ReadMyDocs Documentation

Release 1.9.2

pardahlman

Jul 12, 2017

Contents

1 Getting Started **3**
1.1 Installation . . . 3
1.2 Creating instance . . . 3
1.3 Broker connection . . . 4
1.4 Messaging pattern . . . 5

2 Message Context **7**
2.1 Introduction . . . 7
2.2 Forwarding Context . . . 7
2.3 Default Context . . . 8
2.4 Advanced Context . . . 8
2.5 Custom Context . . . 8

3 Configuration **11**
3.1 Configuration options . . . 11
3.2 vNext configuration file . . . 12
3.3 ConnectionString . . . 13

4 Extending RawRabbit **15**
4.1 Installation . . . 15
4.2 The Extendable Bus Client . . . 15
4.3 Extension boiler plait . . . 15
4.4 List of extensions . . . 16

5 Bulk-fetching messages **17**

6 Update Topology **19**
6.1 Exchange updates . . . 19
6.2 Queue updates . . . 21

7 Message Sequence **23**
7.1 Alternative to RPC . . . 23
7.2 Multi-message sequence . . . 24

8 Publisher Acknowledgements **25**
8.1 Avoiding PublishConfirmException . . . 25

9 Message Priority **27**

9.1 Priority for specific messages 27
9.2 Setting priority based on message type 28

10 Multiple Subscribers **29**
10.1 Default behaviour 29
10.2 Custom Behaviour 29

11 Requeue with delay **31**
11.1 Later execution 31
11.2 Error strategy 31

12 Inner workings **33**
12.1 ChannelFactory 33
12.2 ConsumerFactory 34
12.3 TopologyProvider 34

13 RabbitMq features **35**
13.1 Lazy Queues 35

14 Error Handling **37**
14.1 Publish/Subscribe 37
14.2 Request/Respond 38

15 Logging **39**

16 Attribute based configuration **41**
16.1 Setting up the client 41
16.2 Configure Messages 41
16.3 Override with custom configuration 42

17 Client upgrade **43**
17.1 1.9.0 43
17.2 1.9.5 44

18 Contributing Guidelines **45**
18.1 Create issue 45
18.2 Write code 45
18.3 Commit Code 45
18.4 Create Pull Request 46

RawRabbit is a modern .NET client for communication over RabbitMq. It is written for vNext and is uses Microsoft's new frameworks for logging, configuration and dependecy injection It targets traditional NET runtimes, DNX runtimes and has all the ground work done for .NET Core .

Contents:

CHAPTER 1

Getting Started

Installation

Install the latest version of RawRabbit and RawRabbit.vNext from NuGet.

```
PM> Install-Package RawRabbit
PM> Install-Package RawRabbit.vNext
```

The vNext package contains the convenience class BusClientFactory that can be used to create a default instance of the RawRabbit client. It makes life easier, but is not necesary.

Creating instance

Depending on the scenario, there are a few different ways to instantiate the RawRabbit client. The methods described below all have optional arguments for registering specific subdependeices.

vNext Application wire-up

If the application is bootstrapped from a vNext application, the dependecies and client can be registed by using the AddRawRabbit extension for IServiceCollection

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRawRabbit(); //optional overrides here, too.
}
```

Instance from factory

Create a new client by calling `BusClientFactory.CreateDefault`. If no arguments are provided, the local configuration will be used (`guest` user on `localhost:5672` with virtual host `/`).

```
var raw = BusClientFactory.CreateDefault();
```

Autofac

The package `RawRabbit.DependencyInjection.Autofac` contains modules and extension methods for registering `RawRabbit`.

```
var builder = new ContainerBuilder();
builder.RegisterRawRabbit("guest:guest@localhost:5672/");
var container = builder.Build();
```

Ninject

The package `RawRabbit.DependencyInjection.Ninject` contains modules and extension methods for registering `RawRabbit`.

```
var kernel = new StandardKernel();
kernel.RegisterRawRabbit("guest:guest@localhost:5672/");
```

Broker connection

As soon as the client is instantiated, it will try to connect to the broker. By default `RawRabbit` will try to connect to `localhost`. Configuration can be provided in different ways.

Configuration object

The main configuration object for `RawRabbit` is `RawRabbitConfiguration`.

```
var config = new RawRabbitConfiguration
{
    Username = "user",
    Password = "password",
    Port = 5672,
    VirtualHost = "/vhost",
    Hostnames = { "production" }
    // more props here.
};
var client = BusClientFactory.CreateDefault(config);
```

Configuration can be supplied in configuration files. See the configuration section for more information.

Messaging pattern

Two of the main messaging patterns for RabbitMq are remote procedure calls (sometimes refered to as `RPC` or *request/reply*) and publish/subscribe.

Publish/Subscribe

Implementing the publish/subscribe pattern can be done with just a few lines of code. The `SubscribeAsyn<TMessage>` method takes one argument `Func<TMessage,TMessageContext,Task>` that will be invoked as the message is recived. Read more about the `TMessageContext` in the Message Context section. Publish a message by calling `PublishAsync<TMessage>` with an instance of the message as argument.

```
var client = BusClientFactory.CreateDefault();
client.SubscribeAsync<BasicMessage>(async (msg, context) =>
{
  Console.WriteLine($"Recieved: {msg.Prop}.");
});

await client.PublishAsync(new BasicMessage { Prop = "Hello, world!"});
```

Request/Reply

Similar to *publish/subscribe*, the message handler for a `RequestAsync<TRequest, TResponse>` in invoked with the request and message context. It returns a `Task<TResponse>` that is sent back to the waiting requester.

```
var client = BusClientFactory.CreateDefault();
client.RespondAsync<BasicRequest, BasicResponse>(async (request, context) =>
{
  return new BasicResponse();
});

var response = await client.RequestAsync<BasicRequest, BasicResponse>();
```

Other patterns

While publish/subscribe and request/reply lays in the core of `RawRabbit`, there are other ways to work with messages. The BulkGet extension (from NuGet `RawRabbit.Extensions`) allows for retrieving multiple messages from multiple queues and `Ack`/`Nack` them in bulk:

```
var bulk = client.GetMessages(cfg => cfg
    .ForMessage<BasicMessage>(msg => msg
        .FromQueues("first_queue", "second_queue")
        .WithBatchSize(4))
    .ForMessage<SimpleMessage>(msg => msg
        .FromQueues("another_queue")
        .GetAll()
        .WithNoAck()
    ));
```

CHAPTER 2

Message Context

Introduction

Messages that are sent through `RawRabbit` are delivered with a *message context*. Any class that implements `IMessageContext` can be used as a message context. This means that it is possible to replace the default context with a domain specific context. The goal is to seperate the message and its metadata/context.

Forwarding Context

Message context can be forwarded to subsequent message handlers. This is useful when a consumer communicates with other services that needs the message context to process the message correctly.

```
firstResponder.RespondAsync<FirstRequest, FirstResponse>((req, c) =>
{
   firstResponder
     .PublishAsync(new BasicMessage(), c.GlobalRequestId) //forward context.
     .ContinueWith(t => new FirstReponse());
});
```

Another useful aspect of forwarding message contexts is that the global request id can be traced though the different systems, making it easy to folllow a request from its originator to all systems that handle the message.

Example: User authorization

A user requests data from a UI. The user is authenticated and has a set of claims that allows the user to do access some *(but not all)* data. The request arrives at the backend of the UI. The endpoint knows what claims the user has, but the data is fetched from multiple underlying services communicate with over RabbitMq. Things like authentication and authorization doesn't have anything to do with the *request* itself, but it is something that the services needs to know of for filtering data. The message context for this setup should contain a list of the users claims, so that the service can evaluate if the requested action is authorized.

Default Context

The default message context, `MessageContext`, has only one member; `GlobalRequestId`.

Advanced Context

The `AdvancedMessageContext` contains properties that can be used to requeue message with delay and send negative acknowledgements. Note that there is *nothing magical* with the `AdvancedMessageContext`. It is just a custom context.

Instansiate bus with advanced context

The easiest way to create an instance of a `RawRabbit` client that uses an advanced context is to use the generic `CreateDefault<TMessageContext>` method on `BusClientFactory` (from `RawRabbit.vNext`).

```
var client = BusClientFactory.CreateDefault<AdvancedMessageContext>();
```

Custom Context

The Message Context

There are only two requirements for a message context class. It needs to implement `IMessageContext` and it needs to be serializable/deserializable by the registered `IMessageContextProvider<TMessageContext>` (by default `Newtonsoft.Json`).

```
public class CustomContext : IMessageContext
{
  public string CustomProperty { get; set; }
  public ulong DeliveryTag {get; set;}
  public Guid GlobalRequestId { get; set; }
}
```

The Context Provider

Message contexts are provided to the messages by the registered `IMessageContextProvider`. The default implementation, `MessageContextProvider<TMessageContext>` can be used for most context (typically `POCO` classes).

The Context Enhancer

A recieved message passes through the registered `IContextEnhancer` before any message handler is invoked. The method `WireUpContextFeatures` is called with the current context, consumer and `BasicDeliverEventArgs` (from `RabbitMQ.Client`).

```
public class CustomContextEnhancer : IContextEnhancer
{
  public void WireUpContextFeatures<TMessageContext>(TMessageContext context,␣
↪IRawConsumer consumer, BasicDeliverEventArgs args)
```

```
    where TMessageContext : IMessageContext
  {
    var customContext = context as CustomContext;
    if (customContext == null)
    {
      return;
    }
    customContext.DeliveryTag = args.DeliveryTag;
  }
}
```

The RawRabbit Client

The easist way to create a client is by using the generic `CreateDefault<TMessageContext>` method on `BusClientFactory`.

```
var client = BusClientFactory.CreateDefault<AdvancedMessageContext>();
```

The client can also be resolved from the service collection.

```
var service = new ServiceCollection()
  .AddRawRabbit<CustomContext>()
  .BuildServiceProvider();
var client = service.GetService<IBusClient<CustomContext>>();
```

CHAPTER 3

Configuration

As with most frameworks, the configuration of `RawRabbit` can be specified in either code or configuration. The easiest way to configure a vNext application is by using the optional parameter in the `IServiceCollection` extension:

```
private static void ConfigureApplication(IServiceCollection serviceCollection)
{
    serviceCollection
        .AddRawRabbit(
            cfg => cfg.AddJsonFile("rawrabbit.json"),
            ioc => ioc.AddTransient<ILogger, SerilogLogger>()
        );
}
```

If the application follows the pre vNext standards you can still leverage this syntax by using the `BusClientFactory` in `RawRabbit.vNext` package

```
BusClientFactory.CreateDefault(
    cfg => cfg.AddJsonFile("rawrabbit.json"),
    ioc => ioc.AddTransient<ILogger, SerilogLogger>()
)
```

Configuration options

Connecting to the broker

Username, *password*, *virtual host*, *port* and *hosts* are used for connecting to the host. Hosts is a list of strings that is passed to the registered `IConnectionFactory` when establishing a connection. It uses the default host selection strategy for `RabbitMQ.Client`, which is `RandomHostnameSelector` (as of `3.6.0`).

Recovery From Network Failures

`RawRabbit` supports automatic recovery of connection and topology. `AutomaticRecovery` (`bool`) indicates if recovery of connections, channels and QoS should be performed. If the recovery fails it, `RawRabbit` will wait for `RecoveryInterval` (`TimeSpan`) until retrying again. `AutomaticRecovery` (`bool`) includes recovery of exchanges, queues, bindings and consumers. More information about automatic recovering, see RabbitMq's .NET API guide (under section *Automatic Recovery From Network Failures*)

Operation timeouts

For request/reply, the `RequestTimeout` (`TimeSpan`) specifies the amout of time to wait for a response to arrive. `PublishConfirmTimeout` specifies the time to wait for a publish confirm from the broker.

Default topology settings

The default configuration for topology features (such as queue name, exchange type, auto delete) are specified in the `Exchange` (`GeneralExchangeConfiguration`) and `Queue` (`GeneralQueueConfiguration`) properties. These values can be overriden by custom configuration when specifying an operation.

Other

When `AutoCloseConnection` (`bool`) is set to `true`, a connection will be closed when the last channel has disconnected. Read more about this at RabbitMq's .NET API guide (under section *Disconnecting from RabbitMQ*).

`PersistentDeliveryMode` (`bool`) specifies if messages should be persisted to disk. While it affects performance, it makes the system more stabile for crashes/restart. Read more about it at RabbitMq's AMQP concept (under section *Message Attributes and Payload*)

vNext configuration file

Here's a sample of how the `rawrabbit.json` configuration file could look like

```
{
    "Username": "guest",
    "Password": "guest",
    "VirtualHost": "/",
    "Port": 5672,
    "Hostnames": [ "localhost" ],
    "RequestTimeout": "00:00:10",
    "PublishConfirmTimeout": "00:00:01",
    "RecoveryInterval": "00:00:10",
    "PersistentDeliveryMode": true,
    "AutoCloseConnection": true,
    "AutomaticRecovery": true,
    "TopologyRecovery": true,
    "Exchange": {
        "Durable": true,
        "AutoDelete": true,
        "Type": "Topic"
    },
    "Queue": {
        "AutoDelete": true,
        "Durable": true,
        "Exclusive": true
    }
}
```

ConnectionString

RawRabbit also supports configuration from connection strings. The syntax is username:password@host:port/vhost(?parameter=value). Where

- **username** is the username used for authentication to the broker (string)
- **password** is the password used for authentication to the broker (string)
- **host** is a comma seperated lists of brokers to connect to (string)
- **port** is the port used when connect to a broker (int)
- **vhost** is the virtual host to use on the broker (string)
- **parameters** is a query string like seperated list of parameters (string). Supported parameters are the properties in the RawRabbitConfiguration object, such as requestTimeout, persistentDeliveryMode etc.

The ConnectionStringParser can be used to create a configuration object

```
var connectionString = ConfigurationManager.ConnectionStrings["RabbitMq"];
var config = ConnectionStringParser.Parse(connectionString.ConnectionString);
var client = BusClientFactory.CreateDefault(config);
```

Localhost

```
<connectionStrings>
    <add name="RawRabbit" connectionString="guest:guest@localhost:5672/?
↪requestTimeout=10"/>
</connectionStrings>
```

Multiple hosts

Multiple hosts can specified by using a comma-seperated list.

```
<connectionStrings>
    <add name="RawRabbit" connectionString="admin:admin@host1.production,host2.
↪production:5672/"/>
</connectionStrings>
```

CHAPTER 4

Extending RawRabbit

RawRabbit provides a solid foundation for reliable request/reply and publish/subscribe operations. In addition to this, RawRabbit.Extensions can be used to write extensions to the client, making it possilbe to customize the client for any specific needs. The extension framework exposes a method for resolving registered RawRabbit internal services.

Installation

Install the latest version of RawRabbit.Extensions from NuGet.

```
PM> Install-Package RawRabbit.Extensions
```

The Extendable Bus Client

The ExtendableBusClient is an super class of the normal bus client, that exposes the method GetService<TService> (which is just a wrapper around Microsoft.Extensions.DependencyInjection.IServiceProvider). This method allows you to resolve the registered services that RawRabbit uses. This way, if you for example has a custom IContextProvider that you need to get a hold of, it's just a call away.

Extension boiler plait

```
public static class RawRabbitExtensionExample
{
    public static void DoStuff<TContext>(this IBusClient<TContext> client)
        where TContext : IMessageContext
    {
        var extended = (client as ExtendableBusClient<TMessageContext>);
        if (extended == null)
        {
            //TODO: nice error handling
            throw new InvalidOperationException("");
        }
        var channel = extended.GetService<IChannelFactory>().CreateChannel();
        // resolve stuff, make calls...
    }
}
```

List of extensions

- The BulkGet extension can be used to fetch multiple messages from multiple queues and `ACK`/`NACK` them in bulk.
- The Message Sequence Extension can be used to get a RPC like behaviour, but allows for multiple subscribers to act on the message
- The Update Topology Extension can be used to change topologic features.

CHAPTER 5

Bulk-fetching messages

There are times where it is easier to fetch a bunch of messages and process them in a bulk operation, rather than having an active subscriber that processes the messages as they come. This is not part of the core functionality of `RawRabbit`, but exists as a client extension from the `RawRabbit.Extensions` package.

Getting started with the extensions are easy. Create an bus client using the `RawRabbitFactory.GetExtendableClient()` method. *That's it - you're ready to bulk fetch!*

```
var bulk = client.GetMessages(cfg => cfg
    .ForMessage<BasicMessage>(msg => msg
        .FromQueues("first_queue", "second_queue")
        .WithBatchSize(4))
    .ForMessage<SimpleMessage>(msg => msg
        .FromQueues("another_queue")
        .GetAll()
        .WithNoAck()
    ));
```

The fluent builder lets specify what message type you are interested in retrieving, from what queues and how large the batch should be. If you want to get all messages, simple use `GetAll()` and it will empty the queues.

The result contains method for getting messages by type. You can decide for each message if you want to `Ack` it, `Nack` it or put it back in the queue again.

```
var basics = bulk.GetMessages<BasicMessage>()
foreach (var message in basics)
{
    if (CanBeProcessed(message))
    {
        // do stuff
        message.Ack();
    }
    else
    {
        message.Nack();
    }
}
```

If you feel like performing `Ack`/`Nack` the entire bulk, that's fine too

```
bulk.AckAll();
```

Learn more and try it out yourself by running the `BulkGetTests.cs`

CHAPTER 6

Update Topology

Topology features such as queues and exchanges cannot be updated in RabbitMq. However, sometimes it can be desired to change type, durability or other configuration aspects. This can be done with the `UpdateTopology` extension. It removes topology features and re-declares them based on configuration. The extension is available through `RawRabbit.Extensions` that can be installed via the NuGet console

```
PM> Install-Package RawRabbit.Extensions
```

Exchange updates

Updating an exchanges requires two things, the name of the exchange to update and the new desired configuration. Changing the type and durability of exchange `my_exchange` can be done with a few lines of code.

```
await client.UpdateTopologyAsync(t => t
    .ForExchange("my_exchange")
    .UseConfiguration(e => e
        .WithType(ExchangeType.Topic)
        .WithDurability(false))
);
```

The name of the exchange can also be extracted by the message type and the registered `INamingConvention`

```
await client.UpdateTopologyAsync(c => c
    .ExchangeForMessage<BasicMessage>()
    .UseConfiguration(e => e.WithType(ExchangeType.Topic)));
```

Values that are not provided in the configuration builder will default to the values of the `GeneralExchangeConfiguration` on the registered `RawRabbitConfiguration`. If the general exchange configuration has changed and a solution wide update is desired, the `UseConventionForExchange<TMessage>` method can be used

```
var result = await client.UpdateTopologyAsync(c => c
    .UseConventionForExchange<FirstMessage>()
);
```

Change multiple exchanges

The different signatures can be combined in a number of ways to update exchanges. If multiple update configurations are defined for the same exchange, only the latest one will be used.

```
await client.UpdateTopologyAsync(c => c
    .ForExchange("my_exchange")
    .UseConfiguration(x => x.WithAutoDelete())
    .ExchangeForMessage<BasicMessage>()
    .UseConfiguration(x => x.WithType(ExchangeType.Direct))
    .ExchangeForMessage<SimpleMessage>()
    .UseConventions<BasicMessage>()
    .UseConventionForExchange<FirstMessage>()
    .UseConventionForExchange<SecondMessage>()
    .UseConventionForExchange<ThirdMessage>()
);
```

Downtime

Updating an exchange consists of three steps

1. Deleting exchange
2. Re-declare exchange
3. Re-add existing queue bindings

It is not until all queue bindings have been re-added to an exchange that everything works as expected. The extension method returns an result object that contains information about what bindings that has been re-added and the execution time.

```
var result = await client.UpdateTopologyAsync(t => t
    .ForExchange(exchangeName)
    .UseConfiguration(e => e
        .WithType(ExchangeType.Topic)
        .WithDurability(false))
);

ExchangeConfiguration exchangeConfig = result.Exchanges[0].Exchange;
TimeSpan executionTime = result.Exchanges[0].ExecutionTime;
List<Binding> bindings = result.Exchanges[0].Bindings;
```

Binding Key Transformer

In addition to be able to re-define features of the exchange, the binding key can be updated with the optional argument `bindingKeyTransformer`. This can be useful when adding or removing wildcard routing while changing exchange type from one that supports wildcard and one that does not.

```
await currentClient.UpdateTopologyAsync(c => c
    .ExchangeForMessage<BasicMessage>()
    .UseConfiguration(
        exchange => exchange.WithType(ExchangeType.Direct),
        bindingKey => bindingKey.Replace(".*", string.Empty))
);
```

Queue updates

There are currently no support for updating queues.

CHAPTER 7

Message Sequence

In many scenarios, it is considered good practice to have an event-driven architecture where message streams of subsequent publish and subscribe moves the business transactions forward. However, there are scenarios where this is not an option. One example is handling web requestes, where the caller synchronously waits for a response.

Alternative to RPC

Consider a user login scenario that is handled with `UserLoginRequest` and `UserLoginResponse`.

```
// normal rpc response
client.RespondAsync<UserLoginRequest, UserLoginResponse>(async (request, context) =>
{
    var result = await Authenticate();
    return new UserLoginResponse {Token = result};
});

// normal rpc request
var respons = await client.RequestAsync<UserLoginRequest, UserLoginResponse>();
```

There are a few drawbacks of using this pattern. The way `RPC` is implemented with a private response queue, alternativly a direct-rpc queue, makes the calls private between the requester and responder. This is where the `MessageSequence` extension can be useful.

```
// normal subscribe
client.SubscribeAsync<UserLoginRequest>(async (msg, context) =>
{
    var result = await Authenticate();
    await client.PublishAsync(new UserLoginResponse { Token = result}, context.
↪GlobalMessageId);
});

// equivalent message sequence
var sequence = _client.ExecuteSequence(c => c
    .PublishAsync<UserLoginRequest>()
    .Complete<UserLoginResponse>()
);
```

The return object is a `MessageSequence<TComplete>` where `<TComplete>` is the generic type of the `.Complete<TComplete>` call. The sequence has a `Task<TComplete>` that completes as the `UserLoginResponse` is published. The major difference is that the message sequence rely on the message context's `GlobalRequestId` to match the response to the request, rather than having a private response queue or correlation id. The recieving end of the `UserLoginRequest` looks like this

One of the benifits is that the message sequence "response" is actually a publish that is published on the exchange according to the registered `INamingConvention`. That means that any other subscribers of the `LoginResponse` can act upon the message.

Multi-message sequence

The `MessageSequence` extension provides methods to act upon multiple events.

```
var chain = _client.ExecuteSequence(c => c
    .PublishAsync<UserLoginAttempted>()
    .When<UserGeograficPosition>((msg, ctx) => ActOnGeograficPosition(msg.Position))
    .When<UserContactDetail>((msg, ctx) => ActOnContactDetails(msg.Details))
    .Complete<UserLoggoedIn>()
);
```

Optional messages in chain

The `When` call has an optional parameter that can be used to mark a step in the sequence as optional, meaning that if a message that corresponds to a step later in the sequence is recieved, it skips that step.

```
var chain = _client.ExecuteSequence(c => c
    .PublishAsync<UserLoginAttempted>()
    .When<UserPasswordIsWeak>(
        (msg, ctx) => PromptChangePassword(),
        (cfg) => cfg.IsOptional())
    .Complete<UserLoggoedIn>()
);
```

Abort sequence premature

The optional parameter for the `When` also have a method to indicate that if the messagee is recieved, it aborts the execution of the sequence. All handlers that are marked as aborting execution is by default optional.

```
var chain = _client.ExecuteSequence(c => c
    .PublishAsync<UserLoginAttempted>()
    .When<UserLoginFailed>(
        (msg, ctx) => PromptChangePassword(),
        (cfg) => cfg.AbortsExecution())
    .Complete<UserLoggoedIn>()
);
```

CHAPTER 8

Publisher Acknowledgements

RabbitMq has support for *Confirms/Publisher Acknowledgements*, meaning that a publisher gets a `basic-ack` when the message has been accepted by all queues (or the broker verified that the message is unroutable). `RawRabbit` uses this feature when performing `PublishAsync<TMessage>` calls. The `Task` returned from publish call is completed once the broker has confirmed the published message.

If the message hasn't been confirmed within a specified amount of time, the task will fault with a `PublishConfirmException`. To change the timeout, change the `PublishConfirmTimeout` property on the configuration object.

```
var config = new RawRabbitConfiguration
{
    PublishConfirmTimeout = TimeSpan.FromMilliseconds(500)
};
var publisher = BusClientFactory.CreateDefault(config);
```

There is a *slight* performance hit using using this feature. If you want to disable it, just register the `NoAckAcknowledger` when instantiating the bus client.

```
var publisher = BusClientFactory.CreateDefault(s =>
    s.AddSingleton<IPublishAcknowledger, NoAckAcknowledger>()
);
```

Avoiding PublishConfirmException

There are a few potential reasons for `PublishConfirmException` being thrown. If the broker is in heavy use and/or the application publishes multiple concurrent publishes on the `PublishConfirmTimeout` should be increased. Another option is to register the `NoAckAcknowledger`.

CHAPTER 9

Message Priority

Priority for specific messages

To be able to leverage the Priority Queue feature in RabbitMq, you first have to indicate that the queue to which you are subscribing to has the `x-max-priority` argument. This can be done by using the optional configuration argument on the `SubscribeAsync` method

```
subscriber.SubscribeAsync<BasicMessage>(async (message, context) =>
{
    // do stuff
}, cfg => cfg
    .WithQueue(q => q.WithArgument(QueueArgument.MaxPriority, 3))
    .WithPrefetchCount(1)
);
```

In this example, the prefetch count is sets to one, since the already prefetched messages would be processed before a not prefetched message with higher priority.

Now that you have a queue that honours the priority property, you can send messages to it with priority set. This is also done with the fluent configuration builder. In fact, with the builder you get access to all `BasicProperties` for a message.

```
publisher.PublishAsync(new BasicMessage
{
    Prop = "I am important"
}, configuration: cfg =>
    cfg.WithProperties(p => p.Priority = 9)
);
```

Setting priority based on message type

Sometime you want more of a policy like approach, like “All messages of type X is important”. This can be achieved by implementing a custom `IBasicPropertiesProvider`. In the method `GetProperties<TMessage>(Action<IBasicProperties> custom )` you have access to the message type and it returns the properties that will be set in all outgoing messages.

CHAPTER 10

Multiple Subscribers

By default, RawRabbit assumes that when a message is published (`PublishAsync<TMessage>()`), all *unique*, subscribers (with matching routing key on corresponding exchange) wants it. All subscribers are considered to be unique, except those who are hosted in applications that have multiple instances connected to the broker. This can happend if applications are deployed to multiple servers and connected to the same RabbitMq host (or clustered hosts).

The reason for this behaviour is that in many cases it is unwanted to perform an operation multiple times.

Default behaviour

Example: Confirmation email

A service subscribes to a message `OrderSent`, the service sends an email to the customer. Even if this service has multiple insanse connected to the same broker, only one email should be sent.

The default behaviour is achieved by creating unique queue names that contains:

- queue name (extracted from naming convention)
- the application name (extracted from executing folder)
- a unique counter of subscriber to a message type (given the instance of the bus client). In order to make the queue names shorter, the counter is emitted for the first subscriber.

Note that the unique counter is per instance of `IBusClient`. It is therefore recommended to wire up the bus client as a singelton in the IoC container. If you use the `BusClientFactory` or register the IoC using the ServiceCollection extension `AddRawRabbit()`, this is done for you.

Custom Behaviour

For some scenarios, the default behaviour is not desired. It can be modified on for each subscriber by setting a *subscription id*, or for the entire client by registering a custom `INamingConvention`.

Example: Clear Cache

A service subscribes to a `NewDataAvailable`, the service should clear its cache when recieving this message. If the service has multiple instance connected to the broker, each instanse should recieve the message and clear the cache.

Specifying Subscriber Id

The solution is to specify a unique subscription id for the service.

```
secondSubscriber.SubscribeAsync<BasicMessage>(async (message, context) =>
{
    //do stuff...
}, cfg => cfg.WithSubscriberId("unique_id"));
```

CHAPTER 11

Requeue with delay

RawRabbit supports requeing of messages with a predefined retry time interval. The feature uses the dead letter exchange in combination with the time to live extension. The idea comes from yuserinterface's blog; a message that should be retried later is published to a "retry" exchange on a queue that has the actual exchange as its dead letter exchange and a time to live that matches the desired timespan. In order to use RetryLater, make sure you use an advanced message context.

Later execution

```
client.SubscribeAsync<BasicMessage>(async (message, context) =>
{
    if (CanNotBeProcessed())
    {
        context.RetryLater(TimeSpan.FromMinutes(5));
        return;
    }
    // five minutes later we're here.
});
```

Error strategy

The advanced context has information about

- Original sent date, that is the DateTime when the message was first published
- Number of retries, that is how many times it has been retried. This is useful for error strategies such as "retry three times, then Nack it all together).

The requeue can also be used as an error strategy.

```
client.SubscribeAsync<BasicMessage>(async (message, context) =>
{
    if (context.RetryInfo.NumberOfRetries > 10)
    {
        throw new Exception($"Unable to handle message '{context.GlobalRequestId}'.");
    }
    // more code here...
});
```

CHAPTER 12

Inner workings

This section contains information about the inner workings of `RawRabbit`. It can be a useful reference guide for users who wants to extend or modify the standard behaviour of the framework.

ChannelFactory

The default implementation of the `IChannelFactory` is aptly named `ChannelFactory`. It has two main methods

- `GetChannelAsync` returns an existing open channel that is reused by other operations in the application.
- `CreateChannelAsync` return an new, open channel that the caller is responsible to close.

Avoiding 'Pipelining' exceptions

It is forbidden to perform multiple synchronous operations on the same channel. Note that synchronous and asynchronous in this section does not refer to Microsoft's `Task` execution, but rahter how the call is handled by the broker. Synchronous operations include declaring queues and exchanges. It is not adviced to use `GetChannelAsync` and perform a synchronious operation, as you may get a `Pipelining of requests forbidden` exception.

Managing channel count

The `ChannelFactory` is configured with the `ChannelFactoryConfiguration` object. The default behaviour is to re-use the same open channel whenever `GetChannelAsync` is called. `MaxChannelCount` states the maximum amout of channels in the channel pool.

Initialize multiple channels

The property `InitialChannelCount` can be used to define the number of channels that will be initialied as the `ChannelFactory` is initialzed.

Dynamic scaling of channel count

It is possible to open and close aditional channels if the workload for the currently open channels are above the specified threshold `WorkThreshold`. Note that `EnableScaleUp` and/or `EnableScaleDown` needs to be set to `true` to have scaling enabled. `ScaleInterval` defines the interval for checking if scaling should be performed. If scaling down is enable, the `GracefulCloseInterval` is used to know how long to wait before closing the channel. It is recommended to let the graceful close interval be a couple of minutes to make sure that the channel is not in used in other classes.

Alternative implementations

The `ThreadBasedChannelFactory` uses a `ThreadLocal<IModel>` property to make sure that channels are only used in one thread.

ConsumerFactory

It is the consumer factory's responsibility to wire up and return an `IRawConsumer`. The `IRawConsumer` has to implementations, `EventingRawConsumer` (default) that inherits from `EventingBasicConsumer` and `QueueingRawConsumer` that inherits from `QueueingBasicConsumer`.

TopologyProvider

The `TopologyProvider` has async methods for creating topology features, such as queues and exchanges. In order to prevent pipelinging exception, it uses it's own private channel that is disposes two seconds after last usage. It keeps a list of queues and exchanges that is has declared, so that if a `DeclareQueueAsync` is called for a queue recently declared, it returns without doing a roundtrip to the broker.

CHAPTER 13

RabbitMq features

Lazy Queues

As of `3.6.0` RabbitMq supports Lazy Queues. To configure a specific queue as Lazy, simply use the optional configuration argument and set `QueueMode` to `lazy`.

```
subscriber.SubscribeAsync<BasicMessage>((message, context) =>
//do stuff...
, cfg => cfg
    .WithQueue(q => q
        .WithArgument(QueueArgument.QueueMode, "lazy"))
);
```

CHAPTER 14

Error Handling

The error handling pipeline for `RawRabbit` is contained in the `IErrorHandlingStrategy`. It is granular in the sense that different strategies can be employed depending on messaging pattern. All methods in the `DefaultStrategy` are marked as `virtual` and can easierly be overriden in derived classes.

Publish/Subscribe

There is no error handling in the *publish phase*, as there are only a few things that can go wrong here, and exceptions thrown here would most probably need to be resolved (like Topology missmatch).

Any unhandled exception in an subscriber results in the message being published in the default error exchange, together with the exception and other useful metadata.

The default error exchange

The default error exchange name is resolved from the registered `INamingConventions`. By default, no queues are bound to this exchange, and the message will be dropped by the message broker.

To consume messages from the default error queue, setup a consumer for `HandlerExceptionMessage`

```
client.SubscribeAsync<HandlerExceptionMessage>((message, context) =>
{
    var originalMsg = message.Message;
    var originalContext = context;
    var unhandled = message.Exception;
    return HandleAsync(originalMsg, originalContext, unhandled);
}, c => c
    .WithExchange(e => e.WithName(conventions.ErrorExchangeNamingConvention()))
    .WithQueue(q => q.WithArgument(QueueArgument.MessageTtl, 1000))
    .WithRoutingKey("#"));
```

The routing key `#` secures that all unhandled exceptions are recieved in the message handler. However, the message is published with its original routing key, so it is possible to change the routing key to `SendOrderRequest` or any other message that exists in the solution.

It is optional to use the Queue Time To Live attribute and it might be adjusted for different queues depending on the importance of the message.

Messages can also be fetch in a more batch like behaviour by using the Bulk Get Extension.

Request/Respond

Exceptions thrown in the responder message handler is by default caught and sent back to the requester where it is re-thrown. The re-thrown exception is then again caught by `OnResponseRecievedException`, which does nothing be default. Since the request/respond pattern is synchronious. The behaviour could easerly be change to send the message to the default exchange, but remember that the caller is waiting for a task to finish, otherwise the application itself will stall.

CHAPTER 15

Logging

RawRabbit comes with a console logger, which makes sense when playing around in a console app. However, you probably want to use the same logger as you use in the rest of the project. This can be done by downloading RawRabbit.Logger.Serilog, RawRabbit.Logger.NLog, RawRabbit.Logger.Log4Net or implement your own custom logger. Create a logger is fairly easy, it is a matter of implementing ILogger and ILoggerFactory.

The logger is provided to RawRabbit though the registered ILoggerFactory, so it is enough to register the desired factory to use it in all internal classes

```
RawRabbitFactory.GetDefaultBusClient(
            ioc => ioc.AddSingleton<ILoggerFactory, RawRabbit.Logging.Serilog.
↪LoggerFactory>()
);
```

Similarly for vNext apps

```
collection.AddRawRabbit(
    custom: ioc => ioc.AddSingleton<ILoggerFactory, RawRabbit.Logging.Serilog.
↪LoggerFactory>()
)
```

CHAPTER 16

Attribute based configuration

RawRabbit has support for attribute based configuration in the NuGet package RawRabbit.Attributes.

Setting up the client

In order to get the client to scan messages for attributes, register AttributeConfigEvaluator as the IConfigurationEvaluator

```
var client = BusClientFactory.CreateDefault(ioc => ioc
    .AddSingleton<IConfigurationEvaluator, AttributeConfigEvaluator>()
);
```

Configure Messages

There are different attributes that configure different configuration aspects: QueueAttribute, ExchangeAttribute and RoutingAttribute. Note that for the Request/Respond pattern only the attributes of the request message type is scanned.

```
[Queue(Name = "my_queue", MessageTtl = 300, DeadLeterExchange = "dlx", Durable =␣
↪false)]
[Exchange(Name = "my_topic", Type = ExchangeType.Topic)]
[Routing(RoutingKey = "my_key", NoAck = true, PrefetchCount = 50)]
private class AttributedMessage
{
    public string Property { get; set; }
}
```

Override with custom configuration

The `AttributeConfigEvaluator` looks for configuration attributes and fallback to the default `ConfigurationEvaluator`. It also honors the custom configuration provided in the optional configuraiton argument.

```
client.SubscribeAsync<AttributedMessage>((message, context) =>
{
    tcs.TrySetResult(message);
    return Task.FromResult(true);
}, c => c.WithRoutingKey("overridden"));
```

CHAPTER 17

Client upgrade

1.9.0

In release 1.9.0, the default message routing behaviour was changed so that any published message gets its GlobalMessageId appended to the routing key. A message that previously was published with routingkey foo, will use foo.870A9C90-CDEC-4D8D-870B-50BA121BD88F. This is used in the Message Sequence Extension to route only relevant messages to the different clients. Subscribers to messages use a wildcard routing foo.# and the messages will be delivered to the consumer. Previously, the Direct exchange type was the default type in RawRabbit, but wildcard routing is not supported there, which is why the new default is Topic.

When a consumer is set up, RawRabbit verifies that the exchange to which it want to bind the consumer to exists. If the exchange is exists but it is declared with a different type than the one that exists, an exception will be thrown.

Using existring configuration

The old configuration can be used by registering a "legacy" (pre 1.9.0) configuration

```
var cfg = RawRabbitConfiguration.Local.AsLegacy();
var client = RawRabbitFactory.GetExtendableClient(ioc => ioc.AddSingleton(s => cfg));
```

The AsLegacy extension sets the configuration value RouteWithGlobalId to false and resets the default exchange type to Direct.

Upgrading from < 1.9.0

If you want to use the new configuration on existing environments, the Update Topology Extension can be used to re-declare and re-bind queues with minimal downtime:

```
var client = RawRabbitFactory.GetExtendableClient();
await client.UpdateTopologyAsync(c => c
    .ExchangeForMessage<BasicMessage>()
    .UseConfiguration(
        exchange => exchange.WithType(ExchangeType.Topic),
        bindingKey => $"{bindingKey}.#")
    );
```

By adding the `#` wildcard, the consumer matches zero or more words in the routing key, making it compatible with clients that use the old configuration.

1.9.5

With `1.9.5`, the life time management has been looked over thoroughly. Previously, the base client implemented the `IDisposable` interface, that in turn disposed all of its own resoruces, all the way down the `IChannelFactory`. This is wanted behaviour in applications where the busclient is registered as a single instance with the same life time as the applications. However, in web applications, we might want to build the bus client for each request, customizing dependencies based on the `HttpContext`. Disposing everything in that scenario will lead to a unneccesary performance hit.

To address this, the `IDisposable` interface was removed from the base client, and added to derived clientes in the `Disposable` namespace. This is the client that is returned from the `BusClientFactory` (and the `RawRabbitFactory` for extendable bus clients).

Updraging to 1.9.5

There should be no major problems with this update. If you are using the factory classes for creating bus clients and somehow misses any references in your class, make sure to use

- `RawRabbit.vNext.Disposable.IBusClient` where your previously used `RawRabbit.IBusClient`
- `RawRabbit.Extensions.Disposable.IBusClient` where your previously used `RawRabbit.IBusClient` for extensions.

CHAPTER 18

Contributing Guidelines

You are more than welcome to contribute to `RawRabbit`. Here are some guidelines for the process.

Create issue

With a few exceptions, every commits should be connected to an issue. That means that if you've found a bug or implemented a feature, it should be reported in the issue section.

Write code

Write as beautiful code as possible! `RawRabbit` is indented with tabs and not spaces.

Commit Code

Make sure that all commits start with `(#issue-number)`, like `(#19) Invoke message handlers in sync manner`. This way, the commits will appear in the issue and is easier found from the console `git log --grep #19`.

Follow the official guide lines. In short, the seven rules of a great git commit message should be honored:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters

7. Use the body to explain what and why vs. how

Create Pull Request

Once the feature is developed, create a pull to `stable`.