Minor WSA docs

A little framework for building Micro Services.

This Framework is intended for developing Micro Services in an Web Scale Architecture and has been designed to work easily with .NET core, RabbitMQ, and Docker.

The micro services communicate with each other through Commands and Events. The Framework sends both Events and Commands over RabbitMQ connections.

The Micro Services are designed to be hosted in docker containers.

Prerequisites

# Micro Services

A micro service is a stand-alone piece of functionality that sends and receives commands and events. Micro services can be developed, versioned, deployed, and scaled independently of each other. This framework offers building blocks for building and hosting such micro services.

In this framework, micro services are intended to be deployed in dedicated docker containers and communicate with each other over RabbitMQ (by means of commands and events). The framework also provides a testing environment

## Buiding and hosting a micro service

For hosting a micro service, one needs a MicroserviceHost. Such host object can be constructed by the use of a MicroserviceHostBuilder. With the MicroserviceHostBuilder, most of the configuration can be done by using conventions. Using a fluent API, one can specify additional or alternative event handlers, command handlers, and configuration (later more on the MicroserviceHostBuilder). Finally a host can be created by calling .CreateHost().

public class Program

{

private static EventWaitHandle waitForEnd = new AutoResetEvent(false);

public static void Main(string[] args)

{

var hostbuilder = new MicroserviceHostBuilder()

.UseConventions();

using (MicroserviceHost host = hostbuilder.CreateHost())

{

host.StartHandling();

host.StartHandling();

waitForEnd.WaitOne(); //keep the application running...

}

}

}

# Events

Events are objects/messages that signal that *something has happened* in the system. That *something* has taken place (in the near past) and is irrevocable. Most events carry all relevant information about the event that took place.

In this framework, events are ‘one-way’ messages – fire and forget. They are not sent to a specific target (or targets) but are simply put on the event bus, for all interested parties to read.

## Define an Event

All events must derive from the base type Minor.WSA.Common.DomainEvent.

public class ProductAdded : DomainEvent

{

public ProductAdded() : base("some.routingkey") { }

public Product Product { get; set; }

}

The name of the event is in the present perfect tense. This is to reflect the fact that the event has already occurred.

The routing key indicates the name of the topic to which the event is published.

## Publishing Events

Events are published by the EventPublisher.

using (IEventPublisher publisher = new EventPublisher(busOptions))

{

ProductAdded productAddedEvent = new ProductAdded();

publisher.Publish(productAddedEvent);

}

.

## Receiving Events

The events are received by a micro service. The micro service should have *event handlers* that reside in *event listeners*. These event listeners must be registered with the micro service host.

### Define an event listeners

A micro service handles events in event listeners. An event listener is any class that is attributed with an EventListener Attibute.

[EventListener("queuename")]

public class SomeEventListener

{

}

The EventListenerAttribute specifies the name of the RabbitMQ queue on which this event listener will listen for incoming events. This queue name is mandatory. The RabbitMQ-Queue is created (declared) if it does not yet exist. It is then bound to the RabbitMQ-Exchange that has been specified in (the busOptions of) the MicroServiceHost.

### Register an event listener

The event listener is hooked up to the Micro Service Host by the MicroserviceHostBuilder, either implcitly or explicitly.

#### Register an event listener implicitly

To implicitly register the event listener(s), use the UseConventions() method of the MicroserviceHostBuilder, which will find event listeners, using reflection, in the assembly in which this builder code resides. All classes that have a EventListenerAttribute are registered as being a micro service event listener.

var hostbuilder =

new MicroserviceHostBuilder()

.UseConventions(); // Find Listeners and Controllers using reflection

#### Register an event listener explicitly

To explicitly register the event listener(s), use the AddEventListener<T>() method of the MicroserviceHostBuilder, where T is the type of the event listener:

var hostbuilder =

new MicroserviceHostBuilder()

. AddEventListener<SomeEventHandler>() // Explicitly add Event Listener

### Define a Handle-method

For each event separately, an event handler method has to be defined in the event listener. A event handling method is a method in an event listeners class that has exactly one parameter of a type that inherits from Minor.WSA.Common.DomainEvent. Optionally, in the Topic-Attribute, a topic expression (=RabbitMQ topic) can be specified. All events that have a Routingkey that matches the

Topic expression will be handled by this event handler method. That means that the same event can be handled by more than one event handler method. Only if ALL event handler methods succeed, i.e. none of them throws an exception, the event will be taken from the queue.

[EventListener("queuename")]

internal class AnotherEventHandler

{

public void Handle(SomeEvent evt)

{

}

[Topic("WSA.Test.OtherEvent")]

public void Handle(OtherEvent evt)

{

}

[Topic("\*.\*.OtherEvent")]

public void HandleToo(OtherEvent evt)

{

}

}

.

# MicroserviceHostBuilder

The MicroserviceHostBuilder is used for building and configuring MicroserviceHosts. With it, one can configure the eventbus, add event listeners, and/or configure Dependency Injection.

var builder = new MicroserviceHostBuilder()

.UseConventions()

.WithBusOptions(options) // Configure bus options for RabbitMQ

.AddEventListener<T>(); // Add extra event listeners

// Configure Dependency Injection

Builder.ServiceProvider.AddTransient<IService,Service>();

// Create MicroserviceHost

using (var host = builder.CreateHost())

{

// ...

}

.

# TestBus

The MicroserviceHosts can be configured to use a TestBus, instead of the RabbitMQ bus. With it, o**n**e

THE STUFF BELOW HAS NOT YET BEEN IMPLEMENTED..

# Commands

Commands are objects that represent something that has to be done. In contrast to *events*, commands can fail. They can be rejected by a micro service if the command does not comply with some business rule, or if it conflicts with the current state of the micro service.

Commands always send an acknowledgement back to the sender of the command, with at least information whether the command has succeeded or has failed. It can also return additional data. In this framework, the acknowledgement is always sent asynchronously.

## Handling Commands

The commands, that are received by a micro service, are handled by *command handlers* that reside in *controllers*. These controllers must be registered with the micro service host.

### Define a Controller

A Micro Service handles commands in *controllers*. A controller is any class that is attributed with a Microservice attribute.

[Microservice("Endpointname")]

public class ProductServiceController

{

}

The MicroserviceAttribute specifies the name of the RabbitMQ queue on which the Micro Service will listen for incoming commands. If no endpoint name has been specified, the Micro Service will listen to the RabbitMQ queue with the same name as the controller class (without the ‘Controller’-suffix).

[Microservice()]

// Endpointname defaults to "ProductService", from <<endpointname>>Controller

public class ProductServiceController

{

}

In this example, the name of the queue defaults to ‘ProductService’.

A Micro Service can listen to multiple queues, by defining more than one controller.

WHAT IF MULTIPLE CONTROLLERS LISTEN TO THE SAME QUEUE ???

### Register a Controller

The controller is hooked up to the Micro Service Host by the MicroserviceHostBuilder, either implcitly or explicitly.

#### Register a Controller implicitly

To implicitly register the controller(s), use the UseConventions() method of the MicroserviceHostBuilder, which will find controllers, using reflection, in the assembly in which this builder code resides. All classes that have a MicroserviceAttribute are registered as being a Micro Service command controller.

var hostbuilder =

new MicroserviceHostBuilder()

.UseConventions(); // Find Handlers and Controllers using reflection

#### Register a Controller explicitly

To explicitly register the controller(s), use the AddController<T>() method of the MicroserviceHostBuilder, where T is the type of the controller:

var hostbuilder =

new MicroserviceHostBuilder()

.AddController<ProductServiceController>() // Explicitly add Controller

### Handle Commands

For each command, a command handler has to be defined in the controller. A command handling method is a method in a controller class that has been attributed with a HandlesAttribute. The command handling method must have a single parameter. This parameter specifies the command type that is to be handled by this method. Optionally, in the HandlesAttribute, the full name of a different command type can be specified. The received data of the alternate command type will still be serialized into the parameter type of the method.

[Microservice("Endpointname")]

public class ProductServiceController : IProductService

{

[Handles("Command.Type.Full.Name")]

public void AddProductToInventory(AddProductToInventoryCommand command)

{

}

}

In this example, the ProductServiceController listens to the queue with the name "Endpointname". If a message appears in this queue that is of type CommandTypeFullName, then the AddProductToInventory-method is called by the framework, passing it a AddProductToInventoryCommand that has been populated with the data that was found in the CommandTypeFullName-message.

## Sending Commands

Configuration