Paramore Documentation

Release 7.0.0

Ian Cooper

Jan 18, 2019

Contents

Libraries and supporting examples for use with the Ports and Adapters and CQRS architectural styles for .NET, with support for Task Queues.

View the Project on GitHub

CHAPTER 1

Brighter

How to Implement a Request Handler

To implement a handler, derive from RequestHandler<T> where T should be the **Command** or **Event** derived type that you wish to handle. Then override the base class Handle() method to implement your handling for the Command or Event.

For example, assume that you want to handle the **Command** GreetingCommand

```
public class GreetingCommand : IRequest
{
    public GreetingCommand(string name)
    {
        Id = Guid.NewGuid();
        Name = name;
    }

    public Guid Id { get; set; }
    public string Name { get; private set; }
}
```

Then derive your handler from **RequestHandler<GreetingCommand>** and accept a parameter of that type on the overriden **Handle()** method.

```
public class GreetingCommandHandler : RequestHandler<GreetingCommand>
{
    public override GreetingCommand Handle(GreetingCommand command)
    {
        Console.WriteLine("Hello {0}", command.Name);
        return base.Handle(command);
    }
}
```

What is the difference between a Command and an Event?

We use the term **Request** for a data object containing parameters that you want to dispatch to a handler. Brighter uses the interface **IRequest** for this concept. Confusingly, both **Command** or **Event** which implement **IRequest** are examples of the Command Pattern. It is easiest to say that within Brighter **IRequest** is the abstraction that represents the Command from the Command Pattern.

Why have both **Command** and **Event**? The difference is in how the **Command Dispatcher** dispatches them to handlers.

- A **Command** is an imperative instruction to do something; it only has one handler. We will throw an error for multiple registered handlers of a command.
- An **Event** is a notification that something has happened; it has zero or more handlers.

The difference is best explained by the following analogy. If I say "Bob, make me a cup of coffee," I am giving a Command, an imperative instruction. My expectation is that Bob will make me coffee. If Bob does not, then we have a failure condition (and I am thirsty and cranky). If I say "I could do with a cup of coffee," then I am indicating a state of thirst and caffeine-withdrawal. If Bob or Alice make me a coffee I will be very grateful, but there is no expectation that they will.

So choosing between **Command** or **Event** effects how the **Command Dispatcher** routes requests.

See Dispatching a Request for more on how to dispatch **Requests** to handlers.

Dispatching Requests

Once you have implemented your Request Handler, you will want to dispatch **Commands** or **Events** to that Handler.

Registering a Handler

In order for a **Command Dispatcher** to find a Handler for your **Command** or **Event** you need to register the association between that **Command** or **Event** and your Handler.

The **Subscriber Registry** is where you register your Handlers.

```
var subscriberRegistry = new SubscriberRegistry();
subscriberRegistry.Register<GreetingCommand, GreetingCommandHandler>();
```

Dispatching Requests

Once you have registered your Handlers, you can dispatch requests to them. To do that you simply use the **CommandProcessor.Send()** method passing in an instance of your command.

```
commandProcessor.Send(new GreetingCommand("Ian"));
```

Building a Command Dispatcher

We associate a **Subscriber Registry** with a **Command Processor** by passing it into the constructor of the **Command Processor**. For convenience, we provide a **Commmand Processor Builder** that helps you configure new instances of **Command Processor**.

```
var logger = LogProvider.For<Program>();

var registry = new SubscriberRegistry();
registry.Register<GreetingCommand, GreetingCommandHandler>();


var builder = CommandProcessorBuilder.With()
    .Handlers(new HandlerConfiguration(
        subscriberRegistry: registry,
        handlerFactory: new SimpleHandlerFactory(logger)
    ))
    .DefaultPolicy()
    .NoTaskQueues()
    .RequestContextFactory(new InMemoryRequestContextFactory());

var commandProcessor = builder.Build();
```

We cover configuration of a **Command Processor** in more detail later.

Returning results to the caller.

We use Command-Query separation so a Command does not have return value and **CommandDispatcher.Send()** does not return anything.

This in turn leads to a set of questions that we need to answer about common scenarios:

- How do I handle failure? With no return value, what do I do if my handler fails
- How do I pass information back to the caller? Creation scenarios particularly seem to require the caller knows about identies for created entities.

We discuss these issues below.

Handling Failure

If we don't allow return values, what do you do on failure?

- The basic failure strategy is to throw an exception. This will terminate the request handling pipeline.
- If you want to support Retry, and Circuit Breaker you can use our support for Polly Policies
- You can also build your own exception handling into your Pipeline.
- Finally you can use our support for a Fallback handler to provide backstop exception handling.

Passing Information to the Caller

Sometimes you need to provide information to the caller about the success of the operation. The most common requirement is the Identity of a new created Entity so that you can query for it. For example you are implementing a REST API and in response to a POST request you create a new entity and want to return the entity body in the HTTP response body.

The best approach is to generate the Identity to use for the new Entity and pass that as a parameter on the **Command**, such as Guid or Hi-Lo identity.

But what if you are not be able to do this or want to support it for performance reasons?

In that case add a property to the **Command** that you can initialize from the Handler, for example create a **NewEntityIdentity** property in your command that you write the new entity's identity to in the Handler, and then inspect the property in your **Command** in the calling code after the call to **CommandDispatcher.Send()** completes.

Note that you cannot use this strategy with **CommandDispatcher.Send()** as you have no way to update the **Command** in process.

Using the base class when dispatching a message

All **Command** or **Event** messages derive from **IRequest** and **ICommand** and **IEvent** respectively. So it may seem natural to create a collection of them, for example **List<IRequest>**, and then process a set of messages by enumerating over them.

When you try this, you will encounter the issue that we dispatch based on the concrete type of the **Command** or **Event**. In other words the type you register via the **SubscriberRegistry.** Because **CommandProcessor.Send()** is actually **CommandProcessor.Send<T>()** you need to provide the concrete type in the call for the compiler to determine the type to use with the cool as the concrete type.

If you try this:

```
ICommand command = new GreetingCommand("Ian");
commandProcessor.Send(command);
```

Then you will get this error: *"ArgumentException "No command handler was found for the typeof command Brighter.commandprocessor.ICommand - a command should have exactly one handler.""*

Now, you don't see this issue if you pass the concrete type in, so the compiler can correctly resolve the run-time type.

```
commandProcessor.Send(new GreetingCommand("Ian"));
```

So what can you do if you must pass the base class to the **Command Processor** i.e. because you are using a list.

The workaround is to use the dynamic keyword. Using the dynamic keyword means that the type will be evaluated using RTTI, which will successfully pick up the type that you need.

```
ICommand command = new GreetingCommand("Ian");
commandProcessor.Send((dynamic)command);
```

See this discussion for more.

Building a Pipeline of Request Handlers

Once you are using the features of Brighter to act as a command dispatcher and send or publish messages to a target handler, you may want to use its command processor features to handle orthogonal operations.

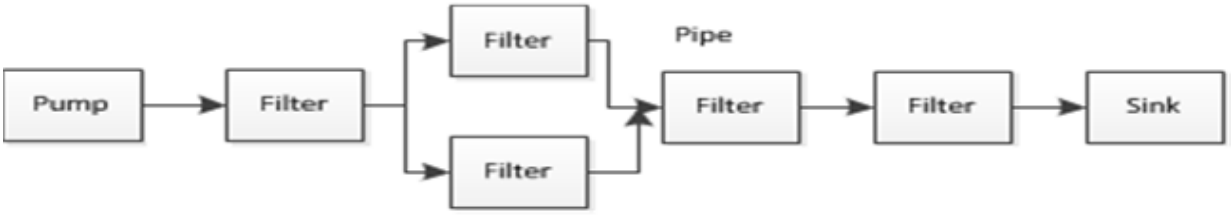
Common examples of orthogonal operations include:

- Logging the Command
- Providing integration with tools for monitoring performance and availability
- Validating the Command
- Supporting idempotency of messages
- Supporting re-sequencing of messages
- Handling exceptions

- Providing Timeout, Retry, and Circuit Breaker support
- Providing undo support, or rollback

The Pipes and Filters Architectural Style

To handle these orthogonal concerns our command processor uses a pipes and filters architectural style: the filters are where processing occurs, they do not share state with other filters, nor do they know about adjacent filters. The pipe is the connector between the filters in our case this is provided by the **IHandleRequests<TRequest>** interface which has a method **IHandleRequests<TRequest> Successor** that allows us to chain filters together.

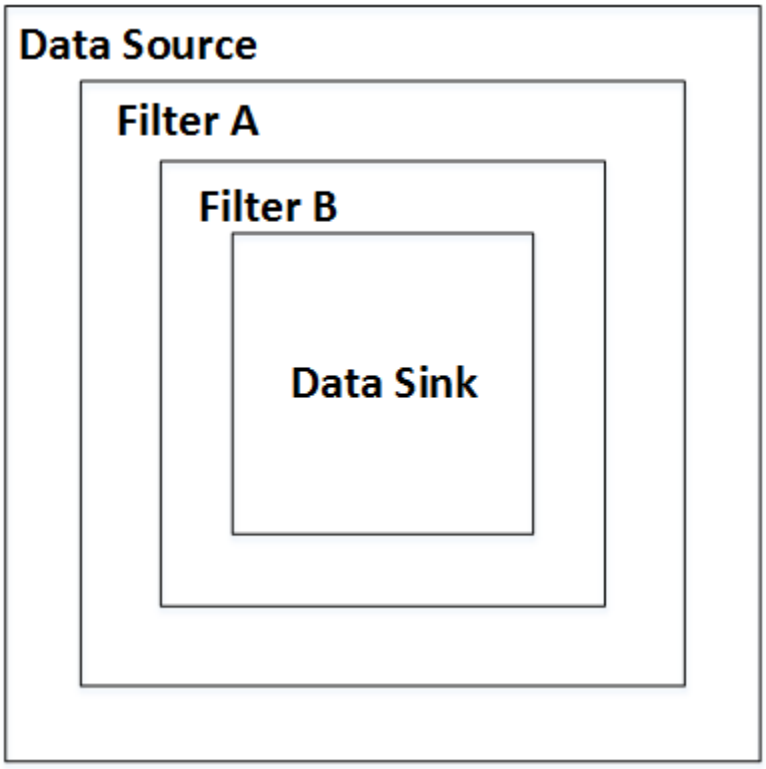


The sink handler is handler that is the receiver you wish to invoke the action on. The pump is the **Command Dispatcher**. We occasionally use *target handler* as a synonym for *sink handler*

The Russian Doll Model

Our pipes and filters approach supports the *Russian Doll Model* of calling the handler pipeline, a context bag for the pipeline, and support for generating a request path description out-of-the-box.

The *Russian Doll Model* is names for the Matryoshka wooden dolls, in which dolls of decreasing sizes are nested one inside another. The importance of this for a pipes and filters pattern style is that each filter in the pipeline is called within the scope of a previous filter in the pipeline.



This is significant because you may desire to act before and after a subsequent filter step. One particular use case is exception handling: a try-catch block that wraps the call to a subsequent step can react to exceptions raised by subsequent steps. This allows us to create policy decisions around exceptions using a library such as Polly and thus support Retry and Circuit Breaker

Our usage of the Russian Doll Model was inspired by FubuMVC

Implementing a Pipeline

The first step in building a pipeline is to decide that we want an orthogonal operation in our pipeline. Let us assume that we want to do basic request logging.

Because you do not want to write an orthogonal handler for every Command or Event type, these handlers should remain generic types. At runtime the HandlerFactory creates an instance of the generic type specialized for the type parameter of the Command or Event being passed along the pipeline.

The limitation here is that you can only make assumptions about the type you receive into the pipeline from the constraints on the generic type.

Although it is possible to implement the IHandleRequests interface directly, we recommend deriving your handler from RequestHandler<T>.

Let us assume that we want to log all requests travelling through the pipeline. (We provide this for you in the Brighter.CommandProcessor packages so this for illustration only). We could implement a generic handler as follows:

```
using System;
using Newtonsoft.Json;
using Brighter.commandprocessor.Logging;
```

```
namespace Brighter.commandprocessor
{
    public class RequestLoggingHandler<TRequest>
        : RequestHandler<TRequest> where TRequest : class, IRequest
    {
        private HandlerTiming _timing;

        public override void InitializeFromAttributeParams(
            params object[] initializerList
        )
        {
            _timing = (HandlerTiming)initializerList[0];
        }

        public override TRequest Handle(TRequest command)
        {
            LogCommand(command);
            return base.Handle(command);
        }

        private void LogCommand(TRequest request)
        {
            logger.InfoFormat("Logging handler pipeline call. Pipeline timing {0}␣
↪target, for {1} with values of {2} at: {3}",
                _timing.ToString(),
                typeof(TRequest),
                JsonConvert.SerializeObject(request),
                DateTime.UtcNow);
        }
    }
}
```

Our Handle method is the method which will be called by the pipeline to service the request. After we log we call **return base.Handle(command)** to ensure that the next handler in the chain is called. If we failed to do this, the *target handler* would not be called nor any subsequent handlers in the chain. This call to the next item in the chain is how we support the 'Russian Doll' model - because the next handler is called within the scope of this handler, we can manage when it is called handle exceptions, units of work, etc.

It is worth remembering that handlers may be called after the target handler (in essence you can designate an orthogonal handler as the sink handler when configuring your pipeline). For this reason ****all**** handlers should remember to call their successor, **even **your** target handler**.

We now need to tell our pipeline to call this orthogonal handler before our target handler. To do this we use attributes. The code we want to write looks like this:

```
class GreetingCommandHandler : RequestHandler<GreetingCommand>
{
    [RequestLogging(step: 1, timing: HandlerTiming.Before)]
    public override GreetingCommand Handle(GreetingCommand command)
    {
        Console.WriteLine("Hello {0}", command.Name);
        return base.Handle(command);
    }
}
```

The **RequestLogging** Attribute tells the Command Processor to insert a Logging handler into the request handling pipeline before (**HandlerTiming.Before**) we run the target handler. It tells the Command Processor that we want it to be the first handler to run if we have multiple orthogonal handlers i.e. attributes (**step: 1**).

We implement the **RequestLoggingAttribute** by creating our own Attribute class, derived from **RequestHandlerAttribute**.

```
public class RequestLoggingAttribute : RequestHandlerAttribute
{
    public RequestLoggingAttribute(int step, HandlerTiming timing)
        : base(step, timing)
        { }

    public override object[] InitializerParams()
    {
        return new object[] { Timing };
    }

    public override Type GetHandlerType()
    {
        return typeof(RequestLoggingHandler<>);
    }
}
```

The most important part of this implementation is the GetHandlerType() method, where we return the type of our handler. At runtime the Command Processor uses reflection to determine what attributes are on the target handler and requests an instance of that type from the user-supplied **Handler Factory**.

Your Handler Factory needs to respond to requests for instances of a **RequestHandler<T>** specialized for a concrete type. For example, if you create a **RequestLoggingHandler<TRequest>** we will ask you for a **RequestLoggingHandler<MyCommand>** etc. Depending on your implementation of HandlerFactory, you may need to register an implementation for every concrete instance of your handler with your underlying IoC container etc.

Note that as we rely on an user supplied implementation of **IAmAHandlerFactory** to instantiate Handlers, you can have any dependencies in the constructor of your handler that you can resolve at runtime. In this case we pass in an ILog reference to actually log to.

You may wish to pass parameter from your Attribute to the handler. Attributes can have constructor parameters or public members that you can set when adding the Attribute to a target method. These can only be compile time constants, see the documentation here. After the Command Processor calls your Handler Factory to create an instance of your type it calls the **RequestHandler.InitializeFromAttributeParams** method on that created type and passes it the object array defined in the **RequestHandlerAttribute.InitializerParams**. By this approach, you can pass parameters to the handler, for example the Timing parameter is passed to the handler above.

It is worth noting that you are limited when using Attributes to provide constructor values that are compile time constants, you cannot pass dynamic information. To put it another way you are limited to value set at design time not at run time.

In fact, you can use this approach to pass any data to the handler on initialization, not just attribute constructor or property values, but you are constrained to what you can access from the context of the Attribute at run time. it can be tempting to set retrieve global state via the Service Locator pattern at this point. Avoid that temptation as it creates coupling between your Attribute and global state reducing modifiability.

Using a Manual Approach

Using an attribute based approach is not an approach favoured by everyone. Some people prefer a more explicit approach to configuring the pipeline.

This is possible, we just don't provide any help out-of-the-box. Although see this issue for a placeholder to fix that.

The trick is to remember that any handler that derives from **IHandleRequests<TRequest>** has a **Successor** and you can build a chain by having the first handler call the second handler's **Handle()** method i.e. **Successor.Handle()**. You can derive from **RequestHandler<T>** and call **base.Handle()** for this, even if you don't want to use the Attribute based pipelines.

In the SubscriberRegistry you just register the first Handler in your pipeline. When we lookup the Handler for the Command in the SubscriberRegistry we will call it's Handle method. It can execute your code, and then call it's Successor (using the Russian Doll approach).

```
var myCommandHandler = new MyCommandHandler();
var myLoggingHandler = new MyLoggingHandler(log);

myLoggingHandler.Successor = myCommandHandler;

var subscriberRegistry = new SubscriberRegistry();
subscriberRegistry.Register<MyCommand, MyLoggingHandler>();
```

It is worth noting that as you control the HandlerFactory, you could also register the sink handler, but when instantiating an instance of it on request, build the pipeline of handlers yourself.

We think it is easier to use attributes, but there may be circumstances where that approach does not work, and so this option is supported as well.

Passing information between Handlers in the Pipeline

A key constraint of the Pipes and Filters architectural style is that Filters do not share state. One reason is that this limits your ability to recompose the pipeline as steps must follow other steps.

However, when dealing with Handlers that implement orthogonal concerns it can be useful to pass context along the chain. Given that many orthogonal concerns have constraints about ordering anyway, we can live with the ordering constraints imposed by passing context. So how do you approach passing context from one Handler to another when it is necessary?

The first thing is to avoid adding extra properties to the Command to support handling state for these orthogonal Filter steps in your pipeline. This couples your **Command** to orthogonal concerns and you really only want to bind it to your **Target Handler**.

Instead we provide a **Context Bag** as part of the Command Dispatcher which is injected into each Handler in the Pipeline. The lifetime of this **Context Bag** is the lifetime of the Request (although you will need to take responsibility for freeing any unmanaged resources you place into the **Context Bag** for example when code called after the Handler that inserts the resource into the Bag returns to the Handler).

```
public class MyContextAwareCommandHandler : RequestHandler<MyCommand>
{
    public static string TestString { get; set; }

    public override MyCommand Handle(MyCommand command)
    {
        LogContext();
        return base.Handle(command);
    }

    private void LogContext()
    {
        TestString = (string)Context.Bag["TestString"];
        Context.Bag["MyContextAwareCommandHandler"] = "I was called and set the␣
↪context";
```

```
    }
}
```

Internally we use the **Context Bag** in a number of the Quality of Service supporting Attributes we provide. See Fallback for example.

Supporting Retry and Circuit Breaker

Brighter is a **'Command Processor <https://brightercommand.github.io/Brighter/ControlBus.html'__** and supports a pipeline of Handlers to handle orthogonal requests.

Amongst the valuable uses of orthogonal requests is patterns to support Quality of Service in a distributed environment: Timeout, Retry, and Circuit Breaker.

Even if you don't believe that you are writing a distributed system that needs this protection, consider that as soon as you have multiple processes, such as a database server, you are.

Brighter uses Polly to support Retry and Circuit-Breaker. Through our Russian Doll Model we are able to run the target handler in the context of a Policy Handler, that catches exceptions, and applies a Policy on how to deal with them.

Using Brighter's UsePolicy Attribute

By adding the **UsePolicy** attribute, you instruct the Command Processor to insert a handler (filter) into the pipeline that runs all later steps using that Polly policy.

```
internal class MyQoSProtectedHandler : RequestHandler<MyCommand>
{
    static MyQoSProtectedHandler()
    {
        ReceivedCommand = false;
    }

    [UsePolicy(policy: "MyExceptionPolicy", step: 1)]
    public override MyCommand Handle(MyCommand command)
    {
        /*Do work that could throw error because of distributed computing␣
↪reliability*/
    }
}
```

To configure the Polly policy you use the PolicyRegistry to register the Polly Policy with a name. At runtime we look up that Policy by name.

```
var policyRegistry = new PolicyRegistry();

var policy = Policy
    .Handle<Exception>()
    .WaitAndRetry(new[]
    {
        1.Seconds(),
        2.Seconds(),
        3.Seconds()
    }, (exception, timeSpan) =>
```

```
    {
        s_retryCount++;
    });

policyRegistry.Add("MyExceptionPolicy", policy);
```

When creating policies, refer to the Polly documentation.

Whilst **Polly** does not support a Policy that is both Circuit Breaker and Retry i.e. retry n times with an interval between each retry, and then break circuit, to implement that simply put a Circuit Breaker UsePolicy attribute as an earlier step than the Retry UsePolicy attribute. If retries expire, the exception will bubble out to the Circuit Breaker.

Retry and Circuit Breaker with Task Queues

When posting a request to a Task Queue we mandate use of a Polly policy to control Retry and Circuit Breaker in case the output channel is not available. These are configured using the constants: **Paramore.RETRYPOLICY** and **Paramore.CIRCUITBREAKER**

Timeout

You should not allow a handler that calls out to another process (e.g. a call to a Database, queue, or an API) to run without a timeout. If the process has failed, you will consumer a resource in your application polling that resource. This can cause your application to fail because another process failed.

Usually the client library you are using will have a timeout value that you can set.

In some scenarios the client library does not provide a timeout, so you have no way to abort.

We provide the Timeout attribute for that circumstance. You can apply it to a Handler to force that Handler into a thread which we will timeout, if it does not complete within the required time period.

```
public class EditTaskCommandHandler : RequestHandler<EditTaskCommand>
{
    private readonly ITasksDAO _tasksDAO;

    public EditTaskCommandHandler(ITasksDAO tasksDAO)
    {
        _tasksDAO = tasksDAO;
    }

    [RequestLogging(step: 1, timing: HandlerTiming.Before)]
    [Validation(step: 2, timing: HandlerTiming.Before)]
    [TimeoutPolicy(step: 3, milliseconds: 300)]
    public override EditTaskCommand Handle(EditTaskCommand editTaskCommand)
    {
        using (var scope = _tasksDAO.BeginTransaction())
        {
            Task task = _tasksDAO.FindById(editTaskCommand.TaskId);

            task.TaskName = editTaskCommand.TaskName;
            task.TaskDescription = editTaskCommand.TaskDescription;
            task.DueDate = editTaskCommand.TaskDueDate;

            _tasksDAO.Update(task);
            scope.Commit();
```

```
        }

        return editTaskCommand;
    }
}
```

Failure and Fallback

If your **RequestHandler.Handle()** call fails you have a number of options:

- The basic failure strategy is to throw an exception. This will terminate the request handling pipeline.
- If you want to support Retry, and Circuit Breaker you can use our support for Polly Policies
- You can also build your own exception handling into your Pipeline.

If none of these strategies succeed, you may want some sort of backstop exception handler, that allows you to take compensating action, such as undoing any partially committed work, issuing a compensating transaction, or queuing work for later delivery (perhaps using the Task Queue).

To support this we provide a **IHandleRequests<TRequest>Fallback** method. In the Fallback method you write your code to run in the event of failure.

Calling the Fallback Pipeline

We provide a **FallbackPolicy** Attribute that you can use on your **IHandleRequests<TRequest>.Handle()** method. The implementation of the **Fallback Policy Handler** is straightforward: it creates a backstop exception handler by encompassing later requests in the Request Handling Pipeline in a try...catch block. You can configure it to catch all exceptions, or just Broken Circuit Exceptions when a Circuit Breaker has tripped.

When the **Fallback Policy Handler** catches an exception it calls the **IHandleRequests<TRequest>.Fallback()** method of the next Handler in the pipeline, as determined by **IHandleRequests<TRequest>.Successor**

The implementation of **RequestHandler<T>.Fallback()** uses the same Russian Doll approach as it uses for **RequestHandler<T>.Handle()**. This means that the request to take compensating action for failure, flows through the same pipeline as the request for service, allowing each Handler in the chain to contribute.

In addition the **Fallback Policy Handler** makes the originating exception available to subsequent Handlers using the **Context Bag** with the key: **CAUSE_OF_FALLBACK_EXCEPTION**

Using the FallbackPolicy Attribute

The following example shows a Handler with **Request Handler Attributes** for Retry and Circuit Breaker policies that is configured with a **Fallback Policy** which catches a **Broken Circuit Exception** (raised when the Circuit Breaker is tripped) and initiates the Fallback chain.

```
public class MyFallbackProtectedHandler: RequestHandler<MyCommand>
{
    [FallbackPolicy(backstop: false, circuitBreaker: true, step: 1)]
    [UsePolicy("MyCircuitBreakerStrategy", step: 2)]
    [UsePolicy("MyRetryStrategy", step: 3)]
    public override MyCommand Handle(MyCommand command)
    {
        /*Do some work that can fail*/
    }
```

```
    public override MyCommand Fallback(MyCommand command)
    {
        if (Context.Bag.ContainsKey(FallbackPolicyHandler<MyCommand>.CAUSE_OF_
↪FALLBACK_EXCEPTION))
        {
            /*Use fallback information to determine what action to take*/
        }
        return base.Fallback(command);
    }
}
```

Event Sourcing

If we dispatch commands to a target handler, and have a pipeline which acts as a preprocessor one obvious orthogonal operation is to log our commands so that we can understand the commands that result in current system state. We can examine the logs, particularly if there is a problem, to understand what commands were sent to the application to create the current state.

We can also infer that if the current application state is a function of those commands, then we could potentially recreate the same application state by replaying those commands.

Now, within a log file, we are going to have to fiddle with sed or awk to pull out the commands into a text file and then run those through something that reposts them. That works but it can be a little awkward and as such is a barrier to entry.

So if we stored these to a Command Store it could be much easier to slice and dice the data, and to extract it for replay.

In his bliki on Event Sourcing Martin Fowler describes using an architecture that "guarantee[s] that all changes to the domain objects are initiated by the event objects" and one implementation approach is that an event processor sequentially logs the event which is then applied to the domain. The system of record can be either the events (perhaps rebuilt overnight) or application state (in which case the events are only used for analysis or recovery.

Brighter has an event processor equivalent, as it is a command processor and dispatcher, so it only needs to persist the commands to enable support for Event Sourcing. So if you make all your changes to the domain through a command dispatcher such as Brighter you can meet the requirements for Event Sourcing by persisting your commands in a way that facilitates querying or replay. As Brighter has a pipeline through it's command processor it is natural to simply add an attribute to the target handler that persists the command before it is applied to domain model. This is essentially a write-ahead log.

Martin lists some issues to consider: new features, defect fixes, and temporal logic. A particular issue is external gateways.

Command or Event Sourcing

One complaint about Martin's article is that a Command is the intent to change the system, but an event is the result of that change. Because this model records the change to state many people prefer to refer to this approach as Command Sourcing and reserve Event Sourcing for Greg Young's related idea of storing the results of those commands, the changes that would be made to application state, so that those changes can be replayed instead of the commands, that could have side affects. We don't explicitly provide help for that approach.

For clarity we will use the term Command Sourcing for Brighter's support for Martin Fowler's description of Event Sourcing

Command Sourcing in Brighter

Brighter supports Command Sourcing through the use of its **UseCommandSourcingAttribute**. By adding the attribute to a handler you gain support for logging that **Command** to a **Command Store** A Command Store needs to implement **IAmACommandStore** and we provide an MSSQL Command Store implementation. You can choose to persist the Command to the Store before or after the handler. We recommend Before as this gives you the assurance that if writing the Command to the Store fails, the Handler will not run, meaning that your Store reflects your application state.

The following code shows a handler marked up for Command Sourcing

```
internal class GreetingCommandHandler : RequestHandler<GreetingCommand>
{
   [UseCommandSourcing(step: 1, timing: HandlerTiming.Before)]
   public override GreetingCommand Handle(GreetingCommand command)
   {
       Console.WriteLine("Hello {0}", command.Name);
       return base.Handle(command);
   }
}
```

Inernally the Monitor Handler that Brighter uses to write to the Command Store takes a reference to an **IAmACommandStore**, so you also need to configure your application to provide an implementation at runtime when you provide instances of the Handler from your Handler Factory implementation. The example code relies on the TinyIoC Inversion of Control container to hookup the Handler and Command Store.

```
private static void Main(string[] args)
{
    var dbPath = Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().
↪GetName().CodeBase.Substring(8)), "App_Data\\CommandStore.sdf");
    var connectionString = "DataSource=\"" + dbPath + "\"";
    var configuration = new MsSqlCommandStoreConfiguration(connectionString, "Commands
↪", MsSqlCommandStoreConfiguration.DatabaseType.SqlCe);
    var commandStore = new MsSqlCommandStore(configuration);

    var registry = new SubscriberRegistry();
    registry.Register<GreetingCommand, GreetingCommandHandler>();

    var tinyIoCContainer = new TinyIoCContainer();
    tinyIoCContainer.Register<IHandleRequests<GreetingCommand>,␣
↪GreetingCommandHandler>();
    tinyIoCContainer.Register<IAmACommandStore>(commandStore);

    var builder = CommandProcessorBuilder.With()
        .Handlers(new HandlerConfiguration(
            subscriberRegistry: registry,
            handlerFactory: new TinyIocHandlerFactory(tinyIoCContainer)
        ))
        .DefaultPolicy()
        .NoTaskQueues()
        .RequestContextFactory(new InMemoryRequestContextFactory());

        var commandProcessor = builder.Build();

        var greetingCommand = new GreetingCommand("Ian");

        commandProcessor.Send(greetingCommand);
```

```
        var retrievedCommand = commandStore.Get<GreetingCommand>(greetingCommand.Id).
↪Result;

        var commandAsJson = JsonConvert.SerializeObject(retrievedCommand);

        Console.WriteLine(string.Format("Command retrieved from store: {0}",␣
↪commandAsJson));

        Console.ReadLine();
}
```

The example code also shows retrieving the command from the store, using the **IAmACommandStore.Get** method, passing in the Id of the Command.

The retrieved command could be replayed, although in this case we simply log it to the console.

Basic Configuration

We want to support using Brighter in your project with the minimum of dependencies on other packages. Specifically we want to avoid a dependency on Inversion Of Control (IoC)framework, or logging framework to give you freedom over the libraries you chose for your project.

Mark Seeman's blogs on a DI Friendly Framework and Message Dispatching without Service Location are highly influential on the current implementation of Brighter. (An earlier version of Brighter used Service Location for message dispatch which resulted in the need for abstraction of the client's IoC implementation of choice).

This does mean that clients have slightly more to implement over simply plugging us into their IoC container, but the loose-coupling from an IoC container is on our opinion worth that cost.

What you need to provide

- You need to provide a **Subscriber Registry** with all of the **Command**s or **Event**s you wish to handle, mapped to their **Request Handlers**.
- You need to provide a **Handler Factory** to create your Handlers
- You need to provide a **Policy Registry** if you intend to use Polly to support Retry and Circuit-Breaker.
- You need to provide a **Request Context Factory**

Subscriber Registry

The Command Dispatcher needs to be able to map **Command**s or **Event**s to a **Request Handlers**. For a **Command** we expect one and only one **Request Handlers** for an event we expect many. Register your handlers with the **Subscriber Registry**

```
var registry = new SubscriberRegistry();
registry.Register<GreetingCommand, GreetingCommandHandler>();
```

We also support an initializer syntax

```
var registry = new SubscriberRegistry()
{
    {typeof(GreetingCommand), typeof(GreetingCommandHandler)}
}
```

Handler Factory

We don't know how to construct your handler so we call a factory, that you provide us, to build this entire dependency chain. This factory needs to implement the interface defined in **IAmAHandlerFactory**.

Brighter manages the lifetimes of handlers, as we consider the request pipeline to be a scope, and we will call your factory again asking to release those handlers once we have terminated the pipeline and finished processing the request. You should take appropriate action to clear up the handler and its dependencies in response to that call

It's worth reading Mark Seeman's article on DI Friendly Frameworks to understand this technique. Brighter originally used a conforming container but switched to user defined factories as per Mark's blog.

You can implement the Handler Factory using an IoC container, in your own code. For example:

```
internal class TinyIocHandlerFactory : IAmAHandlerFactory
{
    private readonly TinyIoCContainer _container;

    public TinyIocHandlerFactory(TinyIoCContainer container)
    {
        _container = container;
    }

    public IHandleRequests Create(Type handlerType)
    {
        return (IHandleRequests)_container.Resolve(handlerType);
    }

    public void Release(IHandleRequests handler)
    {
        var disposable = handler as IDisposable;
        if (disposable != null)
        {
            disposable.Dispose();
        }
    }
}
```

Policy Registry

If you intend to use a Polly Policy to support Retry and Circuit-Breaker then you will need to register the Policies in the **Policy Registry**. Registration requires a string as a key, that you will use in your [UsePolicy] attribute to choose the policy. We provide two keys: CommandProcessor. and C

```
var retryPolicy = Policy.Handle<Exception>().WaitAndRetry(new[] { TimeSpan.
↪FromMilliseconds(50), TimeSpan.FromMilliseconds(100), TimeSpan.
↪FromMilliseconds(150) });
var circuitBreakerPolicy = Policy.Handle<Exception>().CircuitBreaker(1, TimeSpan.
↪FromMilliseconds(500));
var policyRegistry = new PolicyRegistry() { { CommandProcessor.RETRYPOLICY,␣
↪retryPolicy }, { CommandProcessor.CIRCUITBREAKER, circuitBreakerPolicy } };
```

#

which you can then use in code like this

```
[RequestLogging(step: 1, timing: HandlerTiming.Before)]
[UsePolicy(CommandProcessor.CIRCUITBREAKER, step: 2)]
[UsePolicy(CommandProcessor.RETRYPOLICY, step: 3)]
public override TaskReminderCommand Handle(TaskReminderCommand command)
{
    _mailGateway.Send(new TaskReminder(
        taskName: new TaskName(command.TaskName),
        dueDate: command.DueDate,
        reminderTo: new EmailAddress(command.Recipient),
        copyReminderTo: new EmailAddress(command.CopyTo)
    ));

    return base.Handle(command);
}
```

Request Context Factory

You need to provide a factory to give us instances of a Context. If you have no implementation to use, just use the default **InMemoryRequestContextFactory**

Putting it all together

All these individual elements can be passed to a **Command Processor Builder** to help build a **Command Processor**. This has a fluent interface to help guide you when configuring Brighter. The result looks like this:

```
var commandProcessor = CommandProcessorBuilder.With()
    .Handlers(new HandlerConfiguration(subscriberRegistry, handlerFactory))
    .Policies(policyRegistry)
    .NoTaskQueues()
    .RequestContextFactory(new InMemoryRequestContextFactory())
    .Build();
```

We discuss Task Queues later.

Supporting Logging

Logger

We use LibLog so that we do not have to depend on a given logging library. You will want to add a logger, supported by LibLog to your project (as per the instructions for LibLog) if you want to see log output from Brighter.

Testing

LibLog can either keep the ILog used by the client library private to the library, or allow the client library to expose it. Brighter chooses to expose the ILog implementation that we receive from ILog.

Why?

When testing you may want to avoid a dependency on the static logger that LibLog wraps your current logging framework with, as static variables can make tests behave erratially when run in parallel, or as part of a suite. For this reason we tend to add a constructor that explicitly takes an instance of Brighter's ILog to allow you to provide an Test Double of that ILog implementation.

```
[Subject(typeof(PipelineBuilder<>))]
public class When_Building_A_Handler_For_A_Command
{
    private static PipelineBuilder<MyCommand> s_chain_Builder;
    private static IHandleRequests<MyCommand> s_chain_Of_Responsibility;
    private static RequestContext s_request_context;

    private Establish _context = () =>
    {
        var logger = A.Fake<ILog>();
        var registry = new SubscriberRegistry();
        registry.Register<MyCommand, MyCommandHandler>();
        var handlerFactory = new TestHandlerFactory<MyCommand, MyCommandHandler>(() =>
↪ new MyCommandHandler(logger));
        s_request_context = new RequestContext();

        s_chain_Builder = new PipelineBuilder<MyCommand>(registry, handlerFactory,␣
↪logger);
    };

    private Because _of = () => s_chain_Of_Responsibility = s_chain_Builder.Build(s_
↪request_context).First();

    private It _should_have_set_the_context_on_the_handler = () => s_chain_Of_
↪Responsibility.Context.ShouldNotBeNull();
    private It _should_use_the_context_that_we_passed_in = () => s_chain_Of_
↪Responsibility.Context.ShouldBeTheSameAs(s_request_context);
}
```

However, outside of testing, you rarely need to provide the instance of ILog. This is because in the base class **RequestHandler&ltT>** the default constructor simply grabs the current logger from LibLog.

```
///
/// Initializes a new instance of the  class.
///
protected RequestHandler()
: this(LogProvider.GetCurrentClassLogger())
{}
```

The only time you might want to change this, is if you intend to call a more derived constructor in your own code, that contains the ILog used by testing. In this case, you can still keep ILog out of your code, by providing the call to LibLog yourself (**LogProvider.GetCurrentClassLogger()**)

```
public MailTaskReminderHandler(IAmAMailGateway mailGateway, IAmACommandProcessor␣
↪commandProcessor)
    : this(mailGateway, commandProcessor, LogProvider.GetCurrentClassLogger())
{}

public MailTaskReminderHandler(IAmAMailGateway mailGateway, IAmACommandProcessor␣
↪commandProcessor, ILog logger)
    : base(logger)
```

```
{
    _mailGateway = mailGateway;
    _commandProcessor = commandProcessor;
}
```

Implementing a Distributed Task Queue

Brighter provides support for a distributed task queue. Instead of handling a command or event, synchronously and in-process, work can be dispatched to a distributed task queue to be handled asynchronously and out-of-process. The trade-off here is between the cost of distribution (see The Fallacies of Distributed Computing) against performance.

For example you might have an HTTP API a rule that any given request to that API must execute in under 100ms. On measuring the performance of a key POST or PUT operation to your API you find that you exceed this value. Upon realizing that much of your time is spent I/O you consider two options:

- Use the TPL to perform the work concurrently
- Offload the work to a distributed task queue, ack the message, and allow the work to complete asynchronously

A problem with the TPL approach is that your operation can only meet the 100ms threshold if your work can be parallelised such that no sub-task takes longer than 100ms. Your speed is always constrained by the slowest operation that you need to parallelize. If you are I/O bound on a resource experiencing contention beyond 100ms, you will not meet your goal by introducing more threads. Your minimum time is your minimum time.

You might try to fix this by acking (acknowledging) the request, and completing the work asynchronously. This option is particularly attractive if the work is I/O bound as you can process other requests whilst you wait for the I/O to complete.

The downside of the async approach is that you risk that the work will be lost if the server fails prior to completion of the work, or the app simply recycles.

These requirements tend to push you in the direction of Guaranteed Delivery to ensure that work you ack will eventually be handled.

A distributed task queue allows you offload work to another process, to be handled asynchronously (once you push the work onto the queue, you don't wait) and in parallel (you can use other cores to process the work). It also allows you to ensure delivery of the message, eventually (the queue will hold the work until a consumer is available to read it).

In addition use of a distributed task queue allows you to throttle requests - you can hand work off from the web server to a queue that only needs to consume at the rate you have resources to support. This allows you to scale to meet unexpected demand, at the price of eventual consistency.

Brighter's Task Queue Architecture

Brighter implements Task Queues using a Message Broker.

The producer sends a **Command** or **Event** to a Message Broker using **CommandProcessor.Post()**.

We use an **IAmAMessageMapper** to map the **Command** or **Event** to a **Message**. (Usually we just serialize the object to JSON and add to the **MessageBody**), but if you want to use higher performance serialization approaches, such as protobuf-net, the message mapper is agnostic to the way the body is formatted.)

When we deserialize we set the **MessageHeader** which includes a topic (often we use a namespaced name for the **Command** or **Event**).

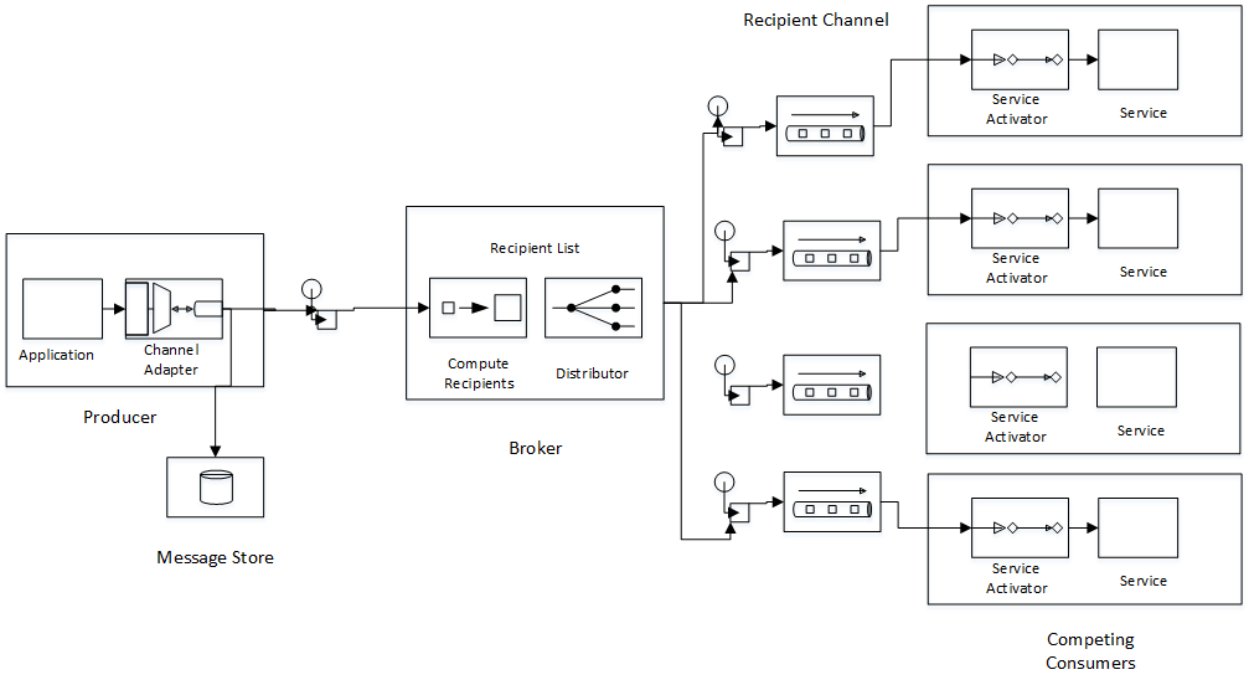
We store the created **Message** in a Message Store for use by **CommandProcessor.Repost()** if we need to resend a failed message.

The Message Broker manages a Recipient List of subscribers to a topic. When it receives a **Message** the Broker looks at the topic in the **MessageHeader** and dispatches the **Message** to the Recipient Channels identified by the Recipient List.

The consumer registers a Recipient Channel to receive messages on a given topic. In other words when the consumer's registered topic matches the producer's topic, the broker dispatches the message to the consumer when it receives it from the producer.

A **Message** may be delivered to multiple Consumers, all of whom get their own copy.

in addition, we can support a Competing Consumers approach by having multiple consumers read from the same Channel to allow us to scale out to meet load.



Do I have to use a Broker, what about MSMQ?

Brighter removes some complexity from its implementation by relying on the Message Broker to provide a number of services. First the Broker provides message routing. The producer does not need to have any idea where the consumers are located, only where the broker is located. This makes it easy to relocate your consumers, and when then begin subscribing the Broker will figure out how to deliver to them. It also supports a recipient list when routing messages: one producer can send to many consumers. Second we rely on the Broker to provide a clustered High Availability (HA) solution to queueing. We want to be able to send a message to the Broker cluster and rely on the Broker to deliver it, eventually.

Without a Broker, using a point-ot-point solution we have to provide a lot of this infrastructure ourselves, such as routing and distribution and how to do so in a way that is HA.

For this reason we don't support a point-to-point approach like MSMQ or sending directly to a service via HTTP.

(We do have an experimental implementation of an HTTP-based broker using the RESTMS specification but it is not production-grade, and only in-memory as of today).

What happens when the consumer receives the message?

A consumer reads the **Message** using the Service Activator pattern to map between an Event Driven Consumer and a Handler.

The use of the Service Activator pattern means the complexity of the distributed task queue is hidden from you. You just write a handler as normal, but call it via post and create a message mapper, the result is that your command is handled reliably, asynchronously, and in parallel with little cognitive overhead. It just works!

What does this look like in code

Instead of using **CommandProcessor.Send()** you use **CommandProcessor.Post()** to send the message

```
var reminderCommand = new TaskReminderCommand(
     taskName: reminder.TaskName,
     dueDate: DateTime.Parse(reminder.DueDate),
     recipient: reminder.Recipient,
     copyTo: reminder.CopyTo);

 _commandProcessor.Post(reminderCommand);
```

You add a message mapper to tell Brighter how to serialize the message for sending to your consumers.

```
public class TaskReminderCommandMessageMapper : IAmAMessageMapper<TaskReminderCommand>
{
    public Message MapToMessage(TaskReminderCommand request)
    {
        var header = new MessageHeader(messageId: request.Id, topic: "Task.Reminder",␣
↪messageType: MessageType.MT_COMMAND);
        var body = new MessageBody(JsonConvert.SerializeObject(request));
        var message = new Message(header, body);
        return message;
    }

    public TaskReminderCommand MapToRequest(Message message)
    {
        return JsonConvert.DeserializeObject<TaskReminderCommand>(message.Body.Value);
    }
}
```

One option is to use a *Core* assembly that contains your domain model, handlers, message mappers etc. and then pull that assembly into endpoints that consume such as services and web endpoints. This makes it easy to move between in-process and out-of-process versions of the handler. It also means you don't end up writing two versions of the mapper one on the consumer side and one on the sender side.

The **'Tasks Example https://github.com/BrighterCommand/Brighter/tree/master/samples>'__** uses this strategy.

This model only works if your library is shared between components that operate on the same bounded context i.e. Continuous Integration that are released together. Never share such an assembly between projects that should be released autonomously as it is a shared dependency. In that case you **must** implement the mapper on both sides.

Then you write a handler as normal.

```
public class MailTaskReminderHandler : RequestHandler<TaskReminderCommand>
{
    private readonly IAmAMailGateway _mailGateway;
```

```
    public MailTaskReminderHandler(IAmAMailGateway mailGateway, IAmACommandProcessor␣
↪commandProcessor)
        : this(mailGateway, commandProcessor, LogProvider.GetCurrentClassLogger())
        {}

    public MailTaskReminderHandler(IAmAMailGateway mailGateway, ILog logger) :␣
↪base(logger)
    {
        _mailGateway = mailGateway;
    }

    [RequestLogging(step: 1, timing: HandlerTiming.Before)]
    [UsePolicy(CommandProcessor.CIRCUITBREAKER, step: 2)]
    [UsePolicy(CommandProcessor.RETRYPOLICY, step: 3)]
    public override TaskReminderCommand Handle(TaskReminderCommand command)
    {
        _mailGateway.Send(new TaskReminder(
        taskName: new TaskName(command.TaskName),
        dueDate: command.DueDate,
        reminderTo: new EmailAddress(command.Recipient),
        copyReminderTo: new EmailAddress(command.CopyTo)
        ));

        return base.Handle(command);
    }
}
```

The Dispatcher

To ensure that messages reach the handlers from the queue you have to use the **Dispatcher**.

The Dispatcher reads messages of input channels. Internally it creates a message pump for each channel, and allocates a thread to run that message pump. The pump consumes messages from the channel, using the **Message Mapper** to translate them into a **Message** and from there a **Command** or **Event**. It then dispatches those to handlers (using the Brighter **Command Processor**.

To use the Dispatcher you need to host it in a consumer application. Usually a console application or Windows Service is appropriate. We recommend using Topshelf to host your consumers.

The following code shows an example of using the **Dispatcher** from Topshelf. The key methods are **Dispatcher.Receive()** to start the message pumps and **Dispatcher.End()** to shut them.

We do allow you to start and stop individual channels, but this is an advanced feature for operating the services.

```
internal class GreetingService : ServiceControl
{
    private Dispatcher _dispatcher;

    public GreetingService()
    {
       /* Configfuration Code Goes here*/
    }

    public bool Start(HostControl hostControl)
    {
```

```
        _dispatcher.Receive();
        return true;
    }

    public bool Stop(HostControl hostControl)
    {
        _dispatcher.End().Wait();
        _dispatcher = null;
        return false;
    }

    public void Shutdown(HostControl hostcontrol)
    {
        if (_dispatcher != null)
            _dispatcher.End();
        return;
    }
}
```

Configuration

So how do we route messages from the channel to the handler? The answer is the framework uses configuration that your provide to do that. Configuration is the subject of this documentation here.

Routing

Publish-Subscribe

Brighter has a default Publish-Subscribe to messaging.

A broker provides an intermediary between the producer of a message and a consumer. A consumer registers interest in messages that have a key or topic. A producer sends messages with a key or topic to a broker, and the broker sends a copy of that message to every subscribing consumer.

In messaging we sometimes refer to the list of subscribers as a **Recipient List**, and because consumers can register their interest at runtime instead of build we sometimes calls this a **Dynamic Recipient List**.

The publish subscribe model works particularly well with an **Event-Driven Architecture** (EDA). In an EDA one process, the publisher, raises an **Event** to indicate that something of interest happened within the process, such as an order being raised or a new customer being added, and subscribers who receive that message can act upon it. Processes can communicate back and forth with one process publishing an **Event** and another system reacting and publishing its **Event** message in turn. A **correlation id**, a unique identifier shared by the messages allows the original producer to correlate events raised by other producers to its message.

The advantage of publish-subscribe is coupling. Because producers do not need to know about consumers and vice-versa then we can change the list of consumers without the producer needing to change, or change the producer without the consumer needing to know.

Routing Publish-Subscribe Messages

A producer routes messages to subscribers by setting a **Topic** on the **MessageHeader**. A **Topic** is just a string that you intend to use as a unique identifier for this message. A simple scheme can be the typename of the event for the

Producer.

When implementing an **IAmAMessageMapper<T>** you set the **Topic** in the **MessageHeader** when serializing your **Command** or **Event** to disk. In the following example we set the **Topic** to *Task.Completed*.

```
public class TaskCompletedEventMapper : IAmAMessageMapper<TaskCompletedEvent>
{
    public Message MapToMessage(TaskCompletedEvent request)
    {
        var header = new MessageHeader(messageId: request.Id, topic: "Task.Completed",
↪ messageType: MessageType.MT_EVENT);
        var body = new MessageBody(JsonConvert.SerializeObject(request));
        var message = new Message(header, body);
        return message;
    }
}
```

On the consumer side we configure **Perfomer**s to subscribe to notifications from a Broker via a **Channel**. That **Channel** subscribes to the **Topic**. So in the above example to receive **TaskCompletedEvent** the **Channel** would need to subscribe to the *Task.Completed* **Topic**.

We can configure consumers with code or configuration.

Configuration is often used to allow us to make run-time changes to subscriber lists easily - by changing the configuration file and restarting the consumer. (Another alternative is to use the Control Bus to configure the consumer at run-time.

To configure a consumer using a configuration file we use the service activator configuration section, which needs to be added to the **<configSections>** element of your configuration file.

```
<section name="serviceActivatorConnections"
         type="paramore.brighter.serviceactivator.ServiceActivatorConfiguration.
↪ServiceActivatorConfigurationSection, paramore.brighter.serviceactivator"
         allowLocation="true"
         allowDefinition="Everywhere" />
```

To configure individual consumers we need to add elements to the **<serviceActivatorConnections>** element.

```
<serviceActivatorConnections>
    <connections>
        <add connectionName="paramore.example.greeting"
             channelName="greeting.command"
             routingKey="greeting.command"
             dataType="Greetings.Ports.Commands.GreetingCommand"
             timeOutInMilliseconds="200" />
    </connections>
</serviceActivatorConnections>
```

The **routingKey** property of the connection must be the same key as used in the message mapper. This is passed to the broker to inform it that we want to subscribe to messages with that routing key on this channel.

Direct Messaging

In direct messaging a producer knows its consumer. A direct message can be fire and forget, which means it does not expect a reply, or request-reply which means that it does. In request-reply the receiver knows its sender as well.

The reason you might choose direct messaging over publish-subscribe is consistency.

When using publish-subscribe work queues we are eventually consistent - at some point in the future we will process the message, relying on 'at least once' delivery properties of the message queue. We don't know anything about 'when' that will happen. This means that two processes in our system may be inconsistent for a period of time - there is latency between them.

Consider an application that needs to bill a customer's credit card.

In an event driven approach, we could make the assumption that the transaction will succeed, raise a request to bill the customer and process the payment asynchronously. The producer of the billing request continues as though the transaction had succeeded. Eventually the customer is billed, and we are consistent. If we fail to bill the customer we have to take compensating action - raising a billing failed event, which may alert an operator and email the customer.

Our reason for taking this approach may be that our payment provider is often slow to respond and we do not want to make the customer wait whilst we handle details of their payment. This may not simply be about responsiveness to the customer - it may be about scaling our system.

In a direct messaging approach, we decide that as many payment transactions fail we do not want to process the order until the payment has been received. At the same time for throughput on our web server we want to work asynchronously and hand off the request to another process which calls the payment provider. Most likely we return a 202 Accepted from our HTTP API with a link to a resource to monitor for the results of the transaction. In our client we display a progress indicator until we have completed the transaction.

In this case, our requirement is that we receive a response to our **Command** to bill.

To route this kind of message the Producer needs to send a reply-address to the Consumer so that it can send a response back. In our case, that reply-address is a topic that the sender subscibes to in order to receive the response.

Usually the Producer creates a topic for all of its replies, and matches request to response via a correlation id. This is simply a unique identifier that the Producer adds to the outgoing message.

To help route direct messages we provide two classes, **Request** and **Reply** but the real work occurs within the message mapper itself.

In the following code snippet we show both the Brighter library's **ReplyAddress** and **Request** as well a derived class **HeartbeatRequest** we use to represent a request for our service to respond with status information.

Note also the correlation id that is added to the **ReplyAddress**.

```
public class ReplyAddress
{
    public ReplyAddress(string topic, Guid correlationId)
    {
        Topic = topic;
        CorrelationId = correlationId;
    }

    public string Topic { get; private set; }
    public Guid CorrelationId { get; private set; }
}

public class Request : Command
{
    public ReplyAddress ReplyAddress { get; private set; }

    public Request(ReplyAddress replyAddress) : base(Guid.NewGuid())
    {
        ReplyAddress = replyAddress;
    }
}

public class HeartbeatRequest : Request
```

```
{
    public HeartbeatRequest(ReplyAddress sendersAddress) : base(sendersAddress)
    {
    }
}
```

When we convert this request into a **Message** via an **IAmAMessageMapper** we set the **MessageHeader** with the topic the Consumer should reply to. We also set the correlation id of the sender's message on the header.

In the following code we also serialize the message back to a **Command** which is then routed by Brighter to a handler. When we serialize back to a **Command** we set the **ReplyAddress** with the Topic and Correlation Id.

```
public class HeartbeatRequestCommandMessageMapper : IAmAMessageMapper
↪<HeartbeatRequest>
{
    public Message MapToMessage(HeartbeatRequest request)
    {
        var header = new MessageHeader(
        messageId: request.Id,
        topic: "Heartbeat",
        messageType: MessageType.MT_COMMAND,
        correlationId: request.ReplyAddress.CorrelationId,
        replyTo: request.ReplyAddress.Topic);

        var json = new JObject(new JProperty("Id", request.Id));
        var body = new MessageBody(json.ToString());
        var message = new Message(header, body);
        return message;
    }

    public HeartbeatRequest MapToRequest(Message message)
    {
        var replyAddress = new ReplyAddress(topic: message.Header.ReplyTo,␣
↪correlationId: message.Header.CorrelationId);
        var request = new HeartbeatRequest(replyAddress);
        var messageBody = JObject.Parse(message.Body.Value);
        request.Id = Guid.Parse((string) messageBody["Id"]);
        return request;
    }
}
```

When we reply, we again use the message mapper to ensure that we route correctly.

Our helper class this time is **Reply** which again encapsulates the reply-to address. We set this from the **Command** in our response. In this code our response to the **HeartbeatRequest** is to respond with a list of running consumers in the service.

```
public class Reply : Command
{
    public ReplyAddress SendersAddress { get; private set; }

    public Reply(ReplyAddress sendersAddress) : base(Guid.NewGuid())
    {
        SendersAddress = sendersAddress;
    }
}

public class HeartbeatReply : Reply
```

```
{
    public HeartbeatReply(string hostName, ReplyAddress sendersAddress) :␣
↪base(sendersAddress)
    {
        HostName = hostName;
        Consumers = new List<RunningConsumer>();
    }

    public string HostName { get; private set; }
    public IList<RunningConsumer> Consumers { get; private set; }
}

public class RunningConsumer
{
    public RunningConsumer(ConnectionName connectionName, ConsumerState state)
    {
        ConnectionName = connectionName;
        State = state;
    }

    public ConnectionName ConnectionName { get; private set; }
    public ConsumerState State { get; private set; }
}
```

Again the key to responding is the **IAmAMessageMapper** implementation which uses the **ReplyAddress** to route the **Message** via its **MessageHeader** back to the caller.

```
internal class HeartbeatReplyCommandMessageMapper : IAmAMessageMapper<HeartbeatReply>
{
    public Message MapToMessage(HeartbeatReply request)
    {
        var header = new MessageHeader(
            messageId:request.Id,
            topic: request.SendersAddress.Topic,
            messageType: MessageType.MT_COMMAND,
            timeStamp: DateTime.UtcNow,
            correlationId: request.SendersAddress.CorrelationId
        );

        var json = new JObject(
            new JProperty("HostName", request.HostName),
            new JProperty("Consumers",
            new JArray(
                from c in request.Consumers
                select new JObject(
                    new JProperty("ConnectionName", c.ConnectionName.ToString()),
                    new JProperty("State", c.State)
                )
                )
            )
        );

        var body = new MessageBody(json.ToString());
        var message = new Message(header, body);
        return message;
    }

    public HeartbeatReply MapToRequest(Message message)
```

```
    {
        var messageBody = JObject.Parse(message.Body.Value);
        var hostName = (string) messageBody["HostName"];
        var replyAddress = new ReplyAddress(message.Header.Topic, message.Header.
↪CorrelationId);

        var reply = new HeartbeatReply(hostName, replyAddress);
        var consumers = (JArray) messageBody["Consumers"];
        foreach (var consumer in consumers)
        {
            var connectionName = new ConnectionName((string)consumer["ConnectionName
↪"]);
            var state = (ConsumerState)Enum.Parse(typeof (ConsumerState), (string)␣
↪consumer["State"]);
            reply.Consumers.Add(new RunningConsumer(connectionName, state));
        }

        return reply;
    }
}
```

Summary

The key to understanding routing in Brighter **IAmAMessageMapper** implementation provides the point at which you control routing by setting the **MessageHeader**.

Distributed Task Queue Configuration

In order to use the distributed task queue you need to configure the Dispatcher.

There are two steps. The first is in-code configuration of your options and the second is configuration of your files in a configuration file.

Why the split?

Generally we prefer to configure design time options in code because that is the easiest to manage and only run-time options in an external configuration file.

In essence, anything that you want to configure post-build i.e. when you deploy needs to go into external configuration, and anything you can configure at build should go into code.

Because you may choose to configure the channels that a service processes at runtime we configure them there. An example use case here is that you may have busy channels that need more consumers to process a backlog. You can add channels to existing services at run-time to help share the load, and then remove those channels later once the backlog has been worked through.

Configuring the Dispatcher in Code

We provide a Dispatch Builder that has a progressive interface to assist you in configuring a **Dispatcher**

You need to consider the following when configuring the Dispatcher

- Logging
- Command Processor
- Message Mappers
- Channel Factory
- Connection List

Of these **Logging** and the **Command Processor** are covered in **Basic Configuration**.

Message Mappers

We use **IAmAMessageMapper<T>** to map between messages in the Task Queue and a **Message**.

A **Message** consists of two parts, a **Message Header** and **Message Body**. The header contains metadata about the message. Key properties are time **TimeStamp**, **Topic**, and **Id**. The body consists of the serialized **IRequest** sent over the Task Queue.

We dispatch a **Message** using either **CommandProcessor.Send()** or **CommandProcessor.Publish()** depending on whether the **MessageHeader.MessageType** is **MT_COMMAND** or **MT_EVENT**.

You create a **Message Mapper** by deriving from **IAmAMessageMapper<TaskReminderCommand>** and implementing the **MapToMessage()** and **MapToRequest** methods.

```
public class TaskReminderCommandMessageMapper : IAmAMessageMapper<TaskReminderCommand>
{
    public Message MapToMessage(TaskReminderCommand request)
    {
        var header = new MessageHeader(messageId: request.Id, topic: "Task.Reminder",␣
↪messageType: MessageType.MT_COMMAND);
        var body = new MessageBody(JsonConvert.SerializeObject(request));
        var message = new Message(header, body);
        return message;
    }

    public TaskReminderCommand MapToRequest(Message message)
    {
        return JsonConvert.DeserializeObject<TaskReminderCommand>(message.Body.Value);
    }
}
```

You then need to register your Message Mapper so that we can find it, using a class that derives from **IAmAMessageMapperRegistry**. We recommend using **MessageMapperRegistry** unless you have more specific requirements.

```
var messageMapperRegistry = new MessageMapperRegistry(messageMapperFactory)
{
    { typeof(GreetingCommand), typeof(GreetingCommandMessageMapper) }
};
```

Channel Factory

The Channel Factory is where we take a dependency on a specific Broker. We pass the **Dispatcher** an instances of **InputChannelFactory** passing it an implementation of **IAmAChannelFactory**. The channel factory is used to create channels that wrap the underlying Message-Oriented Middleware that you are using.

For production use we support RabbitMQ as a Broker. We are actively working on other implementations.

You can see the code for this in the full builder snipped below.

We don't cover details of how to implement a Channel Factory here, for simplicity.

Connection List

Brighter supports configuration of a service activator via code. A Service Activator supports one or more connections.

The most important part of a connection to understand is the **routing key**. This must be the same as the topic you set in the **Message Header** when sending. In addition the **dataType** should be the name of the **Command** or **Event** derived type that you want to deserialize into i.e. we will use reflection to create an instance of this type.

You must set the **connectionName** and **channelName**. The naming scheme is at your discretion. We often use the namespace of the producer's type that serializes into the message on the wire

The **timeOutInMilliseconds** sets how long we wait for a message before timing out. Note that after a timeout we will wait for messages on the channel again, following a delay. This just allows us to yield to receive control messages on the message pump.

```
var connections = new List<Connection>
{
    new Connection(
        new ConnectionName("paramore.example.greeting"),
        new InputChannelFactory(rmqMessageConsumerFactory, rmqMessageProducerFactory),
        typeof(GreetingEvent),
        new ChannelName("greeting.event"),
        "greeting.event",
        timeoutInMilliseconds: 200)
};
```

Creating a Builder

This code fragment shows putting the whole thing together

```
//create message mappers
var messageMapperRegistry = new MessageMapperRegistry(messageMapperFactory)
{
    { typeof(GreetingCommand), typeof(GreetingCommandMessageMapper) }
};

//create the gateway
var rmqMessageConsumerFactory = new RmqMessageConsumerFactory(logger);
_dispatcher = DispatchBuilder.With()
    .CommandProcessor(CommandProcessorBuilder.With()
        .Handlers(new HandlerConfiguration(subscriberRegistry, handlerFactory))
        .Policies(policyRegistry)
        .NoTaskQueues()
        .RequestContextFactory(new InMemoryRequestContextFactory())
        .Build())
    .MessageMappers(messageMapperRegistry)
    .ChannelFactory(new InputChannelFactory(rmqMessageConsumerFactory))
    .Connections(connections)
    .Build();
```

RabbitMQ Configuration

Getting your application to interact with RabbitMQ using Brighter is a trivial task. Simply add the *<rmqMessagingGateway/>* section in your configuration file and ensure that an instance of *RmqMessageProducer* is being used when building the command processor.

The available configuration options are:

- amqpUri: Describes how to connect to RabbitMQ
  - uri: A uri in the format amqp://{user}:{password}@{host}:{port}/{vhost}. Default uri is amqp://guest:guest@localhost:5672/%2f.
  - connectionRetryCount: The retry count for when a connection fails. Default count is 3 retries.
  - retryWaitInMilliseconds: The time in milliseconds to wait before retrying to connect again. Default duration is 1000 ms.
  - circuitBreakTimeInMilliseconds: The time in milliseconds to keep the circuit broken. Default duration is 60000 ms.
- exchange: Describes where messages are sent
  - name: The name of the exchange.
  - type: The type of the exchange. Can be one of:
    - direct (default)
    - fanout
    - headers
    - topic
  - durable: Indicates whether the exchange is durable. Default value is false.
- queues: Defines general settings for queues
  - highAvailability: Indicates whether all queues should be mirrored across all nodes in the cluster. Default value is false.
  - qosPrefetchSize: Allows you to limit the number of unacknowledged messages on a channel (or connection) when consuming (aka "prefetch count"). Default count is 1.

Here's an example of an App.config file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <section name="rmqMessagingGateway" type="paramore.brighter.commandprocessor.
↪messaginggateway.rmq.MessagingGatewayConfiguration.
↪RMQMessagingGatewayConfigurationSection, paramore.brighter.commandprocessor.
↪messaginggateway.rmq" />
    </configSections>
    <rmqMessagingGateway>
        <amqpUri uri="amqp://guest:guest@localhost:5672/%2f" connectionRetryCount="3"␣
↪retryWaitInMilliseconds="1000" circuitBreakTimeInMilliseconds="60000" />
        <exchange name="paramore.brighter.exchange" type="direct" durable="false" />
        <queues highAvailability="false" qosPrefetchSize="1" />
    </rmqMessagingGateway>
</configuration>
```

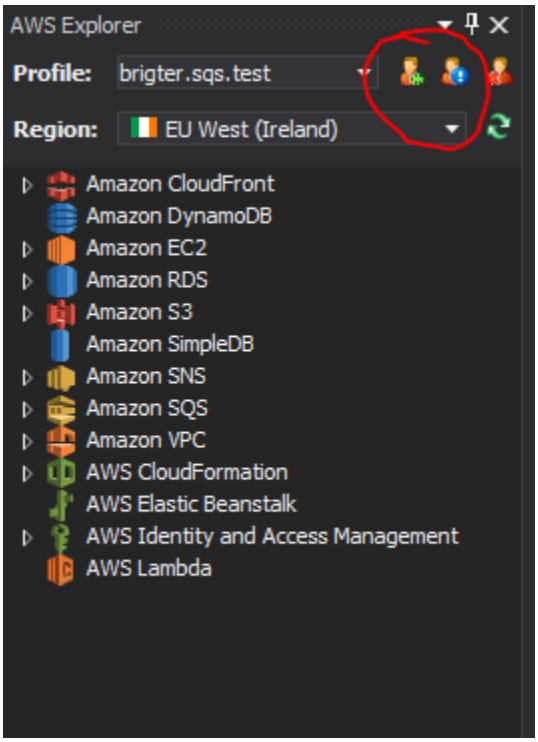
Running Brighter under AWS SQS Infrastructure

Getting your application to interact with SQS using Brighter is a trivial task. But, there are couple of things that needs to be done manually.

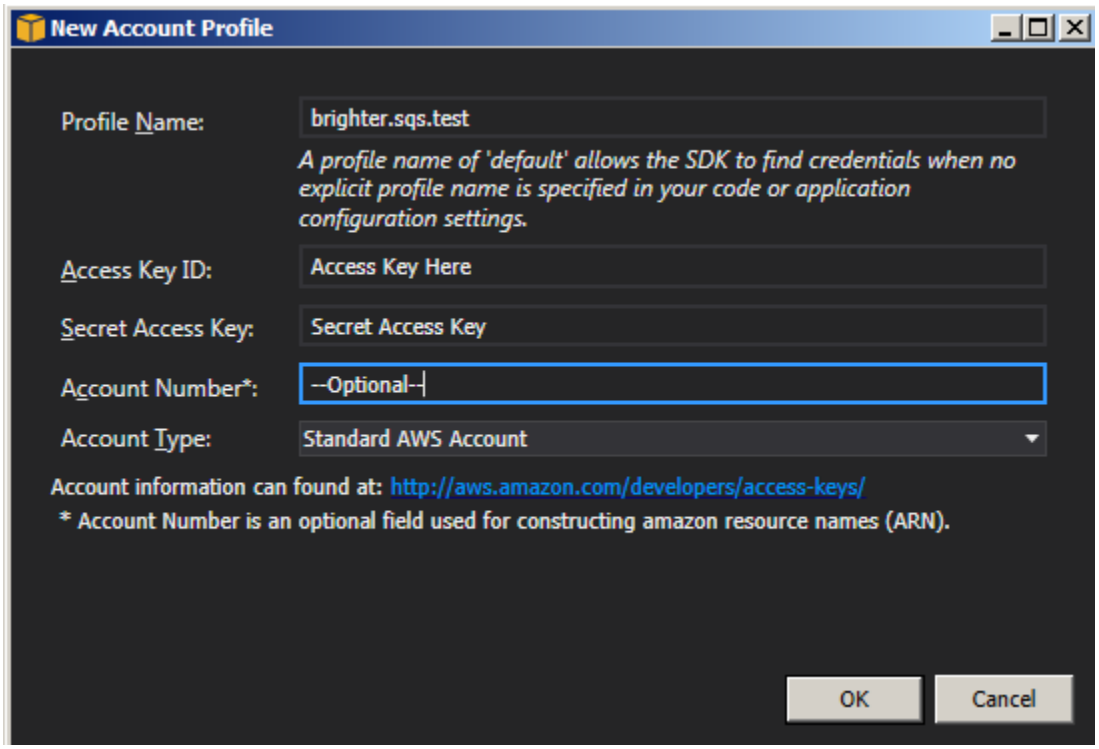
First, you should have your AWS Access Key ID and Secret Access Key ready to be used by AWSSDK. You can check out the AWS documentation on how to create an account here

After setting up your credentials, you can either create a profile in your local computer or you can use the access and secret keys directly in the application configuration file. Here, we will create a profile using AWS Toolkit on Visual Studio:

Click to add profile button on AWS Explorer window.



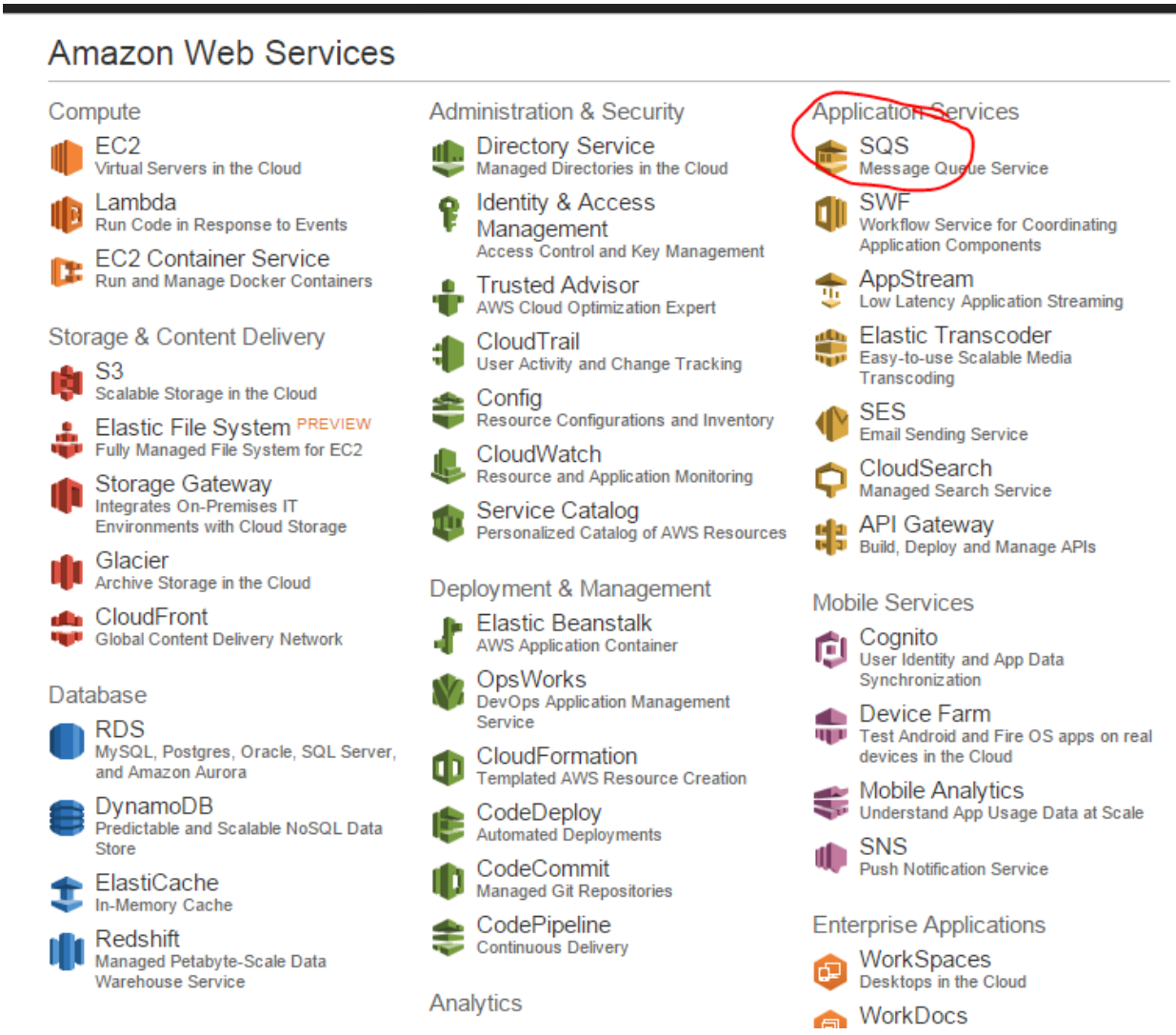
Then fill the form to create your profile.



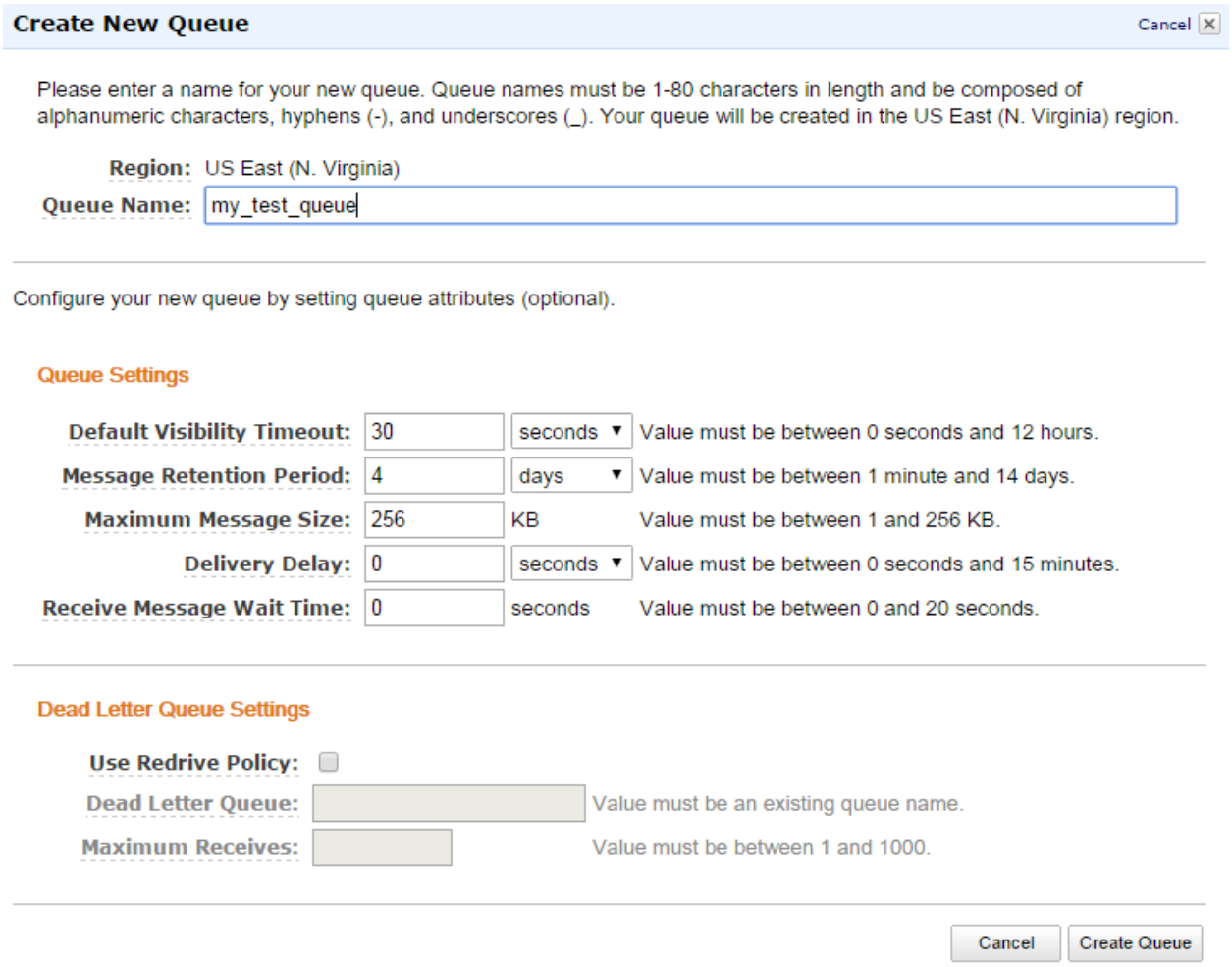
You can find more information about profiles and credentials here.

Creating queues

Unfortunately, brighter will not be creating the queues on request for you; you need to create them manually. To create queues, you should go to your AWS Console.



Then create the queue by filling the form below.

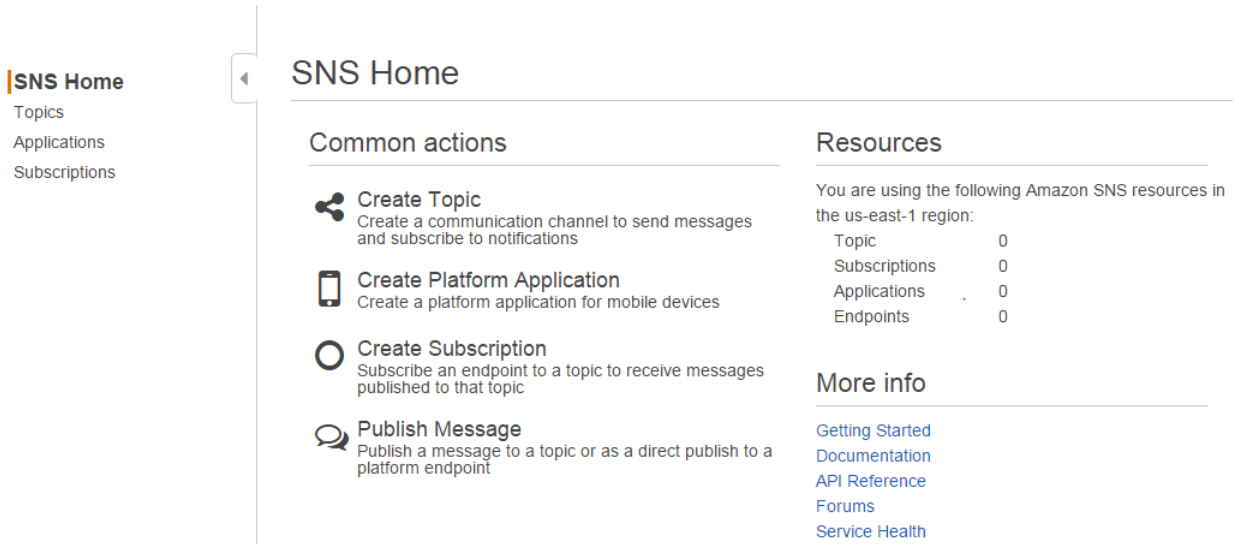


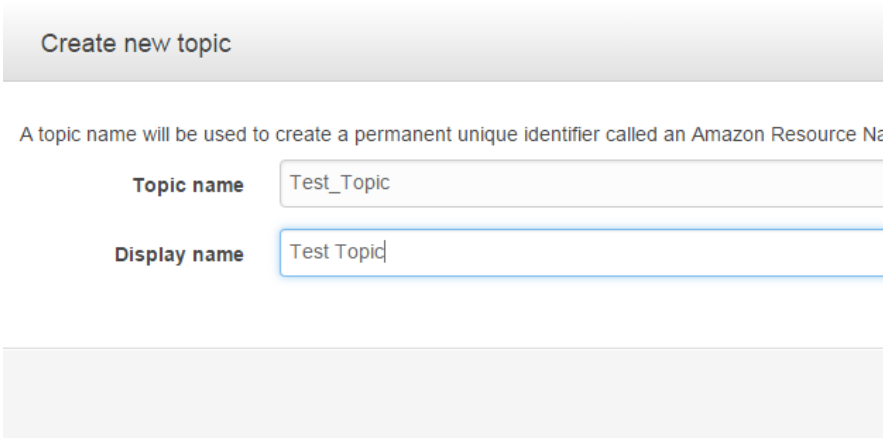
- **Default Visibility Timeout:** The length of time that a message received from a queue will be invisible to other receiving components.
- **Message Retention Period:** The amount of time that Amazon SQS will retain amessage if it doesn't get deleted.
- **Maximum Message Size:** Maximum message size in bytes.
- **Delivery Delay:** The amount of time to delay the first delivery of all messages added to the queue.
- **Receive Message Wait Time:** The maximum amount of time that a long polling receive call will wait for a message to become available before returning empty response.

After saving the queue, get the queue url to put it in the application configuration file.

Posting messages

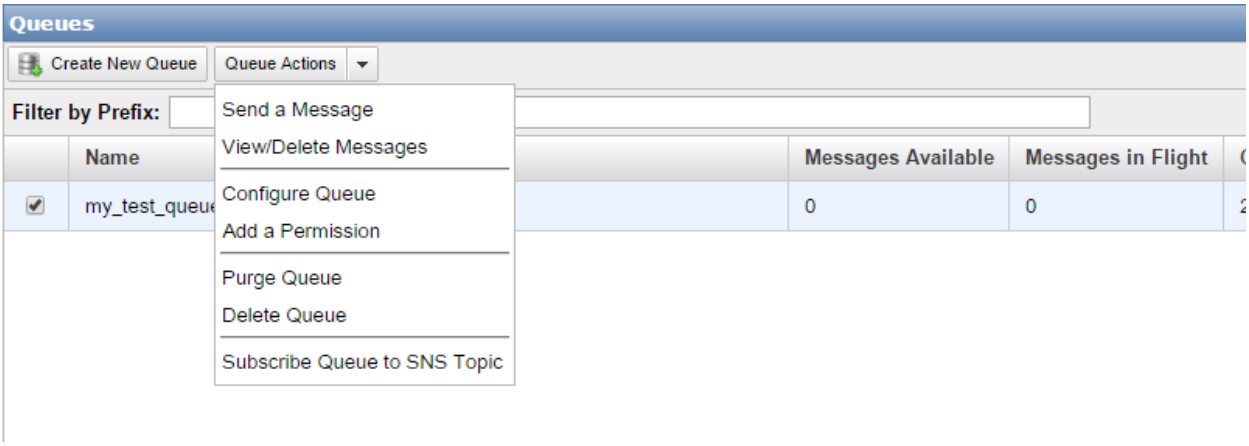
Instead of posting messages directly to the queues, Brighter posts them to a topic so that any queue can listen or stop listening messages without changing any code. AWS SNS is used for that purpose.





You can create a topic by clicking on create topic link

After that, simply go back to your queue list, select a queue and click on Subscribe Queue to SNS Topic link at Queue Actions menu.



Select your SNS queue and now your queue will get the messages which are sent to the SNS topic.

Please follow the link below to see an example application configuration to use AWS SQS:

AWS SQS Configuration

AWS SQS Configuration

Getting your application to interact with SQS using Brighter is a trivial task. Simply add your AWS Credential information in *<aws/>* section in your configuration file and ensure that an instance of *SqsMessageProducer* is being used when building the command processor.

Here's an example of an App.config file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <section name="aws" type="Amazon.AWSSection, AWSSDK" />
    </configSections>
    <aws profileName="brigter.sqs.test" region="eu-west-1" />
</configuration>
```

Brightor will not generate the SQS queues on fly. It is required to create them before hand and define their url in service activator configuration section.

Here's an example of service activator configuration:

```
<serviceActivatorConnections>
    <connections>
        <add connectionName="paramore.example.documentsandfolders.documentcreatedevent
↪" channelName="https://sqs.eu-west-1.amazonaws.com/027649620536/DocumentCreatedEvent
↪" routingKey="DocumentCreatedEvent" dataType="DocumentsAndFolders.Sqs.Ports.Events.
↪DocumentCreatedEvent" timeOutInMilliseconds="5000" requeueDelayInMilliseconds="5000
↪" noOfPerformers="10" />
        <add connectionName="paramore.example.documentsandfolders.documentupdatedevent
↪" channelName="https://sqs.eu-west-1.amazonaws.com/027649620536/DocumentUpdatedEvent
↪" routingKey="DocumentUpdatedEvent" dataType="DocumentsAndFolders.Sqs.Ports.Events.
↪DocumentUpdatedEvent" timeOutInMilliseconds="5000" requeueDelayInMilliseconds="5000
↪" noOfPerformers="10" />
        <add connectionName="paramore.example.documentsandfolders.foldercreateddevent
↪" channelName="https://sqs.eu-west-1.amazonaws.com/027649620536/FolderCreatedEvent"␣
↪routingKey="FolderCreatedEvent" dataType="DocumentsAndFolders.Sqs.Ports.Events.
↪FolderCreatedEvent" timeOutInMilliseconds="5000" requeueDelayInMilliseconds="5000"␣
↪noOfPerformers="10" />
    </connections>
</serviceActivatorConnections>
```

Dispatching Requests Asynchronously

Brighter supports an asynchronous Command Dispatcher and Command Processor.

Using an asynchronous approach to dispatch can be valuable when the work done by a handler can be done concurrently with other work. Instead of blocking on the call to **Send** or **Publish** the calling thread can continue to do work, with a continuation executing once the operation completes. See the MSDN article Asynchronous Programming with Async and Await

Brighter supports using the async...await pattern in .NET to allow your code to avoid blocking. We provide asynchronous versions of the **Command Dispatcher** methods i.e. **CommandProcessor.SendAsync()**, **CommandProcessor.PublishAsync()**, and **CommandProcessor.PostAsync()**.

Usage

In the following example code we register a handler, create a command processor, and then use that command processor to send a command to the handler asynchronously.

Note that this code is the same as the equivalent code for calling the command processor synchonously - apart from the use of async alternatives i.e. **SubscriberRegistry.RegisterAsync()** instead of **SubscriberRegistry.Register()** and **CommandProcessor.SendAsync()** instead of **CommandProcessor.Send()**.

Note also that we have a **SimpleHandlerFactoryAsync** as this factory needs to return handlers that implement **IHandleRequestsAsync** not **IHandleRequests**.

```
private static async Task MainAsync()
{
    var registry = new SubscriberRegistry();
    registry.RegisterAsync<GreetingCommand, GreetingCommandRequestHandlerAsync>();

    var builder = CommandProcessorBuilder.With()
    .Handlers(new HandlerConfiguration(registry, new SimpleHandlerFactoryAsync()))
    .DefaultPolicy()
    .NoTaskQueues()
    .RequestContextFactory(new InMemoryRequestContextFactory());

    var commandProcessor = builder.Build();

    await commandProcessor.SendAsync(new GreetingCommand("Ian"));

    Console.ReadLine();
}
```

Note that line: **Console.ReadLine()** is a continuation. Control passes back to the calling method after the await, and subsequent lines of code run after that method returns.

Registering a Handler

In order for a **Command Dispatcher** to find a Handler for your **Command** or **Event** you need to register the association between that **Command** or **Event** and your Handler.

The **Subscriber Registry** is where you register your Handlers.

The **SubscriberRegistry.RegisterAsync()** expects a handler that implements **IHandleRequestsAsync**

```
var registry = new SubscriberRegistry();
registry.RegisterAsync<GreetingCommand, GreetingCommandRequestHandlerAsync>();
```

Pipelines Must be Homogeneous

Brighter only supports pipelines that are solely **IHandleRequestsAsync** or **IHandleRequests**.

This is due to expectation of the caller using an ***Async** method that the code will execute asynchronously - allowing some handlers in the chain to block would defy that expectations. The async...await pattern is often described as 'viral' because it spreads up the chain of callers to be effective. Brighter is no exception in this regard.

Dispatching Requests

Once you have registered your Handlers, you can dispatch requests to them. To do that you simply use the **CommandProcessor.SendAsync()** (or **CommandProcessor.PublishAsync()** or **CommandProcessor.PostAsync()**) method passing in an instance of your command.

```
await commandProcessor.SendAsync(new GreetingCommand("Ian"));
```

Cancellation

Brighter supports the cancellation of asynchronous operations.

The asynchronous methods: **SendAsync**, **PublishAsync**, and **PostAsync** all accept a **CancellationToken** and pass this token down the pipeline. The parameter defaults to null where the call does not intend to cancel.

The responsibility for checking for a cancellation request lies with the individual handlers, which must determine what action to take if cancellation had been signalled.

The ability of the ***Async** methods to take a cancellation token can be particularly useful with ASP.NET AsyncTimeout see here for more.

Do Not Block When Calling *Async Methods

When calling an asynchronous method you should **await** that method. Avoid using **.Wait** or **.Result** on the **Task** returned by the method, as this turns it back into a blocking call, which is probably not your intent and likely undermines the reason you wanted to use an asynchronous approach in the first place. If you find yourself using **.Wait** or **.Result** then consider whether you would be better using a synchronous pipeline instead.

Therefore you should only call **SendAsync**, **PublishAsync**, or **PostAsync** from a method that is itself async and supports await, otherwise you will block, and there will be no value to having used an async method.

In Ports & Adapters Architecture terms you should use an **Adapter** layer that supports async when calling the **Ports** layer represented by your handlers.

This creates the question: at what point do we stop being async i.e. who waits? This is normally a responsibility of your framework which has to understand that it can use re-use thread to service other requests, thus improving throughput and call back to your continuation when done.

For example ASP.NET Controllers support async can be used to call the ***Async methods** without blocking. This allows ASP.NET to release a thread from the thread pool to service another request whilst the asynchronous operation completes, allowing greater throughput on the server.

Understand Captured Contexts

When an awaited method completes, what thread runs any completion code? The answer depends on the SynchronizationContext which is 'captured' at the point await is called. For ASP.NET or Windows Forms, WPF, or Metro apps then the SynchronizationContext means that the thread that was running at the point we yielded runs the continuation. Otherwise the SynchronizationContext is null and the default Task Scheduler runs the continuation.

Why does this matter? Because if you needed to access anything that is thread local, being called back on the wrong thread means you will not have access to those variables.

A Windows UI for example is single-threaded via a message pump and interacting with the UI requires you to be on that thread. See this article for more.

When awaiting it is possible to configure how the continuation runs - on the SyncronizationContext or using the Task Scheduler, overriding the default behaviour, which is to capture the SynchronizationContext.

```
await MethodAsync(value, ct).ConfigureAwait(true);
```

Library writers are encouraged to default to false i.e. use the Task Scheduler instead of the SychronizationContext.

Brighter adopts this default, but recognizes it might not be what you want if your handler needs to run in the context of the original thread. As a result we let you pass in a parameter on the ***Async** calls to change the behaviour throughout your pipeline.

```
await commandProcessor.SendAsync(new GreetingCommand("Ian"),␣
↪continueOnCapturedContext: true);
```

A handler exposes the parameter you supply to the call to **SendAsync**, **PublishAsync**, or **PostAsync** via a property called **ContinueOnCapturedContext**. That property is true if we want to use the SynchronizationContext and not the Task Scheduler to run our continuation.

```
await base.HandleAsync(command, ct).ConfigureAwait(ContinueOnCapturedContext);
```

We recommend explicitly using this parameter when awaiting within your own handler, such as when calling the next handler in an async pipeline.

Asynchronous vs. Work Queues

One obvious question is: when should I use an asynchronous pipeline to handle work and when should I use a work queue.

Using an asynchronous handler allows us to avoid blocking. This can increase our throughput by allowing us to re-use threads to service new requests. Using this approach, even a single-threaded application can achieve high throughput, if it is not CPU-bound.

Using a work queue allows us to hand-off work to another process, to be executed at some point in the future. This also allows us to improve throughput by freeing up the thread to service new requests. We assume that we can accept dealing with that work at some point in the future i.e. we can be eventually consistent.

One disadvantage of a work queue is that our pattern - ack to callers, and then do the work, can create additional complexity because we must deal with notifying the user of completion, or errors. Because an async operation simply has the caller wait, the programming model is simpler. The trade-off here is that the client of our process is still using resources awaiting for the request with the async operation. If the operation takes time to complete the client may not know if the operation failed and should be timed out, or is still running.

Where work is long-running there is a risk that the server faults, and we lose the long-running work. A work queue provides reliability here, through guaranteed delivery. The queue keeps the work until it is successfully processed and acknowledged.

Our recommendation is to use the async pattern to improve throughput where the framework supports async, such as ASP.NET WebAPI but to continue to hand-off work that takes a long time to complete to a work queue. You may choose to define your own thresholds but we recommend that operations that take longer than 200ms to complete be handed-off. We also recommend that operations that are CPU bound be handed-off as they diminish the throughput of your application.

How to Implement an Asynchronous Request Handler

To implement an asynchronous handler, derive from **RequestHandlerAsync<T>** where *T* should be the **Command** or **Event** derived type that you wish to handle. Then override the base class **RequestHandlerAsync<T>.HandleAsync()**

method to implement your handling for the Command or Event.

For example, assume that you want to handle the **Command** GreetingCommand

```
public class GreetingCommand : IRequest
{
    public GreetingCommand(string name)
    {
        Id = Guid.NewGuid();
        Name = name;
    }

    public Guid Id { get; set; }
    public string Name { get; private set; }
}
```

Then derive your handler from **RequestHandlerAsync<GreetingCommand>** and accept a parameter of that type on the overriden **HandleAsync()** method, along with a nullable cancellation token - which you should default to null.

To ensure that the pipeline runs, you should return the result of the next handler in the chain, by awaiting the base class **HandleAsync()**.

(Because the next element in the pipeline should also be async, you should always await the result of this call.)

```
public class GreetingCommandRequestHandlerAsync : RequestHandlerAsync
{
    public override async Task HandleAsync(GreetingCommand command, CancellationToken?
↪ ct = null)
    {
        var api = new IpFyApi(new Uri("https://api.ipify.org"));

        var result = await api.GetAsync(ct);

        Console.WriteLine("Hello {0}", command.Name);
        Console.WriteLine(result.Success ? "Your public IP addres is {0}" : "Call to␣
↪IpFy API failed : {0}",
        result.Message);
        return await base.HandleAsync(command, ct).ConfigureAwait(base.
↪ContinueOnCapturedContext);
    }
}
```

Note how we use **ConfigureAwait()** when calling the next handler in the chain, and set the value to the **RequestHandlerAsync<GreetingCommand>.ContinueOnCapturedContext** property. This ensures that we utilize any override of the default (which is to use the Task Scheduler) made when the call to **SendAsync**, **PublishAsync**, or **PostAsync** was made.

It is worth noting that although the override forces you to return a **Task<T>** it does not force you to add the **async** keyword to the method to compile. This risks introducing a subtle bug. You can await a method that returns a **Task<T>** but creation of the state machine in the caller depends on the presence of the **async** keyword. If your handler does not await anything, you will not be forced to add the **async** keyword. Your handler will run sychronously in this context, which may not be what you expect.

Remembering to always await the base class **HandleAsync()** mitigates against this as even if your handler does not do asynchronous work, you will be forced to add **async** to the signature.

Building a Pipeline of Async Request Handlers

Once you are using the features of Brighter to act as a command dispatcher and send or publish messages to a target handler, you may want to use its command processor features to handle orthogonal operations.

Implementing a Pipeline

The first step in building a pipeline is to decide that we want an orthogonal operation in our pipeline. Let us assume that we want to do command sourcing.

Because you do not want to write an orthogonal handler for every Command or Event type, these handlers should remain generic types. At runtime the framework will request HandlerFactory creates an instance of the generic type specialized for the type parameter of the Command or Event being passed along the pipeline.

The limitation here is that you can only make assumptions about the type you receive into the pipeline from the constraints on the generic type.

Although it is possible to implement the IHandleRequestsAsync interface directly, we recommend deriving your handler from RequestHandlerAsync<T>.

Let us assume that we want to log all requests travelling through the pipeline. (We provide this for you in the Brighter.CommandProcessor packages so this for illustration only). We could implement a generic handler as follows:

```
public class CommandSourcingHandlerAsync<T> : RequestHandlerAsync<T> where T : class,␣
↪IRequest
{
    private readonly IAmACommandStoreAsync _commandStore;

    public CommandSourcingHandlerAsync(IAmACommandStoreAsync commandStore)
        : this(commandStore, LogProvider.GetCurrentClassLogger())
    { }


    public CommandSourcingHandlerAsync(IAmACommandStoreAsync commandStore, ILog␣
↪logger) : base(logger)
    {
        _commandStore = commandStore;
    }

    public override async Task<T> HandleAsync(T command, CancellationToken? ct = null)
    {
        logger.DebugFormat("Writing command {0} to the Command Store", command.Id);

        await _commandStore.AddAsync(command, -1, ct).
↪ConfigureAwait(ContinueOnCapturedContext);

        return await base.HandleAsync(command, ct).
↪ConfigureAwait(ContinueOnCapturedContext);
    }
}
```

Our HandleAsync method is the method which will be called by the pipeline to service the request. After we log we call **return await base.HandleAsync(command, ct)** to ensure that the next handler in the chain is called.

If we failed to do this, the *target handler* would not be called nor any subsequent handlers in the chain. This call to the next item in the chain is how we support the 'Russian Doll' model - because the next handler is called within the

scope of this handler, we can manage when it is called handle exceptions, units of work, etc.

It is worth remembering that handlers may be called after the target handler (in essence you can designate an orthogonal handler as the sink handler when configuring your pipeline). For this reason ****all**** handlers should remember to call their successor, **even **your** target handler**.

We now need to tell our pipeline to call this orthogonal handler before our target handler. To do this we use attributes. The code we want to write looks like this:

```
internal class GreetingCommandRequestHandlerAsync : RequestHandlerAsync
↪<GreetingCommand>
{
    [UseCommandSourcingAsync(step: 1, timing: HandlerTiming.Before)]
    public override async Task<GreetingCommand> HandleAsync(GreetingCommand command,␣
↪CancellationToken? ct = null)
    {
        var api = new IpFyApi(new Uri("https://api.ipify.org"));

        var result = await api.GetAsync(ct);

        Console.WriteLine("Hello {0}", command.Name);
        Console.WriteLine(result.Success ? "Your public IP addres is {0}" : "Call to␣
↪IpFy API failed : {0}", result.Message);
        return await base.HandleAsync(command, ct).ConfigureAwait(base.
↪ContinueOnCapturedContext);
    }
}
```

The **UseCommandSourcingAsync** Attribute tells the Command Processor to insert a Logging handler into the request handling pipeline before(**HandlerTiming.Before**) we run the target handler. It tells the Command Processor that we want it to be the first handler to run if we have multiple orthogonal handlers i.e. attributes (**step: 1**).

We implement the **UseCommandSourcingAsyncAttribute** by creating our own Attribute class, derived from **RequestHandlerAttribute**.

```
public class UseCommandSourcingAsyncAttribute : RequestHandlerAttribute
{

    public UseCommandSourcingAsyncAttribute(int step, HandlerTiming timing =␣
↪HandlerTiming.Before)
        : base(step, timing)
    { }


    public override Type GetHandlerType()
    {
        return typeof (CommandSourcingHandlerAsync<>);
    }
}
```

The most important part of this implementation is the GetHandlerType() method, where we return the type of our handler. At runtime the Command Processor uses reflection to determine what attributes are on the target handler and requests an instance of that type from the user-supplied **Handler Factory**.

Your Handler Factory needs to respond to requests for instances of a **RequestHandlerAsync<T>** specialized for a concrete type. For example, if you create a**CommandSourcingHandlerAsync<TRequest>** we will ask you for a **CommandSourcingHandlerAsync<MyCommand>** etc. Depending on your implementation of HandlerFactory, you may need to register an implementation for every concrete instance of your handler with your underlying IoC container etc.

Note that as we rely on an user supplied implementation of **IAmAHandlerFactoryAsync** to instantiate Handlers, you can have any dependencies in the constructor of your handler that you can resolve at runtime. In this case we pass in an ILog reference to actually log to.

You may wish to pass parameter from your Attribute to the handler. Attributes can have constructor parameters or public members that you can set when adding the Attribute to a target method. These can only be compile time constants, see the documentation here. After the Command Processor calls your Handler Factory to create an instance of your type it calls the **RequestHandler.InitializeFromAttributeParams** method on that created type and passes it the object array defined in the **RequestHandlerAttribute.InitializerParams**. By this approach, you can pass parameters to the handler, for example the Timing parameter is passed to the handler above.

It is worth noting that you are limited when using Attributes to provide constructor values that are compile time constants, you cannot pass dynamic information. To put it another way you are limited to value set at design time not at run time.

In fact, you can use this approach to pass any data to the handler on initialization, not just attribute constructor or property values, but you are constrained to what you can access from the context of the Attribute at run time. It can be tempting to set retrieve global state via the Service Locator pattern at this point. Avoid that temptation as it creates coupling between your Attribute and global state reducing modifiability.

Monitoring

Brighter emits monitoring information from Task Queues using a configured Control Bus

Configuring Monitoring

Firstly configure a Control Bus in the brighter application to emit monitoring messages

Config file

Monitoring requires a new section to be added to the application config file:

```
<configSections>
    <section name="monitoring" type ="paramore.brighter.commandprocessor.monitoring.
↪Configuration.MonitoringConfigurationSection, Brighter.commandprocessor"␣
↪allowLocation ="true" allowDefinition="Everywhere"/>
</configSections>
```

The monitoring config can then be speicified later in the file:

```
<monitoring>
    <monitor isMonitoringEnabled="true" instanceName="ManagementAndMonitoring"/>
</monitoring>
```

This enables runtime changes to enable/disable emitting of monitoring messages.

Handler confguration

Each handler that requires monitoring must be configured in two stages, a Handler attribute and container registration of a MonitorHandler for the given request:

For example, given:

- TRequest - a Brighter Request, inheriting from IRequest
- TRequestHandler - handles the TRequest, inheriting IHandleRequest <TRequest>

Attribute

The following attribute must be added to the Handle method in the handler, TRequestHandler:

```
[Monitor(step:1, timing:HandlerTiming.Before, handlerType:typeof(TRequestHandler))]
```

Please note the step and timing can vary if monitoring should be after another attribute step, or timing should be emitted after.

Container registration

The following additional handler must be registered in the application container (where `MonitorHandler<T>` is a built-in Brighter handler):

```
container.Register<TRequest, MonitorHandler<TRequest>>
```

Monitor message format

A message is emitted from the Control Bus on Handler Entry and Handler Exit. The following is the form of the message:

```
{
    "Exception": null, // or Exception message
    "EventType": "EnterHandler or ExitHandler",
    "EventTime": "2016-06-21T15:48:26.1390192Z",
    "TimeElapsedMs": 0 or Duration,
    "HandlerName": "...",
    "HandlerFullAssemblyName": "...",
    "InstanceName": "ManagementAndMonitoring",
    "RequestBody": "{\"Id\":\"dc32b35f-bc75-4197-9178-c8310a63e4fb\", ... }",
    "Id": "048cc207-e820-40fa-b931-55b60203fbc2"
}
```

Messages can be processed from the queue and interated with your monitoring tool of choice, for example Live python consumers emitting to console or logstash consumption to the ELK stack using relevant plugins to provide performance raditators or dashboards.

Feature Switches

We provide a **FeatureSwitch** Attribute that you can use on your **IHandleRequests<TRequest>.Handle()** method. The **FeatureSwitch** Attribute that you have configured will determine whether or not the **IHandleRequests<TRequest>.Handle()** will be executed.

Using the Feature Switch Attribute

By adding the **FeatureSwitch** Attribute, you instruct the Command Processor to do one of the following:

- run the handler as normal, this is **FeatureSwitchStatus.On**.
- not execute the handler, this is **FeatureSwitchStatus.Off**.
- detemine whether to run the handler based on a **Feature Switch Registry**, creating of which is described later.

In the following example, **MyFeatureSwitchedHandler** will only be run if it has been configured in the **Feature Switch Registry** and set to **FeatureSwitchStatus.On**.

```
class MyFeatureSwitchedHandler : RequestHandler<MyCommand>
{
    [FeatureSwitch(typeof(MyFeatureSwitchedHandler), FeatureSwitchStatus.Config,␣
↪step: 1)]
    public override MyCommand Handle (MyCommand command)
    {
        /* Do work */
        return base.Handle(command);
    }
}
```

In the second example, **MyIncompleteHandler** will not be run in the pipeline.

```
class MyIncompleteHandler : RequestHandler<MyCommand>
{
    [FeatureSwitch(typeof(MyIncompleteHandler), FeatureSwitchStatus.Off, step: 1)]
    public override MyCommand Handle(MyCommand command)
    {
        /* Nothing implmented so we're skipping this handler */
        return base.Handle(command);
    }
}
```

Building a config for Feature Switches with FluentConfigRegistryBuilder

We provide a **FluentConfigRegistryBuilder** to build a mapping of request handlers to **FeatureSwitchStatus**. For each Handler that you wish to feature switch you supply a type and a status using a fluent API. The valid statuses used in the builder are **FeatureSwitchStatus.On** and **FeatureSwitchStatus.Off**.

```
var featureSwitchRegistry = FluentConfigRegistryBuilder
                            .With()
                            .StatusOf<MyFeatureSwitchedHandler>().
↪Is(FeatureSwitchStatus.On)
                            .StatusOf<MyIncompleteHandler>().Is(FeatureSwitchStatus.
↪Off)
                            .Build();
```

Implementing a custom Feature Switch Registry

The **FluentConfigRegistryBuilder** provides compile time configuration of **FeatureSwitch** Attributes. If this is not suitable to your needs then you can write you own Feature Switch Registry using the IAmAFeatureSwitchRegistry interface. The two requirements of this interface is a MissingConfigStrategy, and an implementation of **StatusOf(Type type)** which returns a **FeatureSwitchStatus**.

The **MissingConfigStrategy** determines how the Command Processor should behave when a Handler is decorated with a **FeatureSwitch** Attribute that is set to **FeatureSwitchStatus.Config** does not exist in the registry.

Your implementation of the **StatusOf** method is used to determine the **FeatureSwitchStatus** of the Handler type that is passed in as a parameter. How you store and retrieve these configurations is then up to you.

In the following example there are two FeatureSwitches configured in the **AppSettings.config**. We then build an **AppSettingsConfigRegistry**. The **StatusOf** method is implemetned to read the config from the App Settings and return the status for the given type.

```
<appSettings>
    <add key="FeatureSwitch::Namespace.MyFeatureSwitchedHandler" value="on"/>
    <add key="FeatureSwitch::Namespace.MyIncompleteHandler" value="off"/>
</appSettings>
```

```
class AppSettingsConfigRegistry : IAmAFeatureSwitchRegistry
{
    public MissingConfigStrategy MissingConfigStrategy { get; set; }

    public FeatureSwitchStatus StatusOf(Type handler)
    {
        var configStatus = ConfigurationManager.AppSettings[$"FeatureSwitch::{handler}
↪"].ToLower();

        switch (configStatus)
        {
            case "on":
                return FeatureSwitchStatus.On;
            case "off":
                return FeatureSwitchStatus.Off;
            default:
                return MissingConfigStrategy is MissingConfigStrategy.SilentOn
                            ? FeatureSwitchStatus.On
                            : MissingConfigStrategy is MissingConfigStrategy.SilentOff
                                ? FeatureSwitchStatus.Off
                                : throw new InvalidOperationException($"No Feature␣
↪Switch configuration for {handler} specified");
        }
    }
}
```

Setting Feature Switching Registry

We associate a **Feature Switch Registry** with a **Command Processor** by passing it into the constructor of the **Command Processor**. For convenience, we provide a **Commmand Processor Builder** that helps you configure new instances of **Command Processor**.

```
var featureSwitchRegistry = FluentConfigRegistryBuilder
                            .With()
                            .StatusOf<MyFeatureSwitchedConfigHandler>().
↪Is(FeatureSwitchStatus.Off)
                            .Build();

var builder = CommandProcessorBuilder
                    .With()
                    .Handlers(new HandlerConfiguration(_registry, _handlerFactory))
                    .DefaultPolicy()
                    .NoTaskQueues()
                    .ConfigureFeatureSwitches(fluentConfig)
```

```
                    .RequestContextFactory(new InMemoryRequestContextFactory());

var commandProcessor = builder.Build();
```

Frquently Asked Questions

Must I manually register all my handlers?

No, of course not. Brighter is intended as a library, not a framework, (see elsewhere) and as such we avoid holding an opinion about how you will create the factories which we call to create instances of your types, or how you will choose to register them. However, many people implement a factory by wrapping their IoC container of choice. (We do this in the samples, wrapping TinyIOC in a number of them). Here is a common approach:

- Implement the HandlerFactory and MessageMapperFactory using your IOC container.
- Use reflection to scan your assemblies for message mappers and request handlers
- Register them with your IOC container
- Register them with SubscriberRegistry or MessageMapperRegistry as appropriate.

This approach means that you do not need to worry about updating your setup to add new handlers and message mappers as you develop them.

In addition, you can write a generic message mapper for those cases where you simply serialize all the properties of a type, and have a strategy for naming the topic that is derived from the class name.

This approach liberates you from boilerplate code for each message mapper. It is most useful where your mapping library for serializing to and from the message (i.e. json serialization) is tolerant to new properties it does not understand.

Is Brighter a Framework or a Library?

The most common way to distinguish between a library and a Framework is that your code calls a library, whereas a library calls your code.

Under this definition Brighter is a library: you create the command dispatcher, and call it, in order to route messages. Although the command processor calls your handler pipeline, and thus calls your factories and handlers, this is just user initiated routing.

However, Service Activator would be considered a Framework because the dispatcher is the application, and whilst you must initiate it in the host, Brighter then runs your code, dispatching requests from queues to your handlers.

Moving away from this technical definition of the difference though, our objective is to have a low footprint, that does not dictate choices in your app