we don't have a lot of good options except exposing web services, which comes with its own set of problems.

Another consideration is resistance from IT departments. It can be difficult at times to get company IT approval for a system design that requires MSMQ and distributed transactions if such support wasn't already available before. Yet in these companies, Microsoft SQL Server is nearly ubiquitous. Using SQL Server as a transport can be a way for developers itching to take advantage of the benefits NServiceBus has to offer, get it in the door, and prove that it can be a valuable asset to the business.

The last common reason to consider a different transport is support for the cloud, where there may not be support for MSMQ or distributed transactions. We need to have different transports available to us to take advantage of those platforms.

## ActiveMQ

Apache ActiveMQ is an open source queuing system that implements the **Java Message Service (JMS)** and thus supports interoperability with Java. Like MSMQ, ActiveMQ supports the DTC which ensures that our message handlers remain fully transactional.

To use ActiveMQ as your transport, include the **NServiceBus.ActiveMQ** package and use the `.UseTransport<ActiveMQ>()` extension method when self-hosting, or if using the NServiceBus host, mark the `EndpointConfig` class with the `UsingTransport<ActiveMQ>` interface.

## RabbitMQ

RabbitMQ is an open source queuing system that implements the **Advanced Message Queuing Protocol (AMQP)**. Rabbit is a great solution for cloud platforms such as Amazon EC2, however it contains no support for the DTC. This means that some of the safeguards offered by NServiceBus and the DTC are not available, meaning you as developer have to take a larger role in maintaining fault tolerance and consistency. This involves manually ensuring that message handlers are idempotent, or handling de-duplication of messages because RabbitMQ does not guarantee once-and-only-once delivery as MSMQ does. To put it mildly, this is not for the faint of heart.

To use Rabbit as your transport, include the **NServiceBus.RabbitMQ** NuGet package and use the `.UseTransport<RabbitMQ>()` extension method when self-hosting, or if using the NServiceBus host, mark the `EndpointConfig` class with the `UsingTransport<RabbitMQ>` interface.

## SQL server

Since you are reading a book about a .NET technology, I can only assume you've heard of Microsoft SQL Server. Using SQL Server as a transport can be a great choice for small projects on teams that already use SQL Server. Because the messaging infrastructure is stored in the same database as your business data, distributed transactions are not needed to obtain fault tolerance. The performance of this transport (in message throughput per second) is on par with MSMQ, however because all endpoints will be querying a single SQL Server instance, this level of performance is shared between all endpoints in your system.

In addition to these benefits, the SQL Server transport will provide a nice way to get data in and out of legacy systems running on SQL Server using stored procedures, triggers, and the like.

To use SQL Server as your transport, include the **NServiceBus.SqlServer** NuGet package and use the `.UseTransport<SqlServer>()` extension method when self-hosting, or if using the NServiceBus host, mark the `EndpointConfig` class with the `UsingTransport<SqlServer>` interface.

## Windows Azure

The Windows Azure transport enables NServiceBus to use either Windows Azure Queues or Windows Azure ServiceBus as the NServiceBus transport, either entirely within the cloud or in a hybrid scenario, where an on-premise endpoint uses Windows Azure to communicate with endpoints in the cloud or with other on-premise endpoints.

Unlike the other transports, the Windows Azure transport is not included in the main NServiceBus installer. Downloads and documentation for it can be found on the Particular Software website.

## Purging the queue on startup

Self-hosting NServiceBus is usually done in a web application or a smart client. Because of this, the messages received when self-hosting (if any) are usually events, and the intent of those events is usually something along the lines of "something has happened on the server, so the data you are holding in cache is now invalid."

If a web application or smart client is offline for a period of time, as is the case when IIS decides to spin down an idle website, a backlog of incoming messages may pile up, and it's quite possible that all of those messages will be instructions to remove items from a cache that is already empty because the application has just started up. So why would we want to bother processing these messages?

To avoid this situation, `NServiceBus` offers the option to purge the input queue when the endpoint starts up, as follows:

```
.PurgeOnStartup(true)
```

## Bus options

The bus options include the type of bus and a few options that control its operation. This is also where we transition from a configuration phase to actual startup, as some of the heavy lifting to wire up all the bus components happens behind the scenes here, now that the brunt of our configuration choices have been made as mentioned earlier.

`NServiceBus` includes only one bus type, `UnicastBus`. Unicast means sending messages to a single destination identified by an address, which is what `NServiceBus` does. This may seem confusing, since published event messages are sent to multiple addresses, but this is simply unicast in the plural.

Since there are no other options, use the `.UnicastBus()` extension method when self-hosting. This method returns a different type in the fluent config chain, making several methods available allowing you to fine-tune certain aspects in ways that are not common, so we will not detail them here. You can find out more by looking at the methods on the `ConfigUnicastBus` class that are returned by `.UnicastBus()`.

One thing that is a worthwhile note for developers familiar with previous versions of `NServiceBus` is that the `LoadMessageHandlers` method, which used to be required in order to actually receive and process messages, is no longer required in `NServiceBus 4.0`. Message handlers are loaded by default, so this method is only needed if you wish to control the handler processing order.

It is possible to create a self-hosted endpoint that is truly a send-only endpoint, that does not process any messages and requires absolutely no queues. We will cover this in the *Send-only endpoints* section in this chapter.

## Startup

With all of the bus settings configured, it's now time to create and start the bus. This is also the time when we will have the opportunity to run startup actions, such as having our queues automatically created.

Going back to our example code, we had:

```
.CreateBus()  
.Start(() => Configure.Instance  
    .ForInstallationOn<Windows>()  
    .Install());
```



The `.CreateBus()` extension method begins the finalization of our configuration by returning an `IStartableBus` instance instead of a `Configure` instance like all of the methods before, which makes it impossible to call more configuration commands after the `.CreateBus()` method.

An `IStartableBus` instance is essentially an `IBus` instance that can be started or stopped, and when we start it, we can optionally pass in an `Action` delegate to perform whatever startup actions will be needed.

The `Configure.Instance.ForInstallationOn<Windows>().Install()` command in the action again takes advantage of assembly scanning to find any classes that implement either the `INeedToInstallSomething` or `INeedToInstallSomething<Windows>` interfaces. By default, this means creating the appropriate queues, but this also means that we can implement the same interfaces to create our own installation actions if we so choose.

## Send-only endpoints

At some point we may create an application that only sends messages to backend services. This is commonly the case with web applications where no message handlers exist. In this case, there is no reason to create a bunch of queues, or waste processing time checking those queues for incoming messages, as we already know there will be none. And if there are no queues, then we don't have any need for an endpoint name either.

In this case we can use a drastically shorter version of the fluent configuration block that skips all these things that we don't need.

```
IBus Bus = Configure.With()  
    .DefaultBuilder()  
    .UseTransport<Msmq>()  
    .UnicastBus()  
    .SendOnly();
```

We still included the transport declaration, but as we learned before, even that would be optional if we are content with MSMQ.

---

This block still includes the bare essentials of defining assembly scanning, dependency injection, transport, and bus type, but short circuits everything else by calling the `.SendOnly()` method which instantly returns the `IBus` instance.

We don't even need an error queue for a send-only endpoint. Indeed, the only other configuration we need is where to send messages, as we discussed in the Message Routing section of *Chapter 2, Messaging Patterns*.

If we look back at our example from *Chapter 1, Getting on the IBus*, we can now see that the web application there would be a perfect candidate for a send-only endpoint. The only reason it was written as it was, was to demonstrate the simple addition of message handlers in *Chapter 2, Messaging Patterns*, when we began processing events in the web application.



[ For an example, check out the `SendOnlyExample` solution included with this chapter's sample code. It is a copy of the code sample from *Chapter 1, Getting on the IBus*, modified to be a send-only endpoint. ]

## Summary

In this chapter we dissected the process of self-hosting an `NServiceBus` endpoint in our own application instead of using the `NServiceBus Host`. There are certainly a lot of options, so you can see why we glossed over it when we were just getting started in *Chapter 1., Getting on the IBus*

Lastly we took a look at send-only endpoints, which enable us to self-host an endpoint much more simply when all we need to do is send messages.

The fluent configuration methods we saw here form the backbone of the extensible architecture of `NServiceBus`, and are used to customize the `NServiceBus` host as well. In the next chapter we will learn how to harness this capability in endpoints hosted by the `NServiceBus Host` as we explore several advanced messaging techniques.



# 5

## Advanced Messaging

Now that we have learned how NServiceBus works from the ground up through our discussion of self-hosting, we can take a deeper look at the NServiceBus host, where messaging is allowed to reach its full potential.

First we will take a look at the extensibility provided by the NServiceBus host process through its extension points and dependency injection. Then we will learn a variety of advanced messaging techniques that we can use to bend NServiceBus to our will.

### Modifying the NServiceBus host

The same extension methods that configure NServiceBus when we are self-hosting can also be used to configure the NServiceBus host. In fact there are many more extension methods than we have covered so far, and some additional independent settings, but before we learn about these, we need to first learn about the host process's extension points and where we can use them.

### General extension points

NServiceBus allows you to configure the host process by implementing specific interfaces that it will find when performing assembly scanning on startup. These interfaces specify a method (usually `Init()` or `Run()` depending on the interface) where you can hook into the configuration system with `NServiceBus.Configure`, or assuming the NServiceBus namespace is already referenced in a `using` declaration, simply `Configure`. For example, to change the message serializer (which we will cover a little later in this chapter) from the default XML to use JSON instead:

```
public class SetupSerializer : INeedInitialization
{
    public void Init()
    {
```

```
        Configure.Serialization.Json();
    }
}
```

`INeedInitialization` is just one extension point. Let's take a look at all of the general extension interfaces, in the order in which they're executed.

- `IWantCustomLogging` allows you to modify the existing logging or specify your own. Because logging is a prerequisite for everything else, this is executed first, but its position early in the process also means that nothing else has been configured yet. So while you may be thinking of configuring something besides logging here, that is probably not the best idea. Note that unlike the rest of the extension interfaces, you must implement this interface on the class that implements `IConfigureThisEndpoint`.
- `IWantToRunBeforeConfiguration` allows you to configure settings before the assembly scanning phase. The primary purpose for this extension point is for unobtrusive mode conventions, which will be covered later in this chapter. Another common use case is to define complex logic to set the endpoint name dynamically, since the endpoint name must also be known early in the configuration process.
- `IWantCustomInitialization` should be implemented only on the class that implements `IConfigureThisEndpoint` and allows you to perform customizations that are unique to that endpoint.
- `INeedInitialization` is similar to `IWantCustomInitialization`, except that it can be implemented by any class and then distributed to multiple endpoints. This runs just after `IWantCustomInitialization` so that the `EndpointConfig` class can register settings that are unique to it first. `INeedInitialization` is the best place to perform common customizations such as setting a custom dependency injection container, or changing the message serializer.
- `IWantToRunBeforeConfigurationIsFinalized` allows you to implement logic just before the configuration is completed. This is a good place to bring together combinations of settings that are dependent upon other settings established during `INeedInitialization`.
- `IWantToRunWhenConfigurationIsComplete` allows you to implement logic just after the configuration has been completed. The main difference in having a completed configuration is that the IoC container is fully started at this point, so classes implementing this interface have full access to full dependency injection.

- `IWantToRunWhenBusStartsAndStops` is the last general extension point. It is unique in that it has `Start()` and `Stop()` methods, allowing you to perform cleanup operations when the endpoint stops, although it is not invoked for send-only endpoints. Classes that implement this interface also benefit from full dependency injection. Note that in previous versions of `NServiceBus` this interface was called `IWantToRunAtStartup` so you will likely see lots of examples on the Internet using this older name.

## Dependency injection

Having all these extension points is nice, but what is it that we will really do there? We can use any of the extension methods introduced in *Chapter 4, Self-Hosting*, to modify how the Bus operates, but we can also make our own modifications. This will usually involve dependency injection.

Let's say that we want to be able to create unit tests to verify how our handlers react to time. Testing with code that is time dependent is difficult since the time is always changing. So instead of using `DateTime.Now` or `DateTime.UtcNow` directly, we'll create an interface to abstract away the implementation, and an implementation class that will provide the true time when we're not running a test.

```
public interface ITimeProvider
{
    DateTime Now { get; }
    DateTime UtcNow { get; }
}

public class DateTimeProvider : ITimeProvider
{
    public DateTime Now
    {
        get { return DateTime.Now; }
    }

    public DateTime UtcNow
    {
        get { return DateTime.UtcNow; }
    }
}
```

We will create another class called `MockTimeProvider` that will allow us to adjust the meaning of `Now` in the middle of a test, but we'll ignore that for now. At the moment we are concerned with how to get our `DateTimeProvider` injected into our message handler classes so that we can use it.

To configure our `DateTimeProvider` into the dependency injection container, we call one of the variants of `Configure.Component()`.

```
Configure.Component<DateTimeProvider>(DependencyLifecycle.  
    SingleInstance);
```

This will look at the `DateTimeProvider` class to determine what services it provides, or in other words, what interfaces it implements. In this case it implements `ITimeProvider`, so it registers that a `DateTimeProvider` object should be returned whenever an `ITimeProvider` parameter is requested.

The `DependencyLifecycle.SingleInstance` enumeration member specifies that only one instance of `DateTimeProvider` will ever be created, and it will be returned on every request for `ITimeProvider`. The other choices for `DependencyLifecycle` are `InstancePerCall`, which specifies that a new `DateTimeProvider` will be instantiated for every request, and `InstancePerUnitOfWork`, which means that a new `DateTimeProvider` will be instantiated for each unit of work. We will cover units of work later in this chapter. We are using `SingleInstance` because our implementation is inherently thread-safe and there's really no reason to create multiple objects.

There are additional overloads for the `Configure.Component` method which accept factory delegates that allow us to specify how to construct the object being injected, but these are generally only used in advanced scenarios.

All of the `Configure.Component` methods return an `IComponentConfig` instance which allows you to instruct the container to set properties on the injected objects.

```
string connStr = "Your DB Connection String";  
Configure.Component<DataStore>(DependencyLifecycle.InstancePerCall)  
    .ConfigureProperty(ds => ds.ConnectionString, connStr);
```

This takes an expression which points to a property on the object, and a value to fill it with. This way your `DataStore` object is injected with the `ConnectionString` property already filled in. You can daisy-chain as many calls to the `ConfigureProperty` method as you need using the fluent style.

Using these methods, you can abstract out a lot of the services your message handlers will need to do their job, resulting in code that is more maintainable and more testable. You will also be able to easily swap out different implementations depending on the environment you are in, such as `Dev`, `Test`, `QA`, or `Production`. We will explore how to do this in more detail in *Chapter 7, Administration*.

## Additional bus settings

At this point, there are a few more service bus settings that we should cover. These are similar to the fluent configuration options but are order agnostic – you can call them at any point before the bus is created. When self-hosting, they would go any place before the `Configure.With()` block, and if you're modifying the `NServiceBus` host, the `INeedInitialization` interface is a good extension point to choose.

## Message serializer

As we have already seen, our `NServiceBus` messages are plain old C# classes or interfaces, but these must be serialized in some way to be transmitted using the underlying queuing infrastructure.

The default choice for a message serializer is XML, which requires no configuration whatsoever, unless you want to set some advanced options on the serializer itself. However, `NServiceBus` also supports JSON, BSON, and Binary serialization out of the box. To make the switch, execute one of these methods before `Configure.With()` or during `INeedInitialization`:

```
Configure.Serialization.Json();
Configure.Serialization.Bson();
Configure.Serialization.Binary();
```

Or, if you have a need to customize the XML serializer, use this method and provide an action to configure the settings you need.

```
Configure.Serialization.Xml(
    Action<XmlSerializationSettings> setCustomSettings);
```

It's worth noting that the XML serializer used by `NServiceBus` is not the same as the built-in .NET XML serializer. There are no attributes to control the output of the XML, and the `KnownTypesAttribute` for representing polymorphic data structures is not supported. In addition, certain types that you might expect to be serializable are not supported.

Why the difference? The .NET serializer is primarily concerned with flexibility and the ability to tailor the output. The `NServiceBus` XML serialization, on the other hand, is primarily concerned with speed.

Message contracts should be very specific and concrete. Message properties cannot be interface types like `ICollection`, `ISet`, or `IDictionary` because there would be no way to instantiate those properties. Similarly, message property types cannot be abstract base classes because there is no way to guarantee that the endpoint receiving the message will know about all the possible inherited types.





## Transaction settings

By default, NServiceBus creates a new transaction scope with the `ReadCommitted` isolation level. If, for example, you would like to change that to `Serializable`, add the following before `Configure.With()` or during `INeedInitialization`:

```
Configure.Transactions.Advanced(x =>
    x.IsolationLevel(IsolationLevel.Serializable));
```

On the other hand, if you would like to disable transactions entirely, do this instead:

```
Configure.Transactions.Disable();
```

## The unobtrusive mode

We have seen how using events can help us to decouple business processes from each other, which is good because we know that taking unnecessary dependencies is a "Bad Thing".

But throughout this book, we have been marking our commands with `ICommand` and our events with `IEvent`. This introduces a dependency on the `NServiceBus.dll` assembly that contains those interfaces; isn't that a bad thing?

As it turns out, it can be. When you create your messages assembly, you compile it against a specific version of `NServiceBus.dll`. Then another service consumes that assembly. Now if you want to update one service to a new version of `NServiceBus`, you have a problem. You have to update both services at once! This is not the glorious decoupled autonomous service utopia we signed up for.

Luckily there is another way to identify our commands and events that does not rely on marker interfaces. The only reason we did things this way in the previous chapters was to make the examples easier to follow while we learned some of the basics.

**Unobtrusive mode** is the capability to identify messages by convention instead of by marker interfaces.

```
Configure.With()
    .DefiningCommandsAs(Func<Type, bool> definesCommands)
    .DefiningEventsAs(Func<Type, bool> definesEvents)
    .DefiningMessagesAs(Func<Type, bool> definesMessages)
```

Each method accepts a `Func<Type, bool>` which allows you to programmatically define which types will be viewed as commands, events, or messages. Most of the time, your conventions should be based on the namespace of the type. If you noticed, the examples in this book have been careful to create folders for commands and events, which means that our message namespaces will follow predictable patterns that we can use to specify our conventions.

While you can define any convention you want, here is a good strategy to start with:

- All message types must have non-null namespaces.
- If your company uses a single top-level namespace, all message types must have a namespace starting with that string. This ensures that your convention won't accidentally apply to classes in any third-party libraries.
- If you do not have a single top-level namespace, such as some of the examples in this book, then at least disqualify any namespace that starts with `NServiceBus` or `System`. You may need to add to this list if you use any third-party libraries that you discover accidentally fit the rest of your conventions.
- Define commands as any type whose namespace ends with `Commands`.
- Define events as any type whose namespace ends with `Events` or `Contracts`.
- Define messages as any type whose namespace ends with `InternalMessages`, or if you prefer, simply `Messages`.

Unfortunately the code to represent this doesn't present very well in book form, so check out all of the code samples for this chapter. Each one defines its conventions in a class called `MessageConventions.cs`, complete with comments to describe why the conventions were chosen.



You may be confused about why there is a separate convention for messages. Aren't all messages either commands or events? As it turns out, there are some messages that are neither, such as reply messages, which we will cover in *Chapter 6, Sagas*.

There are additional conventions to define for other topics we have already covered in *Chapter 3, Preparing for Failure*; the `Express` attribute and the `TimeToBeReceived` attribute. After all, what good is unobtrusive mode if we would still need to reference `NServiceBus.dll` to use these two attributes?

We can create a convention for these attributes like this:

```
.DefiningExpressMessagesAs(t => t.Name.EndsWith("Express"))
.DefiningTimeToBeReceivedAs(t => t.Name.EndsWith("Expires"))
```

```
? TimeSpan.FromSeconds(30)
: TimeSpan.MaxValue)
```

The convention for the `Express` attribute should be suitable for all cases. However the one presented to replace `TimeToBeReceived` has some room for a bit of improvement, because a single duration of 30 seconds may not be appropriate for all instances. Take a look at the **ConventionsSample** project included with this chapter for an example of a novel way to tackle this problem.

All of the remaining examples in this book will use unobtrusive mode, and it is highly recommended as a best practice for all of your `NServiceBus` systems.



Rumor has it that on rare occasions, we developers sometimes screw things up. In an extremely unlikely event that this happens to you while establishing your unobtrusive mode conventions, remember that the `NServiceBus` host outputs the number of message types detected to the console on startup. This can be a helpful way to verify that our conventions are picking up all the types we think they should.

For more detailed message type information, you can interrogate the `MessageMetadataRegistry` type, which the dependency injection container will supply if you ask for it. This technique is demonstrated in the **ConventionsSample** included with this chapter.

## Message versioning

If you've ever had to build and then maintain a web service for any significant amount of time, you probably dealt with the struggles that versioning those systems can have. Adding a new web method is easy enough, but what happens when the business wants to take the `DoSomethingSpecial` web method and add a new parameter to it? Many times this results in frustrated developers creating a `DoSomethingSpecial2` method, and then old methods can never be cleaned up because it may be impossible to ensure that external clients have updated their codebase. Over time these things build up and result in a very messy API that is very difficult for a newcomer to decipher or support.

## Polymorphic dispatch

So how do you support versioning in a message-based system? Let's take another look at an example from *Chapter 2, Messaging Patterns*, the event that announced a user had been created. Of course, now that we are using unobtrusive mode, we have taken off the `IEvent` marker interface.

```
public interface IUserCreatedEvent
```

```

{
    Guid UserId { get; set; }
    string Name { get; set; }
    string EmailAddress { get; set; }
}

```

What's missing from this event? A common requirement in an event is some sort of timestamp. After all, messages are not guaranteed to arrive in order, so if two events are telling us the current price of bananas, without a timestamp how would we know which one was correct?

To add a timestamp to a message without breaking any current subscribers, we can inherit from the original message.

```

public interface IUserCreatedWithTimeEvent : IUserCreatedEvent
{
    DateTime Timestamp { get; set; }
}

```

Now instead of publishing an `IUserCreatedEvent`, we instead publish an `IUserCreatedWithTimeEvent`, and we can have handlers for both message types:

```

// Publisher
Bus.Publish<IUserCreatedWithTimeEvent>(e =>
{
    // Set properties of e
}

// Newer Subscriber
public class UserHandler : IHandleMessages<IUserCreatedWithTimeEvent>
{
    public void Handle(IUserCreatedWithTimeEvent e) { }
}

// Older Subscriber
public class OlderHandler : IHandleMessages<IUserCreatedEvent>
{
    public void Handle(IUserCreatedEvent e) { }
}

```

This works because of **polymorphic dispatch**. When we publish the `IUserCreatedWithTimeEvent`, `NServiceBus` says "Here is a message with properties called `UserId`, `Name`, `EmailAddress`, and `Timestamp`. It is both an `IUserCreatedEvent` and an `IUserCreatedWithTimeEvent`". It will then publish the message to any subscriber who has subscribed to receive either type of event.

We publish one event, and both subscribers get the information they expect to receive. This means that we can upgrade our publisher first, and then worry about upgrading any subscribers later. This gives us the flexibility to upgrade our system component-by-component without having to bring the whole system down.

Polymorphic dispatch enables additional capabilities as well. Consider the following event definition:

```
public interface ICorporateUserCreatedEvent : IUserCreatedEvent
{
    public int CompanyId { get; set; }
}
```

Now we can publish `IUserCreatedEvent` for normal users, which will only invoke the handlers specifically for that event type. For corporate users, we can publish an `ICorporateUserCreatedEvent`, and the message handlers for both the event types will be invoked! This enables us to hook additional functionality to the corporate user that is dependent upon `CompanyId`. When writing these types of polymorphic message handlers, we can package multiple handlers together in one message endpoint. We just need to be mindful that in this situation, two handlers may try to edit the same entity within the same transaction, so it is very helpful if our data persistence technology supports the **Identity Map** pattern so that the two edits of the same entity would be treated as one operation.

## Events as interfaces

Message versioning using inheritance makes a lot of sense. V2 inherits from V1, V3 inherits from V2, and so on. But polymorphic dispatch enables some other interesting scenarios that would never be possible in a traditional web service or RPC system.

Because we have been using interfaces to represent our events, we can take advantage of the fact that interfaces, unlike classes, support multiple inheritance.

Suppose that in addition to our `IUserCreatedEvent`, we also had an `IUserLoggedInEvent`, and because you had to verify your email address after your user was created and before you could log in, these never happened at the same time. Now suppose that we are going to integrate with our corporate system where all email addresses were already verified, and that we need to support single sign-on between these two systems. So if a user comes from the corporate site, they are automatically created and logged in.

This scenario would lead to these message definitions, with the properties removed for brevity:

```
public interface IUserCreatedEvent { }
```

```
public interface IUserLoggedInEvent { }

public interface IUserCreatedBySingleSignInEvent : IUserCreatedEvent,
    IUserLoggedInEvent { }
```

When we publish `IUserCreatedBySingleSignInEvent`, it will be received by subscribers of `IUserCreatedEvent` and `IUserLoggedInEvent`, in addition to endpoints that subscribe to `IUserCreatedBySingleSignInEvent` directly.


## Specifying the handler order

Because of polymorphic dispatch, it's possible that there could be multiple handlers within an endpoint that all act upon the same message, because of the inheritance chain for that message. In addition, it's possible to deploy several handlers for the very same type to the same endpoint, although as each handler adds to the amount of work that must be done within a transaction, this isn't always the best idea.

In these cases, all relevant handlers for a message are organized into a pipeline. One transaction is created, and then each handler is executed for that message, one after the other. There is no guarantee what order the handlers will run in, as the handlers should all do their work autonomously.

However, this is not always the case, especially when it comes to security concerns. Sometimes there will be a handler that must run first whose job is to determine if the message is authorized to run and then shut down the message pipeline if it is not via a call to `Bus.DoNotContinueDispatchingCurrentMessageToHandlers()`, which is a mouthful, but gets the point across.

This method is very descriptive but won't do any good if it's in the handler that's run last! For that reason, an API exists to predetermine the handler ordering within the message processing pipeline.

 One common mistake is to refer to this process as "message ordering" when this could not be further from the truth. In a distributed system, messages may arrive in any order, and nothing can change that. This process sets the message handler ordering – the order that the handlers for a given message will be executed.

To set the message handler ordering when using the `NServiceBus` host, implement the `ISpecifyMessageHandlerOrdering` on your `EndpointConfig` class:

```
public class EndpointConfig : ISpecifyMessageHandlerOrdering
{
    public void SpecifyOrder(Order order)
```

```
{
    // First option, if we have only one handler like message
    // authorization that must go first
    order.SpecifyFirst<ImportantHandler>();

    // Second option, if we need to specify multiple orderings
    order.Specify(First<FirstHandler>
        .Then<SecondHandler>()
        .AndThen<ThirdHandler>()
        .AndThen<AndSoOnHandler>()
    );
}
```

The `.AndThen<THandler>()` method can be called as many times as is necessary, however take care not to overuse the message handler ordering. It is primarily only for security concerns that must run first. If you find you are leaning on it to get your business logic to run correctly, then your message handlers are not as autonomous as they should be.

While a common place to implement the `ISpecifyMessageHandlerOrdering` interface is on the `EndpointConfig` class, this is not required. Any class can implement this method; `EndpointConfig` is just a convenient place to look for it.

The previously mentioned use case of authorization checking is usually implemented in a separate assembly that is commonly said to be part of the IT/Ops service, and is deployed to many endpoints to handle security throughout the system. In this case, the assembly distributed by IT/Ops would contain both the class implementing `ISpecifyMessageHandlerOrdering` as well as the handlers to be run to authorize the messages.

## Message actions

As noted in the previous section, sometimes we will do something in a message handler relating directly to the messaging infrastructure, independent of business logic or data access.

## Stopping a message

As mentioned previously, sometimes we need to stop a message not just in the current handler, but stop all pending handlers from executing as well, normally because of a message authorization scheme. In order to do that, we simply call:

```
IBus.DoNotContinueDispatchingCurrentMessageToHandlers();
```

## Deferring a message

Sometimes we cannot process a message right now and need to wait just a little bit. We can instruct the bus to put the message back on the queue, essentially moving it to the back of the line.

```
IBus.HandleCurrentMessageLater();
```

However, if the queue is currently empty, we will just wind up processing this message again on the next go round, so be careful trying to implement a fairness scheme this way. There is no way to find out how many times a message has been deferred with this method, so it would be easy to get stuck in a loop processing the same message over and over.

A more useful type of deferral is a timed deferral, commonly necessary when calling external web services that have rate limits. In this case, sending the message to the back of the line may not be long enough, so we instruct the bus to defer a message for a specific amount of time.

```
IBus.Defer(DateTime processAt, params object[] messages);  
IBus.Defer(TimeSpan delay, params object[] messages);
```

## Forwarding messages

In *Chapter 3, Preparing for Failure*, we covered message auditing, which sends a copy of all messages received by an endpoint to another address. Sometimes, however, you would like a little more fine-grained control over that, especially if you are only trying to debug a problem with one specific message type.

To forward a message programmatically, call this method:

```
IBus.ForwardCurrentMessageTo(string destination);
```

It is important to note that this only forwards the message – message processing will still continue. Therefore if you are trying to move message handlers from one endpoint to another and are trying to use forwarding to redirect that message, you also may need to call `Bus.DoNotContinueDispatchingCurrentMessageToHandlers()`. This message will still be treated as a successfully processed message and will also end up in the audit queue.

## Message headers

Another common duty of the IT/Ops service is dealing with message metadata (data not directly related to the business purpose of the message) in the message headers.



This allows handling certain tasks at an infrastructure level without having to always remember to put certain properties in commands and events.

The API for dealing with message headers is fairly simple.

```
string IBus.GetMessageHeader(object msg, string key);
void IBus.SetMessageHeader(object msg, string key, string value);
```

In addition to getting and setting your own headers, `NServiceBus` includes several headers for its own purposes, and these can be quite informative and educational. For easy access, the header names used by `NServiceBus` are available as constant strings on the `NServiceBus.Headers` class.

Message headers should only be used for infrastructure purposes, and never for business data. Don't be tempted to use message headers just because two message contracts share a similar property. A good message header use case would be information to authenticate, authorize, or sign a message.

## Unit of work

Because messages are processed in a pipeline of handlers, we will at times have a need to execute code before the first handler and after the last handler. This is necessary for data stores that follow the unit of work pattern such as `NHibernate` or `RavenDB`, where we need to create a database session before handling a message, and then commit or rollback any changes at its completion.

In order to define a unit of work implementation, first you must implement the `NServiceBus.UnitOfWork.IManageUnitsOfWork` interface. Here we have an example that will simply write messages to the console:

```
public class ConsoleUnitOfWork : IManageUnitsOfWork
{
    public void Begin()
    {
        Console.WriteLine("---Begin message---");
    }

    public void End(Exception ex = null)
    {
        Console.WriteLine("---End message---");
    }
}
```

When talking about a unit of work, it's important to realize it's possible to have more than one `NServiceBus` message bundled up in a single transport message, although I won't recommend doing so. A **transport message** is the message as it is represented by the underlying queuing technology, that is, one transport message is the same as one MSMQ message, which may contain multiple `NServiceBus` messages if an array of messages is passed to `Bus.Send()`.

In this case, the `UnitOfWork` execution and the message handler's ambient transaction wraps the entire transport message, not each individual message it contains. It is precisely for this reason that multi-message transport messages are not recommended; they force transactions to grow too large resulting in nasty side effects.

`UnitOfWork` implementations are not automatically invoked; you need to specifically instruct the dependency injection container to use them.

```
public class ConfigureUOW : INeedInitialization
{
    public void Init()
    {
        Configure.Component<ConsoleUnitOfWork>(
            DependencyLifecycle.InstancePerCall);
    }
}
```

While it may seem odd that we need to wire this up ourselves given how much we have seen `NServiceBus` wire up for us already, we need to control this ourselves. Ordering in `UnitOfWork` implementations can be very important, so we can't leave it up to any assembly scanning magic. We need to define the handlers in an order that makes sense.

## Message mutators

**Message mutators** provide the ability to modify a message either on the way into or out of a message handler. There are two different categories of message mutators, applicative and transport message mutators.

An **applicative message mutator** is used to act on individual messages. They are the ideal place to perform tasks such as validation. `NServiceBus` uses this type of mutator internally to implement property encryption and `DataBus` serialization, which we will cover later in this chapter. Access to the message is provided as an object, so usually some amount of reflection is required in order to implement anything of substance.

- Implement `IMutateOutgoingMessages` to manipulate messages being sent from the endpoint
- Implement `IMutateIncomingMessages` to manipulate messages being received by the endpoint
- Implement `IMessageMutator`, which itself implements both of the interfaces above, to manipulate both incoming and outgoing messages

A **transport message mutator** is used to act on a transport message, which may contain one or more application messages, depending on your choice of transport. A transport message mutator gets you down closer to the bare metal, even providing you with access to the transport message's byte array that represents the transmitted contents of the message. A transport message mutator is the perfect place to implement functionality such as full-message encryption, signing, header manipulation, or compression.

- Implement `IMutateOutgoingTransportMessages` to manipulate transport messages being sent from the endpoint
- Implement `IMutateIncomingTransportMessages` to manipulate transport messages being received by the endpoint
- Implement `IMutateTransportMessages`, which implements both of the aforementioned interfaces, to manipulate both incoming and outgoing transport messages

Message mutators are not automatically configured by the dependency injection container for the same reason as a `UnitOfWork` implementations, so you must register them yourself.

```
Configure.Component<MyMessageMutator>(
    DependencyLifecycle.InstancePerCall);
```

## Property encryption

If you want to encrypt an entire message, a message mutator is your best bet. However if you only want to encrypt certain properties within a message, you have another option:

```
public class MessageWithASecretCmd : ICommand
{
    public string ClearText { get; set; }
```

```
    public WireEncryptedString SecretText { get; set; }
}
```

Or if you'd like to use unobtrusive mode conventions:

```
public class MessageWithASecretCmd
{
    public string ClearText { get; set; }
    public string SecretTextEncrypted { get; set; }
}

// Convention Definition
Configure.Instance
    .DefiningEncryptedPropertiesAs(pi =>
        pi.Name.EndsWith("Encrypted"));
```

In order to use property encryption, we will also need to enable it for the endpoint and configure the encryption key in the `App.config` or `Web.config` file:

```
// Endpoint configuration
Configure.Instance.RijndaelEncryptionService();

// Configuration Section
<section name="RijndaelEncryptionServiceConfig"
    type="NServiceBus.Config.RijndaelEncryptionServiceConfig,
        NServiceBus.Core"/>

// Configuration Element
<RijndaelEncryptionServiceConfig
    Key="rq9D+KPPKkK/tMe+mLW6K5YoiydmNI9C"/>
```

Of course, the encrypted data is only as safe as you keep the key, so storing the encryption key in the config file of every single endpoint is probably not the best idea. For this reason, it would be a good idea to store this value in a centralized location and then provide it to each endpoint via a custom configuration provider. We will learn how to do this in *Chapter 7, Administration*.

## Transporting large payloads

When using any message queuing system, you will discover that very large messages are not a good idea. With MSMQ, there is a hard limit of 4 MB per message.

This may seem like a lot, but consider the situation we mentioned in *Chapter 1, Getting on the IBus*, where we're processing images for our clients. You might be able to squeeze most images into an MSMQ message, but you shouldn't bank on it. These days, 12-megapixel cameras that create 2.5 MB JPEG images are fairly commonplace, and once the message serializer Base64 encodes the byte array into the message, you're looking at 3.3 MB. That's way too close for comfort considering the average number of megapixels has nowhere to go but up. Now consider the 256 KB limit on Windows Service Bus Queues, or the 64 KB limit on Windows Azure Queues!

Other queuing systems don't have a hard limit, but that doesn't mean creating huge messages is a good idea! Because of the implications of shuffling all these messages around in memory, there will be a practical limitation even if one is not enforced.

In any case, the more compact your messages are the better your system will perform. Stuffing large objects in your messages is not a best practice, but what other option do we have?

The solution is to take those large objects and transport them by some other means, but doing this manually for every message would be a pain in the neck! Luckily, we don't have to. NServiceBus will use the **Data Bus** to transport these objects for us with just a small bit of configuration.

Within our message contracts we can easily specify that a property, will be transported by the Data Bus and not as part of the body of the message.

```
[TimeToBeReceived("1.00:00:00")]
public class ProcessPhotoCmd : ICommand
{
    public DataBusProperty<byte[]> PhotoBytes { get; set; }
}
```

Notice the wrapper class that indicates the Data Bus property, and also note that we're using the `TimeToBeReceived` attribute which lets the Data Bus know when it can clean up the attachments for messages that have expired. You can omit this attribute if you want, but then you may have to clean up expired items manually.

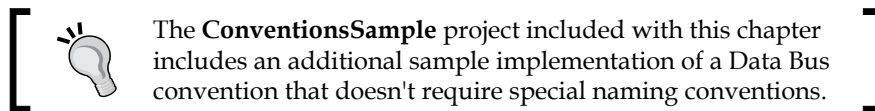
You may be wondering why the Data Bus payload isn't just removed immediately once the message is successfully processed. It's important to remember that if an event includes a Data Bus property, then multiple handlers may need to access the same payload, and it's impossible for any given handler to know how many other subscribers are out there.

Of course, using the `DataBusProperty<T>` class does not fit with unobtrusive mode, but we have a way to specify the Data Bus convention as a `Func<PropertyInfo, bool>` property. Here's a simple example that defines that any property name ending with `DataBus` is a Data Bus property, and a rewritten command definition:

```
Configure.Instance
    .DefiningDataBusPropertiesAs(pi => pi.Name.EndsWith("DataBus"));

[TimeToBeReceived("1.00:00:00")]
public class ProcessPhotoCmd
{
    public byte[] PhotoBytesDataBus { get; set; }
}
```

Note that when using conventions, you can use simple types without a wrapper object. The Data Bus serializer is able to distinguish between the two forms.



We also need to specify what kind of Data Bus any given endpoint will use. This is a decision similar to a message transport or serializer – we make it once for an entire system and use it throughout.

`NServiceBus` comes with one Data Bus implementation out of the box that stores the large objects in a file share.

```
Configure.Instance.FileShareDataBus(@"\\server\share");
```

The `FileShareDataBus` is a great choice for a system that's completely on premise or connected by VPN, where all endpoints will have access to the same file share. You can create your own implementation (for instance, to transfer via FTP, or store in Amazon S3) by implementing the `IDataBus` interface which has fairly simple `Start`, `Put`, and `Get` methods. The `Put` and `Get` methods are self-explanatory, and the `Start` method allows you to start a routine to periodically clean out expired items. Once we create our own data bus implementation, we register it using dependency injection:

```
Configure.Component<MyDataBus>(DependencyLifecycle.SingleInstance);
```

## Exposing web services

Shocking as it may sound, not every computer on the planet runs Windows and the .NET Framework. Sometimes we may even need to exchange data with our cross-platform brethren. The new transports included with NServiceBus 4.0 can be a great way to bridge the platform gap, but when there is only an occasional need to swap data with another platform, NServiceBus allows us to expose a command handler as a WCF web service with very little effort on our part.

In order to expose a web service, we will need a few things:

- A command message, implementing `ICommand` or defined using unobtrusive mode conventions. This will serve as the input to the web service.
- An enum that will serve as the return value for the web service.
- A handler class that processes the command and then performs `Bus.Return<ResponseEnumType>(ResponseEnumType returnCode)` to return the response.

Once we have this in place, we can create a web service very easily, at least as far as the NServiceBus code is concerned:

```
public class MyWebSvc : WcfService<MyCmd, MyResponse>
{
}
```

The NServiceBus host will find this class and wire up the service for you. All that is left is to specify the necessary WCF configuration which we unfortunately cannot escape from.

The main pain point in specifying the WCF configuration is the contract, which takes the stringified form of `IWcfService<MyCmd, MyResponse>` implemented by the `WcfService<MyCmd, MyResponse>` class.

```
<endpoint contract="NServiceBus.IWcfService`2[[MyNS.MyCmd,
MyAssembly, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null],[MyNS.MyResponse, MyAssembly,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]" />
```

Note the backtick after `IWcfService` and the double square brackets. This is how .NET represents the type name for the generic type in plain text.

To see the full WCF configuration, check out the **WcfSample** project included with the code for this chapter. Once the web service is exposed, you can browse to the service's endpoint URL to see the boilerplate web service page and link to the WSDL for the service. The code sample demonstrates connecting to the service using a standard service proxy class generated by Visual Studio.



Hosting a WCF service requires NServiceBus to bind to a port to listen for requests. This may require you to run Visual Studio with elevated privileges.

NServiceBus command handlers exposed as web services are meant to be hosted by the NServiceBus host. While it is technically possible to host it while self-hosting within a web application, doing so would require that the command handler also be hosted within the web application's AppDomain, and that is not recommended.

Additionally, the web service capability should not be used for cross-site messaging. This can be accomplished with the Gateway component, which is discussed in more detail in *Chapter 7, Administration*.

## Summary

In this chapter we learned how to take advantage of the full power of the NServiceBus host. First we explored the interfaces that we can implement to make our own customizations to the NServiceBus host, and learned how we can use dependency injection to inject our own customizations into the framework. We learned how to use additional bus settings, including controlling the message serializer and transaction settings.

Next we learned how to create message assemblies without a dependency on NServiceBus itself, which will grant us the ultimate freedom to perform updates to different components in our system independently.

After that, we learned some details about the messaging pipeline, including how we can take advantage of polymorphic dispatch to version our messages, how we can control the order of message handlers in the pipeline, and actions we can take on the messages when they are in the pipeline. We learned how to manage the unit of work so that we can manage resources at the start and end of the pipeline, and how we can mutate messages coming in or going out so that we can satisfy infrastructure concerns.

Lastly we learned how to encrypt message properties, transport large payloads over the Data Bus, and how we can expose the processing of a command as a WCF service as an option for interacting with remote client code that may even be on a different platform.

Up until now we have been dealing with fairly simple messaging examples where a message comes in, is processed, and has a result. In the next chapter we will learn about sagas, which enable us to organize all of these message handlers together to perform complex business processes that are long-running, sometimes spanning many minutes all the way up to months and years.







Long running business processes are all around us. A retailer can't ship a product until after the credit card has been charged, but only if the user hasn't canceled the order in the meantime. After spending a certain amount of money, a customer attains preferred status, entitling them to free shipping. Frequent flyers with Gold status get more frequent upgrades to first class and access to special airport lounges where they serve free beer. And really, is there anything more important than free beer?

## Long-running processes

Normally these processes are controlled by ugly behemoths called **batch jobs** or **scheduled tasks**. They run in the dead of night and update our data depending upon the business rules of the day. But what happens when this is no longer good enough?

The data needs of most companies are growing year after year. What happens when the database contains so many records that the nightly batch job takes longer than off-peak hours will permit, and causes too much of a performance penalty to be run during peak hours? What happens if you work in an industry where there is no such thing as off-peak hours? And of course let's not forget that most batch jobs are notoriously brittle and prone to failure.

Perhaps the more troublesome problem with the old way of doing things is our own users' expectations. Consider the frequent flyer who has just flown the last mile to achieve Gold status. They can check their miles on their smartphone app and they know they should be able to access the airport lounge with the free beer, but they are denied entry. "It's OK," says the attendant, "you just have to wait for the overnight batch job. You can access the lounge tomorrow."

This isn't any way to treat our customers! Luckily we can create a business process that is reliable and real-time, and leaves all of our batch jobs in the past where they belong.

## Defining a saga

To demonstrate how to build a saga, we will revisit the example project from the first two chapters, modified slightly to support the Unobtrusive Mode conventions we learned in *Chapter 5, Advanced Messaging*. We will be implementing email verification, meaning that before the user is created in the database we will first send them an email containing a code. Then the user must click on a link in that email to prove they own the address before we create them in the database.

Now let's start building our saga in the **UserService** project. We're going to be redefining how the `CreateNewUserCmd` message is handled, so for now, comment out the entirety of the `UserCreator` class.

Sagas define behavior and implement business rules, so it's sometimes helpful to think of them as policies. Therefore we will be calling our saga `VerifyUserEmailPolicy`, and create a new class file for this and start with the following structure, all in the same file.

```
public class VerifyUserEmailPolicy :
    Saga<VerifyUserEmailPolicyData>
{
    private static readonly ILog log = LogManager
        .GetLogger(typeof(VerifyUserEmailPolicy));
}

public class VerifyUserEmailPolicyData : ContainSagaData
{
    public string Name { get; set; }
    [Unique]
    public string EmailAddress { get; set; }
    public string VerificationCode { get; set; }
}
```

You will need to add it using declarations for the `NServiceBus.Saga` and `NServiceBus.Logging` namespaces to get all the references to resolve.

This code defines the saga class itself and the data storage it uses. The data class is the saga's memory and is automatically persisted between messages. The properties shown are the information we will need to store for the duration of our saga. The class also inherits other properties from the `ContainSagaData` base class which are used internally by `NServiceBus` and should be ignored.

Note that we mark the `EmailAddress` property with the `Unique` attribute to let `NServiceBus` know that we only ever plan to have one active saga per email address. This hint allows `NServiceBus` to work with the underlying saga storage to create constraints ensuring that only one thread can modify the saga data at once.

Now let's add a handler for the `CreateNewUserCmd` message that the `UserCreator` class previously handled. Implement `IAmStartedByMessages<CreateNewUserCmd>` on the saga class. This defines a `Handle` (the `CreateNewUserCmd` message) method identical to `IHandleMessages<T>`, but carries the additional instruction to the saga to create a new saga instance if it can't find an existing one.

Implement the `Handle` method as shown:

```
public void Handle(CreateNewUserCmd message)
{
    this.Data.Name = message.Name;
    this.Data.EmailAddress = message.EmailAddress;
    this.Data.VerificationCode = Guid.NewGuid()
        .ToString("n").Substring(0, 4);

    Bus.Send(new SendVerificationEmailCmd
    {
        Name = message.Name,
        EmailAddress = message.EmailAddress,
        VerificationCode = Data.VerificationCode,
        IsReminder = false
    });
}
```



It's important to note that `Bus` is already provided for us in the `Saga`'s base class. This trips a lot of people up! If you start with a normal handler class with an injected `Bus` instance and then convert that handler into a `Saga`, be sure to remove your injected `Bus` property, or it will mask the base class instance and things won't work properly.

Our saga data is available through the base `Saga<T>` class as `this.Data`. Because this message starts the saga, this data needs to be set for the first time. In this example we set `VerificationCode` to a random string by generating a `Guid` and taking the first four characters. In real life we would want this verification code to be longer, but we're going to have to test this later and would not enjoy typing out an entire `Guid` by hand!

After setting up the saga data, we send a new command called `SendVerificationEmailCmd`. The handler for this command should format and send an email to the user containing a link that will include the verification code. The idea behind the `IsReminder` property is that we can enable sending two slightly different versions of the email: one initially and one after some length of time has elapsed to remind the user to verify their account if they have forgotten. We will cover this momentarily in the *Dealing with time* section.

We won't cover creating the command or the handler for this as it should be old hat by now! However an implementation that simply logs the data to the console is included in the full code sample with this chapter. Remember you will also need to define the message routing for the `SendVerificationEmailCmd` in the `App.config` file.

## Finding saga data

Wait a second. We said that `IAmStartedByMessages<T>` tells the saga to create a new saga instance if existing saga data is not found, but how does it know where to look for the saga data?

As it turns out, we need to give `NServiceBus` a little bit of help with this. We do this by overriding the saga's `ConfigureHowToFindSaga()` method.

```
public override void ConfigureHowToFindSaga()
{
    this.ConfigureMapping<CreateNewUserCmd>(msg =>
        msg.EmailAddress).ToSaga(data => data.EmailAddress);
}
```

The `ConfigureMapping<TMessageType>` method accepts an `Expression` that identifies a property on the incoming message to match with the saga data. Then with the result, we call the `ToSaga` method that accepts a similar expression pointing to the matching property in the saga data. So in essence, this entire expression says "Find a property on the `CreateNewUserCmd` message called `EmailAddress`, and then find a saga instance that has the same value for its `EmailAddress`."



There is an additional method of finding a saga, by implementing `IFindSagas<TSagaData>.Using<TMessage>`. This is considerably more advanced and requires interacting directly with the Saga storage. The `ConfigureHowToFindSaga()` method should be sufficient for most use cases.

When two or three properties together uniquely identify the Saga, a much simpler strategy is to combine these values into a single composite key and store that as a separate property on the saga data called `SagaKey` or something to that effect.

## Ending a saga

When the user receives this email, obviously they are going to be very excited to join your site so they are going to click on it immediately. Let's handle this next.

In the website's `HomeController`, add the following action method:

```
public ActionResult VerifyUser(string email, string code)
{
    var cmd = new UserVerifyingEmailCmd
    {
        EmailAddress = email,
        VerificationCode = code
    };

    ServiceBus.Bus.Send(cmd);

    return Json(new { sent = cmd });
}
```

This should look similar to the `CreateUser` action method we already have. The routing information we have in the `Web.config` file already routes all messages from our `UserService.Messages` assembly to the `UserService` endpoint, so we shouldn't need to add anything there.

Now we can handle this new message within our saga. This new message cannot possibly start the saga because our business process cannot create it until after the saga has started. So we must implement `IHandleMessages<UserVerifyingEmailCmd>` on the saga just like we would with a normal message handler. Implement the `Handle` method like this:

```
public void Handle(UserVerifyingEmailCmd message)
{
    if(message.VerificationCode == this.Data.VerificationCode)
    {
        Bus.Send(new CreateNewUserWithVerifiedEmailCmd
        {
            EmailAddress = this.Data.EmailAddress,
            Name = this.Data.Name
        });

        this.MarkAsComplete();
    }
}
```

All we are doing is checking to make sure the incoming verification code matches the one we already have stored in our saga data. If it does, we send another new message (again, available in the sample code but not shown here) to create the user for real, and then call the `MarkAsComplete()` method.

The `MarkAsComplete()` method finishes the saga, and instructs the infrastructure that we can throw away the related saga data, as it will not be needed anymore.

As we have added a new message, we also need to remember to add a new saga mapping:

```
public override void ConfigureHowToFindSaga()
{
    this.ConfigureMapping<CreateNewUserCmd>(msg =>
        msg.EmailAddress).ToSaga(data => data.EmailAddress);
    this.ConfigureMapping<UserVerifyingEmailCmd>(msg =>
        msg.EmailAddress).ToSaga(data => data.EmailAddress);
}
```

When the `CreateNewUserWithVerifiedEmailCmd` message is received, we want to create the user the same way as we did before we started our saga example. To do this, switch over to the `UserCreator.cs` file, uncomment it, and replace `CreateNewUserCmd` with `CreateNewUserWithVerifiedEmailCmd`, both within the interface and the `Handle` method.

---

At this point we have covered all of the user interactions, assuming that the user is paying attention, and we can now test the system.

1. Run **UserService**, **WelcomeEmailService**, and **ExampleWeb**.
2. In the web browser, navigate to:  
`/CreateUser?name=David&email=david@example.com`
3. The **UserService** should tell us that it's sending a verification email with the verification code. Jot that down so we can simulate the email click.
4. To simulate the link click, navigate to:  
`/VerifyUser?email=david@example.com&code={VerificationCode}`
5. The **UserService** should tell us that it's now creating the user.
6. The **WelcomeEmailService** should tell us that it is sending the welcome email.

So, we can see how we have inserted a business policy before the creation of a user. We didn't have to change the schema of our user database to support a "non-verified" type of user, in fact we wouldn't have to change our data layer at all! But we're still missing the element of time.

## Dealing with time

Right now you might be saying that we could have implemented what we have so far using a database table for temporary users, and that all we'd need is a cleanup batch job to clear out the users that never follow through. However, to do so would be to miss the great flexibility that sagas offer.

At their core, sagas are entities that contain multiple message handlers with shared state, but they also offer the ability to set a **timeout**, which is like setting an alarm clock to wake you up at some point in the future. This ensures that the process does not have to stop just because no new messages come in.

But better than just an anonymous alarm clock, we're also able to pass the state into the future, so not only will our saga wake up on command but it will also know why.

In our case, we want two timeouts. When the user first attempts to register, we want a wakeup call two days later, so that we can remind the user to complete the registration, just in case they get busy and forget about us. Then, we want an additional wakeup call after seven days, so that we can remove their information and clean up.



First we need to define our timeout messages. Add these classes to the `VerifyUserEmailPolicy.cs` file, outside the main `VerifyUserEmailPolicy` class.

```
public class VerifyUserEmailReminderTimeout
{
}

public class VerifyUserEmailExpiredTimeout
{
}
```

This is just enough state so that when the timeout is triggered, we will know why. Timeouts are like messages, and they can contain properties to pass Additional state, however the simpler you can make them, the better.

In order to set the alarm clock for these timeouts, add the following code to the end of the `Handle(CreateNewUserCmd message)` method.

```
this.RequestTimeout<VerifyUserEmailReminderTimeout>(
    TimeSpan.FromDays(2));
this.RequestTimeout<VerifyUserEmailExpiredTimeout>(
    TimeSpan.FromDays(7));
```

This overload of the `RequestTimeout` method specifies the type of timeout message, and the timeout duration using a `TimeSpan` object. Since we have no state properties to set, `NServiceBus` will create the timeout for us. There are several other overloads that make the following combinations of options possible:

- Specify the duration as an absolute `DateTime` object or as a `TimeSpan` object
- Supply an instance of the timeout message with any state properties already filled
- Supply an `Action<T>` delegate to set state properties for the timeout message

Now we need to handle our timeouts within the saga. To do this, implement the `IHandleTimeouts<T>` interface on the saga class for both the timeout types. The implementation for this interface is a method named `Timeout`, which is shown here:

```
public void Timeout(VerifyUserEmailReminderTimeout state)
{
    Bus.Send(new SendVerificationEmailCmd
    {
        Name = Data.Name,
        EmailAddress = Data.EmailAddress,
        VerificationCode = Data.VerificationCode,
```

```
        IsReminder = true
    });
}

public void Timeout(VerifyUserEmailExpiredTimeout state)
{
    this.MarkAsComplete();
}
```

As we can see, if the timeout is the reminder use case that occurs after two days, we'll send the same message we used earlier to send the user an email, except this time we specify it will be a reminder email with the `IsReminder = true` flag.

If the timeout is the expiration use case that occurs after seven days, then we will use the `MarkAsComplete()` method to finish up the saga, after which our saga data will be removed.

If any message (whether a normal message or a timeout) arrives at the saga after `MarkAsComplete()` is called, then the infrastructure will ignore that message. Therefore, if the user attempts to verify their account just after it expires, that message will be ignored and the user will not be verified. Conversely, if the user verifies their account an instant before the timeout is fired, then the timeout will be ignored.

## Design guidelines

In any instance where you might use a batch job or scheduled task, or in situations that involve complex ever-changing business requirements, the saga pattern is generally a good fit. However there are some things you should keep in mind.

## Business logic only

While it may be tempting to throw a whole bunch of logic, data access, and the whole kitchen sink into a saga, this is not a good idea.

Although saga data storage is abstracted to be very easy to work with, remember at some point that data needs to be persisted somehow. In production, the default is to use RavenDB, but you can also use a relational database via NHibernate, or roll your own saga storage by creating an implementation of `ISagaPersister` and registering it with the dependency injection framework.

No matter which persistence mechanism is used, if there are a lot of messages being processed in parallel then there will be contention on the saga storage which you don't want to exacerbate, for example, by adding data access to the ambient transaction.

For this reason, message handlers within sagas should be for message processing and business logic only. Messages or timeouts go in, decisions are made, and then messages and/or timeouts come out. Any additional work should be carried out by independent message handlers. Think of the saga like the captain of the ship: the captain gives orders but doesn't actually drive!

So then we have a need to dispatch a command from the saga to an independent handler, and then have that handler report back the status. We dispatch the command the same way we're accustomed to using `Bus.Send()`, but when it comes time to report back, we use `Bus.Reply()` instead:

```
public class MyHandler : IHandleMessages<MyCmd>
{
    public void Handle(MyCmd message)
    {
        // Do work!

        Bus.Reply<ReplyMsg>(m => /* set props of m */);
    }
}
```

There's even a bonus! `NServiceBus` will know that you are replying in response to a command sent from a saga, and it will take care of correlating the reply message back to the correct saga instance. This means in this example you don't even have to add a mapping for the `ReplyMsg` class in `ConfigureHowToFindSaga()`!

At some point your saga may need to report back to the originator, that is, the sender of the message that started the saga. `Bus.Reply()` won't work for this, as it only replies to the sender of the message currently being processed. To do this, call the saga's `ReplyToOriginator()` method instead.


When using unobtrusive mode conventions, reply messages are neither commands nor events. Instead, they must fit the `.DefiningMessagesAs()` convention. In this book we define these messages by a namespace ending with `InternalMessages`.

## Saga lifetime

The example we showed here features a saga with a very definite lifespan, but that isn't the way it always has to be. Sagas don't necessarily ever have to end.

Consider the frequent flyer story from the beginning of the chapter. In order to obtain Gold status to get into the executive suite, you had to have flown 50,000 miles within the past calendar year. How would we write a saga for this?

The saga data would store a running total of miles and your current flyer status. If you flew from New York to Los Angeles, the running total would be incremented by roughly 2800 miles, and a timeout would be set to decrement the running total by 2800 miles 365 days from now. If the running total went over 50,000 or back under 50,000 then events would be published to announce that your Gold status had changed. However the saga would never be completed.

 Savvy readers may notice that there really isn't much conceptual difference between a Saga that never ends and an **Aggregate Root** in the parlance of **Domain Driven Design**.

## Saga patterns

Sagas can be as varied as the developers who create them, but generally they follow two basic patterns.

Sagas that follow the **controller pattern** take an active management role, sending commands to specific endpoints and waiting for their replies before advancing to the next step. This is a straightforward way to direct a process through a workflow. It allows you to repeat sections, make decisions, and respond to error conditions with compensating actions.

Sagas that follow the **observer pattern** will passively listen for the events from other services, and will use that information to coordinate some activity, generally by publishing its own event after all the events it is interested in have been received.

The important thing to remember is that when you're dealing with messaging, messages aren't guaranteed to arrive in order, or in any order whatsoever! Generally this means that multiple incoming events can start the saga, and you need to plan for messages to arrive in the order you would least expect.

These two styles are the reflection of the two types of messages. The controller pattern sends commands, and the observer pattern reacts to events. In reality these two styles can be mixed up in any number of combinations.

An in-depth examination of various saga patterns is beyond the scope of this book, however, the topic has already been expertly covered by NServiceBus Champion Jimmy Bogard. To learn more, check out his series of posts on saga implementation patterns on his blog at <http://lostechies.com/jimmybogard/2013/05/14/saga-patterns-wrap-up/>.

## Retraining the business

Perhaps the biggest challenge when creating sagas is not a technical challenge at all, but a human one. For years we have trained our business stakeholders to think in terms of batch jobs and scheduled tasks. When they give you requirements, it will be couched in these terms.

Just as it requires us as developers to think in a different way when we build messaging systems, so must we retrain our business stakeholders as well. All we need to do is ask the right questions. For example, in our frequent flyer program example, if we point out that running a nightly batch job will mean that the newly minted Gold Member will not be able to access the executive lounge until the next day, business stakeholders will likely be quick to point out that this isn't what they really want, but that they were making assumptions based on the tools they knew to be available. With a little back and forth, you can discover the true system requirements.

## Unit testing

When we were covering saga timeouts, you may have found yourself wondering just how to test something that wasn't supposed to happen for seven days, let alone an entire year!

One option would be to temporarily set all the timeouts to something very short that you could observe before your lunch break, but this approach is problematic for several reasons. It's still hard to test this way, as you must make the timeouts long enough that you have time to be ready to observe what happens, but short enough that you don't get bored, zone out, and miss what it was you were trying to see in the first place. Then you're in for a world of hurt when you eventually forget to set those test values back before committing your code!

A much better approach is to take advantage of the `NServiceBus` testing framework, available through the **NServiceBus.Testing** NuGet package. You gain the ability to verify your long-running business processes quickly, along with all the other benefits of unit testing. Sagas are meant to define constantly changing business rules, so it is very useful to have a suite of automated regression tests to alert you to a problem after changes are made.

To get started testing a saga, create a new class library and use NuGet to install the **NServiceBus.Testing** package and whatever unit testing framework you prefer. All of the examples in this book will use NUnit. In addition, you will need to add references to the assembly that contains your saga and any message assemblies it uses.

The first step is to initialize the testing framework.

```
[TestFixture]
public class VerifyUserEmailPolicyTests
{
    public VerifyUserEmailPolicyTests()
    {
        Test.Initialize();
    }
}
```

This is analogous to starting up the Bus when self-hosting. The `Initialize()` method also has overloads that support specifying the assemblies or types to scan, and we can use the same `AllAssemblies.Except()` directive that self-hosting supports. For the most part, however, the parameterless `Initialize()` method will work fine.

Note that we initialized the testing framework within the class constructor. Some testing frameworks have explicit methods for setup and tear-down. Just be sure that the framework is initialized before you start running tests or you will run into errors when the code you are testing tries to access the Bus.

With the testing framework initialized, we can start to test our saga:

```
var testSaga = Test.Saga<VerifyUserEmailPolicy>();
```

We use the type of our saga as the generic parameter, and this returns an object that will allow us to start our test script. We wouldn't have to assign it to a variable, but it can be helpful in order to separate tests from each other.

Each test follows a pattern that can be expressed using a variation on a simple phrase:

*I expect that {things will happen} when {a trigger occurs}.*

Let's see how this pattern works out with a sample test, and then we'll break it down.

```
CreateNewUserCmd createUser = new CreateNewUserCmd
{
    Name = "David",
    EmailAddress = "david@example.com"
};

string correctCode = null;

testSaga.ExpectSend<SendVerificationEmailCmd>(cmd =>
```

```
{
    correctCode = cmd.VerificationCode;
    return cmd.EmailAddress == createUser.EmailAddress
        && cmd.Name == createUser.Name;
})
.ExpectTimeoutToBeSetIn<VerifyUserEmailReminderTimeout>(
    (timeout, timespan) => timespan == TimeSpan.FromDays(2))
.ExpectTimeoutToBeSetIn<VerifyUserEmailExpiredTimeout>(
    (timeout, timespan) => timespan == TimeSpan.FromDays(7))
.When(saga => saga.Handle(createUser));
```

Going down to the bottom, to the `When()` method, we see that the trigger is the saga receiving the `CreateNewUserCmd` message that starts it. Our expectations are detailed above it:

- We expect the saga to send a `SendVerificationEmailCmd` message. We pass it a delegate to verify that properties are set on the message as we expect. We also use the delegate to get the generated verification code out so that we can use it in the next test.
- We expect a timeout to be set for 2 days in order to trigger the reminder email.
- We expect a timeout to be set for 7 days in order to expire the saga.

From here we could begin the next test without even stopping for a semicolon, but it's useful to separate them so that the **Test/Expect/When** pattern is easier to see in the code, not to mention step through in the debugger.

The testing framework contains several `Expect` methods, covering everything from sending and publishing, not sending or publishing, replying, returning, and setting timeouts, and a few `When` methods to cover receiving messages and timeouts. Besides these, a few other methods are needed to round out the testing suite.

If your saga requires any external dependencies, you can set them up at the beginning of the test.

```
Test.Saga<TSagaType>()
    .WithExternalDependencies(saga =>
        {
            // Configure saga dependencies here
        })
    // Expect clauses
    .When(saga => saga.Handle(command));
```

You can also set incoming properties on a message.

```
Test.Saga<TSagaType>()
    // Expect clauses
    .SetIncomingHeader("Header-Key", "Header-Value")
    .SetMessageId(messageId)
    .When(saga => saga.Handle(command));
```

Lastly, we need to be able to make assertions about whether the saga has completed or not. We can make this assertion between tests, or right after the `When` clause.

```
Test.Saga<TSagaType>
    // Expect clauses
    .When(saga => saga.Handle(command))
    .AssertSagaCompletionIs(true);
```

Note that it is just as useful to assert that a saga is not complete!

For a complete example of a test suite for our example saga, check out the sample code included with this chapter.

## Scheduling

By now you're probably thinking that Saga timeouts sound great, but what if you just need to run something on a schedule? Maybe you're thinking you'll create an `IWantToRunWhenBusStartsAndStops` class with a `Timer`. Well if you are, stop right there! `NServiceBus` can bring the power of timeouts to you without the full ceremony of a saga.

```
Schedule.Every(TimeSpan.FromMinutes(5)).Action(() =>
{
    Bus.Send<DoSomethingEvery5MinutesCmd>();
});

Schedule.Every(TimeSpan.FromMinutes(5)).Action("Task Name", () =>
{
    Bus.Send<DoNamedTaskEvery5MinutesCmd>();
});
```

With an `NServiceBus` scheduled task, you can count on the action happening no matter what, even if you schedule it to run every 24 hours and your IT manager reboots the server every 23 hours.



Generally, you will want to schedule tasks to be run by calling the Schedule API from the Start method of an `IWantToRunWhenBusStartsAndStops` implementation.

The scheduler is kind of like a mini-saga that can send messages at particular times. The important thing to keep in mind when using schedules is to steer clear of any custom logic. The scheduler should only kick off a message – then you can do whatever needs doing within the transactional safety of a message handler. If you find yourself wanting to introduce logic into the schedule, you should upgrade to a full saga.

## Summary

In this chapter we learned how to use sagas to create long-running business processes that bring the reliability we've come to associate with message handlers to a realm previously occupied by brittle and unreliable batch jobs and scheduled tasks.

We learned how to define a saga and its data, and how to find a saga's data based on an incoming message. We learned how to request timeouts so that a saga can wake itself up at some point in the future, and how to end a saga if required.

We learned the importance of restricting a saga's activities to business logic only, segregating data access and other work to other message handlers that can communicate back to the saga with reply messages. We discussed saga lifetime, the controller and observer saga patterns, and the importance of retraining our business stakeholders who may be too used to the old way of doing things to realize what can be accomplished by using messaging.

We learned how we can leverage unit testing to validate our business logic. As sagas tend to represent constantly changing business rules, the ability to unit test will give us the valuable ability to ensure that our test cases do not fail when we make a change to the system.

Lastly, we learned how we can use the scheduling API to reliably schedule messages to be sent on a fixed schedule with minimal fuss.

In the next chapter, we will take a look at how to administer an `NServiceBus` system so that we can successfully manage our code in production.

# 7

## Administration

At this point, we've learned all the basics of NServiceBus and can create a system to do just about anything, but none of that will do us any good if we can't get our code deployed to production.

In this chapter, we will learn about issues related to deploying and managing a production system using NServiceBus. We will learn how to install our processes as services and how to manage the transitions between different environments. We will learn how to monitor our production system, and how to scale that system out as the load increases. Finally, we'll take a look at virtualization, an important component to creating truly reliable systems.

### Service installation

Throughout this book we've been testing our service endpoints by running them with the NServiceBus host as a console process. This is really convenient for development, but when we deploy to a different environment we need the stability of a Windows service. Luckily, NServiceBus makes this ridiculously easy to do. The same NServiceBus host process that runs as a console process during development can install itself as a Windows service with a few simple command-line switches.

```
> NServiceBus.Host.exe -install [Profile(s)]
    -serviceName="MyServiceName"
    -displayName="My Service"
    -description="Description of MyServiceName"
    -username="mydomain\myusername"
    -password="mypassword"
```

This is the basic command that will install an endpoint as a Windows service. Not all of the parameters are required, but this set will cover most cases.

Ignore the profiles for a moment; we will cover these in the *Profiles* section, later.

The service name is the name of the service used in the Windows Registry and in `NET START` or `NET STOP` console commands, whereas the display name and description are the strings that you will see in the Windows Service Manager. If you omit the service name, display name, or description, NServiceBus will assign defaults based on the endpoint name. If you recall from *Chapter 4, Self-Hosting*, the endpoint name defines the pattern for the queues that are created for the endpoint as well. It's very useful that in using this convention, the queue and service names will match.

By default, NServiceBus will append a version string to the display name of the service. This is the version from the `AssemblyFileVersion` attribute on the assembly containing the `EndpointConfig`. In addition, there is an optional `-sideBySide` parameter that will result in the version being appended to the service name (the code name used by the Windows Registry) as well. This is critical if you wish to support side-by-side upgrades, since you can have both old and new versions of a service installed for easy rollback during an upgrade.



It's recommended to come to an agreement with your IT department about how Windows services will be named, and then to always use all three of these parameters when installing to guarantee consistency. The `-username` and `-password` parameters must be used together, and allow you to specify the account the service will run under. If not supplied, the service will run under a local service account.

It is just as easy to uninstall a service:

```
> NServiceBus.Host.exe -uninstall -serviceName:"MyServiceName"
```

Whenever you specify a service name on install, you must also specify that name to uninstall.

If you ever find yourself without a copy of this book (first of all shame on you!) then you can quickly summon a refresher on the host process's command-line options using any of the following:

```
> NServiceBus.Host.exe -?  
> NServiceBus.Host.exe -h  
> NServiceBus.Host.exe --help
```

The help text also includes more information on more advanced command-line switches.



Installing services requires elevated privileges. If **User Account Control (UAC)** is enabled on your system, make sure to launch the console window with elevated privileges first as you won't have the opportunity to do so while the install process is running.

## Profiles

In a lot of software systems, you'll either see a litany of different settings in a configuration file, or a single configuration value and a huge wall of settings in a switch statement in code. Not so with NServiceBus. NServiceBus uses a concept called **profile** to activate different options within an endpoint based on environment or capability.



Many of the dependency injection containers support a similar profile feature. Be sure to check the documentation for your favorite one!

A profile can be activated on an endpoint by including its full class name as a command line parameter, or by including it with the install options when installing an endpoint as a service. For instance, to run an endpoint with the Lite profile, as we have done in some of the examples in this book, run the following code:

```
> NServiceBus.Host.exe NServiceBus.Lite
```

All of the built-in NServiceBus profiles are located directly in the NServiceBus namespace. These profiles fall into two categories; environmental profiles, and feature profiles.

## Environmental profiles

Environmental profiles help you to manage the elements of a system through the different phases of development. NServiceBus includes three environmental profiles: Lite, Integration, and Production.

- In the **Lite** profile, everything is optimized for quick development and the ability to make changes rapidly.
  - Subscription storage, saga storage, and timeouts are all handled in memory.
  - Messages that fail are logged but not forwarded to the error queue.

- Installers are run on startup to enable quick development.
- Logging is done to the console window.
- In the **Integration** profile, durable storage makes it possible to verify a system's operation in Test and QA environments.
  - RavenDB is used for persistence of subscriptions, sagas, and timeouts by default.
  - Installers are run on startup to enable test environment setup.
  - Normal error handling is used.
  - Logging is still done to the console window.
- In the **Production** profile, everything is optimized for real-life operation. This is the default profile if nothing is specified, as "production-ready by default" is one of the core values in the NServiceBus design philosophy.
  - RavenDB is used for persistence of subscriptions, sagas, and timeouts by default.
  - Installers are not run on startup, but are instead run when the service is installed.
  - Normal error handling is used.
  - Logging is done to files instead of the console window.
  - Performance counters are enabled. We will learn more about the performance counters in the *Monitoring* section later in this chapter.

## Feature profiles

Feature profiles help you to modify specific endpoint features programmatically.

- The **Master**, **Distributor**, and **Worker** profiles are used for scaling out an endpoint. We'll discuss these profiles in detail in the *Scaling out* section.
- The **PerformanceCounters** profile activates the performance counters for the endpoint. Because the Production profile inherits from this, the Production profile also enables performance counters by association. We will learn more about this in the *Monitoring* section.
- The **MultiSite** profile enables the gateway component. We will learn more about the gateway in the *Multiple sites* section.

You can activate more than one profile on an endpoint by including multiple parameters. When installing an endpoint, you can place the profile names anywhere within the install command, but I recommend right after the `-install` flag.

## Customizing profiles

Beyond the default behavior for each profile, we can create our own profile-based customizations. This gives us the ability to change the NServiceBus host (usually via dependency injection) similar to how we learned in *Chapter 5, Advanced Messaging*, but specifically targeted to one or more profiles.

Adding code to be executed for a profile is as simple as implementing `IHandleProfile<T>`, where `T` is one of the profile classes that implements `IProfile`.

For example, let's say you had an account expiration policy, and you wanted to use shorter times in your integration environment than in real life. You could represent this as an interface called `IDetermineAccountExpiration`. Then, we could create a class that returns very short expiration times suitable for a test environment and register it only for the Integration profile as follows:

```
public class ConfigTestExpirations : IHandleProfile<Integration>
{
    public void ProfileActivated()
    {
        Configure.Component<TestAccountExpiration>(
            DependencyLifecycle.SingleInstance);
    }
}
```

If we want, we can also implement `IWantTheListOfActiveProfiles` within one of our profile handlers to get all of the active profiles injected into our class. This is what the built-in handlers for the Master, Worker, and Distributor profiles use to make sure they are not applied simultaneously as they are incompatible with each other. For example, you may want to take some action in the Production profile if the current endpoint is also a master node:

```
public class KnowsAllProfiles : IHandleProfile<Production>,
    IWantTheListOfActiveProfiles
{
    public IEnumerable<Type> ActiveProfiles { get; set; }

    public void ProfileActivated()
    {
        if (ActiveProfiles.Contains(typeof(Master)))
        {
            // Do something
        }
    }
}
```

We can also implement the `IWantTheEndpointConfig` interface to have the `EndpointConfig` class injected into our profile handler. This is useful if you want to make decisions based on the endpoint's identity.

To make decisions for any profile, there is an interface called `IHandleAnyProfile` which combines a catch-all profile handler with `IWantTheListOfActiveProfiles`, so that we can make decisions such as "If the Production profile is active, do X, otherwise do Y."

```
public class DecisionProfile : IHandleAnyProfile
{
    public IEnumerable<Type> ActiveProfiles { get; set; }

    public void ProfileActivated()
    {
        if (ActiveProfiles.Contains(typeof(Production))
        {
            // Register a real implementation of something
        }
        else
        {
            // Register a test implementation instead
        }
    }
}
```

We can even create our own profiles by creating an interface that inherits `IProfile`, and then assign actions to them by creating a profile handler for that profile. For instance, as we hinted at in *Chapter 5, Advanced Messaging*, we could create the profile classes `Packt.Dev`, `Packt.Test`, `Packt.QA`, and `Packt.Production`, then use profile handlers to register different dependencies for each environment.

`NServiceBus` will invoke all the profile handlers for the profile(s) we specify on the command line and any base classes or interfaces as well. This means we can inherit from one of the built-in profiles to extend them, or create our own that directly implements `IProfile` to start from scratch.

Profile handlers are invoked between `IWantToRunBeforeConfigurationIsFinalized` and `IWantToRunWhenConfigurationIsComplete`. If multiple profiles are used on the command line, you should not assume that their handlers will execute in any specific order.

## Logging profiles

Similar to how logging received special treatment during endpoint startup, logging also gets special treatment with respect to profiles, in order to have the logging framework set up properly before all of our other code runs.

Implement `IConfigureLoggingForProfile<T>` to make profile-dependent logging choices.

```
public class ProdLogging : IConfigureLoggingForProfile<Production>
{
    public void Configure(IConfigureThisEndpoint specifier)
    {
        // Set up logging
    }
}
```

As we can see, the `EndpointConfig` class is passed directly into the `Configure` method. This is useful so that we can determine the endpoint's identity and use that information to set up the logging. For instance, we may want to configure log files to be written to a UNC path in a directory based on the endpoint name.



Unlike normal profiles, only one logging profile can be activated, as it wouldn't make sense to configure logging more than once. Also, if the `EndpointConfig` class implements `IWantCustomLogging`, then that implementation is controlling and no logging profile will be used.

## Customizing the log level

If you're familiar with `log4Net` or `NLog`, you may be looking for a large configuration block to change the log level, but you won't find it. In part to combat the sixth fallacy of distributed computing – there is one administrator – `NServiceBus` defines most of the logging configuration in code where it is up to the application developer to define. The one bit left to the system administrator is the log level, which the system administrator must be able to adjust at runtime to diagnose an issue.

By default, `NServiceBus` will log at the `INFO` threshold, but this can be adjusted using the following configuration:

```
<!-- Configuration Section -->
<section name="Logging"
    type="NServiceBus.Config.Logging, NServiceBus.Core"/>

<!-- Configuration Element -->
<Logging Threshold="WARN" />
```



The log levels used by NServiceBus, from least severe to most severe, are DEBUG, INFO, WARN, ERROR, and FATAL.



You could also generate the Logging section using the PowerShell `Add-NServiceBusLoggingConfig` cmdlet.

## Managing configurations

Once you begin releasing code out of development, you will need a way to manage configuration changes. When we develop a system we generally run all the endpoints and have all of our message queues on our local machine. When we deploy to new environments generally this changes, and we need to make changes to our configuration.

Our first option is to modify the actual configuration file. As an NServiceBus solution can begin to comprise many different endpoints, I urge you not to try to do this manually.

Visual Studio 2010 introduced a simple method for transforming a `Web.config` file when publishing a website called XML Document Transform. Unfortunately, this only supported `Web.config` files, and `App.config` files were left in the cold. Luckily this problem has been rectified, not by Visual Studio itself, but by a third-party add-on.

The SlowCheetah plugin is an open source Visual Studio 2010/2012 extension that allows you to transform `App.config` files as well as `Web.config` files, using the same transformation language. I encourage you to use this or another similar tool to transform your configuration files for different environments as part of an automated build and deploy process.

There may also be situations where you will want a different way to specify configuration information, perhaps stored in a centralized database. In order to support this, NServiceBus provides the `IProvideConfiguration<T>` interface, which you can implement to provide the information that would normally live in the `App.config` file.

As an example, this class provides the `TransportConfig` that specifies the number of message retries.

```
public class ProvideTransportConfig :
    IProvideConfiguration<TransportConfig>
{
    public TransportConfig GetConfiguration()
```

```
    {  
        return new TransportConfig { MaxRetries = 3 };  
    }  
}
```

Note that you'll have to add a reference to `System.Configuration` in order to reference the configuration classes that inherit from `ConfigurationSection`. Also, this class will be called multiple times—whenever `NServiceBus` needs the information from the configuration section—so you need to be prepared to cache the data if it is expensive to create; for example, data from a database or web service call.

## Monitoring

`NServiceBus` makes it easy to monitor the performance of any endpoint that has the performance counters enabled, which includes any endpoint installed with the production profile. After having installed `NServiceBus` and a service with counters enabled, you will find these counters in the `NServiceBus` category in the Windows Performance Monitor:

- Number of message failures per second
- Number of messages pulled from the input queue per second
- Number of messages successfully processed per second
- Critical time
- SLA Violation Countdown

The last two are especially interesting. critical time is the age of the oldest message in the queue, or in other words, the length of the queue's backlog in seconds. This is important because it is how business stakeholders will judge the capability of your system. In a messaging system the business probably doesn't care what your overall throughput is; what they really care about is that their work gets done within a length of time that they deem reasonable. Critical Time gives you the ability to measure that.

If the business does give you a hard and fast requirement for how fast messages must be processed, we can codify that in our system as a **Service Level Agreement (SLA)** and then as load and the critical time starts to increase, `NServiceBus` will estimate how long it will take before that SLA is breached and deliver that information to us in the form of the SLA Violation Countdown performance counter. This provides a common measurement (in the form of time) to monitor all your endpoints, regardless of their individual SLA settings.

We can define an endpoint's SLA by decorating the `EndpointConfig` class with the `EndpointSLAAttribute`, specifying any string that can be converted to a `TimeSpan`.

Here we define an SLA of one minute.

```
[EndpointSLA("0:01:00")]  
public class Endpoint : IConfigureThisEndpoint, AsA_Publisher  
{  
}
```

Armed with the critical time and an SLA violation countdown, it would even be possible to create a system that is capable of automatically provisioning more workers to handle unexpected loads.

While this information can be very useful, none of that will matter if nobody is listening. It's critically important to work with your IT department to establish monitoring for these performance counters and establish monitoring practices as a regular part of your service deployment process. All performance counters are available via Windows Management Instrumentation (WMI) which makes it very easy to integrate with almost any existing monitoring infrastructure.

A new tool called ServicePulse is currently being developed by Particular Software to help system administrators track and manage these SLA metrics. ServicePulse is currently in pre-beta at the time of this writing.

## Scalability

One of the greatest strengths of NServiceBus is that it allows you to easily add scalability to any system. We can easily add more threads to scale up, or distribute the processing of messages among multiple servers to scale out.

## Scaling up

Multithreading is hard. If you've ever tried to design a multithreaded program then you already know this. NServiceBus makes this a lot easier because it has already divided up all our work into messages that represent discrete, independent tasks. So, as long as we don't do anything stupid in our message handlers (such as sharing state) then we can allow NServiceBus to ramp up the number of threads processing messages with relative safety.

This gives service applications the same flexibility that web servers enjoy. In the same way that HTTP requests are processed by a web server largely in parallel by multiple threads, an NServiceBus endpoint can process messages on multiple threads. The difference is that a web server must process all incoming requests immediately and must spin up more threads in the thread pool to handle an increased load. If there's too much load on a web server, everything jams up.

In an NServiceBus endpoint, we have the luxury of a queue that will hang on to all the incoming messages until we're ready for them. So, unlike a web server, we set a fixed number of threads that are available to process messages, and if there is increased load we don't crash our server, it just takes a little longer for messages to be processed.

To scale up an endpoint, we'll revisit the `TransportConfig` element we first learned about in *Chapter 3, Preparing for Failure*. In the following example, we configure the thread count (the concurrency level) to 3 using the `MaximumConcurrencyLevel` attribute:

```
<TransportConfig MaxRetries="5"
  MaximumConcurrencyLevel="3"
  MaximumMessageThroughputPerSecond="3" />
```

We will also take this opportunity to demonstrate another useful parameter of the `TransportConfig` element, `MaximumMessageThroughputPerSecond`. If omitted, this is set to zero and means that message throughput is limited only by the capability of the hardware. However, we may want to limit throughput artificially if, for example, we are integrating with a third-party web service that imposes a maximum number of requests per second. This automates the process of managing a throughput quota, which can be very difficult to do manually, especially when using multiple processing threads.

## Scaling out

When a website has too much load to be handled by one server, we use a network load balancer to distribute incoming requests to multiple servers. NServiceBus offers a similar method of work distribution out of the box. We don't even need a separate load balancing appliance or server; all the smarts for load balancing are built right into the NServiceBus host process.



People commonly ask why they can't just use a network load balancer to scale out an NServiceBus endpoint the same way they would scale out a web server. NServiceBus with the MSMQ transport uses transactional messaging, and transactional messaging does not work with network load balancers because the acknowledgements can't be returned to the sending machine once the IP address of the sender is masked by the load balancer.

Other transports besides MSMQ use more of a broker style of communication, so this type of scale-out is not required at all. For these broker-style transports, you can simply install the same handler endpoint on additional machines all reading from the same input queue in a competing consumers pattern.

A network load balancer simply assigns incoming requests to a destination server as soon as it arrives. Scaling an NServiceBus endpoint works a little differently through the cooperation of **Distributor** and **Worker** components.

The distributor maintains three queues, as follows:

- An **input queue**, to receive incoming messages
- A **control queue**, to receive notifications that a worker is ready to process a message
- A **storage queue**, to keep track of workers that are idle when no work is available

When a worker is ready to process a message, it will send a ReadyMessage (they look like empty messages with additional control headers on the wire) to the distributor's control queue. If the distributor has a message waiting in its input queue, it will forward that message to the worker's private input queue and the worker will process it. If the distributor has no remaining work, it will store the ReadyMessage in its storage queue so that it knows which workers are available when the next message arrives. The storage queue also enables the distributor to remember which workers are available in case the distributor is restarted.

Any NServiceBus endpoint can operate as either a distributor or a worker by enabling the `Distributor` or `Worker` profiles on the command line when the service endpoint is installed.

In addition to the distributor and worker profiles, there is a third profile that combines the two. A **master node** is an endpoint that acts both as distributor and worker, taking on both the tasks of distributing work between workers, and also acting as a local worker, all in the same process.

A master node offers the ultimate in scaling flexibility. You can deploy an endpoint as a master node to start out, and then as system load increases you can easily provision additional worker nodes pointing back to the original master.

In order to find its master node, all a worker needs to know is the name of the server its master is hosted on.

```
<!-- Config Section -->
<section name="MasterNodeConfig"
  type="NServiceBus.Config.MasterNodeConfig, NServiceBus.Core" />

<!-- Config Element -->
<MasterNodeConfig Node="MasterNodeServer"/>
```



You could also generate the `MasterNodeConfig` section using the PowerShell `Add-NServiceBusMasterNodeConfig` cmdlet.

NServiceBus makes the assumption that the endpoint name will be the same on all servers, so as long as it knows the server where the master node is hosted, it can deduce the names of the master node's control queue and data queue, which are the only queues the worker needs to be able to communicate with the master.

If you are the kind of person who simply must go against the grain, or if you just want to try running the distributor and workers on just one computer for educational value, you'll need to be able to specify the addresses of the master node's control queue and data queue explicitly. You can do this via optional attributes on the `UnicastBusConfig` element we're already familiar with.

```
<UnicastBusConfig
  DistributorControlAddress="Master.Control@MasterServer"
  DistributorDataAddress="Master@MasterServer">
  <MessageEndpointMappings>
    <!-- Normal message routing entries -->
  </MessageEndpointMappings>
</UnicastBusConfig>
```

The real elegance of the NServiceBus master node and worker model is that you deploy the exact same package to every single server that runs the endpoint. The code is the same, and the configuration is the same. The only thing that's different is the profile that you specify on the command line when installing the service. This really makes code updates a snap!

## Decommissioning a worker

If you need to decommission a worker node, some steps need to be taken to do so without losing data.

1. Shut down the worker node's service.
2. Any `ReadyMessages` already sent to the master node will make the master send more work to the worker, so wait for this to stabilize and then manually move any message(s) left in the worker's local input queue to the distributor's input queue.
3. If the endpoint is not high traffic, this may be too long to wait. In this case, you can clear the master node's storage queue, restart all the other workers so they resend their `ReadyMessages`, and then complete step 2.

Future versions of NServiceBus aim to improve this process.

## Multiple sites

Most queuing technologies can only operate within a local network, which presents some problems when communication is needed between geographically separated sites. A canonical example is a headquarters site that must exchange messages with regional offices.

The best approach to geographic separation is to establish a VPN connection between sites. As far as MSMQ is concerned, a VPN makes geographically separated sites all part of the same local network and NServiceBus can operate more or less normally. Some messages will have a little farther to travel than others, of course, and MSMQ's built-in store and forward capability will cover instances when the VPN connection is not always reliable.

The reality, however, is that a VPN is not always a possibility. Failing the availability of a VPN, the only method we can reliably use to communicate between sites and through firewalls is HTTP.

For this, NServiceBus includes the **Gateway** component out of the box. The Gateway takes care of communicating with remote endpoints via HTTP (or HTTPS) so that you don't have to waste time exposing custom web services for this purpose.

As we learned in *Chapter 3, Preparing for Failure*, communicating over HTTP is error-prone, but NServiceBus takes care of this for us. It includes a hash with each message to prevent transmission errors, automatically retries failed messages, and performs deduplication to ensure messages are delivered once and only once.

The gateway does come with some limitations:

- Not all messages are transmitted over the gateway. Transmitting a message to a remote site requires you to opt in by specifying the names of the destination sites.
- Because only select messages are transmitted over the gateway, you should create special messages whose only purpose is this inter-site communication.
- Publish and subscribe is not supported across site boundaries.
- The gateway can only be used to bridge the gap between logically different sites. It cannot be used to facilitate disaster recovery scenarios where the remote site is a copy of the primary site. Use your existing IT infrastructure (SAN snapshots, SQL log shipping) for these scenarios instead.

To send a message over the gateway, we opt in using the `SendToSites` method.

```
Bus.SendToSites(new[] { "SiteA", "SiteB" }, crossSiteCmd);
```

As a bonus, the Gateway includes enough header information with the message so that the receiving end can send a reply message with the standard `Bus.Reply()` method.

The `App.config` file stores the incoming URLs that the gateway's HTTP server will use to listen for incoming messages, as well as the outgoing URLs that correspond to each remote site name. This allows an administrator to update the URLs in configuration without requiring a code update.

Configuring the gateway and securing it with HTTPS (which is optional, but highly recommended) is an advanced topic which is beyond the scope of this book. For more information, check out the gateway's documentation on the Particular website, or explore the gateway sample included with the NServiceBus installation package.

## Managing RavenDB

As mentioned several times during this book, by default NServiceBus will use RavenDB for many of its own data storage needs, including storing subscriptions, saga data, timeout information, and a few other things. RavenDB is an open source second-generation document database that is under OSS and commercial license.

Your NServiceBus license includes a commercial license for RavenDB that is free of charge for the NServiceBus specific storage needs like the ones mentioned above. If you want to use RavenDB for your application data you will need to license RavenDB separately.

While a complete tutorial on RavenDB is not within the scope of this book, it is worth mentioning that if you installed NServiceBus with default options, you can access the RavenDB Management Studio at <http://localhost:8080>. Unlike databases that require a special management application separate from the database, Raven Studio is deployed as a Silverlight application accessible via the database server itself.

In many of the examples in this book, we have added `NServiceBus.Lite` to the command line arguments for a service endpoint to invoke the Lite profile and gain the simplicity of storing everything in memory. If we remove that, however, the endpoint will run under the Production profile and use RavenDB for storage, and then we can poke around the Raven Studio and gain valuable insights on how NServiceBus works.



From the landing page of RavenDB Studio, you can switch between databases from a menu near the upper-right corner. Each service endpoint creates its own Raven database. Once you have selected a database, navigate to the **Collections** tab to view documents grouped by their entity type, then select a collection to view documents in that collection in the grid below.

For the most part, Raven doesn't need to be actively managed at all. The most common task where manual intervention might be required would be to remove the Subscription document for an endpoint that no longer exists, but this is rare.

For more information on RavenDB, visit <http://ravendb.net/>.

## Virtualization

The single most important investment you can make in your infrastructure has nothing to do with NServiceBus at all, and everything to do with hardware virtualization.

The whole point of NServiceBus is to provide transactional messaging so that you don't lose data. This is moot if you host NServiceBus on a physical server that could go up in smoke at any moment, taking its messages down with it. With a properly configured virtualized environment, this risk basically disappears. In the event of a hardware failure, the hypervisor should be able to migrate the virtual machine image to another host, often completely transparently, where it will resume processing messages without missing a beat.

Besides hardware failures, the other main cause of catastrophic system failures is driver problems. In most cases, virtualization removes this problem as well because the drivers are abstracted, generic drivers managed by the virtualization platform, in order to support hosting the guest system on different host architectures. In most cases, an incompatibility like this would only happen when applying system updates such as service packs. In these cases, with virtualization you can make a complete backup of the virtual machine before applying the update and restore it to its original state in the case of a failure.

## Message storage

Whether or not you decide to take advantage of hardware virtualization, it's a smart idea to change the storage location for your messages to a SAN with its own built-in redundancy. If you do not virtualize, this will make it easier to recover your messages in the event of a system failure, or to reattach the storage to a new host that can take over processing for a failed host.

If you do virtualize, your virtual systems are probably already stored on a SAN, so you may be wondering what the benefit of separate storage would be. Keeping messages stored on separate storage from the OS creates less data churn on the drive and can make it easier to perform hot backups of the OS partition without taking the virtual server offline.

To change the storage location for MSMQ, right-click on the **Message Queuing** manager in **Computer Management** and select **Properties**. The relevant paths can be modified on the **Storage** tab.

## Summary

In this chapter we learned how to manage an NServiceBus system in a production environment. We learned how to use the NServiceBus host's ability to install as a Windows Service and how we can use profiles to modify how the host runs in different environments. We also learned how to write our own code to target different profiles, and how to create our own custom profiles.

We learned how to manage configuration as we deploy to new environments, and how we can provide that configuration information programmatically, even loading it from a centralized database.

Next we learned how to monitor a production endpoint using NServiceBus performance counters, and how to define an SLA for an endpoint programmatically. In order to make sure we can meet that SLA, we learned how to scale an endpoint. We learned how to scale up by increasing the maximum concurrency level for an endpoint, and how to scale out by using the distributor to process messages on multiple servers.

In order to bridge the gap between logically different, geographically separated sites, we learned about the Gateway component and its capabilities. Then we ended the chapter by discussing the importance of hardware virtualization to create a reliable infrastructure that we cannot attain with non-virtualized servers.

In the next chapter we will review what we have learned in this book and cover resources where you can find more information.



# 8

## Where to Go from Here?

This book was not meant to be an exhaustive guide on every single gritty detail of NServiceBus, let alone the theories of service-oriented architecture that are the underpinnings of its architecture. Instead, this book aims to be more of a guide to the essentials that will get you off to a running start, creating your own reliable distributed systems as quickly as possible.

As a result of this approach, there is of course more to learn, but let's pause for a moment to take stock of what we have covered within these pages.

### What we've learned

If you recall back at the beginning of *Chapter 1, Getting on the IBus*, I shared several stories of problematic development scenarios, many culled from my own personal experiences. Let's take a look back and reflect on how NServiceBus could come to bear in each situation.

The first developer was getting deadlocks when updating values in several database tables within a transaction. With NServiceBus, we would separate this action into many separate commands, each of which would update the values in one table. The database would be technically inconsistent for a short period of time until all the commands completed execution, but our business stakeholders would most likely be perfectly accepting of this, especially if it means fewer errors encountered by users.

The second developer was losing orders and revenue because of a failure on a third of every three database calls, causing the transaction to roll back, an ugly error, and the user taking their money elsewhere. With NServiceBus, the three database calls would again be handled in a message handler, separate from the website code. During the times when the third call threw an error, automatic retries would kick in, until it was processed correctly. Best of all, the web tier only had to send a command and then was able to report back to the user that the order was accepted, even though all the work hadn't technically been done yet!

The third developer had created an image processing system that was growing too big for the hardware it was running on. With NServiceBus, we would replace the app with an endpoint that processes one image per command. We would configure this endpoint as a Master Node so that it had the capability to scale out. If the load increased a little, we could scale up by increasing the maximum concurrency level to use more processing threads on that machine. If the load increased a lot, we could scale out by adding another Worker endpoint on a different server. We would define the SLA on the endpoint according to our business needs, and we would use the NServiceBus performance counters to ensure we were meeting that SLA, so that we would know when we needed to add more processing resources.

The fourth developer was integrating with a third-party web service and also updating a local database, and was noticing that the data got out of sync as a result of the web service timing out. With NServiceBus, we send a command to update the local database, and once that completes, send a new command to call the web service. When the web service times out, automatic retries ensure that the call is completed successfully.

The fifth developer was sending emails as part of a lengthy business process, and a naïve retry policy was causing end users to get multiple copies of the same email. With NServiceBus, sending the email is handled within a separate handler, so it is isolated from the rest of the business processes and will be sent only once. The part of the business process that is failing will benefit from automatic retries.

The sixth developer had a webpage kick off a long-running backend process, while the browser displayed an interstitial page similar to when you search for flights on a travel site; however, because of unreliable messaging techniques, the integration is brittle and the backend process didn't always execute as planned. With NServiceBus, we can guarantee that the command we send from the website will get picked up by the backend process. The web application subscribes to an event published when the process completes, ensuring that even if the site is served by a web farm, every server will know when the process is finished. The browser can then use either traditional polling via jQuery or a library such as **SignalR** to determine when the time is right to advance to the next page.

The seventh developer added the fields `latitude` and `longitude` to a customer database and now is having difficulty keeping the geolocation information up-to-date with address changes. With `NServiceBus`, we would create one command to update the customer's address, which would update the address and then publish an event to announce the address had changed, which would allow every other department to do what they needed to do to support the change. We would also create a subscriber that would geocode the address and update the latitude and longitude, ensuring that every time the address was updated, the geolocation would be updated as well.

## What next?

- There are many places to go to find out more about `NServiceBus`. Of course I would be remiss if I did not first mention the official `NServiceBus` documentation:  
<http://www.particular.net>
- `NServiceBus` boasts a very active and very friendly user community. You can find them at the official `NServiceBus` discussion group at Yahoo! Groups:  
<http://tech.groups.yahoo.com/group/nservicebus/>
- If you're not in the mood for discussion, Stack Overflow may be a better place to go for any question that needs an answer. Be sure to tag your questions with the `nservicebus` tag. There are several community members (myself included) who troll this tag looking to help out others in need. Also be sure to read other tagged questions. There's a good chance your question has been asked before:  
<http://stackoverflow.com/tags/nservicebus>
- Remember that `NServiceBus` is open source! If you like, you can do a deep dive into the source code by looking up the `NServiceBus` GitHub repository:  
<https://github.com/NServiceBus/NServiceBus>
- Lastly, be sure to check out the `NServiceBus` Champions, a worldwide group of `NServiceBus` community leaders. It is definitely worth your time to follow them on Twitter or read their personal blogs:  
<http://particular.net/champions>



# Index

## Symbols

<add/> element 31  
.And() method 48  
.AndThen<THandler>() method 68  
-Cmd suffix 25  
.CreateBus() extension method 54  
.DefaultBuilder() method 49  
.DefiningMessagesAs() 88  
-Event suffix 25  
-install flag 98  
.NinjectBuilder() 50  
-password parameter 96  
.SendOnly() method 55  
-sideBySide parameter 96  
.UnicastBus() extension method 53  
-username parameter 96  
.UseTransport<ActiveMQ>() method 51  
.UseTransport<Msmq>() 50  
.UseTransport<TransportType>() method 50  
.With() method 48

## A

ACID 22  
ActiveMQ 51  
Add-NServiceBusLoggingConfig cmdlet 102  
Add-NServiceBusMasterNodeConfig cmdlet 107  
Add-NServiceBusMessageForwardingInCaseOfFaultConfig cmdlet 17  
Add-NServiceBusSecondLevelRetriesConfigcmdlet 39

Add-NServiceBusTransportConfigcmdlet 36  
Add-NServiceBusUnicastBusConfig cmdlet 17, 28  
Advanced Message Queuing Protocol (AMQP) 51  
Aggregate Root 89  
AllAssemblies.Except() directive 91  
App.config file 11, 16, 28, 40, 47, 82, 102, 109  
applicative message mutator 71  
assembly  
  scanning 48  
Atomicity 22  
Availability 23

## B

BASE 23  
batch jobs 79  
business  
  retraining 90  
business logic 87, 88  
Bus instance 81  
bus options 53  
Bus property 15, 81  
Bus.Reply() method 109  
Bus.Send() 88

## C

CAP 23  
Checkout button 34  
code  
  obtaining 7, 8  
commands 21



- commands versus events**
  - consistency, achieving with messaging 23, 24
  - eventual consistency 22, 23
- configSection/section element 28**
- ConfigUnicastBus class 53**
- configuration**
  - managing 102, 103
- Configure.Component method 60**
- ConfigureHowToFindSaga() method 82, 83, 88**
- Configure instance 54**
- ConfigureMapping<TMessageType> method 82**
- Configure method 101**
- ConfigureProperty method 60**
- Configure.With() 48, 49, 61**
- Configure.With() block 61**
- ConnectionString property 60**
- Consistency 22, 23**
- controller pattern 89**
- control queue 106**
- CreateNewUserCmd class 26**
- CreateNewUserCmd message 80, 92**
- CreateNewUserWithVerifiedEmailCmd message 84**

## D

- DataBusProperty<T> class 75**
- DataStore object 60**
- DateTime object 86**
- DateTimeProvider class 60**
- deadlock 35**
- demo**
  - retrying 40
- Dependency injection 49, 50, 59, 60**
- DependencyLifecyle.SingleInstance 60**
- design guidelines**
  - business, retraining 90
  - business logic 87, 88
  - saga lifetime 88
  - saga patterns 89
- Distributed Transaction Coordinator (DTC) 51**
- distributor component**
  - control queue 106

- input queue 106
- storage queue 106
- Distributor profile 98**
- Domain Driven Design 89**
- DoSomethingSpecial2 method 64**
- Durability 22**

## E

- EmailAddress property 81**
- EmailSender class 28**
- EndpointConfig class 29, 49-51, 58, 67, 68, 100-103**
- endpoint name 49**
- environmental profiles 97**
  - Integration profile 98
  - Lite profile 97
  - Production profile 98
- error messages**
  - retrying automatically 36
- error queue**
  - repeated errors, sending 37
- events**
  - about 24, 25
  - as interfaces 66
  - publishing 25-27
  - subscribing to 27-29
  - versus commands 21
- eventual consistency 22, 23**
- Expect methods 92**
- Express attribute 63, 64**
- express messaging 40, 41**

## F

- fault tolerance 34, 35**
- feature profiles**
  - about 98
  - Distributor profile 98
  - Master profile 98
  - MultiSite profile profile 98
  - PerformanceCounters profile 98
  - Worker profile 98
- FileShareDataBus 75**
- fluent configuration 47**

## G

gateway  
  about 108  
  disadvantages 108  
GET request 16  
Global.asax.cs file 15  
Guid property 25

## H

Handle(CreateNewUserCmd message)  
  method 86  
Handle method 12, 26, 81, 84  
handler order  
  specifying 67, 68  
HomeController class 15, 29

## I

IBus instance 55  
IComponentConfig instance 60  
IConfigureThisEndpoint 58  
idempotent 44, 45  
IHandleTimeouts<T> interface 86  
IMessageMutator 72  
IMutateIncomingMessages 72  
IMutateIncomingTransportMessages 72  
IMutateOutgoingMessages 72  
IMutateOutgoingTransportMessages 72  
IMutateTransportMessages 72  
INeedInitialization 58  
Initialize() method 91  
Init() method 15, 57  
input queue 106  
install package 8  
Integration profile 98  
IProvideConfiguration<T> interface 102  
Isolation 22  
IsReminder property 82  
IStartableBus instance 54  
ITimeProvider parameter 60  
IUserCreatedEvent interface 25  
IWantCustomInitialization 58  
IWantCustomLogging 58  
IWantTheEndpointConfig interface 100  
IWantToRunAtStartup 59  
IWantToRunBeforeConfiguration 58

IWantToRunBeforeConfigurationIsFinalized 58  
IWantToRunWhenBusStartsAndStops 59  
IWantToRunWhenBusStartsAndStops class 93  
IWantToRunWhenConfigurationIsComplete 58

## J

Java Message Service (JMS) 51

## K

KnownTypesAttribute 61

## L

license developer  
  URL 19  
Lite profile 97, 98  
log level  
  customizing 101, 102

## M

MarkAsComplete() method 84, 87  
master node 106  
Master profile 98  
MaximumConcurrencyLevel attribute 105  
message  
  actions 68  
  auditing 42, 43  
  deferring 69  
  expiring 41, 42  
  forwarding 69  
  headers 69  
  sending, from MVC application 13  
  serializer 61  
  stopping 68  
  versioning 64  
MessageConventions.cs 63  
message handler  
  creating 11, 12  
message mutators 71  
message routing 30, 31  
messages assembly  
  creating 9, 10

- message storage 110
- messaging service
  - consistency, achieving with 23, 24
- Microsoft Message Queuing. *See* MSMQ
- MockTimeProvider 59
- MSMQ 7, 50
- multiple sites 108, 109
- MultiSite profile profile 98
- MVC application
  - message, sending from 13
- MVC website
  - creating 13-17

## N

- NServiceBus
  - about 5-7, 57, 113, 114
  - host, modifying 57
  - transaction settings 62
  - Yahoo! Groups, URL 115
- NServiceBus.ActiveMQ package 51
- NServiceBus Champions
  - URL 115
- NServiceBus.Configure 57
- NServiceBus documentation
  - URL 115
- NServiceBus GitHub repository
  - URL 115
- NServiceBus.Headers class 70
- NServiceBus.Host package 8
- NServiceBus.Interfaces package 8, 9
- NServiceBus.Logging namespace 80
- NServiceBus.Ninject package 50
- NServiceBus NuGet packages
  - NServiceBus.Host package 8
  - NServiceBus.Interfaces package 8
  - NServiceBus package 8
  - NServiceBus.Testing package 9
- NServiceBus package 8
- NServiceBus.Saga namespace 80
- nservicebus tag 115
- NServiceBus.Testing package 9, 90

## O

- observer pattern 89

## P

- Partition 23
- payloads
  - transporting 73, 75
- performance monitoring 103, 104
- PerformanceCounters profile 98
- poison messages 35
- Polymorphic
  - dispatch 64, 66
- processes
  - running 79
- Production profile 98
- profiles
  - about 97
  - environmental profiles 97
  - customizing 99, 100
  - feature profiles 98
  - logging 101
- property encryption 72, 73

## Q

- queue
  - purging, on startup 52

## R

- RabbitMQ 51
- RavenDB
  - managing 109, 110
  - URL 110
- Recoverable messages 41
- remote procedure call (RPC) 21
- ReplyMsg class 88
- ReplyToOriginator() method 88
- RequestTimeout method 86
- ReturnToSourceQueue.exe 37
- Run() 57

## S

- saga
  - about 80-82
  - data, finding 82
  - ending 83, 84
- SagaKey 83
- saga lifetime 88

- saga patterns
  - about 89
  - URL 89
- Saga<T> class 81
- scalability
  - scaling out 105, 106
  - scaling up 104, 105
- scaling out 105-107
- scaling up 104, 105
- scheduled tasks 79
- scheduler 93, 94
- Second Level Retries. *See* SLR
- send-only endpoints 54
- SendToSites method 108
- SendVerificationEmailCmd command 82
- SendVerificationEmailCmd message 92
- service
  - installing 95-97
- service endpoint
  - creating 10, 11
- ServiceInsight
  - URL 38
- Service Level Agreement (SLA) 103
- Service-oriented architecture(SOA) 5
- SignalR 114
- SLR 38, 39
- solution
  - executing 17-20
- SQL Server 52
- Stack Overflow
  - URL 115
- Start Debugging button 17
- Start menu 37
- Start() method 59, 75, 94
- startup
  - about 53
  - queue, purging on 52
- Stop() method 59
- storage queue 106
- store and forward 40

## T

- third-party web service
  - working 24

## time

- dealing with 85, 86
- TimeSpan object 86
- TimeToBeReceived attribute 42, 63, 74
- TimeToBeReceivedOnForwardedMessages
  - parameter 43
- ToSaga method 82
- transaction processing 34, 35
- TransportConfig element 105
- transport message 71
- transport message mutator 72

## U

- Unique attribute 81
- UnitOfWork 70
  - execution 71
- unit testing 90-92
- Unobtrusive mode 62, 64
- User Account Control (UAC) 97
- UserCreator class 26, 80, 81
- UserCreator.cs file 84
- UserService endpoint 83
- UserService window 29
- UsingTransport<ActiveMQ> interface 51
- UsingTransport<RabbitMQ> interface 51

## V

- VerifyUserEmailPolicy class 86
- VerifyUserEmailPolicy.cs file 86
- virtualization
  - about 110
  - message storage 110

## W

- Web.config file 16, 28, 47, 83, 102
- web service
  - exposing 76, 77
  - integrating 43-45
- WelcomeEmailService window 29
- When() method 92
- Windows Azure 52
- worker node
  - decommissioning 107
- Worker profile 98





## Thank you for buying **Learning NServiceBus**

### **About Packt Publishing**

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

### **About Packt Open Source**

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### **Writing for Packt**

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

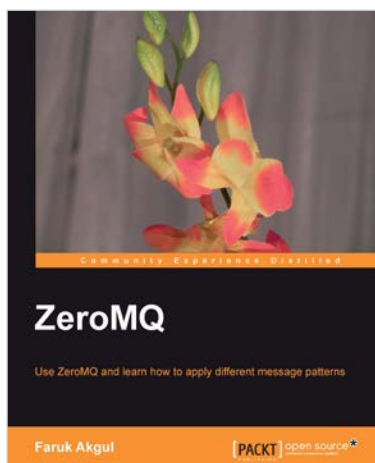


## Instant RabbitMQ Messaging Application Development

ISBN: 978-1-78216-574-3      Paperback: 54 pages

Build scalable message-based applications with RabbitMQ

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn how to build message-based applications with RabbitMQ using a practical Node.js ecommerce example
3. Implement various messaging patterns including asynchronous work queues, publish subscribe and topics.



## ZeroMQ

ISBN: 978-1-78216-104-2      Paperback: 108 pages

Use ZeroMQ and learn how to apply different message patterns

1. Learn fundamental message/queue design patterns
2. Work with multi-threaded programs
3. Work with multiple sockets

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

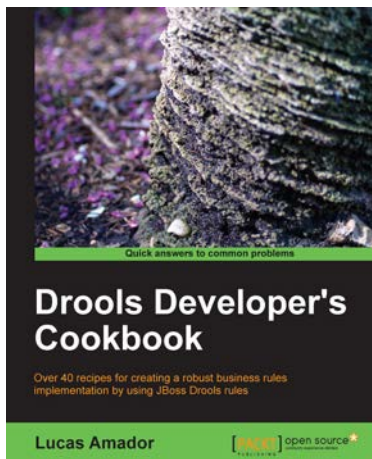


## Oracle Service Bus 11g Development Cookbook How-to

ISBN: 978-1-84968-444-6 Paperback: 522 pages

Over 80 practical recipes to develop service and message-oriented solutions on the Oracle Service Bus

1. Develop service and message-oriented solutions on the Oracle Service Bus following best practices using this book and ebook
2. Extend your practical knowledge of building solutions on the Oracle Service Bus
3. Packed with hands-on cookbook recipes, with the complete and finished solution as an OSB and SOA Suite project, made available electronically for download



## Drools Developer's Cookbook

ISBN: 978-1-84951-196-4 Paperback: 310 pages

Over 40 recipes for creating a robust business rules implementation by using JBoss Drools rules

1. Master the newest Drools Expert, Fusion, Guvnor, Planner and jBPM5 features
2. Integrate Drools by using popular Java Frameworks
4. Part of Packt's Cookbook series: each recipe is independent and contains practical, step-by-step instructions to help you achieve your goal.