NServiceBus Quick Start

This article is part of the **NServiceBus Learning Path (https://particular.net/learn/getting-started)**.

It is **very difficult** to build a distributed software system correctly from scratch. You *could* read all 736 pages of the Enterprise Integration Patterns book (an excellent though very dry reference) and then spend months creating, testing, and documenting a communication framework so that different services can talk to each other. Or instead, you could use a framework that incorporates all those design patterns and guides you straight into the pit of success.

NServiceBus combines decades of distributed systems design experience and expertise, and distills it into one easy-to-use framework. In this tutorial, you'll see how NServiceBus takes all the gruntwork out of system design by handling all of the plumbing for you, taking system design best practices like reliability, failure recovery, and extensibility and baking them right into the software, guiding you toward the pit of success.

You'll also see how the additional tools in the Particular Service Platform make it easy to manage, monitor, and debug.

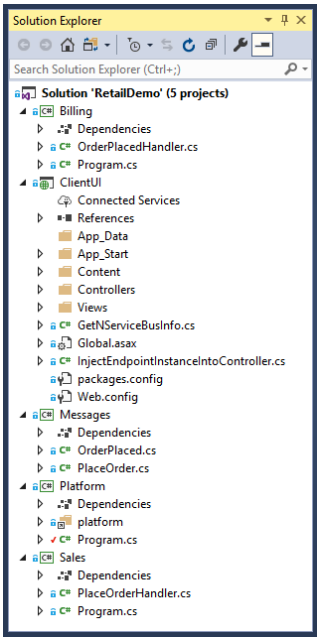
This tutorial skips over some concepts and implementation details in order to get up and running quickly. If you'd prefer to go more in-depth, check out our NServiceBus step-by-step tutorial (/tutorials/nservicebus-step-by-step/). It will teach you the NServiceBus API and important concepts necessary to learn how to build successful message-based software systems.

Download solution

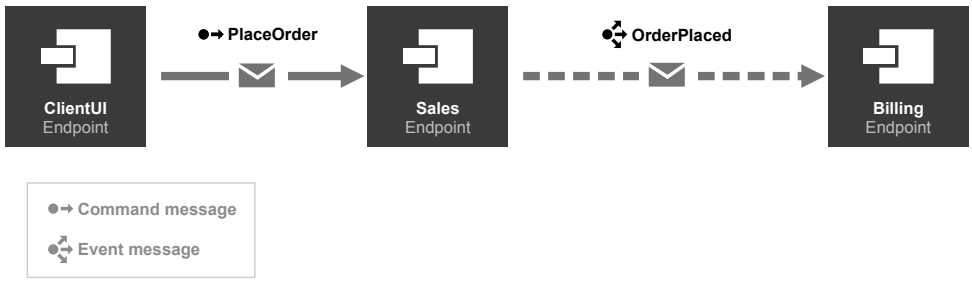
The demo solution doesn't require any prerequisites—no message queue or database to install, just Visual Studio. To get started, download the solution, extract the archive, and then open the **RetailDemo.sln** file.

Project structure

The solution contains five projects. The **ClientUI**, **Sales**, and **Billing** projects are endpoints (/nservicebus/endpoints/) that communicate with each other using NServiceBus messages. The **ClientUI** endpoint is implemented as a web application and is an entry point in our system. The **Sales** and **Billing** endpoints, implemented as console applications, contain business logic related to processing and fulfilling orders. Each endpoint references the **Messages** assembly, which contains the definitions of messages as simple class files. A little further into the tutorial, the **Platform** project will provide a demonstration of the Particular Service Platform, but at the beginning of the tutorial we'll leave its code commented out, and return to it later.



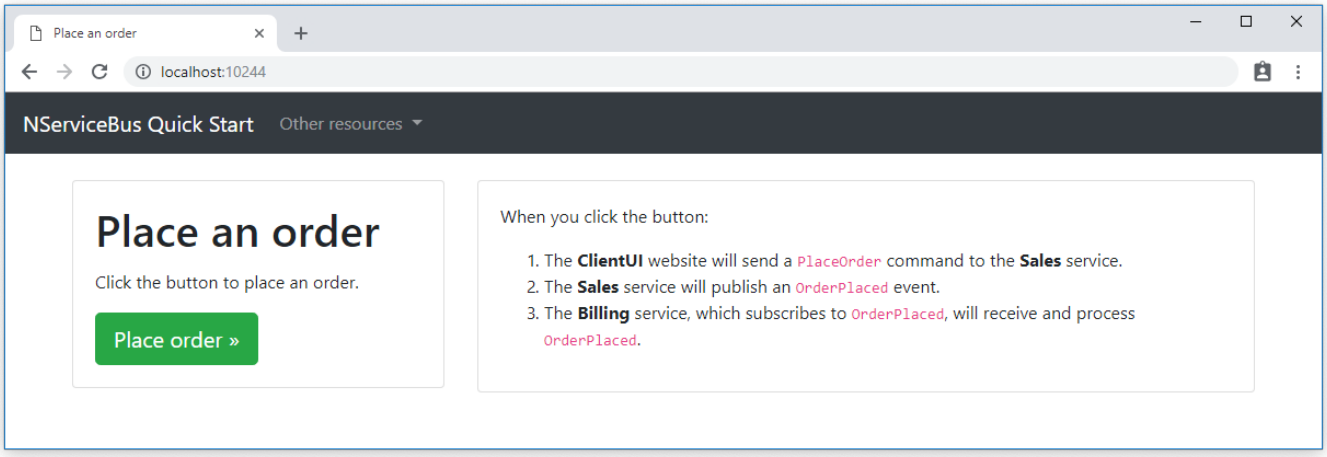
As shown in the diagram below, the **ClientUI** endpoint sends a **PlaceOrder** command to the **Sales** endpoint. As a result, the **Sales** endpoint will publish an **OrderPlaced** event using the publish/subscribe pattern, which will be received by the **Billing** endpoint.

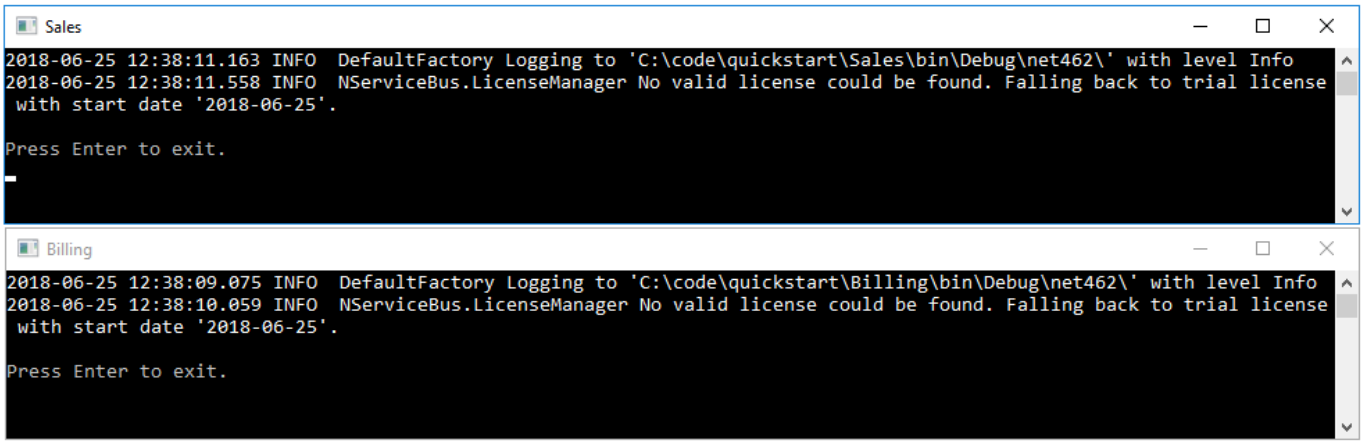


The solution mimics a real-life retail system, where the command (/nservicebus/messaging/messages-events-commands) to place an order is sent as a result of a customer interaction, and the processing occurs in the background. Publishing an event (/nservicebus/messaging/messages-events-commands) allows us to isolate the code to bill the credit card from the code to place the order, reducing coupling and making the system easier to maintain over the long term. Later in this tutorial, we'll see how to add a second subscriber in a new **Shipping** endpoint which would begin the process of shipping the order.

Running the solution

The solution is configured to have multiple startup projects, so when we run the solution (**Debug** > **Start Debugging** or press F5) it should open the web application in your browser, and two console applications, one window for each messaging endpoint. (The Particular Service Platform Launcher console app will also open. Depending on your version of Visual Studio, it may persist or immediately close.)





Did all three windows appear? In versions prior to Visual Studio 2019 16.1, there is a bug (**Link 1**, **Link 2**) that will sometimes prevent one or more projects from launching with an error message "Unable to launch the previously selected debugger. Please choose another." If this is the case, stop debugging and try again. The problem usually happens only on the first attempt.

In the **ClientUI** web application, click the **Place order** button to place an order, and watch what happens in other windows.

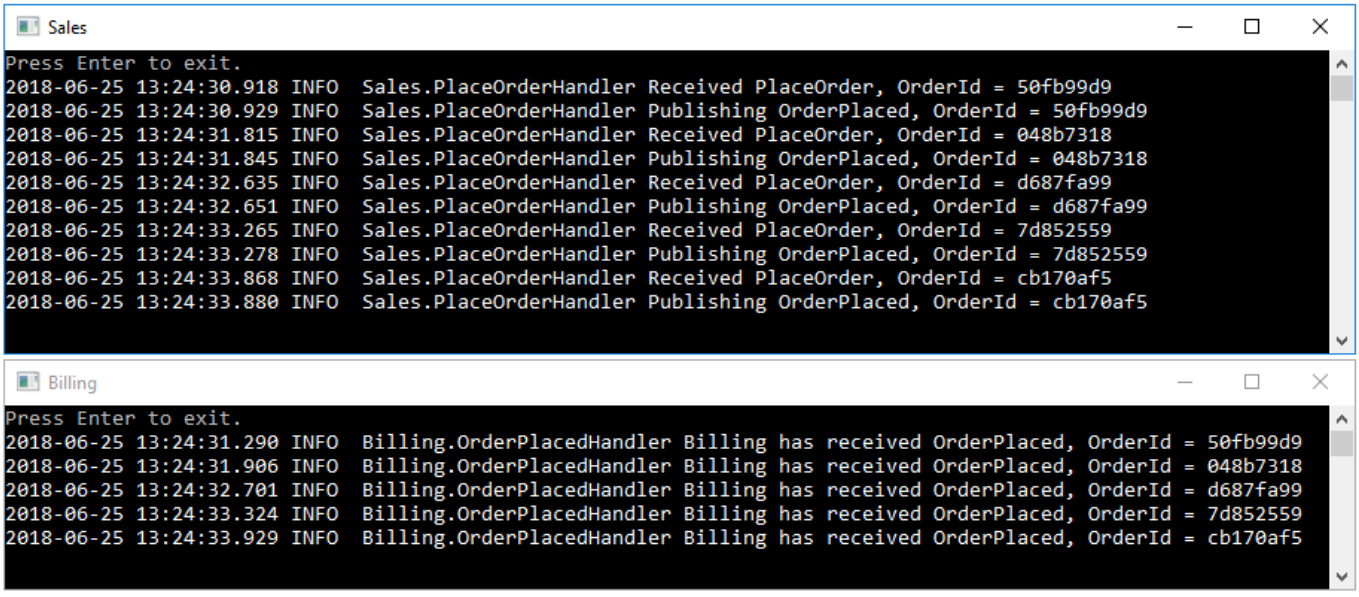
It may happen too quickly to see, but the **PlaceOrder** command will be sent to the **Sales** endpoint. In the **Sales** endpoint window we see:

```
INFO  Sales.PlaceOrderHandler Received PlaceOrder, OrderId = 9b16a5ce
INFO  Sales.PlaceOrderHandler Publishing OrderPlaced, OrderId = 9b16a5ce
```

As shown in the log, the **Sales** endpoint then publishes an **OrderPlaced** event, which will be received by the **Billing** endpoint. In the **Billing** endpoint window we see:

```
INFO  Billing.OrderPlacedHandler Billing has received OrderPlaced, OrderId = 9b16a5ce
```

In the **ClientUI** web application, go back and send more messages, watching the messages flow between endpoints.

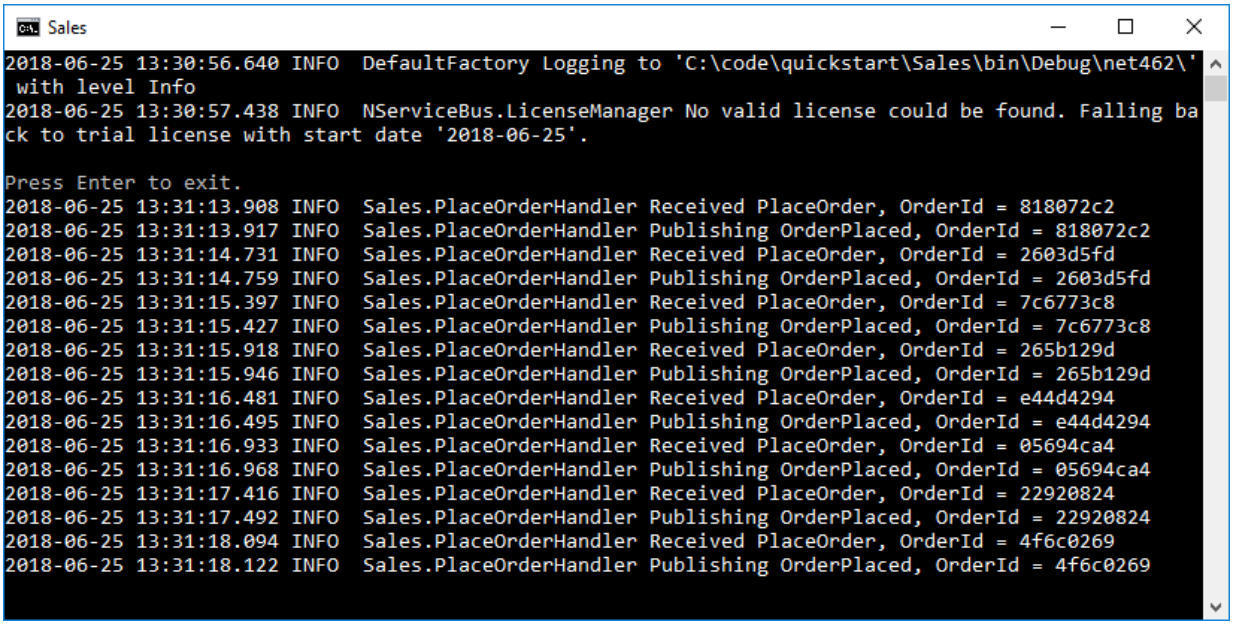


Reliability

One of the most powerful advantages of asynchronous messaging is reliability. Failures in one part of a system aren't propagated and don't bring the whole system down.

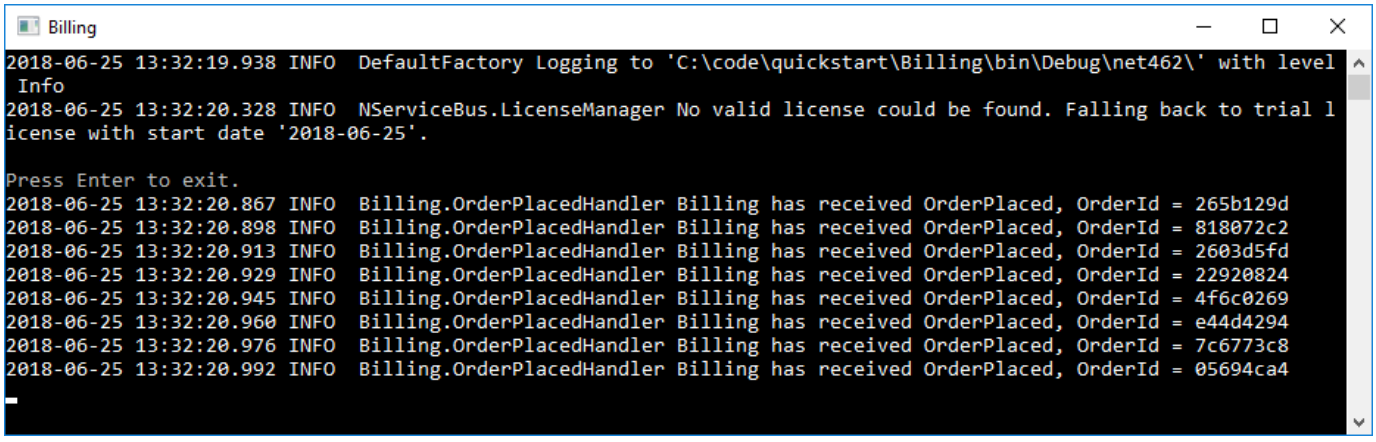
See how that is achieved by following these steps:

1. Stop the solution (if you haven't already) and then in Visual Studio's **Debug** menu, select **Start Without Debugging** or use `Ctrl`+`F5`. This will allow us to stop one endpoint without Visual Studio closing all three.
2. Close the **Billing** window.
3. Send several messages using the button in the **ClientUI** window.
4. Notice how messages are flowing from **ClientUI** to **Sales**. **Sales** is still publishing messages, even though **Billing** can't process them at the moment.



5. Restart the **Billing** application by right-clicking the **Billing** project in Visual Studio's Solution Explorer, then selecting **Debug** > **Start new instance**.

When the **Billing** endpoint starts, it will pick up messages published earlier by **Sales** and will complete the process for orders that were waiting to be billed.



Let's consider more carefully what happened. First, we had two processes communicating with each other with very little ceremony. The communication didn't break down even when the **Billing** service was unavailable. If we had implemented **Billing** as a REST endpoint, the **Sales** service would have thrown an HTTP exception when it was unable to communicate with it and *that request would have been lost*. By using NServiceBus we get a guarantee that even if message processing endpoints are temporarily unavailable, every message will eventually get delivered and processed.

Transient failures

Have you ever had business processes get interrupted by transient errors like database deadlocks? Transient errors often leave a system in an inconsistent state. For example, the order could be persisted in the database but not yet submitted to the payment processor. In such a situation you might have to investigate the database like a forensic analyst, trying to figure out where the process went wrong, and how to manually jump-start it so that the process can complete.

With NServiceBus we don't need manual intervention. If an exception is thrown, then the message handler will automatically retry processing it. That addresses transient failures like database deadlocks, connection issues across machines, file write access conflicts, etc.

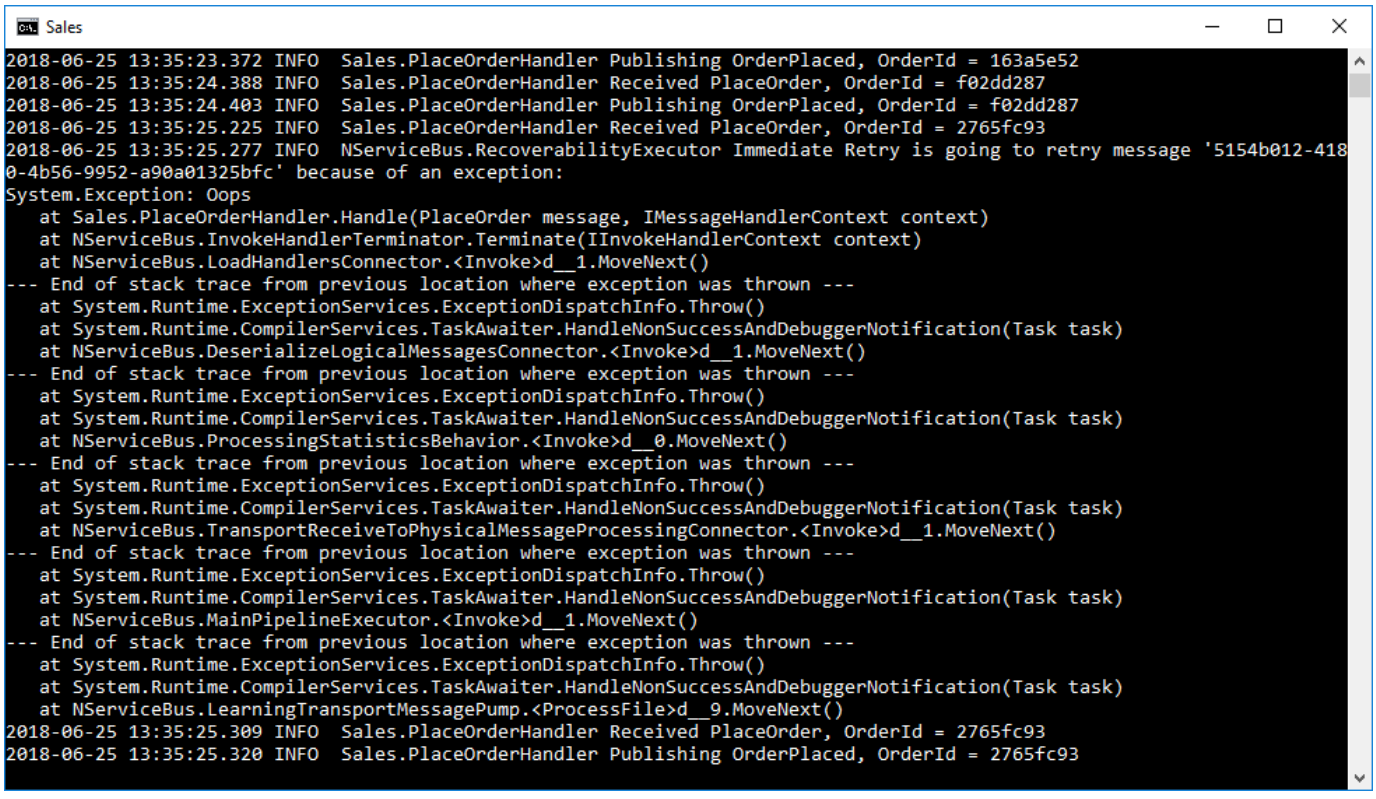
Let's simulate a transient failure in the **Sales** endpoint and see retries in action:

1. Stop the solution (if you haven't already) and in the **Sales** endpoint, locate and open the **PlaceOrderHandler.cs** file.
2. Uncomment the code inside the **ThrowTransientException** region shown here. This will cause an exception to be thrown 20% of the time a message is processed:

COPY CODE | COPY USINGS | EDIT
(https://github.com/Particular/docs.particular.net/edit/master/tutorials/quickstart/Solution/Sales/PlaceOrderHandler.cs#L22)

```
// Uncomment to test throwing transient exceptions
//if (random.Next(0, 5) == 0)
//{
//    throw new Exception("Oops");
//}
```

3. Start the solution without debugging (`Ctrl`+`F5`). This will make it easier to observe exceptions occurring without being interrupted by Visual Studio's Exception Assistant dialog.
4. In the **ClientUI** window, send one message at a time, and watch the **Sales** window.



As we can see in the **Sales** window, 80% of the messages will go through as normal, but when an exception occurs, the output will be different. The first attempt of `PlaceOrderHandler` will throw and log an exception, but then in the very next log entry, processing will be retried and likely succeed.

```
INFO  NServiceBus.RecoverabilityExecutor Immediate Retry is going to retry message '5154b012-4180-4b56-9952-a90a01325bfc' because of an exception:
System.Exception: Oops
    at <long stack trace>
INFO  Sales.PlaceOrderHandler Received PlaceOrder, OrderId = e1d86cb9
```

If you didn't detach the debugger, you must click the **Continue** button in the Exception Assistant dialog before the message will be printed in the **Sales** window.

5. Stop the solution and re-comment the code inside the **ThrowTransientException** region, so no exceptions are thrown in the future.

Automatic retries allow us to avoid losing data or having our system left in an inconsistent state because of a random transient exception. We won't need to manually dig through the database to fix things anymore!

Of course, there are other exceptions that may be harder to recover from than simple database deadlocks. Let's see what happens when a systemic failure occurs.

Systemic failures

In order to use the portable version of the Particular Service Platform included in this tutorial, you'll need to use a Windows operating system.

A systemic failure is one that is simply unrecoverable, no matter how many times we retry. Usually these are just plain old bugs. Most of the time these kinds of failures require a redeployment with new code in order to fix. But what happens to the messages when this happens?

For a good introduction to different types of errors and how to handle them with message-based systems, see **I caught an exception. Now what? (https://particular.net/blog/but-all-my-errors-are-severe)**

Let's cause a systemic failure and see how we can use the Particular Service Platform tools to handle it.

First, let's simulate a systemic failure in the **Sales** endpoint:

1. In the **Sales** endpoint, locate and open the **PlaceOrderHandler.cs** file.
2. Uncomment the code inside the **ThrowFatalException** region shown here. This will cause an exception to be thrown every time the `PlaceOrder` message is processed:

COPY CODE | COPY USINGS | EDIT
(https://github.com/Particular/docs.particular.net/edit/master/tutorials/quickstart/Solution/Sales/PlaceOrderHandler.cs#L30)

```
// Uncomment to test throwing fatal exceptions
//throw new Exception("BOOM");
```

3. In the `Handle` method, comment out all the code past the `throw` statement so that Visual Studio doesn't show a warning about unreachable code.

Next, let's enable the Particular Service Platform tools and see what they do.

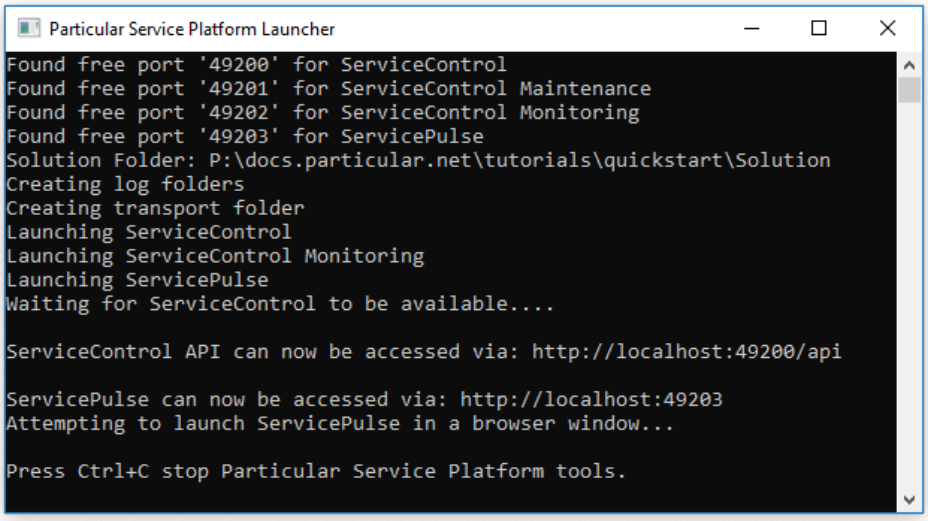
1. In the **Platform** project, locate and open the **Program.cs** file.
2. Uncomment the code inside the **PlatformMain** region shown here. This will cause the platform to launch when we start our project.

COPY CODE | COPY USINGS | EDIT
(https://github.com/Particular/docs.particular.net/edit/master/tutorials/quickstart/Solution/Platform/Program.cs#L8)

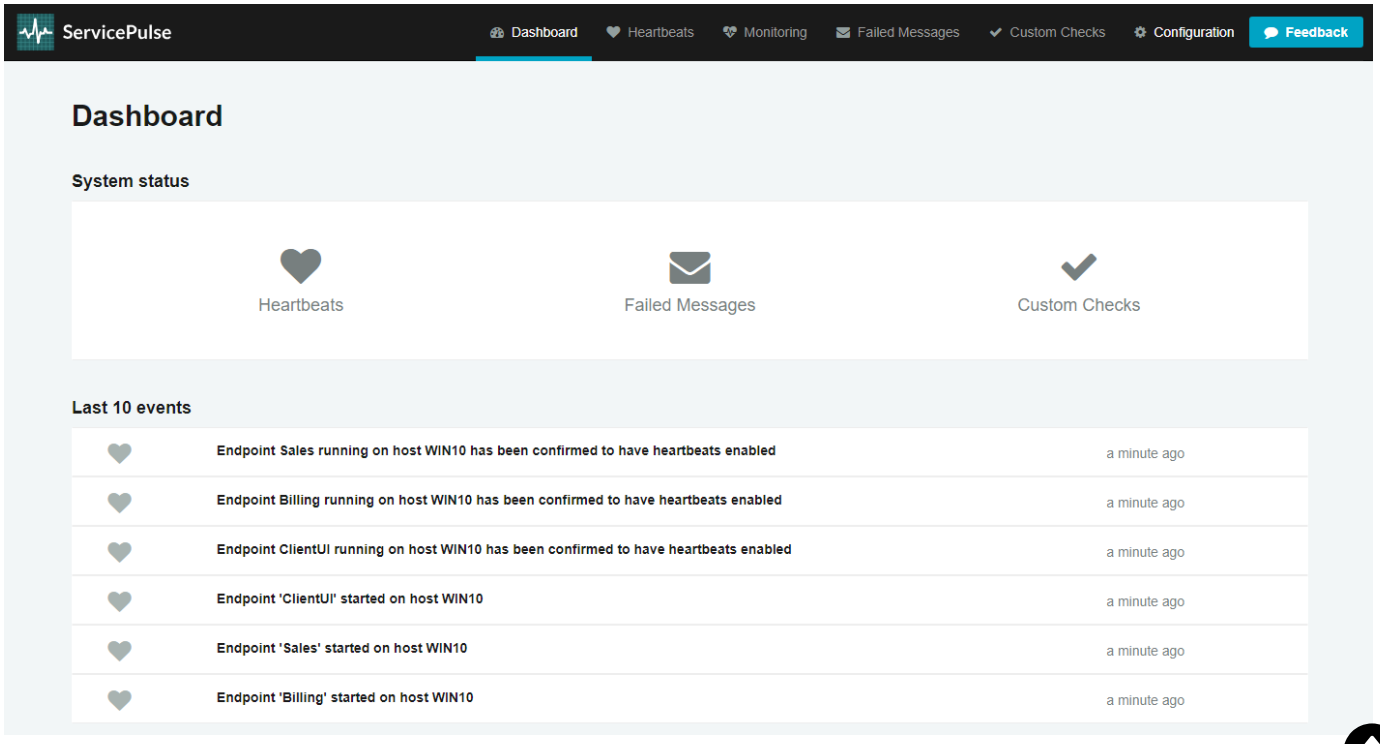
```
static void Main(string[] args)
{
    Console.Title = "Particular Service Platform Launcher";
    //Particular.PlatformLauncher.Launch();
}
```

With those two changes made, start the solution without debugging (`Ctrl`+`F5`). This will make it easier to observe the exceptions and retries without being interrupted by Visual Studio's Exception Assistant dialog.

Along with the windows from before, two new windows will now launch. The first is the **Particular Service Platform Launcher** window, which looks like this:

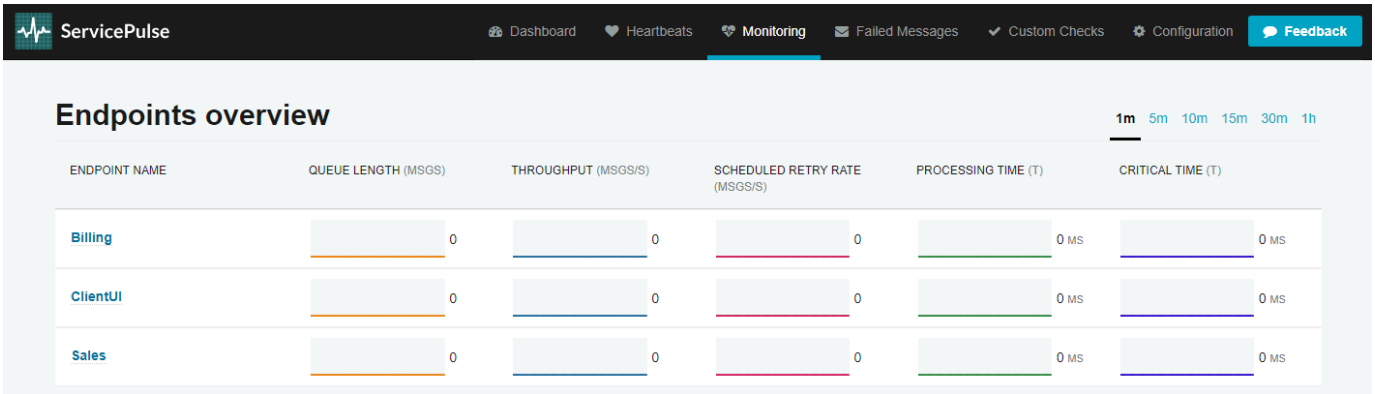


The purpose of this app is to host different tools within a sandbox environment, just for this solution. After a few seconds, the application launches ServicePulse in a new browser window:



The screenshot shows how ServicePulse monitors the operational health of your system. It tracks **Heartbeats** from your messaging endpoints, ensuring that they are running and able to send messages. It tracks **Failed Messages** and allows you to retry them. It also supports **Custom Checks** allowing you to write code that checks the health of your external dependencies (such as connectivity to a web service or FTP server) so you can get a better idea of the overall health of your system.

Another feature of ServicePulse is the **Monitoring** view, which tracks performance statistics for your endpoints:



For a more in-depth look at the monitoring capabilities, check out the Monitoring Demo (/tutorials/monitoring-demo/), which includes a load simulator to create monitoring graphs that aren't flatlined at zero.

For now, let's focus on the **Failed Messages** view. It's not much to look at right now (and that's good!) so let's generate a systemic failure:

1. Undock the ServicePulse browser tab into a new window to better see what's going on.
2. In the **ClientUI** window, send one message while watching the **Sales** window.

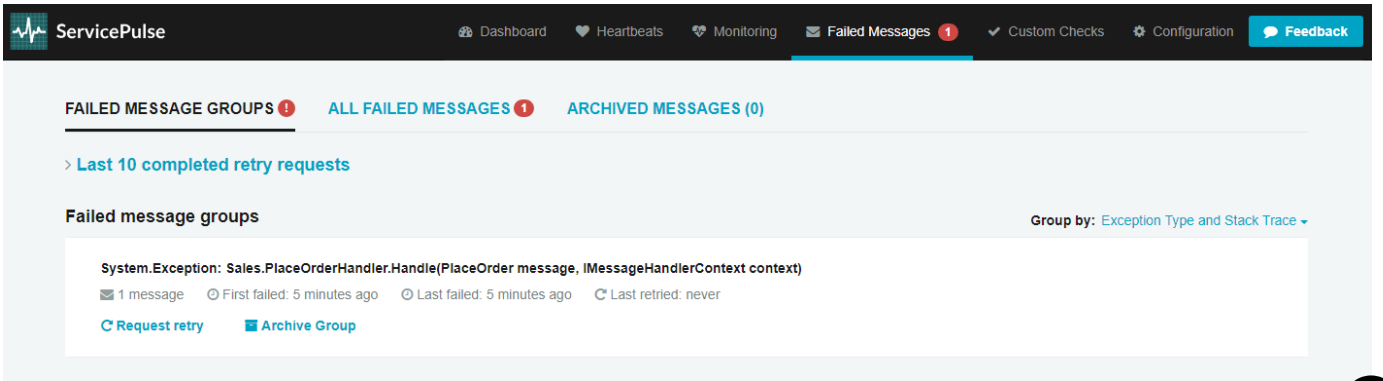
Immediately, we see an exception flash past, followed by an orange WARN message:

```
WARN  NServiceBus.RecoverabilityExecutor Delayed Retry will reschedule message 'ea962f05-7d82-4be1-926a-a9de0174976
7' after a delay of 00:00:10 because of an exception:
System.Exception: BOOM
   at <long stack trace>
```

Ten seconds later, text will flash past again, warning of a 20-second delay. Twenty seconds later, the text will flash again, warning of a 30-second delay. And finally, 30 seconds after that, text will flash by again, ending in red ERROR message:

```
ERROR NServiceBus.RecoverabilityExecutor Moving message 'ea962f05-7d82-4be1-926a-a9de01749767' to the error queue 'e
rror' because processing failed due to an exception:
System.Exception: BOOM
   at <long stack trace>
```

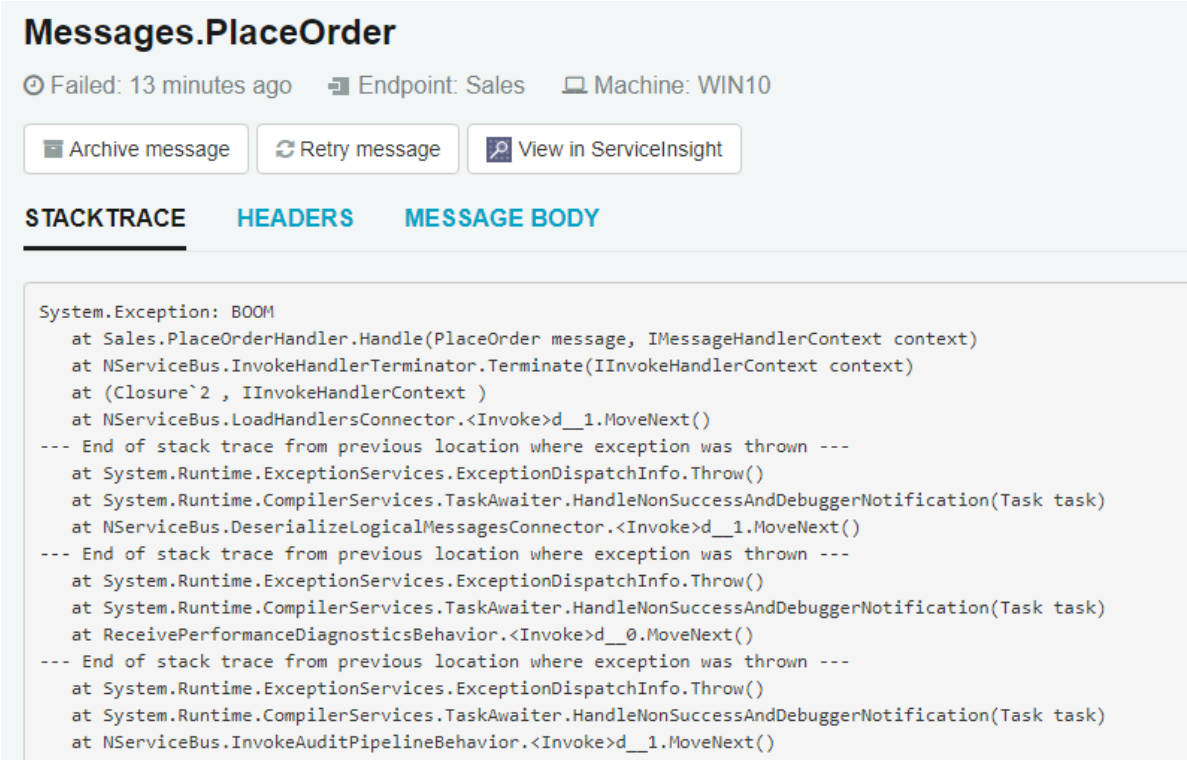
Once the red stack trace appears, check out the **Failed Messages** view in the **ServicePulse** window:



So what happened here? The message couldn't be successfully processed during an immediate round of retries, so it delayed the message for 10 seconds to try again. After that, it could still not be processed successfully, so it delayed the message for an additional 20 seconds, and then 30 seconds, before giving up all hope and transferring the message to an **error queue**, a holding location for poison messages so that other messages behind it can still get processed successfully.

Once the message enters the error queue, ServicePulse takes over, displaying all failed messages grouped by exception type and the location it's thrown from.

If you click on the exception group, it will take you to the list of exceptions within that group. This is not too interesting, since we currently only have one, but if you click again on the individual exception, you will get a rich exception detail view:



No need to go digging through log files to find out what went wrong. ServicePulse provides the exception's stack trace, message headers, and message body right here.

Armed with this information, it should be much easier to track down and fix our bug, so let's do that:

1. Close both browser windows and all console applications.
2. In the **Sales** endpoint, locate and open the **PlaceOrderHandler.cs** file.
3. Comment out the `throw` statement, and uncomment all the code below the **ThrowFatalException** region, returning the code to its original working state.
4. Start the solution again. It won't throw any exceptions so it's okay to attach the debugger this time.
5. Once the **ServicePulse** window launches, navigate to the **Failed Messages** view.

Now our system has been fixed, and we can give that failed message another chance.

1. Move the **Sales** and **Billing** windows around so you can see what happens when you retry the message.
2. In the **ServicePulse** window, click the **Request Retry** link.
3. In the confirmation dialog, click **Yes**, and watch the **Sales** and **Billing** windows.
4. It may take several seconds to enqueue the batch, but eventually you will see the familiar log messages in **Sales** and **Billing**, showing the message being processed successfully as if nothing bad ever happened.

This is a powerful feature. Many systemic failures are the result of bad deployments. A new version is rolled out with a bug, and errors suddenly start appearing that ultimately result in lost data.

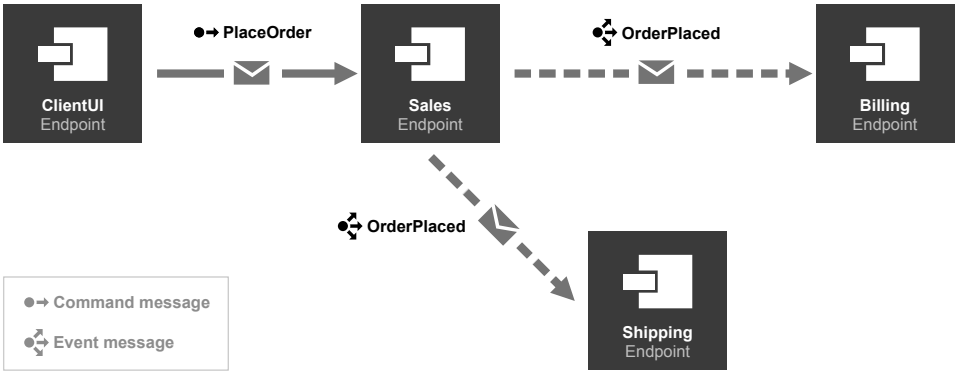
With a message-based system, no data is ever lost, because those failures result in messages being sent to an error queue, not lost to the ether. After a deployment, you can watch ServicePulse, and if messages start to pile up in the error queue, you can revert to the previous known good configuration while you diagnose the problem.

The visual tools in ServicePulse provide a quick way to get to the root cause of a problem and develop a fix. Once deployed, all affected messages (even into the thousands) can be replayed with just a few mouse clicks.

Extending the system

As mentioned previously, publishing events using the Publish-Subscribe pattern (/nservicebus/messaging/publish-subscribe/) reduces coupling and makes maintaining a system easier in the long run. Let's look at how we can add an additional subscriber without needing to modify any existing code.

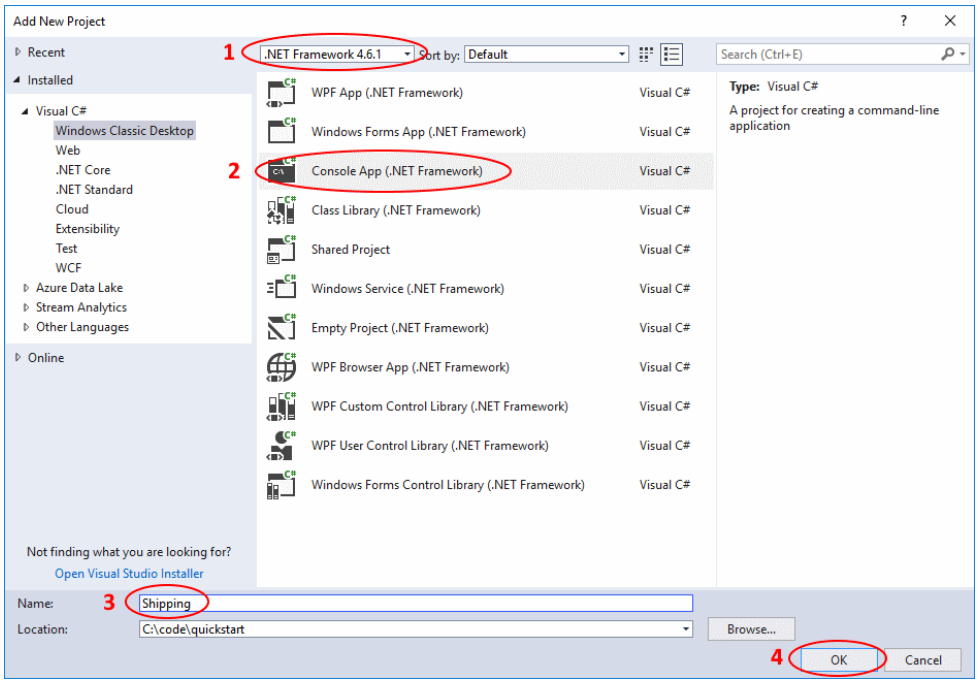
As shown in the diagram, we'll be adding a new messaging endpoint called **Shipping** that will also subscribe to the `OrderPlaced` event.



Create a new endpoint

First we'll create the **Shipping** project and set up its dependencies.

To start, in the **Solution Explorer** window, right-click the **RetailDemo** solution and select **Add** > **New Project**.



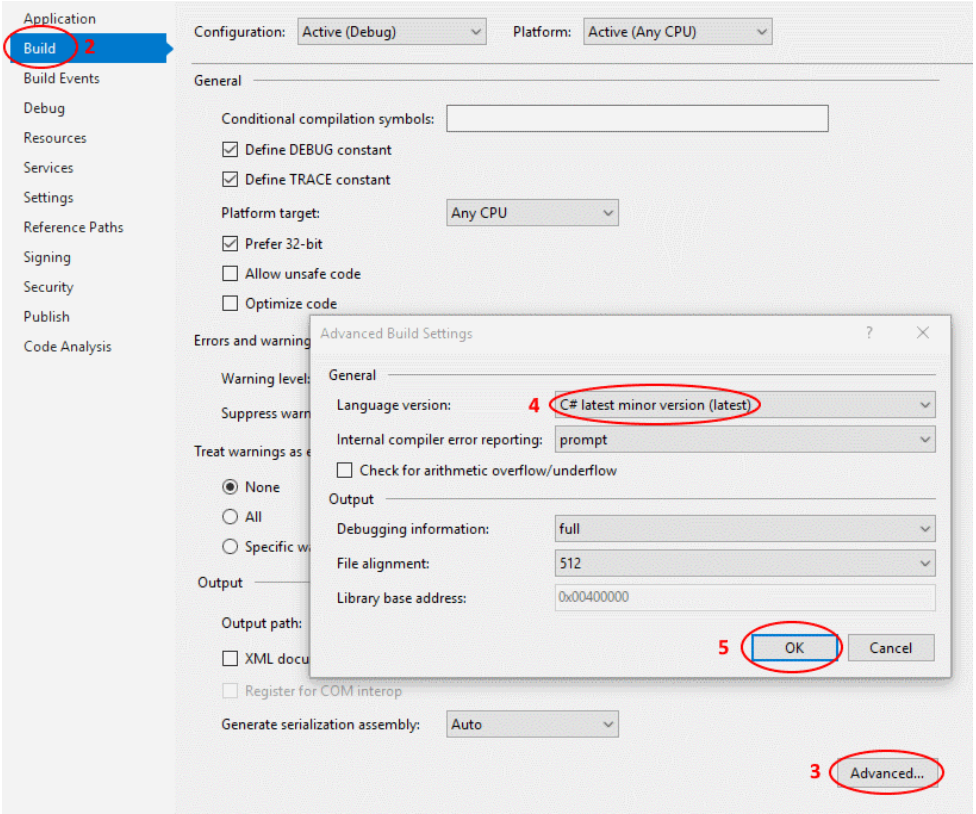
1. In the **Add New Project** dialog, be sure to select at least **.NET Framework 4.6.1** in the dropdown menu at the top of the window for access to the `Task.CompletedTask` API.

2. Select a new **Console App (.NET Framework)** project (or just **Console Application**).
3. Name the project **Shipping**.
4. Click **OK** to create the project and add it to the solution.

> **Tip:** The existing projects in this solution are using the simpler .NET Core-style project file syntax, but the current Visual Studio tooling makes it difficult to do the same for the **Shipping** project. If you'd like to use the newer format, create a project of type **Console App (.NET Core)** and then manually edit the **Shipping.csproj** file and change the `TargetFramework` value from `netcoreapp2.0` to `net461`.
>
> Creating a **Console App (.NET Framework)** project which uses the older `*.csproj` file syntax will work just fine, but will look slightly different in Visual Studio, with nested **Properties**, **References**, and **packages.config** items instead of **Dependencies**.

Depending on your environment, Visual Studio may create the project using C# 7.0. Let's change it to at least C# 7.1 so that we can use nice features like an async Main method:

1. In the **Solution Explorer**, right-click on the **Shipping** project and choose **Properties**.
2. Switch to the **Build** tab.
3. Under the **Output** heading, click the **Advanced…** button in the far lower-right corner.
4. Change **Language version** to **C# latest minor version (latest)**.
5. Click **OK**.
6. Save and close the **Shipping** properties page.



Now, we need to add references to the Messages project, as well as the NServiceBus, NServiceBus.Heartbeat, and NServiceBus.Metrics.ServiceControl packages

1. In the newly-created **Shipping** project, add the `NServiceBus`, `NServiceBus.Heartbeat`, and `NServiceBus.Metrics.ServiceControl` NuGet packages, which are already present in other projects in the solution. In the Package Manager Console window, enter the following commands:

```
Install-Package NServiceBus -ProjectName Shipping
Install-Package NServiceBus.Heartbeat -ProjectName Shipping
Install-Package NServiceBus.Metrics.ServiceControl -ProjectName Shipping
```

2. In the **Shipping** project, add a reference to the **Messages** project, so that we have access to the `OrderPlaced` event.

Now that we have a project for the Shipping endpoint, we need to add some code to configure and start an `NServiceBus` endpoint. In the **Shipping** project, find the auto-generated **Program.cs** file and replace its contents with:

COPY CODE | COPY USINGS | EDIT
(https://github.com/Particular/docs.particular.net/edit/master/tutorials/quickstart/Snippets/Core_7/ShippingProgram.cs#L2)

```
using System;
using System.Threading.Tasks;
using NServiceBus;

namespace Shipping
{
    class Program
    {
        static async Task Main()
        {
            Console.Title = "Shipping";

            // Define the endpoint name
            var endpointConfiguration = new EndpointConfiguration("Shipping");

            // Select the learning (filesystem-based) transport to communicate with other endpoints
            endpointConfiguration.UseTransport<LearningTransport>();

            // Enable monitoring errors, auditing, and heartbeats with the Particular Service Platform tools
            endpointConfiguration.SendFailedMessagesTo("error");
            endpointConfiguration.AuditProcessedMessagesTo("audit");
            endpointConfiguration.SendHeartbeatTo("Particular.ServiceControl");

            // Enable monitoring endpoint performance
            var metrics = endpointConfiguration.EnableMetrics();
            metrics.SendMetricDataToServiceControl("Particular.Monitoring", TimeSpan.FromMilliseconds(500));

            // Start the endpoint
            var endpointInstance = await Endpoint.Start(endpointConfiguration)
                .ConfigureAwait(false);

            Console.WriteLine("Press Enter to exit.");
            Console.ReadLine();

            await endpointInstance.Stop()
                .ConfigureAwait(false);
        }
    }
}
```

We want the **Shipping** endpoint to run when you debug the solution, so use Visual Studio's multiple startup projects feature to configure the **Shipping** endpoint to start along with **ClientUI**, **Sales**, and **Billing**.

Create a new message handler

Next, we need a message handler to process the `OrderPlaced` event. When NServiceBus starts, it will detect the message handler and handle subscribing to the event automatically.

To create the message handler:

1. In the **Shipping** project, create a new class named `OrderPlacedHandler`.
2. Mark the handler class as public, and implement the `IHandleMessages<OrderPlaced>` interface.
3. Add a logger instance, which will allow us to take advantage of the logging system used by NServiceBus. This has an important advantage over `Console.WriteLine()`: the entries written with the logger will appear in the log file in addition to the console. Use this code to add the logger instance to the handler class:

```
static ILog log = LogManager.GetLogger<OrderPlacedHandler>();
```

4. Within the `Handle` method, use the logger to record when the `OrderPlaced` message is received, including the value of the `OrderId` message property:

```
log.Info($"Shipping has received OrderPlaced, OrderId = {message.OrderId}");
```

5. Since everything we have done in this handler method is synchronous, return `Task.CompletedTask`.

When complete, the `OrderPlacedHandler` class should look like this:

COPY CODE | COPY USINGS | EDIT
(https://github.com/Particular/docs.particular.net/edit/master/tutorials/quickstart/Snippets/Core_7/OrderPlacedHandler.cs#L2)

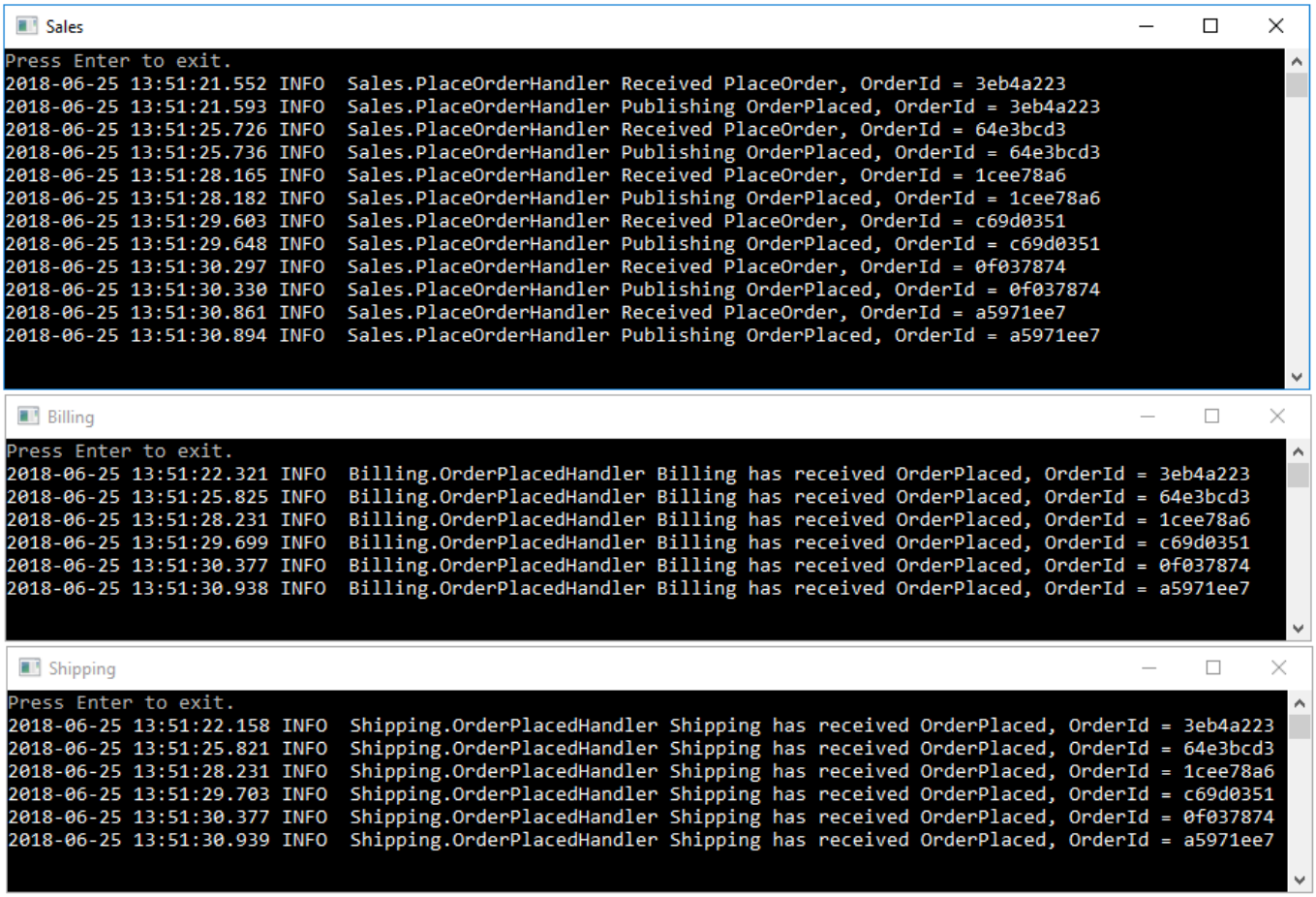
```
using System.Threading.Tasks;
using NServiceBus;
using NServiceBus.Logging;
using Messages;

namespace Shipping
{
    public class OrderPlacedHandler :
        IHandleMessages<OrderPlaced>
    {
        static ILog log = LogManager.GetLogger<OrderPlacedHandler>();

        public Task Handle(OrderPlaced message, IMessageHandlerContext context)
        {
            log.Info($"Shipping has received OrderPlaced, OrderId = {message.OrderId}");
            return Task.CompletedTask;
        }
    }
}
```

Run the updated solution

Now run the solution, and assuming you remembered to update the startup projects, a window for the **Shipping** endpoint will open in addition to the other two.



As you place orders by clicking the button in the **ClientUI** window, you will see the **Shipping** endpoint reacting to `OrderPlaced` events:

```
INFO Shipping.OrderPlacedHandler Shipping has received OrderPlaced, OrderId = 25c5ba63
```

Shipping is now receiving events published by **Sales** without having to change the code in the **Sales** endpoint. Additional subscribers could be added, for example, to email a receipt to the customer, notify a fulfillment agency via a web service, update a wish list or gift registry, or update data on items that are frequently bought together. Each business activity would occur in its own isolated message handler and doesn't depend on what happens in other parts of the system.

> You may also want to take a look at the ServicePulse window, where you should now be able to see heartbeat and endpoint monitoring information for the new endpoint as well.

Summary

In this tutorial, we explored the basics of how a messaging system using NServiceBus works.

We learned that asynchronous messaging failures in one part of a system can be isolated and prevent the entire system failure. That level of resilience and reliability is not easy to achieve with traditional REST-based web services.

We saw how automatic retries provide protection from transient failures like database deadlocks. If we implement a multi-step process as a series of message handlers, then each step will be executed independently and can be automatically retried in case of failures. This means that a stray exception won't abort an entire process, leaving the system in an inconsistent state.

We saw how the tooling in the Particular Service Platform makes running a distributed system much easier. ServicePulse gives us critical insights into the health of a system, and allows us to diagnose and fix systemic failures. We don't have to worry about data loss–once we redeploy our system, we can replay failed messages in batches as if the error had never occurred.

We also implemented an additional event subscriber, showing how to decouple independent bits of business logic from each other. The ability to publish one event and then implement resulting steps in separate message handlers makes the system much easier to maintain and evolve.

Now that you've seen what NServiceBus can do, take the next step and learn how to build a system like this one from the ground up. In the next tutorial, find out how to build the same solution starting from **File** > **New Project**.

Share your accomplishment

Next: NServiceBus from the ground up ❯ (/tutorials/nservicebus-step-by-step/1-getting-started/)

Last modified 3 weeks ago