

Learning NServiceBus

Second Edition

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

Foreword by Udi Dahan, Founder and CEO, Particular Software

David Boike

[PACKT] PUBLISHING enterprise professional expertise distilled

Learning NServiceBus

Second Edition

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

David Boike

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

Learning NServiceBus

Second Edition

Copyright © 2015 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

Second edition: January 2015

Production reference: 1250115

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

ISBN 978-1-78439-292-5

www.packtpub.com

Credits

Author
David Boike

Reviewers
Prashant Brall
Roy Cornelissen
Hadi Eskandari
Daniel Marbach

Commissioning Editor
Dipika Gaonkar

Acquisition Editor
Larissa Pinto

Content Development Editor
Kirti Patil

Technical Editor
Chinmay S. Puranik

Copy Editors
Dipti Kapadia
Vikrant Phadke

Project Coordinator
Kranti Berde

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Bernadette Watkins

Indexer
Monica Ajmera Mehta

Graphics
Sheetal Aute
Abhinash Sahu

Production Coordinators
Arvindkumar Gupta
Nilesh R. Mohite

Cover Work
Arvindkumar Gupta
Nilesh R. Mohite

Foreword

Unlike many people who write a foreword for a book, I have myself not actually written a book yet, so I am probably going about this the wrong way. Also, of all the books that I have read, I have almost never read the foreword, which makes me wonder who is actually reading this.

That being said, I think that 10 years of blogging has actually prepared me very well to write this foreword, and it will end up being roughly as long as my regular blog posts.

In any case, I am extremely happy to see how well the first edition of this *Learning NServiceBus* book has done, and when David asked me to write the foreword for this second edition, I was more than willing to oblige.

Now that you've picked up this book, I think it's fair to assume that you have heard something about NServiceBus, maybe even downloaded it, played with it, or used it on a project or two. What you might not know is the story of how it all began and how we ended up where we are today—seeing the second edition of *Learning NServiceBus* being published.

Early influences

Almost 15 years ago, I was working as a developer on a gigantic command and control system developed in a language called Ada. This is what Wikipedia has to say about Ada:

> "*[Ada] has built-in language support for explicit concurrency, offering tasks, synchronous message passing, protected objects, and non-determinism.*"

These were very important attributes for a system that needed to be super reliable, highly performant, and very scalable. Little did I realize at that time how profound an effect this relatively unknown programming language would have on the rest of my career.

In my next company, a software development firm—the kind that does turnkey projects for its customers—I continued my focus on command and control systems, this time using the fairly new .NET platform from Microsoft. The thing was that many of the abilities that I'd come to rely on in Ada were (to me) curiously absent from .NET.

Like any good craftsman who has gotten used to a certain tool, I started looking around for a replacement, and when I couldn't find one, I went and made it myself.

Starting with MSMQ

Although the main message from Microsoft at the time was all about XML Web Services and .NET Remoting, I found that there was support for a message passing model in .NET that was actually built on a piece of infrastructure that was considerably older (originally built in the NT 3.5.1 days, but released with NT 4.0 in 1997). At that time, MSMQ had made it to version 3.0 which, later on, I learned was the magic number when it came to stability of software built by Microsoft.

Still, I found the System Messaging API to be a little too *close to the metal* for me to feel comfortable using it directly from my application logic, so I created a wrapper library, which gave me some much needed abstraction.

As I moved from one project to the next, I found this little library to be more and more useful and kept on extending and refining it. The developers in my teams seemed to like working with it as well.

After the third or fourth project, I decided to approach the company's management, suggesting that we should consider turning this infrastructure into a product. Seeing how useful it was to us on our projects, I figured that it was a no-brainer.

It turned out that this company had gone down the productization path in the past and had gotten badly burned in the process. As a result, they were very hesitant to make the same mistake again. Realizing that it wasn't going to happen, I managed to get permission to release the infrastructure to the world as an open source project, but only after legal confirmation that there would be no support or other liabilities to the company.

Thus, in 2007, NServiceBus was born.

This also happened to be the time when I started transitioning into my own private consulting practice.

What open source meant in those days

As I continued to use NServiceBus in my work projects, I kept tweaking and expanding it. Version numbers didn't mean a whole lot, and through 2008, the version progressed rapidly from 1.6.1 to a release candidate of 1.9. If it worked on my project, I considered it worthy to ship.

Documentation was practically nonexistent and developers leaned pretty heavily on samples to figure out what the software did, asking questions on the discussion group when they ran into cryptic error messages (of which there were many).

In 2008, the 5-day format of my Advanced Distributed Systems Design course was launched, which was an extension of the 2-day workshop on the same topics I had been teaching since 2006. This course was based on many of the learnings from my old Ada days and continues to serve as the architectural underpinning of much of NServiceBus to this day.

April 2009 brought some stability with the final release of version 1.9 but came with a marked increase in the amount of consulting and training I was doing. This was great for me as I could finally afford to turn down work in order to continue the development of NServiceBus. This worked out great and in March 2010, the much-heralded NServiceBus 2.0 was finally released.

Clouds on the horizon

By almost every measure, things were going great. More companies were adopting NServiceBus and bringing me in for consulting. The Advanced Distributed Systems Design course was getting quite popular, and for a while, I was teaching it almost once a month in a different country around the world. Unfortunately, almost 6 months had gone by without any meaningful development on the code base.

Then it hit me that if another year or two would pass by in this manner, NServiceBus would start to get stale and the companies that had used it in their systems would eventually have to replace it with something else (costing them quite a lot of time and money).

I had lived through this story several times as a consumer of open source projects. As long as the project wasn't too intertwined in our system, it wasn't too painful to remove it. The thing was that NServiceBus was a fairly large framework that supported broad cross-sections of both client-side and server-side logic, so there would be no simple way to replace it.

I tried to figure out how I could guarantee that the development of the code base would stay a priority, yet as long as my primary source of revenue was services (consulting and training), I couldn't see how it would work. The only solution seemed to be to start charging money for NServiceBus licenses. Although larger companies were able to keep an open source core and sell other products around it, I hadn't seen or heard of any one-person operations that had been able to bootstrap their way into that.

I had no idea whether this would work, and whether the open source community that had supported me all this time would accept it or turn their backs on me, but I felt that it needed to be done. Unless there was revenue coming in directly from the features of the product, it wouldn't have a future.

Therefore, in late 2010, I founded the NServiceBus company and steeled myself for the worst.

Unexpected success

Seeing the overwhelmingly positive responses from the community was quite a surprise. Sure, there were those that grumbled, but only a handful ultimately decided to switch to something else.

I had made it—living the dream!

However, lest I paint an overly rosy picture, I knew nothing about running a product company. Pricing was harder than I ever imagined it would be. The legal factor was a "crazy-complex" where, even after the lawyers explained to me all about things such as indemnification, they told me that it was ultimately my decision whether to accept the client's terms or not.

Most importantly though, I felt that I had secured the future of NServiceBus. As long as there was money to be made from it, even if something happened to me, one of the other contributors to the project could afford to take it over.

Fast-forward to today

So much has happened since those early days of 2011 that it could probably fill its own book, and maybe one of these days, I'll put the proverbial pen to paper and make it happen. Anyway, here are the highlights:

- March 2012: NServiceBus 3.0 released. This includes official Azure support for the first time.

- July 2013: NServiceBus 4.0 released. This includes modeling and debugging tools.
- 2013: The company begins to rebrand itself as Particular Software.
- August 2013: The first edition of the *Learning NServiceBus* book comes out.
- November 2013: Monitoring tools released for NServiceBus.
- April 2014: The first release of the integrated Particular Service Platform.
- September 2014: NServiceBus 5.0 released, no longer depending on distributed transactions.

Also, I've got to tell you, the quality of each version has gone up dramatically over time. The level of testing that goes into each release is impressive—looping through every permutation of containers, persistence, transport, and even versions of supported operating systems and databases.

Back to David, and this book

When David wrote the first edition of this *Learning NServiceBus* book, it was something of a defining moment for NServiceBus—there was finally a book. The technology industry is awash in books about almost every piece of software out there, but for most of the time, NServiceBus was absent. Decision makers took us more seriously when we'd bring physical books with us for their teams.

Books matter!

With this latest edition, David goes beyond just covering the changes that happened in NServiceBus version 5.0, and goes even deeper into the reasoning behind those changes and why you'd want to use which features and when.

As one of the most prominent members of the NServiceBus community, David has interacted with the NServiceBus development team on a regular basis, given valuable feedback on our API, and debated with us on our future technology roadmap.

You're in for a treat.

Udi Dahan

Founder and CEO, Particular Software

About the Author

David Boike is a principal consultant with ILM Professional Services, with experience in building and teaching others how to build distributed systems. He is an NServiceBus Champion, official NServiceBus and RavenDB trainer, Xamarin Certified developer, and amateur beer brewer. He lives in the Twin Cities with his wife and two children.

About the Reviewers

Prashant Brall is a principal consultant and senior software architect and developer at Veritec (`www.veritec.com.au`) in Canberra, Australia. He has over 19 years of experience in application development, which includes 4 years of work in the USA for Fortune 500 companies and major financial corporations. As a software professional, Prashant loves crafting software and enjoys writing about his experiences on his blog at `https://prashantbrall.wordpress.com`. He has also reviewed *Instant AutoMapper* published by Packt Publishing.

In his leisure time, he enjoys watching movies with his wife and playing musical instruments such as piano and guitar.

A big thank you to my wife, Jhumur, for her love and support and for being my best friend. I would also like to thank my parents, Mr. Hem Brall and Mrs. Prabha Brall, for encouraging me and showing me the difference between right and wrong throughout my childhood.

Hadi Eskandari, while working on both Java and .NET technologies, has developed enterprise-level applications on both platforms. He is a regular contributor to various open source projects. Currently, he is working as a senior software developer at Readify, which is a leading company on .NET technology in Australia.

I'd like to thank my family for their support. This wouldn't have been possible without your encouragement and help.

Roy Cornelissen works as a software architect in the Netherlands. With over 15 years of experience in IT, he has designed and built many enterprise systems for customers using Microsoft, Particular, and Xamarin technologies.

As a lead consultant at Xpirit, Roy is responsible for the vision and strategy for the Enterprise Mobile Development competence. He specializes in designing and building mobile applications for iOS, Android, and Windows Phone, and using Xamarin technology and architectures for distributed systems running in the cloud.

In 2011, Roy cofounded the Dutch Mobile .NET Developers group, and he is an active member of the Xamarin and NServiceBus communities. He has been awarded the Xamarin Insider and NServiceBus Champion titles.

He is a frequent speaker at software development conferences, such as Microsoft TechDays, Xamarin Evolve, NSBCon, and Gartner Catalyst. He writes articles and blogs about his professional and personal interests.

As an avid amateur cook, Roy shoots for the Michelin stars. He loves photography, graphic design, and playing the guitar in his spare time.

I'd like to thank David Boike for the opportunity to review his excellent book, which has taught me some new things as well, and the NServiceBus community for being an awesome resource of knowledge.

Daniel Marbach is an independent contractor working in tracelight GmbH in Lucerne, Switzerland. His experience spans from mobile application development to client and server development, with a strong tendency towards distributed systems. He is a long-time contributor to and community champion of NServiceBus and its ecosystems. Daniel is a frequent speaker, coach, and passionate blog writer. Recently, he cofounded the .NET Usergroup of Central Switzerland and continues his journey of software development with passion.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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` 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` for more details.

At `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.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter or the *Packt Enterprise* Facebook page.

Table of Contents

Preface 1

Chapter 1: Getting on the IBus 5

- **Why use NServiceBus?** 5
- **Getting the code** 7
  - NServiceBus NuGet packages 9
- **Our first example** 10
- **Creating a message assembly** 10
- **Creating a service endpoint** 11
- **Creating a message handler** 12
- **Sending a message from an MVC application** 14
  - Creating the MVC website 14
- **Running the solution** 18
- **Summary** 21

Chapter 2: Messaging Patterns 23

- **Commands versus events** 23
  - Eventual consistency 24
  - Achieving consistency with messaging 26
- **Events** 28
  - Publishing an event 29
  - Subscribing to an event 31
- **Message routing** 33
- **Summary** 35

Chapter 3: Preparing for Failure 37

- **Fault tolerance and transactional processing** 38
- **Error queues and replay** 40
  - Automatic retries 40
  - Replaying errors 42
  - Second-level retries 42

RetryDemo 44
Messages that expire 45
Auditing messages 46
Web service integration and idempotence 46
Summary 50
Chapter 4: Hosting 51
Hosting types 51
NServiceBus-hosted endpoints 52
Self-hosted endpoints 53
Assembly scanning 54
Choosing an endpoint name 54
Dependency injection 55
Message transport 55
Reasons to use a different transport 56
MSMQ 57
RabbitMQ 57
SQL Server 58
Windows Azure 59
Persistence 59
In-memory persistence 60
NHibernate 60
RavenDB 62
Windows Azure 63
Polyglot persistence 63
Message serialization 63
Transactions 65
Purging the queue on startup 65
Installers 65
Startup 66
Send-only endpoints 67
Summary 67
Chapter 5: Advanced Messaging 69
Unobtrusive mode 69
TimeToBeReceived attribute 71
Message versioning 72
Polymorphic dispatch 73
Events as interfaces 74
Specifying the handler order 75
Message actions 76
Stopping a message 76

Deferring a message 77
Forwarding messages 78
Message headers 78
Property encryption 79
Transporting large payloads 80
Exposing web services 82
Summary 84
Chapter 6: Sagas 85
Long-running processes 85
Defining a saga 86
Finding saga data 88
Ending a saga 89
Dealing with time 91
Design guidelines 93
Business logic only 93
Saga lifetime 95
Saga patterns 95
Messages that start sagas 96
Retraining business stakeholders 97
Persistence concerns 98
RavenDB 98
NHibernate 99
Azure 99
Unit testing 100
Testing events as interfaces 103
Scheduling 104
Summary 105
Chapter 7: Advanced Configuration 107
Extending NServiceBus 108
IConfigureThisEndpoint 108
INeedInitialization 109
IWantToRunWhenBusStartsAndStops 110
Dependency injection 111
Unit of work 113
Message mutators 114
The NServiceBus pipeline 115
Building behaviors 117
Ordering behaviors 118
Replacing behaviors 120
The pipeline context 121

Outbox **122**
DTC 101 122
Life without distributed transactions 123
Outbox configuration 125
Session sharing 126
Summary **126**
Chapter 8: The Service Platform **127**
ServiceControl **128**
ServiceInsight **130**
Endpoint Explorer 131
Messages 131
Main view 131
Flow Diagram 131
Saga 134
Sequence Diagram 136
Other tabs 137
ServicePulse **137**
Endpoint activity 138
Failed messages 139
Custom checks 140
Getting notified 141
ServiceMatrix **142**
Summary **146**
Chapter 9: Administration **147**
Service installation **147**
Infrastructure installers 149
Side-by-side installation 150
Profiles **151**
Environmental profiles 151
Feature profiles 152
Customizing profiles 152
Logging profiles 155
Customizing the log level 156
Managing configurations **156**
Monitoring performance **158**
Scalability **159**
Scaling up 159
Scaling out 160
Decommissioning a MSMQ worker 163
Extreme scale 164
Multiple sites **164**

Virtualization **166**
MSMQ message storage 166
Clustering 167
Transport administration **168**
Summary **169**
Chapter 10: Where to Go from Here? **171**
What we've learned **171**
What next? **173**
Index **175**

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, it is much better to lean on the proven experience of experts of this field. NServiceBus is a framework that gives you a proven asynchronous messaging API and much more.

This book will be your guide to NServiceBus. From sending a simple message to publishing events, implementing complex time-dependent business processes, and deploying systems 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 shows you how to start using the framework. You will learn how to download the framework and send your first message with it.

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

Chapter 3, *Preparing for Failure*, covers concepts such as automatic retry to give you the ability to build a system that can deal with failures.

Chapter 4, *Hosting*, shows how to run and configure NServiceBus to run both within its own host process (as a console application or Windows service), and when hosted in a larger application such as a web project.

Chapter 5, *Advanced Messaging*, delves into advanced topics that will allow you to make the most out of the framework's messaging capabilities.

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

Chapter 7, *Advanced Configuration*, explains how NServiceBus extends and modifies itself to fit any situation or need.

Chapter 8, *The Service Platform*, introduces the extra tools that help you to build, debug, and manage a distributed system from development and into production.

Chapter 9, *Administration*, shows you how to deploy, monitor, and scale a successful NServiceBus system in a production environment.

Chapter 10, *Where to Go from Here?*, summarizes what you have learned in this book and lists additional sources of information.

What you need for this book

This book covers NServiceBus 5.0, and the requirements for this book closely mirror the software it covers:

- Microsoft .NET Framework 4.5
- Visual Studio 2013 or later versions

Additionally, the code samples use ASP.NET MVC 5 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 enterprises. It is assumed that you are quite familiar with the .NET Framework. A passing understanding of ASP.NET MVC concepts 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: "When we included the NServiceBus.Host NuGet package, a reference to `NServiceBus.Host.exe` was added to the class project."

A block of code is set as follows:

```
public interface IUserCreatedEvent : IEvent
{
  Guid UserId{ get; set; }
  string Name { get; set; }
  string EmailAddress { 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 Name { get; set; }
  string EmailAddress { 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, for example, in menus or dialog boxes, appear in the text like this: "In the **Solution Explorer**, right-click on the solution file and click on **Properties**."



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 e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

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 e-mailed 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` 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` 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 binaries and using them to build a simple solution to send a message from a **ASP.NET MVC** website to a backend service for processing.

Why use 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 describes 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 whether their order went through or not, and they're not willing to retry for fear of being charged twice, 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 made it multithreaded, which was a challenge. Then we had to replace the server it was running on because it maxed out. We're adding more clients and it's only a matter of time until we max out the new 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 integrates 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 email is sent over and over until the failure no longer occurs.
- I have a long-running process that kicks 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 keeps loading forever.
- We have a batch job that runs every night during off hours, but it's taking so long to run that it's intruding on regular business hours. Plus waiting for the batch job is a headache. It needs to be more real-time.
- We don't want to keep investing in on-premises infrastructure to deal with potential spikes in traffic. We need to figure out how to transition some of our business processes to run in the cloud.

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 to reduce database locking and blocking and make deadlocks easy to deal with in a reliable way. It will ensure that valuable customer order data is not lost in the deep dark depths of a multi-megabyte logfile.

By the end of the book, you'll be able to build systems that can easily scale out as well as up. You'll also 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, but that would be really dangerous and wasteful. It is a far better strategy to take advantage of all of the industry-leading expertise that has been applied in NServiceBus in the last several years, and concentrate on what your business does best. NServiceBus is the easiest and most capable 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.

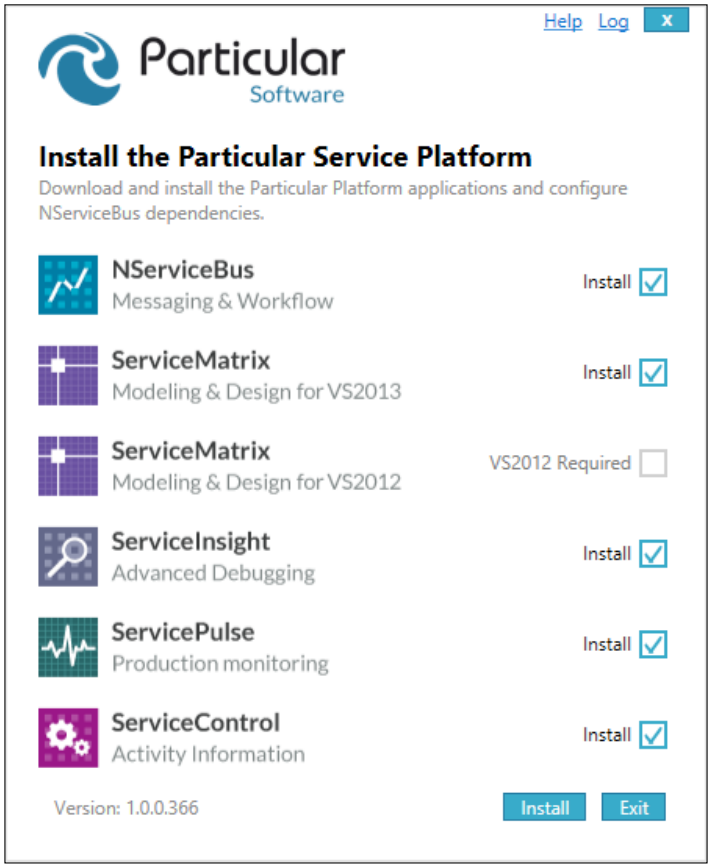
So if you're ready, let's get started creating an NServiceBus solution.

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 look back and explain some of the finer points in more detail.

To get the actual NServiceBus binaries, we will use NuGet exclusively, but before we get started on our first project, we should download and run the installer for the entire Particular Service Platform, which will ensure that your machine is set up properly to run NServiceBus solutions. Additionally, the platform installer will install several other helpful applications that will assist in your NServiceBus development, which we will cover in more detail in *Chapter 8*, *The Service Platform*.

Download the installer from `http://particular.net/downloads` and run it on your machine. The following screenshot depicts the applications installed by the platform installer:



You should select all the options that are available to you. There are two options for **ServiceMatrix** because of differences in how Visual Studio handles add-ins between Visual Studio 2012 and 2013, so you can install whichever matches the version of Visual Studio you use.

In addition to the Service Platform apps, the installer does several things to get your system ready to go:

- **Microsoft Message Queueing** (**MSMQ**): This is installed on your system if it isn't already. MSMQ is the default message transport that provides the durable, transactional messaging that is at the core of NServiceBus (this is only one messaging transport supported by NServiceBus. We will learn about others in *Chapter 4*, *Hosting*).

- **Distributed Transaction Coordinator** (**DTC**): This is configured on your system. It will coordinate transactional data access between resources (such as MSMQ) that support it in order to guarantee that messages are processed once and only once.
- **NServiceBus performance counters**: These are added to help you monitor NServiceBus' performance.

Now that our system is ready to go, we can get started building our first solution by pulling in the NServiceBus NuGet packages.

NServiceBus NuGet packages

Once your computer has been prepared for the first time, you have to include NServiceBus within an application using the NuGet packages.

There are three core NServiceBus NuGet packages:

- `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 used to unit-test NServiceBus endpoints and sagas. We will cover this in more detail in *Chapter 6*, *Sagas*.

If you try installing the NuGet packages first, they will attempt to detect this and direct you to download the entire Particular Service Platform from the website. Without running the installer, it's difficult to verify that everything on your machine is properly prepared, so it's best to download and run the installer before getting started.



In previous versions, there was also a package called `NServiceBus.Interfaces`. It has now been deprecated. Most users should be using unobtrusive mode conventions, which we will cover in depth in *Chapter 5*, *Advanced Messaging*. For simple exercises, wherever `NServiceBus.Interfaces` is used, it should be replaced by the core `NServiceBus` package.

Our first example

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

The following diagram depicts our goal. We will have an ASP.NET MVC web application that will send a command from the `HomeController` process, and then the command will be handled by another process called `UserService`.



Creating a message assembly

The first step while 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.

Follow these steps to create your solution:

1. In Visual Studio, create a new project by creating a new class library. 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` package, which will add the reference to `NServiceBus.Core.dll`:

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

4. Add a new folder to the project called `Commands`.
5. Add a new class file called `CreateNewUserCmd.cs` to the `Commands` folder.
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 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 System.Threading.Tasks;
using NServiceBus;

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

Congratulations! You've created a message. This new message will form the communication contract between the message sender and receiver. This is just a message; now you need to create a service that can receive and process it.

Creating a service endpoint

Now we're going to create a service endpoint—a process that will host code to 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` and `NServiceBus.Core.dll`
   - An `App.config` file
   - A class named `EndpointConfig.cs`
   - Project debug settings to execute `NServiceBus.Host.exe` when we debug

5. In the service project, add a reference to the **UserService.Messages** project you created before.
6. In the `EndpointConfig.cs` class that was generated, replace the text `PLEASE_SELECT_ONE` with `InMemoryPersistence`. You may need to add a `using NServiceBus.Persistence;` declaration to the file if you don't have a tool such as ReSharper to do it for you.

Creating a message handler

Now that we have a service endpoint to host our code, we will create a message handler within the endpoint that will process our message when it arrives:

1. Add a new class called `UserCreator.cs` to the service.
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. In order to keep the examples straightforward, we will skip the details of database access and just demonstrate what will happen by logging a message.

With NServiceBus, you can use any logging framework you like without being dependent upon that framework. NServiceBus internally includes a logging system that logs to both console and file, with an API that looks very much like log4net (previous versions of NServiceBus actually used the log4net framework directly). In *Chapter 7*, *Advanced Configuration*, you will learn how to easily swap these out for the real `log4net` framework, `NLog` framework, or implement an adapter for any logger we like. For now, we are more than content with the built-in logging implementation via the `NServiceBus.Logging` namespace.

Now lets 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 the following statement:

```
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 System.Threading.Tasks;
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. Its okay if you don't understand quite how all of this connects yet. Next, we need to create a way to send the command.



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 e-mailed directly to you.

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 in the form of 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 ensure that 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

Let's follow these directions to get the MVC website set up. We will be using ASP.NET MVC 5 in Visual Studio 2013:

1. Add a new project to your solution. Select the **ASP.NET Web Application** template and name the project `ExampleWeb`. Select the **Empty** template and the **Razor** view engine.
2. On the **New ASP.NET Project** dialog, select the **Empty** template and check the box to add folders and core references for **MVC**.
3. From the NuGet **Package Manager Console**, run this command to install the NServiceBus package:

   ```
   PM> Install-Package NServiceBus -ProjectName ExampleWeb
   ```

4. Add a reference to the **UserService.Messages** project you created before.

Because the MVC project isn't fully controlled by NServiceBus, we have to write a bit of code to get it running.

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

```
using NServiceBus;

namespace ExampleWeb
{
  public static class ServiceBus
  {
    public static IBus Bus { get; private set; }
    private static readonly object padlock = new object();

    public static void Init()
    {
      if (Bus != null)
        return;
      lock (padlock)
      {
        if (Bus != null)
          return;
        var cfg = new BusConfiguration();
        cfg.UseTransport<MsmqTransport>();
        cfg.UsePersistence<InMemoryPersistence>();
        cfg.EndpointName("ExampleWeb");
        cfg.PurgeOnStartup(true);
        cfg.EnableInstallers();

        Bus = NServiceBus.Bus.Create(cfg).Start();
      }
    }
  }
}
```

That was certainly a mouthful! Don't worry about remembering this. The API makes it pretty easy to discover things you need to configure, through IntelliSense. You will learn more about this code in *Chapter 4*, *Hosting*, and I'll explain everything that's going on.

For now, it is sufficient 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.



You may notice the locking pattern used in the previous code to ensure that the Bus instance is initialized only once. This is just one strategy. You could also, for example, utilize a `Lazy<IBus>` instance to the same effect.

Now that we've established the `ServiceBus` class, 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();
  RouteConfig.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 Json(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 very 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.

However, the highlighted method is important because this is where we create an instance of our command class and send it to the bus via the `ServiceBus.Bus` static instance. 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 sections of configuration. 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 of *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 can generate these sections of configuration using the `Add-NServiceBusMessageForwardingInCaseOfFaultConfig` cmdlet and the `Add-NServiceBusUnicastBusConfig` cmdlet.

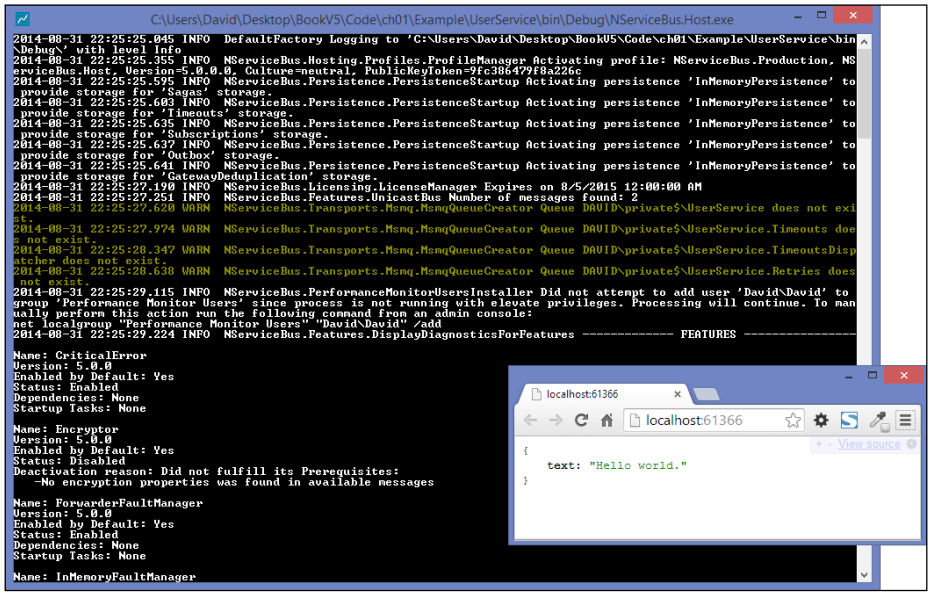
Running the solution

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

1. In **Solution Explorer**, right-click on the solution file and click on **Properties**.
2. From the left side, 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 such 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? You get a result that looks like what is shown in the following screenshot:



When you run the solution, both the MVC website and a console window should appear as shown in the preceding screenshot. 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 that a .NET executable is also an assembly and it can be referenced by another project) and the project was set to run that executable when you debug it.

NServiceBus uses different colors to log messages of different levels of severity. **INFO** messages are logged in white, and **WARN** messages are displayed in yellow. In addition, there can be **DEBUG** messages, also 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 filters 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 single 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, two messages were found. Only one of them is ours; the other is an administrative message, which is used internally by NServiceBus. If it had said that no 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 trial license that allows unrestricted non-production use for a limited time. After that, you need to purchase a commercial license, although Particular may be willing to extend your trial if your situation merits it. For all questions about licensing, go to `http://particular.net/licensing`. Every situation is different, so don't hesitate to contact Particular to find out which licensing structure will work best for you.
- The status of many 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`



If you installed the Service Platform, there could be queues for `error.log` and several queues starting with `particular.servicecontrol` as well. We'll discuss these in depth in *Chapter 8, The Service Platform*.

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 the number in the preceding 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 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. Then that payload was added to an MSMQ message with enough metadata to describe where it came from and where it needs 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 the message from the queue, 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 can 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 in this chapter 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 it still assumes coupling between the sender and the receiver because the sender needs to know where to send the message. 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. Along the way, you'll learn the basics of messaging theory, eventual consistency, and Publish/Subscribe.

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 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.

Eventual consistency

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

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

The challenge lies in building distributed systems due to the following **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. All of these are 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 **Consistency**, **Availability**, and **Partition tolerance**, which is a communication failure between two nodes. The theorem proves that it is impossible to have all three simultaneously.

To better understand CAP, let's consider a gross oversimplification where we need to store and retrieve a single value. We start by storing it on one server. As soon as we store the value, subsequent reads will return the new value, so we have consistency. Because we are dealing with only one node, we cannot have a partition. If our single server is down, we cannot answer requests, thus losing availability. This system is **CP**.

With a second node, we can gain availability by always having at least one node running to respond to requests, but we need to replicate data between nodes so that updates made on one server are present when read from the other.

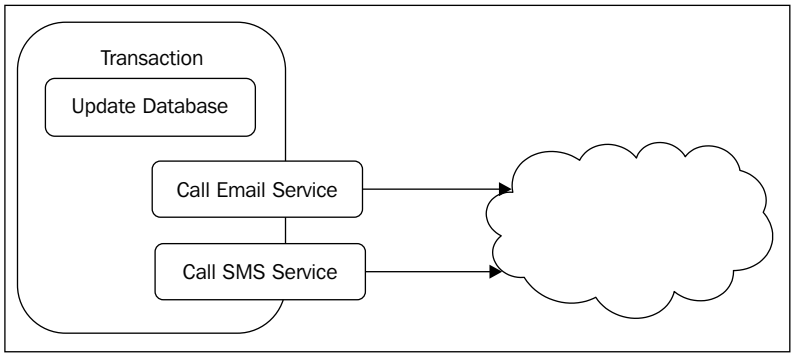
We can decide not to consider a write value as accepted until replication to the second node is complete. Now we have consistency and availability, but if we can't communicate between servers for some reason, we won't be able to commit new values. Our system is not partition tolerant, so this system is **CA**.

Instead, we could use our best efforts to replicate writes, but we cannot enforce it in order to accept the transaction. When a partition is healed, the nodes can simply exchange write logs in order to catch up. This gives us availability and partition tolerance but not consistency because a read during a partition will not necessarily have the most up-to-date value. This system is **AP**.

Because of the fallacies of distributed systems, network partitions are not only possible, but a near-certainty. Since we must have partition tolerance, our only real choices are CP and AP. The need for our systems to be available drives us to select availability and partition tolerance, and embrace a new consistency style called **BASE** (short for **Basically Available, Soft state, Eventual consistency**), as opposed to ACID. We sacrifice consistency for a short time, and 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. This process is shown in the following diagram:



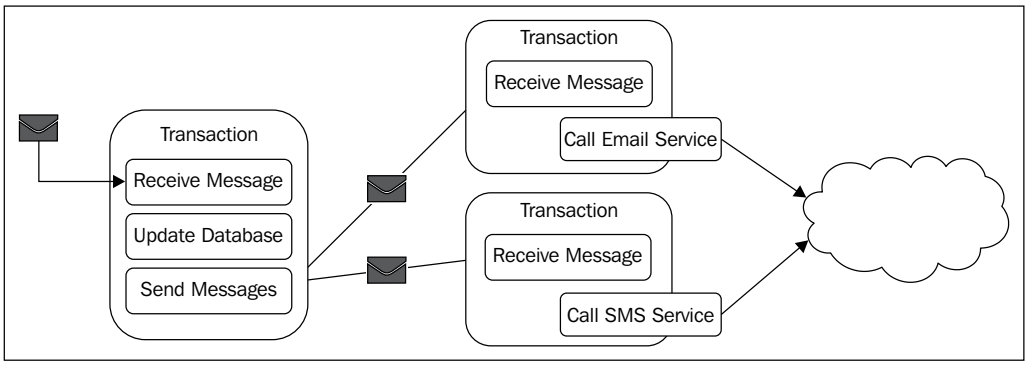
You might have noticed that calling the web services hangs partially outside the transaction. This is because the transaction doesn't ultimately manage the web service call. Once the web service call is made, it will either succeed or fail, regardless of whether the transaction is committed or rolled back.

What happens if the database transaction and the email service call succeed, but 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 users 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 NServiceBus, all of these problems are removed by only taking on as much work as can be successfully accomplished within a transaction. Each NServiceBus 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 database transaction is rolled back, the outgoing messages are not sent, and the original command can be 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. We're assuming that retrying this web service call multiple times would have no ill effects.
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 and the web services have not yet been contacted. This is only a temporary condition. By providing a reliable transport, NServiceBus guarantees 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, which 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*, to see how NServiceBus addresses these guarantees.

Events

If you compare the following definition with that of a command, which was defined earlier in the chapter, you will notice that 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 one 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 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 to rely on the imperative tense for commands and past tense for events instead. 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 so that 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 the messaging theory, let's add the concept of events to our code from *Chapter 1*, *Getting on the IBus*, and see firsthand what they can do for us:



Now that we've gotten our feet wet with NServiceBus, I won't be very 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, inherit 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 UserId { get; set; }
    string EmailAddress { get; set; }
    string Name { get; set; }
  }
}
```

At this point, you may have noticed a few things.

Firstly, 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 duality of 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.

Secondly, you may have noticed that we used an interface to represent the event, rather than a class. 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 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. Open the `UserCreator` class.
2. Add a public property to the class:

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

3. 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 more about 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 really happens 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, 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 when the event arrives.

Now let's add a subscriber to our system and see some Publish/Subscribe in action.

Subscribing to an event

So far, our system 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 library project named `WelcomeEmailService` and delete the `Class1.cs` file.
2. From **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 `EndpointConfig` class to select `InMemoryPersistence`.
5. Create a class named `EmailSender` and implement `IHandleMessages<IUserCreatedEvent>`. Then add logging to simulate sending the welcome email. Your class should look somewhat 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 modify the `UnicastBusConfig` section. The NuGet package inserted it for you, but you have to insert the `MessageEndpointMappings` portion of it. You can copy and paste it directly from the MVC project's `Web.config` file. 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 can always generate the `UnicastBusConfig` section using the `Add-NServiceBus UnicastBusConfig` PowerShell cmdlet.

7. Modify your solution startup project's 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 window, 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, likely at the very end of the output, you should see this:

```
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 the following output:

```
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` endpoint to the `IUserCreatedEvent` message from the `UserService` endpoint.

Now create a user just as 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 is really happening because we are just logging messages to the console, but you can start seeing 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 forums 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 only a bit of so far. By storing the mappings in the configuration allows us to test our system entirely 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 becomes very difficult to manage. As an in-between option, you can also specify routing based on the combination of an assembly and a namespace.

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

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

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 lines of code:

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

For a fully qualified message type, use one of these lines:

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

You can even filter by namespaces, like this:

```
<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 a queue name together with a server name such as `Endpoint="MyQueue@OtherServer"` to address a remote server.

Summary

In this chapter, we started by learning about messaging theory. You 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.

You 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 one or more logical subscribers. You 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. You learned how to configure the message endpoint mappings so that NServiceBus knows where to send our messages.

Now that you have learned how to decouple our business processes through messaging and Publish/Subscribe, you must also learn how to ensure that our messages are processed successfully. In the next chapter, we will see 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 of you 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, but it seems as if evil has won, with the legacy of everything his family had built burning 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 for 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.

Nevertheless, like Alfred, NServiceBus will not give up. If we apply the tools that NServiceBus gives us, we will not lose data even in the face of failure, we can correct the error, and we will make it through.

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 at failure in the face and laugh. We'll discuss error queues, automatic retries, and controlling how those retries occur. We'll also discuss how to deal with messages that may be transient and should not be retried in certain conditions. Lastly, we'll examine the difficulty of web service integrations that do not handle retries cleanly on their own.

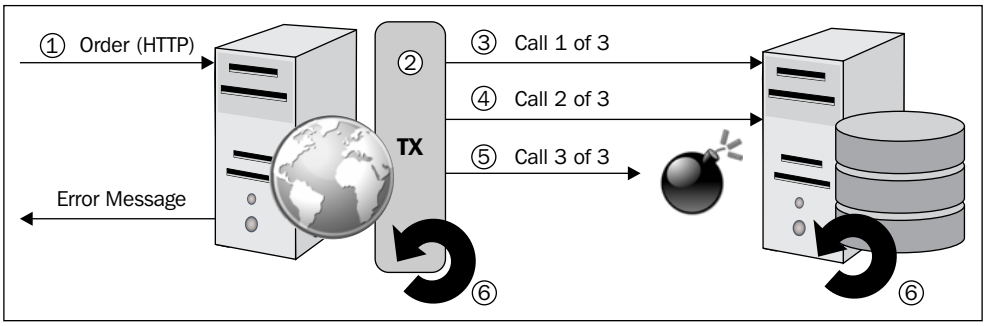
Fault tolerance and transactional processing

In order to understand the fault tolerance we gain from using NServiceBus, let's 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, the following process is happening:

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 **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. This exception causes the transaction to roll back.

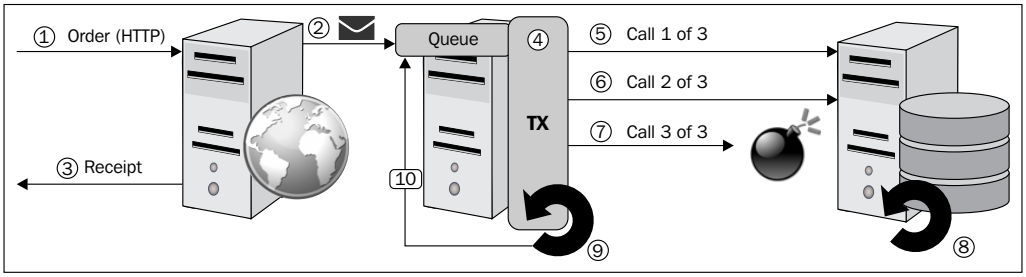
This process is shown in the following diagram:



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. It's even possible that this won't occur during load testing. It will likely occur during heavy load at the worst possible time, for example, 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 through 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, and receives the message within the transaction.
2. **Database Call 1 of 3**: The shopping cart information is retrieved from the database.
3. **Database Call 2 of 3**: An Order record is inserted.
4. **Database Call 3 of 3**: *Deadlock*!
5. The exception causes the database transaction to roll back.
6. The transaction controlling the message also rolls back.
7. 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, the SQL Server exception says **Rerun the transaction**.

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

Error queues and replay

Whenever you talk about automatic retries in a messaging environment, you must invariably consider **poison messages**. A poison message is a message that cannot be immediately resolved by a retry because it will consistently result in an error.

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. At 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 our incoming queue of messages. 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 of this for us. By default, NServiceBus will try to process a message five times, after which it will move the message to 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:

```
<MessageForwardingInCaseOfFaultConfigErrorQueue="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:

```
<MessageForwardingInCaseOfFaultConfigErrorQueue="error@SERVER" />
```

As mentioned previously, five failed attempts form 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 `Add-NServiceBusTransportConfig` PowerShell cmdlet.

Keep two things in mind:

- Depending upon how you read it, `MaxRetries` can be a somewhat confusing name. What it really means is the total number of tries, so a value of 5 will result in the initial attempt plus 4 retries. This has the odd side effect that `MaxRetries="0"` is the same as `MaxRetries="1"`. In both instances, the message would be attempted once.
- 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 configuration.

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 some sort of intervention, which can be human or automated based on your own business rules. 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, detect the divide-by-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 allows us to do this with either the ServiceInsight or ServicePulse tools, which will be covered in depth in *Chapter 8*, *The Service Platform*.

Second-level retries

Automatically retrying error messages and sending repeated errors to an error queue is a pretty good strategy to manage both transient errors, such as deadlocks, and poison messages, such as an unrecoverable exception. However, as it turns out, there is a gray area in between, which is best referred to as **semi-transient** errors. These include incidents such as 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 hordes 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** (**SLR**) that will add additional sets of retries after a wait. By default, the SLR will add three additional retry sessions, with an additional wait of 10 seconds each time.



By contrast, the original set of retries is commonly referred to as **First-Level Retries** (**FLR**).

Let's track a message's full path to complete failure, assuming default settings:

- 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 by 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 that can never be successfully processed.

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 out 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 results in an entire minute of waiting before you see the message 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.



Due to the asynchronous nature of messaging, we should be careful never to assume that messages in a distributed system will arrive in any particular order. However, it is still good to note that the concept of retries exacerbates this problem. If Message A and then Message B are sent in order, and Message B succeeds immediately but Message A has to wait in an error queue for awhile, then they will most certainly be processed out of order.

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"/>
...
<SecondLevelRetriesConfig Enabled="true"
                          TimeIncrease="00:00:10"
                          NumberOfRetries="3" />
```

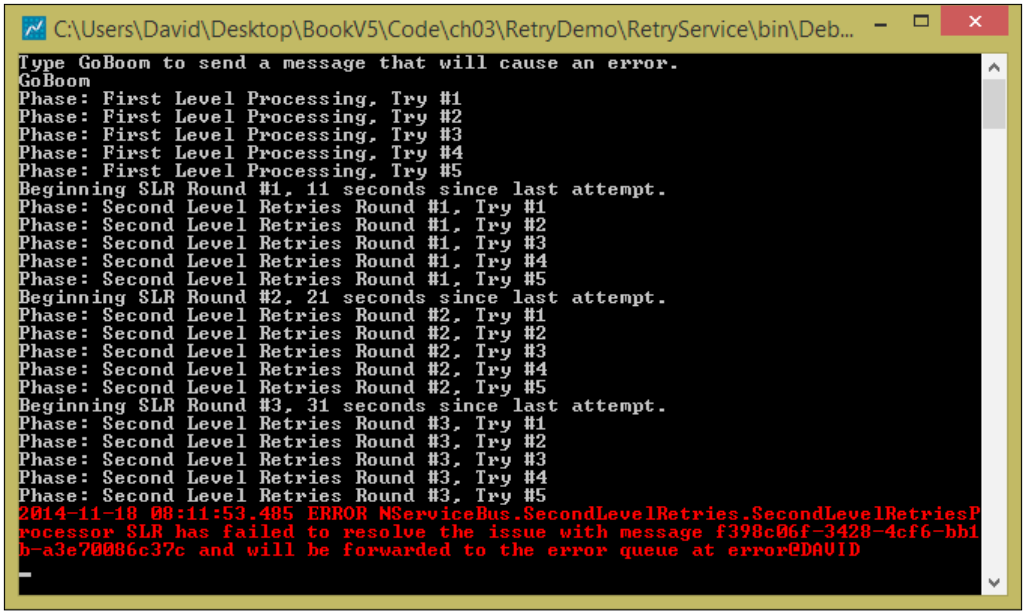


You could also generate the `SecondLevelRetriesConfig` section using the `Add-NServiceBus SecondLevelRetriesConfig` PowerShell cmdlet.

Keep in mind that you may want to disable second-level retries, like first-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-level and second-level retries. Just type `GoBoom` in the console window when prompted and a message will be sent. You can watch the retries as they happen, as shown in the following screenshot:



Play around with the settings in the service project's `App.config` file to see how they affect the output.

Messages that expire

Messages that lose their business value after a specific amount of time are an important consideration with respect to potential failures.

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 cause messages to 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 is simply deleted by the queuing system. NServiceBus isn't actually involved in the deletion at all, it simply tells the queuing system how long to allow the message to live.

If a message fails, however, and arrives at an error queue, NServiceBus will not include the expiration date in order to give you a chance to debug the problem. It would be very confusing to try to find an error message that had disappeared into thin air!

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 the 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. However, this means we are throwing away data, so we need to be very careful when applying this strategy. It should not simply be used as a reaction to low disk space!

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 you run out of available storage, which you would really like to avoid. The exact time limit you use is dependent upon the volume of messages in your system and how much storage your queuing system has available.

You don't even have to design your own tool to monitor these audit messages; the Particular Service Platform has that job covered for you. NServiceBus includes the auditing configuration in new endpoints by default so that ServiceControl, ServiceInsight, and ServicePulse can keep tabs on your system. We will cover these tools in detail in *Chapter 8*, *The Service Platform*.

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 would 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 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 (see the **Fallacies of Distributed Computing** in the previous chapter.):

- A DNS or network failure may not let us contact 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 takes 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 succeeds 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 work takes the form of recording each step we perform on durable storage in real time, and then query that storage to see which steps have been attempted.

In our example of credit card processing, the happy path 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.

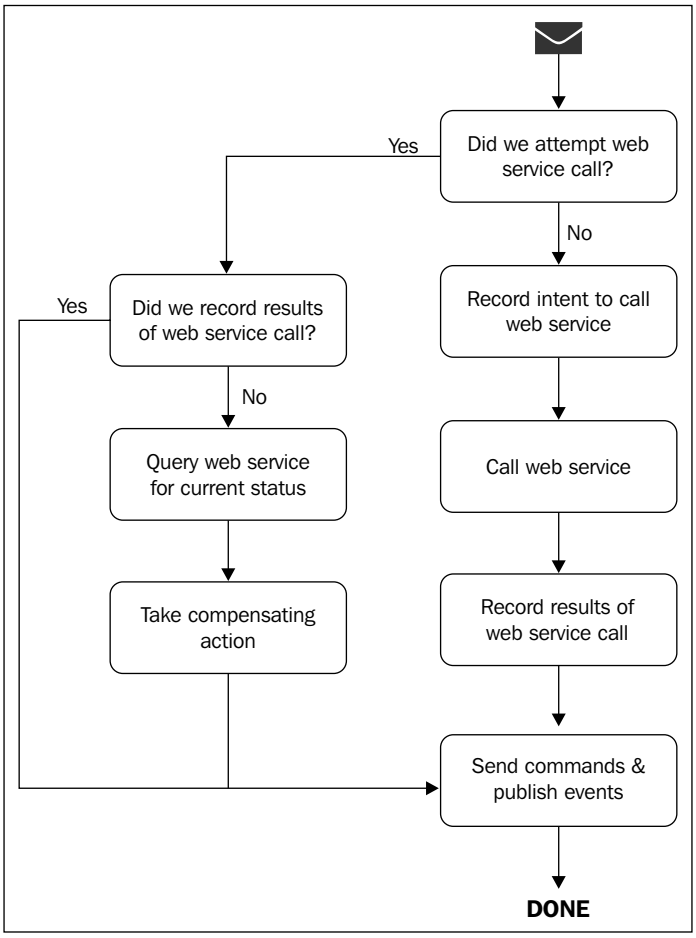
Now, if the message is retried, we can inspect the durable storage and decide what step to jump to and whether any compensating actions need to be taken first.

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 whether we need to retry the authorization or just get the results of the already completed authorization.

If we see that we have already made the web service call and received the 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.

It's important to be able to handle the case where our durable storage throws an exception, rendering us unable to make our state persist. This is why it's so important to record the intent to do something before attempting it—so that we know the difference between never having done something and attempting it but not necessarily knowing the results.

The process we have just discussed is admittedly a bit abstract, and can be visualized much more easily with the help of the following diagram:



The choice of using the durable storage strategy 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` object to suppress the transaction, like this:

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

using (var ts = new TransactionScope(TransactionScopeOption.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. You learned how NServiceBus promises fault tolerance within every message handler so that messages are never dropped or forgotten, but instead retried and then held in an error queue if they cannot be successfully processed. Once we fix the error, or take some other administrative action, we can replay those messages.

In order to avoid flooding our system with useless messages during a failure, you learned how to cause messages that lose their business value after a specific amount of time to expire.

Finally, you learned how to build auditing in 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 NServiceBus hosting works and how we can host NServiceBus in our own processes.

4
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` package. The host process gives us many shortcuts that simplify setting up a messaging endpoint, and as we will see in *Chapter 9*, *Administration*, it even allows us to easily install an endpoint as a Windows service.

The time has come to lift the veil and learn how NServiceBus is hosted, both within the NServiceBus Host and within other processes. Chief among these are web applications, although it is possible to host NServiceBus within **Windows Presentation Foundation** (**WPF**) and Windows Forms apps as well. We must be able to configure and host our messaging infrastructure within all of these environments.

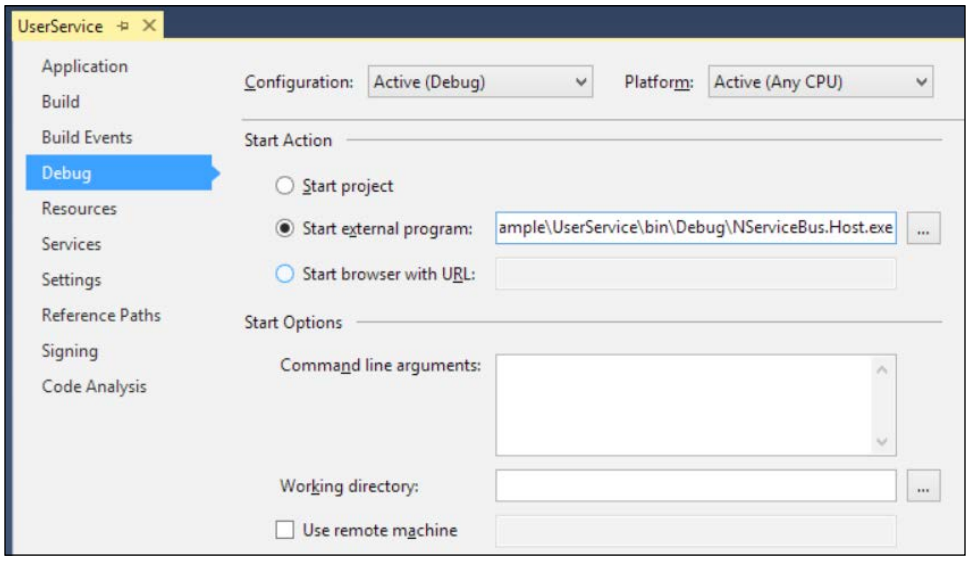
In this chapter, you will learn how to use the NServiceBus configuration methods to host a message bus within any application we want, or to customize an application hosted by the `NServiceBus.Host` package.

Hosting types

Hosting NServiceBus comes down to configuring and then starting an instance of a bus. There are two ways to do this, either as an NServiceBus-hosted endpoint, or as a self-hosted endpoint. In both cases, the focus is on setting up your desired settings via a `BusConfiguration` object.

NServiceBus-hosted endpoints

An NServiceBus-hosted endpoint is a process that is run by executing the `NServiceBus.Host.exe` process, which is a part of the `NServiceBus.Host` NuGet package. We already know that a .NET executable is just a DLL that can be directly executed, and can be referenced just like a normal DLL. The host's NuGet package automates the process of creating a reference to the executable, and then sets the **Debug** tab's **Start Action** to run the host executable. You can see this for yourself by opening a hosted endpoint's properties page and inspecting the settings on the **Debug** tab.



When the host process starts up, it scans all the assemblies in the same directory, looking for a class that implements `IConfigureThisEndpoint`. As we have already seen in our previous examples, the NuGet package creates this class for us, and it looks something like this:

```
public class EndpointConfig : IConfigureThisEndpoint, AsA_Server
{
  public void Customize(BusConfiguration configuration)
  {
    configuration.UsePersistence<InMemoryPersistence>();
  }
}
```

It is primarily this class that gives us access to the `BusConfiguration` instance to set up how the endpoint will operate. For now, let's ignore the `AsA_Server` marker interface. We will discuss it in more detail in *Chapter 7*, *Advanced Configuration*.

NServiceBus-hosted endpoints are easily run as console applications while you are debugging, and then they can be installed as Windows services with simple command-line parameters. We will learn more about this in *Chapter 9*, *Administration*.

Self-hosted endpoints

By contrast, a self-hosted endpoint is an existing process that configures and starts a bus within it. This is commonly a web application, but it could also be a WPF or Windows Forms application. It could even be a console application, although the wisdom of doing so would be debatable, given the large number of features the `NServiceBus.Host` package offers.

To self-host a bus, we just need to create a new instance of `BusConfiguration` and set the necessary settings for our environment, just like the configuration block from our previous samples:

```
var cfg = new BusConfiguration();
cfg.UsePersistence<InMemoryPersistence>();
IBus Bus = NServiceBus.Bus.Create(cfg).Start();
```

When we're done, we start the bus with one simple line of code and then make the `IBus` instance available to our application, commonly as a static property.

This is not a lot of configuration considering everything that's happening to configure a bus, especially compared to previous versions of NServiceBus that sported long blocks of fluently chained methods starting with `Configure.With()` that you will likely see all over the Internet (unfortunately) for years to come. These fluent configuration methods were dependent upon the order of execution and caused quite a bit of grief for everyone involved.

In NServiceBus 5.0, the team has gone out of their way to make the configuration story remarkably more straightforward, providing sensible defaults for most settings, and changing the process so that the order in which items are configured no longer matters.

This does not mean that any power of all of those configuration options has gone away, it's just not always required. Now let's start exploring them, starting with assembly scanning.

Assembly scanning

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 the assemblies in the application to find those types. By default, NServiceBus will scan 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.

Many improvements have been made to the assembly scanning process over the last few versions of NServiceBus in order to automatically ignore assemblies that cannot be scanned. For example, COM DLLs are skipped, as are 64-bit assemblies if running with a 32-bit host process. As a result, this process generally works very well and if there is an issue, a helpful exception message will let you know how to correct the problem.

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` package, the endpoint name is automatically determined by locating the `EndpointConfig` class (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 `NServiceBus.Bus.Create()`.

We can, however, take control and explicitly set the endpoint name if we really want to:

```
cfg.EndpointName("MyEndpoint");
```

Keep in mind that the string need not be a constant. 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.



If a web application is hosted with OWIN, automatic endpoint name assignment will not work, and you must explicitly set the endpoint name as described in the preceding code. Because of this, NServiceBus 6.0 will require setting the endpoint name explicitly. It would be wise to start this practice now in order to make future upgrades more straightforward.

Dependency injection

Dependency injection is a pattern that allows us to avoid hardcoded dependencies and define them at runtime. NServiceBus uses dependency injection to manage a host of dependencies, but the most visible dependency is the `IBus` dependency, which 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 (which is embedded within `NServiceBus.Core.dll`) for its own needs. In most cases, this is just fine, but if you already use a DI container and would like to integrate NuGet package with it directly, you can do so using one of the following NuGet packages that act as an adapter between NServiceBus and your chosen container:

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

Each package will contain a class that inherits `NServiceBus.ContainerDefinition`, which you can use as the generic parameter in the `UseContainer` method, such as this example for the `NServiceBus.Ninject` package:

```
cfg.UseContainer<NServiceBus.Ninject>();
```

We will cover more topics related to dependency injection in *Chapter 7*, *Advanced Configuration*.

Message transport

The most important configuration setting for NServiceBus is the selection of the message transport. **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. For example, here is how you would explicitly select MSMQ, although as the default transport, doing so isn't required:

```
cfg.UseTransport<MsmqTransport>();
```

The `MsmqTransport` type parameter is a class that inherits from `NServiceBus.TransportDefinition`. In addition to being the default transport, MSMQ is also built into the NServiceBus core.

Using any of the other transports also requires an additional NuGet package. The full details about the use of each alternative transport are beyond the scope for this book, but we will cover them briefly.



It's technically possible, though not exactly easy, to implement your own message transport and configure NServiceBus to use it. While it's a fun exercise, it makes apparent how much work the Particular team puts into ensuring reliability with each transport they support. Their team lives on and breathes messaging, and it's probably best to leave this one to the experts.

Reasons to use a different transport

MSMQ has been a 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 an approval from a company's IT department for a system design that requires MSMQ and distributed transactions if such support wasn't already available before. Yet in such 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. They can 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.

MSMQ

Before we dive into alternatives, we should understand MSMQ a little better. MSMQ is a true bus-style messaging system, where each server hosts its own MSMQ service and communicates with other servers in a decentralized fashion. As a decentralized solution, MSMQ provides store-and-forward so that once a message is sent, you can be sure that it will (eventually) arrive at its intended destination. One big advantage of a bus-based queuing infrastructure is that even if individual servers go down from time to time, messages continue to flow through other nodes in the system, with store-and-forward ensuring that no messages are lost.

MSMQ contains no native publish/subscribe mechanism, so NServiceBus must use **storage-based publishing**. This means that when you publish a message, NServiceBus looks for subscribers in some sort of persistent data store. This boils down to a lookup in a database, after which an independent message is sent to each registered subscriber. We will discuss storage and persistence concerns in the next section.

In order to subscribe, NServiceBus uses **message-based subscriptions**, meaning that the subscriber endpoint will send a specialized subscription request message to the publisher endpoint. Of course, all of this happens automatically.

MSMQ also supports the **Distributed Transaction Coordinator** (**DTC**), so by default, all operations within your message handlers operate within a transaction shared by your database and MSMQ, giving you a lot of safety and guaranteeing once-and-only-once message delivery.

You can find a repository of sample NServiceBus projects that use the MSMQ transport, at `https://github.com/Particular/NServiceBus.Msmq.Samples`.

RabbitMQ

RabbitMQ is an open source queuing system that implements the **Advanced Message Queuing Protocol** (**AMQP**). Rabbit is a broker-style transport (the messaging architecture is centralized) and is a great solution for cloud platforms such as Amazon EC2. However, it contains no support for the DTC.

Normally, a lack of DTC support would mean that you, as a developer, could not count on once-and-only-once delivery of messages, which we are accustomed to with MSMQ, but would have to settle for at-least-once delivery instead. This looser delivery guarantee means we need to be on the lookout for messages being processed more than once, which means our handlers would have to be completely idempotent or take compensating actions to account for duplicate messages.

NServiceBus 5.0 introduces the Outbox feature, a way of accounting for these potential duplicate messages so that you, as a developer, don't have to worry about them. The Outbox feature will be covered in more detail in *Chapter 7*, *Advanced Configuration*.

RabbitMQ also contains native Publish/Subscribe support. Instead of sending copies of a message to each subscriber, messages are published to a topic, which fans out to any queues that are subscribed. Also, for a centralized queuing system, you will need to provide a connection string to the RabbitMQ server or clustered server. Clustered servers are recommended to ensure high availability.

RabbitMQ is available via the `NServiceBus.RabbitMQ` NuGet package. You can find a repository of sample NServiceBus projects that use the RabbitMQ transport, at `https://github.com/Particular/NServiceBus.RabbitMQ.Samples`.

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 by 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; a simple database transaction can be used instead. 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 among 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 so on.

SQL Server is obviously not inherently a queuing technology. It works by polling against a queue table that it creates in your database. In order not to ping your SQL Server instance to death, it implements a back-off strategy between messages, waiting for successively longer times between attempts when no messages are available, up to a maximum of one second. When messages are coming in, the back-off wait is minimized so that messages can be received and processed as quickly as possible.

Like MSMQ, the SQL Server transport uses message-based subscriptions and storage-driven publishing. The storage, of course, is best managed in the same SQL Server database, which we will discuss shortly.

The SQL Server transport is available via the `NServiceBus.SqlServer` NuGet package. You can find a repository of sample NServiceBus projects that use the SQL Server transport, at `https://github.com/Particular/NServiceBus.SqlServer.Samples`.

Windows Azure

There are actually two Windows Azure transports: one that supports Windows Azure Queues and another that supports the Windows Azure Service Bus. In general, Azure Queues are simpler, and the Azure Service Bus provides more advanced features at the expense of additional complexity.

Both Azure transports enable NServiceBus to operate either entirely within the cloud or in a hybrid cloud/on-premise scenario. Additionally, both are alike in that they do not support the Distributed Transaction Coordinator. However, Azure Queues are essentially a simple REST-based Get/Put/Peek API, whereas the Azure Service Bus supports native publish/subscribe via topics.

This comparison is very simplistic; each transport has many more of its own idiosyncrasies. For a detailed comparison, you should read the article, *Azure Queues and Service Bus Queues – Compared and Contrasted*, at `http://msdn.microsoft.com/en-us/library/azure/hh767287.aspx`.

The Windows Azure Queues transport is available via the `NServiceBus.Azure.Transports.WindowsAzureStorageQueues` NuGet package, and the Windows Azure Service Bus transport is available via the `NServiceBus.Azure.Transports.WindowsAzureServiceBus` NuGet package. You can find a repository of sample NServiceBus projects that use the Azure transports at `https://github.com/Particular/NServiceBus.Azure.Samples`.

Persistence

Like most applications, every now and then, NServiceBus has the need to store some data, but unlike most applications, NServiceBus does this in a completely configurable way. You can choose to use one of the official persistence libraries according to your organization's needs, or you can choose to write your own.

NServiceBus uses persistence to store subscriptions for transports that use storage-based publishing, as we have already seen. It also stores saga and timeout data, which we will learn about in *Chapter 6*, *Sagas*. Persistence is also used by the Gateway, which we will learn about in *Chapter 9*, *Administration*, and by the Outbox feature, which we will learn about in *Chapter 7*, *Advanced Configuration*.

A persistence mechanism is defined by a class inheriting `NServiceBus.Persistence.PersistenceDefinition`, which is inserted into the `cfg.UsePersistence<T>()` method, which we have already seen. The supported mechanisms available from Particular are `InMemoryPersistence`, `NHibernatePersistence`, `RavenDBPersistence`, and `AzurePersistence`.

In-memory persistence

The examples we have seen so far have all used in-memory persistence:

```
cfg.UsePersistence<InMemoryPersistence>();
```

This is the only persistence mechanism that is built into the NServiceBus core, and as such, it makes up a great choice for quick development, easy demos, unit tests, and (this will be a shocker) sample code in an introductory book, but not much else.

In a production system, you need to use something else, but you do have to make a decision and configure something in most cases, which is what the samples I have shown do. Otherwise, you will get an exception when your endpoint starts up because persistence has not been configured.

In-memory persistence can be very useful during development, especially during fast iterations. With in-memory persistence, it is no big deal to experiment, because nothing is left behind in storage when you stop the endpoints. When using real persistence, outdated stored subscriptions can cause unintended side effects if you're not expecting them.

On the other hand, when using in-memory persistence, it might appear as if an endpoint is not subscribed because subscription request messages get sent between endpoints in an unexpected order. Additionally, if you restart one endpoint, it will not receive all the subscription requests it should have had, and you will find your event handlers are not being run. You can combat this by adjusting the start order of debugging in the solution's property pages, but just knowing about this behavior makes it pretty unlikely that you'll be caught off-guard by its effects.

If you wish to use in-memory persistence during development, you will need a method to easily switch to something more stable for test and production environments. We will discuss strategies for this in *Chapter 9*, *Administration*.

NHibernate

With NHibernate persistence, you can allow NServiceBus to persist its data in either Microsoft SQL Server or Oracle. One might think that using NHibernate would, in theory, allow any relational database to be used, and while you might be able to get it to work, SQL Server and Oracle are the only systems tested and supported by Particular. Surely, any attempt to use Microsoft Access would most certainly end in tears.

After including the `NServiceBus.NHibernate` NuGet package, you can configure its use like this:

```
cfg.UsePersistence<NHibernatePersistence>();
```

Of course, this isn't enough. We will need to know what database to connect to. For that, we use the familiar construct of a connection string:

```
<connectionStrings>
    <add name="NServiceBus/Persistence"
         connectionString="CONN_STR_HERE" />
</connectionStrings>
```

The connection string needs to point to a database with permissions to modify the database structure, in order to create the tables NServiceBus needs to store its data.

The connection string can be specified in the config file, as shown in the preceding code, or if you like, you can also configure in this way:

```
cfg.UsePersistence<NHibernatePersistence>()
    .ConnectionString("CONN_STR_HERE");
```

There are also additional extension methods that can be used after configuring NHibernate persistence to customize things, for instance, to disable schema updates, or to use specialized NHibernate configuration settings, which we won't get into here. One very interesting extension, however, is the enabling of caching for subscription lookups:

```
.EnableCachingForSubscriptionStorage(TimeSpan.FromMinutes(10))
```

In transports that use storage-based publishing, every published message will require a trip to the database to look for the destination endpoints. This is a safe-by-default approach; so that brand new subscribers don't miss out on a message published due to an out-of-date cache. If, however, you have an endpoint that publishes frequently, and you know that your list of subscribers will not be changing, or that new subscribers can tolerate missing messages for a time, then you can ask NServiceBus to cache this list of subscribers for a certain period of time for an easy performance boost.

More documentation for NHibernate persistence is available on the Particular website, and you can find a repository of sample NServiceBus projects that use NHibernate persistence at `https://github.com/Particular/NServiceBus.NHibernate.Samples`.

RavenDB

In NServiceBus 3.0, the default persistence switched from NHibernate to RavenDB, and the RavenDB server was installed as a part of the NServiceBus installation process. In some ways, this was great because a document database was well-equipped to handle some of the unstructured data that NServiceBus needed to store. In addition, it made getting started quickly very easy because NServiceBus could rely on a convention to create the necessary database locally without user intervention—no connection string was needed.

However, this came with trade-offs. The RavenDB binaries were required to be ILMerged (merging the intermediate machine language of an external assembly by a specialized tool after compilation) with the NServiceBus assembly as internal namespaces, which meant you could not specify your own `DocumentStore` object for NServiceBus to use. If you wanted to use RavenDB for your application, there was effectively a wall between NServiceBus and your code that could not be torn down. Additionally, any new versions of the RavenDB assemblies would require a new version of NServiceBus because NServiceBus was hardcoded to the ILMerged versions.

So in NServiceBus 4.0, a new method called Costura was used to embed RavenDB as assembly resources so that a version of the RavenDB client code could be selected at runtime. This solved the ILMerge problems, but instead created incompatibilities when NServiceBus had to select and use different versions of the RavenDB client.

NServiceBus 5.0 takes a new direction, moving the RavenDB code to a separate NuGet package, which can version itself in lockstep with the RavenDB client code.

After including the `NServiceBus.RavenDB` NuGet package, you can configure its use like this:

```
cfg.UsePersistence<RavenDBPersistence>();
```

Similar to NHibernate persistence, you can use a connection string to point to the RavenDB server using the connection string name, `NServiceBus/Persistence/RavenDB`. Alternatively, you can build a RavenDB `IDocumentStore` object yourself and add it in the code as follows:

```
cfg.UsePersistence<RavenDBPersistence>()
    .SetDefaultDocumentStore(documentStore);
```

There are additional extension methods that allow you to specify separate document stores for the various features that NServiceBus stores data for.

More documentation on RavenDB persistence is available on the Particular Software website.

Windows Azure

Clearly, Azure persistence is a great choice if you are using one of the Azure message transports. To use it, include the `NServiceBus.Azure` Nuget package and then configure with this line of code:

```
cfg.UsePersistence<AzureStoragePersistence>();
```

Azure storage requires quite a bit more configuration, depending on whether you are hosting on-premise or in the cloud. For details (about both), refer to the documentation on the Particular Software website.

Polyglot persistence

I have to admit that I really like the word **polyglot** for some reason, which means *able to use several languages*. If you want to use one type of persistence for one feature, and another type of persistence for another feature for some reason, you can actually do so.

For each type of persistence, the `UsePersistence<T>()` method supports a method that will enable you to specify what features to use it for. This is best illustrated by a simple, if somewhat contrived, example:

```
cfg.UsePersistence<InMemoryPersistence>()
    .For(Storage.Subscriptions);

cfg.UsePersistence<NHibernatePersistence>()
    .For(Storage.Timeouts, Storage.Sagas);

cfg.UsePersistence<RavenDBPersistence>()
    .For(Storage.Outbox);
```

Message serialization

As we have already seen, our NServiceBus messages are plain old C# classes or interfaces, but these must be serialized in some way so that they can 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:

```
cfg.UseSerialization<XmlSerializer>();
cfg.UseSerialization<JsonSerializer>();
cfg.UseSerialization<BsonSerializer>();
cfg.UseSerialization<BinarySerializer>();
```

Like the persistence configuration options, the serializers expose various settings through extension methods, although none are important enough to mention here.

While XML serialization is the default serialization in NServiceBus 5.0 (and in all previous versions), this will not always be the case. The plan for future versions of NServiceBus is to switch to JSON serialization as default, as JSON is a lot more efficient than XML in terms of bytes sent over the wire, but this switch will require NServiceBus to be able to simultaneously consume messages formatted with multiple serializers to maintain forward compatibility.

Knowing that a switch to JSON as default is in process, and if you are, as the title of this book suggests, "learning NServiceBus" to begin development of a new system, it would be a good idea to select JSON as your serializer now.



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 serializer, and `KnownTypesAttribute`, which is used to represent polymorphic data structures, is not supported. In addition, certain types that you might expect to be serializable are not supported.

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 such as `IList`, `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.

Transactions

Each message transport has recommended transaction settings that it will select by default, but NServiceBus gives you the ability to override these if you choose to do so. For example, the following command can be used to disable distributed transactions:

```
cfg.Transactions().DisableDistributedTransactions();
```

Several other methods, which are easily discoverable via IntelliSense, will also allow you to enable distributed transactions (as they are not turned on by default for every transport), enable and disable transactions in general, set the transaction isolation level and timeout settings, and control whether or not message handlers get wrapped with a `TransactionScope`.

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 the cache is now invalid."

If a web application or smart client is offline for a period of time, as is the case when **Internet Information Services** (**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:

```
cfg.PurgeOnStartup(true);
```

While this option is quite common when self-hosting a web application, it is generally considered to be a very bad idea on a hosted endpoint, although the same configuration will work in both places.

Installers

Installers are tasks that can run to set up an endpoint for the first time. We will learn how to make our own installers in *Chapter 9*, *Administration*, but the most common installers we usually think about are the tasks that create the endpoint's queues if they are missing.

Whether in a hosted endpoint or in self-hosting, installers will automatically run by default if there is a debugger attached. This is totally a convenience thing; it means that when we press *F5* in Visual Studio, we don't have to worry about ensuring that queues have been created.

If a debugger is not attached, it's a different story. A hosted endpoint will run its installers when it is installed as a Windows Service, and not every time it starts up. This is good practice for hosted endpoints and generally works quite well.

A self-hosted endpoint running in the wild will not run its installers at all unless you ask it to. Your options are to manage queue creation through some type of deployment system (Octopus Deploy, Puppet, or similar systems) or to simply ask NServiceBus to do it for you:

```
cfg.EnableInstallers();
```

More deployment considerations will be discussed in *Chapter 9*, *Administration*.

Startup

In a hosted endpoint, we don't have to do anything to start the bus; that is handled for us. We only set options on the `BusConfiguration` object. When self-hosting, however, we have to explicitly start the bus ourselves. Back in our example code, we had this line:

```
IBus Bus = NServiceBus.Bus.Create(cfg).Start();
```

It really can't get any simpler than that. The `Create()` method returns an `IStartableBus` instance, which we can then start to return the `IBus` instance to share it with the rest of our application.

The `IStartableBus` interface implements `IDisposable`, which means we can use it in a `using` block if we only need the bus for a short time for some reason. For most applications, however, the bus's lifetime will mirror that of the host app. The `IBus` instance is fairly expensive to create (as the length of this chapter can attest), so you won't want to create a new instance every time some action occurs. Instead, create it when the application starts up and dispose it when the application shuts down.

Send-only endpoints

At some point, we may create an application that only sends messages to backend services. This is common 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 that 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 slightly different method to start the bus:

```
ISendOnlyBus SendOnlyBus = NServiceBus.Bus.CreateSendOnly(cfg);
```

A send-only bus contains a subset of the methods found in a normal bus, omitting some of the methods from `IBus` that don't make any sense when you can't receive a message.

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 a full endpoint was to demonstrate the simple addition of message handlers in *Chapter 2*, *Messaging Patterns*, when we began processing events in the web application.

Summary

In this chapter, we dissected the process of hosting and configuring an NServiceBus endpoint both using the NServiceBus Host and in our own application. We now know how to select a message transport, persistence strategy, message serializer, and quite a few other options as well.

At this point, you may be thinking that there are a lot of settings that will need to be repeated for every single endpoint you create, and there could be a lot of endpoints. Well, fear not. At the very least, you could create a factory class in a shared assembly to provide each endpoint with a preconfigured `BusConfiguration` instance. In *Chapter 7*, *Advanced Configuration*, you will learn an even better way to manage this shared configuration, and in *Chapter 9*, *Administration*, you will learn how to manage the differences in it as we move from our development environment to production, and everywhere in between.

Now that we have a deeper understanding of how NServiceBus is hosted, in the next chapter, we will explore several advanced messaging techniques.

5
Advanced Messaging

In the previous chapter, we learned how NServiceBus works from the ground up, providing the foundation needed to ultimately host our message handlers. Now we will learn about advanced messaging techniques that will allow us to take full advantage of those message handlers.

Unobtrusive mode

We have seen how using events can help us to decouple business processes from each other. By keeping associations between loosely coupled modules, we can prevent our system from becoming an interconnected ball of mud that is difficult to maintain in the long term.

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

As it turns out, it can be. When you create your assembly of messages in this way, you compile it against a specific version of `NServiceBus.Core.dll`. Then another service consumes that assembly. If you now 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 marker interfaces. Starting with the `BusConfiguration` instance, `cfg`, which we used in the previous chapter, we can configure our unobtrusive mode conventions like this:

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

Each method accepts a `Func<Type, bool>` parameter 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, in the examples in this book, we 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. Bare classes outside a namespace are fairly uncommon and would be difficult to reason about, so let's just assume we won't do this. You would also want to be sure to check for a null namespace so that your conventions don't throw a null reference attempting to call `StartsWith` or `EndsWith` on the namespace name.
- If your company uses a single top-level namespace on all classes, such as the company name, all message types must have a namespace starting with that company name. This ensures that your convention won't accidentally apply to classes in any .NET Framework or third-party libraries.
- If you do not have a single top-level namespace (such as `Packt.` in this chapter's sample code), then at least specify that any namespace that starts with `NServiceBus` or `System` cannot be a message. 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`.

As a short example, this snippet defines the convention for commands:

```
cfg.Conventions()
  .DefiningCommandsAs(t => t.Namespace != null
    && t.Namespace.StartsWith("Packt.")
    && t.Namespace.EndsWith("Commands"))
```

Check out the code samples for this chapter to see complete examples of unobtrusive conventions, including commands, events, internal messages, and more. Each project defines its conventions in a class called `MessageConventions.cs`, complete with comments to describe why the conventions were chosen. No, seriously! Go and get the downloadable code for this chapter right now. It may be one of the most valuable things in this entire book. Don't worry; I'll wait. Are you back? Good!



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*.

TimeToBeReceived attribute

There is an additional convention we must define—the `TimeToBeReceived` attribute, which was covered in *Chapter 3*, *Preparing for Failure*. After all, what good is unobtrusive mode if we would still need to reference `NServiceBus.Core.dll` to use this attribute?

We can create a simple convention for `TimeToBeReceived`, like this:

```
.DefiningTimeToBeReceivedAs(t => t.Name.EndsWith("Expires")
  ? TimeSpan.FromSeconds(30)
  : TimeSpan.MaxValue)
```

While this will work if our needs are simple, this convention has some room for 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—using reflection to find any attribute called `TimeToBeReceived`, which we would define right in our message assembly.

There are a few more concepts similar to `TimeToBeReceived` that require their own conventions, which will be covered later in this chapter. We will cover conventions for those topics as they are introduced.

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



Rumor has it that, on rare occasions, we developers sometimes screw things up. In the 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 with log-level `INFO`. This can be a helpful way to verify that our conventions are picking up all the types we think they should.

Message versioning

If you've ever had to build and then maintain a web service for any significant amount of time, you would have 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.

So how do you support versioning in a message-based system? Let's take another look at an example from *Chapter 2*, *Messaging Patterns*, containing the event that announced that 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, how would we know without a timestamp which one was correct?

To add a timestamp to a message without breaking any current subscribers, we can inherit from the original message. The following code will add a timestamp to the `IUserCreatedEvent`:

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

Now, instead of publishing an `IUserCreatedEvent` interface, we publish an `IUserCreatedWithTimeEvent` interface, 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) { }
}
```

Polymorphic dispatch

This example works because of **polymorphic dispatch**. When we publish the interface named `IUserCreatedWithTimeEvent`, NServiceBus says, "here is a message with properties called `UserId`, `Name`, `EmailAddress`, and `Timestamp`. It is both an `IUserCreatedEvent` and an `IUserCreatedWithTimeEvent` events." 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 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 for polymorphic dispatch:

```
public interface ICorporateUserCreatedEvent : IUserCreatedEvent
{
    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 interface named `ICorporateUserCreatedEvent`, and the message handlers for both the event types will be invoked. This enables us to hook additional functionality that is dependent upon `CompanyId` to the corporate user. When writing these types of polymorphic message handlers, we can package multiple handlers together into one message endpoint.

Events as interfaces

Message versioning using inheritance makes a lot of sense. It means V2 inherits from V1, V3 inherits from V2, and so on. However, polymorphic dispatch also enables some other interesting scenarios that would never be possible in a traditional web service or **Remote Procedure Call** (**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` interface, we also had an `IUserLoggedInEvent` interface, and because you had to verify your email address after your user was created and before you could log in, these operations never happened at the same time. Now suppose that we are going to integrate an enterprise user database system where all email addresses were already verified. We need to support single sign-on between these two systems. So, if a user comes from the corporate site, where we already know their email address, 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 IUserCreatedBySingleSignOnEvent :
IUserCreatedEvent, IUserLoggedInEvent { }
```

When we publish `IUserCreatedBySingleSignOnEvent`, it will be received by subscribers of `IUserCreatedEvent` and `IUserLoggedInEvent`, in addition to the endpoints that subscribe to `IUserCreatedBySingleSignOnEvent` directly. Each handler gets the data it needs to do its job, and our event definitions provide a clear definition of how the events are interrelated.

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 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 in 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 all the handlers should do their work autonomously.

At some point, it may become necessary to control the order in which handlers run. This is, in most cases, a code smell, as it indicates that your message handlers are not truly autonomous. Perhaps, if handlers must run in a specific order, they should be refactored into one single handler, or the workflow should perhaps be divided so that the procedure is broken up into two different messages that can be processed independently.

In short, every effort should be made to avoid having to control the order of handler execution, but real life is messy and exceptions to the rule do exist.



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 can arrive in any order, and nothing can change that. This process sets the message handler ordering—the order in which the handlers for a given message will be executed.

If there's absolutely no other way around it, you can set message handler ordering on your `BusConfiguration` instance:

```
// To have only one handler execute before all the rest
cfg.LoadMessageHandlers<FirstHandler>();

// Or if we need to specify multiple orderings
cfg.LoadMessageHandlers(First<FirstHandler>.Then<SecondHandler>()
    .AndThen<AndSoOnHandler>());
```

The `.AndThen<THandler>()` method can be called as many times as is necessary, or not at all, to have an ordering of only two handlers, but you should definitely exercise restraint and avoid going overboard.

The Particular team is aware that this API is less than ideal, and so there is a good chance it could be removed in a future version of NServiceBus. You have been warned! This is really a leftover from prior versions of NServiceBus where certain handlers might have needed to be run first in order to do things such as authorize incoming messages. Nowadays, we have much better methods to accomplish these kinds of cross-cutting concerns, such as message mutators and modifying the NServiceBus pipeline directly, which we will learn about in *Chapter 7*, *Advanced Configuration*.



Whenever multiple handlers execute on the same message, it's important to note that two or more handlers may try to edit the same database entity within the same transaction, so it is very helpful if our data persistence technology supports the **Identity Map** pattern. In this pattern, database records are cached in memory during a transaction so that a second edit to the same entity will not overwrite the first. Instead, after the initial load of a record, subsequent loads return the same cached entity, and all edits persist as a single operation. If you're using Entity Framework, NHibernate, or RavenDB, you're covered as they all support this pattern. You can read more about the identity map pattern at `http://martinfowler.com/eaaCatalog/identityMap.html`.

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

For whatever reason, we might need to not only stop a message 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 this method:

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

When we call this method, the message is consumed successfully, and the ambient transaction will be committed. We are just electing to stop running additional handlers on it.

Deferring a message

Sometimes, we cannot process a message immediately 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 empty at that moment, we will just wind up processing this message again on the next round trip, so be careful when trying to implement a fairness scheme in 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.

Be careful if your message handler (or maybe a different message handler) is doing database work before calling the `HandleCurrentMessageLater()` method. No outgoing messages will be sent from a message handler if you call this method, but the ambient transaction will still commit, which means that any modifications you make to the database will persist. This is useful if you want to log the reason why you are deferring the message so that you can make a decision based on that later, but it can be a surprise if you were expecting the database transaction to be rolled back.

A more useful type of deferral is a timed deferral, commonly necessary when calling external web services that have rate limits. In this case, the delay that is possible by 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, object message);
IBus.Defer(TimeSpan delay, object message);
```

Instead of just moving to the back of the line, these messages are sent to the Timeout Manager, which uses a combination of queues and persistent storage to put a message into a holding pattern until it is ready to be dispatched, at which point the Timeout Manager will add it back to the main input queue. We will learn more about timeouts in *Chapter 6*, *Sagas*.

Forwarding messages

In *Chapter 3*, *Preparing for Failure*, we covered message auditing, which means sending 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 use forwarding to redirect that message, you may also need to call `Bus.DoNotContinueDispatchingCurrentMessageToHandlers()`. The forwarded message will still be treated as a successfully processed message and will also end up in the audit queue.

Message headers

NServiceBus messages support bundling 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 used to deal 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 in the `NServiceBus.Headers` class. They're much easier to see, however, in the ServiceInsight tool, which we will cover in *Chapter 8*, *The Service Platform*.

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.

Property encryption

If your message contains certain information that should be encrypted, such as credit card numbers or other data you want to keep away from prying eyes, you can instruct NServiceBus to perform encryption on a property level:

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

If you'd like to use unobtrusive mode conventions, you can use this code:

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

// Convention Definition
cfg.Conventions()
  .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
cfg.RijndaelEncryptionService();

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

// Configuration Element
<RijndaelEncryptionServiceConfig Key="BASE_64_KEY">
    <ExpiredKeys>
      <add Key="OLD_BASE_64_KEY_1" />
      <add Key="OLD_BASE_64_KEY_2" />
    </ExpiredKeys>
</RijndaelEncryptionServiceConfig>

// Or configure the keys in code
cfg.RijndaelEncryptionService("BASE_64_KEY",
optionalListOfExpiredKeys);
```

The concept of expired keys allows you to transition from an expired key to a new key without needing to update every part of your system. Newly created messages containing encrypted properties will be encrypted with the active key, but existing in-flight messages encrypted with an older key will still be successfully decrypted.

Of course, the encrypted data is only as safe as you keep the key, so storing the encryption key in the config file or source code 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 9*, *Administration*.

Property encryption is a good procedure if we only want to encrypt a limited amount of information. If you have a requirement of encrypting entire messages, you can easily do this with message mutators, which we will cover in *Chapter 7*, *Advanced Configuration*.

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 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 (let's not even talk about RAW) are fairly commonplace, and once the message serializer Base64-encodes the byte array in the message, you'll be looking at 3.3 MB. That's way too close for comfort considering that the average number of megapixels has nowhere to go but up. Now consider the limits on most cloud-based message transports that are frequently much less than one megabyte!

Some 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 of these messages in the memory, there will be a practical limitation even though it is not enforced.

In any case, the more compact your messages, the better your system will perform. Stuffing large objects in your messages is not best practice, but what other option do you 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 do this. 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 should 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 might have to clean up expired items manually.

You may be wondering why the data bus payload isn't 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>` wrapper does not fit with unobtrusive mode, but we have a way to specify the data bus convention as a `Func<PropertyInfo, bool>` expression. Here's a simple example that defines that any property name ending with `DataBus` is a data bus property and following is a rewritten command definition:

```
cfg.Conventions()
  .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 the `DataBusProperty<T>` wrapper object. The data bus serializer is able to distinguish between the two forms.



The **ConventionsSample** project included with this chapter contains an additional sample implementation of a DataBus convention that doesn't require special naming conventions.

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:

```
cfg.FileShareDataBus(@"\\server\share");
```

`FileShareDataBus` is a great choice for a system that's completely on premise or connected by VPN, where all endpoints have access to the same file share. You can create your own implementation (for instance, to transfer via FTP, or store in Azure Blob storage, or 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 up expired items.

Any custom data bus implementation has to be specifically registered with the NServiceBus framework. Here is a sneak preview of NServiceBus's dependency injection system, which we will learn about in depth in *Chapter 7*, *Advanced Configuration*. Starting from our familiar `BusConfiguration` instance as follows:

```
cfg.RegisterComponents(reg =>
    reg.RegisterSingleton<IDataBus>(new MyDataBusProvider()));
```

While the data bus is very useful, we must remember that the data bus operation is separated from dispatching the message itself through the message transport. While unlikely, it would be possible for the data bus operation to fail (for instance, due to a failed FTP server or a disk full error) even though the associated message is sent, so we must be ready to handle that eventuality.

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. When there is only an occasional need to swap data with another platform, NServiceBus allows us to expose a command handler as a **Windows Communication Foundation** (**WCF**) web service with very little effort on our part.

In order to expose a web service, we will need the following:

- 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 these 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, from which we cannot escape.

The main pain 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 access 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 to requests. This may require you to run Visual Studio with elevated privileges.

With an NServiceBus handler exposed as a WCF service, we can place a message on the service bus from a remote client that does not run its own instance of the bus, such as a third-party partner, a desktop application written in WPF or WinForms, or even a remote application written in a non-.NET language.

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, which presents problems if you ever plan to load-balance the webapp.

Additionally, the web service capability should not be used for cross-site messaging. This can be accomplished with the Gateway component, which will be discussed in more detail in *Chapter 9*, *Administration*.

Summary

This chapter explored advanced messaging techniques that allow you to really take control of what happens during message handler execution. We started by learning how to create message assemblies without a dependency on NServiceBus itself, which grants 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.

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 to interact with remote client code that may be on a different platform.

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.

6
Sagas

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 get 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 on 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 user's expectations. Consider the frequent flyer that 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 okay," 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 our batch jobs in the past where they belong.

Defining a saga

A **saga** is a different way to represent these long-running business processes, by taking multiple message handlers, similar to the ones we've created in the previous chapters, and organizing them together with shared state that persists over time. A message designed to start the process will create a new instance of stored saga data, and subsequent messages will all share and update this data throughout the saga's lifetime. Each saga message handler retains all of the same capabilities, such as fault tolerance and automatic retry, that we've come to rely upon, but with the simple addition of the stored state.

To demonstrate how to build a saga, we will revisit the example of the 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 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 the user 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`, creating a new class file for this, and starting 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 virtual string Name { get; set; }
  [Unique]
  public virtual string EmailAddress { get; set; }
  public virtual string VerificationCode { get; set; }
}
```

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

This code defines the saga class itself and the data storage it uses. The data class is the saga's memory and automatically persists 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 that ensure that only one thread can modify the saga data at once. I am also marking all the saga properties as virtual because this is required by the NHibernate saga persister, if that's the persistence we elect to use. We will discuss more on this in the following sections.

Before we proceed, note that Visual Studio isn't very happy with us right now because `Saga<TSagaData>` is an abstract class and we haven't implemented a required abstract method. So let's quickly allow Visual Studio to implement the abstract class for us, and remove `NotImplementedException` so that what we'll have left will look like this:

```
protected override void ConfigureHowToFindSaga(
    SagaPropertyMapper<VerifyUserEmailPolicyData> mapper)
{
}
```

This is a significant change in NServiceBus 5.0. In previous versions, this was a virtual method, and forgetting to implement it was the cause of much pain and suffering. We'll return to fill this momentarily.

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 (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 instance in the data store.

Implement the `Handle()` method as shown here:

```
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 of 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 in case they have forgotten. We will cover this later 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` command 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 the 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 bit of help with this, by telling it how to match message properties with saga data. Because the `ConfigureHowToFindSaga` method is now abstract in 5.0, we have already created it, but now we have to fill in the details using the `mapper` object it provides:

```
mapper.ConfigureMapping<CreateNewUserCmd>(msg => msg.EmailAddress)
    .ToSaga(data => data.EmailAddress);
```

The `ConfigureMapping<TMessageType>` method accepts an `Expression` (a specialized lambda expression that can be evaluated at runtime) 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.

For example, if a saga is identified by the combination of `CustomerId` and `OrderId`, you would create a string property called `SagaKey` that equals `CustomerId + "/" + OrderId`, and then pass that value around in all your messages.

Ending a saga

When the user receives this email, they are obviously 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` class, 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 that we already have. The routing information we have in the `Web.config` file 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<UserVerifyingEmail Cmd>` 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 ensure that the incoming verification code matches the one we already have stored in our saga data. If it does, we send another new message (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 to our `ConfigureHowToFindSaga` method:

```
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`, within both 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. **UserService** should tell us that it's sending a verification email with the verification code. Jot that down so that we can simulate the email click.
4. To simulate the link click, navigate to `/VerifyUser?email=david@example.com&code={VerificationCode}`.
5. **UserService** should tell us that it's now creating the user.
6. **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 what is better than just an anonymous alarm clock is that we're also able to pass some state into the future, so our saga will not only wake up on command but also know why it woke up.

In our case, we want two timeouts. When the user first attempts to register, we want a wake-up 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 the database.

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
{
}
```

The difference between these two classes 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 information. 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 that this time, we specify that it will be a reminder email using 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 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 that at some point, data needs to be saved using the persistence strategy you selected for your endpoint, which could be NHibernate or RavenDB. Or you could even roll your own saga storage by creating and registering an implementation of `ISagaPersister` (we will cover dependency injection in *Chapter 7*, *Advanced Configuration*).

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. Additionally, if you use a different database for your business data, you would force the extra overhead of a distributed transaction to the database being used.

For this reason, message handlers within sagas should be used for message processing and business logic only. Messages or timeouts go in, decisions are made, and then messages 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!

Therefore, 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's 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 that, in this example, you don't even have to add a mapping for the `ReplyMsg` class in `ConfigureHowToFindSaga()`. Of course, we could also publish a message from the handler, and the saga could subscribe to it like any other message.

At some point, your saga may need to report back to the originator, that is, the sender of the message that started the saga. The `Bus.Reply()` method won't work in this case, as it only replies to the sender of the message currently being processed. To reply back to the sender of the message that started it all, 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 in this chapter 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 2,800 miles, and a timeout would be set to decrement the running total by 2,800 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.

This is the same concept as any sort of *preferred customer* situation where the sum of something over a given time range drives a business outcome, but this is hardly the only type of never-ending saga. Any sort of scheduler that manages interactions around a given resource could be a never-ending saga. Consider a process charged with downloading and processing an RSS feed for content integration. The saga could be charged with setting a timeout to download the feed again, 15 minutes after it is successfully processed, and if unsuccessful, change the download status to an error state and publish an event so that administrators can take a look at the remote feed, fix it, and then transition it back into the active state.



Savvy readers may notice that there really isn't much of a 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 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.

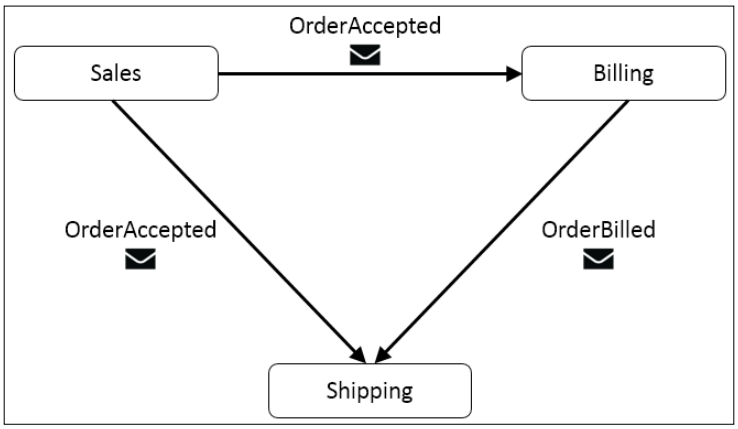
These two styles are the reflection of the two types of messages. The controller pattern sends commands, and the observer pattern reacts to events. In reality, these two styles can be mixed in any number of combinations.

An in-depth examination of various saga patterns is beyond the scope of this book. However, the topic has already been covered thoroughly 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/`.

Messages that start sagas

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

Consider a saga that controls the shipping of a product. In the following diagram, we have **Sales**, **Billing**, and **Shipping** services. The **Sales** service publishes an `OrderAccepted` event, and upon receiving this, the **Billing** service charges the credit card and publishes an `OrderBilled` event.



The **Shipping** service will contain a saga that waits to receive `OrderAccepted` and `OrderBilled`, and after that, the product can be shipped. At first, we might be tempted to say that only `OrderAccepted` can start the saga. After all, `OrderBilled` cannot be published until after the Billing service receives `OrderAccepted`.

But not so fast! Even though, logically, `OrderBilled` cannot happen in time until after `OrderAccepted`, it might be possible for the copy of `OrderAccepted` bound for the **Shipping** service to be delayed temporarily, perhaps because a server is temporarily offline. If this happens, `OrderBilled` could arrive and be processed by the **Shipping** service first.

If `OrderBilled` does not start the saga, then no saga data will be found and the message must be discarded. This is not good. Instead, we need to allow the `OrderBilled` event to start the saga, populate the saga data, and then check to ensure that all of the saga's conditions (both `OrderAccepted` and `OrderBilled` have been received) have been met, no matter which message arrives first.

The key takeaway here is that a saga can be started by more messages than you might at first think. Unless a message cannot occur until after a saga has processed its first message (like `UserVerifyingEmailCmd`, or if the saga itself sends the message), then you should implement it as `IAmStartedByMessages<T>`, not `IHandleMessages<T>`. Of course, for every message implemented as `IAmStartedByMessages<T>`, you should fill in the saga data properties with the assumption that they all are uninitialized.

Retraining business stakeholders

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 likely 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 example of the frequent flyer program, 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 were available. With a little back and forth, you can discover the true system requirements.

Persistence concerns

Storing saga data depends upon a **saga persister** that is able to translate our C# classes into a data store and back again. Because the different persistence mechanisms behave in wildly different ways, this introduces some behavior that we need to be aware of when creating our saga data classes.

We won't really discuss in-memory persistence because it is not fit for production use, and because by just storing the object in memory, it really doesn't have to do much of anything. However, we do need to mention a few things about RavenDB and NHibernate.

RavenDB

RavenDB is a document database that serializes C# class structures to JSON and then stores that JSON in its data store, identified by an ID. This makes it capable of storing a potentially complex object graph without a lot of hassle.

However, one implementation detail of RavenDB is that while storing or loading a document by its ID is a fully transactional operation, querying by any other document property is not. This is because RavenDB uses asynchronously updated indexes for queries by anything other than the ID. This means that the RavenDB saga persister must rely on the `Unique` attribute mentioned earlier in order to build a **pointer document** to the saga itself.

In our example for this chapter, we have `EmailAddress` as our unique attribute. The RavenDB saga persister will generate a unique ID based on the value of the email address, and use that as the ID of a document that will point to the saga document. RavenDB can then return both documents in one fully transactional round trip to the database.

This information might come in useful if you go to RavenDB Studio to look at stored saga data, but the primary takeaway here is just how important the `Unique` attribute is to the RavenDB saga persister. If it is not included, the persister will be forced to query the asynchronously updated index store, which may mean (especially under conditions of high load) that it will fail to find a saga's data, which should actually exist.

NHibernate

As suggested by its name, the NHibernate saga persister uses NHibernate, an object-relational mapper, to map saga data properties to a relational database. This means that NHibernate needs to examine your saga data and create a database table for each saga type, with columns for all of your properties. If your saga contains a child object, this means that property names will be concatenated to force your data into one table. If your saga data contains a collection, either of primitive or complex types, NHibernate will have to create an additional table to store those values.

Saga data must be transactional, and so locks must be taken out on saga data tables. So the more tables your saga data is forced into, the more you might observe locking and contention. When using the NHibernate saga persister, it may be worth your time to minimize the amount of tables necessary to get the best performance.

Additionally, in order to provide the ability to lazy load information from the database without inheriting from a special NHibernate base class, all properties used with the NHibernate saga persister must be marked as `virtual`. If you forget this, you will get a **The following types may not be used as proxies** exception. Luckily, these exceptions also point you to exactly what you need to correct.

NHibernate requires virtual properties because behind the scenes, it will create a class that inherits from your saga data class, intercepting your requests for data. This means that if a saga message handler doesn't need access to a collection (and by extension, an additional table) it won't bother to load from (and put a lock on) that table. In order to inherit from your class and intercept your properties, all those properties must be marked as `virtual`. It's a small price to pay for a pretty important performance optimization.

Azure

Windows Azure includes many cloud services, but the NServiceBus persistence mechanism for Azure utilizes Azure Table Storage Service, which is fairly limited in comparison to a full relational database. If you are using Azure storage for your sagas, your saga data must be flat (there must be no nested entities or collections) and only a limited set of data types (`string`, `int`, `long`, `double`, `bool`, `DateTime`, `Guid`, and `byte[]`) are supported. Full details about Azure storage can be found here at `http://msdn.microsoft.com/en-us/library/azure/dd179338.aspx`.

For the other uses of NServiceBus persistence, such as subscription storage, Azure Table Storage works quite well because the data is very structured to begin with. In sagas, it can be quite a bit more constraining in all but the simplest of sagas, but you may be able to get by with a few design workarounds. For example, in a saga where you are tracking completion of multiple tasks with numeric IDs, you could store a list of IDs as a comma-delimited list inside a string instead of a nested collection.

If the Azure storage proves to be too limiting, you have other options that you could utilize. The RavenDB saga persister can be used from Azure to connect to a RavenDB database hosted in RavenHQ (cloud-hosted RavenDB is available at `http://ravenhq.com/`) or a RavenDB database hosted on Azure, which you can find in the Azure Marketplace. You can also use the NHibernate saga persister to connect to SQL Azure. As we learned in *Chapter 4*, *Hosting*, it is fairly simple to select a different persistence mechanism just for saga data, using the following pattern:

```
cfg.UsePersistence<TPersistence>().For(Storage.Sagas);
```

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 so short that you could observe it 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 for you to have time to be ready and observe what happens, but short enough so 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, which is 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 with testing a saga, create a new class library and use NuGet to install the **NServiceBus.Testing** package and whichever unit testing framework you prefer. All 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. Mostly, 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, which returns an object that will allow us to start our test script. We wouldn't have to assign our saga 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 this 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 as follows:

- We expect the saga to send a `SendVerificationEmailCmd` message. We pass it to 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 cause the saga to expire.

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.

Testing events as interfaces

In this book, I have advocated implementing events as interfaces for the reasons outlined in *Chapter 5, Advanced Messaging*. This does introduce a bit of a wrinkle; without a concrete class, you can't build an instance of an event to feed it to the saga under test.

The test framework has a helper for the following code:

```
// Option 1: Create an instance of a message that we can set
properties on
var evt = Test.CreateInstance<TMessage>();
// Option 2: Create an instance of a message and run an initializer
action on it
var evt = Test.CreateInstance<TMessage>(Action<TMessage> action);
```

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:

```
public class ScheduleTasks : IWantToRunWhenBusStartsAndStops
{
  public IBus Bus { get; set; }
  public Schedule Schedule { get; set; }

  public void Start()
  {

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

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

  public void Stop() { }
}
```

The scheduler is like a mini-saga that can send messages at particular times. The important thing to keep in mind while using schedules is to steer clear of any custom logic. The scheduler should only initiate a message. Then you can do whatever needs to be done 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, or use a more full-featured scheduling library such as Quartz.NET.

On endpoint startup, a scheduled task is dispatched immediately, and then the timeout for the next run is only valid for the current run of the `AppDomain`. Therefore, it is possible for a task to run more frequently than the schedule if, for example, the server restarts in the middle. As such, this feature is best suited for short-range timeouts of the order of minutes rather than long timeouts of multiple days.

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 also 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.

Then 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 might be too used to the old way of doing things to realize what can be accomplished 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 help us 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
Advanced Configuration

Neo: What are you trying to tell me? That I can dodge bullets?

Morpheus: No, Neo. I'm trying to tell you that when you're ready, you won't have to.

-The Matrix (Warner Bros., 1999)

When I first saw *The Matrix*, before it was kind of ruined by its two sequels, it absolutely blew my mind. At this point in the movie, Neo has been shown the Matrix and he has discovered that he can manipulate it, to a certain extent, to be stronger, faster, and more awesome than everyone else. However, Neo hasn't yet fully made the leap and realized that he can remake the Matrix in any way he pleases.

You are Neo right now. You have learned a lot about the Matrix, that is, NServiceBus, but you are still playing by its rules. In reality, NServiceBus is an amazingly pluggable and configurable piece of software. Some of its rules can be bent. Others can be broken. In this chapter, you will learn how to configure and reconfigure NServiceBus into something completely unrecognizable, if you choose. I'll teach you how to bend the spoon.

First, you will learn how to modify NServiceBus through its general extension points using custom dependency injection. After this, you'll learn a few more advanced configuration constructs before the (new for 5.0) NServiceBus pipeline is covered, which provides the ultimate ability to add or replace behaviors in the incoming and outgoing message pipelines in order to remake NServiceBus in any way you might choose.

Why dodge bullets when you can stop them in their tracks? I'll leave the uncomfortable black leather outfits up to you.

Extending NServiceBus

NServiceBus allows you to configure the host process by implementing specific interfaces that it will find when performing assembly scanning on startup. This includes two configuration interfaces, `IConfigureThisEndpoint` and `INeedInitialization`, that define a `Customize` method, which gives you access to the `BusConfiguration` instance, and an `IWantToRunWhenBusStartsAndStops` endpoint startup interface, which allows you to perform nonconfiguration startup tasks.

Let's take a look at these interfaces in the order in which they are executed.

IConfigureThisEndpoint

We have already seen this interface, as the single class that implements it (commonly called the `EndpointConfig`) defines an endpoint hosted by the NServiceBus Host process. When the host process starts up, it scans all the assemblies in the runtime directory for this interface, and when it's found, NServiceBus gives the class implementing it the first chance to configure NServiceBus through the `BusConfiguration` object.

The `EndpointConfig` also sometimes implements at least one other interface, either `AsA_Client` or `AsA_Server`. `AsA_Server` signifies that this is a server endpoint, and as such, it should be set up to be a transactional endpoint, not purging the input queue on startup. By contrast, `AsA_Client` indicates that the endpoint operates more like a web application: nontransactional and with no message handlers as well as purging the input queue on startup. This is fairly uncommon, so most of the time, we will use `AsA_Server` for our NServiceBus Host endpoints.



In the previous versions of NServiceBus, there was also an `AsA_Publisher` interface to indicate that the endpoint will publish messages, requiring setup of subscription storage and a few other things. (I personally would tend to forget about this during live coding demos, much to my repeated chagrin.)

In NServiceBus 5.0, `AsA_Publisher` is deprecated. NServiceBus is smart enough to configure settings for message publishing when necessary.

While `AsA_Server` still exists in the NServiceBus 5.0 public API, its behavior is the default and its implementation is a no-op, so it isn't really required to include it. `AsA_Client` is the only meaningful modifier interface left.

The configuration of NServiceBus settings is now order independent, so allowing the endpoint-specific settings to be made first in the `EndpointConfig` class allows the internal features of NServiceBus to configure themselves without stepping on your toes.

INeedInitialization

The workhorse of NServiceBus configuration is `INeedInitialization`. Every class that implements this interface will be detected by NServiceBus on startup and it will be activated, both in an NServiceBus Host endpoint and in a self-hosted endpoint. Similar to `IConfigureThisEndpoint`, it offers a `Customize()` method that gives you access to the `BusConfiguration` instance:

```
public class MyInit : INeedInitialization
{
  public void Customize(BusConfigurationcfg)
  {
    // Perform customizations here
  }
}
```

Because these classes are automatically detected and invoked and the signature of the `Customize` method is exactly the same as `IConfigureThisEndpoint`, this is the perfect place to centralize all the endpoint settings that are common to your entire system. You can create a single assembly named `YourCompany.Conventions` with all of your settings defined in the `INeedInitialization` classes, and then drop this assembly into each and every endpoint, both hosted and self-hosted. With this in place, your `EndpointConfig` classes can be, for the most part, completely empty.



In the previous versions of NServiceBus, the order of certain configuration steps made a lot more difference, and as a result, there were a lot more configuration interfaces that provided access to the config at different times in the startup process. Now that the precise ordering of different configuration steps is no longer necessary, these interfaces from the older versions of NServiceBus are all obsolete:

- `IWantCustomInitialization`
- `IWantToRunBeforeConfiguration`
- `IWantCustomLogging`

In addition to these that are already obsolete, the following two extension points will eventually be made obsolete in NServiceBus 6.0, so it will be unwise to start using them now:

- `IWantToRunBeforeConfigurationIsFinalized`
- `IWantToRunWhenConfigurationIsComplete`

It is one of the great achievements of NServiceBus 5.0 that you can use `INeedInitialization` for all of your configuration needs.

IWantToRunWhenBusStartsAndStops

This is the last general extension point. It is unique since it is not necessarily a configuration point but a way to define the application startup (or teardown) tasks when the service bus has been fully configured. It has the `Start()` and `Stop()` methods, allowing you to perform cleanup operations when the endpoint stops, although `Stop()` is not invoked for send-only endpoints.



In the previous versions of NServiceBus, this interface was called `IWantToRunAtStartup`, so you will most likely see lots of examples on the Internet using this old name.

Because they are not focused on configuring the system, classes that implement `IWantToRunWhenBusStartsAndStops` are usually more geared toward the local endpoint code and are not generally shared in conventions assemblies between multiple endpoints. You also have full access to dependency injection, meaning that you can inject an `IBus` instance and send or publish messages. As such, it is probably only a matter of time until you find yourself creating one for debugging to quickly send a message each time you press the *Enter* key.

It is important to remember that unlike a message handler, these methods have no ambient transactions or try/catch semantics. So, it is possible for an uncaught exception here to cause the entire process to fail. Therefore, it's not a good idea to do a lot of work here. Send a message and have it processed transactionally instead!

The `Stop()` method can be a little difficult to observe. When an endpoint is installed as a Windows service, it fires when the service stops. You can also observe it from a console window if you interrupt the process by pressing *Ctrl* + *C*.

An implementation of `IWantToRunWhenBusStartsAndStops` is a great place to create a quick interface in order to test messages during debugging by allowing you to send messages based on the console input. Apart from this, it isn't common to have widespread use of them in a production system. One possible production use case will be to provision a resource needed by the endpoint at startup and then tear it down when the endpoint stops.

Dependency injection

You already know that you can use the `BusConfiguration` instance provided by `IConfigureThisEndpoint` or `INeedInitialization` to set many of the options introduced in *Chapter 4*, *Hosting*, which gives you a great amount of control over how the bus operates, but you can also make your own customizations. This will usually involve dependency injection.

Let's say that we want to be able to create unit tests in order 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 the implementation of retrieving the current time 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 DateTimeUtcNow
    {
        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` class injected into our message handler classes so that we can use it.

To configure our `DateTimeProvider` class in the dependency injection container, we start with our `BusConfiguration` instance from either our `EndpointConfig` class or an `INeedInitialization` class and begin to register our components. We'll continue to call the `BusConfiguration` instance by the name `cfg` for brevity. Here, I will use the multiline lambda expression (with the curly braces after `=>`) because it is fairly common to want to register more than one dependency:

```
cfg.RegisterComponents(reg =>
{
  reg.ConfigureComponent<DateTimeProvider>(
   DependencyLifecycle.SingleInstance);
});
```

The preceding code will look at the `DateTimeProvider` class to determine which services it provides, or in other words, which 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 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` instance will be created for every request, and `InstancePerUnitOfWork`, which means that a new `DateTimeProvider` instance will be created 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 `ConfigureComponent` method, which accept factory delegates that allow you to specify how to construct the object being injected, but these are generally only used in advanced scenarios.

All the `ConfigureComponent` methods return an `IComponentConfig` instance that allows you to instruct the container to set properties on the injected objects:

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

The preceding code takes an expression that 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 a fluent style.

Using these methods, you can abstract a lot of services that your message handlers will need to do their job, resulting in a 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 Development, Test, QA, or Production. We will explore how to do this in more detail in *Chapter 9*, *Administration*.

Unit of work

Because messages are processed in a pipeline of handlers, you will, at times, have 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 you need to create a database session before handling a message and then commit or rollback any changes at its completion. One method to accomplish this is by using a unit of work implementation. We will discuss another method when we cover the pipeline later in this chapter.

In order to define a unit of work implementation, you must first 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---");
  }
}
```

The `Begin()` method is invoked before the first message handler is executed, and the `End()` method is invoked after all the message handlers have completed execution. If an error occurs during message processing, an exception is passed into the `End()` method, otherwise it will be null.

Unit of work implementations are not automatically invoked; you need to specifically instruct the dependency injection container to use them, as follows:

```
public class ConfigureUOW : INeedInitialization
{
    public void Customize(BusConfiguration cfg)
    {
       cfg.RegisterComponents(reg =>
       {
           reg.ConfigureComponent<ConsoleUnitOfWork>(
               DependencyLifecycle.InstancePerCall);
       });
    }
}
```

While it might 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 unit of work 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 in 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. Access to a message is provided as an object, so usually some amount of reflection is required to implement anything of substance. There are three possible interfaces to implement to create an applicative message mutator:

- 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 the previously mentioned interfaces, to manipulate both incoming and outgoing messages

A **transport message mutator** is used to act on transport messages, which contain the message serialized to a byte stream and ready to pass to the message transport, getting you much closer to the bare metal. You can even manipulate the bytes of your message, which means that a transport message mutator is the perfect place to implement functionality such as full-message encryption, signing, header manipulation, or compression. There are three possible interfaces to implement to create a transport message mutator:

- 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 unit of work implementations, so you must register them yourself:

```
reg.ConfigureComponent<MyMessageMutator>(
  DependencyLifecycle.InstancePerCall);
```

The NServiceBus pipeline

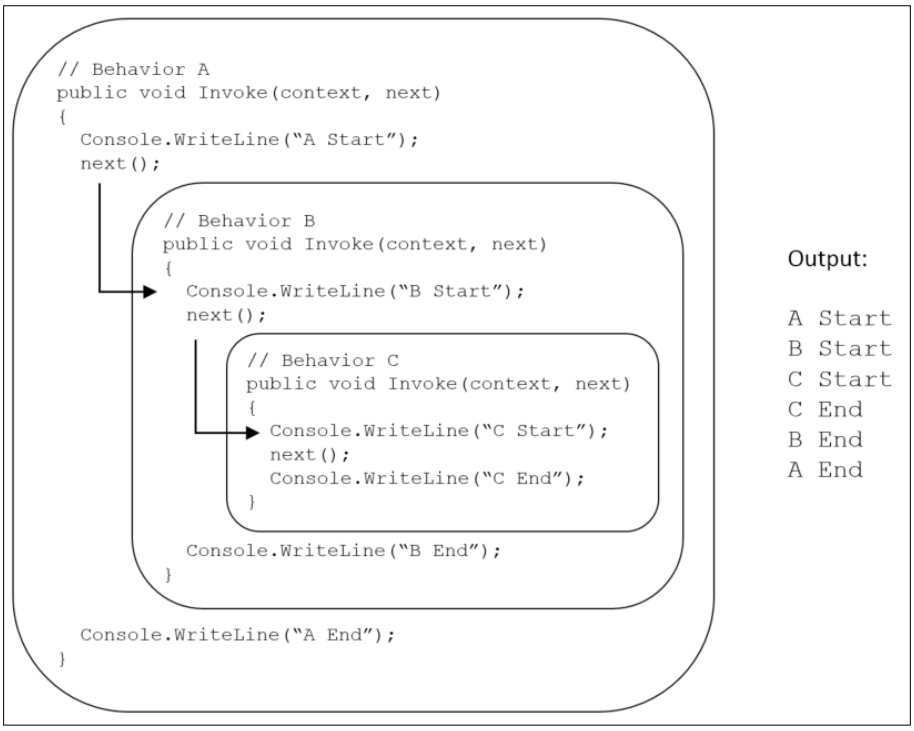
Abstractions such as unit of work and message mutators are useful when they perfectly fit what we need to do with them. However, they are abstractions after all, and all nontrivial abstractions, to some degree, are leaky.

What does this mean? If you take a look back at the API for a unit of work, for example, we only get a `Begin()` and `End()` method to do any useful work. While this is extremely valuable in some cases, it can also be extremely limiting. Try using it to wrap the message processing in a try/catch or using block, for instance, you will quickly find that the abstraction is fighting against you, not for you.

With NServiceBus 5.0, you get the full power of the new NServiceBus pipeline, giving you the ability to get ridiculously close to the metal and customize nearly every aspect of how NServiceBus sends and receives messages.

The pipeline is new in NServiceBus 5.0 and was originally conceived by the Particular team as a way to simplify and streamline how all the different parts of NServiceBus were wired up together. It was also very useful internally for providing customers with customized solutions to unique and novel requirements, but they quickly found that it will be very useful as a general NServiceBus feature as well.

The pipeline itself is based on a Russian Doll Model, meaning that various behaviors are nested within each other, similar to Russian nesting dolls, as shown in the following diagram:



As you can see in the preceding diagram, each behavior is required to call the `next()` action, which transfers control to the next behavior in the chain, and after a behavior is completed, the control falls back to the outer behavior.

In NServiceBus 5.0, many features that were previously implemented by other means have all been refactored to exist as behaviors. If you're of the curious type, reflecting on the NServiceBus assemblies with a tool such as JetBrains dotPeek to find the source code for the classes that implement `IBehavior<T>` can be a very educational experience.

Building behaviors

A **behavior** is a class that contains an `Invoke()` method that takes some action on an incoming or outgoing message and passes along control to the next step by calling the `next()` action. In addition, a behavior can communicate with other behaviors in the pipeline via a shared context object.

Here's the simplest possible behavior for an incoming message. Since it does nothing, I'll call it a `NoOpBehavior` class:

```
public class NoOpBehavior : IBehavior<IncomingContext>
{
  public void Invoke(IncomingContext context, Action next)
  {
    next();
  }
}
```

The corresponding behavior for an outgoing message will use an `OutgoingContext` parameter in place of the `IncomingContext` parameter shown. This behavior does nothing, but the lack of distractions serves to illustrate exactly how a behavior operates. It is called at some point when a message is being processed; it can take some actions on the message or communicate with other behaviors via the context, and then it cedes control to the next behavior in the line by calling the `next()` action.

What is really useful is that eventually, the next step in the chain will be completed and the code execution will resume in our behavior just after the `next()` action is called. This opens up a lot of interesting possibilities such as the following:

```
public void Invoke(IncomingContext context, Action next)
{
    // Take action before the next behavior begins
    // by calling next() at the end of Invoke
    next();
}

public void Invoke(IncomingContext context, Action next)
{
    next();
    // Take action after the next behavior ends
    // by calling next() at the beginning of Invoke
}

public void Invoke(IncomingContext context, Action next)
{
    // Wrap the next action in a using or try/catch/finally
```

```
        using(var something = new DisposableThing())
        {
            // Add the thing to the context for use in other locations
            // in the pipeline
            context.Set(something);
            next();
        }
    }
```

Of course, when you fall out of the end of your `Invoke()` method, control will be delivered back to the behavior before you in line. All of this is possible without any one behavior knowing much of anything about any of the others. All that each one needs is the rules for how to follow each other in line and someone to set them up in the proper order.

Ordering behaviors

Behaviors don't really have any concept of ordering on their own. Instead, they are contained within **steps**, which are basically named items arranged in an order. You can think of steps as buckets arranged in a line, which can contain behaviors.

NServiceBus contains 16 well-known steps in the aptly named `WellKnownStep` class. These well-known steps, in their order of execution, are given in the following table:

Incoming pipeline	**Outgoing pipeline**
`CriticalTime`	`MutateOutgoingMessages`
`AuditProcessedMessage`	`CreatePhysicalMessage`
`CreateChildContainer`	`SerializeMessage`
`ExecuteUnitOfWork`	`MutateOutgoingTransportMessage`
`MutateIncomingTransportMessage`	`DispatchMessageToTransport`
`DeserializeMessages`	
`ExecuteLogicalMessages`	
`MutateIncomingMessages`	
`LoadHandlers`	
`InvokeSaga`	
`InvokeHandlers`	

These well-known steps, of course, contain their own behaviors, which perform the activities suggested by their names, but they also serve as landmarks that control the ordering of steps added by NServiceBus plugins or by us. To do this, we create a class that inherits `RegisterStep` class.

Here is an example that will place a behavior just before `LoadHandlers`. As you'll see in just a bit, this makes it possible to use your behavior to inject dependencies into your message handler classes.

It's customary to make this step registration class an inner class of the behavior class itself (that is, nested inside it) in order to keep the two together:

```
public class MyBehaviorRegistration : RegisterStep
{
  public MyBehaviorRegistration()
     : base("StepName", typeof (MyBehavior), "Step Description")
  {
    this.InsertBefore(WellKnownStep.LoadHandlers);
  }
}
```

The `RegisterStep` base class contains the `InsertBefore`, `InsertAfter`, `InsertBeforeIfExists`, and `InsertAfterIfExists` methods to control where the step will be placed. Each method accepts either a `WellKnownStep` instance or the step's name as a string.

Keep in mind that a step might not necessarily exist. A notable example is the well-known step `InvokeSaga`, which is not created if no sagas are present in the code at runtime. You can specify multiple rules of both the `Insert*` and `Insert*IfExists` types in order to determine a step's position. NServiceBus will make sure that all are heeded, or it will throw an exception if you have asked for something that is impossible.

One last thing is needed to wire up our step containing our behavior. The ordering of the steps is critical, so they must be registered manually with the pipeline:

```
public class MyStepInit : INeedInitialization
{
  public void Customize(BusConfigurationcfg)
  {
    cfg.Pipeline.Register<MyBehaviorRegistration>();
  }
}
```

Using an independent `INeedInitialization` parameter in this manner will allow you to register multiple steps in their correct order. However, if you have one step that is largely independent, you can easily combine step registration and initialization in the same class:

```
public class MyBehaviorRegistration : RegisterStep,
  INeedInitialization
```

```
{
  public MyBehaviorRegistration()
     : base("StepName", typeof (MyBehavior), "Step Description")
  {
    this.InsertBefore(WellKnownStep.InvokeHandlers);
  }

  public void Customize(BusConfigurationcfg)
  {
    cfg.Pipeline.Register<MyBehaviorRegistration>();
  }
}
```

If you recall that steps are like buckets that might contain a behavior, the registration process makes a lot more sense. We need an `IBehavior` class to contain our code, a `RegisterStep` class to hold the behavior and control its ordering, and an `INeedInitialization` class to register the behavior.

The code sample included with this chapter contains an example of a pipeline behavior that illustrates the ability to wrap `next()` within a `using` block, including all the code necessary to register the step with the pipeline.

Replacing behaviors

In rare situations, you might decide that you want to completely replace the NServiceBus implementation of a behavior. For example, let's say you wanted to make the transition from XML to JSON serialization, but you couldn't update every endpoint at the same time. Some will be upgraded and will send JSON, but others will still be sending XML. You will need a deserialization behavior that is capable of sniffing the transport message's contents, determining whether it was JSON or XML, and deserializing accordingly.

It will be silly to remove a step only to create a new step in the same position. In this situation, we can simply swap out the existing behavior with a new one. Take the following code snippet as an example:

```
public class ReplaceDeserializeStep : INeedInitialization
{
  public void Customize(BusConfigurationcfg)
  {
    cfg.Pipeline.Replace(WellKnownStep.DeserializeMessages,
      typeof(XmlOrJsonDeserializeBehavior),
      "Deserializes all the things!");
  }
}
```

The hard part, of course, is writing the code for the behavior itself, because it will replace the built-in NServiceBus code; this isn't a decision to undertake lightly. The code in the built-in NServiceBus behaviors is extensively tested and battle-ready; proceed at your own risk!

The pipeline context

The *hello world* of pipeline behaviors just prints some stuff out to the console, but this isn't all that useful. After all, we can do this much more easily with a unit of work implementation. What really makes a behavior useful is the context object.

Both `IncomingContext` and `OutgoingContext` inherit from the `BehaviorContext` class, which gives you access to the `Builder` in order to do dependency injection, and a series of methods (`Get`/`Set`, `TryGet`/`TrySet`, and `Remove`) that make the context function similar to a strongly-typed property bag.

With the `Builder`, you can create objects in your behavior, insert them into the context, and then have the container grab that instance back out of the context when it builds the message handler classes. You will commonly want to do this with a database context, such as an Entity Framework `DbContext` or RavenDB `IDocumentSession`. The behavior will create the object and wrap it in a proper `using` block, and then you will want access to this from within all your message handlers.

There are a couple tricks to pull this off. First, you will need to do this before the `LoadHandlers` step, because this is where the container instantiates the handler classes, and you have to have your database context `DbContext` or `IDocumentSession` or whatever else registered in the context before then, so that they can be injected into the handlers. The second trick is to use the `Builder` to fetch the `PipelineExecutor` class for the dependency injection registration:

```
// Register the dependency that the behavior will create
cfg.RegisterComponents(reg =>
  reg.ConfigureComponent<TDependency>(builder => {
    // First we use the builder to get the PipelineExecutor.
    var pipelineExec = builder.Build<PipelineExecutor>();
    // Now that we have that, get at the current
    // pipeline context and fetch our dependency
    return pipelineExec.CurrentContext.Get<TDependency>();
  }, DependencyLifecycle.InstancePerCall));
```

The sample behavior in this chapter's sample code contains a complete example of how to do this.

In addition to what is provided by `BehaviorContext`, the `IncomingContext` parameter also provides access to the incoming physical and logical messages. These messages will give you the ability to create behaviors that mimic message mutators, access to the current message handler class being executed, and a `DoNotInvokeAnyMoreHandlers()` method, which gives you a functionality similar to `Bus.DoNotContinueDispatchingCurrentMessageToHandlers()` but with significantly fewer keystrokes!

The `OutgoingContext` parameter also gives you access to the outgoing physical and logical messages, in addition to the incoming transport messages for the cases where you are sending a message from a message handler, plus some delivery options for the message being sent.

These are really just building blocks, which have the ability to tweak everything about incoming and outgoing messages at their most basic level. This might not give you the power to actually dodge bullets, but it is certainly enough to do a ton of interesting NServiceBus customizations much more easily than has ever been possible.

Outbox

One of the new features in NServiceBus 5.0 is important to understand, especially if you plan to run NServiceBus in the cloud. The Outbox feature is designed as a replacement for the **Distributed Transaction Coordinator** (**DTC**), and it is turned on by default for the RabbitMQ transport (which does not support DTC) but is turned off by default for other transports that support the DTC.

In order to understand Outbox, we should first understand, at least at a high level, what the DTC does for us.

DTC 101

Many resources, such as SQL Server or MSMQ for example, support local transactions that are unique to that resource. The `SqlTransaction` class is an example of a local transaction on a SQL Server resource.

DTC is a service built into Windows that is capable of controlling multiple local transactions simultaneously. A resource that supports the DTC can automatically upgrade its local transaction to a distributed transaction that is managed by the DTC. When we do a transactional receive against MSMQ, for example, and then perform database work, the transaction is promoted to a DTC transaction.

The DTC then becomes responsible for coordinating all these local transactions that have been enlisted in the distributed transaction using a two-phase commit process, which ensures that either all resource managers commit successfully or all of them abort it, ensuring consistency. In the first phase of a two-phase commit, called the **Prepare phase**, the DTC asks all the enlisted resource managers whether they are prepared to commit. In the second phase, called the **Commit phase**, the DTC gives all resource managers clearance to commit, assuming that all parties responded with a yes during the Prepare phase.

In its default configuration with the MSMQ transport, the DTC gives you the ability to retry your failed messages at will. Most of the time, your endpoint behavior will consist of receiving messages, making database changes, and sending new messages. These are all transactional activities that are wrapped up in one distributed transaction, so you can rest easy with the guarantee that all will succeed together or all will fail together.

However, the DTC does have its costs. All the chatter can be very expensive. When you're mostly on one machine, as will be the case with a local MSMQ and one database resource, it's not such a big deal. However, in the cloud, where resources aren't even guaranteed to live in the same datacenter, all of this communication to get all the enlisted cohorts on the same page can have a deleterious effect on your system's performance. Nobody claims that DTC transactions are blazing fast, but we generally accept them for the convenience and security they offer.

Life without distributed transactions

So, how would life be without distributed transactions? If you can guarantee that your message handlers are all idempotent, then you wouldn't need to do anything. With idempotent message handlers, messages can be processed any number of times and can fail at any point and we'd be no worse for wear. Unfortunately, life isn't always this tidy.

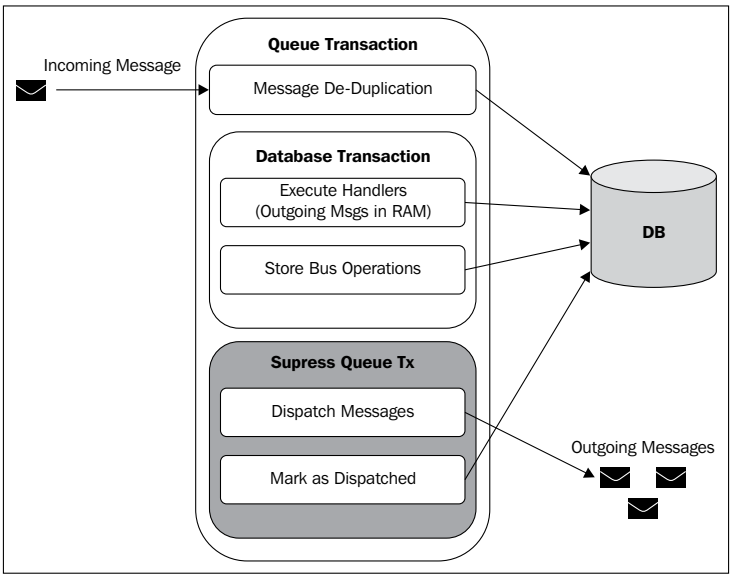
Let's say that in our message handler, we're creating a new entity and then publishing an event with the new entity's ID. Clearly, this is not idempotent; if the endpoint were to fail after the database transaction is committed but before the event is published, we can have duplicate *ghost* entities created. So, as a first step, we will need to run incoming messages through a **deduplication process**, meaning that we log a list of completed message IDs in the same database where we store our business data, taking advantage of that same local transaction. Then, if we find ourselves handling the very same message again, we know that we can safely discard it.

This addresses part of the problem but does not solve it. Because we want to publish an event during our message handling logic, and the sending of this message happens within the scope of a separate transaction, that publish will send out its message immediately, while the database transaction containing our business table updates and deduplication updates can roll back. In this case, our event will be announcing the creation of a database entity that was rolled back and, therefore, does not exist! Distributed transactions would have saved us from this problem because the outgoing message would also enlist in the transaction and would not actually have been dispatched until the transaction was committed.

The Outbox feature solves this problem by also deduplicating messages on the way out. Instead of talking directly to the queuing infrastructure, outgoing messages are stored in the database within the same transaction as the business data updates and the incoming message deduplication. After the database transaction successfully commits, NServiceBus can then read the outgoing messages back out of the database and forward them on to the queuing system.

If the server crashes after the database transaction but before the Outbox messages are sent out, NServiceBus will read the Outbox messages again and send them to their respective queues. It is possible that this can generate duplicate messages on the way out, but because the Outbox contains the unique IDs for the outgoing messages, these messages will have matching IDs and will be handled by the incoming message deduplication logic.

The following diagram provides a visual representation of message handling using the Outbox feature:



It's critical, however, to understand that while the figure shows the database transaction nested within the queue transaction, one does not control the other; they are completely independent transactions. It is only through the Outbox's coordination and message deduplication that you can make it behave as if you were using the DTC instead.

Outbox configuration

As mentioned previously, Outbox is enabled by default for the RabbitMQ transport because it doesn't support the DTC. However, we know that the DTC adds a *chitchat* overhead to our endpoints, so it is not completely inconceivable that you might want to use Outbox instead in certain situations in order to try to eke out every last bit of performance from an endpoint while still retaining transactional safety.



As of the release of NServiceBus 5.0, Outbox storage has only been implemented for NHibernate persistence, although this is likely to change in the future. For up-to-date information, visit `http://docs.particular.net/nservicebus/no-dtc`.

In these cases, you need to be very careful if the Outbox endpoint will be sending any messages to any DTC endpoints or to an older NServiceBus version 4.x endpoint. Remember that the way the Outbox works, it is entirely possible to end up sending duplicate messages, which another Outbox-enabled endpoint will handle through deduplication. If duplicates arrive at a DTC-enabled endpoint, it is not going to perform deduplication and will assume that each message must be processed.

Because of this potential for double processing messages, any DTC-enabled endpoint receiving messages from an Outbox-enabled endpoint must be idempotent! Really, this warning can extend to any non-Outbox endpoint, but if an endpoint doesn't have the DTC or the Outbox, then you're already on your own as far as idempotence is concerned anyway.

In order to make it nearly impossible to accidentally mess this up, enabling the Outbox on any transport other than RabbitMQ requires a double opt-in. First, you must enable Outbox from a `BusConfiguration` instance:

```
cfg.EnableOutbox();
```

Then, you also must add a bit of config:

```
<appSettings>
  <add key="NServiceBus/Outbox" value="true" />
</appSettings>
```

Together, these two settings should ensure that you don't accidentally enable the Outbox on a DTC-enabled transport without really thinking it through.



Although SQL Server supports the DTC, there is a supported configuration for using the Outbox with the SQL Server transport, but the configuration is a little too involved to be covered here. For configuration details, please visit the official documentation at `http://docs.particular.net/nservicebus/no-dtc`.

Session sharing

For Outbox to work, the Outbox storage must be in the same database as the business data you're acting upon. Otherwise, it will be possible for the business data to commit (you create the entity in the database) and the Outbox transaction to roll back, causing no messages to go out. Alternatively, the Outbox transaction can commit and the business data transaction can roll back. Either situation is bad.

In order to accomplish this, you must share the NHibernate session between the Outbox and the business data. To do this, inject a `NHibernateStorageContext` dependency into your handler, which will give you access to the current `System.Data.IDbConnection`, `NHibernate.ISession`, and `NHibernate.ITransaction`.

Summary

In this chapter, you learned how to configure NServiceBus to its fullest potential. First, we explored the interfaces that we can implement in order to make our own customizations and learned how we can use dependency injection to inject our own dependencies into the framework.

You learned a few easy ways to extend the NServiceBus pipeline via the unit of work pattern and message mutators, which can be useful if the goal we have in mind aligns well with the API. Failing that, you learned how to take control of the entire incoming and outgoing message pipelines in order to insert your own behaviors or replace existing ones.

Lastly, you learned about Outbox, which gives you equivalent protection to the DTC when it isn't available to you or if you purposefully decide to throw it out.

Like Neo in the Matrix, you can now exercise complete control over your environment when you're jacked into your NServiceBus code and bend the framework to your wishes. In the next chapter, you will discover the rest of the tools in the Service Platform, and you will learn how they make your NServiceBus messaging systems easier to debug, monitor, and build.

8
The Service Platform

Leaving aside my first programming experiences as a kid, banging out AppleTalk BASIC on my Apple IIGS, the first time I started any serious computer programming was in C and C++ as part of the electrical engineering curriculum in college, before I had the realization that I was destined to program computers for a living. For the record, all I remember from electrical engineering is that *V=IR*, *P=IV*, and everything else that is important can be essentially derived using calculus.

In order to complete our programming assignments in school, we would access a university server using SSH (or yikes, could it have been telnet?), type our programs in `vi`, `emacs`, or `pico`, and then compile them with `gcc`.

I loved programming but I absolutely hated that. I have never been any good at `vi`; I struggle to remember how to tell it I want to quit. I was okay with `pico`, but I seem to remember spending more time on one assignment figuring out how to set up an FTP file sync (so that I could write code locally in my graphical editor) than I did actually writing the code for the assignment itself.

I hope I don't offend any of my colleagues and contemporaries nowadays who use `vim`. You have to do what works for you, but it wasn't for me.

I love my IDE. Maybe that's an extension of my engineering training, a love of tools. I love how it gives me IntelliSense so that I don't have to remember everything at once. I also love how it shows me a lot of information about my project at a glance, and how it reminds me what to do when I'm stupid.

Being stuck without an IDE is a lot like the feeling you get when trying to run a distributed messaging system without appropriate tooling. This was what it was like to run a system based on NServiceBus 2.0 when I started using the framework, and it's very similar to what it would be like with any other service bus framework.

The tools that Windows provides to manage MSMQ, for lack of a better word, suck. This led many developers to fork over dollars on a tool called QueueExplorer, but even that tool had no awareness of how NServiceBus used MSMQ, so certain operations might have resulted in unintended side effects. I tried building my own tools, but I wasn't in the business of building tools, so predictably, they were awful. Of course, now NServiceBus supports many different transports, making this problem much more difficult.

The team at Particular realized this and built tools to address the need for different platforms. There are separate tools to make the business of managing, debugging, monitoring, and building distributed systems easier than ever before. Together with NServiceBus, they constitute the Particular Service Platform. Consider them your IDE for building and maintaining distributed systems.

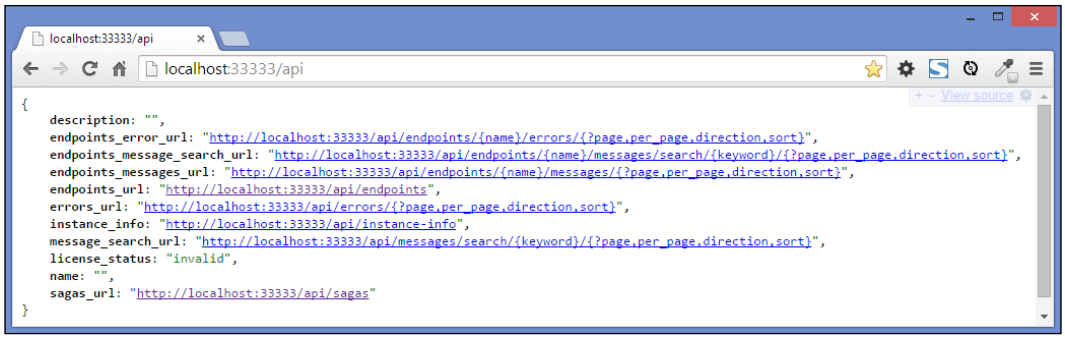
ServiceControl

The heart of the Service Platform is ServiceControl, an invisible service that runs on a central server and stores information about every message that runs through your system. Actually, if we were to use the metaphor of the human body, ServiceControl is the brain. It knows everything about your system that is knowable.

You may have noticed that the `app.config` file for every endpoint you've created has included an `AuditConfig` section pointing to a queue named `audit`. This means that every endpoint has been forwarding a copy of every processed message to the `audit` queue. Lucky for you, that queue has not overflowed yet, because ServiceControl (which you installed with the Platform Installer) has been busy consuming all of those messages and feeding the data into an embedded RavenDB database.

From the data stored in this database, ServiceControl exposes a REST API that allows the rest of the Service Platform tools to interact with the data. By default, this API is exposed at `http://localhost:33333/api`.

If you browse to the preceding link, you'll get a JSON response that gives some insight into the available endpoints, should you choose to go down that route, as shown in the following screenshot:



In addition to slogging through copies of every message successfully processed out of the `audit` queue, ServiceControl is also receiving the messages from your `error` queue, storing data in its database, and then dropping those messages into an `error.log` queue, ready to be returned to their original source queues in the future if you choose.

If we allowed the ServiceControl database to retain all of its data forever, our hard disks would fill up just the same as if it built up in the `audit` queue. Therefore, we can't really keep ServiceControl's data forever. By default, data for an audit message (which is itself a copy of the actual original message) is stored for 30 days, measured in hours. The process to purge the data is run once a minute so that it never has to work through too much of backlog at once.



As a part of being safe by default, error messages are never purged, since an error message is the primary copy of a message.

You can configure the expiration policy to store messages for a longer or shorter period; you just need to be prepared to handle the disk space requirements of your decision. However, the size should be kept to a manageable amount. If you require true long-term storage of audit messages, perhaps for compliance with a governmental mandate, it would be advisable to use the ServiceControl API to extract audit messages (and perhaps only a subset by type) and store them in an external database.

If you do extend the expiration policy, it is important to note that the underlying RavenDB store will not automatically give space back to the OS when the size of the database itself contracts.

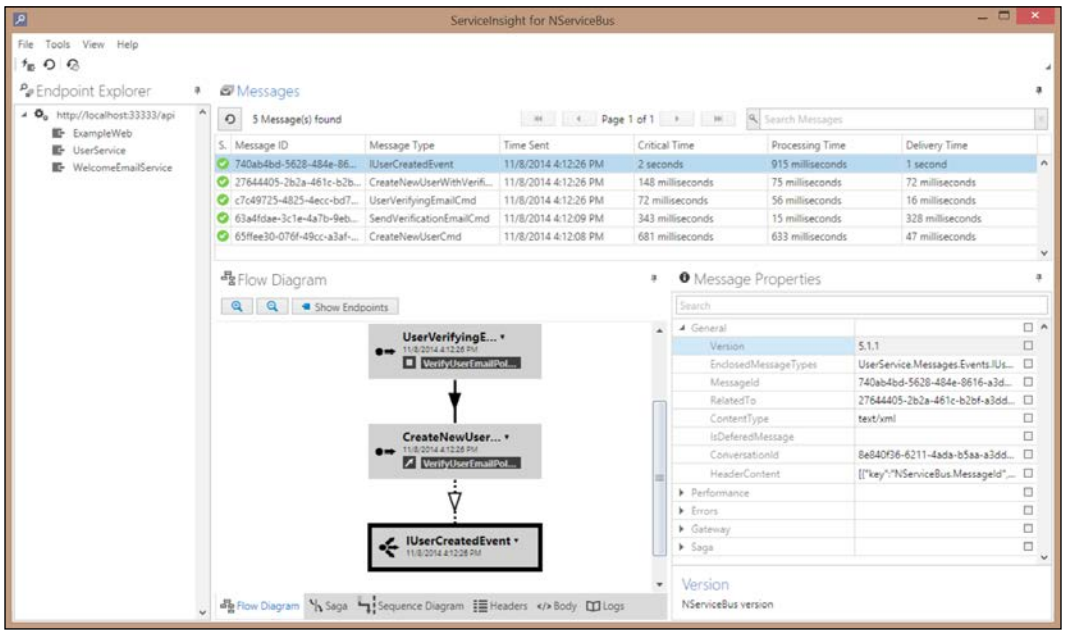
The expiration policy along with several other settings (API host, port, log directory, database location, and more) is configurable via the `ServiceControl.exe.config` file in your ServiceControl installation folder. For full details, visit the ServiceControl documentation at `http://docs.particular.net/servicecontrol/`.

If you are using a transport besides MSMQ, you will definitely need to configure it to use your chosen transport at the very least. This will also require downloading the assemblies for that transport, and configuring ServiceControl to use them. For detailed instructions, see `http://docs.particular.net/servicecontrol/multi-transport-support`.

ServiceInsight

If ServiceControl is the brain of your distributed system, then **ServiceInsight** is the eyes. The centralized API serving data about our messages enables us to have a desktop application that provides a GUI to visualize those messages and their interactions. This is the application that can help us debug and understand what our messaging system is up to. It is also one of the primary ways in which we can return a failed message to its original source, regardless of what messaging transport our system uses.

When you launch ServiceInsight, you will be greeted by a window containing several dockable areas that you can rearrange as you please. This window is shown in the following screenshot:



Endpoint Explorer

The **Endpoint Explorer** panel shows the ServiceControl API URL as a root node with a listing of endpoint names under it. You can click on each endpoint to filter the **Messages** window to display only messages flowing through that endpoint, or you can click on the ServiceControl URL to show all messages again.

If you go the **Tools** menu and select **Connect to ServiceControl**, you can select a different ServiceControl URL to connect to, for example, to switch from viewing development information on your own system, to a QA or Production system on a remote server. Remember, the all of the data comes from the REST API, so you only need to know the URL.

Messages

The **Messages** window panel shows a grid of recently received messages, with **MessageID**, **Message Type**, **Time Sent**, **Critical Time**, **Processing Time**, and **Delivery Time** for each.

The **Message Properties** window shows all the attributes of the message, separated by categories. Of particular note is the **Errors** category, which will contain the full exception trace for an `error` queue message. No need to dig through logfiles; it's all right here!

If you want, you can quickly filter messages using the **Search Messages** box just above the grid, which searches across pretty much any field.

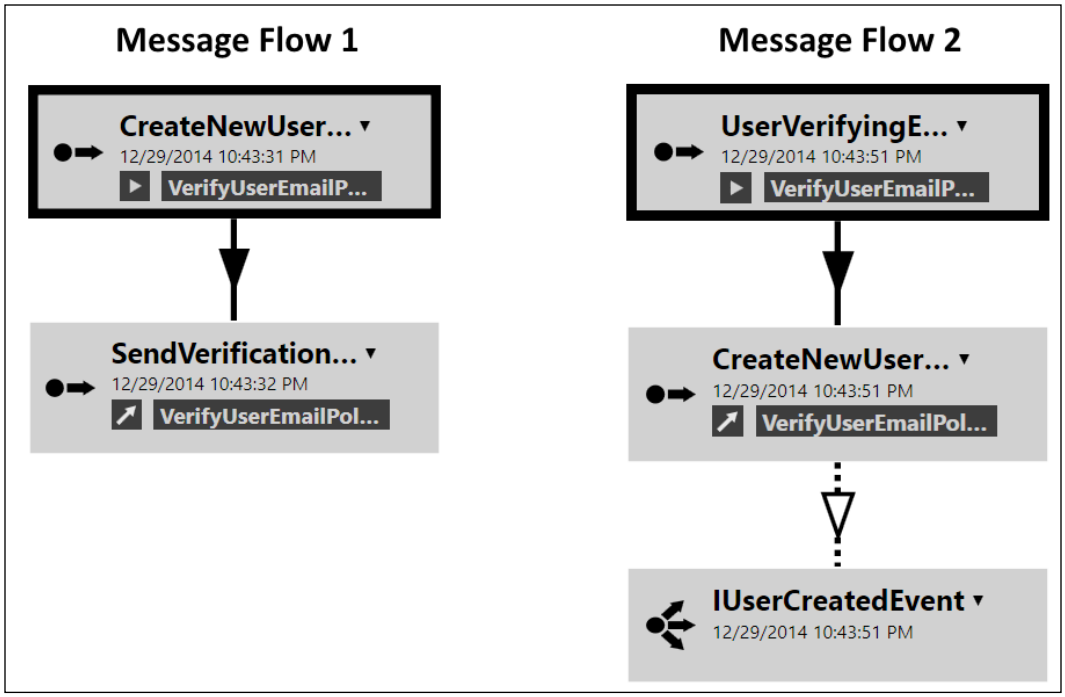
Main view

While this area does not have a label identifying it, ServiceInsight contains a main region that mostly displays visualizations for a specific message. It contains tabs for **Flow Diagram**, **Saga**, **Sequence Diagram**, **Headers**, **Body**, and **Logs**.

Flow Diagram

The **Flow Diagram** tab of the main area (shown as active in the preceding screenshot) shows all messages that have a cause-and-effect relationship with each other. Each box represents a message. Commands are shown with a single-arrow icon, while events are shown as a three-arrow fan-out icon. Additionally, the arrows between messages show a distinction between commands (solid line and arrowhead) and events (dotted line and open arrow). The **Show Endpoints** button can be toggled to show the endpoint that sends (above) and receives (below) each message.

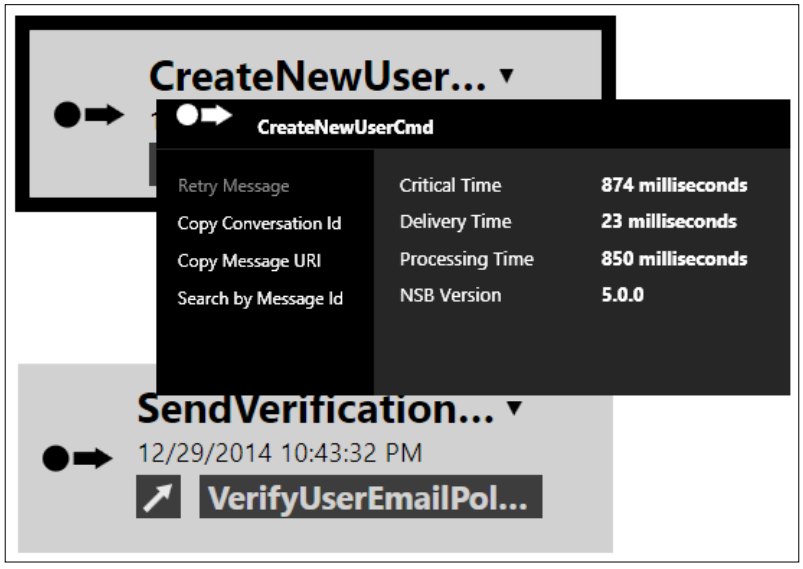
If we rerun the solution from *Chapter 6*, *Sagas*, wherein we introduced the saga, which included verifying the email address through the browser, we will see **5 message(s) found** in the **Messages** view, resulting in two message flows as shown here:



The first flow starts with our `CreateNewUserCmd` message, resulting in a `SendVerificationEmailCmd` message being sent. When NServiceBus sends a message while another is being processed, it passes along same value in a `ConversationId` header, allowing ServiceInsight to determine that they are related in the same message flow.

Next, a `UserVerifyingEmailCmd` message is sent as the result of our input in the MVC app, so this is the start of a new conversation, and thus, a new message flow. The result of this command is a `CreateNewUserCmd` message being sent, which results in an `IUserCreatedEvent` message being published.

When you click on a message's type label, a pop-up box will display some information on the right, but the really useful parts are the actions on the left.



Copy Conversation Id and **Copy Message URI** allow you to capture information about what you're looking at so that you can send it to another developer. The Message URI is especially interesting. ServiceInsight registers the `si://` URI scheme so that you can exchange a URI with others by email or chat. When they click on it on their system, it will launch ServiceInsight and instantly show them what you're seeing.

Search by Message Id will put the current message's ID into the search box so that you can hone in on messages directly related to the current message.

Retry Message, which would be available in case of an error, will return the message to its source queue. Ideally, you aren't seeing any errors right now, so go back to the code in *Chapter 6*, *Sagas*, throw an exception in one of the handlers, and then refresh ServiceInsight. You should see an exchange where the message throwing the exception is shown in red. Then fix the error and use ServiceInsight to retry the error. In moments, you can refresh and the message will have changed to a Successful state.

Saga

If you look at the flow diagram, just under the timestamps in most of the messages for the code from *Chapter 6*, *Sagas*, you will see inverted text identifying our `VerifyUserEmailPolicy` saga. An icon to the left of the saga name denotes whether the saga was initiated/updated by the message, originated from the saga (sent during a saga handler), or completed the saga.

If you click on the saga information, it will switch over to the **Saga** tab, where (unless you have skipped ahead) you will likely see nothing interesting. This is because ServiceInsight needs a bit more information about the saga to draw any fancy diagrams for you.

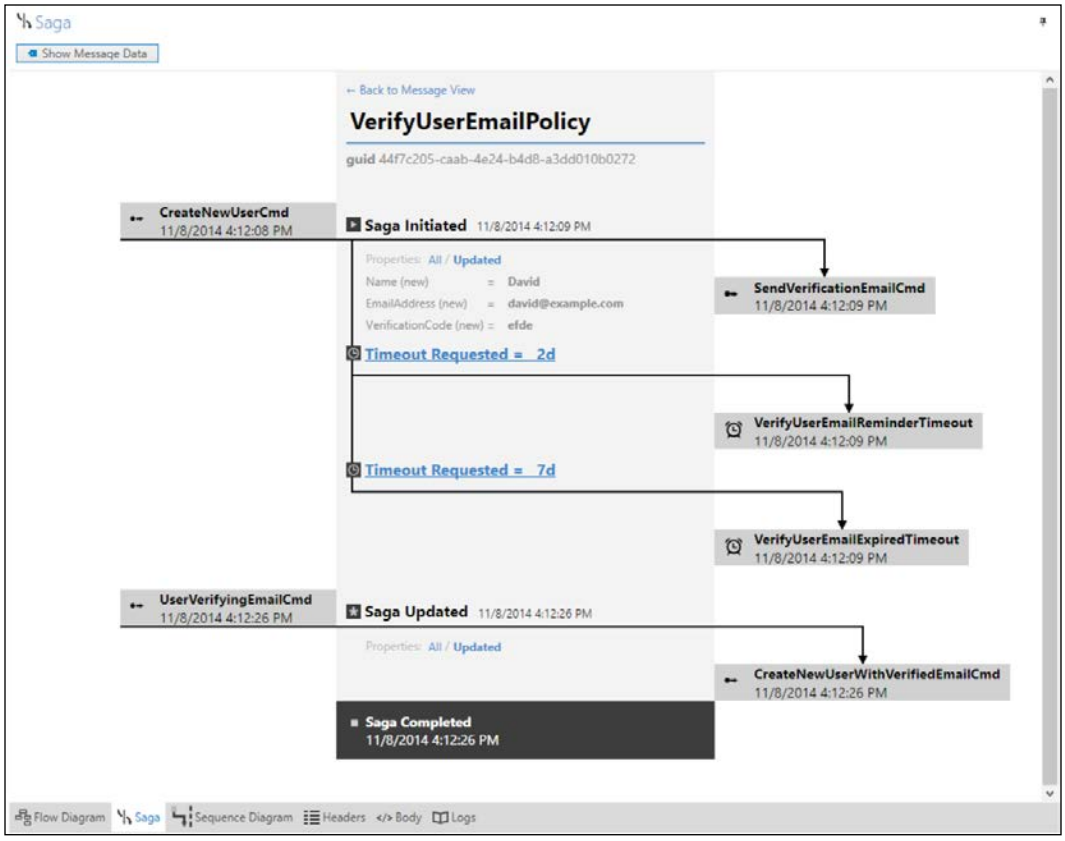
To fix this problem, we just add an additional NuGet package to any endpoint where we have sagas deployed:

```
PM>Install-Package ServiceControl.Plugin.Nsb5.SagaAudit
```



You are certainly welcome to do this yourself if you like, but if you're in a hurry, the code sample for this chapter is a copy of the code from *Chapter 6*, *Sagas*, with additional items added to demonstrate the capabilities of the Service Platform tools.

With the additional package installed, we can run through the procedure of creating and verifying a new user, and when we examine the **Saga** tab in ServiceInsight, we will see a flow diagram similar to what is shown in the following screenshot:

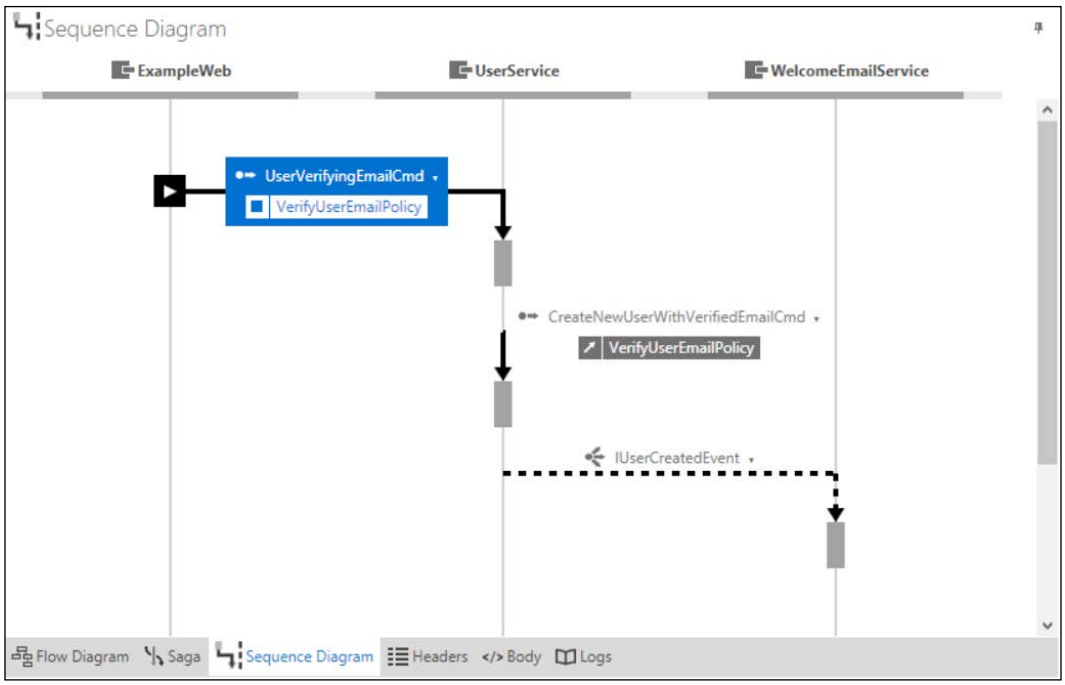


The diagram shows incoming messages on the left, and how they caused outgoing messages and timeouts on the right. It also shows any changes to the saga data caused by the message. In this case, we didn't wait around for a timeout, but in the case of a timeout, that will also be shown as an input on the left side. If the saga completes, we will see the date of completion at the bottom. At each stage, we see timestamps. If we click on the **Show Message Data** button, we will see the properties (which are not in the screenshot) of each incoming and outgoing message as well.

This visualization is a very powerful tool to help us understand a concept that is admittedly fairly abstract and instantly visualize how message flows affect each other, which helps us to verify that the result we got was the result we intended.

Sequence Diagram

A sequence diagram is a time-honored mechanism to visualize how different actors interact, and in what order. This is especially important for a messaging system, where the behavior of the system is largely defined by endpoints communicating with each other via messages.



The preceding diagram shows each endpoint involved in a conversation along the top axis, with time (not to scale) increasing as you travel downward. Messages sent from one endpoint to another travel horizontally between parallel endpoint lines, and messages sent by one endpoint and received by the same endpoint (as in the `CreateNewUserWithVerifiedEmailCmd` in the diagram) simply travel down that endpoint's timeline. Most messages are shown just by the message's type name with saga information (if available), but the message currently selected in the message list is highlighted in a blue box.

By clicking on any message, you can get the same information that you get from other views within ServiceInsight, or you can elect to retry messages that have failed. Additionally, clicking on any saga name will switch views over to the **Saga** tab as previously mentioned.

Message handlers are shown by gray bars along an endpoint's timeline, which will display critical time, delivery time, and processing time if you hover over them.

NServiceBus developers have been drawing sequence diagrams for years to describe and document messaging systems. They are a brand new addition to ServiceInsight, but the utility they offer in validating a system's design and assisting in debugging cannot be understated.

Other tabs

There are a few other tabs within the main region that share space with **Flow Diagram** and **Saga**. They don't really merit an entire section, but are definitely useful:

- The **Headers** tab displays all the message headers for the selected message with the exact name and raw formatting used in the message itself. This differs from the **Message Properties** window where much of the same data is displayed, but it is shown cleaned up, categorized, and fit for human understanding. If you need to copy and paste a raw value, this is the place to do it. The Headers tab is also the place to look for any custom header you have added to a message, which will not display in the **Message Properties** window.
- The **Body** tab will display the message as serialized to the transport. For example, if you use the XML Serializer, you will see XML in this view.
- The **Logs** tab shows the details of ServiceInsight communicating with ServiceControl. If you ever have a problem with ServiceInsight or ServiceControl, this information could be very helpful to send to Particular Support.

ServicePulse

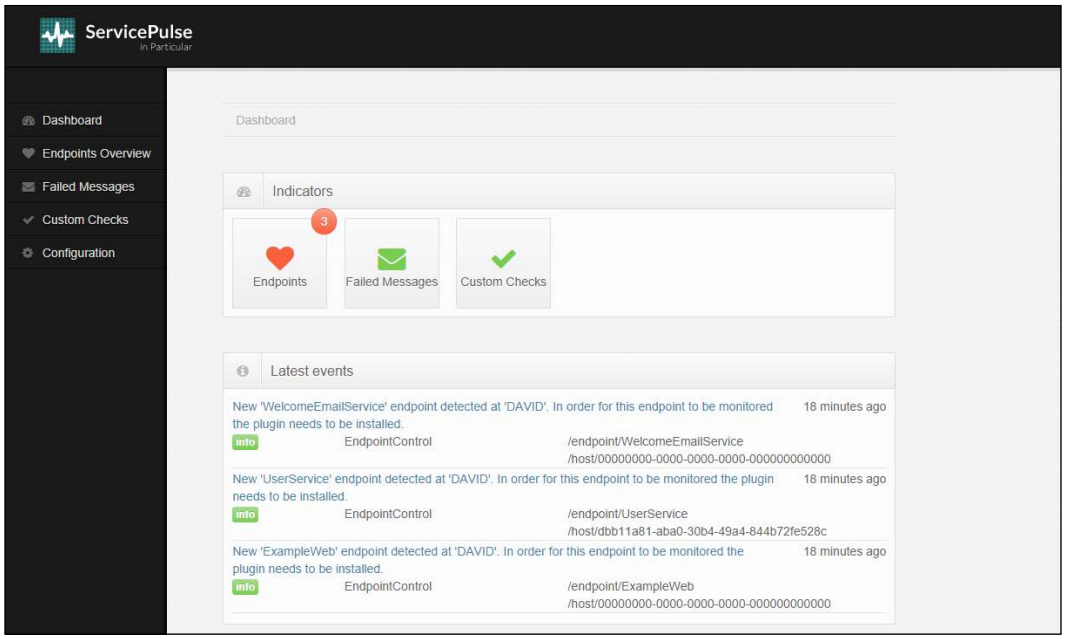
If ServiceControl is our brain, and ServiceInsight is our eyes, then **ServicePulse** is our ears, listening for things that go wrong. It's our nose too, as it lets us know when something doesn't smell quite right.



Of course, the metaphor about hearing and smelling is a bit misplaced because ServicePulse is just a web application that queries ServiceControl data. It's ServiceControl that knows these things, but it's ServicePulse that shows it to us.

ServicePulse is a web application that displays the health of our system. It allows us to see which endpoints are running and responsive, and which ones may have failed. It shows us messages that have failed and allows us to retry or archive those messages. It also allows us to configure custom checks to ensure that resources outside our system (that our system requires) are also healthy.

The default URL for ServicePulse is `http://localhost:9090`, but of course, just as with ServiceControl, this is configurable. If you launch ServicePulse, you will see the **Dashboard** view, which shows an overview of **Endpoints**, **Failed Messages**, and **Custom Checks**, along with a log of recent events.



Now let's dig into each of these three capabilities.

Endpoint activity

The first time you load ServicePulse, you may find that it's reporting your endpoints as inactive. Well, this is probably true; you are likely not running any of the demos from this book at the moment, so those endpoints are indeed down, but even if you do have an endpoint running, it may show up as unhealthy.

This is because the healthy state in ServicePulse depends upon heartbeat messages sent periodically from each endpoint to the central `audit` queue, but NServiceBus doesn't do this by default. To enable this behavior, we need to add a reference to a NuGet package:

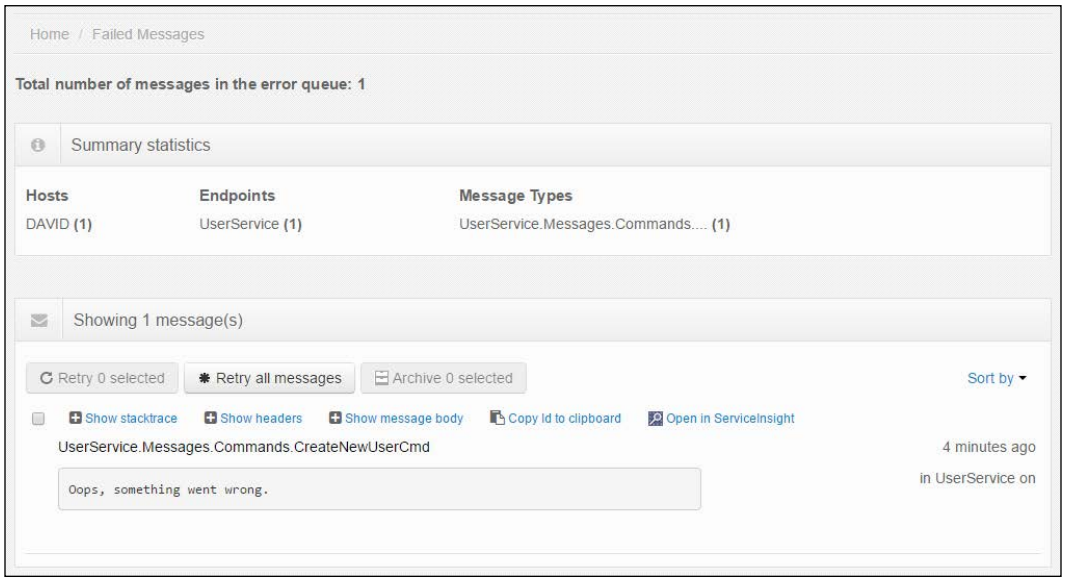
```
PM> Install-Package ServiceControl.Plugin.Nsb5.Heartbeat
```

With this package installed in our endpoint, heartbeats will be sent to the ServiceControl queue every 30 seconds, and if ServiceControl fails to receive that message, the endpoint will be considered as failed.

This turns out to be really useful because we can see at a glance that all our endpoints are running and responsive in a real way. We will see endpoints as failed if the server is down, if there is a network partition or if the Windows service is technically running (as far as the service control manager can tell) but isn't successfully processing messages for some reason.

Failed messages

Now that we know our endpoints are running and processing messages, we would like to know when any of those messages fail and go to the `error` queue. In ServiceInsight, we saw all messages, both successful and failed, but in ServicePulse we are focusing specifically on the failures. This makes ServicePulse the ideal tool for use by operations personnel.

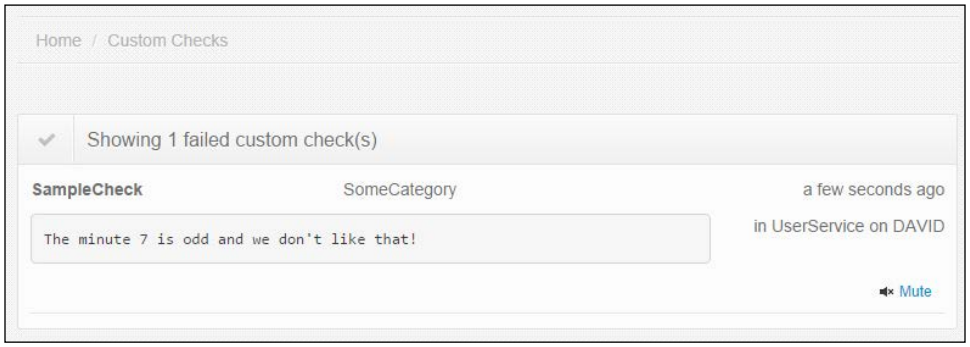


Below the summary is a list of all failures, with the ability to archive or retry a single message or batches of messages. Again, we can examine the full exception trace, and there is no need to dig through the logfiles.

Custom checks

Our NServiceBus system does not exist in a vacuum. It will depend upon other things to operate; for example, we might need an FTP server or web service to be up for our messages to be successfully processed. If a remote server goes down, we can find out about it before our messages start to fail. With NServiceBus and ServicePulse, we can create custom checks that run in alongside our message handling code written for custom checks. These custom checks will monitor our external resources for us and report the state through ServicePulse so that we will have a clearer view of all the things affecting our system at any given moment.

When a custom check enters a failed state, we see a notification in ServicePulse similar to what is shown in the following screenshot:



We can create a custom check directly within any endpoint. All we need to do is reference a NuGet package and create a class to perform the check.

First, we include the NuGet package within our endpoint:

```
PM> Install-Package ServiceControl.Plugin.Nsb5.CustomChecks
```

Once we've included the package, we have access to the `PeriodicCheck` class, which we can extend to create our own custom check:

```
public class CheckOnFtpServer : PeriodicCheck
{
  publicMyCustomCheck() :
    base("id", "category", TimeSpan.FromMinutes(1))
  {
  }

  public override CheckResultPerformCheck()
  {
    // Check the server. If everything is ok...
```

```
            returnCheckResult.Pass;
            // Otherwise...
            returnCheckResult.Failed("Failure reason");
        }
    }
```

We have to override the constructor, and when we do so, we provide an identifier, a category, and how often the check should be run. Within the constructor, we could also set up anything else required for the check, if necessary.

Within the `PerformCheck()` method, we perform the steps to check on our external resource and return `CheckResult` to indicate success or failure. In the case of failure, we can include a reason (for example, the message from an exception) so that we know a little more about the cause of the failure.

Custom checks are wired up automatically. Just create the class and NServiceBus will run them for you.



It's also possible to create a different kind of custom check where you control how and when the check is triggered, as opposed to creating it on a rigid schedule, by extending the `CustomCheck` class and calling the `ReportPass()` and `ReportFailed(reason)` methods. This is ideal to use when reporting the status of resources that have events, to report a change in their status, so that you can instantly report its status to ServicePulse without delay.

The sample code for this chapter includes a custom check called `SampleCustomCheck` within the `UserService` project, which runs every 15 seconds and reports failure whenever the minute of the current time is an odd number. You can run the project, switch to the **Custom Checks** section of ServicePulse, and then watch the warning appear or disappear as each minute passes.

Getting notified

The only thing that's left now is how to learn about a message failure without needing to hover in front of your computer and staring at ServicePulse all day and night.

Notifications are a very personal thing. Some organizations might want an email sent to a support account. Others might want to receive an SMS notification. Other possibilities include invoking some public API, or a HipChat notification, just to name a few. It's impossible to make everyone happy, so the Service Platform doesn't have any notification method built in. Instead, it gives you the tools to create a notification according to whatever business rules you see as suitable.

ServiceControl contains an assembly called `ServiceControl.Contracts`, which contains the `MessageFailed` event. You can create a separate NServiceBus endpoint that subscribes to this message and then notifies you according to your preference.

The sample code for this chapter includes a `FailureMonitor` project that contains the basic code to listen for and respond to message failures. There are a few important things to keep in mind:

- The `ServiceControl.Contracts` assembly can be included in your project most easily by installing the `ServiceControl.Contracts` NuGet package.
- ServiceControl uses JSON serialization for its internal messages, so your subscribing endpoint will need to use the JSON serializer even if the rest of your system uses XML.
- You will need to supply unobtrusive mode conventions to identify the ServiceControl events as events.
- Remember to register the subscription in your `App.config` file. The `MessageFailed` event is published from the `Particular.ServiceControl` endpoint.
- Within your notification handler, you can construct a URL that will open the message in ServiceInsight, which can be very helpful to quickly debug an issue.

Refer to the sample code to see all of this in action. From within the message handler, you have access to a wealth of information about the failed message—practically anything that would be available to you within ServiceInsight. Note that you can also create notifiers for the time when heartbeats stop or are restored, by subscribing to the `HeartbeatStopped` and `HeartbeatRestored` events.

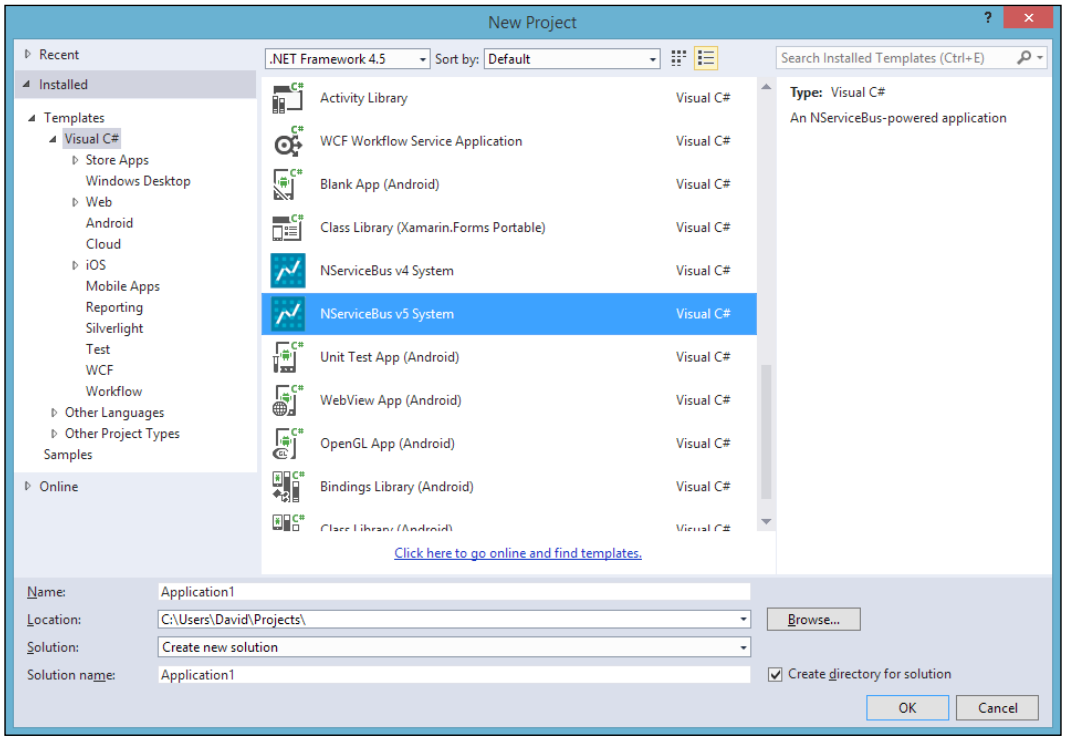
With this information and a little extra hand-rolled code, you'll be well on your way to emailing, texting, or chatting failure notifications in no time.

ServiceMatrix

The last piece of the Service Platform is our hands and feet (remember our human body metaphor?)—it allows us to build things and move quickly. In other words, ServiceMatrix is a graphical tool that helps us prototype and build NServiceBus systems quickly.

ServiceMatrix should have been installed in Visual Studio when you installed the Service Platform. If not, it can be installed separately from `http://particular.net/downloads` or directly within Visual Studio from the **Extensions and Updates** dialog in the **Tools** menu. Note that there are different versions of ServiceMatrix for Visual Studio 2012 and 2013, so make sure you install the correct one for your version.

To get started, open Visual Studio and create a new project, selecting the **NServiceBus v5 System** template under the **Visual C#** category.

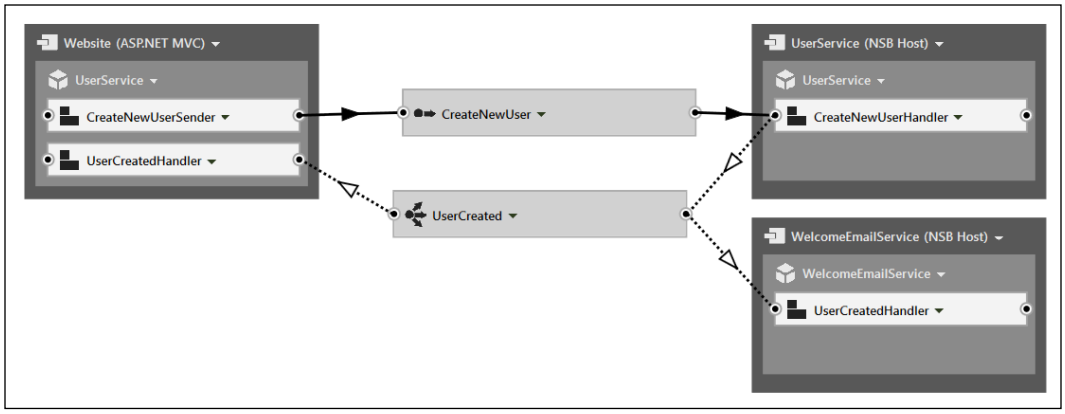




If you only have **NServiceBus System** available, it means you are using an old version of ServiceMatrix. Update the version through **Extensions and Updates** in the Visual Studio **Tools** menu.

After a lot of boilerplate is generated for you, you'll have a design surface inviting you to create a new endpoint, and a **Solution Builder** tool window, in addition to the familiar **Solution Explorer**. From the **Solution Builder**, you can manage the elements in your solution from a conceptual level of endpoints and services, rather than the physical level of assemblies and classes.

The following screenshot depicts the design of the `ServiceMatrixExample` project included with this chapter's sample code, which is a simple recreation of the new user example we've been using throughout this book:



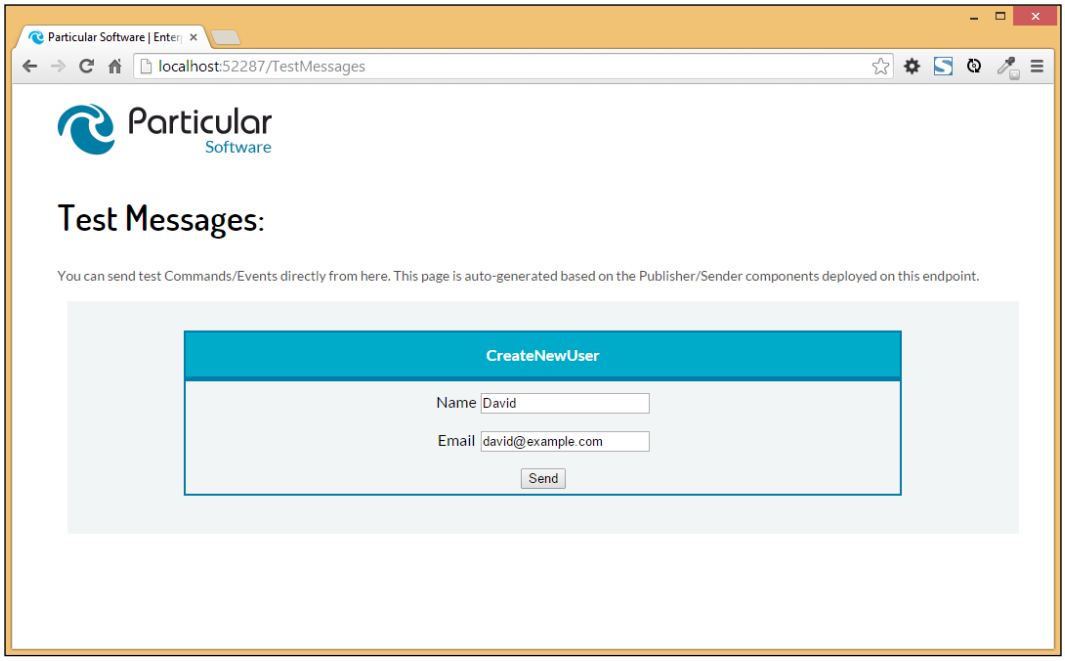
ServiceMatrix enforces the best practices of rigid service-oriented architecture. In doing so, it separates the concept of services from endpoints. A service is a logical boundary. An endpoint, on the other hand, is a physical unit of deployment, or in other words, an executable. Components from multiple services (multiple handlers) can be deployed to the same endpoint, and conversely, the components from one service can be split among multiple handlers. You can see in the screenshot how elements of the **UserService** service are deployed in the **Website** endpoint and in the **UserService** endpoint. Sometimes, perhaps often, services and endpoints can share the same name like **WelcomeEmailService**, which can either increase or decrease confusion, depending on your point of view.

When you launch a solution created with ServiceMatrix, several things will happen.

Firstly, any NServiceBus Host endpoint within the solution will be launched. ServiceMatrix will automatically add the NuGet packages for heartbeats and saga instrumentation so that the corresponding ServiceInsight and ServicePulse features will work without any specific action on your part.

Secondly, ServiceInsight will be automatically launched with the search field prepopulated to a debug session identifier, which all endpoints generated during the debug session will carry. This makes it easy to identify the results of your test run in ServiceInsight.

Thirdly, ServiceMatrix will create a web interface for any MVC website that allows you to test sending any messages through your system, complete with form fields to fill in any properties you add in the code.



This is quite a bit easier than using query string parameters to populate commands as we used in the previous chapters.

ServiceMatrix brings the platform full-circle. You can quickly design your system graphically, determine which messages are sent by which services, deploy the message handlers to the endpoints, test your message flows, and then evaluate your work in ServiceInsight. If you like, you can continue to develop your system in ServiceMatrix, or if you wish, you can disconnect ServiceMatrix from your solution and proceed with only the code at any point. However, after disconnecting, you cannot go back to ServiceMatrix because newly added items will not be in sync with ServiceMatrix any longer.

This book does not aim to be an exhaustive guide to ServiceMatrix. To do that topic justice, we might have to write another book. The tool is also changing and being improved upon so quickly that any attempt to document it completely on printed pages would be a fool's errand. The best way to get comfortable with the tool is to start experimenting with it. Of course, there is also documentation available at `http://docs.particular.net/servicematrix/`, which can be updated far more easily than this text.

Summary

With the Service Platform, Particular delivers on their promise to create the most developer-friendly service bus for .NET. Other packages might be cheaper, or even free, but you often get what you pay for, and you'll end up paying countless developer hours dealing with the other guy's shortcomings.

With the Particular Service Platform, ServiceControl is your brain, knowing everything about all the messages flowing through your system. ServiceInsight becomes your eyes, giving you the ability to see how messages interact and visualize message flows. ServicePulse constitutes your ears and nose, listening for trouble and letting you know if it smells like something is burning. And finally, ServiceMatrix serves as our hands and feet, allowing us to *move fast and break things* as Facebook CEO Mark Zuckerberg once said, except that I will leave breaking things as optional and up to you.

Now that we have a handle on all of the tools of the Service Platform, in the next chapter, we will cover how to administer an NServiceBus messaging system in production.

9
Administration

By now, 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 also learn how to monitor our production system and scale that system out as the load increases. Finally, we'll take a look at virtualization, an important component used to create truly reliable systems.

Service installation

Throughout this book, we've been testing our service endpoints by running them with `NServiceBus.Host.exe` as a console process. This is really convenient for development; when we deploy it to a different environment, we need stability in 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, allowing us to set the service name and description, and also to run the service under permissions other than the default service account. Not all of the parameters are required, but this set of parameters will cover most cases.

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

The `serviceName` is the name of the service used in the Windows Registry and in the `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 default values based on the endpoint name. If you recall from *Chapter 4*, *Hosting*, the endpoint name defines the pattern for the queues that are created for the endpoint as well. The fact that the queue and service names will match when this convention is used is very useful.

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` class.



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 these parameters are not supplied, the service will run under a local service account.

It is easy to uninstall a service, which you can do like this:

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

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

If you ever find yourself without a copy of this book, then you can quickly summon a refresher on the host process's command-line options using any of the following commands:

```
> 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 you launch the console window with elevated privileges first because you won't have the opportunity to do so while the install process is running.

Infrastructure installers

Whenever you deploy an endpoint, it's recommended that you uninstall and reinstall the endpoint to give infrastructure installers an opportunity to run. **Infrastructure installers** are components that make a modification to the system in order for the endpoint to run, such as the component that creates the endpoint's queues, and the component that initializes performance counters for the endpoint.

When we're developing locally, infrastructure installers run automatically when there is a debugger attached, or in other words, when we hit *F5* within Visual Studio. When we deploy to production, however, this is not the case. The infrastructure installers will require elevated privileges to create queues and performance counters, and then the endpoint will generally run as a less privileged user.



When using MSMQ, pay special attention to the privileges assigned to the queues. Mismatches in queue permissions (for example, if the infrastructure installers in a web application created the queues as the NetworkService user before configuring alternate credentials) can cause all sorts of problems when NServiceBus cannot read messages from the queue.

It's also possible to create your own infrastructure installers by creating a class that implements `INeedToInstallSomething`. This enables you to create your own necessary bits of infrastructure with elevated privileges at endpoint installation time:

```
public class CustomInstaller : INeedToInstallSomething
{
  public void Install(string identity, Configure config)
  {
    // You are probably running as administrator. Be nice!
  }
}
```

While NServiceBus host endpoints will run infrastructure installers when the endpoint is installed as a service, for a web endpoint, you must determine how to handle this capability yourself. You can either create queues manually as part of the deployment process (this can be accomplished with PowerShell for example) or you can include `cfg.EnableInstallers()` on your `BusConfiguration` instance to request that NServiceBus create them when the web application starts up.

Side-by-side installation

When deploying an endpoint update to production, it's very valuable to be able to deploy it side by side with the version it is replacing. This gives you a true zero-downtime upgrade, as the old endpoint will continue to run and process messages even as you are installing the new one.

To support this scenario, add the `-sideBySide` flag during service installation, which will result in the version (which, of course, you will need to manage) being appended to the service name (the code name used by the Windows Registry) as well as to the service description.

After installation, you will have two services (MyService 1.0.0 and MyService 2.0.0) installed and running, acting as competing consumers against the same input queue, and both actively processing messages.

Now that both are processing messages, you can actively monitor the endpoint logs and error queue. If you see anything fishy, you can disable the V2 endpoint, return any failed messages to the source queue (to be processed by the V1 endpoint) and look into the issue.

Once you are confident that the upgrade was successful, you can simply deactivate and uninstall the V1 endpoint.

Of course, in order to attempt a side-by-side installation, your code must be able to deal with new messages arriving at the older endpoint and being processed by that endpoint. Side-by-side deployments are best suited for situations where overall message interactions are the same but handler implementations have changed.

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. This is not so with NServiceBus. NServiceBus uses a concept called a **profile** to activate different options within an endpoint based on environment or capability.



Many of the dependency injection containers support a feature similar to profiles. If you are using a custom DI container, it may be a better idea to rely on your container's features for use with your own code. The exception where you must use NServiceBus profiles is when you are registering custom code to be used by NServiceBus itself.

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 installation options when installing an endpoint as a service. For instance, to run an endpoint with the `Lite` profile, run the following code:

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

All 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 can 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 previous versions of NServiceBus, environmental profiles controlled a lot more functionality, but as of NServiceBus 5.0, the differences are fairly slight and mostly still exist only for backward compatibility. By default, an NServiceBus endpoint will run in the Production profile, so many developers won't have a need for them, but if you choose to take advantage of them, they can help to switch features on and off for different phases of your development life cycle.

Out of the box, there are a few tiny differences between the environmental profiles:

- **Lite profile**: This will not forward failed messages to the error queue so that you don't have to deal with cleaning up many errors potentially generated during iterative development.
- **Integration profile**: This does not have any remarkable aspects, and it arguably only exists for backward compatibility with older versions of NServiceBus. However, it could be useful to install endpoints to staging environments with this profile, as you can write your own profile handlers (which we will see shortly), which could accomplish tasks such as switching to different persistence strategies.
- **Production profile**: This is the default and is just as unremarkable as the Integration profile, but it also inherits from the **PerformanceCounters** feature profile that we will learn about shortly, so all of its attributes are also invoked.

Feature profiles

Feature profiles help you to modify specific endpoint features programmatically:

- **PerformanceCounters profile**: This activates the performance counters for the endpoint. Because the Production profile inherits from this profile, the Production profile also enables performance counters by association. We will learn more about this in the *Monitoring* section.
- **MsmqMaster**, **MsmqDistributor**, and **MsmqWorker profiles**: These (available via a separate `NServiceBus.Distributor.Msmq` NuGet package) are used for scaling out an endpoint using the MSMQ transport. We'll discuss these profiles in detail in the *Scaling out* 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 placing them 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 what we learned in *Chapter 7*, *Advanced Configuration*, 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 the `IProfile` marker interface.

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 config)
  {
    // Leave empty, see the tip box below
  }

  public void ProfileActivated(BusConfiguration config)
  {
    config.RegisterComponents(reg =>
      reg.ConfigureComponent<TestAccountExpiration>(
      DependencyLifecycle.SingleInstance));
  }
}
```



Unfortunately, the transition from the static `Configuration` class in NServiceBus 4.x to the `BusConfiguration` instance in NServiceBus 5.0 makes the implementation of a profile handler a bit awkward in the short term. In the `IHandleProfile<T>` interface, the `ProfileActivated(Configure config)` method is marked as obsolete (and will be removed in NServiceBus 6.0), but as of now, it still exists, and to have a valid class that implements the interface, your profile handler must implement the obsolete method anyway. So for now, implement the method and leave it empty. NServiceBus will not call it. If you want, you can mark your method with `ObsoleteAttribute` as well.

If we want, we can also implement `IWantTheListOfActiveProfiles` within one of our profile handlers to get all the active profiles injected into our class. This is what the scale-out handlers for the MsmqMaster, MsmqWorker, and MsmqDistributor profiles use to ensure that 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; }

  // Obsolete method omitted
```

```
    public void ProfileActivated(BusConfiguration config)
    {
      if(ActiveProfiles.Contains(typeof(MsmqMaster))
      {
        // 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; }

  // Obsolete method omitted

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

We can even create additional profiles to handle tasks for us. For example, say a subset of our endpoints needed to have support for the Data Bus, as covered in *Chapter 5*, *Advanced Messaging*:

```
namespace Packt
{
  public class NeedsDataBus : IProfile { }
  public class DataBusSetup: IHandleProfile<NeedsDataBus>
  {
    public void ProfileActivated(BusConfiguration config)
```

```
        {
          // Set up the data bus here
        }
      }
    }
```

With this additional custom profile, we can easily configure the Data Bus on any endpoint that needs it by adding `Packt.NeedsDataBus` to the installation script.

NServiceBus will invoke all the profile handlers for the profile (or profiles) we specify on the command line and any base classes or interfaces as well. This means we can inherit a profile 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 `IConfigureThisEndpoint.Customize()` and `INeedInitialization.Customize()`. If multiple profiles are used on the command line, you should not assume that their handlers will execute in any specific order.

Logging profiles

Because the logging framework must be set up properly before the rest of our code runs, logging gets special treatment with respect to profiles. You can 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 to the `Configure` method. This is useful because we can determine the endpoint's identity and use that information to set up the logging. For instance, we may want to configure logfiles to be written to a **Universal Naming Convention** (**UNC**) path in a directory based on the endpoint name.



Unlike normal profiles, only one logging profile can be activated, as it won't make sense to configure logging more than once.

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 the 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 the least severe to the 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 our message queues on our local machine. When we deploy to new environments, we will need to deploy endpoints to multiple machines and 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 to transform a `Web.config` file when publishing a website, called XML Document Transform. Unfortunately, this process only supported `Web.config` files, which was a major oversight, leaving `App.config` files out in the cold. Luckily, time has passed and tooling has gotten better, so you don't need to let this limitation slow you down these days.

One fairly low-tech solution is the SlowCheetah plugin for Visual Studio. This extension allows you to transform `App.config` files as well as `Web.config` files right within Visual Studio, using the same transformation language. This can be utilized as a part of an automated build-and-deploy process.

A more comprehensive solution would be to use a full deployment automation system such as Octopus Deploy. In addition to automating deployments to all of your environments, Octopus Deploy natively supports configuration transforms. Deployment automation software is a great investment to make for any software system, but doubly so for distributed systems that are comprised of many processes that can be deployed independently.

Also, there may 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 TransportConfigGetConfiguration()
  {
    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.

You can even use `IProvideConfgiuration<T>` for your own custom configuration sections. Instead of using the `ConfigurationManager` class from the .NET Framework's `System.Configuration` to access config data, you can instead call `cfg.GetSettings().GetConfigSection<MyConfigSection>()` from any place where you have a `BusConfiguration` instance. This will first look for a registered configuration provider class, and if none is found, it will default to the values in the `App.config` or `Web.config` file.

Monitoring performance

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, your business stakeholders probably don't care what your overall throughput is; what they really care about is whether their work gets done within a length of time that they deem reasonable. Critical time gives you the ability to measure that.

If business stakeholders give you a hard-and-fast requirement for how fast messages must be processed, you can codify that in your 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. NServiceBus doesn't do anything by default when the SLA is breached, but the performance counter arms you with the data so that you can take a response that is appropriate to your business requirements.

We can define an endpoint's SLA by decorating the `EndpointConfig` class with `EndpointSLAAttribute`, specifying any string that can be converted to `TimeSpan`. Here, we define an example of an SLA of one minute:

```
[EndpointSLA("0:01:00")]
public class Endpoint : IConfigureThisEndpoint, AsA_Server
{
    // Body omitted
}
```

An alternate way to specify the Endpoint SLA is from a `BusConfiguration` instance inside `EndpointConfig`, `INeedInitialization`, and so on, which allows you to use the more friendly methods of specifying `TimeSpan`:

```
public void Customize(BusConfiguration cfg)
{
  cfg.EnableSLAPerformanceCounter(TimeSpan.FromMinutes(1));
}
```

Armed with the critical time and an SLA violation countdown, and given today's easy access to cloud resources, it's fairly simple 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 it 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.

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 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), we can allow NServiceBus to ramp up the number of threads that process 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 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 we set a fixed number of threads that are available to process messages, unlike a web server. If the load increases, 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:

```
<TransportConfigMaxRetries="5"
  MaximumConcurrencyLevel="3"
  MaximumMessageThroughputPerSecond="3" />
```

We will also take this opportunity to demonstrate another useful parameter of the `TransportConfig` element, `MaximumMessageThroughputPerSecond`. If omitted, this defaults 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, thus increasing total capacity. We can do the same thing with NServiceBus. The mechanism differs just slightly depending upon your chosen transport.

For broker-style transports (RabbitMQ, SQL Server, and Windows Azure), scaling out couldn't be simpler. All we have to do is install the same handler endpoint on additional machines, all connecting to the same input queue. This creates a **competing consumer pattern**, where multiple servers (in fact, multiple threads on multiple servers) compete to process messages from a single source.

MSMQ is a bit of a different beast in this respect. People commonly ask why they can't just use a network load balancer to scale out an MSMQ endpoint the same way they would scale a web server. Unfortunately, MSMQ's transactional message processing does not work with load balancers because the message acknowledgements can't be returned to the sending machine once the IP address of the sender is masked by the load balancer.

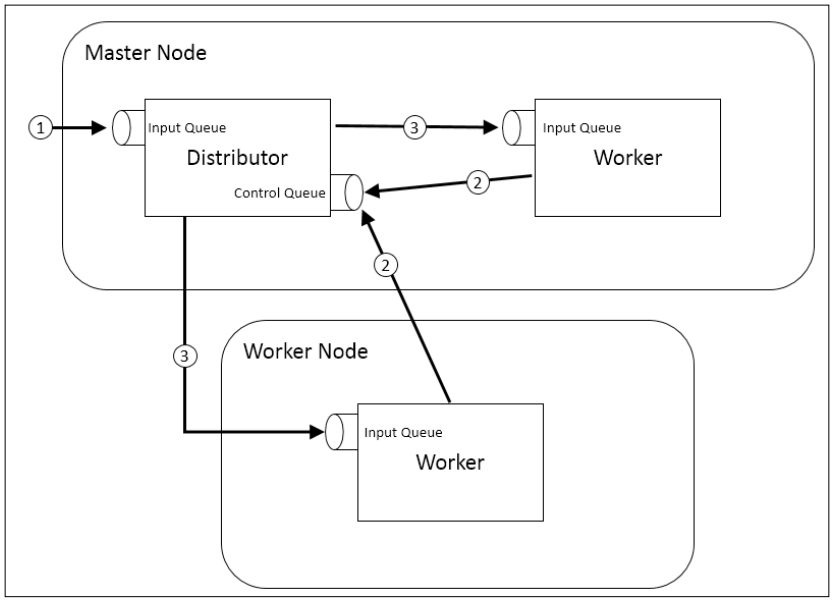
Instead, we can support scaling out an MSMQ-based NServiceBus endpoint using the `NServiceBus.Distributor.Msmq` NuGet package. When this package is included in an endpoint, that endpoint can be installed in one of three different modes:

- An endpoint installed as a **distributor** will monitor the main input queue and distribute incoming messages to registered workers who report that they are ready for work.
- An endpoint installed as a **worker** will maintain its own input queue and process messages arriving at that queue. When it has completed processing a message, it will request more work from the distributor.
- A **master node** combines the functionality of a distributor and worker in the same endpoint instance.

All of this is made possible with a few additional queues that NServiceBus creates automatically. The distributor maintains the following queues:

- **Input queue**: This is used to receive incoming messages
- **Control queue**: This is used to receive notifications that a worker is ready to process a message
- **Storage queue**: This is used to keep track of workers that are idle when no work is available

A master node and a worker node working together can be represented diagramatically as follows:



Incoming messages (**1**) arriving at the input queue of a **master node** will be intercepted by the distributor component running inside the process. The master node contains both the distributor component and a worker instance.

When a **worker** instance is ready to process a message, it will send `ReadyMessage` (**2**) to the distributor's control queue (the ready messages look like empty messages with additional control headers on the wire). If the distributor has a message waiting in its input queue, it will forward that message to the worker's private input queue (**3**) and the worker will process it. If the distributor has no remaining work, it will store the `ReadyMessage` in its storage queue (not pictured) 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 with the distributor NuGet package can operate as either a distributor or a worker by enabling the `NServiceBus.MsmqDistributor` or `NServiceBus.MsmqWorker` profiles on the command line when the service endpoint is installed.



These profile names have been changed from NServiceBus 4.x, where the distributor components were included in the core NServiceBus assembly and did not contain the `Msmq-` prefixes. The distributor components were removed from the core to prevent confusion because they are only necessary for the MSMQ transport.

A master node offers the ultimate scaling flexibility. You can deploy an endpoint as a master node to start out, and 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 can also generate the `MasterNodeConfig` section using the PowerShell `Add-NServiceBusMasterNodeConfig` cmdlet.

NServiceBus assumes 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 only one computer for educational value, you'll need 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 same package to every server that runs the endpoint. The code is the same and the configuration is the same for every server that runs the endpoint. 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 MSMQ worker

Because MSMQ worker nodes maintain their own input queue, just turning off a worker could result in messages remaining there, unable to be processed by other workers. Therefore, in order to decommission an MSMQ worker, some steps need to be taken to avoid losing data.

First, you will need the NServiceBus PowerShell cmdlets. These are available in the `NServiceBus.PowerShell` NuGet package. For full details on how to import these cmdlets to Visual Studio or to a full PowerShell console window, see `http://docs.particular.net/nservicebus/managing-nservicebus-using-powershell`.

Now that you have access to the cmdlets, let's assume that you are decommissioning a worker from the MyEndpoint endpoint on Server B, while the distributor for MyEndpoint lives on Server A. We can now execute this:

```
PS> Remove-NServiceBusMSMQWorker MyEndpoint@ServerB
  MyEndpoint@ServerA
```

The cmdlet takes care of figuring out the exact name of the distributor's control queue and sends it a message to deregister the worker. Inside the distributor, the Session ID for the worker in question is set to `disconnected`. Now, any incoming `ReadyMessage` the worker sends to the distributor will not match, and the worker won't receive any more work.

Once the worker chews through all the messages assigned to it and its input queue is empty, you can safely shut down and uninstall the endpoint. If you executed the cmdlet in an error, simply restart the endpoint. A new session ID will be generated and the distributor will assign the worker additional work as requested.

For broker-style transports, all the messages are stored centrally, so you can simply shut down any worker at will with no adverse side effects.

Extreme scale

Because MSMQ relies on the distributor to scale out, it does suffer from some limitations not present in a broker-style transport, where you can simply add more nodes at any point. The distributor presents a single chokepoint for the overall throughput, and messages can only be processed as fast as the distributor can dish out work to worker nodes. Generally, a distributor will max out at about 500 messages per second using the distributor included in NServiceBus 5.0.

Because an MSMQ system is decentralized, each node has its own maximum throughput. By contrast, a broker-based system will have a maximum throughput for the entire system, which is shared amongst all nodes connected to it.

In order to obtain extreme scaling with MSMQ, it is necessary to partition the incoming messages between multiple distributors, each of which can have multiple connected workers. This can be done by sending messages directly to a specific endpoint using `Bus.Send("Queue@SpecificServer", msg)`, using either a natural partition key (such as a client ID) or in a round-robin fashion. In this case, your application is taking ownership of the routing of messages, rather than the configured message endpoint mappings.

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 NServiceBus is concerned, a VPN connection makes geographically separated sites part of the same local network, and NServiceBus can operate more or less normally. Of course, some messages will have a little farther to travel than others, and the message transport's built-in capabilities will cover instances when the VPN connection is not always reliable.

The reality, however, is that a VPN connection is not always a possibility. If it is not available, the only method we can reliably use to communicate between sites and through firewalls is HTTP.

For this, NServiceBus provides the **gateway** component in the `NServiceBus.Gateway` NuGet package. 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.

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 that 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, and so on) for these scenarios instead.

To send a message over the gateway, we use 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 at `http://docs.particular.net/nservicebus/the-gateway-and-multi-site-distribution`.

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 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, driver problems are the other main cause of catastrophic system failures. 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 take a complete backup of the virtual machine before applying the update, and restore it to its original state in case of a failure.

MSMQ message storage

If you are using MSMQ as your transport, then MSMQ stores its messages on the same hard drive as the host operating system by default, which can be problematic. Whether or not you decide to take advantage of hardware virtualization, it's a smart idea to change the storage location of your messages to a **Storage Area Network** (**SAN**) with its own built-in redundancy. If you do not virtualize, this network 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 the processing from 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 in 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 in the **Storage** tab.

Clustering

Because broker-based transports are centralized in nature, they represent a single point of failure. Therefore, any broker-style transport should be clustered in order to ensure they are available to your endpoints at all times. As you embrace NServiceBus, you will quickly notice your message broker becoming the heart of your business processes, so you need to have resources and personnel in place to maintain the infrastructure in good working order at all times.

For the SQL Server transport, this means ensuring that your SQL Server instance is deployed on a failover cluster that is well-monitored and maintained by a team of DBAs familiar with SQL failover clustering. For RabbitMQ, this means establishing a RabbitMQ cluster with highly available queues. It's important to note that queues in RabbitMQ are located only on a single node in a cluster by default.

Of course, when using the Azure transport, Microsoft will take care of your concerns about high availability for you.

As a bus-style transport, MSMQ is not centralized in nature, so a failure of any one node only affects the messages traveling through that node as long as it is down, leaving the rest of the system free to operate more or less normally. However, you may need to make selected nodes highly available, perhaps to ensure that SLAs are met, on nodes where the requirement for high availability outweighs the increased complexity.

It is possible to deploy a Windows service powered by the NServiceBus Host as a clustered resource on a Windows Failover Cluster. In this case, you also need a clustered MSMQ and DTC instance. Generally, you will cluster only an endpoint deployed as a master node, or perhaps only as a distributor. The failover cluster ensures that messages continue to flow at all times, and separate servers containing only worker nodes provide redundant processing ability.

While clustering MSMQ endpoints, you will also want to ensure that all nodes can access the shared persistence for subscription storage, timeouts, sagas, and so on. If you are using NHibernate persistence, for example, with a centralized SQL Server database, then you don't have to worry (assuming your database is highly available). If you are using a more distributed persistence strategy, such as independent RavenDB databases on each node, then you will probably need to cluster the RavenDB instance for the highly available node as well.

Transport administration

Of course, administering NServiceBus also means administering the underlying message transport. NServiceBus does a pretty good job of freeing you from the nitty-gritty of your underlying transport, but it can't completely absolve you of dealing with it from time to time.

For the MSMQ transport, the NServiceBus community champion, *Daniel Marbach*, has compiled an exhaustive list of resources that are very helpful both in diagnosing potential problems and deepening your understanding of MSMQ's inner workings, at `http://www.planetgeek.ch/2014/09/02/administration-of-msmq/`.

For ActiveMQ, the best source of information is the official ActiveMQ documentation at `http://activemq.apache.org/`.

For RabbitMQ, the best source is the official RabbitMQ server documentation at `https://www.rabbitmq.com/admin-guide.html`.

Most developers electing to use SQL Server as a transport should view SQL Server itself as a fairly known quantity, with existing organizational assets in place to monitor and maintain it. The tables and infrastructure created by the SQL Server transport are easy to inspect in SQL Server Management Studio or to profile with SQL Server Profiler. Even so, you may want to take a look at the source code for the SQL Server transport at `https://github.com/Particular/NServiceBus.SqlServer` to gain an insight into what is being generated and executed as part of the NServiceBus integration.

Finally, for the Azure transports, you may want to explore the official Service Bus documentation at `http://azure.microsoft.com/en-us/documentation/services/service-bus/`, the official Azure Storage Queues documentation at `http://azure.microsoft.com/en-us/documentation/services/storage/`, and the NServiceBus Azure Transport documentation at `http://docs.particular.net/nservicebus/windows-azure-transport`.

Summary

In this chapter, we learned how to manage an NServiceBus system in a production environment, how to use the NServiceBus host's ability to install as a Windows Service, and how to 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 by 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 meet that SLA, we learned how to scale an endpoint. We also learned how to scale up by increasing the maximum concurrency level for an endpoint, and how to scale out using a competing consumer pattern for broker-style transports, or using the distributor component for MSMQ.

In order to bridge the gap between logically different and geographically separated sites, we learned about the gateway component and its capabilities. We discussed the importance of hardware virtualization in creating reliable infrastructure that we cannot attain with non-virtualized servers. Then we ended the chapter by briefly discussing clustering options for high availability and reviewing some sources for administration information for each of the supported message transports.

In the next chapter, we will review what we have learned in this book and cover resources where you can find more information about NServiceBus.

10
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 has aimed to be more of a guide to the essentials that will give you a running start so that you can create your own reliable distributed systems as quickly as possible.

As a result of this approach, there is more to learn, but let's pause for a moment to take stock of what we have covered in 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 from my own personal experience. Let's take a look back and reflect on how NServiceBus could prove useful in each situation.

In the first scenario, we were getting deadlocks when updating values in several database tables within a transaction. With NServiceBus, we would separate this action into many different commands, each of which would update the values in one table. By dividing the process, we would create less locking, minimizing the chance of deadlocks in the first place, and automated retry would ensure that the occasional deadlock didn't pose a huge problem. The database would be technically inconsistent for a short period of time until all the commands completed execution, but our business stakeholders would be very happy that end users are no longer getting error messages.

In the second scenario, we were losing orders and revenue because of a transient database error, such as a deadlock or even a database failover event. With NServiceBus, the database calls would be handled in a message handler, separate from the website code. During the times when the database threw an error, automatic retries would kick off until the command 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 of the work hadn't technically been done yet.

In the third scenario, an image processing system 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. 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 an SLA on the endpoint according to our business needs, and we would use the NServiceBus performance counters to ensure that we were meeting that SLA so that we would know when we needed to add more processing resources.

In the fourth scenario, we were integrating with a third-party web service and also updating a local database. 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, we send a new command to call the web service. When the web service times out, automatic retries ensure that the call is completed successfully.

In the fifth scenario, we were 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 the mail is sent only once. The part of the business process that is failing will benefit from automatic retries.

In the sixth scenario, a webpage began a long-running backend process, while the browser displayed an interstitial page similar to the page you see when you search for flights on a travel site. Because of unreliable messaging techniques, the integration was brittle and the backend process didn't always execute as planned. With NServiceBus, we ensure 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 real-time web socket library such as SignalR to determine the right time to advance to the next page.

In the seventh scenario, a nightly batch job designed to be run during off hours was taking so long that it was intruding on peak hours. With NServiceBus, we design sagas to react to events as they happen so that we don't need to run big batch jobs to update the whole database every night. As a bonus, our business processes become more real-time, which likely aligns much more closely with what our business stakeholders really want.

In the final scenario, we were spending way too much money purchasing on-premise infrastructure to handle traffic spikes that happened only rarely. We were looking for a way to transition business processes to the cloud so that we could dynamically provision additional infrastructure as needed and at much lower cost. With NServiceBus, we can transfer some of our workload to the message handlers that we deploy on Windows Azure, where we can scale them up or down as needed.

Hopefully, this book has demonstrated how NServiceBus can ease these pains, helping you to create software systems that are more reliable, scalable, and maintainable. It might even make building those systems more fun. Investing in NServiceBus as a part of your business infrastructure will pay dividends for years to come.

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, at `http://docs.particular.net`.
- NServiceBus boasts a very active and user-friendly community. You can find them at the official Particular Software discussion group, at `https://groups.google.com/forum/#!forum/particularsoftware`.
- 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 (including me) who use 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 at `http://stackoverflow.com/tags/nservicebus`.
- Remember that while NServiceBus is not free, it is still open source software. If you like, you can dive deep into the source code by looking up the NServiceBus GitHub repository, at `https://github.com/Particular/NServiceBus`. There are many other repositories within the Particular organization as well, containing the sources to all the pieces of the Particular Service Platform puzzle.
- Lastly, be sure to check out the NServiceBus Champions (`http://particular.net/champions`), a worldwide group of NServiceBus community leaders. It is definitely worth your time to follow them on Twitter or read their personal blogs.

Index

A

ACID (Atomicity, Consistency, Isolation, and Durability) 24
ActiveMQ
URL 168
Advanced Message Queuing Protocol (AMQP) 57
applicative message mutator 114
assembly, NServiceBus
scanning 54
Azure
about 99
storage, URL 99
Azure Storage Queues documentation
URL 168

B

BASE (Basically Available, Soft state, Eventual consistency) 25
batch jobs 85
Begin() method 114
behaviors
building 117, 118
ordering 118-120
replacing 120
business stakeholders
retraining 97

C

CAP theorem (Consistency, Availability, and Partition tolerance) 25
clustering 167
commands
consistency, achieving with messaging 26, 27
eventual consistency 24, 25
versus events 23
competing consumer pattern 160
configurations
managing 156, 157
control queue 161
critical time 158
custom checks, ServicePulse 140, 141

D

DataBus 81, 82
deadlock 40
dependency injection 55, 111, 112
design, saga
business logic only 93, 94
business stakeholders, retraining 97
guidelines 93
messaging 96, 97
saga lifetime 95
saga patterns 95
Distributed Transaction Coordinator (DTC) 9, 57 122

E

endpoint activity, ServicePulse 138, 139
Endpoint Explorer, ServiceInsight 131
endpoint name, NServiceBus
selecting 54
environmental profiles
about 151
integration profile 152
lite profile 152

production profile 152
error queues and replay
about 42
automatic retries 40, 41
RetryDemo 44
second-level retries 42, 43
errors
replaying 42
events
about 28
as interfaces 74, 75
publishing 29-31
subscribing to 31-33
versus commands 23

F

Fallacies of Distributed Computing 24, 47
fault tolerance 38-40
feature profiles 152
MSMQMaster, MSMQDistributor, and MSMQWorker profiles 152
PerformanceCounters profile 152
First-Level Retries (FLR) 43
Flow Diagram, ServiceInsight 131-134

G

gateway
URL 165
GitHub repository
URL 173

H

handler order
specifying 75, 76
hosting
startup 66
hosting, types
about 51
NServiceBus-hosted endpoints 52
self-hosted endpoints 53

I

IConfigureThisEndpoint 108
idempotent 47
Identity Map pattern 76
INeedInitialization 109
in-memory persistence 60
input queue 161
installers 65, 66
integration profile 152
Internet Information Services (IIS) 65
IWantToRunWhenBusStartsAndStops 110

L

licensing
URL 19
lite profile 152
log level
customizing 156

M

MarkAsComplete() method 90
message
actions 76
assembly, creating 10, 11
auditing 46
deferring 77
expiry 45
forwarding 78
handler, creating 12, 13
handler order, specifying 75, 76
headers 78
large payloads, transporting 80-82
property encryption 79, 80
sending, from MVC application 14
sending 23
stopping 76
unobtrusive mode 70, 71
versioning 72
message mutator
applicative message mutator 114
transport message mutator 115
message routing 33-35
message serialization 63, 64
Messages, ServiceInsight 131
messages, ServicePulse
failed 139
message transport 55
message, versioning
about 72

events, as interfaces 74
polymorphic dispatch 73
Microsoft Message Queuing (MSMQ)
about 85, 57
Distributed Transaction Coordinator (DTC) 57
Model View Controller. *See* **MVC**
MSMQ
URL 168
MSMQMaster, MSMQDistributor, and MSMQWorker profiles 152
MSMQ message storage 166
MSMQ worker
decommissioning 163
multiple sites 164
MVC
about 5
website, creating 14-17
MVC application
message, sending from 14

N

NHibernate 99
NHibernate persistence 60, 61
NServiceBus
about 5-7
assembly, scanning 54
champions, URL 173
code, retrieving 7-9
dependency injection 55
documentation, URL 173
endpoint name, selecting 54
example 10
extending 108
fault tolerance 38-40
hosting 51
IConfigureThisEndpoint 108
INeedInitialization 109
IWantToRunWhenBusStartsAndStops 110
learnings 171-173
message assembly, creating 10, 11
message-based subscriptions 57
message handler, creating 12
message, sending from MVC application 14
message transport 55, 56
Microsoft Message Queuing (MSMQ) 57
MVC website, creating 14-17
Particular Software discussion group, URL 173
performances, monitoring 158, 159
persistence 59
RabbitMQ 57
service endpoint, creating 11, 12
solution, running 18-20
SQL Server 58
storage-based publishing 57
transport, need for 56
Windows Azure 59
NServiceBus 5.0
about 115
Outbox 122
URL 125
NServiceBus Azure Transport documentation
URL 168
NServiceBus command
and RPC request, differences 23
NServiceBus-hosted endpoints 52
NServiceBus NuGet packages
about 9
NServiceBus 9
NServiceBus.Host 9
NServiceBus.Testing 9
NServiceBus performance counters 9
NServiceBus PowerShell cmdlets
URL 163

O

Outbox
about 122
configuring 125
distributed transactions, life without 123, 124
DTC 101 122, 123
session, sharing 126

P

PerformanceCounters profile 152
performances
monitoring 158, 159
persistence, NServiceBus
about 59

in-memory persistence 60
NHibernate persistence 60, 61
Polyglot persistence 63
RavenDB persistence 62
Windows Azure persistence 63
persistence, saga
about 98
Azure 99
NHibernate 99
RavenDB 98
pipeline, NServiceBus
about 115, 116
behavior, building 117, 118
behavior, ordering 118-120
behavior, replacing 120
behaviors 121, 122
poison messages 40
Polyglot persistence 63
polymorphic dispatch 73
production profile 152
profiles
about 151
customizing 152-155
environmental profiles 151, 152
feature profiles 152
logging 155
property
encrypting 79, 80

Q

QueueExplorer 128
queues
control queue 161
input queue 161
purging, on startup 65
storage queue 161

R

RabbitMQ
about 57
URL 58, 168
RavenDB 98
RavenDB persistence 62
Remote Procedure Call (RPC) 23, 74
RetryDemo 44, 45
RPC request
and NServiceBus command, differences 23

S

saga
controller pattern 95
data, finding 88, 89
defining 86-88
design, guidelines 93
ending 89-91
implementation patterns, blog 96
lifetime 95
long-running processes 85
messaging 96, 97
observer pattern 96
patterns 95
persistence 98
time, dealing with 91-93
saga, ServiceInsight 134, 135
scalability
about 159
scaling out 160
scaling up 159, 160
scaling out
about 160-162
endpoint, installation modes 161
extreme scale 164
scheduled tasks 85
scheduling 104
second-level retries 42, 43
self-hosted endpoints 53
semi-transient errors 42
send-only endpoints 67
sequence diagram, ServiceInsight 136, 137
Service Bus documentation
URL 168
ServiceControl
about 128, 129
documentation, URL 129
URL 128
service endpoint
creating 11, 12
ServiceInsight
about 130
Endpoint Explorer 131
Main view 131

Messages 131
ServiceInsight, main view
about 131
Body tab 137
Flow Diagram 131-133
Headers tab 137
Logs tab 137
other tabs 137
saga 134, 135
sequence diagram 136, 137
service installation
about 147, 148
infrastructure installers 149
side-by-side installation 150
Service Level Agreement (SLA) 158
ServiceMatrix
about 142-145
URL 145
Service-oriented architecture (SOA) 5
ServicePulse
about 137, 138
custom checks 140, 141
endpoint activity 138, 139
failed messages 139
notifications, getting 141, 142
SLA Violation Countdown performance counter 158
SQL Server 58
SQL Server transport
URL 168
Stack Overflow
URL 173
storage queue 161

T

time, saga
dealing with 91-93
TimeToBeReceived attribute 71
transactional processing 38-40
transactions 65
transport
administration 168
transport message mutator 115

U

unit of work 113, 114
unit testing
about 100-103
events, as interfaces 103
Universal Naming Convention (UNC) 155
unobtrusive mode 69-71
User Account Control (UAC) 149

V

virtualization
about 166
clustering 167
MSMQ message storage 166

W

Web service
exposing 82-84
integration 46-49
WellKnownStep class 118
Windows Azure. *See* **Azure**
about 59
URL 59
Windows Azure persistence 63
Windows Communication Foundation (WCF) web service 82
Windows Management Instrumentation (WMI) 159
Windows Presentation Foundation (WPF) 51



Thank you for buying

Learning NServiceBus

Second Edition

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Enterprise

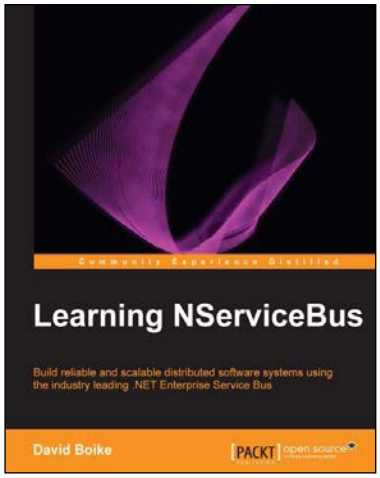
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `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, then please 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.

[PACKT] enterprise
professional expertise distilled
PUBLISHING

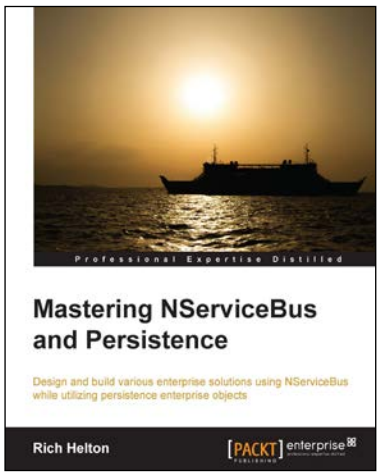


Learning NServiceBus

ISBN: 978-1-78216-634-4 Paperback: 136 pages

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

1. Replace batch jobs with a reliable process.
2. Create applications that compensate for system failure.
3. Build message-driven systems.



Mastering NServiceBus and Persistence

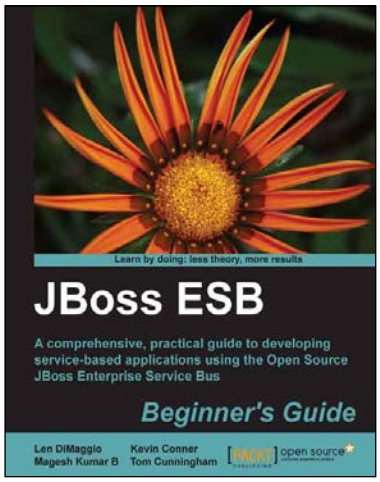
ISBN: 978-1-78217-381-6 Paperback: 286 pages

Design and build various enterprise solutions using NServiceBus while utilizing persistence enterprise objects

1. Learn how to utilize the robust features of NServiceBus to create, develop, and architect C# enterprise systems.
2. Customize NServiceBus to use persistent components to meet your business needs.
3. Explore the vast opportunities to extend NServiceBus for uses beyond basic enterprise systems using this practical tutorial.

Please check **www.PacktPub.com** for information on our titles

[PACKT] enterprise
professional expertise distilled
PUBLISHING

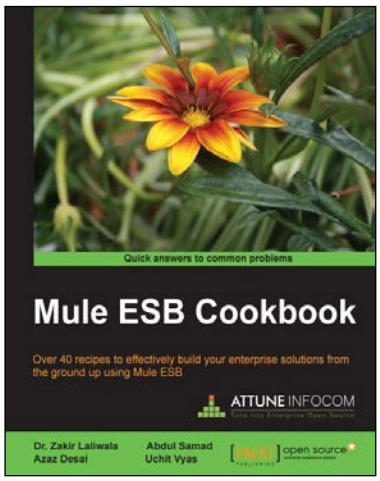


JBoss ESB Beginner's Guide

ISBN: 978-1-84951-658-7 Paperback: 320 pages

A comprehensive, practical guide to developing service-based applications using the Open Source JBoss Enterprise Service Bus

1. Develop your own service-based applications, from simple deployments through to complex legacy integrations.
2. Learn how services can communicate with each other and the benefits to be gained from loose coupling.
3. Contains clear, practical instructions for service development, highlighted through the use of numerous working examples.



Mule ESB Cookbook

ISBN: 978-1-78216-440-1 Paperback: 428 pages

Over 40 recipes to effectively build your enterprise solutions from the ground up using Mule ESB

1. Step-by-step practical recipes to get started with Mule ESB 3.4.
2. Learn to effectively use Mule ESB in a real-world scenario.
3. Expert advice on using filters, connecting with cloud, integrating with web services, and much more.