A Brighter Future

Dispatchers, Processors, and Task Queues
Who are you?

• Software Developer for 20 years
  – Worked mainly for ISVs
    • Reuters, SunGard, Misys, Huddle
  – Worked for a couple of MIS departments
    • DTI, Beazley

• Microsoft MVP for C#
  – Interested in OO, SOA, EDA,, Messaging, REST
  – Interested in Agile methodologies and practices

• No smart guys
  – Just the guys in this room
Intelligent collaboration for the Enterprise
Agenda

- The Command Invoker Pattern
- A Tour of Brighter
- Command Dispatcher
  - The Command Dispatcher Pattern
- The Task Queue Pattern
  - Service Activator Pattern
- Command Processor
  - The Command Processor Pattern
  - Timeout, Retry & Circuit Breaker

- Configuring Brighter:
  - The problem with Inversion of Control
  - Configuration Steps
- Management and Monitoring
  - The Control Bus
- Where do I go now?
  - Places to go, people to see
- Bright Future
  - Roadmap
  - Q&A
Things that might help

• If you are having trouble reading from the back
• This slide deck is available at https://github.com/iancooper/Presentations
• The screencasts are in the same repository
• The code is at https://github.com/iancooper/Brighter-Tutorial
All presentations are equal...

... But some are more equal than others.

I give this presentation at user groups, internal brown bags, and conferences.

That can give me 120 mins, 90 mins, 60 mins, or 45 mins.

So we may skip some content if you are in a shorter session... Don’t panic, it’s ordered by importance.
Endpoints should be free of domain concerns

THE COMMAND INVOKER
Command Invoker Pattern

Problem: How can a service API reuse common domain logic while enabling both synchronous and asynchronous request processing?

Solution: Create command objects that fully encapsulate common request processing logic. Instantiate and invoke these commands from within the service, or forward them to an asynchronous background process.

When a service is implemented as a **Command Invoker**, all domain and exception-handling logic is removed from the service and moved to Command objects.

The code that is left over in the service does very little.

In its simplest form, the service selects a Command, provides request data to the Command, and calls a method that causes it to begin processing the request.
Implementation Options

• Invoking commands from within the service
  – Have the service instantiate the command and execute it directly.
    • Processing time is short and spikes in load are not an issue.
  – Forward the request to a background process
    • Time required is not short or spikes in load occur.
    • Can instantiate the command and forward to a background request processor by means of a queue or database.
    • Can also forward request to background process that instantiates the command and services it.

• Implementing a command
  – May use a transaction script; no domain just direct manipulation of state in the database.
  – May use a domain model
Always look on the bright side of life

A TOUR OF BRIGHTER
Hello World

In this demo we show you how to use a command dispatcher to implement the classic Hello World application.

Our goal is to show you Brighter’s Command Dispatcher, as simply as possible
How do I get Brighter?

We are an MIT-licensed library available via NuGet.

Look for `paramore.brighter.*`

`paramore.brighter.commandprocessor` is the core package for command dispatcher and command processor.

`paramore.brighter.serviceactivator` is the core package for tasks queues.

Most of the other packages in `paramore.brighter` are implementations of the abstract components defined in the other two libraries.

The source is on Github.
THE COMMAND DISPATCHER
Command Design Pattern

The command design pattern **encapsulates a request as an object**, allowing **reuse**, **queuing** or **logging** of requests, or **undoable** operations. It also serves to decouple the implementation of the request from the requestor. The caller of a command object does not need to understand how the command is actioned, only that the command exists.

When the caller and the implementer are decoupled it becomes easy to replace or refactor the implementation of the request, without impacting the caller - our system is more modifiable.

Our ability to test the command in isolation of the caller - allows us to test our behaviour without mocking a framework - we can instantiate the command, provide 'fake' parameters to it and confirm the results.

We can also re-use the command from multiple callers.

**Command** - Declares an interface for executing an operation.

**ConcreteCommand** – Defines a binding between a **Receiver** object and an action. Implements Execute by invoking the corresponding operation(s) on Receiver.

**Client** – creates a ConcreteCommand object and sets its receiver.

**Invoker** - asks the command to carry out the request.
Command Sequence Diagram

• An Invoker object knows about the concrete Command object.
• The Invoker issues a request by calling Execute on the Command.
• When commands are un-doable, the Command stores state for undoing the command prior to invoking Execute.
• The Command object invokes operations on its Receiver to carry out the request
Command Design Pattern

In addition we can structure a system transactionally using commands.

A command is essentially a transactional boundary.

We can generalize the pattern for a command as:

1: Begin Transaction
2: Load from Repository
3: Operate on Domain
4: Flush to Repository
5: Commit Transaction
6: Invoke Event Encapsulating Notification

- In DDD we call the object graph that is a transactional boundary an **Aggregate** (also a coarse-grained lock).
- We may need to notify other aggregates that can be eventually consistent of the change within the transactionally consistent boundary.
- Because this command is the result of another command, we often call it an Event.
- A Command is an instruction, and Event is a notification.
- The pattern to use here is Observer. A publisher notifies subscriber of an event.
- Because the handling of that notification is in itself likely to be a transactional boundary for a different aggregate we can encapsulate the event with the Command design pattern as well.
- By raising an Event for subscribers, we also allow multiple listeners.
Implementing a Command

Implementing the command pattern is simple.

We implement a command interface, that has a method to execute the command.

We then create a concrete instance of the command that derives from this interface. The invoking class executes the command without being directly coupled to the receiver which the command uses to implement the action requested.

```java
public interface IAmACommand
{
    void Execute();
}

public class UpdateMyThingForFooFooCommand
{
    public UpdateMyThingForFooFooCommand(/* .. parameters ...*/)
    {
        /* Initialize state of command for members */
    }

    public void Execute()
    {
        /*Other Stuff*/
    }
}
```
Command Dispatcher

To decouple we want to be able alter the implementation of the commands that we call without altering the invoker.

We introduce a Command Handler for each Command that can be invoked, that all implement a specific interface. Command-handlers are registered with a Command Dispatcher (and can also be removed) at run-time.

When the application issues a command, it notifies the Command Dispatcher which dispatches to the command handler(s) associated with the command.

This decouples dispatch from processing.

• The Command Dispatcher allows dynamic registration and removal of Command Handlers, it is an administrative entity that manages linking of commands to the appropriate command handlers.
• It is an implementation of the Observer pattern in that hooks together publishers and subscribers.
• Command Dispatcher registration requires a key – provided by the Command Dispatcher for the Commands it can service, using getKey(). [In practice we often use RTTI for this].
• The Command Handler is fired, when a command with the same name (key) is sent to the Command Dispatcher.
• The Command Dispatcher is a repository of key-value pairs (key,, Command Handler) and when the Command Dispatcher is called it looks up the command’s key in the repository. If there is a match it calls the appropriate method(s) on the handler to process the Command.
**Command Dispatcher**

**Invoker** - has a list of *Commands* that are to be executed

**Command** - represents the request to be processed, encapsulating the parameters to be passed to the command-handler to perform the request

**Command Handler** - specifies the interface that any command handler must implement

**Concrete Command Handler** – implements the request

**Command Dispatcher** – Allows dynamic registration of *Command Handlers* and looks up handlers for commands, by matching command and handler key.

**Client** – registers Commands with the Command Dispatcher.
Command Dispatcher

We want to separate an Action-Request object that contains the identity of the action we want to perform, and the parameter for that action from the Action-Handler which knows how to perform that action.

A Command Dispatcher is an object that links the Action-Request with the appropriate Action-Handler.

We may distinguish between a Command Action-Request that has one Action Handler and an Event Action-Request that has many
Dispatching a Command

Implement a Request Handler

```csharp
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&lt;GreetingCommand&gt; and accept a parameter of that type on the overridden Handle() method.

```csharp
class GreetingCommandHandler : RequestHandler&lt;GreetingCommand&gt;
{
    public GreetingCommandHandler(ILog logger) : base(logger) {}
    public override GreetingCommandHandle(GreetingCommand command)
    {
        Console.WriteLine("Hello {0}", command.Name);
        return base.Handle(command);
    }
}
```

Dispatch to a 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.

```csharp
var subscriberRegistry = new SubscriberRegistry();
subscriberRegistry.Register&lt;GreetingCommand, GreetingCommandHandler&gt;();
```

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.

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

In this demo we go into detail on how to set up dispatch for commands using Brighter.

First, we will show you the pattern for a handler and registering your handler to receive commands. We will then show you sending a command to that handler.

Next, we will modify the code to show an Event being published, and show you a handler subscribing to that Event.
THE TASK QUEUE PATTERN
Task Queues

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.

- **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 acknowledging the request, and completing the work asynchronously.**
  - The downside of the asynchronous 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.**
Lightweight Broker Architecture
Posting to a Task Queue

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

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

_commandProcessor.Post(reminderCommand);
```
Serializing the Message

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

```csharp
public class TaskReminderCommandMessageMapper : IAmAMessageMapper<TaskReminderCommand>
{
    public Message MapToMessage(TaskReminderCommand request)
    {
        var header = new MessageHeader(messageId: request.Id, topic: "Task.Reminder", messageType
        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);
    }
}
```
Handling the Message

Then you write a handler as normal.

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

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

    public GreetingService()
    {
        /* Configuration 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;
    }
}
Queue

In this demo we will look at implementing a long running operation – sending a confirmation email – using a task queue – allowing us to ack the original request rapidly without waiting for the operation to complete.

We will show a Windows service in TopShelf that is able to consume messages from a queue, and dispatch them to a handler.

We will also show sending messages to that queue from the existing HTTP API endpoint.
Controlling Service Quality at Ports

THE COMMAND PROCESSOR
Command Processor

The Command Processor pattern separates the request for a service from its execution.

A Command Processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

A Command Dispatcher and a Command Processor are similar in that both divorce the caller of a Command from invoker of that Command.

However, the motivation is different. A Dispatcher seeks to decouple the caller from the invoker to allow us to easily extend the system without modification to the caller.

Conversely the motivation behind a Command Processor is to allow us to implement orthogonal operations such as logging, or scheduling without forcing the sender or receiver to be aware of them. It does this by giving those responsibilities to the invoker.
Command Processor

The Command Processor pattern provides details of how to manage a set of commands used to separate the request for service from its execution. The Command Processor accepts requests for Commands, schedules and executes them, manages their history (such as storing for later undo or replay) and performs any other orthogonal housekeeping operations. The client is decoupled from the responsibilities of managing and scheduling commands.

A Command Processor enforces quality of service and maximizes throughput.

A Command Processor forms a juncture at which concerns like: retry, timeout and circuit breaker can be implemented for all commands.
Building a Pipeline of Request Handlers

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-sequecing 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 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 named 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.
Implementing a Pipeline

Create your own Attribute class, derived from RequestHandlerAttribute.

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

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

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

Create your handler, initialize from attribute if required.

```csharp
using System;
using Newtonsoft.Json;
using paramore.brighter.commandprocessor.Logging;

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

        public RequestLoggingHandler(IRequest logger)
        {
            base(logger);
        }

        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}
            _timing.ToString(),
typeof(TRequest).
JsonConvert.SerializeObject(request).
DateTime.UtcNow);
        }
    }
}
```
Building a Pipeline of Request Handlers

In this demo we will look at using a pipeline to handle a request.

We will look at adding out-of-box attributes to a handler and then we will look at building your own handler for a pipeline step.
Availability

When building a service we need to think about key patterns to support Availability.

- Retry
- Timeout
- Circuit Breaker

The Command Processor pattern can be used to provide support for these Availability concerns, allowing the handler code to focus on surfacing the domain model and not infrastructure concerns.

The Fallacies of Distributed Computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.
The Timeout Pattern

Provide a timeout on any inter-process operation.

The timeout pattern is usable with any client-server request.

The timeout pattern is also usable with thread resource pools.

Any operation that blocks a thread, should have a timeout.

Usually a timeout is a parameter on an API call that indicates how long to wait [in milliseconds] for the operation to complete before returning control to the caller.

If you are using a third-party API that has no timeout, you may need to make the request on a thread that does have a timeout, and when that thread times out assume that the operation failed.

• Timeouts are a scaling issue. Because an API call without a timeout consumes resources, a stalled network call, or thread, prevents other requests from being executed. You might not notice the failure to use timeouts on a system with abundant resources, but as your resources dwindle failure to use timeouts to free resources will cause pain.

• Assume that a timeout will occur at some point

• Asynchronous operations still need to time out. All the caller may not be blocked, resources are still being consumed to manage the asynchronous operation that will not be released until the operation aborts.
The Retry Pattern

In the presence of unreliable calls between two components our first approach to achieving a high quality of service is to recognize that many such failures are transient: a timeout because a resource is busy, temporary loss of connectivity, the loss of one node that will be replaced by failover to another.

In this case the fault is self-correcting, the node comes up, the load on the database or server declines and their is capacity for our call, or network connectivity is restored. This means that our call will succeed if we retry after a delay to allow the transient fault to resolve.

The Retry pattern is simply that if the call fails, we can try again. It is important to have an upper bound on retries in case a fault that appears transient is not. See Circuit Breaker as well.

- If the fault is permanent, and unlikely to succeed, for example a login failure due to invalid credentials – don’t retry.
- If the fault is caused by a rare event, for example packet loss or other corruption, consider an immediate retry as the server may be able to respond.
- If the fault is caused by load, such as SQL Timeouts, or 429 Too Many Requests, then back-off for a period before retrying. Failure to observe this can lead to an Application Denial of Service Attack i.e. we overload a struggling server.
- In the case of a 429 for example, the Retry-After header will tell you how long to back off for.
The Circuit Breaker Pattern

The Circuit Breaker pattern prevents an application from executing an operation that is likely to fail, thus freeing up resources that would otherwise be consumed waiting for a timeout and retry cycle to occur.

Once normal service has been restored to the server or resource pool the circuit breaker pattern allows detection of the resumption of service.

A Circuit Breaker acts as a proxy to operations that can fail. It has one of three states:

Closed: Requests are routed as normal, on a failure a counter is incremented and if the threshold is exceeded within a time limit, the circuit breaker opens.

Open: No calls are allowed, and are failed automatically by the proxy. After a specified time interval the circuit breaker is moved to half-open

Half-Open: A call is allowed. On a failure the breaker moves to Open, on a success it moves to Closed

Adding Availability Patterns to the Brighter pipeline

In this demo we show you how to add attributes to your handlers that provide support for availability.

We also discuss the Polly library.

In addition we will talk about Fallback scenarios.
Hooking it all together

CONFIGURING BRIGHTER
Command Processor

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 Commands or Events 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 an logger to map to LibLog's ILog
- You need to provide a Request Context Factory
- If you want to Post a message to a Task Queue then you need to provide configuration information so that we can connect to the Broker to send a message.
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:

```scala
val commandProcessor = CommandProcessorBuilder.With()
  .Handlers(new HandlerConfiguration(subscriberRegistry, handlerFactory))
  .Policies(policyRegistry)
  .Logger(logger)
  .NoTaskQueues()
  .RequestContextFactory(new InMemoryRequestContextFactory())
  .Build();
```
Dispatcher

Two Part Configuration
When dealing with infrastructure we provide a two-part configuration strategy: code and configuration.

If the configuration setting is decided at **design time**, we put it into code.

If the configuration setting is decided at **run time**, we put it into a configuration file.

This allows us to make changes to deployed software, without rebuilding the software where those choices relate to deployment into an environment.

What you need to provide

- You need to provide an **ILog** logger to us so that we can provide diagnostics to you.
- You need to provide a **Command Processor**, which we use to dispatch messages to handlers.
- We use **Message Mappers** to de-serialize a message into an **IRequest** that we can dispatch.
- We need a **Channel Factory** which abstracts opening a channel/queue for your broker so we can read messages from it.
- We need a **Connection List** which tells us what messages to read from which channels. We use a datatype channel approach.
// create message mappers
var messageMapperRegistry = new MessageMapperRegistry(messageMapperFactory)
{
    {typeof(GreetingCommand), typeof(GreetingCommandMessageMapper)}
};

// create the gateway
var rmqMessageConsumerFactory = new RmqMessageConsumerFactory(logger);
var builder = DispatchBuilder
    .With()
    .Logger(logger)
    .CommandProcessor(CommandProcessorBuilder.With()
        .Handlers(new HandlerConfiguration(subscriberRegistry, handlerFactory))
        .Policies(policyRegistry)
        .Logger(logger)
        .NoTaskQueues()
        .RequestContextFactory(new InMemoryRequestContextFactory())
    .Build()
    .MessageMappers(messageMapperRegistry)
    .ConnectionFactory(new InputConnectionFactory(rmqMessageConsumerFactory))
    .ConnectionsFromConfiguration();
_dispatcher = builder.Build();
What’s Going On

MANAGEMENT AND MONITORING
The Control Bus

Application Message Flow
The Application Message Flow is the stream of messages flowing from producer to consumer in our application that we use to carry out the application’s work.

The use of messaging results in a distributed system. An question soon becomes: how do we manage and monitor the components in that system?

One solution is to use the messaging middleware itself, listening on a channel for information about the behaviour of the system and sending command and in response sending control messages to components over the messaging infrastructure.

The control bus is the set of channels etc. that we use for control flow as opposed to application flow. They are separate so that control does not impact application and vice-versa.
Control Bus

- **Configuration**: each component has configurable parameters (timeouts, formats, resources, collaborators etc.). Components listen to the control bus for configuration changes.
  - In Brighter we let you control the connections in a Service Activator over the Control Bus
- **Heartbeat**: Allows a component on the Control Bus to monitor the health of other components in the system. It may be able to re-route or re-introduce in response to failure.
- **Test Message**: Allows us to determine message flow, performance etc. by sending a tracer bullet through the infrastructure.
- **Exceptions**: Components can send exception conditions to the control bus. This channel can be evaluated to determine if there is a possible compensation, manual or automatic.
- **Statistics**: Feedback on the throughput of the system: number of messages processed, time taken
- **Console**: A Control Bus often has a dashboard to allow monitoring of the health of the system
<table>
<thead>
<tr>
<th>Feature</th>
<th>Today</th>
<th>Tomorrow</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>🟢</td>
<td></td>
<td>Stop or Start a Dispatcher, Introduce New Performers, Add Connections. Introduce support Zero Configuration via Consul.</td>
</tr>
<tr>
<td>Heartbeat</td>
<td>🟢</td>
<td>🟢</td>
<td>Plan to offer advice/support for tooling using Consul etc. to allow heartbeat and restart/re-introduction.</td>
</tr>
<tr>
<td>Test Message</td>
<td></td>
<td>🟥</td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>🟢</td>
<td>🟢</td>
<td>Exceptions can be passed to the Control Bus if using the Monitor attribute. Support coming for an Invalid Message Queue</td>
</tr>
<tr>
<td>Statistics</td>
<td>🟢</td>
<td>🟥</td>
<td>Statistics passed to the Control Bus provided that you use the Monitor attribute.</td>
</tr>
<tr>
<td>Console</td>
<td>🟢</td>
<td>🟢</td>
<td>You can view the statistics via CLI today, visual tomorrow. MessageStore viewer available, CommandStore viewer to come.</td>
</tr>
</tbody>
</table>
Monitoring

In this demo we show you how to monitor your Brighter application so that you can gain greater insight into how Brighter works.
WANT TO KNOW MORE?

Next steps to understand Brighter better.
Visit our GitHub pages for tutorials and docs

http://iancooper.github.io/Paramore/Brighter.html
Clone (or fork!) the GitHub repo
https://github.com/iancooper/Paramore
Join us in our Gitter chat room

https://gitter.im/iancooper/Paramore
I’ve got to wear shades

BRIGHT FUTURE
Road Map

• Dashboard for Control Bus
• *Async Command Dispatcher methods
  – Separate Async and Sync pipelines to avoid issues blocking on async pipelines in ASP.NET etc.
• Python and F# support
  – Dispatcher, Processor and ServiceActivator
• Brokers: Azure Service Bus, Redis, Event Store
• Stores: Postgres SQL