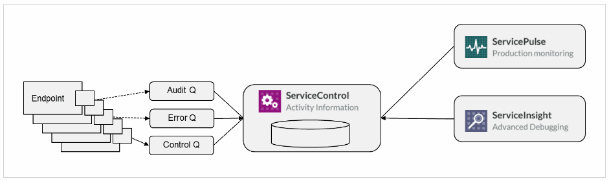
**NServiceBus**

* Takes care of serialization/deserialization of messages.
* Provides a neat model for dispatching messages w. handlers, polymorphic dispatch, arranging handlers in a pipeline etc.
* Gives you a host process that can be F5-debugged as well as installed as a Windows service.

Messaging and workflow framework that helps create distributed systems that are scalable, reliable and easy to modify. It supports various messaging patterns, handles long-running business processes in the form of sagas and provides abstraction over multiple queuing technologies. While most queuing technologies try to make guarantees regarding 'at least once' or even 'exactly once' delivery, they often fall short of this promise. NServiceBus contains mechanisms to automatically solve intermittent delivery problems by retrying messages and falling back to an error queue where they can be exposed by the rest of the Platform for human intervention (ServiceControl, ServicePulse).

The goal of the Platform is to provide a set of tools that make the building and maintenance of messaging systems easier. The tools are tailored to common needs of a messaging system and 'just work' out of the box, enabling developers to focus on other important challenges such as understanding their business domain better. Currently, the Particular Service Platform consists of NServiceBus, ServiceControl, ServiceInsight and ServicePulse.

Moreover, NServiceBus is thoroughly extensible and can be tailored for many requirements. It is compatible with several technologies and many elements of the system can be replaced with custom implementations.

|  |  |
| --- | --- |
|  |  |

Summary

# **Hello World Step by Step in C#**

**Step 1:** Create four project

A Console Application named Client // to send messages with NServiceBus.  
A Console Application named Server //   
A Console Application named Subscriber  
A Class Library named Shared

**Step 2:** In the Client, Server, and Subscriber projects, add a reference to the Shared project.

|  |  |
| --- | --- |
| Add below classes in shared project  public class PlaceOrder : ICommand  {  public Guid Id { get; set; }  public string Product { get; set; }  }  public class OrderPlaced : IEvent  {  public Guid OrderId { get; set; }  } | A **command** is a request to perform a task, which is sent to one specific location. In contrast, an **event** is an announcement that something has happened, which is published from one location and can be consumed by multiple subscribers.  The easiest way to differentiate commands and events is to use the **ICommand** and **IEvent** marker interfaces. |

**Step 3:** Add the NServiceBus NuGet package  
Install the NServiceBus NuGet package in all projects.

Step 4: Add code in Client Project

|  |  |
| --- | --- |
| static void Main()  {  AsyncMain().GetAwaiter().GetResult();  }  static async Task AsyncMain()  {  // This makes it easier to tell console windows apart  Console.Title = "Samples.StepByStep.Client";  // The endpoint name will be used to determine queue names and serves  // as the address, or identity, of the endpoint  var endpointConfiguration = new EndpointConfiguration(  endpoint Name: "Samples.StepByStep.Client");  endpointConfiguration.SendFailedMessagesTo("error");  // Use JSON to serialize and deserialize messages (which are just  // plain classes) to and from message queues  endpointConfiguration.UseSerialization<JsonSerializer>();  // Ask NServiceBus to automatically create message queues  endpointConfiguration.EnableInstallers();  // Store information in memory for this example, rather than in  // a database. In this sample, only subscription information is stored  endpointConfiguration.UsePersistence<InMemoryPersistence>();  // Initialize the endpoint with the finished configuration  var endpointInstance = await Endpoint.Start(endpointConfiguration)  .ConfigureAwait(false);  try  {  await SendOrder(endpointInstance);  }  finally  {  await endpointInstance.Stop()  .ConfigureAwait(false);  }  }  static async Task SendOrder(IEndpointInstance endpointInstance)  {  Console.WriteLine("Press enter to send a message");  Console.WriteLine("Press any key to exit");  while (true)  {  var key = Console.ReadKey();  Console.WriteLine();  if (key.Key != ConsoleKey.Enter)  {  return;  }  var id = Guid.NewGuid();  var placeOrder = new PlaceOrder  {  Product = "New shoes",  Id = id  };  await endpointInstance.Send("Samples.StepByStep.Server", placeOrder);  Console.WriteLine($"Sent a PlaceOrder message with id: {id:N}");  }  } | The Client application must be ready to send messages with NServiceBus. A PlaceOrder command defined in code, and NServiceBus initialized, a loop (marked with yellow) created to send a new command message every time the Enter key is pressed.  The Client is sending the PlaceOrder command to an endpoint named Samples.StepByStep.Server, which will not exist yet. A server endpoint must be created to handle that command. |

Step 5: **Server**

|  |  |
| --- | --- |
| static void Main()  {  AsyncMain().GetAwaiter().GetResult();  }  static async Task AsyncMain()  {  Console.Title = "Samples.StepByStep.Server";  var endpointConfiguration = new EndpointConfiguration("Samples.StepByStep.Server");  endpointConfiguration.UseSerialization<JsonSerializer>();  endpointConfiguration.EnableInstallers();  endpointConfiguration.UsePersistence<InMemoryPersistence>();  endpointConfiguration.SendFailedMessagesTo("error");  var endpointInstance = await Endpoint.Start(endpointConfiguration)  .ConfigureAwait(false);  try  {  Console.WriteLine("Press any key to exit");  Console.ReadKey();  }  finally  {  await endpointInstance.Stop()  .ConfigureAwait(false);  }  }  public class PlaceOrderHandler : IHandleMessages<PlaceOrder>  {  static ILog log = LogManager.GetLogger<PlaceOrderHandler>();  public Task Handle(PlaceOrder message, IMessageHandlerContext context)  {  log.Info($"Order for Product:{message.Product} placed with id: {message.Id}");  log.Info($"Publishing: OrderPlaced for Order Id: {message.Id}");  var orderPlaced = new OrderPlaced  {  OrderId = message.Id  };  return context.Publish(orderPlaced);  }  } | Like the client, the Server application needs to be configured as an NServiceBus endpoint.  In the Server application, add the following code to the Program class:  Create a new class in the Server project named PlaceOrderHandler as highlighted in left side.  PlaceOrderHandler class is the message handler that processes the PlaceOrder command being sent by the Client. A handler is where a message is processed; very often, this will involve saving information from the message into a database, calling a web service, or some other business function.  The handler class is automatically discovered by NServiceBus because it implements the IHandleMessages<T> interface. The dependency injection system (which supports constructor or property injection) injects the IMessageHandlerContext instance into the handler to access messaging operations. When a PlaceOrder command is received, NServiceBus will create a new instance of the PlaceOrderHandler class and invoke the Handle method. |

Step 6: **Subscriber**

|  |  |
| --- | --- |
| static void Main()  {  AsyncMain().GetAwaiter().GetResult();  }  static async Task AsyncMain()  {  Console.Title = "Samples.StepByStep.Subscriber";  var endpointConfiguration = new EndpointConfiguration("Samples.StepByStep.Subscriber");  endpointConfiguration.UseSerialization<JsonSerializer>();  endpointConfiguration.EnableInstallers();  endpointConfiguration.UsePersistence<InMemoryPersistence>();  endpointConfiguration.SendFailedMessagesTo("error");  var endpointInstance = await Endpoint.Start(endpointConfiguration)  .ConfigureAwait(false);  try  {  Console.WriteLine("Press any key to exit");  Console.ReadKey();  }  finally  {  await endpointInstance.Stop()  .ConfigureAwait(false);  }  }  **Message Event**  public class OrderCreatedHandler : IHandleMessages<OrderPlaced>  {  static ILog log = LogManager.GetLogger<OrderCreatedHandler>();  public Task Handle(OrderPlaced message, IMessageHandlerContext context)  {  log.Info($"Handling: OrderPlaced for Order Id: {message.OrderId}");  return Task.CompletedTask;  }  } | Like the client and the server, the Subscriber application also needs to be configured as an NServiceBus endpoint.  In the Subscriber application, add the following code to the Program class:  In fact, all of those activities could be handled by different subscribers to the same event. The fact that these disparate tasks can be accomplished in completely separate message handlers instead of one monolithic process is one of the many strengths of the Publish/Subscribe messaging pattern. Each subscriber is focused on one task, and any failure in one doesn't affect the others.  The subscriber needs to inform the publisher that it wants to receive OrderPlaced events when they are published.  To do that, In the Subscriber application, create an App.config file and add the following XML configuration (rounded with red marker): |
|  | |
| The Assembly and Type values must exactly match the full name (including namespace) for the OrderPlaced event.  The important part is in the MessageEndpointMappings section. The add directive identifies messages of type OrderPlaced within the Shared assembly. The Endpoint attribute determines that subscription requests should be sent to the Samples.StepByStep.Server endpoint. Since that endpoint is responsible for publishing OrderPlaced events, the subscription requests must be directed there as well.  When the Subscriber endpoint initializes, it will read this configuration. Because the endpoint also contains a message handler for OrderPlaced, it will send a special subscription message to the Samples.StepByStep.Server endpoint. When that endpoint receives the subscription request, it will store it locally. In this sample, in-memory storage will be used, but in a production system a database would be used instead. When publishing a message, it can consult the subscriber list and send a copy to every subscriber that expressed interest. | |