Learning NServiceBus
Second Edition

Building complex software systems is hard, but it doesn’t have to be! NServiceBus is a powerful, yet lightweight, open source messaging framework used to design distributed .NET enterprise systems. Entirely pluggable, yet simple to use, it can help you apply the principles of messaging and SOA to build robust, scalable, and maintainable systems.

This book is an essential guide that will help you become a messaging guru. From sending a simple message, to publishing events, to implementing complex long-running business processes, you’ll learn everything you need to know to start building complex software systems in no time.

Who this book is written for
If you are a .NET developer who wants to eliminate the problems related to defective third-party web service integration or batch job failures, then this is the book for you. It is also perfect for those of you who are new to NServiceBus and service-oriented architecture and would like to learn how you can streamline all of your development efforts.

What you will learn from this book
- Create systems that can be maintained and upgraded without downtime
- Make your web service integrations reliable
- Create code that automatically compensates for failures
- Apply the principles of messaging theory and eventual consistency
- Decouple and simplify business processes with Publish/Subscribe
- Replace batch jobs with business processes that are reliable, adaptable, and testable
- Create software that can scale horizontally as well as vertically
- Monitor your software for health, performance, and adherence to SLAs


Foreword by Udi Dahan, Founder and CEO, Particular Software
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Getting on the IBus”
- A synopsis of the book’s content
- More information on *Learning NServiceBus Second Edition*

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.
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.
- 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
Learning NServiceBus
Second Edition

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.
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.
Getting on the IBus

- 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*.
Getting on the IBus

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:

![Install the Particular Service Platform]

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:
   ```cmd
   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:

```csharp
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:

   ```csharp
   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 let’s finish our fake implementation for the handler:

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

   ```csharp
   private static readonly ILog log =
       LogManager.GetLogger(typeof(UserCreator));
   ```
2. To handle the command, remove `NotImplementedException` and replace it with the following statement:

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

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

```csharp
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](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](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:

```csharp
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();

            }
        }
    }
}
```

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.
Getting on the IBus

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:

```csharp
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:

```csharp
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);
}
Chapter 1

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:

```xml
<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
Getting on the IBus

- exampleweb.timeouts
- exampleweb.timeoutdispatcher
- userservice
- userservice.retries
- userservice.timeouts
- userservice.timeoutdispatcher

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.
Where to buy this book