**Lab 103 – Publish Subscribe using MSMQ** (rev 1.3)

In this type of messaging, the publisher of the message does not know the specifics of those who wish to receive the message. Subscribers explicitly opt in to start receiving events. The subscribers need to know which endpoint is responsible for publishing a message. Messages of significant importance to business are published as events. Since these messages convey a certain something happened, they are named in the passive past tense; e.g., OrderAccepted, OrderShipped, OrderCancelled. Since they have a significant importance, there are often multiple parties interested in when they happen, who subscribe to get that information.

For durability, the publisher stores its list of subscribers in a database. By default this is RavenDB. A different lab exercise teaches how to switch from using RavenDB to a different kind of storage such as SQL Server for the publisher.

Lab Objectives

**In this lab you will learn**

• How to implement the Publish/Subscribe messaging pattern using NServiceBus

• How to set up the transport of your choice for message delivery using MSMQ

• How to define events

• How to publish events on the server

• How to add subscribers for the events that are published

• How to use POCOs for messages and how to load these messages in the NserviceBus endpoints using Unobtrusive conventions.

Lab Prerequisites

This lab already has these components pre-installed:

* Visual Studio 2012
* NServiceBus Infrastructure – DTC
* NServiceBus Infrastructure – MSMQ
* NServiceBus Infrastructure – RavenDB version 2.0.2261.0
* NServiceBus Infrastructure – Performance Counters

**NOTE**: To install NserviceBus infrastructure on your machines, use Powershell commandlets.

Problem Definition

FastCars would like to offer its frequent clients with a reward program. When a client becomes preferred, FastCars offers its customers a free weekend compact car rental, which expires in 30 days.

In this lab, you will define the event schema for the ClientBecamePreferred event. Next you will create a publisher endpoint capable of publishing this event. And finally, you will create a subscriber endpoint which will receive this event, and print a message for the client for the free rental, when the customer becomes preferred. You will implement the solution using MSMQ as the transport for the message exchange.

**Estimated time to complete:**

60 mins

Completed Lab Solution

The complete solution for this exercise that uses NServiceBus version 4.x can be found under C:\Hands on Labs\Completed Solutions\Lab 103 – Publish Subscribe using MSMQ

The complete solution using Version 5.x and this manual is here:

<https://github.com/Particular/HandsOnLabs/tree/master/Lab103>

Terms of Use

By using this Hands-on Lab, you agree to the following terms:

The technology/functionality described in this Hands-on Lab is provided by NServiceBus in a “sandbox” testing environment for purposes of obtaining your feedback and to provide you with a learning experience. You may only use the Hands-on Lab to evaluate such technology features and functionality and provide feedback to NServiceBus. You may not use it for any other purpose. You may not modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this Hands-on Lab or any portion thereof.

COPYING OR REPRODUCTION OF THE HANDS-ON LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS HANDS-ONLAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE. THE TECHNOLOGY/CONCEPTS REPRESENTED IN THIS HANDS-ON LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK. WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS. YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCITONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.

**FEEDBACK**. If you give feedback about the technology features, functionality and/or concepts described in this Hands-on Lab to NServiceBus, you give to NServiceBus, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a NServiceBus software or service that includes the feedback. You will not give feedback that is subject to a license that requires NServiceBus to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

NServiceBus Ltd. HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE HANDS-ON LAB , INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. NServiceBus DOES NOT MAKE ANY ASSURANCES OR REPRESENTATIONS WITH REGARD TO THE ACCURACY OF THE RESULTS, OUTPUT THAT DERIVES FROM USE OF THE VIRTUAL LAB, OR SUITABILITY OF THE INFORMATION CONTAINED IN THE VIRTUAL LAB FOR ANY PURPOSE.

Contents

[Exercise: Publish Subscribe using MSMQ as transport 5](#_Toc405366509)

[Task 1: Create the message schema (FastCars.Events) 5](#_Toc405366510)

[Task 2: Create a publisher endpoint (FastCars.CustomerRelations) and configure to MSMQ as the message transport 5](#_Toc405366511)

[Task 3: Create a subscriber endpoint (FastCars.Promotions) 7](#_Toc405366512)

[Task 4: Run the solution. 9](#_Toc405366513)

[Exercise 2: Durability 9](#_Toc405366514)

[Task1: Bring the subscriber down 9](#_Toc405366515)

[Task2: Publish a few events 9](#_Toc405366516)

[Task3: Restart the subscriber 10](#_Toc405366517)

[Exercise 3: Unobtrusive Conventions 10](#_Toc405366518)

[Task1 – Remove NServiceBus dependency in the FastCars.Events project 10](#_Toc405366519)

[Task2 – Define a common unobtrusive convention to be used for all endpoints 10](#_Toc405366520)

[Task 3 – Change the publisher endpoint to use the new convention 11](#_Toc405366521)

[Task 3 – Change the subscriber endpoint to use the new convention 12](#_Toc405366522)

[Task 4 – Run the solution 12](#_Toc405366523)

# Exercise: Publish Subscribe using MSMQ as transport

## Task 1: Create the message schema (FastCars.Events)

1. Open a new instance of Visual Studio 2012 and make sure to run as an administrator.
2. Open the solution FastCars.CustomerRelations in **C:\Hands on Labs\Exercises\Lab 103 – Publish Subscribe using MSMQ**
3. Install NServiceBus nuget package in the FastCars.Events project. To do this, in the Package Manager Console command prompt (Tools -> Library Package Manager -> Package Manager Console), type:

**Install-package NServiceBus FastCars.Events**

1. Define your event schema for ClientBecamePreferred in ClientBecamePreferred.cs and resolve using statement for the interface IEvent:

namespace FastCars.Events

{

using System;

using NServiceBus;

public class ClientBecamePreferred : IEvent

{

public Guid ClientId { get; set; }

public DateTime PreferredUntil {get;set;}

}

}

1. Make sure the project compiles without any errors.

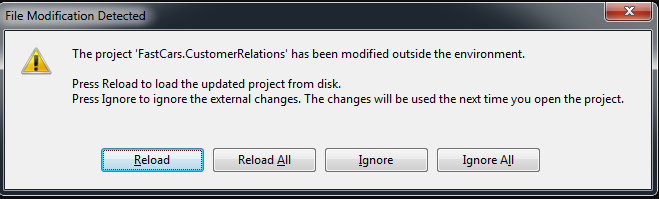
## Task 2: Create a publisher endpoint (FastCars.CustomerRelations) and configure to MSMQ as the message transport

1. In the class library project FastCars.CustomerRelations add a project reference to the FastCars.Events project.
2. Install NServiceBus.Host nuget packagein the FastCars.CustomerRelations project. To do this, **g**o to the Package Manager Console and type:

**Install-package NServiceBus.Host FastCars.CustomerRelations –DependencyVersion HighestMinor**

Including the dependency version switch ensures that the most latest version of NServiceBus core will always be used. Nuget defaults to the LatestPatch, hence the override.

1. NServiceBus automatically adds the proper configuration necessary for the endpoint. When prompted to reload the project, choose Reload All:



1. Starting from NServiceBus version 5, it is essential to pick the Persistence of choice. The generated code is as below:

public class EndpointConfig : IConfigureThisEndpoint

{

public void Customize(BusConfiguration configuration)

{

// NServiceBus provides the following durable storage options

// To use RavenDB, install-package NServiceBus.RavenDB and then use configuration.UsePersistence<RavenDBPersistence>();

// To use SQLServer, install-package NServiceBus.NHibernate and then use configuration.UsePersistence<NHibernatePersistence>();

// If you don't need a durable storage you can also use, configuration.UsePersistence<InMemoryPersistence>();

// more details on persistence can be found here: http://docs.particular.net/nservicebus/persistence-in-nservicebus

//Also note that you can mix and match storages to fit you specific needs.

//http://docs.particular.net/nservicebus/persistence-order

configuration.UsePersistence<PLEASE\_SELECT\_ONE>();

}

}

Since we are using this for development/debugging, we can use InMemoryPersistence. Change the highlighted line to: **configuration.UsePersistence<InMemoryPersistence>();**

1. In the Bootstrapper.cs class modify the code such that every time the ‘Enter’ key is pressed, the ClientBecamePreferred event is published.
   1. To publish events you need the IBus interface. Add a public get setter for the IBus in your class. NServiceBus automatically injects the IBus into this class:

public IBus Bus { get; set; }

* 1. To publish an event every time we press Enter, implement the IWantToRunWhenBusStartsAndStops interface. When you are done, your bootstrapper class might look like this:

namespace FastCars.CustomerRelations

{

using System;

using NServiceBus;

using FastCars.Events;

public class Bootstrapper : IWantToRunWhenBusStartsAndStops

{

public IBus Bus { get; set; }

public void Start()

{

Console.WriteLine("Press Enter to publish an event");

while (Console.ReadLine() != null)

{

Bus.Publish<ClientBecamePreferred>(m =>

{

m.ClientId = Guid.NewGuid();

m.PreferredUntil = DateTime.Today.AddDays(30);

});

Console.WriteLine("Published ClientBecamePreferred event");

}

}

public void Stop()

{

}

}

}

1. Compile your solution and make sure it builds as expected.

## Task 3: Create a subscriber endpoint (FastCars.Promotions)

1. In the project FastCars.Promotions add a project reference to the FastCars.Events project.
2. **Install NServiceBus.Host nuget package:** In Package Manager Console and type

**Install-package NServiceBus.Host FastCars.Promotions –DependencyVersion HighestMinor**

1. NServiceBus automatically adds the proper configuration necessary for the endpoint. When prompted to reload the project, choose Reload All.
2. Select persistence in the EndpointConfig.cs to use InMemoryPersistence. **configuration.UsePersistence<InMemoryPersistence>();**
3. To subscribe to the event, the subscriber needs to explicitly show interest in the event to the publisher responsible for publishing the event. To do this, change the app.config as follows:

In the unicast bus config section, specify two things: the event that is of interest to this subscriber and the endpoint (queue address) responsible for publishing this event:

<?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" />

</configSections>

<MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />

<UnicastBusConfig ForwardReceivedMessagesTo="audit">

<MessageEndpointMappings>

<!--To register a specific type in an assembly -->

<add Assembly="FastCars.Events" Type="FastCars.Events.ClientBecamePreferred" Endpoint="FastCars.CustomerRelations" />

</MessageEndpointMappings>

</UnicastBusConfig>

</configuration>

1. Once the subscriber has expressed interest, the subscriber also needs to implement a message handler that will be invoked when the event is raised. To do this, implement the interface IHandleMessages<ClientBecamePreferred> in ClientBecamePreferredHandler.cs and resolve the using statements.

namespace FastCars.Promotions

{

using System;

using NServiceBus;

using FastCars.Events;

public class ClientBecamePreferredHandler : IHandleMessages<ClientBecamePreferred>

{

public void Handle(ClientBecamePreferred message)

{

Console.WriteLine("Client became preferred, send them a new free rental offer");

}

}

}

1. Compile your solution and make sure it builds as expected.

## Task 4: Run the solution.

1. Run the FastCars.CustomerRelations endpoint first (Right click on the project, Debug and select Start new instance). This will automatically create the needed queues, since this is being run within the Visual Studio Debugger.
2. Next run the FastCars.Promotions endpoint. This endpoint will send a subscription message for the ClientBecamePreferred event to the FastCars.CustomerRelations endpoint:
3. Press the Enter key in FastCars.CustomerRelations and watch the handler on FastCars.Promotions get invoked.

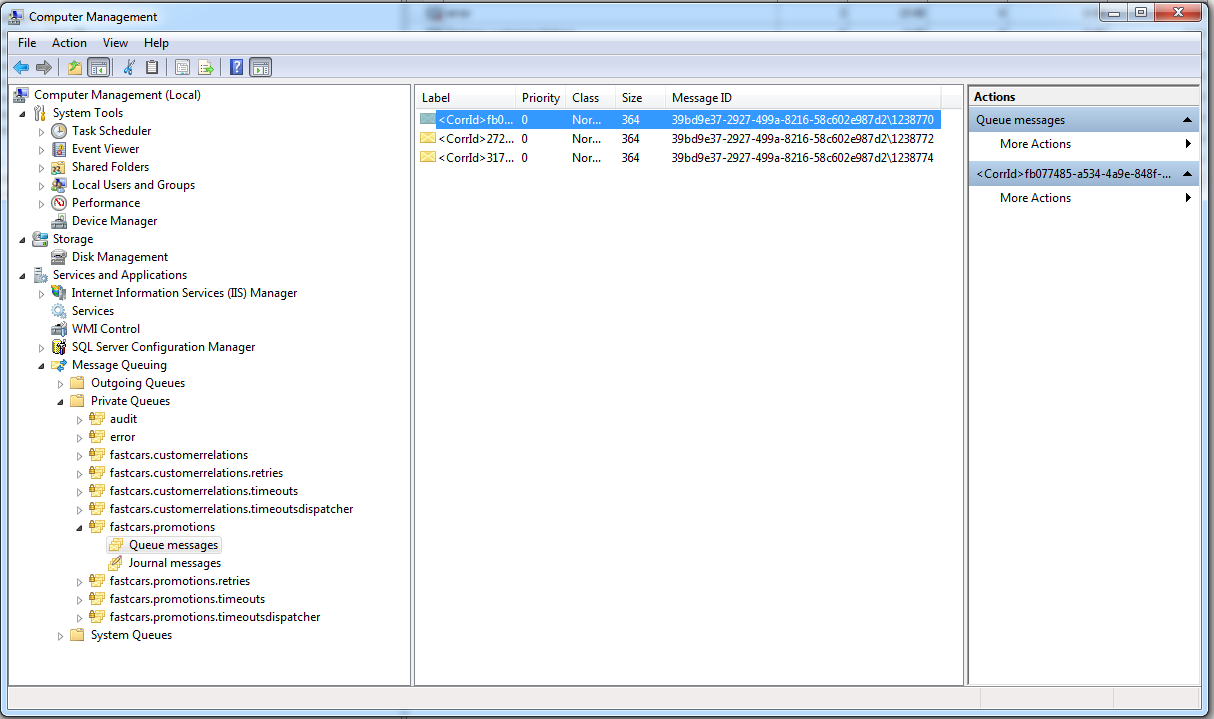
# Exercise 2: Durability

## Task1: Bring the subscriber down

Close the FastCars.Promotions endpoint.

## Task2: Publish a few events

1. Press Enter a few times in the FastCars.CustomerRelations endpoint to publish a few events.
2. View the Queues in the Computer Management for FastCars.Promotions queue.



## Task3: Restart the subscriber

Restart the FastCars.Promotions endpoint and verify that the subscriber now receives all the events that occurred when the subscriber was down.

NOTE: Since we are using InMemory persistence which is volatile, if you bring down the publisher, the publisher’s subscription storage will be reset and when restarted the publisher will not know who the subscribers are. For that reason, a more durable persistence such as SQLServer or RavenDB therefore is advised for use in production.

To use SqlServer: <http://docs.particular.net/nservicebus/relational-persistence-using-nhibernate>

To use RavenDB: <http://docs.particular.net/nservicebus/using-ravendb-in-nservicebus-connecting>

# Exercise 3: Unobtrusive Conventions

In the first exercise, we built the publisher and subscriber endpoints that shared the message schema, but the message schema had a dependency on NServiceBus. In order to use message schema as POCO and remove this dependency on NServiceBus interfaces assembly, NServiceBus offers the unobtrusive conventions. For more details, please see:

<http://docs.particular.net/nservicebus/unobtrusive-mode-messages>

In this exercise, we will convert the Exercise 1 to use unobtrusive conventions instead.

## Task1 – Remove NServiceBus dependency in the FastCars.Events project

1. **Uninstall NServiceBus nuget package:** In the Package Manager Console prompt type:

**uninstall-package NServiceBus FastCars.Events**

1. Remove the IEvent marker interface and the using NServiceBus reference in class ClientBecamePreferred.cs

namespace FastCars.Events

{

using System;

public class ClientBecamePreferred

{

public Guid ClientId { get; set; }

public DateTime PreferredUntil { get; set; }

}

}

1. Make sure that the project builds successfully.

## Task2 – Define a common unobtrusive convention to be used for all endpoints

1. Add a new class library project called **FastCars.SharedConventions** to the solution.
2. **Install NServiceBus nuget package:** Go to the Package Manager Console and type

**Install-package NServiceBus –ProjectName FastCars.SharedConventions**

1. Rename class1.cs to ConventionExtensions.cs and implement an extension method for **BusConfiguration** as follows:

namespace FastCars.SharedConventions

{

using NServiceBus;

public static class ConventionExtensions

{

public static void ApplyCustomConventions(this BusConfiguration busConfiguration)

{

var conventions = busConfiguration.Conventions();

conventions.DefiningCommandsAs(t => t.Namespace != null && t.Namespace.StartsWith("FastCars") && t.Namespace.EndsWith("Commands"));

conventions.DefiningEventsAs(t => t.Namespace != null && t.Namespace.StartsWith("FastCars") && t.Namespace.EndsWith("Events"));

conventions.DefiningMessagesAs(t => t.Namespace != null && t.Namespace.StartsWith("FastCars") && t.Namespace.EndsWith("Messages"));

// In a similar fashion you can define conventions for EncrypedProperties, DataBusProperties, ExpressMessages and TimeToBeRecevied

}

}

}

1. Ensure that the project builds successfully without any errors.

## Task 3 – Change the publisher endpoint to use the new convention

1. In project FastCars.CustomerRelations, add a project reference to FastCars.SharedConventions
2. In the EndpointConfig.cs class, add the configuration for the conventions as follows:

namespace FastCars.CustomerRelations

{

using FastCars.SharedConventions;

using NServiceBus;

public class EndpointConfig : IConfigureThisEndpoint

{

public void Customize(BusConfiguration configuration)

{

configuration.UsePersistence<InMemoryPersistence>();

configuration.ApplyCustomConventions();

}

}

}

## Task 3 – Change the subscriber endpoint to use the new convention

1. In project FastCars.Promotions, add a project reference to FastCars.SharedConventions
2. In the EndpointConfig.cs class, add the configuration for the convention as follows:

namespace FastCars.Promotions

{

using FastCars.SharedConventions;

using NServiceBus;

public class EndpointConfig : IConfigureThisEndpoint

{

public void Customize(BusConfiguration configuration)

{

configuration.UsePersistence<InMemoryPersistence>();

configuration.ApplyCustomConventions();

}

}

}

## Task 4 – Run the solution

1. Clean the solution and Rebuild all the projects in the solution.
2. Start both FastCars.CustomerRelations endpoint and FastCars.Promotions endpoint
3. Publish a few events on the FastCars.CustomerRelations endpoint by pressing Enter.
4. Make sure that the FastCars.Promotions endpoint receives the events.

Congratulations on building a reliable publish/subscribe system using MSMQ as your message transport.