Enterprise Development with NServiceBus

This is the workbook for the Enterprise Development with NServiceBus course. In it you will find all the exercises for the course as well as the solutions for them.

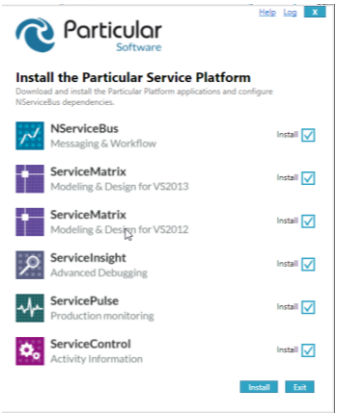
NServiceBus Basics

Using NuGet you open up a Visual Studio(make sure to run as Administrator) and install the NServiceBus package (`Install-Package NServiceBus.Host`).

Regardless of how you get access to the binaries NServiceBus will make sure that all of the necessary operating system services (MSMQ and DTC) are running and that RavenDB is installed properly (for troubleshooting see [1] and [2]). By far the most convenient way to get NServiceBus up and running is to use the platform installer. The platform brings along helpful tools for business application monitoring.

Commented [DM1]: Talk here about the docs repository and how users can improve the doco by sending PR to particular.

[Using Platform Installer http://particular.net/downloads]



Health check ServicePulse: http://localhost:9090/
Health check ServiceControl: http://localhost:33333/api/

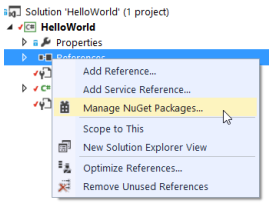
[Using PowerShell : http://particular.net/articles/managing-nservicebus-using-powershell]

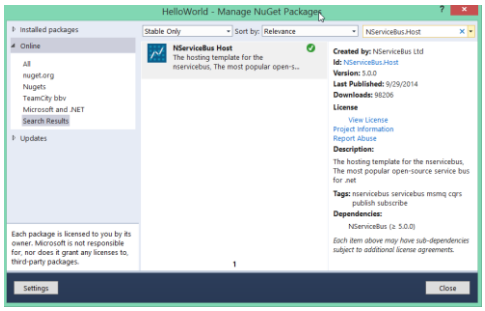
For all of the exercises in this workbook, .net 4.5 will be used since NServiceBus 5 is a 4.5 only framework.

[1] http://docs.particular.net/nservicebus/transactions-message-processing
[2] http://docs.particular.net/nservicebus/using-ravendb-in-nservicebus-installing

Exercise 1. Hello World

1. Create a new Visual Studio solution containing a class library project - call it HelloWorld.
2. Install NServiceBus.Host and NServiceBus.RavenDB via NuGet and follow the prompts to ensure the infrastructure is correctly configured





- Notice how it sets everything up for you automatically so that you can hit F5 right after the install completes.

3. Now also make EndpointConfig inherit from `IWantToRunWhenBusStartsAndStops`, adding an output to the console saying "Hello Distributed World" as shown here:

```
using NServiceBus;
using NServiceBus.Logging;
using NServiceBus.Persistence;

namespace HelloWorld
{
    public class EndpointConfig: IConfigureThisEndpoint, AsA_Client,
IWantToRunWhenBusStartsAndStops
    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.UsePersistence<RavenDBPersistence>();
        }

        public void Start()
        {
            LogManager.GetLogger("EndpointConfig").Info("Hello Distributed World!");
        }

        public void Stop()
        {

        }
    }
}
```

Commented [DM2]: Mention that apart from v5 NServiceBus will not enforce default transport. You have to choose!

Commented [UD3]: IConfigureThisEndpoint is a marker interface and only one class in the process should implement it (or an exception is thrown).

Commented [SF4]: Talk about having multiple class (in multiple DLLs) implementing `IWantToRunWhenBusStartsAndStops`.
The assembly scanning of NServiceBus will automatically pick them up. Useful for decoupling the startup logic of a system.
Also mention that you usually want to have separate classes implement `IConfigureThisEndpoint` and `IWantToRunWhenBusStartsAndStops`.

4. Add a app.config file to your project and add the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig, NServiceBus.Core" />
    <section name="UnicastBusConfig" type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core" />
    <section name="AuditConfig" type="NServiceBus.Config.AuditConfig, NServiceBus.Core" />
  </configSections>
  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings />
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

5. Compile and run. You should see "Hello Distributed World!" on the console application.

Exercise 2. Logging

1. Add the following configuration in addition the MessageForwardingInCaseOfFaultConfig section in your app.config, rebuild and run:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    . . .
    <section name="Logging" type="NServiceBus.Config.Logging, NServiceBus.Core" />
    . . .
  </configSections>

  <Logging Threshold="DEBUG" />
  <MessageForwarding . . .
</configuration>
```

Commented [UD5]: Mention that the default logging level is INFO. Also that there is a way to influence the level of logging without touching config or code - profiles, but that those will be covered later.

Now you should see a lot more log entries on the console. This configuration model controls logging for the whole endpoint. For finer grained control, standard log4net configuration can be used.

For example, to divert all log entries to a logfile you need to

2. Install NServiceBus.Logging.Log4net via nuget.
3. Tell NServiceBus to get the log settings from app.config by adding the necessary LogManager calls in your endpoint config.

```
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client,
IWantToRunWhenBusStartsAndStops
{
        public void Customize(BusConfiguration configuration)
        {
            log4net.Config.XmlConfigurator.Configure();
            LogManager.Use<Log4NetFactory>();

            configuration.UsePersistence<RavenDBPersistence>();
        }

        public void Start()
        {
            LogManager.GetLogger("EndpointConfig").Info("Hello Distributed World!");
        }

        public void Stop()
        {

        }
}
```

4. set your configuration as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="log4net"
type="log4net.Config.Log4NetConfigurationSectionHandler,log4net"/>
    . . .
  </configSections>

  <log4net debug="false">
    <appender name="FileAppender" type="log4net.Appender.FileAppender">
      <file value="myLog.txt"/>
      <appendToFile value="true"/>
      <layout type="log4net.Layout.PatternLayout">
        <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt; -
%m%n"/>
      </layout>
    </appender>
    <root>
      <level value="DEBUG"/>
      <appender-ref ref="FileAppender"/>
    </root>
  </log4net>
   . . .
</configuration>
```

Now return to the original configuration.

Exercise 3. One-way Messaging

1. Add a new project, call it Messages
2. Add a new class called RequestMessage
3. Add a string property to the Request class called SaySomething so the Request class looks like this:

```
namespace Messages
{
    public class RequestMessage
    {
        public string SaySomething { get; set; }
    }
}
```

Commented [UD6]: Talk about the ability to have messages without a default constructor, and get-only properties. Can be a useful technique for putting some validation logic in the message itself to decrease the likelihood of a sender sending an invalid message.

Commented [UD7]: Talk about the kinds of properties supported, including dictionaries (which aren't supported by standard XML serialization in .net).

4. Add the new Messages project as a reference to the HelloWorld project
5. Add a new class to the HelloWorld project called MessageSender as follows:

```
namespace HelloWorld
{
       using NServiceBus.Logging;
       using Messages;
       using NServiceBus;

       class MessageSender : IWantToRunWhenBusStartsAndStops
    {
        public IBus Bus { get; set; }

        public void Start()
        {
            var message = new RequestMessage { SaySomething = "Say something" };

            Bus.Send("helloWorldServer", message);

            LogManager.GetLogger("MessageSender").Info("Sent message.");
        }

        public void Stop()
        {
        }
    }
}
```

6. Next, change EndpointConfig to the following:

```
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client {}
```

Commented [UD8]: Describe the AsA_* interfaces as shortcuts for specifying NServiceBus behaviors, one of them being the use of XML serialization. Don't go into too much detail as the behaviors will be shown later.

7. Configure now the message convention in the `EndpointConfig`. Add a convention that specifies all classes ending with Message in the Message Assembly as messages.

   The `EndpointConfig` should look now like this:

```
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
 {
     public void Customize(BusConfiguration configuration)
     {
         configuration.Conventions()
             .DefiningMessagesAs(t => t.Assembly == typeof(RequestMessage).Assembly &&
t.Name.EndsWith("Message"));

         ...
     }
 }
```

8. Build and run.
9. You should see an error telling you that the "destination queue 'helloWorldServer ' could not be found."
10. Stop debugging
11. Create the queue called "helloWorldServer" by going to Server Explorer in Visual Studio, navigate through the local machine to Message Queues, and from there to Private Queues. Right-click on Private Queues and select Create Queue... and enter ”helloWorldServer” checking the box "Make queue transactional".
12. Build and run.
13. Navigate to the ”helloWorldServer” queue in Server Explorer in Visual Studio, and open the "Queue messages" node. That's the message.

14. With the message selected, go to the Properties pane in Visual Studio, select BodyStream property, and click on the "..." button. Scrolling to the right should show the following:

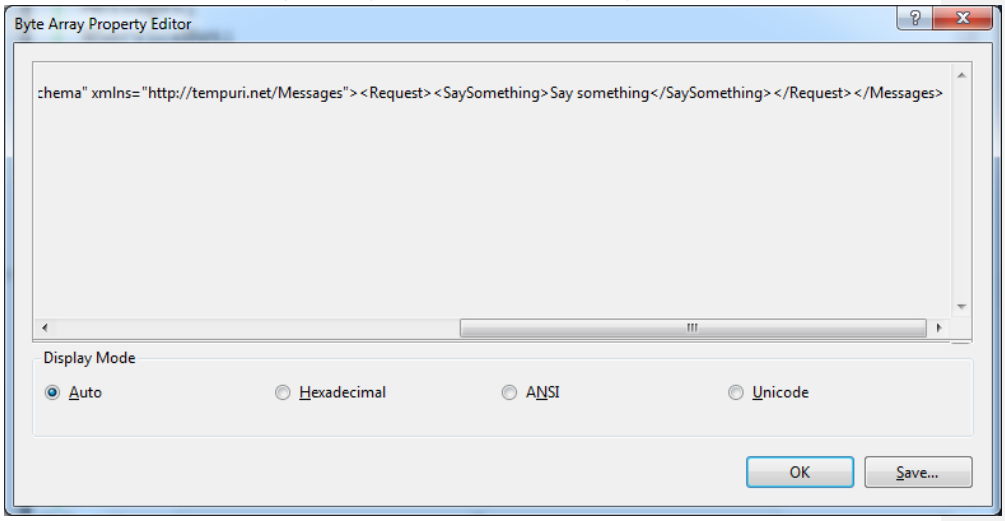


Figure 1 XML contents of the MSMQ message

Notice the namespace is http://tempuri.net/Messages. We probably don't want our system to go into production with that namespace so let's change it. But first, purge the message from the queue by right-clicking on the "Queue messages" node under the ”helloWorldServer” queue and clicking "Clear messages".

Exercise 4. Custom XML Namespace

In NServiceBus, namespaces are defined at an endpoint level rather than at a message level. So, in order to set the namespace for XML serialization on our Hello World endpoint, open EndpointConfig.cs and add the following:

```
namespace HelloWorld
{
    using Messages;
    using NServiceBus;
    using NServiceBus.Persistence;

    public class EndpointConfig : IConfigureThisEndpoint, AsA_Client,
    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.UseSerialization<XmlSerializer>()
                .Namespace("http://acme.com/");
            ...
        }
    }
}
```

Build and run again. Open up the same queue as before and look at the message inside it. You may need to right-click the queue and select "Refresh" to see the message appear. Notice the changed namespace.

Commented [DM9]: Mention: If someone new v4 mention that in v5 the ordering of the configuration calls doesn't matter anymore.

Also mention that NServiceBus does intense assembly scanning in the current AppDomain. The behavior of the assembly scanning can be tweaked like that:

```
configuration.AssembliesToScan(AllAssemblies.Except().And());
```

Furthermore mention the possibility to swap out the container used by NServiceBus. Default is AutoFac. You can change the container by installing the NServiceBus container adapter of preferences and calling (i.ex. with ninject)

```
configuration.UseContainer<NinjectBuilder>(c => c.ExistingKernel());
```

Exercise 5. Configurable Routing

Rather than hard-coding the destination to which messages get sent, we'll now use the configuration file. First change the Start method of the MessageSender class as follows:

```
        public void Start()
        {
            var message = new RequestMessage {SaySomething = "Say something"};

            Bus.Send(message);

            LogManager.GetLogger("MessageSender").Info("Sent message.");
        }
```

Add the following to your app.config:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
. . .
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Messages.RequestMessage, Messages" Endpoint="helloWorldServer" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
</configuration>
```

Notice how we specified the type of the message "Messages.RequestMessage, Messages" and indicated the endpoint to which it will be sent, ”helloWorldServer”. You can also include just the name of the assembly, "Messages" to indicate that all types in the assembly should be routed to the same endpoint.

Exercise 6. Processing Messages

Now add another project in the same way you added the HelloWorld project in Exercise 1, calling it HelloWorldServer. Add a reference to the Messages project.

The Class1 file in the new project should be as follows:

```
namespace HelloWorldServer
{
    using Messages;
    using NServiceBus;
    using NServiceBus.Persistence;

    public class EndpointConfig: IConfigureThisEndpoint, AsA_Server,
    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.UseSerialization<XmlSerializer>()
                .Namespace("http://acme.com/");

            configuration.Conventions()
                .DefiningMessagesAs(t => t.Assembly == typeof(RequestMessage).Assembly &&
t.Name.EndsWith("Message"));

            configuration.UsePersistence<RavenDBPersistence>();
        }    }
}
```

Notice here the use of AsA_Server rather than AsA_Client. This is the preferred choice for endpoints that process messages.

Add a new class to the project, call it RequestHandler, and have it implement the interface IHandleMessages<RequestMessage> as follows:

```
using log4net;
using Messages;
using NServiceBus;
using NServiceBus.Logging;

namespace HelloWorldServer
{
    class RequestHandler : IHandleMessages<RequestMessage>
    {
        public void Handle(RequestMessage message)
        {
            LogManager.GetLogger("RequestHandler").Info(message.SaySomething);
        }
    }
}
```

Commented [a10]: Mention that the endpoint name is defaulted to the name space of the endpoint config

Commented [UD11]: Mention that there are differences between the client and server setups - things like transactionality and impersonation.

Commented [DM12]: Emphasize the fact that endpoints that talk to each other must use the same serialization mechanism AND namespace.

Commented [DM13]: As_AServer: Indicates that the endpoint behaves as a server. It is configured as a transactional endpoint that does not purge messages on startup;
As_APublisher: Indicates that the endpoint is a publisher that can publish events, and extends the server role. An endpoint configured as a publisher cannot be configured as client at the same time;
As_AClient: Indicates that the endpoint is a client. A client endpoint is configured as a non-transactional endpoint that purges messages on startup.

Finally, right-click on the solution, click "Set Startup Projects...", and set both the HelloWorld and HelloWorldServer as projects to start.

Build and run. You should see the HelloWorldServer print out "Say something" on the console.

Commented [UD14]: Mention that both processes don't need to be online for communication to work, as opposed to RPC.
Pair people up, have the client on one machine send to a server on the other machine. Tell them to change the client config to include the @OtherMachine after "helloWorldServer".
Disconnect the client machine from the network and run again. Tell them to go to Computer management and look at the Outgoing Queues. Then reconnect and see the messages arrive and be processed.
Tell them that it's the Msmq service (mqsvc.exe) which is responsible for this "store and forward" messaging.

Exercise 7. Discarding messages

Part 1: Conventions Configuration for expiry

1. Add an Attribute Expires to the Messages project as follows:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ExpiresAttribute : Attribute
{
    public ExpiresAttribute(int expiresAfterSeconds)
    {
        ExpiresAfter = TimeSpan.FromSeconds(expiresAfterSeconds);
    }

    public TimeSpan ExpiresAfter { get; private set; }
}
```

2. Decorate the RequestMessage with the new attribute and specify that it expires after 60 seconds.

```
[Expires(60)]
public class RequestMessage
{
    public string SaySomething { get; set; }
}
```

3. Extend the `EndpointConfig` of the client and server with the "time to be received" conventions like this:

```
        public void Customize(BusConfiguration configuration)
        {
            …

            configuration.Conventions()
                .DefiningMessagesAs(t => t.Assembly == typeof(RequestMessage).Assembly &&
t.Name.EndsWith("Message"))
                .DefiningTimeToBeReceivedAs(GetExpiration);
            …
        }

        private static TimeSpan GetExpiration(Type type)
        {
            dynamic expiresAttribute = type.GetCustomAttributes(true)
                        .SingleOrDefault(t => t.GetType()
                        .Name == "ExpiresAttribute");
            return expiresAttribute == null
                       ? TimeSpan.MaxValue
                       : expiresAttribute.ExpiresAfter;
        }
```

Run only the client - see the message in the "helloWorldServer" queue. Wait for a minute or two. Refresh the queue. Notice that the message is no longer in the queue.

Part 2: Centralized Conventions Configuration for expiry

In the first part we duplicated the conventions configuration to all projects. This can be a huge maintenance problem especially in large system. So we will now extract the conventions into a separate project.

1. Add a new Class Library called Conventions
2. Reference NServiceBus.Core (via NuGet)
3. Add a class ConventionsConfiguration to the new project
4. Implement the INeedInitialization interface to the new class
5. Move the conventions configuration to the Init method.
6. Change the Message convention check for an attribute named MessageAttribute to remove the reference to the Messages assembly

   The class should now look like this:

```
public class ConventionsConfiguration : INeedInitialization

    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.Conventions()
                .DefiningMessagesAs(type => type.GetCustomAttributes(true).Any(t =>
t.GetType().Name == "MessageAttribute"))
                .DefiningTimeToBeReceivedAs(GetExpiration);
        }

        private TimeSpan GetExpiration(Type type)
        {
            dynamic expiresAttribute = type.GetCustomAttributes(true)
                        .SingleOrDefault(t => t.GetType()
                        .Name == "ExpiresAttribute");

            return expiresAttribute == null
                       ? TimeSpan.MaxValue
                       : expiresAttribute.ExpiresAfter;
        }
    }
```

1. Add an attribute MessageAttribute to the Messages assembly and add it to the RequestMessage
2. Rename RequestMessage to Request. Be sure to update the config as well.
3. Remove the conventions configurations from the client and server
4. Add a reference to the conventions assembly to both the client and server
5. Run both client and server and see how they still do the same.

Exercise 8. Exceptions

Add the following to the app.config of the HelloWorldServer project

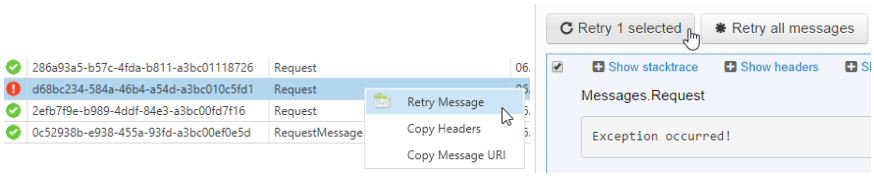
```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig" type="NServiceBus.Config.
MessageForwardingInCaseOfFaultConfig, NServiceBus.Core"/>

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

   </configSections>

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

1. Change the RequestHandler class in HelloWorldServer so that it throws an exception.
2. Make sure ErrorQueue attribute of the MessageForwardingInCaseOfFaultConfig section of the app.config file is set to "error".
3. Build and run.
4. Notice the log output of HelloWorldServer that indicates the message was retried several times.
5. Notice that the retries will resume after 10,20,30 seconds. This is the Second Level Retries
6. Look at the "error" queue, see that it now contains the same message that was sent.
7. Change the RequestHandler class back to what it was before so it doesn't throw an exception anymore.
8. Run only HelloWorldServer.
9. Open either ServicePulse or ServiceInsight and retry the failed message.



10. Notice that HelloWorldServer receives the message and that the error queue is now empty.
11. Change the number of retries by adding the following to your app.config:
    a. `<TransportConfig MaxRetries="5" />`

Commented [a15]: Mention that 5 retries is the default and the section can be left out if the defaults are ok

Exercise 9. Authorization

Add a new class to HelloWorldServer called Auth. Have it stop the message pipeline based on some header.

```
using log4net;
using NServiceBus;

namespace HelloWorldServer
{
    class Auth : IHandleMessages<IMessage>
    {
        public IBus Bus { get; set; }

        public void Handle(IMessage message)
        {
            if (!Authorized(message.GetHeader("user")))
            {
                LogManager.GetLogger("Auth").Warn("User not authorized.");
                Bus.DoNotContinueDispatchingCurrentMessageToHandlers();
            }
            else
                LogManager.GetLogger("Auth").Info("User authorized.");
        }

        bool Authorized(string user)
        {
            return user == "udi";
        }
    }
}
```

Make sure that the Auth message handler is configured to run first by adding a class called Ordering which implements ISpecifyMessageHandlerOrdering.

```
using NServiceBus;

namespace HelloWorldServer
{
    internal class Ordering : ISpecifyMessageHandlerOrdering
    {
        public void SpecifyOrder(Order order)
        {
            order.SpecifyFirst<Auth>();
        }
    }
}
```

Build and run. Notice that the server doesn't output the message contents any more.

Modify the HelloWorld project so that the server will behave as before.

```
using log4net;
using Messages;
using NServiceBus;

namespace HelloWorld
{
    class MessageSender : IWantToRunAtStartup
    {
        public IBus Bus { get; set; }

        public void Run()
        {
            Bus.OutgoingHeaders["user"] = "udi";
            var message = new Request {SaySomething = "Say something"};
            Bus.Send(message);
            LogManager.GetLogger("MessageSender").Info("Sent message.");
        }

        public void Stop()
        {

        }
    }
}
```

Build and run.

Exercise 10. Dependency Injection

Add an interface to HelloWorldServer called ISaySomething as follows:

```
namespace HelloWorldServer
{
    public interface ISaySomething
    {
        string InResponseTo(string request);
    }
}
```

Now add a class that implements it called SaySomething as follows:

```
namespace HelloWorldServer
{
    class SaySomething : ISaySomething
    {
        public string InResponseTo(string request)
        {
            return "Responding to " + request;
        }
    }
}
```

Now change the RequestHandler class to use the new interface (using constructor injection):

```
using Messages;
using NServiceBus;
using NServiceBus.Logging;

namespace HelloWorldServer
{
    class RequestHandler : IHandleMessages<Request>
    {
        public RequestHandler(ISaySomething something)
        {
            saysSomething = something;
        }
        private ISaySomething saysSomething;

        public void Handle(Request message)
        {
            LogManager.GetLogger("RequestHandler").Info(message.SaySomething);
            LogManager.GetLogger("RequestHandler").Info(
                saysSomething.InResponseTo(message.SaySomething));
        }
    }
}
```

Finally, register the SaySomething in the container by added a new class called ComponentInitializer which implements INeedInitialization in HelloWorldServer:

```
    class ComponentInitializer : INeedInitialization
    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.RegisterComponents(c =>
c.ConfigureComponent<SaySomething>(DependencyLifecycle.InstancePerCall));
        }
    }
```

Build and run.

Exercise 11. Encryption

Add an EncryptedAttribute to the Messages project as follows:

```
namespace Messages
{
    using System;

    [AttributeUsage(AttributeTargets.Property)]
    public class EncryptedAttribute : Attribute
    {
    }
}
```

Change the Request class in the Messages project as follows:

```
using NServiceBus;

namespace Messages
{
    [Expires(60)]
    [Message]
    public class Request
    {
        [Encrypted]
        public string SaySomething { get; set; }
    }
}
```

Add the following to the configSections of both HelloWorld and HelloWorldServer:

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

Then add the following to the both configuration files:

```
<RijndaelEncryptionServiceConfig Key="gdDbqRpqdRbTs3mhdZh9qCaDaxJXl+e7"/>
```

Also, add a call to .RijndaelEncryptionService() to the EndpointConfig file of both projects.

Build - but only run HelloWorld - not the server.

Open the "helloWorldServer" queue - see that the contents of the SaySomething node are now encrypted. Now run the HelloWorldServer project - notice that it is able to unencrypt the contents successfully.

Exercise 12. Overriding Configuration

Now we're going to remove the encryption key from the configuration files and move it to an “external service”.

1. Add a Class Library project and call it SecurityServiceAdapter.
2. Add references to NServiceBus.dll, NServiceBus.Core.dll, and System.Configuration.
3. Add a class to SecurityServiceAdapter called ConfigOverride as follows:

```
namespace SecurityServiceAdapter
{
    using NServiceBus.Config;
    using NServiceBus.Config.ConfigurationSource;

    public class ConfigOverride : IProvideConfiguration<RijndaelEncryptionServiceConfig>
    {
        public RijndaelEncryptionServiceConfig GetConfiguration()
        {
            return new RijndaelEncryptionServiceConfig
                       {
                           //this key could be fetched from a REST/WS call
                           Key = "gdDbqRpqdRbTs3mhdZh9qCaDaxJXl+e7"
                       };
        }
    }
}
```

Commented [a16]: Mention that NSB will automatically configure all classes implementing IProvideConfiguration

Commented [a17]: Mention that this is only called once when the endpoint is initializing.

Commented [a18]: Mention that in a real life scenario this would be a call to a external service

4. Remove the RijndaelEncryptionServiceConfig section from your projects
5. Add a reference to the SecuritySystemAdapter project.
6. Build and run. Try putting a breakpoint in the GetConfiguration() method to see that the calls do indeed happen.

Exercise 13. Web App Hosting

Add a Asp.Net MVC Web Application project to the solution called MvcApplication1, choose the Empty template, and add references to NServiceBus.dll, NServiceBus.Core.dll, log4net.dll, and the SecurityServiceAdapter, Conventions, and Messages projects. Make sure to include the new project in the startup projects for the solution.

Initialize the bus in the Startup class of the Owin pipeline method as follows:

```
namespace MvcApplication1
{
using Microsoft.Owin.BuilderProperties;
using NServiceBus;
using NServiceBus.Persistence;
using Owin;

[assembly: OwinStartupAttribute(typeof(MvcApplication1.Startup))]

namespace MvcApplication1
{
    public partial class Startup
    {
        public static ISendOnlyBus Bus { get; private set; }

        public void Configuration(IAppBuilder app)
        {
            var properties = new AppProperties(app.Properties);
            var token = properties.OnAppDisposing;

            var configuration = new BusConfiguration();

            configuration.ScaleOut().UseSingleBrokerQueue();

            configuration.UsePersistence<RavenDBPersistence>();

            Bus = NServiceBus.Bus.CreateSendOnly(configuration);

            token.Register(() => Bus.Dispose());
        }
    }
}}
```

Add A reference to the Conventions and the SecurityServiceAdapter project, so we include the message definitions for this endpoint.

Hint: There are a few configurations missing in the code above. Look into the other endpoint configurations and try to refactor common configuration into the conventions assembly.

Create a new controller called SaySomethingController by right clicking on the controller's folder and then send a message in the event handler as follows:

```
using System.Web.Mvc;
```

```
namespace MvcApplication1.Controllers
{
    public class SaySomethingController : Controller
    {
        public ActionResult Index()
        {
            Startup.Bus.Send<Request>(m => m.SaySomething = "Say 'WebApp'.");

            return new ContentResult{Content ="Message sent"};
        }
    }
}
```

Now open the web.config file and fill in the appropriate UnicastBusConfig to specify the destination address.

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="UnicastBusConfig" type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core"/>
  </configSections>

  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Messages.Request, Messages" Endpoint="helloWorldServer" />
    </MessageEndpointMappings>
  </UnicastBusConfig>

</configuration>
```

Build and run. Change the url to http://localhost:{port assigned}/SaySomething. Notice that when HelloWorldServer processes it, it cannot authorize the request. This time we will use a outgoing transport message mutator to fix the issue.

```
    public class AuthenticationMutator : IMutateOutgoingTransportMessages,
        INeedInitialization
    {
        public void MutateOutgoing(LogicalMessage logicalMessage, TransportMessage
transportMessage)
        {
            //in a real life scenario you would probably get the user from
            // the HTTP pipeline
            transportMessage.Headers["user"] = "udi";
        }

        public void Customize(BusConfiguration configuration)
        {
            configuration.RegisterComponents(c =>
c.ConfigureComponent<AuthenticationMutator>(DependencyLifecycle.InstancePerCall));
        }
    }
```

Build and run again - see that the messages now are authorized.

Exercise 14. Full Duplex

Add a new message type to the Messages project as follows:

```
public class RequestWithResponse : Request {}
```

Modify the controller to be an asynchronous controller:

```
public class SaySomethingController : AsyncController
{
    public async Task<ActionResult> Index()
    {
        var message = new RequestWithResponse("Say 'WebApp'.");

        var response = await Startup.Bus.Send(message).Register();

        return new ContentResult { Content = "Response from server " + response };
    }
}
```

Notice the use of Register to setup a callback to handle the server response

Add a message handler in the HelloWorldServer project to handle the new message type as follows:

```
public class RequestWithResponseHandler : IHandleMessages<RequestWithResponse>
{
    public IBus Bus { get; set; }

    public void Handle(RequestWithResponse message)
    {
        Bus.Return(message.SaySomething.Value.Length % 2);
    }
}
```

Notice the use of Bus.Return to return a simple integer value back to the client.

Build and run.

Take note of the exception:

> "No destination specified for message Messages.RequestWithResponse. Message cannot be sent. Check the UnicastBusConfig section in your config file and ensure that a MessageEndpointMapping exists for the message type."

Change the web.config file's UnicastBusConfig section to specify that all messages from the Messages project should be routed to the ”helloWorldServer” queue as follows:

```
<UnicastBusConfig>
  <MessageEndpointMappings>
    <add Assembly="Messages" Endpoint=”helloWorldServer” />
  </MessageEndpointMappings>
</UnicastBusConfig>
```

Notice the use of just the assembly name "Messages" where before we used the fully qualified type names of the form: Namespace.Type, Assembly

Build and run.

Notice that the HelloWorldServer gets a queue not found exception. The reason for this is that running in SendOnly mode means that no input queue is created for your website. To fix this we need to make the website a regular send/receive endpoint. Modify the configuration in Startup class of the Owin pipeline:

```
using Microsoft.Owin;
using Microsoft.Owin.BuilderProperties;
using NServiceBus;
using NServiceBus.Persistence;
using Owin;

[assembly: OwinStartupAttribute(typeof(MvcApplication1.Startup))]

namespace MvcApplication1
{
    public partial class Startup
    {
        public static IBus Bus { get; private set; }

        public void Configuration(IAppBuilder app)
        {
            var properties = new AppProperties(app.Properties);
            var token = properties.OnAppDisposing;

            var configuration = new BusConfiguration();

            configuration.ScaleOut().UseSingleBrokerQueue();

            configuration.UsePersistence<RavenDBPersistence>();

            Bus = NServiceBus.Bus.Create(configuration).Start();

            token.Register(() => Bus.Dispose());
        }
    }
}
```

Build and Run. Notice that a new queue called MvcApplication1 is created.

Commented [a19]: Mention that this is the endpointname that is defaulted to the namespace of Global.asax.cs

It should now work as expected. Try to change the length of the string

Now change the message handler such that it sleeps for 5 seconds - build and run again. Notice that the page does not finish rendering for an additional 5 seconds.

Exercise 15. Unit Testing

Add a new project called HelloWorldServer.Tests. Also reference the Messages, Conventions and HelloWorldServer projects.

Install the following nuget package:

```
Install-package nservicebus.testing
```

Now add a reference to your testing framework of choice. The example below is using NUnit.

Now we're going to write a test that checks that if an empty string is passed in the message that the RequestWithResponseHandler will call Bus.Return with the parameter 0:

```
using Messages;
using NServiceBus.Testing;
using NUnit.Framework;

namespace HelloWorldServer.Tests
{
    [TestFixture]
    public class Class1
    {
        [Test]
        public void TestRequestHandler()
        {
            Test.Initialize();

            Test.Handler<RequestWithResponseHandler>()
                .ExpectReturn<int>(i => i == 0)
                .OnMessage<RequestWithResponse>(m => m.SaySomething = "");
        }
    }
}
```

Build and run the unit test.

Exercise 16. Multiple Responses

Add the following message types to the Messages project:

```
[Message]
public class Query
{
    public Query(int numberOfResponses)
    {
        NumberOfResponses = numberOfResponses;
    }
    public int NumberOfResponses { get; private set; }
}

[Message]
public class QueryResult
{
    public QueryResult(string saySomething)
    {
        SaySomething = saySomething;
    }
    public string SaySomething { get; private set; }
}
```

Create a new regular controller that sends a Query message to the server, hard code the NumberOfResponses to 10 for now. Add a message handler to MvcApplication1 that handles the QueryResult message - have it simply call a .ToString() on the message. Put a breakpoint on that code.

Modify the startup class of the Owin pipeline of MvcApplication1 to call this line:

```
configuration.DisableFeature<AutoSubscribe>();
```

Add a message handler to HelloWorldServer that handles the Query message - have it loop according to the NumberOfResponses, and Bus.Reply with a QueryResult message containing the loop variable's .ToString():

```
using Messages;
using NServiceBus;

namespace HelloWorldServer
{
    public class QueryHandler : IHandleMessages<Query>
    {
        public IBus Bus { get; set; }

        public void Handle(Query message)
        {
            for (int i=0; i < message.NumberOfResponses; i++)
                Bus.Reply<QueryResult>(m => m.Something = i.ToString());
        }
    }
}
```

Put a breakpoint on the Bus.Reply line. Build and run. Notice that MvcApplication1 does not receive any of the replies until the loop has completed - ultimately not providing the "streaming" behavior we wanted.

Exercise 17. Streaming Responses

Create a new project called HelloWorldQueryServer based on HelloWorldServer. Make sure you give it a separate input queue. Also, do not configure the endpoint as a server (since we don't want the transactional behavior) but rather as a client.

Move the QueryHandler class from HelloWorldServer to the new HelloWorldQueryServer.

Finally, change the routing of MvcApplication1 to send Query messages to the new endpoint.

Build and run.

Notice that this time the replies are received by MvcApplication1 even before loop has completed.