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.

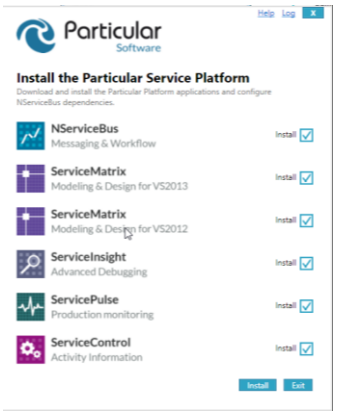
NServiceBus Basics

Using NuGet you open a VisualStudio (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 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/

RavenDB installation: http://docs.particular.net/nservicebus/ravendb/installation

SQL Server express 2016: https://www.microsoft.com/en-us/download/details.aspx?id=52679

SQL Server Management studio: https://msdn.microsoft.com/library/mt238290.aspx

For all exercises in this workbook, .net 4.6.1 will be used since NServiceBus 6 is compatible with 4.5.2 framework and above.

[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.NHibernate 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.

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

    /*
                This class configures this endpoint as a Server. More information about how to
configure the NServiceBus host
                can be found here: http://particular.net/articles/the-nservicebus-host
       */
    public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
    {
        public void Customize(EndpointConfiguration configuration)
        {
            configuration.UsePersistence<RavenDBPersistence>();
        }

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

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

3. Add another class, call it RunWhenEndpointStartsAndStops and add an output to the console saying "Hello Distributed World" as shown here

```
namespace HalloWorld
{
    using System.Threading.Tasks;
    using NServiceBus;
    using NServiceBus.Logging;

    class RunWhenEndpointStartsAndStops : IWantToRunWhenEndpointStartsAndStops
    {
        public Task Start(IMessageSession session)
        {
            // perform startup logic
            LogManager.GetLogger("EndpointConfig").Info("Hello Distributed World!");

            return Task.CompletedTask;
        }

        public Task Stop(IMessageSession session)
        {
            // perform shutdown logic
```

```
            return Task.CompletedTask;
        }
    }
}
```

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

```
<?xml version="1.0" encoding="utf-8" ?>
  <configuration>
    <connectionStrings>
      <add name="NServiceBus/Persistence" connectionString="Data
Source=.\SqlExpress;Database=HalloWorld.Persistence;Integrated Security=True"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

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

Exercise 2. Logging

1. Add the following to your app.config file:

```
<configSections>
   <section name="Logging" type="NServiceBus.Config.Logging, NServiceBus.Core" />
 </configSections>
 <Logging Threshold="DEBUG" />
```

2. Now you should see a lot more log entries on the console. This configuration model controls logging for the whole endpoint.

3. For finer grained control, standard log4net configuration can be used.
4. Install NServiceBus.Log4Net using NuGet (`Install-Package NServiceBus.Log4Net`)
5. Add the following in your EndpointConfig.cs Customize method:

```
            // ConfigureLog4Net

        var layout = new PatternLayout
        {
            ConversionPattern = "%d [%t] %-5p %c [%x] - %m%n"
        };
        layout.ActivateOptions();
        var consoleAppender = new ConsoleAppender
        {
            Threshold = Level.Info,
            Layout = layout
        };
        consoleAppender.ActivateOptions();

        BasicConfigurator.Configure(consoleAppender);

        LogManager.Use<Log4NetFactory>();
```

6. Build and run, 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 log file you need to.
7. Now un do all the changes we did relating to logging.

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; }
    }
}
```

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

```
namespace HalloWorld
{
    using System.Threading.Tasks;
    using Messages;
    using NServiceBus;
    using NServiceBus.Logging;

    class MessageSender : IWantToRunWhenEndpointStartsAndStops
    {
        public async Task Start(IMessageSession session)
        {
            // perform startup logic
            var message = new RequestMessage { SaySomething = "Say something" };

            await session.Send("helloWorldServer", message).ConfigureAwait(false);

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

        public Task Stop(IMessageSession session)
        {
            // perform shutdown logic
            return Task.CompletedTask;
        }
    }
}
```

5. Next, configure the message convention in the `EndpointConfig`.
6. Add a convention that specifies all classes ending with Message in the Message Assembly as messages.

   The EndpointConfig should look now like this:

```
namespace HalloWorld
{
    using Messages;
    using NHibernate.Cfg;
    using NServiceBus;
    using NServiceBus.Persistence;
```

```
    public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
    {
        public void Customize(EndpointConfiguration endpointConfiguration)
        {
            //TODO: NServiceBus provides multiple durable storage options,
              including SQL Server, RavenDB, and Azure Storage Persistence.
            // Refer to the documentation for more details on specific options.
            var persistence = endpointConfiguration.UsePersistence<NHibernatePersistence>();

            // configur in code
            var nhConfig = new Configuration();
            nhConfig.SetProperty(Environment.ConnectionProvider,
                   "NHibernate.Connection.DriverConnectionProvider");
            nhConfig.SetProperty(Environment.ConnectionDriver,
                    "NHibernate.Driver.Sql2008ClientDriver");
            nhConfig.SetProperty(Environment.Dialect, "NHibernate.Dialect.MsSql2008Dialect");
            nhConfig.SetProperty(Environment.ConnectionStringName,
                     "NServiceBus/Persistence");

            persistence.UseConfiguration(nhConfig);

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

            // NServiceBus will move messages that fail repeatedly to a separate "error"
                    queue. We recommend
            // that you start with a shared error queue for all your endpoints for easy
                    integration with ServiceControl.
            endpointConfiguration.SendFailedMessagesTo("error");

            // NServiceBus will store a copy of each successfully process message in a
                    separate "audit" queue. We recommend
            // that you start with a shared audit queue for all your endpoints for easy
                    integration with ServiceControl.
            endpointConfiguration.AuditProcessedMessagesTo("audit");
        }
    }
}
```

7. Build and run.
8. You should see an error telling you that the "destination queue 'helloWorldServer ' could not be found."
9. Stop debugging
10. 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".
11. Build and run.
12. Navigate to the `helloWorldServer` queue in Server Explorer in Visual Studio, and open the "Queue messages" node. That's the message.

13. 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:

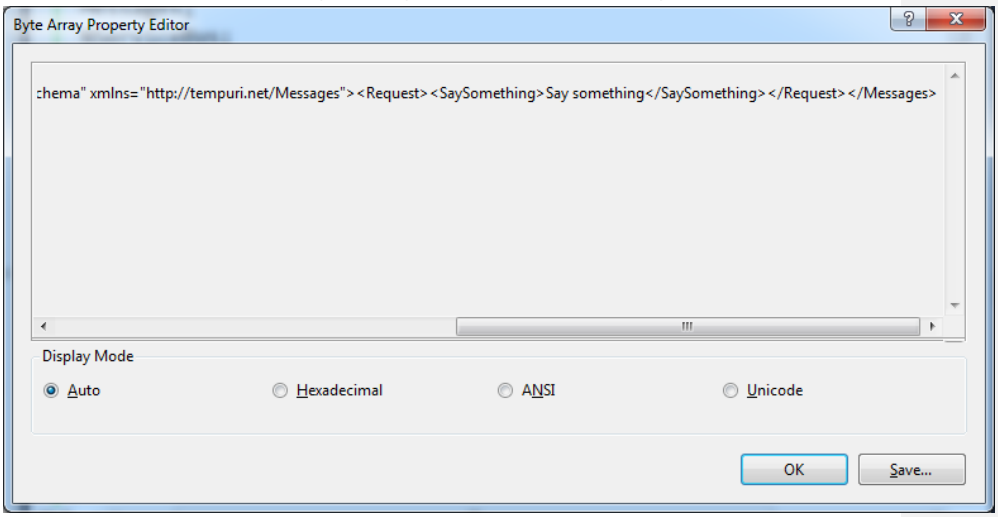


Figure 1 XML contents of the MSMQ message

14. purge the message from the queue by right-clicking on the "Queue messages" node under the ‘helloWorldServer’ queue and clicking "Clear messages"

Exercise 4. 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:

```
namespace HalloWorld
{
    using System.Threading.Tasks;
    using Messages;
    using NServiceBus;
    using NServiceBus.Logging;

    class MessageSender : IWantToRunWhenEndpointStartsAndStops
    {
        public async Task Start(IMessageSession session)
        {
            // perform startup logic
            var message = new RequestMessage { SaySomething = "Say something" };

            await session.Send(message).ConfigureAwait(false);

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

        public Task Stop(IMessageSession session)
        {
            // perform shutdown logic
            return Task.CompletedTask;
        }
    }
}
```

Add the following to your app.config:

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

For more information see:
https://docs.particular.net/nservicebus/messaging/routing
and https://docs.particular.net/nservicebus/msmq/routing#using-message-endpoint-mappings

Exercise 5. 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.

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

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

    public class EndpointConfig : IConfigureThisEndpoint,AsA_Server
    {
        public void Customize(EndpointConfiguration endpointConfiguration)
        {
            //TODO: NServiceBus provides multiple durable storage options, including SQL
Server, RavenDB, and Azure Storage Persistence.
            // Refer to the documentation for more details on specific options.
            var persistence = endpointConfiguration.UsePersistence<NHibernatePersistence>();

            // configur in code
            var nhConfig = new Configuration();
            nhConfig.SetProperty(Environment.ConnectionProvider,
"NHibernate.Connection.DriverConnectionProvider");
            nhConfig.SetProperty(Environment.ConnectionDriver,
"NHibernate.Driver.Sql2008ClientDriver");
            nhConfig.SetProperty(Environment.Dialect, "NHibernate.Dialect.MsSql2008Dialect");
            nhConfig.SetProperty(Environment.ConnectionStringName,
"NServiceBus/Persistence");

            persistence.UseConfiguration(nhConfig);

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



            // NServiceBus will move messages that fail repeatedly to a separate "error"
queue. We recommend
            // that you start with a shared error queue for all your endpoints for easy
integration with ServiceControl.
            endpointConfiguration.SendFailedMessagesTo("error");

            // NServiceBus will store a copy of each successfully process message in a
separate "audit" queue. We recommend
            // that you start with a shared audit queue for all your endpoints for easy
integration with ServiceControl.
            endpointConfiguration.AuditProcessedMessagesTo("audit");
        }
    }
}
```

2. Add an app.config file and add the connection string like so:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="NServiceBus/Persistence"
         connectionString="Data
Source=.\SqlExpress;Database=HalloWorld.Persistence;Integrated Security=True"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

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

Commented [DM2]: 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.

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

```
namespace HelloWorldServer
{
    using System.Threading.Tasks;
    using Messages;
    using NServiceBus;
    using NServiceBus.Logging;

    class RequestHandler : IHandleMessages<RequestMessage>
    {
        public Task Handle(RequestMessage message, IMessageHandlerContext context)
        {
            LogManager.GetLogger("RequestHandler").Info(message.SaySomething);

            return Task.CompletedTask;
        }
    }
}
```

4. Finally, right-click on the solution, click "Set Startup Projects...", and set both the HelloWorld and HelloWorldServer as projects to start.
5. Build and run. You should see the HelloWorldServer print out "Say something" on the console.

Commented [UD3]: 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 6. 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;
        }
```

4. 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 'Customize' method.

Change the Message convention check for an attribute named MessageAttribute to remove the reference to the Messages assembly

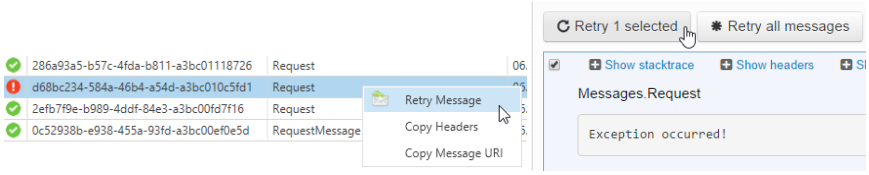
The class should now look like this:

```
namespace Conventions
{
    using System;
    using System.Linq;
    using NServiceBus;
    public class ConventionsConfiguration : INeedInitialization
    {
        public void Customize(EndpointConfiguration configuration)
        {
            // MessageConventions
            configuration.Conventions()
               .DefiningMessagesAs(type => type.GetCustomAttributes(true).Any(t =>
t.GetType().Name == "MessageAttribute"))
               .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;
        }
    }
}
```

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

Part 3: Exceptions

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



9. Notice that HelloWorldServer receives the message.
10. Change the number of retries to 2 by adding the following to your endpointconfig file:

```
var recoverability = endpointConfiguration.Recoverability();

recoverability.Immediate(settings =>
{
    settings.NumberOfRetries(2);
});
```

Exercise 7. 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 addeding 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 8. Encryption

1. Add an EncryptedAttribute to the Messages project as follows:

```
namespace Messages
{
    using System;

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

2. Change the Request class in the Messages project as follows:

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

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

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

4. Then add the following to the both configuration files:

```
  <RijndaelEncryptionServiceConfig Key="tWIrVnPy3Smd2r/xcSb9r/1wcVY9/3KWbrzIaemx5yw="
                                   KeyIdentifier="2015-10"
                                   KeyFormat="Base64">
  </RijndaelEncryptionServiceConfig>
```

5. Add a call to endpointConfiguration.RijndaelEncryptionService(); to the EndpointConfig file of both projects.

Add the property encryption convention in ConventionsConfiguration.cs like below:

```
configuration.Conventions()
            .DefiningEncryptedPropertiesAs(type => type.GetCustomAttributes(true).Any(t =>

            t.GetType().Name == "EncryptedAttribute"));
```

6. Build - but only run HelloWorld - not the server.
7. Open the "helloWorldServer" queue - see that the contents of the SaySomething node are now encrypted.
8. Now run the HelloWorldServer project - notice that it can unencrypt the contents successfully.

Exercise 9. 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.Core using Nuget.
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
            {
                Key = "gdDbqRpQdRbTs3mhdZh9qCaDaxJXl+e6",
                KeyIdentifier = "2015-10",
                KeyFormat = KeyFormat.Base64,
                ExpiredKeys = new RijndaelExpiredKeyCollection
                {
                    new RijndaelExpiredKey
                    {
                        Key = "abDbqRpQdRbTs3mhdZh9qCaDaxJXl+e6",
                        KeyIdentifier = "2015-09",
                        KeyFormat = KeyFormat.Base64
                    },
                    new RijndaelExpiredKey
                    {
                        Key = "cdDbqRpQdRbTs3mhdZh9qCaDaxJXl+e6"
                    },
                }
            };
        }
    }
}
```

4. Remove the RijndaelEncryptionServiceConfig section in the app.confic files 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 10. Web App Hosting

1. Add an Asp.Net MVC Web Application project to the solution called WebApplication1, choose the MVC template, and add references to the SecurityServiceAdapter, Conventions, and Messages projects.
2. Using NuGet add the following packages to the project, make sure to include the new project in the startup projects for the solution.
    a. NServiceBus,
    b. NServiceBus.Nhibernate,
    c. Autofac 4.2.0,
    d. Autofac.Mvc5 Version: 4.0.0,
    e. NServiceBus.Autofac Version: 6.0.0
3. Initialize the bus in the Application_Start method as follows:

```
namespace WebApplication1
{
    using System.Web.Mvc;
    using System.Web.Routing;
    using Autofac;
    using Autofac.Integration.Mvc;
    using NHibernate.Cfg;
    using NServiceBus;
    using NServiceBus.Persistence;

    public class MvcApplication : System.Web.HttpApplication
    {
        IEndpointInstance _endpoint;

        protected void Application_Start()
        {
            var builder = new ContainerBuilder();

            // Register MVC controllers.
            builder.RegisterControllers(typeof(MvcApplication).Assembly);

            // Set the dependency resolver to be Autofac.
            var container = builder.Build();

            var endpointConfiguration = new
EndpointConfiguration("HelloWorld.WebApplication");
            // instruct NServiceBus to use a custom AutoFac configuration
            endpointConfiguration.UseContainer<AutofacBuilder>(
                customizations: customizations =>
                {
                    customizations.ExistingLifetimeScope(container);
                });

            var persistence = endpointConfiguration.UsePersistence<NHibernatePersistence>();

            // configur in code
            var nhConfig = new Configuration();
            nhConfig.SetProperty(Environment.ConnectionProvider,
"NHibernate.Connection.DriverConnectionProvider");
            nhConfig.SetProperty(Environment.ConnectionDriver,
"NHibernate.Driver.Sql2008ClientDriver");
            nhConfig.SetProperty(Environment.Dialect, "NHibernate.Dialect.MsSql2008Dialect");
```

```
            nhConfig.SetProperty(Environment.ConnectionStringName,
"NServiceBus/Persistence");

            persistence.UseConfiguration(nhConfig);

            // NServiceBus will move messages that fail repeatedly to a separate "error"
queue. We recommend
            // that you start with a shared error queue for all your endpoints for easy
integration with ServiceControl.
            endpointConfiguration.SendFailedMessagesTo("error");

            // NServiceBus will store a copy of each successfully process message in a
separate "audit" queue. We recommend
            // that you start with a shared audit queue for all your endpoints for easy
integration with ServiceControl.
            endpointConfiguration.AuditProcessedMessagesTo("audit");

            // turn encryption on
            endpointConfiguration.RijndaelEncryptionService();

            endpointConfiguration.EnableInstallers();

            _endpoint = Endpoint.Start(endpointConfiguration).GetAwaiter().GetResult();

            var updater = new ContainerBuilder();
            updater.RegisterInstance(_endpoint);
            updater.Update(container);

            DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
            AreaRegistration.RegisterAllAreas();
             RegisterRoutes(RouteTable.Routes);
        }

        static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new
                {
                    controller = "Home",
                    action = "Links",
                    id = UrlParameter.Optional
                });
        }

        protected void Application_End()
        {
            _endpoint?.Stop().GetAwaiter().GetResult();
        }
    }
}
```

4. Modify the home controller to look like this:

```
namespace WebApplication1.Controllers
{
    using System.Web.Mvc;

    public class HomeController : Controller
    {
```

```
        public ActionResult Links()
        {
            return View();
        }
    }
}
```

5. 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:

```
namespace WebApplication1.Controllers
{
    using System.Threading.Tasks;
    using System.Web.Mvc;
    using Messages;
    using NServiceBus;

    public class SaySomethingController : AsyncController
    {
        readonly IEndpointInstance _endpoint;

        public SaySomethingController(IEndpointInstance endpoint)
        {
            this._endpoint = endpoint;
        }

        [HttpGet]
        public ActionResult Index()
        {
            ViewBag.Title = "SaySomething";
            return View("Index");
        }

        [HttpPost]
        [AsyncTimeout(50000)]
        public async Task<ActionResult> IndexAsync(string textField)
        {
            int number;

            if (!int.TryParse(textField, out number))
            {
                return View("Index");
            }

            var command = new Request {SaySomething = "Say 'WebApp'." + number};

            await _endpoint.Send(command).ConfigureAwait(false);

            return IndexCompleted();
        }

        public ActionResult IndexCompleted()
        {
            ViewBag.Title = "Say Something";
            return View("Index");
        }
    }
}
```

6. Add a view named index.chtml in the 'views/shared' folder and add the following code to it:

```
@using (Html.BeginForm())
{
    <p>@ViewBag.Title</p>
    <p>
        Enter a number below and click "Go".<br />
    </p>
    <br />
    @Html.TextBox("textField")
    <input type="submit" value="Go" />
    <br /><br />
    @ViewBag.ResponseText

    <br /><br /><br /><br />
    @Html.ActionLink("Home", "Links", "Home")
}
```

7. Add a folder in views named ‘SaySomething’ and add a file named ‘SaySomething.chtml’, add the following code to the file:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>SendLinks</title>
</head>
<body>
    <div>
        <li>@Html.ActionLink("SendAsync - an AsyncController uses NServiceBus",
"SaySomethingSendTask", "SaySomething")</li>
    </div>
</body>
</html>
```

8. Delete the ‘home.chtml’ in the in the ‘vies/home’ folder and add a new file named ‘Links.chtml’, add the following code:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>SendLinks</title>
</head>
<body>
    <div>
        <li>@Html.ActionLink("SendAsync - an AsyncController uses NServiceBus", "Index",
"SaySomething")</li>
    </div>
</body>
</html>
```

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

```
<?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>

  <connectionStrings>
    <add name="NServiceBus/Persistence"
         connectionString="Data
Source=.\SqlExpress;Database=HalloWorld.Persistence;Integrated Security=True"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
...
</configuration>
```

10. Build and run, click the link to fill in the form and send a message, you can see the message being processed at the HalloWorldServer endpoint.

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

Exercise 11. 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 18. Pub Sub

We will now publish an event and add a subscriber to it.

Add an event to the Messages project called SomethingWasSaid

```
namespace Messages
{
    [Event]
    public class SomethingWasSaid
    {
        public Guid Guid { get; set; }
    }
}
```

Add a Guid to the Request message

Add the guid to the snder.

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

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

        public void Start()
        {
            Bus.OutgoingHeaders["user"] = "udi";

            var message = new Request { SaySomething = "Say something", Guid = Guid.NewGuid()
};

            Bus.Send(message);

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

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

Open HelloWorldServer.RequestHandler.cs

Add an IBus Property and a parameter to the constructor

Publish an event in the handler

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

       class RequestHandler : IHandleMessages<Request>
    {
           private readonly ISaySomething saysSomething;
           private readonly IBus bus;

           public RequestHandler(ISaySomething something, IBus bus)
           {
               this.bus = bus;
               saysSomething = something;
           }

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

LogManager.GetLogger("RequestHandler").Info(saysSomething.InResponseTo(message.SaySomething))
;

            bus.Publish<SomethingWasSaid>(e => { e.Guid = message.Guid; });
        }
    }
}
```

Add the event definition in the conventions:

```
namespace Conventions
{
    using System;
    using System.Linq;
    using NServiceBus;

    public class ConventionsConfiguration : INeedInitialization
    {
        public void Customize(BusConfiguration configuration)
        {
            configuration.Conventions()
                .DefiningMessagesAs(type => type.GetCustomAttributes(true).Any(t =>
t.GetType().Name == "MessageAttribute"))
                .DefiningEventsAs(type => type.GetCustomAttributes(true).Any(t =>
t.GetType().Name == "EventAttribute"))
                .DefiningTimeToBeReceivedAs(GetExpiration)
                .DefiningEncryptedPropertiesAs(p => p.GetCustomAttributes(true).Any(t =>
t.GetType().Name == "EncryptedAttribute"));
        }

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

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

Now let's add a subscriber

Create a new project named HelloWorldSubscriber

Install the NServiceBus.Host using nugget

Add a handler

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

    public class SomethingWasSaidHandler : IHandleMessages<SomethingWasSaid>
    {
        public IBus Bus { get; set; }

        public void Handle(SomethingWasSaid message)
        {
            LogManager.GetLogger("SomethingWasSaidHandler")
            .Info("====================================");
            LogManager.GetLogger("SomethingWasSaidHandler")
            .Info(string.Format("Handling SomethingWasSaid event with id: {0}",
       message.Guid));
        }
    }
}
```

Add the message mapping to point at the publisher

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<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>
      <add Messages="Messages" Type="Messages.SomethingWasSaid" Endpoint="HelloWorldServer"
/>
    </MessageEndpointMappings>
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

Make HelloWorld, HelloWorldServer and HelloWorldSubscriber startup projects.

Build and run

Exercise 19. Sagas and Integrations

This material will be distributed in the class and we will walk through the code and demo it.