Tapeti Documentation

Mark van Renswoude

Apr 26, 2023

Table of contents

CHAPTER 1

Introduction

‘Small to medium-sized and classified as “Least Concern” by the IUCN.’
Wikipedia*[0]

Tapeti is a wrapper for the RabbitMQ .NET Client designed for long-running microservices. It’s main goal is to minimize the amount of messaging code required, and instead focus on the higher-level flow.

Tapeti is built using .NET Standard 2.0 and mostly tested using .NET 4.7.

1.1 Key features

- Consumers are declared using MVC-style controllers and are registered automatically based on annotations
- Publishing requires only the message class, no transport details such as exchange and routing key
- *Flow extension* (stateful request - response handling with support for parallel requests)
- No inheritance required
- Graceful recovery in case of connection issues, and in contrast to most libraries not designed for services, during startup as well
- Extensible using middleware

1.2 What it is not

Tapeti is not a general purpose RabbitMQ client. Although some behaviour can be overridden by implementing various interfaces, it enforces it’s style of messaging and assumes everyone on the bus speaks the same language.

There is no support for TLS connections, nor are there any plans to support it. The author is of the opinion the message bus should be considered an internal, highly available, service and recommends self-hosted REST API’s behind an SSL proxy for communicating over public interfaces.

[0] Before it was marked as “Endangered” in 2019 and the pun no longer works.

CHAPTER 2

Getting started

This guide is a step by step introduction. If you want to know more about how Tapeti works, for example how it determines the exchange and routing keys, see *In-depth*.

2.1 Install packages

I'll assume you are familiar with installing NuGet.org packages into your project.

Find and install the *Tapeti* package. This will also install *Tapeti.Annotations*, which contains the various attributes.

You will need an integration package as well for your IoC (Inversion of Control) container of choice. Various containers are supported by default:

- SimpleInjector (Tapeti.SimpleInjector)
- Autofac (Tapeti.Autofac)
- Castle Windsor (Tapeti.CastleWindsor)
- Ninject (Tapeti.Ninject)
- Unity (Tapeti.UnityContainer)

SimpleInjector is used in all examples. The "01-PublishSubscribe" example included in the source shows how the other integration packages can be used.

Note: If you need support for your favourite library, implement *IDependencyContainer* using the existing packages' source as a reference and replace *SimpleInjectorDependencyResolver* with your class name in the example code.

2.2 Configuring Tapeti

First create an instance of TapetiConfig, tell it which controllers to register and have it gather all the information by calling Build. Then pass the result to a TapetiConnection.

```
using SimpleInjector;
using Tapeti;

internal class Program
{
    private static void Main()
    {
        var container = new Container();
        var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
            .RegisterAllControllers()
            .Build();

        using (var connection = new TapetiConnection(config)
        {
            Params = new TapetiConnectionParams
            {
                Host = "localhost"
            }
        })
        {
            connection.SubscribeSync();

            // Start service
        }
    }
}
```

Note: RegisterAllControllers without parameters searches the entry assembly. Pass an Assembly parameter to register other or additional controllers. You can call RegisterAllControllers multiple times with different assemblies.

Caution: Tapeti attempts to register it's default implementations in the IoC container during configuration, as well as when starting the connection (to register IPublisher). If your container is immutable after the initial configuration, like SimpleInjector is, make sure that you run the Tapeti configuration before requesting any instances from the container.

2.3 Defining a message

A message is simply a plain object which can be serialized using Json.NET.

```
public class RabbitEscapedMessage
{
    public string Name { get; set; }
    public string LastKnownHutch { get; set; }
}
```

2.4 Creating a message controller

To handle messages you need what Tapeti refers to as a "message controller". It is similar to an ASP.NET controller if you're familiar with those, but it handles RabbitMQ messages instead of HTTP requests.

All you need to do is create a new class and annotate it with the MessageController attribute and a queue attribute. The name and folder of the class is not important to Tapeti, though you might want to agree on a standard in your team.

The queue attribute can be either *DynamicQueue* or *DurableQueue*. The attribute can be set for the entire controller (which is considered the default scenario) or specified / overridden per message handler.

DynamicQueue will create a queue with a name generated by RabbitMQ which is automatically deleted when your service stops. Bindings will be added for the messages handled by the controller. You will typically use dynamic queues for scenarios where handling the message is only relevant while the service is running (for example, updating a service's cache or performing live queries).

DurableQueue requires a queue name as the parameter. By default, the queue is assumed to be present already and Tapeti will throw an exception if it does not. If you want Tapeti to create and update the durable queues as well, see *Durable queues* in *In-depth*.

```
[MessageController]
[DynamicQueue("monitoring")]
public class MonitoringController
{
}
```

Note: Notice the parameter to DynamicQueue. This defines the prefix. If specified, the queue name will begin with the supplied value, followed by a unique identifier, so it can be more easily recognized in the RabbitMQ management interface.

2.5 Handling incoming messages

Any public method in a message controller is considered a message handler. There are a few requirements which are enforced by Tapeti. Below are the default requirements, although some extension packages (like the *Flow extension*) add their own or alter these requirements.

- The first parameter must be the message class.
- The return type can be void, Task, Task<message class> or a message class.

The name of the method is not important to Tapeti. Any parameter other than the first will be resolved in two ways:

1. Registered middleware can alter the behaviour of parameters. Tapeti includes one by default for Cancellation-Token parameters, see *Parameter binding* in *In-depth*.
2. Any remaining parameters are resolved using the IoC container, although it is considered best practice to use the constructor for dependency injection instead.

A new controller is instantiated for each message, so it is safe to use public or private fields to store state while handling the message. Just don't expect it to be there for the next message. If you need this behaviour, take a look at the *Flow extension*!

```
[MessageController]
[DynamicQueue]
public class MonitoringController
```

(continues on next page)

(continued from previous page)

```
{
    public void LogEscape(RabbitEscapedMessage message)
    {
        Logger.Warning($"This is a beige alert. {message.Name} has escaped." +
                       $"It was last seen in {message.LastKnownHutch}.");
    }
}
```

Note: If you're doing anything asynchronous in the message handler, make it async as well! Simply change the return type to "Task" or "async Task".

If the method returns a message object, that object is published as if it was a reply to the incoming message, maintaining the correlationId and respecting the replyTo header. See *In-depth* for request-response requirements.

2.6 Publishing messages

To send a message, get a reference to IPublisher using dependency injection and call the Publish method. For example, to broadcast another message from a message handler:

```
public class LogMessage
{
    public string Level { get; set; }
    public string Description { get; set; }
}


[MessageController]
[DynamicQueue]
public class MonitoringController
{
    private readonly IPublisher publisher;

    public MonitoringController(IPublisher publisher)
    {
        this.publisher = publisher;
    }

    public async Task LogEscape(RabbitEscapedMessage message)
    {
        await publisher.Publish(new LogMessage
        {
            Level = "Beige",
            Description = $"{message.Name} has escaped." +
                          $"It was last seen in {message.LastKnownHutch}."
        });
    }
}
```

2.7 Connection parameters

If you don't want to use the default connection parameters, which is probably a good idea in a production environment, you can manually specify the properties for TapetiConnectionParams or get them from your configuration of choice. Tapeti provides with two helpers.

2.7.1 App.config / Web.config

The TapetiAppSettingsConnectionParams class can be used to load the connection parameters from the AppSettings:

```
using (var connection = new TapetiConnection(config)
{
    Params = new TapetiAppSettingsConnectionParams()
})
```

The constructor accepts a prefix parameter, which defaults to "rabbitmq:". You can then specify the values in the appSettings block of your App.config or Web.config. Any omitted parameters will use the default value.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="rabbitmq:hostname" value="localhost" />
    <add key="rabbitmq:port" value="5672" />
    <add key="rabbitmq:virtualhost" value="/" />
    <add key="rabbitmq:username" value="guest" />
    <add key="rabbitmq:password" value="guest" />
    <add key="rabbitmq:prefetchcount" value="50" />
    <add key="rabbitmq:managementport" value="15672" />
    <add key="rabbitmq:clientproperty:application" value="Example" />
  </appSettings>
</configuration>
```

The last entry is special: any setting which starts with "clientproperty:", after the configured prefix, will be added to the ClientProperties set. These properties are visible in the RabbitMQ Management interface and can be used to identify the connection.

2.7.2 ConnectionString

Tapeti also includes a helper which can parse a connection string style value which is mainly for compatibility with EasyNetQ. It made porting our applications slightly easier. EasyNetQ is a very capable library which includes high- and low-level wrappers for the RabbitMQ Client as well as a Management API client, and is worth checking out if you have a use case that is not suited to Tapeti.

To parse a connection string, use the ConnectionStringParser.Parse method. You can of course still load the value from the AppSettings easily:

```
using (var connection = new TapetiConnection(config)
{
    Params = Tapeti.Helpers.ConnectionStringParser.Parse(
      ConfigurationManager.AppSettings["RabbitMQ.ConnectionString"])
})
```

An example connection string:

```
host=localhost;username=guest;password=prefetchcount=5
```

Supported keys are:

- hostname
- port
- virtualhost
- username
- password
- prefetchcount
- managementport

Any keys in the connection string which are not supported will be silently ignored.

CHAPTER 3

Compatibility

3.1 ASP.NET Core

When integrating Tapeti into an ASP.NET Core service, depending on your naming conventions you may run into an issue where ASP.NET tries to register all your messaging controllers as API controllers. This is because by default any class ending in “Controller” will be picked up by ASP.NET.

You can rename your Tapeti controller classes as long as there are no persisted Tapeti Flows for that controller.

Alternatively, you can filter the classes that ASP.NET will register by using a custom ControllerFeatureProvider.

3.1.1 Whitelist ASP.NET controllers

If all your ASP.NET controllers use the [ApiController] attribute, you can simply whitelist classes which are annotated with that attribute, ensuring the class name is not relevant to ASP.NET:

```
public class APIControllersOnlyFeatureProvider : ControllerFeatureProvider
{
    protected override bool IsController(TypeInfo typeInfo)
    {
        return typeInfo.IsClass && typeInfo.IsPublic && !typeInfo.IsAbstract &&
            typeInfo.GetCustomAttribute<ApiControllerAttribute>() != null;
    }
}
```

3.1.2 Blacklist Tapeti controllers

If instead you want the default ASP.NET behaviour but only exclude Tapeti messaging controllers, you can decorate the default feature provider as follows:

```
public class ExcludeMessagingControllerFeatureProvider : ControllerFeatureProvider
{
    protected override bool IsController(TypeInfo typeInfo)
    {
        return base.IsController(typeInfo) &&
            typeInfo.GetCustomAttribute<MessageControllerAttribute>() == null;
    }
}
```

3.1.3 Replace feature provider

In either case, replace the default ControllerFeatureProvider in the ConfigureServices method of your startup class:

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddControllers()
        .ConfigureApplicationPartManager(manager =>
        {
            var controllerFeatureProvider = manager.FeatureProviders
                .FirstOrDefault(provider => provider is ControllerFeatureProvider);

            if (controllerFeatureProvider != null)
                manager.FeatureProviders.Remove(controllerFeatureProvider);

            manager.FeatureProviders.Add(new APIControllersOnlyFeatureProvider());
            // or: manager.FeatureProviders.Add(new␣
↪ExcludeMessagingControllerFeatureProvider());
        });
}
```

CHAPTER 4

In-depth

4.1 Messages

As described in the Getting started guide, a message is a plain object which can be serialized using Json.NET.

When communicating between services it is considered best practice to define messages in separate class library assemblies which can be referenced in other services. This establishes a public interface between services and components without binding to the implementation.

4.2 Parameter binding

Tapeti will bind the parameters of message handler methods using the registered binding middleware.

Although stated in the Getting started guide that the first parameter is always assumed to be the message class, this is in fact handled by one of the default binding middleware implementations instead of being hardcoded in Tapeti. All of the default implementations play nice and will only apply to parameters not already bound by other middleware, making it easy to extend or change the default behaviour if desired.

In addition to the message class parameter, two additional default implementations are included:

4.2.1 CancellationToken

Similar to ASP.NET, Tapeti will bind parameters of type CancellationToken to a token which is cancelled when the connection to the RabbitMQ server is closed.

Note: This does not indicate whether the connection was closed by the application or lost unexpectedly, either scenario will cancel the token. This is by design, as any message in-flight will be put back on the queue and redelivered anyways.

Internally this CancellationToken is called ConnectionClosed, but any name can be used. For example:

```
public async Task<CountResponseMessage> CountRabbits(CountRequestMessage message,
    CancellationToken cancellationToken)
{
    var count = await rabbitRepository.Count(cancellationToken);

    return new CountRabbitsResponseMessage
    {
        Count = count
    };
}
```

4.2.2 Dependency injection

Any parameter not bound by any other means will be resolved using the IoC container which is passed to the Tapeti-Connection.

Note: It is considered best practice to use the constructor for dependency injection instead.

4.3 Enums

Special care must be taken when using enums in messages. For example, you have several services consuming a message containing an enum field. Some services will have logic which depends on a specific value, others will not use that specific field at all.

Then later on, you want to add a new value to the enum. Some services will have to be updated, but the two examples mentioned above do not rely on the new value. As your application grows, it will become unmanageable to keep all services up-to-date.

Tapeti accounts for this scenario by using a custom deserializer for enums. Enums are always serialized to their string representation, and you should never rely on their ordinal values. When deserializing, if the sending service sends out an enum value that is unknown to the consuming service, Tapeti will deserialize it to an out-of-bounds enum value instead of failing immediately.

This effectively means that as long as your code has conditional or switch statements that can handle unknown values, or only perform direct comparisons, the existing logic will run without issues.

In addition, Tapeti does not allow you to pass the value as-is to another message, as the original string representation is lost. If it detects the out-of-bounds value in an enum field of a message being published, it will raise an exception.

However, Tapeti cannot analyze your code and the ways you use the enum field. So if you ignored all advice and used the ordinal value of the enum to store somewhere directly, you are left with the value 0xDEADBEEF, and you will now know why.

4.4 Durable queues

Before version 2.0, and still by default in the newer versions, Tapeti assumes all durable queues are set up with the proper bindings before starting the service.

However, since this is very inconvenient you can enable Tapeti to manage durable queues as well. There are two things to keep in mind when enabling this functionality:

1) The RabbitMQ management plugin must be enabled
2) The queue name must be unique to the service to prevent conflicting updates to the bindings

To enable the automatic creation of durable queues, call EnableDeclareDurableQueues or SetDeclareDurableQueues on the TapetiConfig:

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .EnableDeclareDurableQueues()
    .RegisterAllControllers()
    .Build();
```

The queue will be bound to all message classes for which you have defined a message handler. If the queue already existed and contains bindings which are no longer valid, those bindings will be removed. Note however that if there are still messages of that type in the queue they will be consumed and cause an exception. To keep the queue backwards compatible, see the next section on migrating durable queues.

4.5 Migrating durable queues

Note: This section assumes you are using EnableDeclareDurableQueues.

As your service evolves so can your message handlers. Perhaps a message no longer needs to handled, or you want to split them into another queue.

If you remove a message handler the binding will also be removed from the queue, but there may still be messages of that type in the queue. Since these have nowhere to go, they will cause an error and be lost.

Instead of removing the message handler you can mark it with the standard .NET `[Obsolete]` attribute:

```
[MessageController]
[DurableQueue("monitoring")]
public class ObsoleteMonitoringController
{
    [Obsolete]
    public void HandleEscapeMessage(RabbitEscapedMessage message)
    {
        // Handle the message like before, perform the necessary migration,
        // or simply ignore it if you no longer need it.
    }
}
```

Messages will still be consumed from the queue as long as it exists, but the routing key binding will removed so no new messages of that type will be delivered.

The `[Obsolete]` attribute can also be applied to the entire controller to mark all message handlers it contains as obsolete.

If all message handlers bound to a durable queue are marked as obsolete, including other controllers bound to the same durable queue, the queue is a candidate for removal. During startup, if the queue is empty it will be deleted. This action is logged to the registered ILogger.

If there are still messages in the queue it's pending removal will be logged but the consumers will run as normal to empty the queue. The queue will then remain until it is checked again when the application is restarted.

4.6 Request - response

Messages can be annotated with the Request attribute to indicate that they require a response. For example:

```
[Request(Response = typeof(BunnyCountResponseMessage))]
public class BunnyCountRequestMessage
{
    public string ColorFilter { get; set; }
}

public class BunnyCountResponseMessage
{
    public int Count { get; set; }
}
```

Message handlers processing the BunnyCountRequestMessage *must* respond with a BunnyCountResponseMessage, either directly or at the end of a Flow when using the *Flow extension*.

```
[MessageController]
[DurableQueue("hutch")]
public class HutchController
{
    private IBunnyRepository repository;

    public HutchController(IBunnyRepository repository)
    {
        this.repository = repository;
    }

    public async Task<BunnyCountResponseMessage>␣
↪HandleCountRequest(BunnyCountRequestMessage message)
    {
        return new BunnyCountResponseMessage
        {
            Count = await repository.Count(message.ColorFilter)
        };
    }
}
```

Tapeti will throw an exception if a request message is published but there is no route for it. Tapeti will also throw an exception if you do not return the correct response class. This ensures consistent flow across services.

If you simply want to broadcast an event in response to a message, do not use the return value but instead call IPublisher.Publish in the message handler.

In practise your service may end up with the same message having two versions; one where a reply is expected and one where it's not. This is not considered a design flaw but a clear contract between services. It is common and recommended for the request message to inherit from the base non-request version, and implement two message handlers that internally perform the same logic.

While designing Tapeti this difference has been defined as *Transfer of responsibility* which is explained below.

4.7 Transfer of responsibility

When working with microservices there will be dependencies between services.

Sometimes the dependency should be on the consumer side, which is the classic publish-subscribe pattern. For example, a reporting service will often listen in on status updates from various other services to compose a combined report. The services producing the events simply broadcast the message without concerning who if anyone is listening.

Sometimes you need another service to handle or query data outside of your responsibility, and the Request - Response mechanism can be used. Tapeti ensures these messages are routed as described above.

The third pattern is what we refer to as “Transfer of responsibility”. You need another service to continue your work, but a response is not required. For example, you have a REST API which receives and validates a request, then sends it to a queue to be handled by a background service.

Messages like these must not be lost, there should always be a queue bound to it to handle the message. Tapeti supports the [Mandatory] attribute for these cases and will throw an exception if there is no queue bound to receive the message:

```
[Mandatory]
public class SomeoneHandleMeMessage
{
}
```

4.8 Routing keys

The routing key is determined by converting CamelCase to dot-separated lowercase, leaving out “Message” at the end if it is present. In the example below, the routing key will be “something.happened”:

```
public class SomethingHappenedMessage
{
    public string Description { get; set; }
}
```

This behaviour is implemented using the IRoutingKeyStrategy interface. For more information about changing this, see *Overriding default behaviour*

Note: As you can see the namespace in which the message class is declared is not used in the routing key. This means you should not use the same class name twice as it may result in conflicts. The exchange strategy described below helps in differentiating between the messages, but to avoid any confusion it is still best practice to use unambiguous message class names or use another routing key strategy.

4.9 Exchanges

The exchange on which the message is published and consumers are expected to bind to is determined by the first part of the namespace, skipping “Messaging” if it is present. In the example below, the exchange will be “Example”:

```
namespace Messaging.Example.Events
{
    public class SomethingHappenedMessage
    {
        public string Description { get; set; }
    }
}
```

This behaviour is implemented using the IExchangeStrategy interface. For more information about changing this, see *Overriding default behaviour*

4.10 Overriding default behaviour

Various behaviours of Tapeti are implemented using interfaces which are resolved using the IoC container. Tapeti will attempt to register the default implementations, but these can easily be replaced with your own version. For example:

```
// Nod to jsforcats.com
public class YellItRoutingKeyStrategy : IRoutingKeyStrategy
{
    public string GetRoutingKey(Type messageType)
    {
        return messageType.Name.ToUpper() + "!!!!";
    }
}


container.Register<IRoutingKeyStrategy, YellItRoutingKeyStrategy>();
```

The best place to register your implementation is before calling TapetiConfig.

CHAPTER 5

Validating messages

To validate the contents of messages, Tapeti provides the Tapeti.DataAnnotations package. Once installed and enabled, it verifies each message that is published or consumed using the standard System.ComponentModel.DataAnnotations.

To enable the validation extension, include it in your TapetiConfig:

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .WithDataAnnotations()
    .RegisterAllControllers()
    .Build();
```

5.1 Annotations

All ValidationAttribute derived annotations are supported. For example, use the Required attribute to indicate a field is required:

```
public class RabbitEscapedMessage
{
    [Required]
    public string Name { get; set; }

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

Or the Range attribute to indicate valid ranges:

```
public class RabbitBirthdayMessage
{
    [Range(1, 15, ErrorMessage = "Sorry, we have no birthday cards for ages below {1}␣
↪or above {2}")]
    public int Age { get; set; }
}
```

5.2 Required GUIDs

Using the standard validation attributes it is tricky to get a Guid to be required, as it is a struct which defaults to Guid.Empty. Using Nullable<Guid> may work, but then your business logic will look like it is supposed to be optional.

For this reason, the Tapeti.DataAnnotations.Extensions package can be installed from NuGet into your messaging package. It contains the RequiredGuid attribute which specifically checks for Guid.Empty.

```
public class RabbitBornMessage
{
    [RequiredGuid]
    public Guid RabbitId { get; set; }
}
```

CHAPTER 6

Flow extension

Flow in the context of Tapeti is inspired by what is referred to as a Saga or Conversation in messaging. It enables a controller to communicate with other services, temporarily yielding it's execution while waiting for a response. When the response arrives the controller will resume, retaining the original state of it's public fields.

This process is fully asynchronous, the service initiating the flow can be restarted and the flow will continue when the service is back up (assuming the queues are durable and a persistent flow state store is used).

6.1 Request - response pattern

Tapeti implements the request - response pattern by allowing a message handler method to simply return the response message. Tapeti Flow extends on this concept by allowing the sender of the request to maintain it's state for when the response arrives.

See *In-depth* on defining request - response messages.

6.2 Enabling Tapeti Flow

To enable the use of Tapeti Flow, install the Tapeti.Flow NuGet package and call `WithFlow()` when setting up your TapetiConfig:

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .WithFlow()
    .RegisterAllControllers()
    .Build();
```

6.3 Starting a flow

To start a new flow you need to obtain an IFlowStarter from your IoC container. It has one method in various overloads: `Start`.

Flow requires all methods participating in the flow, including the starting method, to be in the same controller. This allows the state to be stored and restored when the flow continues. The `IFlowStarter.Start` call does not need to be in the controller class.

The controller type is passed as a generic parameter. The first parameter to the Start method is a method selector. This defines which method in the controller is called as soon as the flow is initialised.

```
await flowStart.Start<QueryBunniesController>(c => c.StartFlow);
```

The start method can have any name, but must be annotated with the `[Start]` attribute. This ensures it is not recognized as a message handler. The start method and any further continuation methods must return either Task<IYieldPoint> (for asynchronous methods) or simply IYieldPoint (for synchronous methods).

```
[MessageController]
[DynamicQueue]
public class QueryBunniesController
{
    public DateTime RequestStart { get; set; }

    [Start]
    public IYieldPoint StartFlow()
    {
        RequestStart = DateTime.UtcNow();
    }
}
```

Often you'll want to pass some initial information to the flow. The Start method allows one parameter. If you need more information, bundle it in a class or struct.

```
await flowStart.Start<QueryBunniesController>(c => c.StartFlow, "pink");

[MessageController]
[DynamicQueue]
public class QueryBunniesController
{
    public DateTime RequestStart { get; set; }

    [Start]
    public IYieldPoint StartFlow(string colorFilter)
    {
        RequestStart = DateTime.UtcNow();
    }
}
```

Note: Every time a flow is started or continued a new instance of the controller is created. All public fields in the controller are considered part of the state and will be restored when a response arrives, private and protected fields are not. Public fields must be serializable to JSON (using JSON.NET) to retain their value when a flow continues. Try to minimize the amount of state as it is cached in memory until the flow ends.

6.4 Continuing a flow

When starting a flow you're most likely want to start with a request message. Similarly, when continuing a flow you have the option to follow it up with another request and prolong the flow. This behaviour is controlled by the IYieldPoint that must be returned from the start and continuation handlers. To get an IYieldPoint you need to inject the IFlowProvider into your controller.

IFlowProvider has a method `YieldWithRequest` which sends the provided request message and restores the controller when the response arrives, calling the response handler method you pass along to it.

The response handler must be marked with the `[Continuation]` attribute. This ensures it is never called for broadcast messages, only when the response for our specific request arrives. It must also return an IYieldPoint or Task<IYieldPoint> itself.

If the response handler is not asynchronous, use `YieldWithRequestSync` instead, as used in the example below:

```
[MessageController]
[DynamicQueue]
public class QueryBunniesController
{
    private IFlowProvider flowProvider;

    public DateTime RequestStart { get; set; }


    public QueryBunniesController(IFlowProvider flowProvider)
    {
        this.flowProvider = flowProvider;
    }

    [Start]
    public IYieldPoint StartFlow(string colorFilter)
    {
        RequestStart = DateTime.UtcNow();

        var request = new BunnyCountRequestMessage
        {
            ColorFilter = colorFilter
        };

        return flowProvider.YieldWithRequestSync<BunnyCountRequestMessage,␣
↪BunnyCountResponseMessage>
            (request, HandleBunnyCountResponse);
    }


    [Continuation]
    public IYieldPoint HandleBunnyCountResponse(BunnyCountResponseMessage message)
    {
        // Handle the response. The original RequestStart is available here as well.
    }
}
```

You can once again return a `YieldWithRequest`, or end it.

6.5 Ending a flow

To end the flow and dispose of any stored state, return an end yieldpoint:

```
[Continuation]
public IYieldPoint HandleBunnyCountResponse(BunnyCountResponseMessage message)
{
    // Handle the response.

    return flowProvider.End();
}
```

6.6 Flows started by a (request) message

Instead of manually starting a flow, you can also start one in response to an incoming message. You do not need access to the IFlowStarter in that case, simply return an IYieldPoint from a regular message handler:

```
[MessageController]
[DurableQueue("hutch")]
public class HutchController
{
    private IBunnyRepository repository;
    private IFlowProvider flowProvider;

    public string ColorFilter { get; set; }


    public HutchController(IBunnyRepository repository, IFlowProvider flowProvider)
    {
        this.repository = repository;
        this.flowProvider = flowProvider;
    }

    public IYieldPoint HandleCountRequest(BunnyCountRequestMessage message)
    {
        ColorFilter = message.ColorFilter;

        return flowProvider.YieldWithRequestSync<CheckAccessRequestMessage,␣
↪CheckAccessResponseMessage>
          (
              new CheckAccessRequestMessage
              {
                  Username = "hutch"
              },
              HandleCheckAccessResponseMessage
          );
    }


    [Continuation]
    public IYieldPoint HandleCheckAccessResponseMessage(CheckAccessResponseMessage␣
↪message)
    {
        // We must provide a response to our original BunnyCountRequestMessage
```

(continues on next page)

(continued from previous page)

```
        return flowProvider.EndWithResponse(new BunnyCountResponseMessage
        {
            Count = message.HasAccess ? await repository.Count(ColorFilter) : 0
        });
}
```

Note: If the message that started the flow was a request message, you must end the flow with EndWithResponse or you will get an exception. Likewise, if the message was not a request message, you must end the flow with End.

6.7 Parallel requests

When you want to send out more than one request, you could chain them in the response handler for each message. An easier way is to use `YieldWithParallelRequest`. It returns a parallel request builder to which you can add one or more requests to be sent out, each with it's own response handler. In the end, the Yield method of the builder can be used to create a YieldPoint. It also specifies the converge method which is called when all responses have been handled.

An example:

```
public IYieldPoint HandleBirthdayMessage(RabbitBirthdayMessage message)
{
    var sendCardRequest = new SendCardRequestMessage
    {
        RabbitID = message.RabbitID,
        Age = message.Age,
        Style = CardStyles.Funny
    };

    var doctorAppointmentMessage = new DoctorAppointmentRequestMessage
    {
        RabbitID = message.RabbitID,
        Reason = "Yearly checkup"
    };

    return flowProvider.YieldWithParallelRequest()
        .AddRequestSync<SendCardRequestMessage, SendCardResponseMessage>(
          sendCardRequest, HandleCardResponse)

        .AddRequestSync<DoctorAppointmentRequestMessage,␣
↪DoctorAppointmentResponseMessage>(
          doctorAppointmentMessage, HandleDoctorAppointmentResponse)

        .YieldSync(ContinueAfterResponses);
}

[Continuation]
public void HandleCardResponse(SendCardResponseMessage message)
{
    // Handle card response. For example, store the result in a public field
}

[Continuation]
```

(continues on next page)

(continued from previous page)

```
public void HandleDoctorAppointmentResponse(DoctorAppointmentResponseMessage message)
{
    // Handle appointment response. Note that the order of the responses is not␣
↪guaranteed,
    // but the handlers will never run at the same time, so it is safe to access
    // and manipulate the public fields of the controller.
}

private IYieldPoint ContinueAfterResponses()
{
    // Perform further operations on the results stored in the public fields

    // This flow did not start with a request message, so end it normally
    return flowProvider.End();
}
```

A few things to note:

1) The response handlers do not return an IYieldPoint themselves, but void (for AddRequestSync) or Task (for AddRequest). Therefore they can not influence the flow. Instead the converge method as passed to Yield or YieldSync determines how the flow continues. It is called immediately after the last response handler.
2) The converge method must be private, as it is not a valid message handler in itself.
3) You must add at least one request, or specify the NoRequestsBehaviour parameter for Yield/YieldSync explicitly.

Note that you do not have to perform all the operations in one go. You can store the result of `YieldWithParallelRequest` and conditionally call `AddRequest` or `AddRequestSync` as many times as required.

6.8 Adding requests to a parallel flow

As mentioned above, you can not start a new parallel request in the same flow while the current one has not converged yet. This is enforced by the response handlers not returning an IYieldPoint.

You can however add requests to the current parallel request while handling one of the responses. This is equivalent to adding the request to the parallel flow builder initially, and will delay calling the converge method until a response has been received to this new request as well.

To add an additional request, include a second parameter in the continuation method of type IFlowParallelRequest. The continuation method also needs to be async to be able to await the IFlowParallelRequest.AddRequest[Sync] methods. For example:

```
[Continuation]
public async Task HandleDoctorAppointmentResponse(DoctorAppointmentResponseMessage␣
↪appointment,
    IFlowParallelRequest parallelRequest)
{
    // Now that we have the appointment details, we can query the patient data
    await parallelRequest.AddRequestSync<PatientRequestMessage,␣
↪PatientResponseMessage>(
        new PatientRequestMessage
        {
            PatientID = appointment.PatientID
        },
```

(continues on next page)

(continued from previous page)

```
        HandlePatientResponse);
}
```

6.9 Persistent state

By default flow state is only preserved while the service is running. To persist the flow state across restarts and reboots, provide an implementation of IFlowRepository to `WithFlow()`.

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .WithFlow(new MyFlowRepository())
    .RegisterAllControllers()
    .Build();
```

Tapeti.Flow includes an implementation for SQL server you can use as well. First, make sure your database contains a table to store flow state:

```
create table Flow
(
    FlowID uniqueidentifier not null,
    CreationTime datetime2(3) not null,
    StateJson nvarchar(max) null,
    constraint PK_Flow primary key clustered(FlowID)
);
```

Then install the Tapeti.Flow.SQL NuGet package and register the SqlConnectionFlowRepository by passing it to WithFlow, or by using the `WithFlowSqlRepository` extension method before calling `WithFlow`:

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .WithFlowSqlRepository("Server=localhost;Database=TapetiTest;Integrated␣
↪Security=true")
    .WithFlow()
    .RegisterAllControllers()
    .Build();
```

Caution: The controller and method names for response handlers and converge methods are stored in the flow and must be valid when they are loaded again. Keep that in mind if you want to refactor the code; either keep the original class and method temporarily for backwards compatibility, optionally redirecting them internally to the new code, or make sure there are no persisted flows remaining.

CHAPTER 7

Transient requests

The Tapeti.Transient extension provides an RPC mechanism for request - responses.

While the author does not recommend this pattern for services, opting instead for a self-hosted REST API and a discovery mechanism like Consul if an immediate response is required and a queue is not (like when providing an API for a frontend application), it can still be useful in certain situations to use the Tapeti request - response mechanism and be able to wait for the response.

After enabling Tapeti.Transient you can use `ITransientPublisher` to send a request and await it's response, without needing a message controller.

7.1 Enabling transient requests

To enable the transient extension, include it in your TapetiConfig. You must provide a timeout after which the call to RequestResponse will throw an exception if no response has been received (for example, when the service handling the requests is not running).

Optionally you can also provide a prefix for the dynamic queue registered to receive responses, so you can distinguish the queue for your application in the RabbitMQ management interface. If not provided it defaults to "transient".

```
var config = new TapetiConfig(new SimpleInjectorDependencyResolver(container))
    .WithTransient(TimeSpan.FromSeconds(30), "myapplication.transient")
    .RegisterAllControllers()
    .Build();
```

7.2 Using transient requests

To send a transient request, inject ITransientPublisher into the class where you want to use it and call RequestResponse:

```
public class BirthdayHandler
{
    private ITransientPublisher transientPublisher;

    public BirthdayHandler(ITransientPublisher transientPublisher)
    {
        this.transientPublisher = transientPublisher;
    }

    public async Task SendBirthdayCard(RabbitInfo rabbit)
    {
        var response = await transientPublisher.RequestResponse
↪<SendCardRequestMessage, SendCardResponseMessage>(
          new SendCardRequestMessage
          {
              RabbitID = rabbit.RabbitID,
              Age = rabbit.Age,
              Style = CardStyles.Funny
          });

        // Handle the response
    }
}
```

CHAPTER 8

Tapeti.Cmd

Tapeti.Cmd has been moved to it's own repository at https://github.com/MvRens/Tapeti.Cmd. Along with it, the documentation is now available at https://tapeticmd.readthedocs.io/.