MassTransit Documentation

Release 3.0

Chris Patterson, Dru Sellers, and Travis Smith

Jun 20, 2017

Contents

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit is a free, open source, lightweight message bus for creating distributed applications using the .NET framework. MassTransit provides an extensive set of features on top existing message transports, resulting in a developer friendly way to asynchronously connect services using message-based conversation patterns. Message-based communication is a reliable and scalable way to implement a service oriented architecture.

CHAPTER 1

What is MassTransit?

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit is a free, open source, lightweight message bus for creating distributed applications using the .NET framework. MassTransit provides an extensive set of features on top existing message transports, resulting in a developer friendly way to asynchronously connect services using message-based conversation patterns. Message-based communication is a reliable and scalable way to implement a service oriented architecture.

A bit of the back story?

We are often asked why MassTransit was created, well here's the story. :)

In 2007, Chris Patterson (@phatboyg) and Dru Sellers (@drusellers) met, for what Dru thinks is the first time, at the first ALT.NET conference in Austin, TX. It was at this conference that Chris and Dru not only realized that they had a lot of the same problems to solve, but also how much the standard tooling provided by Microsoft just didn't fit their needs. Surrounded by the best and brightest in .NET, the energy was there to build better tooling that supported testable processes. Combined with an awareness of the latest advances in tooling, libraries, and coding practices; they decided that a better option must exist. After searching the .NET ecosystem for a tool that would help them achieve their goals, the only real option was the venerable NServiceBus. After reviewing NServiceBus, it was determined that the only real dependency injection container supported was Spring.NET. It also become obvious that NServiceBus wasn't quite ready for external contributors to come onboard. For these reasons, they decided to embark on the quixotic trek of building their own service bus (seriously, how hard could it be?? LOL).

Initially the goals were as much about learning distributed message based systems, as well as building something both of their companies could use. The first commit was pushed to GoogleCode on 12/26/2007, and shortly there after both Dru and Chris went to production with MassTransit and both of their companies have had success in getting value out of their efforts.

After four years of continued success, Chris and Dru continued to push forward on their Journey, and were joined by Travis Smith (@TravisTheTechie). The near future should bring much for the MassTransit community as RabbitMQ became the broker of choice, lessening the focus on MSMQ.

In early 2014, after a few years of research and design, work was started on an entirely new MassTransit. In order to embrace the world of asynchronous programming, as well as leveraging the power of advanced messaging platforms like RabbitMQ, a foundational rewrite was required. Much of the code in MassTransit was written prior to the introduction of the Task Parallel Library (or TPL), and even the .NET 4.0 support was before async and await were added to the language.

To eliminate a ton of extremely complex code, support for MSMQ was completely ripped out, including all of the routing support that had to built because of MSMQ's lack of message routing. The remaining code was rewritten from bottom to top, resulting in an entirely new, completely asynchronous, and highly optimized framework for message processing.

The philosophy

First and foremost, we are not an Enterprise Service Bus (ESB). While MassTransit is used in several enterprises it isn't a swiss army knife, we are not driven by sales to be a million features wide, and an inch deep. We focus on a few key concepts and try to make them as robust as possible.

We don't do doodleware, you won't find a designer, we are all about the keyboard samurais, the true in-the-trenches coder. That's who we are, and those are our friends. If you want to draw, use a whiteboard.

We don't do FTP->WS-deathstar->BS (not that you can't, its just not in the box). We focus on the experience of using one transport in a given environment, and we try to make it as smooth as possible.

MassTransit is built to be used inside the firewall, its not built to be used as a means to communicate with external vendors (it can be, again its just not in the box), its meant to be used for getting your corporate services talking to each other and making building internal software easier.

CHAPTER 2

Installing MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

This section of the online docs will explain how to get MassTransit into your project. It will also show you where to get help, how to report bugs, etc. Hopefully, you will find it useful as you explore the MassTransit framework.

Prerequisites

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit is written in C# and requires a compatible .NET framework. MassTransit is built and tested with .NET 4.5, and the shipped assemblies target this version. Visual Studio 2015 is used, and the new C# 6 syntax is used.

For comparison, my development machines is a MacBook Pro hosting a Windows Server 2012 R2 virtual machine (via VMware Fusion), with Visual Studio 2015 and Resharper 9.x installed. My secondary rig is a Razer Blade running Windows 10, which also works just fine.

Note: With the release of Mono 4.0, and the upcoming release of .NET Core, MassTransit will be soon target both Windows and Mac/Linux applications. At this time, however, this has not been fully tested by the primary contributors.

Transports

MassTransit leverages existing message transports, so you will need to have a supported transport installed.

In Memory

The in memory transport is included with MassTransit. No additional software is required.

RabbitMQ

To use RabbitMQ, download and install the version appropriate for your operating system. Once the broker is installed, enable some additional plug-ins for management and message tracking.

Then, install the `MassTransit.RabbitMQ` package in your project and follow the RabbitMQ configuration guide.

Azure Service Bus

Azure Service Bus is a generic, cloud-based messaging system for connecting just about anything—applications, services, and devices—wherever they are. Connect apps running on Azure, on-premises systems, or both. You can even use Service Bus to connect household appliances, sensors, and other devices like tablets or phones to a central application or to each other.

To use Azure Service Bus with MassTransit, install the `MassTransit.AzureServiceBus` package in your project and follow the Service Bus configuration guide.

How to install

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

NuGet

The simplest way to install MassTransit into your solution/project is to use NuGet.

```
nuget Install-Package MassTransit
```

However, the NuGet packages don't contain the MassTransit.RuntimeServices executable and database SQL scripts. The RuntimeServices system routes messages to multiple subscribers via the Subscription Service. If you plan to use the "UseSubscriptionService" feature, then you'll need to get compile that from source.

Then you will need to add references to

- MassTransit.dll
- MassTransit.<Transport>.dll (RabbitMQ and Azure Service Bus)
- MassTransit.<ContainerSupport>.dll (Castle, AutoFac, and StructureMap)

Compiling from source

Lastly, if you want to hack on MassTransit or just want to have the actual source code you can clone the source from github.com.

To clone the repository using git try the following:

```
git clone git://github.com/MassTransit/MassTransit.git
```

Note: The default branch for this project is develop. This is done to make development easier. The master branch in this case represents gold code.

Build dependencies

To compile MassTransit from source you will need the following developer tools installed:

- .NET 4.5 SDK or later

Compiling

To compile the source code, drop to the command line and type:

```
.\build.bat
```

If you look in the `.\build_output` folder you should see the binaries.

Getting Help

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Writing distributed applications is hard, and sometimes you need help. There are several ways to get in touch with the developers behind MassTransit. In most cases, you can expect a response within a few days, sometimes sooner. The speed of the response is going to depend upon how well you've done your homework before raising the white flag.

However, before attempting to contact a developer directly, there are many active forums for support that are available.

Stack Overflow

There is a MassTransit tag on Stack Overflow which has many questions that have already been asked. Several developers regularly monitor this tag for new questions, so that's a great place to start. Be sure to search and see if your question has already been asked, that is the fastest way to an answer if someone else has already experienced the same issue.

Before you just post your question, however, spend a few moments to compose your thoughts and formulate your question. There is nothing as pointless as simply telling us "MassTransit does not work for me" with no further

information to give any clue to why. Before you post, search the web to see if your question hass already been asked or even answered. And if it has been, you will already have your answer.

Mailing List

A long history of support is available on the mailing list, which is hosted on Google Groups. There are a wide vareity of developers using MassTransit on the list, and often times they can help. If you are going to use MassTransit, joining the mailing list is a great way to participate in the community.

When posting on the mailing list, pick a good subject line, and if the subject of the thread changes, please change the subject to match. Some of us deal with hundreds of emails per day and we need all the help we can get to pick the interesting or important ones.

Also, if you are seeing some unexpected behavior, generate a **'diagnostic result'_** and post it as a Gist so that it can be reviewed (mail attachments rarely work, so avoid them). This will help a lot towards understanding how your configuration is setup.

Twitter

A fast way to get our attention is on Twitter, by shooting a tweet tagged `#mtproj`. Assuming you can fit your question in 140 characters or less. A few of us have saved searches on this tag, which makes questions easy to find (strangely, MassTransit is the worst term to search – go figure). There are several spread out across the US, so think about the time zone when you tweet. Again, post a link to a Gist with source, configuration, log files, etc. If it is complex, it's likely you'll be pushed to hit the mailing list.

GitHub Issues

Please do not open an issue on github, unless you have spotted an actual bug in MassTransit. If you are unsure, ask on the mailing list, and if we confirm it's a bug, we'll ask you to create the issue. Issues are not the place for questions, and they'll likely be closed.

This policy is in place to avoid bugs being drowned out in a pile of sensible suggestions for future enhancements and calls for help from people who forget to check back if they get it and so on.

How to report bugs

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

MassTransit can be a tricky beast to debug, with a multi-threaded system, trying to track down the issue can be a bit painful.

So if you run into a bug, please spend a minute collecting the right information to help us fix the bug.

The most valuable piece of information you can give us, is always give us a failing unit test, if you can't give us that then how to reproduce the bug in a step by step fashion. Other wise its going to be a lot of back and forth until we can better understand and get to a failing unit test.

CHAPTER 3

Using MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

In order to use and understand MassTransit, a set of recipe-based usage examples are provided below.

Creating a message contract

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

In MassTransit, a message contract is defined using the .NET type system. Messages can be defined using both classes and interfaces, however, it is suggested that types use read-only properties and no behavior.

Note: It is strongly suggested to use interfaces for message contracts, based on experience over several years with varying levels of developer experience. MassTransit will create dynamic interface implementations for the messages, ensuring a clean separation of the message contract from the consumer.

An example message to update a customer address is shown below.

```
namespace Company.Application.Contracts
{
        using System;
```

```
        public interface UpdateCustomerAddress
        {
                Guid CommandId { get; }
                DateTime Timestamp { get; }
                string CustomerId { get; }
                string HouseNumber { get; }
                string Street { get; }
                string City { get; }
                string State { get; }
                string PostalCode { get; }
        }
}
```

A common mistake when engineers are new to messaging is to create a base class for messages, and try to dispatch that base class in the consumer – including the behavior of the subclass. Ouch. This always leads to pain and suffering, so just say no to base classes.

Specifying message names

There are two main message types, *events* and *commands*. When choosing a name for a message, the type of message should dictate the tense of the message.

Commands

A command tells a service to do something. Commands are sent (using `Send`) to an endpoint, as it is expected that a single service instance performs the command action. A command should never be published.

Commands should be expressed in a verb-noun sequence, following the *tell* style.

Example Commands:

- UpdateCustomerAddress
- UpgradeCustomerAccount
- SubmitOrder

Events

An event signifies that something has happened. Events are published (using `Publish`) using either `IBus` or the `ConsumeContext` within a message consumer. An event should never be sent directly to an endpoint.

Events should be expressed in a noun-verb (past tense) sequence, indicating that something happened.

Example Events:

- CustomerAddressUpdated
- CustomerAccountUpgraded
- OrderSubmitted, OrderAccepted, OrderRejected, OrderShipped

Correlating messages

There are several built-in message headers that can be used to correlate messages. However, it is also completely acceptable to add properties to the message contract for correlation. The default headers available include:

CorrelationId An explicit correlation identifier for the message. If the message contract has a property named `CorrelationId`, `CommandId`, or `EventId` this header is automatically populated on Send or Publish. Otherwise, it can be manually specified using the `SendContext`.

RequestId When using the `RequestClient`, or the request/response message handling of MassTransit, each request is assigned a unique `RequestId`. When the message is received by a consumer, the response message sent by the `Respond` method (on the `ConsumeContext`) is assigned the same `RequestId` so that it can be correlated by the request client. This header should not typically be set by the consumer, as it is handled automatically.

ConversationId The conversation is created by the first message that is sent or published, in which no existing context is available (such as when a message is sent or published from a message consumer). If an existing context is used to send or publish a message, the `ConversationId` is copied to the new message, ensuring that a set of messages within the same *conversation* have the same identifier.

InitiatorId When a message is created within the context of an existing message, such as in a consumer, a saga, etc., the `CorrelationId` of the message (if available, otherwise the `MessageId` may be used) is copied to the `InitiatorId` header. This makes it possible to combine a chain of messages into a graph of producers and consumers.

MessageId When a message is sent or published, this header is automatically generated for the message.

Creating a message consumer

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

A message consumer is a class that consumes one or more message types, specified by the `IConsumer<T>` interface, where `T` is the message type.

```
public class UpdateCustomerConsumer :
    IConsumer<UpdateCustomerAddress>
{
    public async Task Consume(ConsumeContext<UpdateCustomerAddress> context)
    {
        await Console.Out.WriteLineAsync($"Updating customer: {context.Message.
↪CustomerId}");

        // update the customer address
    }
}
```

When a consumer is subscribed to a receive endpoint, and a message consumed by the consumer is received by the endpoint, an instance of the consumer is created (using a consumer factory, which is covered –> here <–). The message (wrapped in a `ConsumeContext`) is then delivered to the consumer via the `Consume` method.

The `Consume` method is asynchronous, and returns a Task. The task is awaited by MassTransit, during which time the message is unavailable to other receive endpoints. If the consume method completes successfuly (a task status of RanToCompletion), the message is acknowledged and removed from the queue.

Note: If the consumer faults (such as throwing an exception, resulting in a task status of Faulted), or is somehow cancelled (TaskStatus of Canceled), the exception is propagated back up the pipeline where it can ultimately be retried or moved to an error queue.

Connecting a message consumer

For a consumer to receive messages, the consumer must be connected to a receive endpoint. This is done during bus configuration, particularly within the configuration of a receive endpoint.

An example of connecting a consumer to a receive endpoint is shown below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Consumer<UpdateCustomerConsumer>();
    });
});
```

The example creates a bus, which connects to the RabbitMQ running on the local machine, using the default username and password (guest/guest). On that bus. a single receive endpoint is created with the name *customer_update_queue*. The consumer is connected using the simplest method, which accepts a consumer class with a default constructor.

Note: When a consumer is connected to a receive endpoint, the combined set of message types consumed by all of the consumers connected to the same receive endpoint are *subscribed* to the queue. The subscription method varies by broker, in the case of RabbitMQ exchange bindings are created for the message types to the exchange/queue for the receive endpoint.

These subscriptions are persistent, and remain in place after the process exits. This ensures that messages published or sent that would be delivered to one of the receive endpoint consumers are saved even is the process is terminated. When the process is started, messages waiting in the queue will be delivered to the consumer(s).

The above example connects the consumer using a default constructor consumer factory. There are several other consumer factories supported, as shown below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = ...;

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        // an anonymous factory method
        e.Consumer(() => new YourConsumer());
```

```
        // an existing consumer factory for the consumer type
        e.Consumer(consumerFactory);

        // a type-based factory that returns an object (container friendly)
        e.Consumer(consumerType, type => container.Resolve(type));

        // an anonymous factory method, with some middleware goodness
        e.Consumer(() => new YourConsumer(), x =>
        {
            // add middleware to the consumer pipeline
            x.UseLog(ConsoleOut, async context => "Consumer created");
        });
    });
});
```

Connecting to an existing bus

Once a bus has been created, the receive endpoints have been created and cannot be modified. The bus itself, however, provides a temporary (auto-delete) queue which can be used to receive messages. To connect a consumer to the bus temporary queue, a series of *Connect* methods can be used.

> **Warning:** Published messages will not be received by the temporary queue. Because the queue is temporary, when consumers are connected no bindings or subscriptions are created. This makes it very fast for transient consumers, and avoid thrashing the message broker with temporary bindings.

The temporary queue is useful to receive request responses and faults (via the response/fault address header) and routing slip events (via an event subscription in the routing slip).

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = ...;
});

busControl.Start();

ConnectHandle handle = busControl.ConnectConsumer<FaultConsumer>();
...
handle.Disconnect(); // disconnect the consumer from the bus pipeline
```

In addition to the `ConnectConsumer` method, methods for each consumer type are also included (`ConnectHandler`, `ConnectInstance`, `ConnectSaga`, and `ConnectStateMachineSaga`).

Connecting an existing consumer instance

While using a consumer instance per message is highly suggested, it is possible to connect an existing consumer instance which will be called for every message. The consumer *must* be thread-safe, as the `Consume` method will be called from multiple threads simultaneously. To connect an existing instance, see the example below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = ...;

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
```

```
    {
        e.Instance(existingConsumer);
    });
});
```

Handling undeliverable messages

If the configuration of an endpoint changes, or if a message is mistakenly sent to an endpoint, it is possible that a message type is received that does not have any connected consumers. If this occurs, the message is moved to a *_skipped* queue (prefixed by the original queue name). The original message content is retained, and additional headers are added to indicate the host which moved the message.

Handling messages without a consumer

While creating a consumer is the preferred way to consume messages, it is also possible to create a simple message handler. By specifying a method, anonymous method, or lambda method, a message can be consumed on a receive endpoint.

To configure a simple message handler, refer to the example below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = ...;

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Handler<UpdateCustomerAddress>(context =>
            return Console.Out.WriteLineAsync($"Update customer address received:
↪{context.Message.CustomerId}"));
    });
});
```

In this case, the method is called for each message received. No consumer is created, and no lifecycle management is performed.

Observing messages via IObserver

With the addition of the `IObserver` interface, the concept of an observer was added to the .NET framework. MassTransit supports the direct connection of observers to receive endpoints.

Note: Unfortunately, observers are not asynchronous. Because of this, it is not possible to play nice with the async support provided by the compiler when using an observer.

An observer is defined using the built-in `IObserver<T>` interface, as shown below.

```
public class CustomerAddressUpdatedObserver :
    IObserver<ConsumeContext<CustomerAddressUpdated>>
{
    public void OnNext(ConsumeContext<CustomerAddressUpdated> context)
    {
        Console.WriteLine("Customer address was updated: {0}", context.Message.
↪CustomerId);
```

```
    }

    public void OnError(Exception error)
    {
    }

    public void OnCompleted()
    {
    }
}
```

Once created, the observer is connected to the receive endpoint similar to a consumer.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = ...;

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Observer<CustomerAddressUpdatedObserver>();
    });
});
```

Producing messages

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

An application or service can produce messages using two different methods. A message can be sent using `Send` or a message can be published using `Publish`. The behavior of each method is very different, but it easy to understand by looking at the type of messages involved with each particular method.

When a message is sent, it is delivered to a specific endpoint using a `DestinationAddress`. When a message is published, it is not sent to a specific endpoint, but is instead broadcasted to any consumers which have *subscribed* to the message type. For these two separate behavior, we describe messages sent as commands, and messages published as events.

Note: These are discussed in depth in the *Creating a message contract* section of the documentation.

Sending commands

Sending a command to an endpoint requires an `ISendEndpoint` reference, which can be obtained from any send endpoint provider (an object that supports `ISendEndpointProvider`). The application should always use the object closest to it to obtain the send endpoint, and it should do it every time it needs it – the application should not cache the send endpoint reference.

For instance, an `IBus` instance is a send endpoint provider, but it should *never* be used by a consumer to obtain an `ISendEndpoint`. `ConsumeContext` can also provide send endpoints, and should be used since it is *closer* to the

consumer.

Note: This cannot be stressed enough – always get send endpoints from the closest interface to the application code. There is extensive logic to tie message flows together using conversation, correlation, and initiator identifiers. By skipping a level and going outside the closest scope, that information can be lost which prevents the useful trace identifiers from being properly handled.

To obtain a send endpoint from a send endpoint provider, use the GetSendEndpoint() method as shown below. Once the send endpoint is returned, it can be used to a send a message.

```
public async Task SendOrder(ISendEndpointProvider sendEndpointProvider)
{
        var endpoint = await sendEndpointProvider.GetSendEndpoint(_serviceAddress);

        await endpoint.Send(new SubmitOrder(...));
}
```

There are many overloads for the `Send` method. Because MassTransit is built around filters and pipes, pipes are used to customize the message delivery behavior of Send. There are also some useful overloads (via extension methods) to make sending easier and less noisy due to the pipe construction, etc.

Sending via interfaces

Since the general recommendation is to use interfaces, there are convenience methods to initialize the interface without requiring the creation of a message class underneath. While versioning of messages still requires a class which supports multiple interfaces, a simple approach to send an interface message is shown below.

```
public interface SubmitOrder
{
        string OrderId { get; }
        DateTime OrderDate { get; }
        decimal OrderAmount { get; }
}

public async Task SendOrder(ISendEndpoint endpoint)
{
        await endpoint.Send<SubmitOrder>(new
        {
                OrderId = "27",
                OrderDate = DateTime.UtcNow,
                OrderAmount = 123.45m
        });
}
```

Setting message headers

There are a variety of message headers available which are used for correlation and tracking of messages. It is also possible to override some of the default behaviors of MassTransit when a fault occurs. For instance, a fault is normally *published* when a consumer throws an exception. If instead the application wants faults delivered to a specific address, the `FaultAddress` can be specified via a header. How this is done is shown below.

```
public interface SubmitOrder
{
```

```
        string OrderId { get; }
        DateTime OrderDate { get; }
        decimal OrderAmount { get; }
}

public async Task SendOrder(ISendEndpoint endpoint)
{
        await endpoint.Send<SubmitOrder>(new
        {
                OrderId = "27",
                OrderDate = DateTime.UtcNow,
                OrderAmount = 123.45m
        }, context => context.FaultAddress = new Uri("rabbitmq://localhost/order_
↪faults"));
}
```

Publishing events

Messages are published similarly to how messages are sent, but in this case, a single `IPublishEndpoint` is used. The same rules for endpoints apply, the closest instance of the publish endpoint should be used. So the `ConsumeContext` for consumers, and `IBus` for applications that are published outside of a consumer context.

To publish a message, see the code below.

```
public interface OrderSubmitted
{
        string OrderId { get; }
        DateTime OrderDate { get; }
}

public async Task NotifyOrderSubmitted(IPublishEndpoint publishEndpoint)
{
        await publishEndpoint.Publish<OrderSubmitted>(new
        {
                OrderId = "27",
                OrderDate = DateTime.UtcNow,
        });
}
```

Correlation identifiers

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

In a distributed message-based system, message correlation is very important. Since operations are potentially executing across hundreds of nodes, the ability to correlate different messages to build a path through the system is absolutely necessary for engineers to troubleshoot problems.

The headers on the message envelope provided by MassTransit already make it easy to specify correlation values. In fact, most are setup by default if not specified by the developer.

MassTransit provides the interface `CorrelatedBy<T>`, which can be used to setup a default correlationId. This is used by sagas as well, since all sagas have a unique `CorrelationId` for each instance of the saga. If a message implements `CorrelatedBy<Guid>`, it will automatically be directed to the saga instance with the matching identifier. If a new saga instance is created by the event, it will be assigned the `CorrelationId` from the initiating message.

For message types that have a correlation identifier, but are not using the `CorrelatedBy` interface, it is possible to declare the identifier for the message type and MassTransit will use that identifier by default for correlation.

```
MessageCorrelation.UseCorrelationId<YourMessageClass>(x => x.SomeGuidValue);
```

Note: This should be called before you start the bus. We currently recommend that you put all of these in a static method for easy grouping and then call it at the beginning of the MassTransit configuration block.

Most transactions in a system will end up being logged and wide scale correlation is likely. Therefore, the use of consistent correlation identifiers is recommended. In fact, using a `Guid` type is highly recommended. MassTransit uses the NewId library to generate identifiers that are unique and sequential that are represented as a `Guid`. The identifiers are clustered-index friendly, being ordered in a way that SQL Server can efficiently insert them into a database with the *uniqueidentifier* as the primary key. Just use `NewId.NextGuid()` to generate an identifier – it's fast, fun, and all your friends are doing it.

Note: So, what does correlated actually mean? In short it means that this message is a part of a larger conversation. For instance, you may have a message that says `New Order (Item:Hammers; Qty:22; OrderNumber:45)` and there may be another message that is a response to that message that says `Order Allocated(OrderNumber:45)`. In this case, the order number is acting as your correlation identifier, it ties the messages together.

Configuring MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

To configure MassTransit in your application, it depends upon the application type. There are several application types, which are covered below. In these examples, the use of a dependency injection framework is not included. Using a container (such as Autofac, StructureMap, etc.) with MassTransit is covered separately.

Configuring MassTransit in a console application

A console application has a `Main` entry point, which is part of the `Program.cs` class by default. The example below configures a simple bus instance that publishes an event with a value entered.

```
1 namespace EventPublisher
2 {
3     public interface ValueEntered
4     {
5         string Value { get; }
6     }
```

```
7
8       public class Program
9       {
10          public void static Main()
11          {
12              var busControl = ConfigureBus();
13              busControl.Start();
14
15              do
16              {
17                  Console.WriteLine("Enter message (or quit to exit)");
18                  Console.Write("> ");
19                  string value = Console.ReadLine();
20
21                  if("quit".Equals(value, StringComparison.OrdinalIgnoreCase))
22                      break;
23
24                  busControl.Publish<ValueEntered>(new
25                  {
26                      Value = value
27                  });
28              }
29              while (true);
30
31              busControl.Stop();
32          }
33
34          static IBusControl ConfigureBus()
35          {
36              return Bus.Factory.CreateUsingRabbitMq(cfg =>
37              {
38                  cfg.Host(new Uri("rabbitmq://localhost"), h =>
39                  {
40                      h.Username("guest");
41                      h.Password("guest");
42                  });
43              });
44          }
45      }
46  }
```

In the example, the bus is configured and started, after which a publishing loop allows values to be entered and published. When the loop exits, the `busControl` variable is disposed, which stops the bus.

Configuring MassTransit in a Windows service

A Windows service is recommended for consuming commands and events as it provides an autonomous execution environment for message consumers. The service can be started and stopped using the service control manager, as well as monitored by operations tools.

Note: To create a Windows service, we strongly recommend using Topshelf, as it was built specifically for this purpose. Topshelf is easy to use, has zero dependencies, and creates a service that can be self-installed without additional tools.

The important aspect of configuring a bus in a Windows service is to ensure that the bus is only configured and started

when the service is *started*.

```
namespace EventService
{
    using Topshelf;

    public class Program
    {
        public int static Main()
        {
            return HostFactory.Run(cfg => cfg.Service(x => new␣
↪EventConsumerService());
        }
    }

    class EventConsumerService :
        ServiceControl
    {
        IBusControl _bus;

        public bool Start(HostControl hostControl)
        {
            _bus = ConfigureBus();
            _bus.Start();

            return true;
        }

        public bool Stop(HostControl hostControl)
        {
            _bus?.Stop(TimeSpan.FromSeconds(30));

            return true;
        }

        IBusControl ConfigureBus()
        {
            return Bus.Factory.CreateUsingRabbitMq(cfg =>
            {
                var host = cfg.Host(new Uri("rabbitmq://localhost"), h =>
                {
                    h.Username("guest");
                    h.Password("guest");
                });

                cfg.ReceiveEndpoint(host, "event_queue", e =>
                {
                    e.Handler<ValueEntered>(context =>
                        Console.Out.WriteLineAsync($"Value was entered: {context.
↪Message.Value}"));
                })
            });
        }
    }
}
```

Configuring MassTransit in a web application

Configuring a bus in a web site is typically done to publish events, send commands, as well as engage in request/response conversations. Hosting receive endpoints and persistent consumers is not recommended (use a service as shown above).

In a web application, the `HttpApplication` class methods of Application_Start and Application_End are used to configure/start the bus and stop the bus respectively.

Note: While many MassTransit samples use Topshelf, web applications are an exception where the standard web application conventions are followed.

```
public class MvcApplication :
    HttpApplication
{
    static IBusControl _busControl;

    public static IBus Bus
    {
        get { return _busControl; }
    }

    protected void Application_Start()
    {
        _busControl = ConfigureBus();
        _busControl.Start();
    }

    protected void Application_End()
    {
        _busControl.Stop(TimeSpan.FromSeconds(10));;
    }

    IBusControl ConfigureBus()
    {
        return Bus.Factory.CreateUsingRabbitMq(cfg =>
        {
            var host = cfg.Host(new Uri("rabbitmq://localhost"), h =>
            {
                h.Username("guest");
                h.Password("guest");
            });
        });
    }
}

public class NotifyController :
    Controller
{
    public async Task<ActionResult> Put(string value)
    {
        await MvcApplication.Bus.Publish<ValueNotified>(new
        {
            Value = value
        });

        return View();
```

```
    }
}

public class CommandController :
    Controller
{
    public async Task<ActionResult> Send(string value)
    {
        var endpoint = await MvcApplication.Bus.GetSendEndpoint(_serviceAddress);

        await endpoint.Send<SubmitValue>(new
        {
            Timestamp = DateTime.UtcNow,
            Value = value
        });

        return View();
    }
}
```

The above example is kept simple, providing a static `MvcApplication.Bus` property to access the bus instance (for publishing events, and sending commands to endpoints). Newer version of ASP.NET have built-in dependency resolution, in which case the `IBus` should be registered so that controllers can specify the dependency in the constructor. In fact, the inherited `IPublishEndpoint` and `ISendEndpointProvider` should also be registered.

The example controllers show how to publish and send messages as well.

Configuring a container

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

When MassTransit was originally built, it required a dependency injection container to work. Unfortunately, in 2008 there were few .NET developers who knew what Inversion of Control (IOC) or Dependency Injection (DI) was, and even fewer who used it. Several versions later, the requirement was dropped and container support was put back where it belonged - around the edges of the application instead of at the core of the framework.

Fortunately, the world has changed and DI is more mainstream, particularly with newer applications. And container support in MassTransit has stayed up to date, including all of the major containers.

Note: Dependency Injection styles are a personal choice that each developer or organization must make on their own. We recognize this choice, and respect it, and will not judge those who don't use a particular container or style of dependency injection. In short, we care.

Hey! Where is my container??

Containers come and go, so if you don't see your container here, or feel that the support for you container is weaksauce, pull requests are always welcome. Using an existing container it should be straight forward to add support for your favorite ÜberContainer.

Configuring Autofac

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Autofac is a powerful and fast container, and is well supported by MassTransit. Nested lifetime scopes are used extensively to encapsulate dependencies and ensure clean object lifetime management. The following examples show the various ways that MassTransit can be configured, including the appropriate interfaces necessary.

Note: Requires NuGets `MassTransit`, `MassTransit.AutoFac`, and `MassTransit.RabbitMQ`

Note: Consumers should not typically depend upon `IBus` or `IBusControl`. A consumer should use the `ConsumeContext` instead, which has all of the same methods as `IBus`, but is scoped to the receive endpoint. This ensures that messages can be tracked between consumers, and are sent from the proper address.

```
using System;
using System.Reflection;
using System.Threading.Tasks;
using Autofac;
using MassTransit;

namespace Example
{
    public class UpdateCustomerAddressConsumer : MassTransit.IConsumer<object>
    {
        public async Task Consume(ConsumeContext<object> context)
        {
            //do stuff
        }
    }

    class Program
    {

        public static void Main(string[] args)
        {
            var builder = new ContainerBuilder();

            // register a specific consumer
            builder.RegisterType<<UpdateCustomerAddressConsumer>();

            // just register all the consumers
            builder.RegisterConsumers(Assembly.GetExecutingAssembly());

            builder.Register(context =>
            {
                var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
                {
                    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
                    {
```

```
                        h.Username("guest");
                        h.Password("guest");
                    });

                    cfg.ReceiveEndpoint("customer_update_queue", ec =>
                    {
                        ec.LoadFrom(context);
                    });
                });

                return busControl;
            })
                .SingleInstance()
                .As<IBusControl>()
                .As<IBus>();

            var container = builder.Build();

            var bc = container.Resolve<IBusControl>();
            bc.Start();
        }
    }
}
```

Using a Module

Autofac modules are great for encapsulating configuration, and that is equally true when using MassTransit. An example of using modules with Autofac is shown below.

```
class ConsumerModule :
    Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<UpdateCustomerAddressConsumer>();

        builder.RegisterType<SqlCustomerRegistry>()
            .As<ICustomerRegistry>();
    }
}

class BusModule :
    Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(context =>
        {
            var busSettings = context.Resolve<BusSettings>();

            var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
            {
                var host = cfg.Host(busSettings.HostAddress, h =>
                {
                    h.Username(busSettings.Username);
                    h.Password(busSettings.Password);
```

```
                });

                sbc.ReceiveEndpoint(busSettings.QueueName, ec =>
                {
                    ec.LoadFrom(context);
                })
            });
        })
            .SingleInstance()
            .As<IBusControl>()
            .As<IBus>();
    }
}

public IContainer CreateContainer()
{
     var builder = new ContainerBuilder();

    builder.RegisterModule<BusModule>();
    builder.RegisterModule<ConsumerModule>();

    return builder.Build();
}

public void CreateContainer()
{
    _container = new Container(x =>
    {
        x.AddRegistry(new BusRegistry());
        x.AddRegistry(new ConsumerRegistry());
    });
}
```

Registering State Machine Sagas

By using an additional package *MassTransit.Automatonymous.Autofac* you can also register state machine sagas

```
var builder = new ContainerBuilder();
// register everything else

// register saga state machines, assuming Saga1 and Saga2 are in different assemblies
builder.RegisterStateMachineSagas(typeof(Saga1).Assembly, typeof(Saga2).Assembly);

// registering saga state machines from current assembly
builder.RegisterStateMachineSagas(Assembly.GetExecutingAssembly());

// do not forget registering saga repositories (example for NHibernate)
var mappings = mappingsAssembly
    .GetTypes()
    .Where(t => t.BaseType != null && t.BaseType.IsGenericType &&
        (t.BaseType.GetGenericTypeDefinition() == typeof(SagaClassMapping<>) ||
        t.BaseType.GetGenericTypeDefinition() == typeof(ClassMapping<>)))
    .ToArray();
builder.Register(c => new SqlServerSessionFactoryProvider(connString, mappings).
↪GetSessionFactory())
    .As<ISessionFactory>()
    .SingleInstance();
```

```
builder.RegisterGeneric(typeof(NHibernateSagaRepository<>))
    .As(typeof(ISagaRepository<>));
```

and load them from a contained when configuring the bus.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(busSettings.HostAddress, h =>
    {
        h.Username(busSettings.Username);
        h.Password(busSettings.Password);
    });

    sbc.ReceiveEndpoint(busSettings.QueueName, ec =>
    {
        // loading consumers
        ec.LoadFrom(context);

        // loading saga state machines
        ec.LoadStateMachineSagas(context);
    })
});
```

Configuring Ninject

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The following example shows how to configure a simple Ninject container, and include the bus in the container. The two bus interfaces, `IBus` and `IBusControl`, are included.

Note: Consumers should not typically depend upon `IBus` or `IBusControl`. A consumer should use the `ConsumeContext` instead, which has all of the same methods as `IBus`, but is scoped to the receive endpoint. This ensures that messages can be tracked between consumers, and are sent from the proper address.

```
public static void main(string[] args)
{
    var kernel = new StandardKernel();

    // register a specific consumer
    kernel.Bind<UpdateCustomerAddressConsumer>().ToSelf();

    // just register all the consumers using Ninject.Extensions.Conventions
    kernel.Bind(x =>
    {
        x.FromThisAssembly()
            .SelectAllClasses()
            .InheritedFrom<IConsumer>()
            .BindToSelf();
    });
```

```
    var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
    {
        var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
        {
            h.Username("guest");
            h.Password("guest");
        });

        sbc.ReceiveEndpoint("customer_update_queue", ec =>
        {
            ec.LoadFrom(kernel);
        })
    });

    kernel.Bind<IBus>()
        .ToProvider(new CallbackProvider<IBus>(x => x.Kernel.Get<IBusControl>()));

    busControl.Start();
}
```

Note: The behavior with Ninject is slightly different, in that the current AppDomain types are checked against the container and if any consumer types are registered, they are resolved from the container. The unit tests pass, and it works, but just be aware that container metadata is not being used to support this feature. There is some history on this, found at the Ninject issue.

Configuring StructureMap

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The following example shows how to configure a simple StructureMap container, and include the bus in the container. The two bus interfaces, `IBus` and `IBusControl`, are included.

Note: Consumers should not typically depend upon `IBus` or `IBusControl`. A consumer should use the `ConsumeContext` instead, which has all of the same methods as `IBus`, but is scoped to the receive endpoint. This ensures that messages can be tracked between consumers, and are sent from the proper address.

```
public static void main(string[] args)
{
    var container = new Container(cfg =>
    {
        // register each consumer
        cfg.ForConcreteType<UpdateCustomerAddressConsumer>();

        //or use StructureMap's excellent scanning capabilities
    });
```

```
    var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
    {
        var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
        {
            h.Username("guest");
            h.Password("guest");
        });

        sbc.ReceiveEndpoint("customer_update_queue", ec =>
        {
            ec.LoadFrom(container);
        })
    });

    container.Configure(cfg =>
    {
        For<IBusControl>()
            .Use(busControl);
        Forward<IBusControl, IBus>();
    });

    busControl.Start();
}
```

Using a Registry

In larger applications, the use of registries is common. In fact, this is one of the better ways to use MassTransit with StructureMap, as it ensures consistent usage across services.

```
class BusRegistry :
    Registry
{
    public BusRegistry()
    {
        For<IBusControl>(new SingletonLifecycle())
            .Use(context => Bus.Factory.CreateUsingInMemory(x =>
            {
                x.ReceiveEndpoint("customer_update_queue", e => e.LoadFrom(context));
            }));
        Forward<IBusControl, IBus>();
    }
}

class ConsumerRegistry :
    Registry
{
    public ConsumerRegistry()
    {
        ForConcreteType<UpdateCustomerAddressConsumer>();

        For<ICustomerRegistry>()
            .Use<SqlCustomerRegistry>();
    }
}

public void CreateContainer()
```

```
{
    _container = new Container(x =>
    {
        x.AddRegistry(new BusRegistry());
        x.AddRegistry(new ConsumerRegistry());
    });
}
```

Configuring Unity

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The following example shows how to configure a simple Unity container, and include the bus in the container. The two bus interfaces, `IBus` and `IBusControl`, are included.

Note: Consumers should not typically depend upon `IBus` or `IBusControl`. A consumer should use the `ConsumeContext` instead, which has all of the same methods as `IBus`, but is scoped to the receive endpoint. This ensures that messages can be tracked between consumers, and are sent from the proper address.

```
public static void main(string[] args)
{
    var container = new UnityContainer();

    // register each consumer
    container.RegisterType<UpdateCustomerAddressConsumer>(new␣
↪ContainerControlledLifetimeManager());

    // scan for types
    var consumerTypes = new TypeFinder().FindTypesWhichImplement(typeof(IConsumer));
    foreach (var consumerType in consumerTypes)
    {
        container.RegisterType(consumerType, new␣
↪ContainerControlledLifetimeManager());
    }

    var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
    {
        var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
        {
            h.Username("guest");
            h.Password("guest");
        });

        sbc.ReceiveEndpoint("customer_update_queue", ec =>
        {
            ec.LoadFrom(container);
        })
    });
```

```
    container.RegisterInstance<IBusControl>(busControl);
    container.RegisterInstance<IBus>(busControl);

    busControl.Start();
}
```

Configuring Castle Windsor

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The following example shows how to configure a simple Castle Windsor container, and include the bus in the container. The two bus interfaces, `IBus` and `IBusControl`, are included.

Note: Consumers should not typically depend upon `IBus` or `IBusControl`. A consumer should use the `ConsumeContext` instead, which has all of the same methods as `IBus`, but is scoped to the receive endpoint. This ensures that messages can be tracked between consumers, and are sent from the proper address.

```
public static void main(string[] args)
{
    var container = new WindsorContainer();

    // register each consumer manually
    container.Register(Component.For<YourConsumer>().LifestyleTransient());

    //or use Windsor's excellent scanning capabilities
    container.Register(AllTypes.FromThisAssembly().BasedOn<IConsumer>());

    var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
    {
        var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
        {
            h.Username("guest");
            h.Password("guest");
        });

        sbc.ReceiveEndpoint("customer_update_queue", ec =>
        {
            ec.EnableMessageScope();
            ec.LoadFrom(container);
        })
    });

    container.Register(Component.For<IBus>().Instance(busControl));
    container.Register(Component.For<IBusControl>().Instance(busControl));

    busControl.Start();
}
```

Using a Windsor Installer

It is sometimes useful to put the initialization into an installer, so that it can use existing components from the container for configuration.

```
public class MassTransitInstaller :
    IWindsorInstaller
{
    public void Install(IWindsorContainer container, Castle.MicroKernel.SubSystems.
↪Configuration.IConfigurationStore store)
    {
        var busConfig = container.Resolve<BusConfiguration>();

        var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
        {
            var host = cfg.Host(busConfig.HostAddress, h =>
            {
                h.Username(busConfig.Username);
                h.Password(busConfig.Password);
            });

            sbc.ReceiveEndpoint(busConfig.QueueName, ec =>
            {
                ec.EnableMessageScope();
                ec.LoadFrom(container);
            })
        });

        container.Release(busConfig);

        container.Register(Component.For<IBus>().Instance(busControl));
        container.Register(Component.For<IBusControl>().Instance(busControl));

        busControl.Start();
    }
}
```

Crafting a request/response conversation

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Request/response is a common pattern in application development, where a component sends a request to a service and continues once the response is received. In a distributed system, this can increase the latency of an application since the service many be hosted in another process, on another machine, or may even be a remote service in another network. While in many cases it is best to avoid request/response use in distributed applications, particularly when the request is a command, it is often necessary and preferred over more complex solutions.

Fortunately for .NET developers, C# 5.0 introduced the async/await keywords, making it easier to program applications that call services asynchronously. By using *Tasks* and the *async* and *await* keywords, developers can write procedural code and avoid the complex use of callbacks and handlers. Additionally, multiple asynchronous requests can be

executed at once, reducing the overall execution time to that of the longest request.

Creating the message contracts

To get started, the message contracts need to be created. In this example, an order status check is being created.

```
public interface CheckOrderStatus
{
    string OrderId { get; }
}

public interface OrderStatusResult
{
    string OrderId { get; }
    DateTime Timestamp { get; }
    short StatusCode { get; }
    string StatusText { get; }
}
```

Creating the message request client

Most interactions of the request/response nature consist of four elements: the request arguments, the response values, exception handling, and the time to wait for a response. The .NET framework gives us one additional element, a `CancellationToken` which can prematurely cancel waiting for the response. The request client optimizes these elements into an easy-to-use interface:

```
public interface IRequestClient<TRequest, TResponse>
{
    Task<TResponse> Request(TRequest request, CancellationToken cancellationToken);
}
```

The interface is simple, a single method that accepts the request and returns a Task that can be awaited. The interface declares the request and response types, making it useful for dependency management using dependency injection. In fact, by using the request client, an application can be completely free of any MassTransit concerns such as message contexts and endpoints. The configuration of the application can defined the endpoints and connections and register them in a dependency injection container, keeping the configuration complexity at the outer edge of the application.

To create a request client, the provided `MessageRequestClient` can be used.

```
Uri address = new Uri("loopback://localhost/order_status_check");
TimeSpan requestTimeout = TimeSpan.FromSeconds(30);

IRequestClient<CheckOrderStatus, OrderStatusResult> client =
    new MessageRequestClient<CheckOrderStatus, OrderStatusResult>(bus, address,␣
↪requestTimeout);
```

Once created, the request client instance can be registered with the dependency resolver using the `IRequestClient` interface type. Once registered, a controller can use the client via a constructor dependency.

```
public class RequestController :
    Controller
{
    IRequestClient<CheckOrderStatus, OrderStatusResult> _client;

    public RequestController(IRequestClient<CheckOrderStatus, OrderStatusResult>␣
↪client)
```

```
    {
        _client = client;
    }

    public async Task<ActionResult> Get(string id)
    {
        var command = new CheckOrderStatus
        {
            OrderId = id
        };

        var result = await _client.Request(command);

        return View(result);
    }
}
```

The controller method will send the command, and return the view once the result has been received. The syntax is significantly cleaner than dealing with message object, consumer contexts, responses, etc. And since async/await and messaging are both about asynchronous programming, it's a natural fit.

Composing multiple requests

If there were multiple requests to be performed, it is easy to wait on all results at the same time, benefiting from the concurrent operation.

```
public class RequestController :
    Controller
{
    IRequestClient<RequestA, ResultA> _clientA;
    IRequestClient<RequestB, ResultB> _clientB;

    public RequestController(IRequestClient<RequestA, ResultA> clientA, IRequestClient
↪<RequestB, ResultB> clientB)
    {
        _clientA = clientA;
        _clientB = clientB;
    }

    public async Task<ActionResult> Get()
    {
        var requestA = new RequestA();
        Task<ResultA> resultA = _clientA.Request(requestA);

        var requestB = new RequestB();
        Task<ResultB> resultB = _clientB.Request(requestB);

        await Task.WhenAll(resultA, resultB);

        var model = new Model(resultA.Result, resultB.Result);

        return View(model);
    }
}
```

The power of concurrency, for the win!

Handling exceptions

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Let's face it, bad things happen. Networks partition, data servers crash, remote endpoints get busy and fail to respond. And when bad things happen, exceptions get thrown. And when exceptions get thrown, people die. Okay, maybe that's a bit dramatic, but the point is, exceptions are a fact of software development.

Fortunately, MassTransit provides a number of features to help your application recover from and deal with exceptions. But before getting into that, an understanding of what happens when a message is consumed is needed.

Take, for example, a consumer that simply throws an exception.

```
public class UpdateCustomerAddressConsumer :
    IConsumer<UpdateCustomerAddress>
{
    public Task Consume(ConsumeContext<UpdateCustomerAddress> context)
    {
        throw new Exception("Very bad things happened");
    }
}
```

When a message is delivered to the consumer, the consumer throws an exception. With a default bus configuration, the exception is caught by middleware in the transport (the `MoveExceptionToTransportFilter` to be exact), and the message is moved to an *_error* queue (prefixed by the receive endpoint queue name). The exception details are stored as headers with the message, for analysis and to assist in troubleshooting the exception.

In addition to moving the message to an error queue, MassTransit also generates a `Fault<T>` event. If the received message specified a `FaultAddress` header, the fault is sent to that address. If a fault address is not found, and a `ResponseAddress` is present, the fault is sent to the response address. If neither address is present, the fault is published.

Retrying messages

In some cases, the exception may be a transient condition, such as a database deadlock, a busy web service, or some similar type of situation which usually clears up on a second attempt. With these exception types, it is often desirable to retry the message delivery to the consumer, allowing the consumer to try the operation again.

```
public class UpdateCustomerAddressConsumer :
    IConsumer<UpdateCustomerAddress>
{
    ISessionFactory _sessionFactory;

    public async Task Consume(ConsumeContext<UpdateCustomerAddress> context)
    {
        using(var session = _sessionFactory.OpenSession())
        using(var transaction = session.BeginTransaction())
        {
            var customer = session.Get<Customer>(context.Message.CustomerId);
            // update customer address properties from message
```

```
            session.Update(customer);

            transaction.Commit();
        }
    }
}
```

With this consumer, an `ADOException` can be thrown if the update fails. In this case, the operation should be retried before moving the message to the error queue. This can be configured on the receive endpoint. Shown below is a retry policy which attempts to deliver the message to a consumer five times before throwing the exception back up the pipeline.

```
ISessionFactory sessionFactory = CreateSessionFactory();

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.UseRetry(Retry.Immediate(5));
        e.Consumer(() => new UpdateCustomerAddressConsumer(sessionFactory));
    });
});
```

The `UseRetry` method is an extension method that configures a middleware filter, in this case the `RetryFilter`. There are a variety of retry policies available, which are detailed in the reference section.

Note: In this example, the `UseRetry` is at the receive endpoint level. Additional retry filters can be added at the bus and consumer level, providing flexibility in how different consumers, messages, etc. are retried.

Consuming Faults

After all of the various retry policies have executed, the bus will generate a fault which you can consume. Below is a simple example of consuming a fault thrown by the consumer above.

```
public class UpdateCustomerAddressFaultConsumer :
    IConsumer<Fault<UpdateCustomerAddress>>
{

    public async Task Consume(ConsumeContext<Fault<UpdateCustomerAddress>> context)
    {
        var originalMessage = context.Message.Message;
        var exceptions = context.Message.Exceptions;

        //Do something interesting.
    }
}
```

Observing messages

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit supports a number of message observes, making it possible to monitoring when messages are received, consumed, sent, and published. Each type of observer is configured separately, keeping the interfaces lean and focused.

Warning: Observers should not be used to modify or intercept messages. To intercept messages (either to add headers, or modify the message contents), create or use an existing middleware component.

Observing received messages

To observe received messages immediately after they are delivered by the transport, create a class that implements the `IReceiveObserver` interface, and connect it to the bus as shown below.

```
public class ReceiveObserver :
    IReceiveObserver
{
    public Task PreReceive(ReceiveContext context)
    {
        // called immediately after the message was delivery by the transport
    }

    public Task PostReceive(ReceiveContext context)
    {
        // called after the message has been received and processed
    }

    public Task PostConsume<T>(ConsumeContext<T> context, TimeSpan duration, string␣
↪consumerType)
        where T : class
    {
        // called when the message was consumed, once for each consumer
    }

    public Task ConsumeFault<T>(ConsumeContext<T> context, TimeSpan elapsed, string␣
↪consumerType, Exception exception) where T : class
    {
        // called when the message is consumed but the consumer throws an exception
    }

    public Task ReceiveFault(ReceiveContext context, Exception exception)
    {
        // called when an exception occurs early in the message processing, such as␣
↪deserialization, etc.
    }
}
```

Then connect the observer to the bus before starting it, as shown.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Consumer<UpdateCustomerConsumer>();
    });
});

var observer = new ReceiveObserver();
var handle = busControl.ConnectReceiveObserver(observer);
```

Observing consumed messages

If the receive context isn't super interesting, perhaps the actual consumption of messages might float your boat. A consume observer implements the `IConsumeObserver` interface, as shown below.

```
public class ConsumeObserver :
    IConsumeObserver
{
    Task IConsumeObserver.PreConsume<T>(ConsumeContext<T> context)
    {
        // called before the consumer's Consume method is called
    }

    Task IConsumeObserver.PostConsume<T>(ConsumeContext<T> context)
    {
        // called after the consumer's Consume method is called
        // if an exception was thrown, the ConsumeFault method is called instead
    }

    Task IConsumeObserver.ConsumeFault<T>(ConsumeContext<T> context, Exception␣
↪exception)
    {
        // called if the consumer's Consume method throws an exception
    }
}
```

To connect the observer, use the `ConnectConsumeObserver` method before starting the bus.

Observing specific consumed messages

Okay, so it's obvious that if you've read this far you want a more specific observer, one that only is called when a specific message type is consumed. We have you covered there too, as shown below.

```
public class ConsumeObserver<T> :
    IConsumeMessageObserver<T>
    where T : class
{
    Task IConsumeMessageObserver<T>.PreConsume(ConsumeContext<T> context)
```

```
    {
        // called before the consumer's Consume method is called
    }

    Task IConsumeMessageObserver<T>.PostConsume(ConsumeContext<T> context)
    {
        // called after the consumer's Consume method was called
        // again, exceptions call the Fault method.
    }

    Task IConsumeMessageObserver<T>.ConsumeFault(ConsumeContext<T> context, Exception␣
↪exception)
    {
        // called when a consumer throws an exception consuming the message
    }
}
```

To connect the observer, use the `ConnectConsumeMessageObserver` method before starting the bus.

Observing sent messages

Okay, so, incoming messages are not your thing. We get it, you're all about what goes out. It's cool. It's better to send than to receive. Or is that give? Anyway, a send observer is also available.

```
public class SendObserver :
    ISendObserver
{
    public Task PreSend<T>(SendContext<T> context)
        where T : class
    {
        // called just before a message is sent, all the headers should be setup and␣
↪everything
    }

    public Task PostSend<T>(SendContext<T> context)
        where T : class
    {
        // called just after a message it sent to the transport and acknowledged␣
↪(RabbitMQ)
    }

    public Task SendFault<T>(SendContext<T> context, Exception exception)
        where T : class
    {
        // called if an exception occurred sending the message
    }
}
```

To connect the observer, you already guessed it, use the `ConnectSendObserver` method before starting the bus.

Observing published messages

In addition to send, publish is also observable. Because the semantics matter, absolutely. Published messages are also sent, so if you're observing both outbound message flows, you will get duplicates. Using the MessageId to link them up as it's unique for each message.

```
public class PublishObserver :
    IPublishObserver
{
    public Task PrePublish<T>(PublishContext<T> context)
        where T : class
    {
        // called right before the message is published (sent to exchange or topic)
    }

    public Task PostPublish<T>(PublishContext<T> context)
        where T : class
    {
        // called after the message is published (and acked by the broker if RabbitMQ)
    }

    public Task PublishFault<T>(PublishContext<T> context, Exception exception)
        where T : class
    {
        // called if there was an exception publishing the message
    }
}
```

Finally, to connect the observer, use the `ConnectPublishObserver` method before starting the bus.

These are a ton of interfaces, and they offer a lot of information about how the system is behaving under the hood. So use them, abuse them, bend them, and break them. Just realize, they are immediate, so don't be slow or your messaging will be equally slow.

Observing lifecycle events

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

When integrating a framework into your application, it can be useful to understand when the framework is "doing stuff." Whether it is starting up, shutting down, or anything in between, being notified and thereby able to take action is a huge benefit.

MassTransit supports a number of lifecycle events that can be observed, making it easy to build components that are started and stopped along with the bus.

Note: A good example of this is the `UseInMemoryMessageScheduler`, which is part of the Quartz.NET integration package. Using the lifecycle events, Quartz is able to be started and stopped on a receive endpoint without any additional development. And that saves you time.

To observe bus events, create a class which implements `IBusObserver`, as shown below.

```
public class BusObserver :
    IBusObserver
{
    public Task PostCreate(IBus bus)
```

```
    {
        // called after the bus has been created, but before it has been started.
    }

    public Task CreateFaulted(Exception exception)
    {
        // called if the bus creation fails for some reason
    }

    public Task PreStart(IBus bus)
    {
        // called just before the bus is started
    }

    public Task PostStart(IBus bus, Task busReady)
    {
        // called once the bus has been started successfully. The task can be used to␣
↪wait for
        // all of the receive endpoints to be ready.
    }

    public Task StartFaulted(IBus bus, Exception exception)
    {
        // called if the bus fails to start for some reason (dead battery, no fuel,␣
↪etc.)
    }

    public Task PreStop(IBus bus)
    {
        // called just before the bus is stopped
    }

    public Task PostStop(IBus bus)
    {
        // called after the bus has been stopped
    }

    public Task StopFaulted(IBus bus, Exception exception)
    {
        // called if the bus fails to stop (no brakes)
    }
}
```

Bus observers can only be configured during bus configuration. To connect a bus observer during bus configuration, refer to the example shown below.

```
var busObserver = new BusObserver();

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
```

```
        e.Consumer<UpdateCustomerConsumer>();
    });

    cfg.BusObserver(busObserver);
});
```

Connecting endpoints

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Once the bus is started, additional endpoints may be connected. An additional endpoint may be needed for a temporary process, or to allow an application to receive responses on a dedicated queue versus using the existing bus receive endpoint.

Connecting an endpoint

To connect a new receive endpoint to a RabbitMQ host, the host is used.

```
IRabbitMqHost host = ...; // the host configured on the bus

var handle = await host.ConnectReceiveEndpoint("secondary_queue", x =>
{
    x.Consumer<MyConsumer>(...);
    x.Handler<MyMessage>(...);
});
```

Consumers registered on the receive endpoint are configured the same as receive endpoints configured during bus creation, and will have their messages types subscribed to the receive endpoint's queue. And since receive endpoint is configured using the host, all of the transport features available on the host are available for the receive endpoint (such as setting a prefetch count, or specifying message queue properties).

To connect a new receive endpoint to an Azure Service Bus host, the same syntax is used but with the `IServiceBusHost` reference.

Disconnecting an endpoint

When an endpoint has been connected to an existing bus, it should be shut down before shutting down the bus. The handle returned by the `ConnectReceiveEndpoint` call should be used to `Stop` the receive endpoint.

```
await handle.StopAsync();
```

Retry policies

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

When you register various consumers of messages, one of the configuration elements you have control over is the retry policy.

```
Bus.Factory.CreateUsingInMemory(cfg =>
{
    cfg.ReceiveEndpoint("queue_name", ep =>
    {
        ep.Handler(async cxt => {});
        ep.Handler(async cxt => {}, endpointConfig =>
        {
          endpointConfig.Retry(Retry.None);
        });
    });
});
```

Retry options

None

Immediate

`Retry.Immediate(5)`

Would retry 5 times with no delay.

Intervals

`Retry.Intervals(5.Seconds(), 5.Seconds())`

Would retry 2 times at 0:05, 0:10.

Exponential

Takes a number of retries and attempts to do so for each interval.

`Retry.Exponential(5, 5.Seconds(), 5.Seconds(), 5.Seconds())`

Would retry 5 times at 0:05, 0:10, 0:15, 0:20, 0:25. ??

Incremental

Takes a number of retries and attempts to do so for each interval.

`Retry.Incremental(5, 5.Seconds(), 5.Seconds())`

Would retry 5 times at 0:05, 0:10, 0:15, 0:20, 0:25.

Except

Selected

All

Filter

Managing transactions

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

Note: This is an advanced concept, and scenarios vary, so consider this guidance, not gospel.

The message pipeline in MassTransit is asynchronous, leveraging the Task Parallel Library (TPL) extensively to maximum thread utilization. This means that receiving an individual message may involve several threads over the lifecycle of the consumer. To prevent strange things from happening, developers should avoid using any *static* or *thread static* variables as these are one of the main causes of errors in asynchronous programming.

The .NET `System.Transactions` namespace is a static hound, with many applications following the model of using a transaction scope to wrap a transactional operation.

```
public class Repository
{
    public void Save(Entity entity)
    {
        using(var scope = new TransactionScope())
        {
            SaveEntity(entity);

            scope.Complete();
        }
    }
}
```

In this example, the creation of a `TransactionScope` actually sets a static variable, `Transaction.Current`, to the created or ambient transaction. That word *ambient* should be a big clue – it's using a static variable (in this case, it's actually a thread static, but anyway).

It turns out that the above example is simple, and works, because there are no asynchronous methods. But that also means that the method blocks the thread while the database performs work (which takes an eternity in CPU time). Most databases support asynchronous operations (including Entity Framework), so it is logical to assume that using those methods would increase thread utilization.

It is also often requested that a set of operations be managed as a *unit of work*. A single transaction is shared across multiple operations that are committed as a single unit. If the commit fails, everything is undone and the message is faulted (or, retried based on the retry policy).

Sharing a transaction

MassTransit includes transaction middleware to share a single committable transaction across any number consumers and any dependencies used by the those consumers. To use the middleware, it must be added to the bus or receive endpoint.

```
Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "event_queue", e =>
    {
        e.UseTransaction(x =>
        {
            Timeout = TimeSpan.FromSeconds(90);
            IsolationLevel = IsolationLevel.ReadCommitted;
        });

        e.Consumer<UpdateCustomerAddressConsumer>();
    })
});
```

For each message, a new `CommittableTransaction` is created. This transaction can be passed to classes that support transactional operations, such as `DbContext`, `SqlCommand`, and `SqlConnection`. It can also be used to create any `TransactionScope` that may be required to support a synchronous operation.

To use the transaction directly in a consumer, the transaction can be pulled from the `ConsumeContext`.

```
public class TransactionalConsumer :
    IConsumer<UpdateCustomerAddress>
{
    readonly SqlConnection _connection; // ctor injected

    public async Task Consume(ConsumeContext<UpdateCustomerAddress> context)
    {
        var transactionContext = context.GetPayload<TransactionContext>();

        _connection.EnlistTransaction(transactionContext.Transaction);

        using (SqlCommand command = new SqlCommand(sql, _connection))
        {
            using (var reader = await command.ExecuteReaderAsync())
            {
            }
        }

        // the connection lifetime should be managed by a container
        // or perhaps another more specific middleware component.
    }
}
```

The connection (and by use of the connection, the command) are enlisted in the transaction. Once the method completes, and control is returned to the transaction middleware, if no exceptions are thrown the transaction is committed (which should complete the database operation). If an exception is thrown, the transaction is rolled back.

While not shown here, a class that provides the connection, and enlists the connection upon creation, should be added to the container to ensure that the transaction is not enlisted twice (not sure that's a bad thing though, it should be ignored). Also, as long as only a single connection string is enlisted, the DTC should not get involved. Using the same transaction across multiple connection strings is a bad thing, as it will make the DTC come into play which slows the world down significantly.

Displaying configuration

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

A bus instance is composed of many classes, all of which are wired together to form a connection pipeline of message processing goodness. This brings a bit of complexity, as there are many moving parts behind the curtain. To help troubleshoot and understand how a bus is configured, it is possible to probe the bus and return an object graph of the bus.

To probe bus configuration, use the `GetProbeResult` method as shown below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/test"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    sbc.ReceiveEndpoint("input_queue", ec =>
    {
        ec.Consumer<UpdateCustomerAddressConsumer>();
    })
});

ProbeResult result = busControl.GetProbeResult();

Console.WriteLine(result.ToJsonString());
```

The resulting output for the configuration above would be similar to the following.

```
{
  "resultId": "7f280000-2961-000c-1dcc-08d2c68a08ac",
  "probeId": "7f280000-2961-000c-fd27-08d2c68a08ab",
  "startTimestamp": "2015-09-26T15:49:16.594521Z",
  "duration": "00:00:00.4850036",
  "host": {
    "machineName": "LOCALHOST",
    "processName": "TestService",
    "processId": 5808,
    "assembly": "MassTransit",
    "assemblyVersion": "3.0.13.0",
    "frameworkVersion": "4.0.30319.42000",
    "massTransitVersion": "3.0.13.0",
    "operatingSystemVersion": "Microsoft Windows NT 6.3.9600.0"
```

```
  },
  "results": {
    "bus": {
      "address": "rabbitmq://[::1]:5672/test/bus-testservice-
↪xhwyyybjcryy3ofjbdjcpnoenx?durable=false&autodelete=true&prefetch=8",
      "host": {
        "type": "RabbitMQ",
        "host": "[::1]",
        "port": 5672,
        "virtualHost": "test",
        "username": "guest",
        "password": "*****",
        "connected": true
      },
      "receiveEndpoint": [
        {
          "transport": {
            "type": "RabbitMQ",
            "queueName": "input_queue",
            "exchangeName": "input_queue",
            "prefetchCount": 16,
            "durable": true,
            "queueArguments": {},
            "exchangeArguments": {},
            "purgeOnStartup": true,
            "exchangeType": "fanout",
            "bindings": [
              {
                "exchange": {
                  "exchangeName": "TestService.Contracts:UpdateCustomerAddress",
                  "exchangeType": "fanout",
                  "durable": true,
                  "arguments": {}
                },
                "routingKey": "",
                "arguments": {}
              }
            ]
          },
          "filters": [
            {
              "filterType": "deadLetter",
              "filters": {
                "filterType": "move",
                "destinationAddress": "rabbitmq://[::1]:5672/test/input_queue_skipped?
↪bind=true&queue=input_queue_skipped"
              }
            },
            {
              "filterType": "rescue",
              "filters": {
                "filterType": "moveFault",
                "destinationAddress": "rabbitmq://[::1]:5672/test/input_queue_error?
↪bind=true&queue=input_queue_error"
              }
            },
            {
              "filterType": "deserialize",
```

```
              "deserializers": {
                "json": {
                  "contentType": "application/vnd.masstransit+json"
                },
                "bson": {
                  "contentType": "application/vnd.masstransit+bson"
                },
                "xml": {
                  "contentType": "application/vnd.masstransit+xml"
                }
              },
              "pipe": {
                "TestService.Contracts.UpdateCustomerAddress": {
                  "filters": {
                    "filterType": "instance",
                    "type": "MassTransit.Testing.MultiTestConsumer+Of<TestService.
↪Contracts.UpdateCustomerAddress>"
                  }
                }
              }
            }
          ]
        },
        {
          "transport": {
            "type": "RabbitMQ",
            "queueName": "bus-testservice-xhwyyybjcryy3ofjbdjcpnoenx",
            "exchangeName": "bus-testservice-xhwyyybjcryy3ofjbdjcpnoenx",
            "prefetchCount": 8,
            "autoDelete": true,
            "queueArguments": {
              "x-expires": 60000
            },
            "exchangeArguments": {
              "x-expires": 60000
            },
            "exchangeType": "fanout",
            "bindings": []
          },
          "filters": [
            {
              "filterType": "deadLetter",
              "filters": {
                "filterType": "move",
                "destinationAddress": "rabbitmq://[::1]:5672/test/bus-testservice-
↪xhwyyybjcryy3ofjbdjcpnoenx_skipped?bind=true&queue=bus-testservice-
↪xhwyyybjcryy3ofjbdjcpnoenx_skipped"
              }
            },
            {
              "filterType": "rescue",
              "filters": {
                "filterType": "moveFault",
                "destinationAddress": "rabbitmq://[::1]:5672/test/bus-testservice-
↪xhwyyybjcryy3ofjbdjcpnoenx_error?bind=true&queue=bus-testservice-
↪xhwyyybjcryy3ofjbdjcpnoenx_error"
              }
            },
```

```
            {
              "filterType": "deserialize",
              "deserializers": {
                "json": {
                  "contentType": "application/vnd.masstransit+json"
                },
                "bson": {
                  "contentType": "application/vnd.masstransit+bson"
                },
                "xml": {
                  "contentType": "application/vnd.masstransit+xml"
                }
              },
              "pipe": {}
            }
          ]
        }
      ]
    }
  }
}
```

CHAPTER 4

Using Sagas

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The ability to ochestrate a series of events is a powerful feature, and MassTransit makes this possible.

A saga is a long-lived transaction managed by a coordinator. Sagas are initiated by an event, sagas orchestrate events, and sagas maintain the state of the overall transaction. Sagas are designed to manage the complexity of a distributed transaction without locking and immediate consistency. They manage state and track any compensations required if a partial failure occurs.

We didn't create it, we learned it from the original Cornell paper and from Arnon Rotem-Gal-Oz's description.

Creating Automatonymous State Machines

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/Automatonymous.

The latest version of this page can be found here.

Automatonymous is a state machine library built by the same team that created MassTransit. Automatonymous provides a friendly syntax for declaring a state machine, including the states, events (both trigger event and data events are supported), and behaviors. The simple syntax makes it easy to get started with your own state machines, while including many advanced features that make it extremely flexible in a variety of business contexts.

Like MassTransit, Automatonymous is free, open source, and licensed under the very permissive Apache 2.0 license, making usable at no cost to anyone for both commercial and non-commercial use.

Automatonymous Quick Start

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

So you've got the chops and want to get started quickly using Automatonymous. Maybe you are a bad ass and can't be bothered with reading documentation, or perhaps you are already familiar with the Magnum StateMachine and want to see what things have changed. Either way, here it is, your first state machine configured using Automatonymous.

```
class Relationship
{
    public State CurrentState { get; set; }
    public string Name { get; set; }
}

class RelationshipStateMachine :
    MassTransitStateMachine<Relationship>
{
    public RelationshipStateMachine()
    {
        Event(() => Hello);
        Event(() => PissOff);
        Event(() => Introduce);

        State(() => Friend);
        State(() => Enemy);

        Initially(
            When(Hello)
                .TransitionTo(Friend),
            When(PissOff)
                .TransitionTo(Enemy),
            When(Introduce)
                .Then(ctx => ctx.Instance.Name = ctx.Data.Name)
                .TransitionTo(Friend)
        );
    }

    public State Friend { get; private set; }
    public State Enemy { get; private set; }

    public Event Hello { get; private set; }
    public Event PissOff { get; private set; }
    public Event<Person> Introduce { get; private set; }
}

class Person
{
    public string Name { get; set; }
}
```

Seriously?

Okay, so two classes are defined above, one that represents the state (`Relationship`) and the other that defines the behavior of the state machine (`RelationshipStateMachine`). For each state machine that is defined, it is expected that there will be at least one instance. In Automatonymous, state is separate from behavior, allowing many instances to be managed using a single state machine.

Note: For some object-oriented purists, this may be causing the hair to raise on the back of your neck. Chill out, it's not the end of the world here. If you have a penchant for encapsulating behavior with data (practices such as domain model, DDD, etc.), recognize that programming language constructs are the only thing in your way here.

Tracking State

State is managed in Automatonymous using a class, shown above as the `Relationship`.

Defining Behavior

Behavior is defined using a class that inherits from `MassTransitStateMachine`. The class is generic, and the state type associated with the behavior must be specified. This allows the state machine configuration to use the state for a better configuration experience.

Note: It also makes Intellisense work better.

States are defined in the state machine as properties. They are initialized by default, so there is no need to declare them explicitly unless they are somehow special, such as a Substate or Superstate.

Creating Instances

Creating the State Machine

Raising Events

Once a state machine and an instance have been created, it is necessary to raise an event on the state machine instance to invoke some behavior. There are three or four participants involved in raising an event: a state machine, a state machine instance, and an event. If the event includes data, the data for the event is also included.

The most explicit way to raise an event is shown below.

```
var relationship = new Relationship();
var machine = new RelationshipStateMachine();

await machine.RaiseEvent(relationship, machine.Hello);
```

If the event has data, it is passed along with the event as shown.

```
var person = new Person { Name = "Joe" };

await machine.RaiseEvent(relationship, machine.Introduce, person);
```

Lifters

Lifters allow events to be raised without knowing explicit details about the state machine or the instance type, making it easier to raise events from objects that do not have prior type knowledge about the state machine or the instance. Using an approach known as *currying* (from functional programming), individual arguments of raising an event can be removed.

For example, using an event lift, the state machine is removed.

```
var eventLift = machine.CreateEventLift(machine.Hello);

// elsewhere in the code, the lift can be used
await eventLift.Raise(relationship);
```

The instance can also be lifted, making it possible to raise an event without any instance type knowledge.

```
var instanceLift = machine.CreateInstanceLift(relationship);
var helloEvent = machine.Hello;

// elsewhere in the code, the lift can be used
await instanceLift.Raise(helloEvent);
```

Lifts are commonly used by plumbing code to avoid dynamic methods or delegates, making code clean and fast.

Sample State Machine

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Automatonymous is a state machine engine written by Chris Patterson, aka, PhatBoyG.

Note: The following code will require installing a few NuGet dependencies.

- `MassTransit`
- `Automatonymous`
- `MassTransit.Automatonymous`

Defining a state machine saga

To define a state machine saga, create a class that inherits from `MassTransitStateMachine`. The `MassTransit.Automatonymous` NuGet package must be referenced.

Note: This section is written using the shopping cart sample, which is hosted on GitHub.

```
public class ShoppingCartStateMachine :
    MassTransitStateMachine<ShoppingCart>
{
```

```
    public ShoppingCartStateMachine()
    {
        InstanceState(x => x.CurrentState);
```

The state machine class, and the specification of the property for the current state of the machine are defined first.

```
Event(() => ItemAdded, x => x.CorrelateBy(cart => cart.UserName, context => context.
↪Message.UserName)
    .SelectId(context => Guid.NewGuid()));
```

The event that is observed when an item is added to the cart, along with the correlation between the state machine instance and the message are defined. The id generator for the saga instance is also defined.

```
Event(() => Submitted, x => x.CorrelateById(context => context.Message.CartId));
```

The order submitted event, and the correlation for that order.

```
Schedule(() => CartExpired, x => x.ExpirationId, x =>
{
    x.Delay = TimeSpan.FromSeconds(10);
    x.Received = e => e.CorrelateById(context => context.Message.CartId);
});
```

In order to schedule the timeout, a schedule is defined, including the time delay for the scheduled event, and the correlation of the event back to the state machine.

Now, it is time for the actual behavior of the events and how they interact with the state of the *ShoppingCart*.

```
Initially(
    When(ItemAdded)
        .Then(context =>
        {
            context.Instance.Created = context.Data.Timestamp;
            context.Instance.Updated = context.Data.Timestamp;
            context.Instance.UserName = context.Data.UserName;
        })
        .ThenAsync(context => Console.Out.WriteLineAsync($"Item Added: {context.Data.
↪UserName} to {context.Instance.CorrelationId}"))
        .Schedule(CartExpired, context => new CartExpiredEvent(context.Instance))
        .TransitionTo(Active)
    );
```

Initially defined events that can create a state machine instance. In the above, the properties of the instance are initialized, and then the *CartExpired* event is scheduled, after which the state is set to *Active*.

```
During(Active,
    When(Submitted)
        .Then(context =>
        {
            if (context.Data.Timestamp > context.Instance.Updated)
                context.Instance.Updated = context.Data.Timestamp;
            context.Instance.OrderId = context.Data.OrderId;
        })
        .ThenAsync(context => Console.Out.WriteLineAsync($"Cart Submitted: {context.
↪Data.UserName} to {context.Instance.CorrelationId}"))
        .Unschedule(CartExpired)
        .TransitionTo(Ordered),
```

While the shopping cart is active, if the order is submitted, the expiration is canceled (via *Unschedule*) and the state is set to Ordered.

```
When(ItemAdded)
    .Then(context =>
    {
        if (context.Data.Timestamp > context.Instance.Updated)
            context.Instance.Updated = context.Data.Timestamp;
    })
    .ThenAsync(context => Console.Out.WriteLineAsync($"Item Added: {context.Data.
↪UserName} to {context.Instance.CorrelationId}"))
    .Schedule(CartExpired, context => new CartExpiredEvent(context.Instance)),
```

If another item is added to the cart, the *CartExpired* event is scheduled, and the existence of a previously scheduled event (via the *ExpirationId* property) is used to cancel the previously scheduled event.

```
When(CartExpired.Received)
    .ThenAsync(context => Console.Out.WriteLineAsync($"Item Expired: {context.
↪Instance.CorrelationId}"))
    .Publish(context => new CartRemovedEvent(context.Instance))
    .Finalize()
);
```

If the *CartExpired* event is received, the cart removed event is published and the shopping cart is finalized.

```
    SetCompletedWhenFinalized();
}
```

Signals that the state machine instance should be deleted if it is finalized. This is used to tell Entity Framework to delete the row from the database.

```
public State Active { get; private set; }
public State Ordered { get; private set; }
```

The states of the shopping cart (*Initial* and *Final* are built-in states).

```
public Schedule<ShoppingCart, CartExpired> CartExpired { get; private set; }
```

The schedule definition for the CartExpired event.

```
    public Event<CartItemAdded> ItemAdded { get; private set; }
    public Event<OrderSubmitted> Submitted { get; private set; }
}
```

The events that are observed by the state machine (the correlations are defined earlier in the state machine).

The state machine is generic, and requires a state class (because sagas are stateful), so that is defined below. The state class has the values that are persisted between events.

```
class ShoppingCartState :
    SagaStateMachineInstance
{
    public Guid CorrelationId { get; set; }
```

The CorrelationId is the primary key of the saga state instance. It is assigned either from a property on the initial message that creates the saga instance, or can be generated using `NewId.NextGuid()`, which ensures a nice ordered sequential identifier.

```
public string CurrentState { get; set; }
```

The current state of the saga, which can be saved as a *string* or an *int*, depending upon your database requirements. An *int* is smaller, but requires that all valid states be mapped to integers during the definition of the state machine.

```
public Guid? ExpirationId { get; set; }
```

This is an identifier that is used by the state machine's scheduling feature, to capture the scheduled message identifier. Message scheduling within sagas is a powerful feature, which is described later.

```
    public string UserName { get; set; }

    public DateTime Created { get; set; }
    public DateTime Updated { get; set; }


    public Guid? OrderId { get; set; }
}
```

The remainder of the properites are relevant to the application, and are saved when properly mapped using the saga repository (which can be any supported storage engine, Entity Framework and NHibernate are supported out of the box).

Connecting the saga to a receive endpoint

To connect the state machine saga to a receive endpoint, a saga repository is used, along with the state machine instance.

```
var repository = new InMemorySagaRepository<ShoppingCartState>();

_busControl = Bus.Factory.CreateUsingRabbitMq(x =>
{
    IRabbitMqHost host = x.Host(...);

    x.ReceiveEndpoint(host, "shopping_cart_state", e =>
    {
        e.PrefetchCount = 8;
        e.StateMachineSaga(_machine, repository);
    });

    x.UseInMemoryMessageScheduler(); // for testing, to make it easy
});
```

Combining events (think Fork/Join)

Multiple events can be combined into a single event, for the purposes of joining together multiple operations. To define a combined event, the `Event` syntax has an overload.

```
public Event<OrderReady> Ready { get; private set; }
public Event<PaymentApproved> Approved { get; private set; }
public Event<StockVerified> Verified { get; private set; }

CompositeEvent(() => OrderReady, x => x.OrderReadyStatus, PaymentApproved,␣
↪StockVerified);
```

Once both events have been delivered to the state machine, the third event, *OrderReady*, will be triggered.

Note: The order of events being declared can impact the order in which they execute. Therefore, it is best to declare composite events at the end of the state machine declaration, after all other events and behaviors are declared. That way, the composite events will be raised *after* the dependent event behaviors.

Persisting Saga Instances

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Sagas are stateful event-based message consumers – they retain state. Therefore, saving state between events is important. Without persistent state, a saga would consider each event a new event, and orchestration of subsequent events would be meaningless.

Identity

Saga instances are identified by a unique identifier (`Guid`), represented by the `CorrelationId` on the saga instance. Events are correlated to the saga instance using either the unique identifier, or alternatively using an expression that correlates properties on the saga instance to each event. If the `CorrelationId` is used, it's always a one-to-one match, either the saga already exists, or it's a new saga instance. With a correlation expression, the expression might match to more than one saga instance, so care should be used – because the event would be delivered to all matching instances.

Note: Seriously, don't sent an event to all instances – unless you want to watch your messages consumers lock your entire saga storage engine.

Publishing and Sending From Sagas

Sagas are completely message-driven and therefore not only consume but also publish events and send commands. However, if your saga received a lot of messages coming roughly at the same time and the endpoint is set to process multiple messages in parallel - this can lead to a conflict between message processing and saga persistence.

This means that there could be more than one saga state updates that are being persisted at the same time. Depending on the saga repository type, this might fail for different reasons - versioning issue, row or table lock or eTag mismatch. All those problems are basically saying that you are having a concurrency issue.

It is normal for the saga repository to throw an exception in such case but if your saga is publishing messages, they were already published but the saga state has not been updated. MassTransit will eventually use retry policy on the endpoint and more messages will be send, potentially leading to mess. Or, if there are no retry policies configured, messages might be sent indicating that the process needs to continue but saga instance will be in the old state and will not accept any further messages because they will come in a wrong state.

This issue is common and can be solved by postponing the message publish and send operations until all persistence work is done. All messages that should be published, are collected in a buffer, which is called "outbox". MassTransit implements this feature and it can be configured by adding these lines to your endpoint configuration:

```
    c.ReceiveEndpoint("queue", e =>
    {
        e.UseInMemoryOutbox();
        // other endpoint configuration here
    }
```

Storage Engines

MassTransit supports several storage engines, including NHibernate, Entity Framework, and MongoDB. Each of these are setup in a similar way, but examples are shown below for each engine.

Entity Framework

Entity Framework seems to be the most common ORM for class-SQL mappings, and SQL is still widely used for storing data. So it's a win to have it supported out of the box by MassTransit. The code-first mapping example below shows the basics of getting started.

```
public class SagaInstance :
    SagaStateMachineInstance
{
    public SagaInstance(Guid correlationId)
    {
        CorrelationId = correlationId;
    }

    protected SagaInstance()
    {
    }

    public string CurrentState { get; set; }
    public string ServiceName { get; set; }
    public Guid CorrelationId { get; set; }
}


public class SagaInstanceMap :
    SagaClassMapping<SagaInstance>
{
    public SagaInstanceMap()
    {
        Property(x => x.CurrentState);
        Property(x => x.ServiceName, x => x.Length(40));
    }
}
```

The repository is then created on the context factory for the `DbContext` is available.

```
SagaDbContextFactory contextFactory = () =>
    new SagaDbContext<SagaInstance, SagaInstanceMap>(_connectionString);

var repository = new EntityFrameworkSagaRepository<SagaInstance>(contextFactory);
```

MongoDB

MongoDB is an easy to use saga repository, because setup is easy. There is no need for class mapping, the saga instances can be persisted easily using a MongoDB collection.

```
public class SagaInstance :
    SagaStateMachineInstance
{
    public SagaInstance(Guid correlationId)
    {
        CorrelationId = correlationId;
    }

    protected SagaInstance()
    {
    }

    public string CurrentState { get; set; }
    public string ServiceName { get; set; }
    public Guid CorrelationId { get; set; }
}
```

The saga repository is created using the simple syntax:

```
var database = new MongoClient("mongodb://127.0.0.1").GetDatabase("sagas");
var repository = new MongoDbSagaRepository<SagaInstance>(database);
```

Each saga instance will be placed in a collection specific to the instance type.

NHibernate

Although NHibernate is not being actively developed recently, it is still widely used and is supported by MassTransit for saga storage. The example below shows the code-first approach to using NHibernate for saga persistence.

```
public class SagaInstance :
    SagaStateMachineInstance
{
    public SagaInstance(Guid correlationId)
    {
        CorrelationId = correlationId;
    }

    protected SagaInstance()
    {
    }

    public string CurrentState { get; set; }
    public string ServiceName { get; set; }
    public Guid CorrelationId { get; set; }
}


public class SagaInstanceMap :
    SagaClassMapping<SagaInstance>
{
    public SagaInstanceMap()
    {
```

```
        Property(x => x.CurrentState);
        Property(x => x.ServiceName, x => x.Length(40));
    }
}
```

The `SagaClassMapping` base class maps the `CorrelationId` of the saga, and handles some of the basic bootstrapping of the class map. All of the properties, including the property for the `CurrentState` (if you're using state machine sagas), must be mapped by the developer. Once mapped, the `ISessionFactory` can be created using NHibernate directly. From the session factory, the saga repository can be created.

```
ISessionFactory sessionFactory = CreateSessionFactory();
var repository = new NHibernateSagaRepository<SagaInstance>(sessionFactory);
```

Redis

Redis is a very popular key-value store, which is known for being very fast.

Redis does not support queries, therefore Redis saga persistence only supports correlation by id. If you try to use correlation by expressions, you will get a "not implemented" exception.

Saga persistence for Redis uses `ServiceStack.Redis` library and it support both BSD-licensed v3.9.71 and the latest commercial versions as well.

Saga instance class must implement `IHasGuid` interface and the `Id` property, that must return the value of the `CorrelationId` property:

```
public int Version { get; set; }
```

When version of the instance that is being updated will be lower than the expected version, the saga repository will trow an exception and force the message to be retried, potentially resolving the issue.

The Redis saga repository requires `ServiceStack.Redis.IRedisClientsManager` as constructor parameter. For containerless initialization the code would look like:

```
var redisConnectionString = "redis://localhost:6379";
var repository = new RedisSagaRepository<SagaInstance>(new␣
↪RedisManagerPool(redisConnectionString));
```

If you use a container, you can use the code like this (example for Autofac):

```
var redisConnectionString = "redis://localhost:6379";
builder.Register<IRedisClientsManager>(c => new␣
↪RedisManagerPool(redisConnectionString)).SingleInstance();
builder.RegisterGeneric(typeof(RedisSagaRepository<>)).As(typeof(ISagaRepository<>)).
↪SingleInstance();
```

CHAPTER 5

Using Courier

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Developing applications using a distributed, message-based architecture significantly increases the complexity of performing operations transactionally, where an end-to-end set of steps much be completed entirely, or not at all. In an application using an ACID database, this is typically done using SQL transactions, where partial operations are rolled back if the transaction cannot be completed. However, this doesn't scale when the steps being to include dependencies outside of a single database. And in the distributed, *micro-services* based architectures, the use of a single ACID database is shrinking to completely non-existent.

MassTransit Courier is a mechanism for creating and executing distributed transactions with fault compensation that can be used to meet the requirements previously within the domain of database transactions, but built to scale across a large system of distributed services. Courier also works well with MassTransit sagas, which add transaction monitoring and recoverability.

Using a Routing Slip

A routing slip specifies a sequence of processing steps called *activities* that are combined into a single transaction. As each activity completes, the routing slip is forwarded to the next activity in the itinerary. When all activities have completed, the routing slip is completed and the transaction is complete.

A key advantage to using a routing slip is it allows the activities to vary for each transaction. Depending upon the requirements for each transaction, which may differ based on things like payment methods, billing or shipping address, or customer preference ratings, the routing slip builder can selectively add activities to the routing slip. This dynamic behavior is in contrast to a more explicit behavior defined by a state machine or sequential workflow that is statically defined (either through the use of code, a DSL, or something like Windows Workflow).

MassTransit Courier

MassTransit Courier is a framework that implements the routing slip pattern. Leveraging a durable messaging transport and the advanced saga features of MassTransit, Courier provides a powerful set of components to simplify the use of routing slips in distributed applications. Combining the routing slip pattern with a [state machine such as Automatonymous](https://github.com/phatboyg/Automatonymous) results in a reliable, recoverable, and supportable approach for coordinating and monitoring message processing across multiple services.

In addition to the basic routing slip pattern, MassTransit Courier also supports compensations which allow activities to store execution data so that reversible operations can be undone, using either a traditional rollback mechanism or by applying an offsetting operation. For example, an activity that holds a seat for a patron could release the held seat when compensated.

Activities

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

In MassTransit Courier, an *Activity* refers to a processing step that can be added to a routing slip. To create an activity, create a class that implements the *Activity* interface.

```
public class DownloadImageActivity :
    Activity<DownloadImageArguments, DownloadImageLog>
{
    Task<ExecuteResult> Execute(ExecutionContext<DownloadImageArguments> context);
    Task<CompensationResult> Compensate(CompensateContext<DownloadImageLog> context);
}
```

The *Activity* interface is generic with two arguments. The first argument specifies the activity's input type and the second argument specifies the activity's log type. In the example shown above, *DownloadImageArguments* is the input type and *DownloadImageLog* is the log type. Both arguments must be interface types so that the implementations can be dynamically created.

Execute Activities

An *Execute Activity* is an activity that only executes and does not support compensation. As such, the definition of a log type is not required.

```
public class ValidateImageActivity :
    ExecuteActivity<ValidateImageArguments>
{
    Task<ExecuteResult> Execute(ExecutionContext<DownloadImageArguments> context);
}
```

Implementing an activity

An activity must implement two interface methods, *Execute* and *Compensate*. The *Execute* method is called while the routing slip is executing activities and the *Compensate* method is called when a routing slip faults and needs to be compensated.

When the *Execute* method is called, an *execution* argument is passed containing the activity arguments, the routing slip *TrackingNumber*, and methods to mark the activity as completed or faulted. The actual routing slip message, as well as any details of the underlying infrastructure, are excluded from the *execution* argument to prevent coupling between the activity and the implementation. An example *Execute* method is shown below.

```
async Task<ExecutionResult> Execute(Execution<DownloadImageArguments> execution)
{
    DownloadImageArguments args = execution.Arguments;
    string imageSavePath = Path.Combine(args.WorkPath,
        execution.TrackingNumber.ToString());

    await _httpClient.GetAndSave(args.ImageUri, imageSavePath);

    return await execution.Completed(new DownloadImageLogImpl(imageSavePath));
}
```

Once activity processing is complete, the activity returns an *ExecutionResult* to the host. If the activity executes successfully, the activity can elect to store compensation data in an activity log which is passed to the *Completed* method on the *execution* argument. If the activity chooses not to store any compensation data, the activity log argument is not required. In addition to compensation data, the activity can add or modify variables stored in the routing slip for use by subsequent activities.

Note: In the example above, the activity creates an instance of a private class that implements the *DownloadImageLog* interface and stores the log information in the object properties. The object is then passed to the *Completed* method for storage in the routing slip before sending the routing slip to the next activity.

Compensating an activity

When an activity fails, the *Compensate* method is called for previously executed activities in the routing slip that stored compensation data. If an activity does not store any compensation data, the *Compensate* method is never called. The compensation method for the example above is shown below.

```
Task<CompensationResult> Compensate(Compensation<DownloadImageLog> compensation)
{
    DownloadImageLog log = compensation.Log;
    File.Delete(log.ImageSavePath);

    return compensation.Compensated();
}
```

Using the activity log data, the activity compensates by removing the downloaded image from the work directory. Once the activity has compensated the previous execution, it returns a *CompensationResult* by calling the *Compensated* method. If the compensating actions could not be performed (either via logic or an exception) and the inability to compensate results in a failure state, the *Failed* method can be used instead, optionally specifying an *Exception*.

Building a routing slip

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Developers are discouraged from directly implementing the *RoutingSlip* message type and should instead use a *RoutingSlipBuilder* to create a routing slip. The *RoutingSlipBuilder* encapsulates the creation of the routing slip and includes methods to add activities, activity logs, and variables to the routing slip. For example, to create a routing slip with two activities and an additional variable, a developer would write:

```
var builder = new RoutingSlipBuilder(NewId.NextGuid());
builder.AddActivity("DownloadImage", "rabbitmq://localhost/execute_downloadimage", new
    {
        ImageUri = new Uri("http://images.google.com/someImage.jpg")
    });
builder.AddActivity("FilterImage", "rabbitmq://localhost/execute_filterimage");
builder.AddVariable("WorkPath", "\\dfs\work");

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

Each activity requires a name for display purposes and a URI specifying the execution address. The execution address is where the routing slip should be sent to execute the activity. For each activity, arguments can be specified that are stored and presented to the activity via the activity arguments interface type specify by the first argument of the *Activity* interface. The activities added to the routing slip are combined into an *Itinerary*, which is the list of activities to be executed, and stored in the routing slip.

Note: Managing the inventory of available activities, as well as their names and execution addresses, is the responsibility of the application and is not part of the MassTransit Courier. Since activities are application specific, and the business logic to determine which activities to execute and in what order is part of the application domain, the details are left to the application developer.

Executing the routing slip

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Once built, the routing slip is executed, which sends it to the first activity's execute URI. To make it easy and to ensure that source information is included, an extension method on *IBus* is available, the usage of which is shown below.

```
await bus.Execute(routingSlip);
```

It should be pointed out that if the address for the first activity is invalid or cannot be reached, an exception will be thrown by the *Execute* method.

Monitoring routing slip execution

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

During routing slip execution, events are published when the routing slip completes or faults. Every event message includes the *TrackingNumber* as well as a *Timestamp* (in UTC, of course) indicating when the event occurred:

- RoutingSlipCompleted
- RoutingSlipFaulted
- RoutingSlipCompensationFailed

Additional events are published for each activity, including:

- RoutingSlipActivityCompleted
- RoutingSlipActivityFaulted
- RoutingSlipActivityCompensated
- RoutingSlipActivityCompensationFailed

By observing these events, an application can monitor and track the state of a routing slip. To maintain the current state, an Automatonymous state machine could be created. To maintain history, events could be stored in a database and then queried using the *TrackingNumber* of the routing slip.

Subscriptions

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

By default, routing slip events are published – which means that any subscribed consumers will receive the events. While this is useful getting started, it can quickly get out of control as applications grow and multiple unrelated routing slips are used. To handle this, subscriptions were added (yes, added, because they weren't though of until we experienced this ourselves).

Subscriptions are added to the routing slip at the time it is built using the `RoutingSlipBuilder`.

```
builder.AddSubscription(new Uri("rabbitmq://localhost/log-events"), RoutingSlipEvents.
↪All);
```

This subscription would send all routing slip events to the specified endpoint. If the application only wanted specified events, the events can be selected by specifying the enumeration values for those events. For example, to only get the `RoutingSlipCompleted` and `RoutingSlipFaulted` events, the following code would be used.

```
builder.AddSubscription(new Uri("rabbitmq://localhost/log-events"), RoutingSlipEvents.
↪Completed | RoutingSlipEvents.Faulted);
```

It is also possible to tweak the content of the events to cut down on message size. For instance, by default, the `RoutingSlipCompleted` event includes the variables from the routing slip. If the variables contained a large document, that document would be copied to the event. Eliminating the variables from the event would reduce the message size, thereby reducing the traffic on the message broker. To specify the contents of a routing slip event subscription, an additional argument is specified.

```
builder.AddSubscription(new Uri("rabbitmq://localhost/log-events"), RoutingSlipEvents.
↪Completed, RoutingSlipEventContents.None);
```

This would send the `RoutingSlipCompleted` event to the endpoint, without any of the variables be included (only the main properties of the event would be present).

Note: Once a subscription is added to a routing slip, events are no longer published – they are only sent to the addresses specified in the subscriptions. However, multiple subscriptions can be specified – the endpoints just need to be known at the time the routing slip is built.

Custom events

It is also possible to specify a subscription with a custom event, a message that is created by the application developer. This makes it possible to create your own event types and publish them in response to routing slip events occurring. And this includes having the full context of a regular endpoint `Send` so that any headers or context settings can be applied.

To create a custom event subscription, use the overload shown below.

```
    // first, define the event type in your assembly
public interface OrderProcessingCompleted
{
    Guid TrackingNumber { get; }
    DateTime Timestamp { get; }

    string OrderId { get; }
    string OrderApproval { get; }
}

// then, add the subscription with the custom properties
    builder.AddSubscription(new Uri("rabbitmq://localhost/order-events"),␣
↪RoutingSlipEvents.Completed,
        x => x.Send<OrderProcessingCompleted>(new
    {
        OrderId = "BFG-9000",
        OrderApproval = "ComeGetSome"
    }));
```

In the message contract above, there are four properties, but only two of them are specified. By default, the base `RoutingSlipCompleted` event is created, and then the content of that event is *merged* into the message created in the subscription. This ensures that the dynamic values, such as the `TrackingNumber` and the `Timestamp`, which are present in the default event, are available in the custom event.

Custom events can also select with contents are merged with the custom event, using an additional method overload.

CHAPTER 6

Using middleware

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit is composed of a network of pipelines, which are used to dispatch messages from the transport, through the receive endpoint, past deserialization, and ultimately to the consumers. And these pipelines are entirely asynchronous, making them very fast and very flexible.

Middleware components can be added to every pipeline in MassTransit, allowing for complete customization of message processing. And the granular ways that middleware can be applied make it easy to focus a particular behavior into a single receive endpoint, a single consumer, a saga, or the entire bus.

Middleware components are configured via extension methods on any pipe configurator (`IPipeConfigurator<T>`), and the extension methods all begin with `Use` to separate them from other methods.

The details of many of the built-in middleware components follow.

Using the circuit breaker

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found **here_**.

A circuit breaker is used to protect resources (remote, local, or otherwise) from being overloaded when in a failure state. For example, a remote web site may be unavailable and calling that web site in a message consumer takes 30-60

seconds to time out. By continuing to call the failing service, the service may be unable to recover. A circuit breaker detects the repeated failures and trips, preventing further calls to the service and giving it time to recover. Once the reset interval expires, calls are slowly allowed back to the service. If it is still failing, the breaker remains open, and the timeout interval resets. Once the service returns to healthy, calls flow normally as the breaker closes.

Read Martin Fowler's description of the pattern here.

To add the circuit breaker to a receive endpoint:

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.UseCircuitBreaker(cb =>
        {
            cb.TrackingPeriod = TimeSpan.FromMinutes(1);
            cb.TripThreshold = 15;
            cb.ActiveThreshold = 10;
            cb.ResetInterval = TimeSpan.FromMinutes(5);
        });

        e.Consumer(() => new UpdateCustomerAddressConsumer(sessionFactory));
    });
```

There are four settings that can be adjusted on a circuit breaker.

TrackingPeriod The window of time before the success/failure counts are reset to zero. This is typically set to around one minute, but can be as high as necessary. More than ten seems really strange to me.

TripThreshold This is a percentage, and is based on the ratio of successful to failed attempts. When set to 15, if the ratio exceeds 15%, the circuit breaker opens and remains open until the `ResetInterval` expires.

ActiveThreshold This is the number of messages that must reach the circuit breaker in a tracking period before the circuit breaker can trip. If set to 10, the trip threshold is not evaluated until at least 10 messages have been received.

ResetInterval The period of time between the circuit breaker trip and the first attempt to close the circuit breaker. Messages that reach the circuit breaker during the open period will immediately fail with the same exception that tripped the circuit breaker.

Using the rate limiter

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

A rate limiter is used to restrict the number of messages processed within a time period. The reason may be that an API or service only accepts a certain number of calls per minute, and will delay any subsequent attempts until the rate

limiting period has expired.

Note: The rate limiter will delay message delivery until the rate limit expires, so it is best to avoid large time windows and keep the rate limits sane. Something like 1000 over 10 minutes is a bad idea, versus 100 over a minute. Try to adjust the values and see what works for you.

There are two modes that a rate limiter can operate, but only of them is currently supported (the other may come later).

To add a rate limiter to a receive endpoint:

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.UseRateLimit(1000, TimeSpan.FromSeconds(5));

        e.Consumer(() => new UpdateCustomerAddressConsumer(sessionFactory));
    });
```

The two arguments supported by the rate limiter include:

RateLimit The number of calls allowed in the time period.

Interval The time interval before the message count is reset to zero.

Using the latest filter

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

The latest filter is pretty simple, it keeps track of the latest message received by the pipeline and makes that value available. It seems pretty simple, and it is, but it is actually useful in metrics and monitoring scenarios.

Note: This filter is actually usable to capture any context type on any pipe, so you know.

To add a latest to a receive endpoint:

```
ILatestFilter<ConsumeContext<Temperature>> tempFilter;

var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
```

```
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Handler<Temperature>(context => Task.FromResult(true), x =>
        {
            x.UseLatest(x => x.Created = filter => tempFilter = filter);
        })
    });
```

Creating your own middleware

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Middleware components are configured using extension methods, to make them easy to discover. By default, a middleware configuration method should start with Use.

An example middleware component that would log exceptions to the console is shown below.

```
Bus.Factory.CreateUsingInMemory(cfg =>
{
    cfg.UseExceptionLogger();
});
```

The extension method creates the pipe specification for the middleware component, which can be added to any pipe. For a component on the message consumption pipeline, use `ConsumeContext` instead of any `PipeContext`.

```
public static class ExampleMiddlewareConfiguratorExtensions
{
    public static void UseExceptionLogger<T>(this IPipeConfigurator<T> configurator)
        where T : class, PipeContext
    {
        configurator.AddPipeSpecification(new ExceptionLoggerSpecification<T>());
    }
}
```

The pipe specification is a class that adds the filter to the pipeline. Additional logic can be included, such as configuring optional settings, etc. using a closure syntax similar to the other configuration classes in MassTransit.

```
public class ExceptionLoggerSpecification<T> :
    IPipeSpecification<T>
    where T : class, PipeContext
{
    public IEnumerable<ValidationResult> Validate()
    {
        return Enumerable.Empty<ValidationResult>();
    }

    public void Apply(IPipeBuilder<T> builder)
```

```
    {
        builder.AddFilter(new ExceptionLoggerFilter<T>());
    }
}
```

Finally, the middleware component itself is a filter connected to the pipeline (inline). All filters have absolute and complete control of the execution context and flow of the message. Pipelines are entirely asynchronous, and expect that asynchronous operations will be performed.

Warning: Do not use legacy constructs such as .Wait, .Result, or .WaitAll() as these can cause blocking in the message pipeline. While they might work in same cases, you've been warned!

```
public class ExceptionLoggerFilter<T> :
    IFilter<T>
    where T : class, PipeContext
{
    long _exceptionCount;
    long _successCount;
    long _attemptCount;

    public void Probe(ProbeContext context)
    {
        var scope = context.CreateFilterScope("exceptionLogger");
        scope.Add("attempted", _attemptCount);
        scope.Add("succeeded", _successCount);
        scope.Add("faulted", _exceptionCount);
    }

    public async Task Send(T context, IPipe<T> next)
    {
        try
        {
            Interlocked.Increment(ref _attemptCount);

            await next.Send(context);

            Interlocked.Increment(ref _successCount)
        }
        catch (Exception ex)
        {
            Interlocked.Increment(ref _exceptionCount);

            await Console.Out.WriteLineAsync($"An exception occurred: {ex.Message}");

            // propagate the exception up the call stack
            throw;
        }
    }
}
```

CHAPTER 7

Scheduling messages

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Time is important, especially in distributed applications. Quartz.NET is an excellent scheduler, and is used in many applications to schedule jobs in a variety of ways.

MassTransit uses Quartz.NET to schedule messages, making it possible to build complex time-based workflows. Several extensions are available to message consumers, as well as middleware for using message scheduling.

In a production system, Quartz.NET is run as a service with multiple instances active for high availability and load balancing. Quartz can use any SQL database to coordinate scheduled jobs across servers, making it suitable for this type of use.

There is a standalone MassTransit service, MassTransit.QuartzService, which can be installed and used on servers for this purpose. It is configured via the `app.config` file and is a good example of how to build a standalone MassTransit service.

Note: This service will likely move to be hosted in the new MassTransit.Host, making the service logic reusable across any message transport.

Using the scheduling API

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

The scheduling API consists of several extension methods that send messages to an endpoint where the Quartz scheduling consumers are connected.

Configuring the quartz address

The bus has an internal context that is used to make it so that consumers that need to schedule messages do not have to be aware of the specific scheduler type being used, or the message scheduler address. To configure the address, use the extension method shown below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.UseMessageScheduler(new Uri("rabbitmq://localhost/quartz"));
});
```

Once configured, messages may be scheduled from any message consumer as shown below.

Scheduling a message from a consumer

To schedule a message, call the `ScheduleSend` method with the message to be delivered.

```
public interface ScheduleNotification
{
    DateTime DeliveryTime { get; }
    string EmailAddress { get; }
    string Body { get; }
}

public interface SendNotification
{
    string EmailAddress { get; }
    string Body { get; }
}

public class ScheduleNotificationConsumer :
    IConsumer<ScheduleNotification>
{
    Uri _notificationService;

    public async Task Consume(ConsumeContext<ScheduleNotification> context)
    {
        await context.ScheduleSend(_notificationService,
            context.Message.DeliveryTime,
            new SendNotification
            {
                EmailAddress = context.Message.EmailAddress,
                Body =  context.Message.Body
```

```
            });
    }

    class SendNotificationCommand :
        SendNotification
    {
        public string EmailAddress { get; set; }
        public string Body { get; set; }
    }
}
```

The `ScheduleMessage` command message will be sent to the Quartz endpoint, which will schedule a job in Quartz to deliver the message (and save the message body to be delivered). When the job is triggered, the message will be sent to the destination address.

Scheduling a message from the bus

If a message needs to be scheduled from the bus itself (not in the context of consuming a message), the SendEndpoint for the quartz scheduler should be retrieved and used to schedule the send.

```
var schedulerEndpoint = await bus.GetSendEndpoint(_schedulerAddress);

await schedulerEndpoint.ScheduleSend(_notificationService,
    context.Message.DeliveryTime,
    new SendNotification
    {
        EmailAddress = context.Message.EmailAddress,
        Body =  context.Message.Body
    });
```

This should only be used outside of a consume context, however, as the lineage of the message will be lost (things like ConversationId, InitiatorId, etc.).

Scheduling a recurring message

You can also schedule a message to be send to you periodically. This functionality uses the Quartz.Net periodic schedule feature and requires some knowledge of cron expressions.

To request a recurring message, you need to use *ScheduleRecurringSend* extension method, which is available for both *Context* and *SendEndpoint*. This message requires a schedule object as a parameter, which must implement *RecurringSchedule* interface. Since this interface is rather broad, you can use the default abstract implementation *DefaultRecurringSchedule* as the base class for your own schedule.

```
public class PollExternalSystemSchedule : DefaultRecurringSchedule
{
    public PollExternalSystemSchedule()
    {
        CronExpression = "* * * * *"; // this means every minute
    }
}

public class PollExternalSystem {}
```

```
var schedulerEndpoint = await bus.GetSendEndpoint(_schedulerAddress);

var scheduledRecurringMessage = await schedulerEndpoint.
↪ScheduleRecurringSend(InputQueueAddress, new PollExternalSystemSchedule(), new␣
↪PollExternalSystem());
```

When you stop your service or just have any other need to tell Quartz service to stop sending you these recurring messages, you can use the return value of *ScheduleRecurringSend* to cancel the recurring schedule.

```
await bus.CancelScheduledRecurringMessage(scheduledRecurringMessage);
```

You can also cancel using schedule id and schedule group values, which are part of the recurring schedule object.

Hosting Quartz In-Memory

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

One of the nice features of quartz is that it can run entirely in memory without any additional dependencies.

To use Quartz in-memory for message scheduling:

1. Use the NuGet package manager to add the `MassTransit.Quartz` package to your project.
2. Add the extension method to your bus configuration.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.UseInMemoryScheduler();
});
```

The `UseInMemoryScheduler` method initializes Quartz.NET for standalone in-memory operation, and adds a receive endpoint to the bus named "quartz", which hosts the consumers for scheduling messages.

Warning: Using the in-memory scheduler uses non-durable storage. If the process terminates, any scheduled messages will be lost, immediately, never to be found again. For any production system, using a standalone service is recommended with persistent storage.

Redelivering messages

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

There are situations where a message cannot be processed, either due to an unavailable resource or a situation in which message ordering is important (you should try not to depend upon message order, but sometimes it is an easy workaround). In these situations, scheduling a message for redelivery is a powerful tool.

Note: Sometimes this behavior is referred to as a *Second Level Retry*.

Specifying redelivery

MassTransit makes it easy to schedule messages for redelivery. In the example below, the Quartz service is running as a separate service on the */quartz* queue.

```
public class UpdateCustomerAddressConsumer :
    IConsumer<UpdateCustomerAddress>
{
    public async Task Consume(ConsumeContext<ScheduleNotification> context)
    {
        try
        {
            // try to update the database
        }
        catch (CustomerNotFoundException exception)
        {
            // schedule redelivery in one minute
            context.Redeliver(TimeSpan.FromMinutes(1));
        }
    }
}
```

To enable the `Redeliver` method, the Quartz endpoint must be setup on the bus.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.UseMessageScheduler(new Uri("rabbitmq://localhost/quartz"));
});
```

The redelivered message includes two additional message headers:

MT-Redelivery-Count The number of redelivery attempts the message has had. The first attempt is number 1.

MT-Scheduling-DeliveredAddress The address where the message was last delivered and subsequently scheduled for redelivery.

Scheduling with Azure Service Bus

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

Azure Service Bus allows the enqueue time of a message to be specified, making it possible to schedule messsages without the use of a separate message scheduler. MassTransit makes it easy to take advanced of this feature by configuring the bus scheduler to specify the enqueue time for scheduled messages.

Configuring the enqueue time scheduler

To configure the bus (or a receive endpoint) to use the enqueue time for message scheduling, add the code below to the configuration.

```
var busControl = Bus.Factory.CreateUsingAzureServiceBus(cfg =>
{
    var host = cfg.Host(serviceAddress, h =>
    {
        // ...
    });

    cfg.UseServiceBusMessageScheduler();
});
```

This configures the bus scheduler, which is available via the MessageSchedulerContext interface. Once configured, the message scheduling extensions can be used (which are available on the ConsumeContext). For example, to schedule a message for future delivery from within a message consumer.

```
public class ScheduleNotificationConsumer :
    IConsumer<AssignSeat>
{
    Uri _schedulerAddress;
    Uri _notificationService;

    public async Task Consume(ConsumeContext<AssignSeat> context)
    {
        if(context.Message.ReservationTime - DateTime.Now < TimeSpan.FromHours(8))
        {
            // assign the seat for the reservation
        }
        else
        {
            // seats can only be assigned eight hours before the reservation

            context.ScheduleMessage(context.Message.ReservationTime - TimeSpan.
↪FromHours(8), context.Message);
        }
```

```
    }
}
```

This will schedule the message to be delivered to the consumer endpoint at the specified time.

Note: If the message should be sent to a different endpoint, the destination address can be specified as an additional parameter.

CHAPTER 8

Advanced Topics

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit has several advanced features and provides some powerful capabilities to meet specific requirements. To keep the main documentation easy to follow, advanced topics are covered separately as they are, well, advanced.

Versioning messages

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Versioning of messages is going to happen, services evolve and requirements change.

Versioning existing message contracts

Consider a command to fetch and cache a local copy of an image from a remote system.

```
public interface FetchRemoteImage
{
    Guid CommandId { get; }
    DateTime Timestamp { get; }
    Uri ImageSource { get; }
```

```
    string LocalCacheKey { get; }
}
```

After the initial deployment, a requirement is added to resize the image to a maximum dimension before saving it to the cache. The new message contract includes the additional property specifying the dimension.

```
public interface FetchRemoteImage
{
    Guid CommandId { get; }
    DateTime Timestamp { get; }
    Uri ImageSource { get; }
    string LocalCacheKey { get; }
    int? MaximumDimension { get; }
}
```

By making the *int* value nullable, commands that are submitted using the original contract can still be accepted as the missing value does not break the new contract. If the value was added as a regular *int*, it would be assigned a default value of zero, which may not convey the right information. String values can also be added as they will be *null* if the value is not present in the serialized message. The consumer just needs to check if the value is present and process it accordingly.

Versioning existing events

Consider an event to notify that an image has been cached is now available.

```
public interface RemoteImageCached
{
    Guid EventId { get; }
    DateTime Timestamp { get; }
    Guid InitiatingCommandId { get; }
    Uri ImageSource { get; }
    string LocalCacheKey { get; }
}
```

An application will publish the event using an implementation of the class, as shown below.

```
class RemoteImageCachedEvent :
    RemoteImageCached
{
    Guid EventId { get; set; }
    DateTime Timestamp { get; set; }
    Guid InitiatingCommandId { get; set; }
    Uri ImageSource { get; set; }
    string LocalCacheKey { get; set; }
}
```

The class implements the event interface, and when published, is delivered to consumers that are subscribed to the *RemoteImageCached* event interface. MassTransit dynamically creates a backing class for the interface, and populates the properties with the values from the serialized message.

> Note that you cannot dynamically cast the *RemoteImageCached* interface in the consumer to the RemoteImageCachedEvent, as the actual class is not deserialized. This can be confusing, but is intentional to prevent classes (and the behavior that comes along with it) from being serialized and deserialized.

As the event evolves, additional event contracts can be defined that include additional information without modifying the original contract. For example.

```
public interface RemoteImageCachedV2
{
    Guid EventId { get; }
    DateTime Timestamp { get; }
    Guid InitiatingCommandId { get; }
    Uri ImageSource { get; }

    // the string is changed from LocalCacheKey to a full URI
    Uri LocalImageAddress { get; }
}
```

The event class is then modified to include the additional property, while still implementing the previous interface.

```
class RemoteImageCachedEvent :
    RemoteImageCached,
    RemoteImageCachedV2
{
    Guid EventId { get; set; }
    DateTime Timestamp { get; set; }
    Guid InitiatingCommandId { get; set; }
    Uri ImageSource { get; set; }
    string LocalCacheKey { get; set; }
    Uri LocalImageAddress { get; set; }
}
```

When the event class is published now, both interfaces are available in the message. When a consumer subscribes to one of the interfaces, that consumer will receive a copy of the message. It is important that both interfaces are not consumed in the same context, as duplicates will be received. If a service is updated, it should use the new contract.

Note: Note that ownership of the contract belongs to the event publisher, not the event observer/subscriber. And contracts should not be shared between event producers as this can create some extensive leakage of multiple events making it difficult to consume unique events.

As mentioned above, depending upon the interface type subscribed, a dynamic backing class is created by MassTransit. Therefore, if a consumer subscribes to RemoteImageCached, it is not possible to cast the message to RemoteImageCachedV2, as the dynamic implementation does not support that interface.

Note: It should be noted, however, that on the IConsumeContext interface, there is a method to TryGetContext<T> method, which can be used to attempt to deserialize the message as type T. So it is possible to check if the message also implements the new version of the interface and not process as the original version knowing that the new version will be processed on the same message consumption if both types are subscribed.

The message is a single message on the wire, but the available/known types are captured in the message headers so that types can be deserialized from the message body.

A lot of flexibility and power, it's up to the application developer to ensure that it is used in a way that ensures application evolution over time without requiring forklift/switchover upgrades due to breaking message changes.

Long-running tasks using Turnout

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Message consumers are similar to web controllers in that they are intended to live for a short time – the time it takes to process a single message. However, there are times when the processing *initiated* by a message takes long time (for instance, longer than a minute). Rather than block the message consumer and preventing it from consuming additional messages, a way to start an asynchronous task is required.

Turnout enables the execution of long-running tasks initiated by a message consumer. Turnout manages the lifetime of the task, and appropriately handles failure and handing off to other nodes in the event of a server failure.

Note: Turnout is an early-access feature, and it's only been tested with some basic scenarios. It works, but there are rough edges, so consider it like running with scissors.

Configurating a Turnout

To configure a turnout, a turnout endpoint is created for the command message type.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.UseInMemoryScheduler();

    cfg.TurnoutEndpoint<AuditCustomerHistory>(host, "audit_consumer_history", e =>
    {
        x.SuperviseInterval = TimeSpan.FromSeconds(30);
        x.SetJobFactory(async context =>
        {
            await Task.Delay(TimeSpan.FromMinutes(7), context.CancellationToken);
        })
    });
});
```

Turnout - Under the hood

When a turnout endpoint is created, the queue name specified is used to create several queues. First, the *audit_consumer_history* queue is created, as it is the queue specified by the developer. This is where the commands are sent, just like a normal consumer.

A second queue, *audit_consumer_history-expired* is also created, and is used as the dead-letter queue for scheduled messages which are not consumed by the service. More on this in a minute, but this is used to handle node failures gracefully.

Note: Node failures such as this are still being developed, so, consider some of this forward looking. The simplest case already works though.

A third queue, *turnout-???*, is created to allow the node to send commands to itself to supervise the state of the Task. Messages are sent to this endpoint by the node (each node gets a unique queue) using the message scheduler (this is a great place to use the built-in scheduling of Azure Service Bus, or the delayed exchange with RabbitMQ).

Job supervision

Rather than rely on in-process timers, Turnout uses the queuing and scheduling features to supervise jobs. Messages are scheduled every interval to check on the status of the job. If the job completed, it is removed from the job registry. If it is still running, a new supervision command is scheduled. Each command is scheduled for a specific time, and the time-to-live is set to the supervision interval (plus a delta to ensure it has time to be received).

If the node crashes (does not cleanly shut down), the messages will not be received and will dead letter to the *-expired* queue. All nodes listen on that queue, and if a node receives a command that was a job being processed on another node, it will handle that as a faulted command. Since the node is typically running smoothly, the messages are consumed on time.

Note: Faulted job handling has yet to be decided, but will be configurable to either follow a retry policy, or just report that the job faulted.

Stopping a node

When a node is stopped, any running jobs are aborted (different than being cancelled). This way, the job is reported as aborted and another node can pickup the job to continue processing.

Interoperability

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

In MassTransit, developers specify types for messages. MassTransit's serializers then perform the hard work of converting the types to the serializer format (such as JSON, XML, BSON, etc.) and then back again.

To interoperate with other languages and platforms, the message structure is important.

Content type

To support custom message types, MassTransit uses a transport-level header to specify the message format. MassTransit simultaneously supports the following message formats on a single transport.

- json (application/vnd.masstransit+json)
- bson (application/vnd.masstransit+bson)

- xml (application/vnd.masstransit+xml)

If you enable encryption:

- aes (application/vnd.masstransit+aes)

If you configure the binary serializer:

- binary (application/vnd.masstransit+binary)

Register custom types would during endpoint/bus configuration.

JSON/BSON/XML

MassTransit uses a message envelope to encapsulate the built-in message headers, as well as the message payload. The envelope properties on the wire include:

```
string MessageId
string CorrelationId
string ConversationId
string InitiatorId
string RequestId
string SourceAddress
string DestinationAddress
string ResponseAddress
string FaultAddress
DateTime? ExpirationTime
IDictionary<string, object> Headers
object Message
string[] MessageType
HostInfo Host
```

The *Id* values should be convertible to a GUID/UUID or they will fail. All are optional, but MessageId should be present at a minimum.

The *Address* values should be convertible to a URI that is a valid MassTransit endpoint address.

The *MessageType* entries should be URNs, which are convertible to .NET types. MassTransit defines the format of the URN in the following structure:

The *Host* is an internal data type, but is a set of strings that define the host that produced the message.

```
string MachineName
string ProcessName
int ProcessId
string Assembly
string AssemblyVersion
string FrameworkVersion
string MassTransitVersion
string OperatingSystemVersion
```

Examples include:

urn:message:MyProject.Messages:UpdateAccount urn:message:MyProject.Messages.Events:AccountUpdated
urn:message:MyProject:ChangeAccount urn:message:MyProject.AccountService:MyService+AccountUpdatedEvent

The last one is a nested class, as indicated by the '+' symbol.

Example message

This is a minimal message:

```
{
    "destinationAddress": "rabbitmq://localhost/input_queue",
    "headers": {},
    "message": {
        "value": "Some Value",
        "customerId": 27
    },
    "messageType": [
        "urn:message:MassTransit.Tests:ValueMessage"
    ]
}
```

Encrypted Messages

If you use the encrypted message serializer, it uses BSON under the hood. The encryption format is AES-256. Assuming the same Key/IV pair, an encrypted message should be compatible across the wire.

CHAPTER 9

Samples

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Samples are a great way to learn about MassTransit features. By seeing a feature used in a working application, it makes it easier to explore and understand. A developer can clone the repository, open the solution in Visual Studio, and walk through the code.

The new samples are standalone repositories, which use NuGet to pull dependencies exactly as a developer would use MassTransit.

The samples are described in the following sections.

Courier sample

.. attention:: **This page is obsolete!**

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here_.

.. _here: http://masstransit-project.com/MassTransit/learn/samples/courier.html

Courier is MassTransit's routing-slip implementation, which makes it possible to orchestrate distributed services into a business transaction. This sample demonstrates how to create and execute a routing slip, record routing slip events, and track transaction state using Automatonymous.

> Automatonymous is a free, open-source state machine library with native MassTransit support.

Git the sample

1. Clone the source down to your machine. `git clone https://github.com/MassTransit/Sample-Courier.git`
2. Open the solution using Visual Studio

The sample includes multiple console applications, which can be started simultaneously using the solution properties start options.

Request response

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Clone the sample: GitHub Repository

This sample demonstrates how to create a client that sends a request to a service which responds with a response.

The abandoned cart saga

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Clone the sample: GitHub Repository

This was a fun sample, created in response to a blog post on how to send an email to a customer that abandoned a shopping cart. My response to that post is located here.

CHAPTER 10

Configuring MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Note: This is earlier documentation that is being migrated into the Using section, so start there before coming here.

Now that you have MassTransit installed into your project, we need to get it configured so that you can start bringing the awesome! First we have a quickstart which will be followed up with a few configuration examples.

Show me the code!

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

All right, all right, already. Here you go. Below is a functional setup of MassTransit.

```
1  public class YourMessage { public string Text { get; set; } }
2  public class Program
3  {
4      public static void Main()
5      {
6          var bus = Bus.Factory.CreateUsingRabbitMq(sbc =>
7          {
```

```
            var host = sbc.Host(new Uri("rabbitmq://localhost"), h =>
            {
                h.Username("guest");
                h.Password("guest");
            });

            sbc.ReceiveEndpoint(host, "test_queue", ep =>
            {
                ep.Handler<YourMessage>(context =>
                {
                    return Console.Out.WriteLineAsync($"Received: {context.Message.
↪Text}");
                });
            });
        });

        bus.Start();

        bus.Publish(new YourMessage{Text = "Hi"});

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();

        bus.Stop();
    }
}
```

What is this doing?

If we are going to create a messaging system, we need to create a message. `YourMessage` is a .NET class that will represent our message. Notice that it's just a Plain Old CLR Object (or POCO).

Next up, we need a program to run our code. Here we have a standard issue command line `Main` method. To setup the bus we start with the static class `Bus` and work off of the `Factory` extension point. From there we call the `CreateUsingRabbitMQ` method to setup a RabbitMQ bus instance. This method takes a lambda whose first and only argument is a class that will let you configure every aspect of the bus.

One of your first decisions is going to be "What transport do I want to run on?" Here we have choosen RabbitMQ (`Bus.Factory.CreateUsingRabbitMQ()`) because its the defacto bus choice for MassTransit.

After that we need to configure the RabbitMQ host settings `sbc.Host()`. The first argument sets the machine name and the virtual directory to connect to. After that you have a lambda that you can use to tweak any of the other settings that you want. Here we can see it setting the username and password.

Now that we have a host to listen on, we can configure some receiving endpoints `sbc.ReceiveEndpoint`. We pass in the host connection to listen on, then which queue do we want to listen on, and finally a lambda to register each handler that we want to use.

Lastly, in the configuration, we have the `Handler<YourMessage>` method which subscribes a handler for the message type `YourMessage` and takes an `async` lambda (oh yeah baby TPL) that is given a context class to process. Here we access the message by traversing `context.Message` and then writing to the console the text of the message.

And now we have a bus instance that is fully configured and can start processing messages. We can grab the `busControl` that we created and call `Start` on it to get everyting rolling. We again `await` on the result and now we can go.

We can call the `Publish` method on the `busControl` and we should see our console write out the output.

Bus configuration

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Trying to get multiple applications to talk to each other is not a simple problem. Often times the biggest is just getting everything configured. With over eight years of experience in setting up message based systems the developers of MassTransit have tried to ensure that the defaults cover the majority of the decisions you will have to make while minimizing the configuration code you have to deal with. We hope that the options are clear and make sense why you need to select them. Below are some of the options you have:

Selecting a message transport

The first decision is what transport are you going to use? RabbitMQ or Azure Service Bus? If you don't know which one to choose I suggest reading up on the two and see which one works better for your environment.

```
Bus.Factory.CreateUsingInMemory(cfg => {});
Bus.Factory.CreateUsingRabbitMQ(cfg => {});
Bus.Factory.CreateUsingAzureServiceBus(cfg => {});
```

Warning: The InMemory transport is a great tool for testing, as it doesn't require a message broker to be installed or running. It's also very fast. But it isn't durable, and messages are gone if the bus is stopped or the process terminates. So, it's generally not a smart option for a production system. However, there are places where durability it not important so the cautionary tale is to proceed with caution.

Specifying a Host

Once the transport has been selected, the message host(s) must be configured. The host settings are transport specific, so the available options will vary. For instance, the InMemory transport does not require any configuration, because it's, well, in memory.

```
var busControl = Bus.Factory.CreateUsingInMemory(cfg =>
{
});
```

RabbitMQ Specific

For RabbitMQ, a URI specifying the host (and virtual host, default is /) should be provided, along with additional configuration for the username/password, as well as options on the transport.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/a_virtual_host"), h =>
    {
        h.Username("guest");
        h.Password("guest");
```

```
    });
});
```

You can also specify the nodes in a cluster to use the RabbitMQ driver's built-in failover capabilities. When a connection is interrupted, a new node will be selected and connected to.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://myclustername/a_virtual_host"), h =>
    {
        h.Username("guest");
        h.Password("guest");
        h.UseCluster(c =>
        {
            c.Node("rabbit1");
            c.Node("rabbit2");
            c.Node("rabbit3");
        });
    });
});
```

Note: The `myclustername` value specified in the `Uri` is strictly cosmetic when using clustering in this way. The actual Uri will be rewritten to use a node hostname from the cluster node list.

Azure Specific

For Azure Service Bus, a URI specifying the namespace should be provided, along with the `TokenProvider` for a token with **manage** permissions.

```
var busControl = Bus.Factory.CreateUsingAzureServiceBus(cfg =>
{
    var host = cfg.Host(new Uri("sb://my-namespace.servicebus.windows.net/"), h =>
    {
        h.TokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(
↪"KeyName", "keyvalue");
    });
});
```

Specifying a receive endpoint

Once the hosts are configured, any number of receive endpoints can be configured. No receive endpoints are required, a send/publish only bus is totally legit. An example of configuring a RabbitMQ host with a single receive endpoint is shown below.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });
```

```
    cfg.ReceiveEndpoint(host, "service_queue", ep =>
    {
    });
});
```

Selecting an outbound message serializer

By default, outbound messages are serialized using JSON and inbound messages that are in JSON, BSON, or XML can be deserialized. To use a different outbound message format, the default serializer can be changed. If a custom serializer has been created, use the `SetDefaultSerializer` extension to specify the factory methods for the custom serializer.

```
var busControl = Bus.Factory.CreateUsingInMemory(cfg =>
{
    cfg.UseBinarySerializer();
    cfg.UseBsonSerializer();
    cfg.UseJsonSerializer();
    cfg.UseXmlSerializer();
});
```

Transport configuration

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

Each transport has different configuration options.

RabbitMQ configuration options

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

> **Warning:** This page has not been updated yet.

This is the recommended approach for configuring MassTransit for use with RabbitMQ.

```
Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    cfg.Host(new Uri("rabbitmq://a-machine-name/a-virtual-host"), host =>
    {
        host.Username("username");
```

```
        host.Password("password");
    });
});
```

Routing topology

About the RabbitMQ routing topology in place with MassTransit.

- networks are segregated by vhosts
- we generate an exchange for each queue so that we can do direct sends to the queue. it is bound as a fanout exchange
- for each message published we generate series of exchanges that go from concrete class to each of its subclass / interfaces these are linked together from most specific to least specific. This way if you subscribe to the base interface you get all the messages. or you can be more selective. all exchanges in this situation are bound as fanouts.
- the subscriber declares his own queue and his queue exchange - he then also declares/binds his exchange to each of the message type exchanges desired
- the publisher discovers all of the exchanges needed for a given message, binds them all up and then pushes the message into the most specific queue letting RabbitMQ do the fanout for him. (One publish, multiple receivers!)
- control queues are exclusive and auto-delete - they go away when you go away and are not shared.
- we also lose the routing keys. WIN!

RabbitMQ with SSL

```
ServiceBusFactory.New(c =>
{
    c.ReceiveFrom(inputAddress);
    c.UseRabbitMqRouting();
    c.UseRabbitMq(r =>
    {
        r.ConfigureHost(inputAddress, h =>
        {
            h.UseSsl(s =>
            {
                s.SetServerName(System.Net.Dns.GetHostName());
                s.SetCertificatePath("client.p12");
                s.SetCertificatePassphrase("Passw0rd");
            });
        });
    });
});
```

You will need to configure RabbitMQ to support SSL also http://www.rabbitmq.com/ssl.html.

RabbitMQ with CloudAMQP

It is not necessary to set SSL specific configuration parameters in order to connect using SSL. What is required is specifying the appropriate SSL specific port (usually 5671 as opposed to the non-ssl port for RabbitMQ which is typically 5672).

```
ServiceBusFactory.New(c =>
{
    c.UseRabbitMq(r =>
    {
        r.ConfigureHost(host, port, virtualHost, h =>
        {
            h.UseSsl(s =>{ });
        });
    });
});
```

Azure Service Bus configuration options

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

```
Bus.Factory.CreateUsingAzureServiceBus(cfg =>
{
    cfg.Host(new Uri("sb://localhost"), host =>
    {
        host.OperationTimeout = TimeSpan.FromSeconds(5);
        host.TokenProvider = new ????();
    });
});
```

About Azure Service Bus

In-Memory transport

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Warning: The in-memory transport is designed for use within a single process only. It is not possible to use the in-memory transport to communicate between multiple processes (even if they are on the same machine). By the way, it is possible to share the same in-memory transport with multiple bus instances *within the same process* by configuring the transport provider using InMemoryTransportCache (see below).

Warning: The InMemory transport is a great tool for testing, as it doesn't require a message broker to be installed or running. It's also very fast. But it isn't durable, and messages are gone if the bus is stopped or the process terminates. So, it's generally not a smart option for a production system. However, there are places where durability it not important so the cautionary tale is to proceed with caution.

The In-Memory transport uses the `loopback` address (a holdover from previous version of MassTransit). The host doesn't matter, and the queue_name is the name of the queue.

loopback://localhost/queue_name

```
var busControl = Bus.Factory.CreateUsingInMemory(cfg =>
{
    cfg.ReceiveEndpoint("queue_name", ep =>
    {
       //configure the endpoint
    })
});
```

Sharing transports

While it seems weird, and again, it's probably only useful in test scenarios, the transport cache can be shared across bus instances. To share a transport cache, use the syntax below.

```
var inMemoryTransportCache = new InMemoryTransportCache(Environment.ProcessorCount);

var busControl = Bus.Factory.CreateUsingInMemory(cfg =>
{
    cfg.SetTransportProvider(inMemoryTransportCache);
});
```

As many bus instances as desired can be share the same cache. Again, useful for testing. Not sure I'd want to use this anywhere else.

Common gotchas

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Trying to share a queue

Note: While a common mistake in MassTransit 2.x, the new receive endpoint syntax of MassTransit 3 should make it easier to recognize that queue names should not be shared.

Each receive endpoint needs to have a unique queue name! If multiple receive endpoints are created, each should have a different queue name so that messages are not skipped.

If two receive endpoints share the same queue name, yet have different consumers subscribed, messages which are received by one endpoint but meant for the other will be moved to the _skipped queue. It would be like sharing a mailbox with your neighbor, sometimes you get all the mail, sometimes they get all the mail.

Send only bus

If you need to only send or publish messages, don't create any receive endpoints. The bus will automatically create a temporary queue for the bus which can be used to publish events, as well as send commands and do request/response conversations.

How do I load balance consumers across machines?

To load balance consumers, the process with the receive endpoint can be hosted on multiple servers. As long as each receive endpoint has the same consumers registered, the messages will be received by the first available consumer across all of the machines.

What links two bus instances together?

This is a common question. The binding element, really is the message contract. If you want message A, then you subscribe to message A. The internals of MT wires it all together.

Why aren't queue / message priorities supported?

Message Priorities are used to allow a message to jump to the front of the line. When people ask for this feature they usually have multiple types of messages all being delivered to the same queue. The problem is that each message has a different SLA (usually the one with the shorter time window is the one getting the priority flag). The problem is that w/o priorities the important message gets stuck behind the less important/urgent ones.

The solution is to stop sharing a single queue, and instead establish a second queue. In MassTransit you would establish a second instance of IServiceBus and have it subscribe to the important/urgent message. Now you have two queues, one for the important things and one for the less urgent things. This helps with monitoring queue depths, error rates, etc. By placing each IServiceBus in its own Topshelf host / process you further enhance each bus's ability to process messages, and isolate issues / downtime.

Reading

http://www.udidahan.com/2008/01/30/podcast-message-priority-you-arent-gonna-need-it/ http://lostechies.com/jimmybogard/2010/11/18/queues-are-still-queues/

I want to know if another bus is subscribed to my message.

> So, if you try to program this way, you're going to have a bad time. ;)

Knowing that you have a subscriber is not the concern of your application. It is something the system architect should know, but not the application. Most likely, we just need to introduce all of the states in our protocol more explicitly, by using a Saga.

CHAPTER 11

Troubleshooting MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Issues and possible solutions

- Messages are not being received by bus instances

Possible solutions

Do you have multiple buses listening on the same queue?

CHAPTER 12

Migrating from MassTransit v2.x to MassTransit v3

To migrate an application built with an earlier version of MassTransit, there are few changes that need to be considered.

Note: MSMQ is not supported in MassTransit 3. If you are still using MSMQ, you will need to remain on the 2.x version or migrate to RabbitMQ or Azure Service Bus.

The MassTransit v2.x API

In MassTransit 2.x, a bus was created for a transport using the `ServiceBusFactory` syntax.

```
public class Program
{
    public static void Main()
    {
        IServiceBus bus = ServiceBusFactory.New(sbc =>
        {
            sbc.UseRabbitMq();
            sbc.ReceiveFrom("rabbitmq://localhost/input_queue");
            sbc.Subscribe(subs =>
            {
                subs.Consumer<MyConsumer>();
            });
        });

        PublishMessage(bus);

        bus.Dispose();
    }

    public static void PublishMessage(IServiceBus bus)
    {
        bus.Publish(new YourMessage { Text = "Hi" });
```

```
    }
}
```

The MassTransit 3 API

The syntax for creating a bus using MassTransit 3 is different, where the transport is an initial decision point that must be made. The default retry policy should also be specified, as the default is no longer five attempts before moving to the error queue.

```
public class Program
{
    public static void Main()
    {
        IBusControl busControl = Bus.Factory.CreateUsingRabbitMq(sbc =>
        {
            var host = sbc.Host(new Uri("rabbitmq://localhost"), h =>
            {
                h.Username("guest");
                h.Password("guest");
            });

            sbc.UseRetry(Retry.Immediate(5));

            sbc.ReceiveEndpoint(host, "input_queue", ep =>
            {
                ep.Consumer<MyConsumer>();
            });
        });

        busControl.Start();

        PublishMessage(busControl)
            .Wait();

        busControl.Stop();
    }

    public static Task PublishMessage(IBus bus)
    {
        return bus.Publish(new YourMessage { Text = "Hi" });
    }
}
```

Major Changes

There are several API changes to consider, so they are summarized here.

IServiceBus to IBus

The `IServiceBus` interface is gone, replaced with `IBus`. This breaking change was done to ensure that in the switch to the new async methods that there were no accidental "didn't await" situations.

Also, `IBus` is really just a collection of other interfaces. In this case, it's unlikely that any part of the an application would ever need to take a dependency on `IBus` directly, but should instead opt for a narrower interface, such as `ISendEndpointProvider` or `IPublishEndpoint`. Each has a particular usefulness, but should only be used in cases where there is not an existing context which can be used.

Consumes<T>.* to IConsumer<T>

The clever `Consumes<T>.All` (and the related `Consumes<T>.Context`) are no longer supported. Instead, consumers should now use the single `IConsumer<T>` interface.

```
class AbConsumer :
    IConsumer<A>,
    IConsumer<B>
{
    public async Task Consume(ConsumeContext<A> context)
    {
    }

    public async Task Consume(ConsumeContext<B> context)
    {
    }
}
```

All consumer methods are now async and include the `ConsumeContext<T>` argument. The `context` parameter is incredibly useful, and should be used for anything message related. Both `IPublishEndpoint` and `ISendEndpointProvider` are implemented by the context, and should be used to send or publish messages. Doing so ensures that the `ConversationId` and `InitiatorId` are properly carried through the system.

Receive endpoints

In MassTransit v2, a separate bus had to be created for every queue. With MassTransit 3, that is no longer the case. Any number of receive endpoints can be configured on a single bus, reducing the overhead and memory usage, as well as the number of broker connections. This really helps with broker performance, as well as simplifies configuration.

It's also completely legal to create a bus with no receive endpoints. In this case, the bus is meant only for publish/send, as well as request/response. A temporary queue is created for the bus, on which responses can be received.

State machine sagas

Automatonymous is the only support state machine saga format with MassTransit 3. Magnum has been completely eradicated from the code base, with the new state machine engine being the go-forward standard. The integration with Automatonymous is great, including a specialized `MassTransitStateMachine` class, to allow advanced messages features such as request/response and timeouts to be supported.

Courier

The routing slip engine is now built into the main assembly, and has been updated to support event subscriptions (instead of just publishing all routing slip events). The routing slips are not backwards compatible, as the syntax has been improved to support better troubleshooting and event history. The API is mostly the same, though, so it's easy to migrate.

Living document

While the above items are just a few of the changes, this document will continue to be updated in response to questions about how to migrate code using previous features arise.

CHAPTER 13

Understanding MassTransit

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit is a large framework, and there is a lot going on under the hood. To understand what is happening, there are a number of subjects covered below that explain in detail how everything works.

Note: Okay, not everything – at least not yet. But we're working on getting it all documented.

Key terminology

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

When getting started using MassTransit, it is a good idea to have a handle on the terminology used in messaging. To ensure that you are on the right path when looking at a class or interface, here are some of the terms used when working with MassTransit.

Messages and serialization

MassTransit is a service bus, and a service bus is designed to move *messages*. At the lowest level, a message is a chunk of JSON, XML, or even binary data. When using a statically typed language (such as C#), a message is represented

by an instance of a class (or interface) that has relevant properties, each of which can be a value, list, dictionary, or even another nested class.

When using MassTransit, messages are sent and received, published and subscribed, as types. The translation (called serialization) between the textual representation of the message (which is JSON, XML, etc.) and a type is handled using a *message serializer*. The default serialization varies (for MSMQ, the framework uses XML by default, for RabbitMQ JSON is used instead). The default serialization can be changed when a service bus is being configured.

```
sbc.UseJsonSerializer(); // uses JSON by default
sbc.UseXmlSerializer();  // uses XML by default
sbc.UseBsonSerializer(); // uses BSON (binary JSON) by default
```

Sagas

All of the receiver types above are stateless by design, the framework makes no effort to correlate multiple messages to a single receiver. Often it is necessary to orchestrate multiple messages, usually of different types, into a saga (sometimes called a workflow). A saga is a long-running transaction that is managed at the application layer (instead of, for example, inside of a database or a distributed transaction coordinator). MassTransit allows sagas to be declared as a regular class or as a state machine using a fluent interface.

The key difference for sagas is that the framework manages the saga instance and correlates messages to the proper saga instance. This correlation is typically done using a *CorrelationId*, which is an interface (called *CorrelatedBy*). Messages correlated an individual saga must be done using a **Guid**. Sagas may also *observe* messages that are not correlated directly to the saga instance, but this should be done carefully to avoid potentially matching a message to hundreds of saga instances which may cause database performance issues.

```
public class MySaga :
    ISaga,
    InitiatedBy<MyInitialMessage>,
    Orchestrates<MyFollowUpMessage>
{
    public Guid CorrelationId { get; set; }
    public Task Consume(ConsumeContext<MyInitialMessage> message)
    {}
    public Task Consume(ConsumeContext<MyFollowUpMessage> message)
    {}
}
```

Transports and endpoints

MassTransit is a framework, and being a framework has certain rules. The first of which is known as the Hollywood principle – "Don't call us, we'll call you." Once the bus is configured and running, the receivers are called by the framework as messages are received. There is no need for the application to poll a message queue or repeated call a framework method in a loop.

Note: A way to understand this is to think of a message consumer as being similar to a controller in a web application. With a web application, the socket and HTTP protocol are under the hood, and the controller is created and action method called by the web framework. MassTransit is similar, in that the message reception is handled by MT, which then creates the consumer and calls the Consume method.

To initiate the calls into your application code, MassTransit creates an abstraction on top of the messaging platform (such as RabbitMQ).

Transports

The transport is at the lowest level and is closest to the actual message broker. The transport communicates with the broker, responsible for sending and receiving messages. The send and receive sections of the transport are completely independent, keeping reads and writes separate in line with the Command Query Responsibility Segregation pattern.

Receive endpoints

A receive endpoint receives messages from a transport, deserializes the message body, and routes the message to the consumers. Applications do not interact with receive endpoints, other than to configure and connect consumers. The rest of the work is done entirely by MassTransit.

Send endpoints

A send endpoint is used by an application to send a message to a specific adddress. They can be obtained from the `ConsumeContext` or the `IBus`, and support a variety of message types.

Endpoint addressing

MassTransit uses Universal Resource Identifiers (URIs) to identify endpoints. URIs are flexible and easy to include additional information, such as queue or exchange types. An example RabbitMQ endpoint address for *my_queue* on the local machine would be:

```
rabbitmq://localhost/my_queue
```

Under the hood

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit hides all the details of messages and delivery from the developer. However, when there are issues it is important to know how it works so you can troubleshoot the issues.

Setup

To see how this plays out, consider the following message types:

```
namespace MT.Messages {
        interface ISomeMessage {}
}
```

Configure the bus that will listen to `ISomeMessage` with an endpoint of `my_endpoint`.

Starting a bus

When creating the endpoint above you indicated the name of the queue where the messages will end up. See queue naming rules below. Starting the bus with the consumers registered, causes the following configuration to happen:

- A queue named ‘‘my_endpoint‘‘is created for all messages on this endpoint
- An exchange named `my_endpoint` is created for all messages on this endpoint
- An exchange named ‘‘MT.Messages.ISomeMessage‘‘is created for the message
- An exchange to exchange binding from `MT.Messages.ISomeMessage` to `my_endpoint` is created
- A binding from the `my_endpoint` exchange to `my_endpoint` queue is created.

Note: All exchanges created are of type FanOut

Publishing a message

When you publish a message on the bus here is what happens

- Publish `MT.Messages.ISomeMessage`
- This message gets published by the publishing logic to the exchange `MT.Messages.ISomeMessage`.
- The message is routed by messaging infrastructure to the `my_endpoint` exchange.
- The message is then routed to the `my_endpoint` queue.

If you publish a message before the consumer has been started (and created its configuration), the exchange `MT.Messages.ISomeMessage` will be created. It will not be bound to anything until the consumer starts, so if you publish to it, the message will just disappear.

Queues

- Each application you write should use a unique queue name.
- **If you run multiple copies of your consumer service, they would listen to the same queue (as they are copies).** This would mean you have multiple applications listening to `my_endpoint` queue This would result in a ‘competing consumer’ scenario. (Which is what you want if you run same service multiple times)
- If there is an exception from your consumer, the message will be sent to `my_endpoint_error` queue.
- If a message is received in a queue that the consumer does not know how to handle, the message will be sent to `my_endpoint_skipped` queue.

Design Benefits

- Any application can listen to any message and that will not affect any other application that may or may not be listening for that message
- Any application(s) that bind a group of messages to the same queue will result in the competing consumer pattern.
- You do not have to concern yourself with anything but what message type to produce and what message type to consume.

Faq

- **How many messages at a time will be simultaniously processed?**
  - Each endpoint you create represents 1 queue. That queue can receive any number of different message types (based on what you subscribe to it)
  - The configuration of each endpoint you can set the number of consumers with a call to `PrefetchCount(x)`.
  - This is the total number of consumers for all message types sent to this queue.
  - In MT2, you had to add ?prefetch=X to the Rabbit URL. This is handled automatically now.
- **Can I have a set number of consumers per message type?** Yes. This uses middleware. `x.Consumer(new AutofacConsumerFactory<...>(), p => p.UseConcurrencyLimit(1)); x.PrefetchCount=16;` PrefetchCount should be relatively high, a multiple of your concurrency limit for all message types so that RabbitMQ doesn't choke delivery messages due to network delays. Always have a queue ready to receive the message.
- **When my consumer is not running, I do not want the messages to wait in the queue. How can I do this?**

  There are two ways. Note that each of these imply you would never use a 'competing consumer' pattern, so make sur

  1. Set `PurgeOnStartup=true` in the endpoint configuration. When the bus starts, it will empty the queue of all messages.
  2. Set `AutoDelete=true` in the endpoint configuration. This causes the queue to be removed when your application stops.
- **How are Retrys handled?** This is handled by middleware Each endpoint has a retry policy.
- **Can I have a different retry policy per each message type?** No. This is set at an endpoint level. You would have to have a specific queue per consumer to achieve this.

What does MassTransit add to RabbitMQ?

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

MassTransit is a lightweight service bus for building distributed .NET applications. The main goal is to provide a consistent, .NET friendly abstraction over the message transport (whether it is RabbitMQ, Azure Service Bus, etc.). To meet this goal, MassTransit brings a lot of the application-specific logic closer to the developer in an easy to configure and understand manner.

The benefits of using MassTransit over the message transport, as opposed to using the raw transport APIs and building everything from scratch, are shown below. These are just a few, and some are more significant than others. The fact that the hosting of your consumers, handlers, sagas, etc. are all managed consistently with a well documented production ready framework is the biggest advantage. You can also find numerous blog posts, podcasts, and articles written about MassTransit online.

Concurrency

Concurrent, asynchronous message consumers for maximum receive throughput and high server utilization.

Connection management

The network is unreliable. If the application is disconnected from the message broker, MassTransit takes care of reconnecting and making sure all of the exchanges, queues, and bindings are restored.

Exception, retries, and poison messages

Your message consumers don't need to know about broker acknowledgement protocols. If your message consumer runs to completion, the message is acknowledged and removed from the queue. If you throw an exception, MassTransit uses a retry policy to redeliver the message to the consumer. If the retries are exhausted due to continued failures or other reasons, MassTransit moves the message to an error queue. If the message did not reach a consumer due to being misrouted to the queue, the message is moved to a skipped queue.

Serialization

C# is a statically typed language, and developers work with types. RabbitMQ works with bytes. So how do you format a message over the wire? How do you handle different date/time formats (local, UTC, unspecified)? How do your deal with numbers, are they integers, longs, or decimals? MassTransit has already thought about this and implemented sensible defaults for you. And there are many serializers provided out of the box, including JSON, BSON, and XML as well as the .NET binary formatter as a last resort.

You can even protect your messages using AES-256 encryption, to keep prying eyes away and to ensure the safety of private information (to meet PCI or HIPAA requirements).

Message header and correlation

Designing a common message envelope can be a nitty-gritty affair until things stabilize. And MassTransit is already stable having been used in production since 2008. The format is well documented and has been tested with billions of messages. Furthermore, the envelope includes headers for tracking messages, including conversations, correlations, and requests. The address and host information in the envelope make it easy to build any messaging pattern.

Consumer lifecycle management

MassTransit handles consumer creation and disposal, and integrates with most major dependency injection containers using their built-in lifetime scope management. This ensures that depedencies are created and destroyed as part of the message consumption pipeline.

Routing

MassTransit provides a heavily production tested convention for using RabbitMQ exchanges to route published messages to the subscribed consumers. The structure is CPU and memory friendly, which keeps RabbitMQ happy.

Rx integration

Interested in or already using Reactive Extensions? MassTransit makes it easy to connect Rx to RabbitMQ.

Unit testing made easy

One of the first rules of unit testing is to avoid hitting infrastructure. And RabbitMQ is just that. MassTransit includes a high-performance in-memory transport for testing every consumer using the same code that would be used in production. And the MassTransit.TestFramework NuGet package (which uses NUnit 2.6.4) includes an InMemoryTestFixture that handles the setup and teardown of the bus so you can easily test your message consumers.

Sagas

Sagas are a powerful abstraction that supports message orchestration with durable state. Whether you use the original somewhat explicit syntax, or the powerful state machine syntax of **Automatonymous**, you can build highly available distributed workflow and coordination services easily. MassTransit supports both Entity Framework and NHibernate, using code-based mapping and migrations to simply code deployments and upgrades.

Scheduling using Quartz.NET

MassTransit has strong integration with Quartz.NET, to make it easy to schedule messages for future delivery. This brings distributed applications into the fourth dimension, making time a first-class citizen. Some incredibly powerful routing systems have been built by the authors using Quartz in combination with other MassTransit features.

Monitoring performance counters

Keeping an eye on your services performance is critical, and having the right tools is a huge plus. MassTransit updates a range of performance counters as messages are processed so operations can keep an eye on message flow and compare the throughput to that of RabbitMQ.

Publishing messages

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

When a message is published (by way of a call to `bus.Publish`), it's important to understand what MassTransit actually does under the hood. While the explicit implementation details depend upon the message transport being used, the general pattern is the same.

MassTransit follows the publish subscribe message pattern, where a copy of the message is delivered to each subscriber. The message transport determines how the actual routing is performed, but the conventions of each transport are described below.

Routing on RabbitMQ

RabbitMQ provides powerful routing capabilities out of the box, in the form of exchanges and queues. Exchanges can be bound to queues, as well as other exchanges, making it easy to create a message routing fabric. MassTransit leverages exchanges and queues combined with the .NET type system to connect subscribers to publishers.

MassTransit uses the message type to declare exchanges and exchange bindings that match the hierarchy of types implemented by the message type. Interfaces are declared as separate exchanges (using a fully-qualified type name that is compatible with the naming structure of exchanges) and bound to the published message type exchange. When a message is first published, the exchanges are declared once, and then used for the life of the channel.

Note: Private types, such as classes, are declared as auto-delete so they do not clutter up the exchange namespace.

Once declared, published messages are to the message type exchange, and copies are routed to all the subscribers by RabbitMQ. This approach was based on an article on how to maximize routing performance in RabbitMQ.

This dynamic, type-based routing model has proved very powerful in many large applications. The ability to add new consumers to an existing message publisher is a great way to manage dependencies and keep projects from becoming tightly coupled.

To see how this plays out, consider the following message types:

```
namespace Company.Messages
{
    public interface CustomerAddressUpdated
    {
    }

    public interface UpdateCustomerAddress
    {
    }

    public class UpdateCustomerAddressCommand :
        UpdateCustomerAddress
    {
    }
}
```

Once the messages have been published, exchanges are created in RabbitMQ for each of the message types:

```
Exchanges

Company.Messages.CustomerAddressUpdated
Company.Messages.UpdateCustomerAddress
Company.Messages.UpdateCustomerAddressCommand
    - Includes a binding to Company.Messages.UpdateCustomerAddress
```

When a receive endpoint is started, the second half of the exchange/queue binding is performed, where the consumer subscriptions are bound to the consumer message type exchanges, closing the loop.

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    var host = cfg.Host(new Uri("rabbitmq://localhost/"), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    cfg.ReceiveEndpoint(host, "customer_update_queue", e =>
    {
        e.Consumer<UpdateCustomerConsumer>();
    });
});
```

This results in the creation of a queue, as well as a binding to the queue from the `UpdateCustomerAddress` exchange.

```
Exchanges

customer_update_queue
    - Includes a binding from Company.Messages.UpdateCustomerAddress

Queues

customer_update_queue
    - Includes a binding from the customer_update_queue exchange
```

Because RabbitMQ only allows messages to be sent to exchanges, an exchange matching the name of the queue is created and bound to the queue. This makes it easy to send messages directly to the queue using the same name. It's actually a pretty cool abstraction, and RabbitMQ makes it very flexible by allowing exchange-to-exchange bindings. By keeping the bindings at the exchange level, it eliminates any impact to message flow. Dru shared his experience with this as well.

Balancing the load

Because RabbitMQ is a message broker, it supports multiple readers from the same queue. This makes it super easy to setup a load balancing scenario where the same service is running on multiple servers, each of which is connected to the same queue. As messages arrive on the queue, they are delivered to the first available consumer that can receive the message. To get good load balancing, it's important to set the `` `PrefetchCount` `` to a sensible value on the consumer so that messages are well distributed.

Routing on Azure Service Bus

MassTransit uses a similar approach for Azure Service Bus, but uses Topics, Subscriptions, and Queues.

Note: More details to come...

Inheritance and message class design

> **Attention: This page is obsolete!**
>
> New documentation is located at http://masstransit-project.com/MassTransit.
>
> The latest version of this page can be found here.

That said, I would advise you to think about the following things: #. Interface-based inheritance is OK #. Class-based inheritance is to be approached with caution #. Composing messages together ends up pushing us into content-based routing which is something we don't recommend #. Message Design is not OO Design (A message is just state, no behavior) There is a greater focus on interop and contract design. #. As messages are more about contracts, we suggest subscribing to interfaces that way you can easily evolve the implementation of the message. #. A big base class may cause pain down the road as each change will have a larger ripple. This can be especially bad when you need to support multiple versions.

Performance counters

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

Masstransit has support for updating Windows performance counters. Chris has a post introducing them - Performance Counters Added to MassTransit. http://lostechies.com/chrispatterson/2009/10/14/performance-counters-added-to-masstransit/

User permissions

The user running your mass transit enabled application will need access to update the performance counters.

`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib`

If Masstransit does not detect the performance counters it wishes to write to it will attempt to create them. If the user credentials do not have administrative access likely they will not have the ability to create the performance counters and errors will be logged.

Windows installer

When deploying your mass transit enabled application it is possible to have Windows Installer create your performance counters for you. Below is Xml used by Wix 3.0 to define the Masstransit performance counters.

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"  xmlns:msmq="http://schemas.
↪microsoft.com/wix/MsmqExtension" xmlns:util="http://schemas.microsoft.com/wix/
↪UtilExtension">

...

<Component Id="masstransit_performance_counters" Guid="{E68DCC22-AD78-4bfe-A1F6-
↪29AA189FD76C}">
  <util:PerformanceCategory Id="perfCategoryMassTransit" Name="MassTransit" Help=
↪"MassTransit Performance Counters" MultiInstance="yes">
    <util:PerformanceCounter Name="Consumer Threads" Help="The current number of␣
↪threads processing messages." Type="numberOfItems32"/>
    <util:PerformanceCounter Name="Receive Threads" Help="The current number of␣
↪threads receiving messages."  Type="numberOfItems32"/>
    <util:PerformanceCounter Name="Received/s" Help="The number of messages received␣
↪per second" Type="rateOfCountsPerSecond32"/>
    <util:PerformanceCounter Name="Published/s" Help="The number of messages␣
↪published per second" Type="rateOfCountsPerSecond32"/>
    <util:PerformanceCounter Name="Sent/s" Help="The number of messages sent per␣
↪second" Type="rateOfCountsPerSecond32"/>
    <util:PerformanceCounter Name="Messages Received" Help="The total number of␣
↪message received." Type="numberOfItems32"/>
    <util:PerformanceCounter Name="Messages Published" Help="The total number of␣
↪message published." Type="numberOfItems32"/>
    <util:PerformanceCounter Name="Messages Sent" Help="The total number of message␣
↪sent." Type="numberOfItems32"/>
    <util:PerformanceCounter Name="Average Consumer Duration" Help="The average time␣
↪a consumer spends processing a message." Type="averageCount64"/>
```

```
    <util:PerformanceCounter Name="Average Consumer Duration Base" Help="The average␣
↪time a consumer spends processing a message." Type="averageBase"/>
    <util:PerformanceCounter Name="Average Receive Duration" Help="The average time␣
↪to receive a message." Type="averageCount64"/>
    <util:PerformanceCounter Name="Average Receive Duration Base" Help="The average␣
↪time to receive a message." Type="averageBase"/>
    <util:PerformanceCounter Name="Average Publish Duration" Help="The average time␣
↪to publish a message." Type="averageCount64"/>
    <util:PerformanceCounter Name="Average Publish Duration Base" Help="The average␣
↪time to publish a message." Type="averageBase"/>
  </util:PerformanceCategory>
</Component>
```

Videos

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

There are several videos of presentations featuring MassTransit.

Event Driven Architecture Presented at the North Dallas .NET User Group in February, 2010 by Chris Patterson.

http://www.drowningintechnicaldebt.com/ShawnWeisfeld/archive/2010/02/04/event-driven-architecture-by-chris-patterson-north-dalla
net.aspx

Others I'm sure, I just need to find them and link them here.

CHAPTER 14

Loving the community

Attention: This page is obsolete!

New documentation is located at http://masstransit-project.com/MassTransit.

The latest version of this page can be found here.

MassTransit has a great community of developers, many of which have contributed their time and energy to help make MassTransit what it is today. To that end, I felt it would be useful to compile some of the blog posts and articles that have been written by developers using MassTransit.

Reading about MassTransit

- Loosely Coupled Labs
  - MassTransit 3 Update A Simple Publish Subscribe Example
  - Error Handling in MassTransit Consumers
  - Monitoring RabbitMQ
- Running MassTransit with Topshelf
- SignalR Chat with MassTransit 3

CHAPTER 15

Indices and tables

- genindex
- modindex
- search