SimpleBus Documentation

Release latest

Cliff Odijk, Matthias Noback

December 12, 2018

Simplebus is a organization that helps you to use CQRS and event sourcing in your application. Get started by reading more about these concepts LINK or by digging in to common use cases LINK.

CHAPTER 1

Features and limitations

Why we do not have queries. Why we chose not to return thins from command handlers.

CHAPTER 2

Package design

Why so many packages. Refer to Matthias Noback's Principle of package design.

Getting started

Step by step how to include libraries and bundles

Organization overview

The organization has quite a few packages but each of them are very small. The packages have a single responsibility. This page will describe all packages and what they should be used for.

MessageBus

Generic classes and interfaces for messages and message buses. The most common middleware does also live here. Both commands and events are messages.

Asynchronous

To enable asynchronous messages with SimpleBus. This package contains strategies for publishing messages, producers and consumers. To use this package you will need a serializer and a library that can publish messages on some kind of queue.

Serialization

Generic classes and interfaces for serializing messages. This will put messages in an envelope and serialize the body of the envelope.

JMSSerializerBridge

Bridge for using JMSSerializer as message serializer with SimpleBus/Serialization.

DoctrineORMBridge

Bridge for using commands and events with Doctrine ORM. This will allow you do execute commands in a Doctrine transaction. It will also handle your entities domain events.

DoctrineDBALBridge

Bridge for using SimpleBus with Doctrine DBAL. This will allow you do execute commands in a Doctrine transaction.

SymfonyBridge

Bridge for using command buses and event buses in Symfony projects. This package contains the CommandBusBundle, EventBusBundle and DoctrineOrmBridgeBundle.

AsynchronousBundle

Symfony bundle for using SimpleBus/Asynchronous

JMSSerializerBundleBridge

A small bundle to use the JMSSerializerBridge with Symfony.

RabbitMQBundleBridge

Use OldSoundRabbitMQBundle with SimpleBus/AsynchronousBundle.

Introduction to CQRS and event sourcing

What it is, why, the defition. Why not retinging from commands is good. Read more about it on Matthias Noback blog posts.

Contributing

The documentation

We are happy for documentation contributions. This section will show you how to get up and running with contribution the SimpleBus documentations. The documentation is formatted in `reStructuredText`. For this we use Sphinx, a tool that makes it easy to create beautiful documentation. Assuming you have Python already installed, install Sphinx:

```
$ pip install sphinx sphinx-autobuild
```

Download GIT repository

Before you can start contributing the documentations you have to, fork the repository, clone it and create a new branch with the following commands:

```
$ git clone https://github.com/your-name/repo-name.git
$ git checkout -b documentation-description
```

After cloning the documentation repository you can open these files in your preferred IDE. Now it's time to start editing one of the the `.rst` files. For example the `contributing.rst` and add the information you are missing in the project.

Install the dependencies

This documentation is making use of external open source dependencies. You can think of the Read the Docs theme and the Sphinx Extensions for PHP and Symfony. You can install these by the following command.

```
$ pip install -r requirements.txt
```

Building the documentation

After you have installed the open source dependencies and changed some files, you can manually rebuild the documentation HTML output. You can see the result by opening the `_build/html/index.html` file.

```
$ make html
```

Note: You can use `sphinx-autobuild` to auto-reload your docs. Run `make autobuild` instead of `make html`.

Spelling

This documentation makes use of the Sphinx spelling extension, a spelling checker for Sphinx-based documentation. You run this by the following command:

```
$ make spelling
```

If there are some technical words that are not recognized, then you have to add them to `spelling_word_list.txt`. Please fill in this glossary in alphabetical order. As an example, you'll see the output below for the word `symfony` that's not found in the `contributing.rst` file.

```
contributing.rst:55:symfony:
```

Commit & pull request

Now it's time to commit your changes and push it to your repository. The last step to finish your contribution, is to create an pull requests for your valuable contribution. Thank you!

Asynchronous example

This article will explain how to use asynchronous messages with Symfony. We will assume that you know the basics of SimpleBus, CQRS and event sourcing. This is just **an** example. you could of course have a working asynchronous set up with SimpleBus and Symfony in a different way and with different libraries.

Installation

Install Simplebus, async support, message serializer and the RabbitMQBundle.

```
composer require simple-bus/asynchronous-bundle simple-bus/symfony-bridge simple-bus/doctrine-orm-bri
```

Register the bundles in Symfony's AppKernel.php

```
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            ...
            new SimpleBus\SymfonyBridge\SimpleBusCommandBusBundle()
            new SimpleBus\SymfonyBridge\SimpleBusEventBusBundle()
            new SimpleBus\SymfonyBridge\DoctrineOrmBridgeBundle()
            new SimpleBus\AsynchronousBundle\SimpleBusAsynchronousBundle()
            new SimpleBus\RabbitMQBundleBridge\SimpleBusRabbitMQBundleBridgeBundle()
            new SimpleBus\JMSSerializerBundleBridge\SimpleBusJMSSerializerBundleBridgeBundle()
            new OldSound\RabbitMqBundle\OldSoundRabbitMqBundle()
            new JMS\SerializerBundle\JMSSerializerBundle()
        )
        // ...
    }
    // ...
}
```

Configuration

There is quite a lot of moving parts in this configuration. Most if it is to configure the queue and make sure RabbitMqBundle is aware of SimpleBus' consumers and producers.

```
// app/config/config.yml
parameters:
  app.command_queue: 'commands'
  app.event_queue: 'events'

simple_bus_rabbit_mq_bundle_bridge:
  commands:
    producer_service_id: old_sound_rabbit_mq.asynchronous_commands_producer
  events:
    producer_service_id: old_sound_rabbit_mq.asynchronous_events_producer

simple_bus_asynchronous:
  events:
    strategy: 'predefined'

old_sound_rabbit_mq:
```

```
  connections:
    default:
      host:     "127.0.0.1"
      port:     5672
      user:     'guest'
      password: 'guest'
      vhost:    '/'
      lazy:     false
      connection_timeout: 3
      read_write_timeout: 3

      # requires php-amqplib v2.4.1+ and PHP5.4+
      keepalive: false

      # requires php-amqplib v2.4.1+
      heartbeat: 0
  producers:
    asynchronous_commands:
      connection:       default
      exchange_options: { name: '%app.command_queue%', type: "direct" }

    asynchronous_events:
      connection:       default
      exchange_options: { name: '%app.event_queue%', type: "direct" }

  consumers:
    asynchronous_commands:
      connection:       default
      exchange_options: { name: '%app.command_queue%', type: direct }
      queue_options:    { name: '%app.command_queue%' }
      callback:         simple_bus.rabbit_mq_bundle_bridge.commands_consumer

    asynchronous_events:
      connection:       default
      exchange_options: { name: '%app.command_queue%', type: direct }
      queue_options:    { name: '%app.command_queue%' }
      callback:         simple_bus.rabbit_mq_bundle_bridge.events_consumer
```

Usage

The first thing we need to do is to create a command and tag the command handler as asynchronous. You do that with the `asynchronous_command_handler` tag.

```
services:
  command_handler.email.SendEmailToAllUsers:
    class: App\Message\CommandHandler\Email\SendEmailToAllUsersHandler
    autowire: true
    tags:
      - { name: 'asynchronous_command_handler', handles: App\Message\Command\Email\SendEmailToAllUse
```

You can of course to the very same with events subscribers. When tagging event subscribers as asynchronous you should use the `asynchronous_event_subscriber` tag.

SimpleBus will automatically make sure that the messages get put on the queue. There is not special way you would create and handle asynchronous messages.

```php
$this->container->get('command_bus')->handle(new SendEmailToAllUsers());
```

Consuming Messages

There is different strategies you could use to consume messages from the queue. One simple solution is to run the following commands. They will start listening on incoming messages and consume them. If you are using these commands it is recommended to set up `supervisord`.

```
php app/console rabbitmq:consume asynchronous_events
php app/console rabbitmq:consume asynchronous_commands
```

Implementing a command bus

The classes and interfaces from this package can be used to set up a command bus. The characteristics of a command bus are:

- It handles *commands*, i.e. imperative messages
- Commands are handled by exactly one *command handler*
- The behavior of the command bus is extensible: *middlewares* are allowed to do things before or after handling a command

Setting up the command bus

At least we need an instance of `MessageBusSupportingMiddleware`:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;

$commandBus = new MessageBusSupportingMiddleware();
```

Finish handling a command, before handling the next

We want to make sure that commands are always fully handled before other commands will be handled, so we add a specialized middleware for that:

```php
use SimpleBus\Message\Bus\Middleware\FinishesHandlingMessageBeforeHandlingNext;

$commandBus->appendMiddleware(new FinishesHandlingMessageBeforeHandlingNext());
```

Defining the command handler map

Now we also want commands to be handled by exactly one command handler (which can be any callable). We first need to define the collection of handlers that are available in the application. We should make this *command handler map* lazy-loading, or every command handler will be fully loaded, even though it is not going to be used:

```php
use SimpleBus\Message\CallableResolver\CallableMap;
use SimpleBus\Message\CallableResolver\ServiceLocatorAwareCallableResolver;

// Provide a map of command names to callables. You can provide actual callables, or lazy-loading one
$commandHandlersByCommandName = [
```

```
    'Fully\Qualified\Class\Name\Of\Command' => ... // a "callable"
];
```

Each of the provided "callables" can be one of the following things:

- An actual PHP callable,
- A service id (string) which the service locator (see below) can resolve to a PHP callable,
- An array of which the first value is a service id (string), which the service locator can resolve to a regular object, and the second value is a method name.

For backwards compatibility an object with a `handle()` method also counts as a "callable".

```
// Provide a service locator callable. It will be used to instantiate a handler service whenever requ
$serviceLocator = function ($serviceId) {
    $handler = ...;

    return $handler;
}

$commandHandlerMap = new CallableMap(
    $commandHandlersByCommandName,
    new ServiceLocatorAwareCallableResolver($serviceLocator)
);
```

Resolving the command handler for a command

The name of a command

First we need a way to resolve the name of a command. You can use the fully-qualified class name (FQCN) of a command object as its name:

```
use SimpleBus\Message\Name\ClassBasedNameResolver;

$commandNameResolver = new ClassBasedNameResolver();
```

Or you can ask command objects what their name is:

```
use SimpleBus\Message\Name\NamedMessageNameResolver;

$commandNameResolver = new NamedMessageNameResolver();
```

In that case your commands have to implement `NamedMessage`:

```
use SimpleBus\Message\Name\NamedMessage;

class YourCommand implements NamedMessage
{
    public static function messageName()
    {
        return 'your_command';
    }
}

.. rubric:: Implementing your own ``MessageNameResolver``
   :name: implementing-your-own-messagenameresolver

If you want to use another rule to determine the name of a command,
```

```
create a class that implements
``SimpleBus\Message\Name\MessageNameResolver``.
```

Resolving the command handler based on the name of the command

Using the `MessageNameResolver` of your choice, you can now let the *command handler resolver* find the right command handler for a given command.

```
use SimpleBus\Message\Handler\Resolver\NameBasedMessageHandlerResolver;

$commandHandlerResolver = new NameBasedMessageHandlerResolver(
    $commandNameResolver,
    $commandHandlerMap
);
```

Finally, we should add some middleware to the command bus that calls the resolved command handler:

```
use SimpleBus\Message\Handler\DelegatesToMessageHandlerMiddleware;

$commandBus->appendMiddleware(
    new DelegatesToMessageHandlerMiddleware(
        $commandHandlerResolver
    )
);
```

Using the command bus: an example

Consider the following command:

```
class RegisterUser
{
    private $emailAddress;
    private $plainTextPassword;

    public function __construct($emailAddress, $plainTextPassword)
    {
        $this->emailAddress = $emailAddress;
        $this->plainTextPassword = $plainTextPassword;
    }

    public function emailAddress()
    {
        return $this->emailAddress;
    }

    public function plainTextPassword()
    {
        return $this->plainTextPassword;
    }
}
```

This command communicates the intention to "register a new user". The message data consists of an email address and a password in plain text. This information is required to execute the desired behavior.

The handler for this command looks like this:

```php
class RegisterUserCommandHandler
{
    ...

    public function handle(RegisterUser $command)
    {
        $user = User::register(
            $command->emailAddress(),
            $command->plainTextPassword()
        );

        $this->userRepository->add($user);
    }
}
```

We should register this handler as a service and add the service id to the *command handler map*. Since we have already fully configured the command bus, we can just start creating a new command object and let the command bus handle it. Eventually the command will be passed as a message to the `RegisterUserCommandHandler`:

```php
$command = new RegisterUser(
    'matthiasnoback@gmail.com',
    's3cr3t'
);

$commandBus->handle($command);

.. rubric:: Implementing your own command bus middleware
   :name: implementing-your-own-command-bus-middleware
```

It's very easy to extend the behavior of the command bus. You can create a class that implements `MessageBusMiddleware`:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusMiddleware;

/**
 * Marker interface for commands that should be handled asynchronously
 */
interface IsHandledAsynchronously
{
}

class HandleCommandsAsynchronously implements MessageBusMiddleware
{
    ...

    public function handle($message, callable $next)
    {
        if ($message instanceof IsHandledAsynchronously) {
            // handle the message asynchronously using a message queue
            $this->messageQueue->add($message);
        } else {
            // handle the message synchronously, i.e. right-away
            $next($message);
        }
    }
}
```

You should add an instance of that class as middleware to any `MessageBusSupportingMiddleware` instance (like the command bus we created earlier):

```php
$commandBus->appendMiddleware(new HandleCommandsAsynchronously());
```

Make sure that you do this at the right place, before or after you add the other middlewares.

Calling `$next($message)` will make sure that the next middleware in line is able to handle the message.

Logging messages

To log every message that passes through the command bus, add the `LoggingMiddleware` right before the `DelegatesToMessageHandlerMiddleware`. Make sure to set up a PSR-3 compliant logger first:

```php
use Psr\Log\LoggerInterface;
use Psr\Log\LogLevel;

// $logger is an instance of LoggerInterface
$logger = ...;
$loggingMiddleware = new LoggingMiddleware($logger, LogLevel::DEBUG);
$commandBus->appendMiddleware($loggingMiddleware);
```

Continue to read about the perfect complement to the command bus: the event bus.

Implementing an event bus

The classes and interfaces from this package can also be used to set up an event bus. The characteristics of an event bus are:

- It handles *events*, i.e. informational messages
- Zero or more *event subscribers* will be notified of the occurance of an event
- The behavior of the event bus is extensible: *middlewares* are allowed to do things before or after handling an event

Setting up the event bus

At least we need an instance of `MessageBusSupportingMiddleware`:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;

$eventBus = new MessageBusSupportingMiddleware();
```

Finish handling an event, before handling the next

We want to make sure that events are always fully handled before other events will be handled, so we add a specialized middleware for that:

```php
use SimpleBus\Message\Bus\Middleware\FinishesHandlingMessageBeforeHandlingNext;

$eventBus->appendMiddleware(new FinishesHandlingMessageBeforeHandlingNext());
```

Defining the event subscriber collection

We want any number of event subscribers to be notified of a given event. We first need to define the collection of event subscribers that are available in the application. We should make this *event subscriber collection* lazy-loading, or every event subscriber will be fully loaded, even though it is not going to be used:

```
use SimpleBus\Message\CallableResolver\CallableCollection;
use SimpleBus\Message\CallableResolver\ServiceLocatorAwareCallableResolver;

// Provide a map of event names to callables. You can provide actual callables, or lazy-loading ones
$eventSubscribersByEventName = [
    Fully\Qualified\Class\Name\Of\Event::class => [ // an array of "callables",
        ...,
        ...
    ]
    ...
];
```

Each of the provided "callables" can be one of the following things:

- An actual PHP callable,
- A service id (string) which the service locator (see below) can resolve to a PHP callable,
- An array of which the first value is a service id (string), which the service locator can resolve to a regular object, and the second value is a method name.

For backwards compatibility an object with a `notify()` method also counts as a "callable".

```
// Provide a service locator callable. It will be used to instantiate a subscriber service whenever
$serviceLocator = function ($serviceId) {
    $handler = ...;

    return $handler;
};

$eventSubscriberCollection = new CallableCollection(
    $eventSubscribersByEventName,
    new ServiceLocatorAwareCallableResolver($serviceLocator)
);
```

Resolving the event subscribers for an event

The name of an event

First we need a way to resolve the name of an event. You can use the fully-qualified class name (FQCN) of an event object as its name:

```
use SimpleBus\Message\Name\ClassBasedNameResolver;

$eventNameResolver = new ClassBasedNameResolver();
```

Or you can ask event objects what their name is:

```
use SimpleBus\Message\Name\NamedMessageNameResolver;

$eventNameResolver = new NamedMessageNameResolver();
```

In that case your events have to implement `NamedMessage`:

```
use SimpleBus\Message\Name\NamedMessage;

class YourEvent implements NamedMessage
{
    public static function messageName()
    {
        return 'your_event';
    }
}

.. rubric:: Implementing your own ``MessageNameResolver``
   :name: implementing-your-own-messagenameresolver

If you want to use another rule to determine the name of an event,
create a class that implements
``SimpleBus\Message\Name\MessageNameResolver``.
```

Resolving the event subscribers based on the name of the event

Using the `MessageNameResolver` of your choice, you can now let the *event subscribers resolver* find the right event subscribers for a given event.

```
use SimpleBus\Message\Subscriber\Resolver\NameBasedMessageSubscriberResolver;

$eventSubscribersResolver = new NameBasedMessageSubscriberResolver(
    $eventNameResolver,
    $eventSubscriberCollection
);
```

Finally, we should add some middleware to the event bus that notifies all of the resolved event subscribers:

```
use SimpleBus\Message\Subscriber\NotifiesMessageSubscribersMiddleware;

$eventBus->appendMiddleware(
    new NotifiesMessageSubscribersMiddleware(
        $eventSubscribersResolver
    )
);
```

Using the event bus: an example

Consider the following event:

```
class UserRegistered
{
    private $userId;

    public function __construct(UserId $userId)
    {
        $this->userId = $userId;
    }

    public function userId()
    {
        return $this->userId;
```

```
    }
}
```

This event conveys the information that "a new user was registered". The message data consists of the unique identifier of the user that was registered. This information is required for event subscribers to act upon the event.

A subscriber for this event looks like this:

```
class SendWelcomeMailWhenUserRegistered
{
    ...

    public function notify($message)
    {
        $user = $this->userRepository->byId($message->userId());

        // send the welcome mail
    }
}
```

We should register this subscriber as a service and add the service id to the *event subscriber collection*. Since we have already fully configured the event bus, we can just start creating a new event object and let the event bus handle it. Eventually the event will be passed as a message to the `SendWelcomeMailWhenUserRegistered` event subscriber:

```
$userId = $this->userRepository->nextIdentity();

$event = new UserRegistered($userId);

$eventBus->handle($event);
```

Implementing your own event bus middleware

It's very easy to extend the behavior of the event bus. You can create a class that implements `MessageBusMiddleware`:

```
use SimpleBus\Message\Bus\Middleware\MessageBusMiddleware;

/**
 * Marker interface for domain events that should be stored in the event store
 */
interface DomainEvent
{
}

class StoreDomainEvents implements MessageBusMiddleware
{
   // ...

    public function handle($message, callable $next)
    {
        if ($message instanceof DomainEvent) {
            // store the domain event
            $this->eventStore->add($message);
        }

        // let other middlewares do their job
```

```
        $next($message);
    }
}
```

You should add an instance of that class as middleware to any `MessageBusSupportingMiddleware` instance (like the event bus we created earlier):

```
$eventBus->appendMiddleware(new StoreDomainEvents());
```

Make sure that you do this at the right place, before or after you add the other middlewares.

Calling `$next($message)` will make sure that the next middleware in line is able to handle the message.

Logging messages

To log every message that passes through the event bus, add the `LoggingMiddleware` right before the `NotifiesMessageSubscribersMiddleware`. Make sure to set up a PSR-3 compliant logger first:

```
use Psr\Log\LoggerInterface;
use Psr\Log\LogLevel;
use SimpleBus\Message\Logging\LoggingMiddleware;

// $logger is an instance of LoggerInterface
$logger = ...;
$loggingMiddleware = new LoggingMiddleware($logger, LogLevel::DEBUG);
$eventBus->appendMiddleware($loggingMiddleware);
```

Continue to read about recording events and handling them.

Recording events and handling them

While the *command bus* handles a command, certain events will take place. It might be important to *record these events* and, when the command has been fully handled, notify other parts of the system about the events that were recorded.

This can be accomplished by using *message recorders*. These are objects with the ability to record messages. From the outside these messages can be retrieved, and erased:

```
interface ContainsRecordedMessages
{
    public function recordedMessages();

    public function eraseMessages();
}
```

Collecting events

Publicly

The default implementation, which has a public `record()` method as well, is the `PublicMessageRecorder`:

```
use SimpleBus\Message\Recorder\PublicMessageRecorder;

$publicMessageRecorder = new PublicMessageRecorder();
```

```php

$event = new UserRegistered(...);

$publicMessageRecorder->record($event);

$recordedEvents = $publicMessageRecorder->recordedEvents();
// $recordedEvents is an array containing the previously recorded $event object
```

Privately

When you use domain events, your domain entities will generate events while you change them. You record those events inside the entity. Later, when the changes have been persisted and the database transaction has succeeded, you should collect the recorded events and handle them:

```php
$entity->changeSomething();
// $entity generates a SomethingChanged event and records it internally

// start transaction
$entityManager->persist($entity);
// commit transaction

$events = $entity->recordedEvents();

// handle the events
foreach ($events as $event) {
    $eventBus->handle($event);
}
```

You can give your entities the ability to record their own events by letting them implement the `RecordsMessages` interface and using the `PrivateMessageRecorderCapabilities` trait:

```php
use SimpleBus\Message\Recorder\RecordsMessages;
use SimpleBus\Message\Recorder\PrivateMessageRecorderCapabilities;

class YourEntity implements RecordsMessages
{
    use PrivateMessageRecorderCapabilities;

    public function changeSomething()
    {
        ...

        $this->record(new SomethingChanged());
    }
}
```

Handling events

Handling publicly recorded events

Events are recorded while a command is handled. We only want to handle the events themselves *after* the command has been completely and successfully been handled. The best option to accomplish this is to add a piece of middleware to the command bus. This middleware needs the *message recorder* to find out which events were recorded during the handling of a command, and it needs the *event bus* to actually handle the recorded events:

```php
use SimpleBus\Message\Recorder\HandlesRecordedMessagesMiddleware;

$commandBus->appendMiddleware(new HandlesRecordedMessagesMiddleware(
    $publicMessageRecorder,
    $eventBus
));
```

Make sure to add this middleware *first*, before adding any other middleware. Like mentioned before: we only want events to be handled when we know that everything else has gone well.

Only the command bus handled recorded events automatically

When using a standard setup (like described above), *only* the command bus automatically handles recorded events. If you want to dispatch new events in for example event subscribers, you shouldn't record the event, but just inject the event bus as a constructor argument and let it handle the new event right-away.

Handling domain events

When you privately record events inside your domain entities, you need to collect those recorded events manually. Your database abstraction library, ORM or ODM probably offers a way to hook into the process of persisting the entities and collecting them somehow. After the command has been handled successfully and the transaction has been committed, you can iterate over those entities and collect their recorded events.

Handling domain events with Doctrine ORM

SimpleBus comes with a Doctrine ORM bridge. Using this package you can collect recorded events from Doctrine ORM entities. See its README file for further instructions.

Combining multiple message recorders

If you have multiple ways in which you record events, e.g. using the `PublicMessageRecorder` and using domain events, you can combine those into one message recorder, which aggregates the recorded messages:

```php
use SimpleBus\Message\Recorder\AggregatesRecordedMessages;

$aggregatingMessageRecorder = new AggregatesRecordedMessages(
    [
        $publicMessageRecorder,
        $domainEventsMessagesRecorder,
        ...
    ]
);
```

Finally, you can provide this aggregating message recorder to the `HandlesRecordedMessagesMiddleware` and it will act as if it is a single message recorder.

```php
$commandBus->appendMiddleware(new HandlesRecordedMessagesMiddleware(
    $aggregatingMessageRecorder,
    $eventBus
));
```

Asynchronous

This package contains generic classes and interfaces which can be used to process messages asynchronously using a SimpleBus MessageBus instance.

@TODO The intro should explain what it does.

Publishing messages

When a `Message` should not be handled by the message bus (i.e. command or event bus) immediately (i.e. synchronously), it can be *published* to be handled by some other process. This library comes with three strategies for publishing messages:

1. A message will always *also* be published.
2. A message will only be published when the message bus isn't able to handle it because there is no handler defined for it.
3. A message will be published only if its name exists in a predefined list.

Strategy 1: Always publish messages

This strategy is very useful when you have an event bus that notifies event subscribers of events that have occurred. If you have set up the event bus, you can add the `AlwaysPublishesMessages` middleware to it:

```
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;
use SimpleBus\Asynchronous\MessageBus\AlwaysPublishesMessages;
use SimpleBus\Asynchronous\Publisher\Publisher;
use SimpleBus\Message\Message;

// $eventBus is an instance of MessageBusSupportingMiddleware
$eventBus = ...;

// $publisher is an instance of Publisher
$publisher = ...;

$eventBus->appendMiddleware(new AlwaysPublishesMessages($publisher));

// $event is an object
$event = ...;

$eventBus->handle($event);
```

The middleware publishes the message to the publisher (which may add it to some a queue of some sorts). After that it just calls the next middleware and lets it process the same message in the usual way.

By applying this strategy you basically allow other processes to respond to any event that occurs within your application.

Strategy 2: Only publish messages that could not be handled

This strategy is useful if you have a command bus that handles commands. If you have set up the command bus, you can add the `PublishesUnhandledMessages` middleware to it:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;
use SimpleBus\Asynchronous\MessageBus\PublishesUnhandledMessages;
use SimpleBus\Asynchronous\Publisher\Publisher;
use Psr\Log\LoggerInterface;
use Psr\Log\LogLevel;

// $commandBus is an instance of MessageBusSupportingMiddleware
$commandBus = ...;

// $publisher is an instance of Publisher
$publisher = ...;

// $logger is an instance of LoggerInterface
$logger = ...;

// $logLevel is one of the class constants of LogLevel
$logLevel = LogLevel::DEBUG;

$commandBus->appendMiddleware(new PublishesUnhandledMessages($publisher, $logger, $logLevel));

// $command is an object
$command = ...;

$commandBus->handle($command);
```

Because of the nature of commands (they have a one-to-one correspondence with their handlers), it doesn't make sense to always publish a command. Instead, it should only be published when it *couldn't be handled by your application*. Possibly some other process knows how to handle it.

If no command handler was found and the command is published, this will be logged using the provided `$logger`.

Strategy 3: Only publish predefined messages

This strategy is useful when you know what messages you want to publish.

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;
use SimpleBus\Asynchronous\MessageBus\AlwaysPublishesMessages;
use SimpleBus\Asynchronous\Publisher\Publisher;
use SimpleBus\Message\Message;
use SimpleBus\Message\Name\MessageNameResolver;

// $eventBus is an instance of MessageBusSupportingMiddleware
$eventBus = ...;

// $publisher is an instance of Publisher
$publisher = ...;

// $messageNameResolver is an instance of MessageNameResolver
$messageNameResolver = ...;

// The list of names will depend on what MessageNameResolver you are using.
$names = ['My\\Event', 'My\\Other\\Event'];

$eventBus->appendMiddleware(new PublishesPredefinedMessages($publisher, $messageNameResolver, $names)

// $event is an object
$event = ...;
```

```
23
24 $eventBus->handle($event);
```

Consuming messages

When a message has been *published*, for instance to some kind of queue, another process should be able to *consume* it, i.e. receive and process it.

A message consumer actually consumes serialized envelopes, instead of the messages themselves. A consumer then restores the `Envelope` by deserializing it and finally it can restore the `Message` itself by deserializing the serialized message carried by the `Envelope`.

To ease integration of existing messaging software with `SimpleBus/Asynchronous`, this library contains a standard implementation of a `SerializedEnvelopeConsumer`. It deserializes a serialized `Envelope`, then lets the message bus handle the `Message` contained in the `Envelope`.

```
1  use SimpleBus\Asynchronous\Consumer\StandardSerializedEnvelopeConsumer;
2  use SimpleBus\Serialization\Envelope\Serializer\MessageInEnvelopeSerializer;
3  use SimpleBus\Message\Bus\MessageBus;
4
5  // $messageSerializer is an instance of MessageInEnvelopeSerializer
6  $messageSerializer = ...;
7
8  // $messageBus is an instance of MessageBus
9  $messageBus = ...;
10
11 $consumer = StandardSerializedEnvelopeConsumer($messageSerializer, $messageBus);
12
13 // keep fetching serialized envelopes
14 while ($aSerializedEnvelope = ...) {
15     // this causes $messageBus to handle the deserialized Message
16     $consumer->consume($aSerializedEnvelope);
17 }
```

For more information about envelopes and serializing messages, take a look at the documentation of SimpleBus/Serialization.

Routing keys

A routing key is a concept that originates from RabbitMQ: it allows you to let particular groups of messages be routed to specific queues, which may then be consumed by dedicated consumers.

Whether or not you use RabbitMQ, you might need the concept of a routing key somewhere in your application. This library contains an interface `RoutingKeyResolver` and two very simple standard implementations of it:

1. The `ClassBasedRoutingKeyResolver`: when asked to resolve a routing key for a given `Message`, it takes the full class name of it and replaces `\` with `.`.
2. The `EmptyRoutingKeyResolver`: it always returns an empty string as the routing key for a given `Message`.

Additional properties

"Additional properties" is a concept that originates from RabbitMQ: it allows you to add metadata or otherwise configure a message before it is sent to the server.

Whether or not you use RabbitMQ, you might need these additional (message) properties somewhere in your application. This library contains an interface `AdditionalPropertiesResolver` and one implementation of that interface, the `DelegatingAdditionalPropertiesResolver` which accepts an array of `AdditionalPropertiesResolver` instances. It lets them all step in and provide values:

```
use SimpleBus\Asynchronous\Properties\DelegatingAdditionalPropertiesResolver;
use SimpleBus\Asynchronous\Properties\AdditionalPropertiesResolver;

class MyPropertiesResolver implements AdditionalPropertiesResolver
{
    public function resolveAdditionalPropertiesFor($message)
    {
        // determine which properties to use

        return [
            'content-type' => 'application/xml'
        ];
    }
}

$delegatingResolver = new DelegatingAdditionalPropertiesResolver(
    [
        new MyPropertiesResolver(),
        ...
    ]
);

// $message is some message (e.g. a command or event)
$message = ...;

$properties = $delegatingResolver->resolveAdditionalPropertiesFor($message);
```

DoctrineDBALBridge

This package provides a command bus middleware that can be used to integrate SimpleBus/MessageBus with Doctrine DBAL.

It provides an easy way to wrap command handling in a database transaction.

@TODO The intro should explain what it does.

Getting started

Installation

Using Composer:

```
composer require simple-bus/doctrine-dbal-bridge
```

Preparations

To use the middleware provided by the library, set up a command bus, if you didn't already do this:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;

$commandBus = new MessageBusSupportingMiddleware();
...
```

Make sure to also properly set up a Doctrine connection:

```php
// $connection is an instance of Doctrine\DBAL\Driver\Connection
$connection = ...;
```

Transactions

It is generally a good idea to wrap command handling in a database transaction. If you want to do this, add the `WrapsMessageHandlingInTransaction` middleware to the command bus. Provide an instance of the Doctrine `Connection` interface that you want to use.

```php
use SimpleBus\DoctrineDBALBridge\MessageBus\WrapsMessageHandlingInTransaction;

// $connection is an instance of Doctrine\DBAL\Driver\Connection
$connection = ...;

$transactionalMiddleware = new WrapsMessageHandlingInTransaction($connection);

$commandBus->addMiddleware($transactionalMiddleware);
```

When an exception is thrown, the transaction will be rolled back. If not, the transaction is committed.

DoctrineORMBridge

This package provides command bus middlewares that can be used to integrate SimpleBus/MessageBus with Doctrine ORM.

It provides an easy way to wrap command handling in a database transaction and handle domain events generated by entities.

@TODO The intro should explain what it does.

Getting started

Installation

Using Composer:

```
composer require simple-bus/doctrine-orm-bridge
```

Preparations

To use the middlewares provided by the library, set up a command bus and an event bus, if you didn't already do this:

```php
use SimpleBus\Message\Bus\Middleware\MessageBusSupportingMiddleware;

$commandBus = new MessageBusSupportingMiddleware();
...
```

```
$eventBus = new MessageBusSupportingMiddleware();
...
```

Make sure to also properly set up an entity manager:

```
// $entityManager is an instance of Doctrine\ORM\EntityManager
$entityManager = ...;
```

Now add the available middlewares for *transaction handling* and *domain events*.

Transactions

It is generally a good idea to wrap command handling in a database transaction. If you want to do this, add the `WrapsMessageHandlingInTransaction` middleware to the command bus. Provide an instance of the Doctrine `ManagerRegistry` interface and the name of the entity manager that you want to use.

```
use SimpleBus\DoctrineORMBridge\MessageBus\WrapsMessageHandlingInTransaction;

/*
 * $managerRegistry is an instance of Doctrine\Common\Persistence\ManagerRegistry
 *
 * For example: if you use Symfony, use the "doctrine" service
 */
$managerRegistry = ...;

$transactionalMiddleware = new WrapsMessageHandlingInTransaction($managerRegistry, 'default');

$commandBus->addMiddleware($transactionalMiddleware);
```

Note: Once you have added this middleware, you shouldn't call `EntityManager::flush()` manually from inside your command handlers anymore.

Domain events

Using the message recorder facilities from `SimpleBus/MessageBus` you can let Doctrine ORM collect domain events and subsequently let the event bus handle them.

Make sure that your entities implement the `ContainsRecordedMessages` interface. Use the `PrivateMessageRecorderCapabilities` trait to conveniently record events from inside the entity:

```
use SimpleBus\Message\Recorder\ContainsRecordedMessages;
use SimpleBus\Message\Recorder\PrivateMessageRecorderCapabilities;

class YourEntity implements ContainsRecordedMessages
{
    use PrivateMessageRecorderCapabilities;

    public function changeSomething()
    {
        // record new events like this:

        $this->record(new SomethingChanged());
    }
}
```

Then set up the *event recorder* for Doctrine entities:

```
use SimpleBus\DoctrineORMBridge\EventListener\CollectsEventsFromEntities;

$eventRecorder = new CollectsEventsFromEntities();

$entityManager->getConnection()->getEventManager()->addEventSubscriber($eventRecorder);
```

The event recorder will loop over all the entities that were involved in the last database transaction and collect their internally recorded events.

After a database transaction was completed successfully these events should be handled by the event bus. This is done by a specialized middleware, which should be appended to the command bus *before* the middleware that is responsible for handling the transaction.

```
use SimpleBus\DoctrineORMBridge\MessageBus\WrapsMessageHandlingInTransaction;

use SimpleBus\Message\Bus\MessageBus;

$eventDispatchingMiddleware = new HandlesRecordedMessagesMiddleware($eventProvider, $eventBus);
// N.B. append this middleware *before* the WrapsMessageHandlingInTransaction middleware
$commandBus->appendMiddleware($eventDispatchingMiddleware);

$transactionalMiddleware = new WrapsMessageHandlingInTransaction($entityManager);
$commandBus->appendMiddleware($transactionalMiddleware);
```

Note: The `MessageBusSupportingMiddleware` class also has a `prependMiddleware()` method, which you can use to prepend middleware instead of appending it.

JMSSerializerBridge

@TODO Add docs

Serialization

This package contains generic classes and interfaces which can be used to serialize SimpleBus messages.

@TODO The intro should explain what it does.

Message envelopes

Before an instance of `SimpleBus\Message\Message` can be serialized to JSON, XML, etc. it has to be wrapped inside an envelope. The envelope contains some metadata about the message, e.g. the type of the message (its fully qualified class name - FQCN) and the message itself. `SimpleBus/Serialization` comes with a default implementation of an envelope, which can be used like this:

```
use SimpleBus\Serialization\Envelope\DefaultEnvelope;

// $message is an object
$message = ...;

$envelope = DefaultEnvelope::forMessage($message);
```

```

$fqcn = $envelope->messageType();
$message = $envelope->message();
```

Because the message itself is an object and needs to be transformed to plain text in order to travel over a network, you should serialize the message itself using an *object serializer* and get a new envelope instance with the serialized message:

```
// $serializedMessage is a string
$serializedMessage = ...;

$envelopeWithSerializedMessage = $envelope->withSerializedMessage($serializedMessage);
```

The new `Envelope` only contains the serialized message. Using the *object serializer* you can now safely serialize the entire envelope.

If an `Envelope` contains a serialized message and you have deserialized that message, you can get a new envelope by providing the actual message:

```
// $deserializedMessage is an instance of Message
$deserializedMessage = ...;

$envelopeWithActualMessage = $envelopeWithSerializedMessage->withMessage($deserializedMessage);
```

Custom envelope types

You may want to use your own type of envelopes, containing extra metadata like a timestamp, or the identifier of the machine that produced the message. In that case you can just implement your own `Envelope` class:

```
use SimpleBus\Serialization\Envelope\DefaultEnvelope;

class MyEnvelope extends DefaultEnvelope
{
    ...
}

// or

class MyEnvelope implements Envelope
{
    ...
}
```

Envelope factory

The message serializer uses an `EnvelopeFactory` to delegate the creation of envelopes to, so if you want to use your own type of envelopes, you should implement an envelope factory yourself as well:

```
use SimpleBus\Serialization\Envelope\EnvelopeFactory;
use SimpleBus\Message\Message;

class MyEnvelopeFactory implements EnvelopeFactory
{
    public function wrapMessageInEnvelope(Message $message)
    {
        return MyEnvelope::forMessage($message);
```

```
    }

    public function envelopeClass()
    {
        return 'Fully\Qualified\Class\Name\Of\MyEnvelope';
    }
}
```

Object serializer

An object serializer is supposed to be able to serialize *any object* handed to it. `SimpleBus/Serializer` contains a simple implementation of an object serializer, which uses the native PHP `serialize()` and `unserialize()` functions:

```
// $envelope is an instance of Envelope, containing a serialized message
$envelope = ...;

$serializer = NativeObjectSerializer();
$serializedEnvelope = $serializer->serialize($envelope);

$deserializedEnvelope = $serializer->deserialize($serializedEnvelope, get_class($envelope));
```

Note: You are encouraged to use a more advanced serializer like the JMSSerializer. SimpleBus/JMSSerializerBridge contains an adapter for the SimpleBus `ObjectSerializer` interface.

Using JSON or XML as the serialized format a message is better readable and understandable for humans, but more importantly, it's platform-independent.

Message serializer

In order to to send a message (object) over the network it needs to be wrapped in an `Envelope`. At the other end it may be unwrapped and processed. This standard procedure is implemented inside the `StandardMessageInEnvelopeSerializer`:

```
use SimpleBus\Serialization\Envelope\DefaultEnvelopeFactory;
use SimpleBus\Serialization\NativeObjectSerializer;
use SimpleBus\Serialization\Envelope\Serializer\StandardMessageInEnvelopeSerializer;

$envelopeFactory = new DefaultEnvelopeFactory();
$objectSerializer = new NativeObjectSerializer();

$serializer = StandardMessageInEnvelopeSerializer($envelopeFactory, $objectSerializer);

// $message is an object
$message = ...;

// $serializedEnvelope will be a string
$serializedEnvelope = $serializer->wrapAndSerialize($message);

...

// $deserializedEnvelope will be an instance of the original Envelope
$deserializedEnvelope = $serializer->unwrapAndDeserialize($serializedEnvelope);
```

```php

// $message will be an object which is a copy of the original message
$message = $deserializedEnvelope->message();
```

Getting started with Symfony

Using the Symfony framework will hide some of the complexity compared to when use are interacting directly with the components. The SymfonyBridge package contains the following bundles which can be used to integrate SimpeBus with a Symfony application:

- CommandBusBundle.
- EventBusBundle.
- DoctrineORMBridgeBundle.

Are you upgrading from a previous version? Read the upgrade guide.

Installation

Download the SymfonyBridge with composer.

```
composer require simple-bus/symfony-bridge
```

When composer is done you can enable the bundles you want in the AppKernel.php

```php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            //...
            new SimpleBus\SymfonyBridge\SimpleBusCommandBusBundle(),
            new SimpleBus\SymfonyBridge\SimpleBusEventBusBundle(),
            new SimpleBus\SymfonyBridge\DoctrineOrmBridgeBundle(),
        )
        //...
    }
    //...
}
```

Read more how you use the bundles in the documentation pages for CommandBusBundle, EventBusBundle and DoctrineORMBridgeBundle.

CommandBusBundle

Using the building blocks supplied by the `SimpleBus/MessageBus` library you can create a command bus, which is basically a message bus, with some middlewares and a map of message handlers. This is described in the `documentation of CommandBus`.

Using the command bus

This bundle provides the `command_bus` service which is an instance of `SimpleBus\SymfonyBridge\Bus\CommandBus`. Wherever you like, you can let it handle commands, e.g. inside a container-aware controller:

```
// $command is an arbitrary object that will be passed to the command handler
$command = ...;

$this->get('command_bus')->handle($command);
```

However, you are encouraged to properly inject the `command_bus` service as a dependency whenever you need it:

```
services:
    some_service:
        class: Acme\Foobar
        arguments:
            - "@command_bus"
```

This bundle can be used with Symfony's Autowiring out of the box.

Simply inject `SimpleBus\SymfonyBridge\Bus\CommandBus` in your controller or service:

```
namespace App\Controller;

use SimpleBus\SymfonyBridge\Bus\CommandBus;

class UpdatePhoneNumberController
{
    private $commandBus;

    public function __construct(CommandBus $commandBus)
    {
        $this->commandBus = $commandBus;
    }

    public function __invoke(Request $request)
    {
        $this->commandBus->handle(new SavePhoneNumberCommand($request->get('phone')));
    }
}
```

Registering command handlers

As described in the `MessageBus documentation` you can delegate the handling of particular commands to command handlers. This bundle allows you to register your own command handlers by adding the `command_handler` tag to the command handler's service definition:

```
services:
    register_user_command_handler:
        class: Fully\Qualified\Class\Name\Of\RegisterUserCommandHandler
        tags:
            - { name: command_handler, handles: Fully\Qualified\Class\Name\Of\RegisterUser }
```

Note: Command handlers are lazy-loaded

Since only one of the command handlers is going to handle any particular command, command handlers are lazy-loaded. This means that their services should be defined as public services (i.e. you can't use `public: false` for them).

Command handlers are callables

Any service that is a PHP callable itself can be used as a command handler. If a service itself is not callable, SimpleBus looks for a `__invoke` or `handle` method and calls it. If you want to use a custom method, just add a `method` attribute to the `command_handler` tag:

```
services:
    register_user_command_handler:
        ...
        tags:
            - { name: command_handler, handles: ..., method: registerUser }
```

Setting the command name resolving strategy

To find the correct command handler for a given command, the name of the command is used. This can be either 1) its fully-qualified class name (FQCN) or, 2) if the command implements the `SimpleBus\Message\Name\NamedMessage` interface, the value returned by its static `messageName()` method. By default, the first strategy is used, but you can configure it in your application configuration:

```
# app/config/config.yml
command_bus:
    # default value for this key is "class_based"
    command_name_resolver_strategy: named_message
```

When you change the strategy, you also have to change the value of the `handles` attribute of your command handler service definitions:

```
services:
    register_user_command_handler:
        class: Fully\Qualified\Class\Name\Of\RegisterUserCommandHandler
        tags:
            - { name: command_handler, handles: register_user }
```

Make sure that the value of `handles` matches the return value of `RegisterUser::messageName()`.

Adding command bus middleware

As described in the MessageBus documentation you can extend the behavior of the command bus by adding middleware to it. This bundle allows you to register your own middleware by adding the `command_bus_middleware` tag to the middleware service definition:

```
services:
    specialized_command_bus_middleware:
        class: YourSpecializedCommandBusMiddleware
        public: false
        tags:
            - { name: command_bus_middleware, priority: 100 }
```

By providing a value for the `priority` tag attribute you can influence the order in which middlewares are added to the command bus.

Note: Middlewares are not lazy-loaded

Whenever you use the command bus, you also use all of its middlewares, so command bus middlewares are not lazy-loaded. This means that their services should be defined as private services (i.e. you should use `public: false`). See also: Marking Services as public / private

Logging

If you want to log every command that is being handled, enable logging in `config.yml`:

```
# app/config/config.yml
command_bus:
    logging: ~
```

Messages will be logged to the `command_bus` channel.

Event bus bundle

Using the building blocks supplied by the `SimpleBus/MessageBus` library you can create an event bus, which is basically a message bus, with some middlewares and a collection of message subscribers. This is described in the `documentation of EventBus`.

Using the event bus

This bundle provides the `event_bus` service which is an instance of `SimpleBus\SymfonyBridge\Bus\MessageBus`. Wherever you like, you can let it handle events, e.g. by fetching it inside a container-aware controller:

```
// $event is an arbitrary object that will be passed to the event subscriber
$event = ...;

$this->get('event_bus')->handle($event);
```

However, you are encouraged to properly inject the `event_bus` service as a dependency whenever you need it:

```
services:
    some_service:
        class: Acme\Foobar
        arguments:
            - "@event_bus"
```

This bundle can be used with Symfony's Autowiring out of the box.

Simply inject `SimpleBus\SymfonyBridge\Bus\EventBus` in your controller or service:

```
namespace App\Service;

use SimpleBus\SymfonyBridge\Bus\EventBus;

class SomeService
{
    private $eventBus;

```

```
    public function __construct(EventBus $eventBus)
    {
        $this->eventBus = $eventBus;
    }

    public function __invoke()
    {
        $this->eventBus->handle(new SomethingHappenedEvent());
    }
}
```

Registering event subscribers

As described in the `EventBus documentation` you can notify event subscribers about the occurrence of a particular event. This bundle allows you to register your own event subscribers by adding the `event_subscriber` tag to the event subscriber's service definition:

```
services:
    user_registered_event_subscriber:
        class: Fully\Qualified\Class\Name\Of\UserRegisteredEventSubscriber
        tags:
            - { name: event_subscriber, subscribes_to: Fully\Qualified\Class\Name\Of\UserRegistered
```

Note: Event subscribers are lazy-loaded

Since only some of the event subscribers are going to handle any particular event, event subscribers are lazy-loaded. This means that their services should be defined as public services (i.e. you can't use `public: false` for them).

Event subscribers are callables

Any service that is a PHP callable itself can be used as an event subscriber. If a service itself is not callable, SimpleBus looks for a `__invoke` or `notify` method and calls it. If you want to use a custom method, just add a `method` attribute to the `event_subscriber` tag:

```
services:
    user_registered_event_subscriber:
        ...
        tags:
            - { name: event_subscriber, subscribes_to: ..., method: userRegistered }
```

If you are using Autowiring you can use the following configuration:

```
services:
    _defaults:
        autowire: true
        autoconfigure: true

    App\Subscriber\:
        resource: '%kernel.project_dir%/src/Subscriber'
        public: true
        tags: [{ name: 'event_subscriber' }]
```

This will search for all subscribers in the `src/Subscriber` directory and automatically detects the event that the subscriber is subscribing to.

One subscriber listening to multiple events

When you have 1 subscriber that is listening to multiple events you might want to set the `register_public_methods` attribute to `true`:

```yaml
services:
    _defaults:
        autowire: true
        autoconfigure: true

    App\Subscriber\:
        resource: '%kernel.project_dir%/src/Subscriber'
        public: true
        tags: [{ name: 'event_subscriber', register_public_methods: true }]
```

With the following code for the subscriber:

```php
namespace App\Subscriber;

use App\Event\EventAddedEvent;
use App\Event\VenueAddedEvent;

class ElasticSearchSubscriber
{
    public function onEventAdded(EventAddedEvent $event)
    {
        // Add the event to ElasticSearch
    }

    public function onVenueAdded(VenueAddedEvent $event)
    {
        // Add the venue to ElasticSearch
    }
}
```

SimpleBus automatically detects that `ElasticSearchSubscriber` wants to subscribe to both `EventAddedEvent` and `VenueAddedEvent`.

Setting the event name resolving strategy

To find the correct event subscribers for a given event, the name of the event is used. This can be either 1) its fully-qualified class name (FQCN) or, 2) if the event implements the `SimpleBus\Message\Name\NamedMessage` interface, the value returned by its static `messageName()` method. By default, the first strategy is used, but you can configure it in your application configuration:

```yaml
event_bus:
    # default value for this key is "class_based"
    event_name_resolver_strategy: named_message
```

When you change the strategy, you also have to change the value of the `subscribes_to` attribute of your event subscriber service definitions:

```yaml
services:
    user_registered_event_subscriber:
        class: Fully\Qualified\Class\Name\Of\UserRegisteredEventSubscriber
        tags:
            - { name: event_subscriber, subscribes_to: user_registered }
```

Make sure that the value of `subscribes_to` matches the return value of `UserRegistered::messageName()`.

Adding event bus middlewares

As described in the MessageBus documentation you can extend the behavior of the event bus by adding middlewares to it. This bundle allows you to register your own middlewares by adding the `event_bus_middleware` tag to middleware service definitions:

```
services:
    specialized_event_bus_middleware:
        class: YourSpecializedEventBusMiddleware
        public: false
        tags:
            - { name: event_bus_middleware, priority: 100 }
```

By providing a value for the `priority` tag attribute you can influence the order in which middlewares are added to the event bus.

Note: Middlewares are not lazy-loaded

Whenever you use the event bus, you also use all of its middlewares, so event bus middlewares are not lazy-loaded. This means that their services should be defined as private services (i.e. you should use `public: false`). See also: Marking Services as public / private

---

Event recorders

Recording events

As explained `in the documentation of MessageBus` you can collect events while a command is being handled. If you want to record new events you can inject the `event_recorder` service as a constructor argument of a command handler:

```
use SimpleBus\Message\Recorder\RecordsMessages;

class SomeInterestingCommandHandler
{
    private $eventRecorder;

    public function __construct(RecordsMessages $eventRecorder)
    {
        $this->eventRecorder = $eventRecorder;
    }

    public function handle($command)
    {
        ...

        // create an event
        $event = new SomethingInterestingHappened();

        // record the event
        $this->eventRecorder->record($event);
```

```
    }
}
```

The corresponding service definition looks like this:

```
services:
    some_interesting_command_handler:
    arguments:
        - @event_recorder
    tags:
        - { name: command_handler, handles: Fully\Qualified\Name\Of\SomeInteresting
```

Recorded events will be handled after the command has been completely handled.

Registering your own message recorders

In case you have another source for recorded message (for instance a class that collects domain events like the `DoctrineORMBridge` does), you can register it as a message recorder:

```
use SimpleBus\Message\Recorder\ContainsRecordedMessages;

class PropelDomainEvents implements ContainsRecordedMessages
{
    public function recordedMessages()
    {
        // return an array of Message instances
    }

    public function eraseMessages()
    {
        // clear the internal array containing the recorded messages
    }
}
```

The corresponding service definition looks like this:

```
services:
    propel_domain_events:
        class: Fully\Qualified\Class\Name\Of\PropelDomainEvents
        public: false
        tags:
            - { name: event_recorder }
```

Note: Logging

If you want to log every event that is being handled, enable logging in `config.yml`:

```
event_bus:
    logging: ~
```

Messages will be logged to the `event_bus` channel.

Doctrine ORM and domain events

As described in the documentation of the SimpleBus/DoctrineORMBridge package library it provides:

- A command bus middleware which wraps the handling of commands inside a database transaction
- A command bus middleware which collects domain events recorded by entities and lets the event bus handle them

Install SimpleBus/DoctrineORMBridge

Before you continue, first install the `simple-bus/doctrine-orm-bridge` package in your project:

```
composer require simple-bus/doctrine-orm-bridge
```

When you enable the `DoctrineORMBridgeBundle` in your project, both features will be automatically registered as command bus middlewares:

```
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            ...
            new SimpleBus\SymfonyBridge\DoctrineOrmBridgeBundle()
        )
        ...
    }
    ...
}
```

You can optionally configure which entity manager and connection should be used:

```
# in config.yml

doctrine_orm_bridge:
    entity_manager: default
    connection: default
```

Upgrade guide

From 3.x to 4.0

Version `4.0` works with `SimpleBus/MessageBus 2.0` so you have to make the changes descibred in its upgrade guide as well.

The biggest change for the `SymfonyBridge` package is that command handler and event subscriber services don't have to have `handle` or `notify` methods respectively:

1. If the services are valid callables already (i.e. they have a public `__invoke()` method), then they are used as they are.
2. If the service has a public `handle()` method, that method will be used.
3. If the service has a public `notify()` method, that method will be used.
4. Otherwise you have to specify which method should be called in the tag attributes:

```
- { name: command_handler, handles: ..., method: theMethodThatShouldBeCalled }

# or
```

```
- { name: event_subscriber, subscribes_to: ..., method: theMethodThatShouldBeCalled }
```

This means that in theory you can now also have one handler handle different commands in different methods, and subscribers which subscribe to multiple events. This is not recommended in most cases, but at least you have this option now.

From 1.0 to 2.0

Commands

Before:

```php
use SimpleBus\Command\Command;

class FooCommand implements Command
{
    public function name()
    {
        return 'foo';
    }
}
```

After:

```php
use SimpleBus\Message\Type\Command;

class FooCommand implements Command
{
    // no name() method anymore
}
```

Or:

```php
use SimpleBus\Message\Type\Command;
use SimpleBus\Message\Name\NamedMessage;

class FooCommand implements Command, NamedMessage
{
    public static function messageName()
    {
        return 'foo';
    }
}
```

See below for more information about this change.

Events

Before:

```php
use SimpleBus\Event\Event;

class BarEvent implements Event
{
    public function name()
    {
```

```
        return 'bar';
    }
}
```

After:

```
use SimpleBus\Message\Type\Event;

class BarEvent implements Event
{
    // no name() method anymore
}
```

Or:

```
use SimpleBus\Message\Type\Event;
use SimpleBus\Message\Name\NamedMessage;

class BarEvent implements Event, NamedMessage
{
    public static function messageName()
    {
        return 'bar';
    }
}
```

See below for more information about this change.

Command handlers

Before:

```
use SimpleBus\Command\Handler\CommandHandler;
use SimpleBus\Command\Command;

class FooCommandHandler implements CommandHandler
{
    public function handle(Command $command)
    {
        ...
    }
}
```

After:

```
use SimpleBus\Message\Handler\MessageHandler;
use SimpleBus\Message\Message;

class FooCommandHandler implements MessageHandler
{
    public function handle(Message $command)
    {
        ...
    }
}
```

You can register this handler like this:

```
services:
    foo_command_handler:
        class: Fully\Qualified\Class\Name\Of\FooCommandHandler
        tags:
            - { name: command_handler, handles: Fully\Qualified\Class\Name\Of\FooCommand }
```

Or, if you let commands implement `NamedMessage`:

```
services:
    foo_command_handler:
        class: Fully\Qualified\Class\Name\Of\FooCommandHandler
        tags:
            - { name: command_handler, handles: foo }
```

Event subscribers

Before:

```
use SimpleBus\Event\Handler\EventHandler;
use SimpleBus\Event\Event;

class BarEventHandler implements EventHandler
{
    public function handle(Event $event)
    {
        ...
    }
}
```

After:

```
use SimpleBus\Message\Subscriber\MessageSubscriber;
use SimpleBus\Message\Message;

class BarEventSubscriber implements MessageSubscriber
{
    public function notify(Message $message)
    {
        ...
    }
}
```

You can register this subscriber like this:

```
services:
    bar_event_subscriber:
        class: Fully\Qualified\Class\Name\Of\BarEventSubscriber
        tags:
            - { name: event_subscriber, subscribes_to: Fully\Qualified\Class\Name\Of\BarEvent }
```

Or, if you let events implement `NamedMessage`:

```
services:
    bar_event_subscriber:
        class: Fully\Qualified\Class\Name\Of\BarEventSubscriber
        tags:
            - { name: event_subscriber, subscribes_to: bar }
```

Named messages

If instead of the FQCN you want to keep using the command/event name as returned by its `messageName()` method, you should configure this in `config.yml`:

```
command_bus:
    # the name of a command is considered to be its FQCN
    command_name_resolver_strategy: class_based

event_bus:
    # the name of an event should be returned by its messageName() method
    event_name_resolver_strategy: named_message
```

This strategy then applies to all your commands or events.

Command and event bus middlewares

Previously you could define your own command bus and event bus behaviors by implementing `CommandBus` or `EventBus`. As of version 2.0 in both cases you should implement `MessageBusMiddleware` instead:

```
use SimpleBus\Message\Bus\Middleware\MessageBusMiddleware;

class SpecializedCommandBusMiddleware implements MessageBusMiddleware
{
    public function handle(Message $message, callable $next)
    {
        // do whatever you want

        $next($message);

        // maybe do some more things
    }
}
```

Please note that the trait `RemembersNext` doesn't exist anymore. Instead of calling `$this->next()` you should now call `$next($message)`.

You should register command bus middleware like this:

```
services:
    specialized_command_bus_middleware:
        class: Fully\Qualified\Class\Name\Of\SpecializedCommandBusMiddleware
        tags:
            - { name: command_bus_middleware, priority: 0 }
```

The same for event bus middleware, but then you should use the tag `event_bus_middleware`. The priority value for middlewares works just like it did before. Read more in the CommandBusBundle and EventBusBundle documentation.

Event providers have become event recorders

If you have entities that collect domain events, you should implement `ContainsRecordedMessages` instead of `ProvidesEvents` and use the trait `PrivateMessageRecorderCapabilities` instead of `EventProviderCapabilities`. The `raise()` method has been renamed to `record()`.

```php
use SimpleBus\Message\Recorder\ContainsRecordedMessages;
use SimpleBus\Message\Recorder\PrivateMessageRecorderCapabilities;

class Entity implements ContainsRecordedMessages
{
    use PrivateMessageRecorderCapabilities;

    public function someFunction()
    {
        // $event is an instance of Message
        $event = ...;

        $this->record($event);
    }
}
```

If you had registered event providers using the service tag `event_provider`, you should change that to `event_recorder`.

Read more about event recorders in the EventBusBundle documentation.

AsynchronousBundle

This bundle integrates async component with the Symfony framework

Install with

```
composer require simple-bus/asynchronous-bundle
```

```php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            ...
            new SimpleBus\AsynchronousBundle\SimpleBusAsynchronousBundle(),
            ...
        );
    }
}
```

Configuration

@TODO Show the standard config

Public services

@TODO What services exists when the bundle is enabled.

Getting started

Introduction

This bundle defines a new command and event bus, to be used for processing asynchronous commands and events. It also adds middleware to existing the command and event buses which publishes messages to be processed asynchronously. See also the documentation of SimpleBus/Asynchronous.

First, enable the `SimpleBusAsynchronousBundle` in your `AppKernel` class.

Provide an object serializer

The first thing you need to do is to provide a service that is able to serialize any object. This service needs to implement `SimpleBus\Serialization\ObjectSerializer`.

```
# in config.yml
simple_bus_asynchronous:
    object_serializer_service_id: your_object_serializer
```

Note: Use an existing object serializer

Instead of creating your own object serializer, you should install the SimpleBus/JMSSerializerBundle. Once you register this bundle in your `AppKernel` as well, it will automatically register itself as the preferred object serializer. So if you do, don't forget to remove the key `simple_bus_asynchronous.object_serializer_service_id` from your config file.

Provide message publishers

Next, you need to define services that are able to publish commands and events, for example to some message queue. These services should both implement `SimpleBus\Asynchronous\Publisher\Publisher`. When you have defined them as services, mention their service id in the configuration:

```
# in config.yml
simple_bus_asynchronous:
    commands:
        publisher_service_id: your_command_publisher
    events:
        publisher_service_id: your_event_publisher
```

Note: Use existing publishers

Instead of writing your own publishers, you can use existing publisher implementations.

As part of SimpleBus a RabbitMQBundle has been provided which automatically registers command and event publishers to publish serialized messages to a RabbitMQ exchange.

Logging

To get some insight into what goes on in the consumer process, enable logging:

```
# in config.yml
simple_bus_asynchronous:
    commands:
        ...
```

```
        logging: ~
    events:
        ...
        logging: ~
```

This will log consumed messages to the `asynchronous_command_bus` and `asynchronous_event_bus` channels respectively.

Choose event strategy

When handling events you have two predefined strategies to choose from. Either you publish *all* events to the message queue (*always* strategy) or you only publish the events that have a registered asynchronous subscriber (*predefined* strategy). If your application is the only one that is consuming messages you should consider using the **predefined** strategy. This will reduce the message overhead on the message queue.

```
simple_bus_asynchronous:
    events:
        strategy: 'predefined' # default: 'always'
```

You can also use Your own strategy by defining custom **strategy_service_id**

```
simple_bus_asynchronous:
    events:
        strategy:
            strategy_service_id: your_strategy_service
```

Using Autowiring

This bundle can be used with Symfony's Autowiring out of the box.

Simply inject `SimpleBus\AsynchronousBundle\Bus\AsyncronousCommandBus` or *SimpleBusAsynchronousBundleBusAsyncronousEventBus* in your service.

RabbitMQBundleBridge

The `SimpleBusRabbitMQBundleBridgeBundle` allows you to publish and consume SimpleBus messages using the OldSoundRabbitMQBundle.

Getting started

First, enable SimpleBusAsynchronousBundle in your Symfony project. Next enable `SimpleBusRabbitMQBundleBridgeBundle` and OldSoundRabbitMqBundle.

Handling commands asynchronously

If you want commands to be handled asynchronously, you should first configure `OldSoundRabbitMqBundle`:

```
# in config.yml
old_sound_rabbit_mq:
    # don't forget to provide the connection details
    ...
    producers:
```

```
        ...
        asynchronous_commands:
            connection:       default
            exchange_options: { name: 'asynchronous_commands', type: direct }
    consumers:
        ...
        asynchronous_commands:
            connection:       default
            exchange_options: { name: 'asynchronous_commands', type: direct }
            queue_options:    { name: 'asynchronous_commands' }
            # use the consumer provided by SimpleBusRabbitMQBundleBridgeBundle
            callback:         simple_bus.rabbit_mq_bundle_bridge.commands_consumer
```

Now enable asynchronous command handling:

```
# in config.yml
simple_bus_rabbit_mq_bundle_bridge:
    commands:
        # this producer service will be defined by OldSoundRabbitMqBundle,
        # its name is old_sound_rabbit_mq.%producer_name%_producer
        producer_service_id: old_sound_rabbit_mq.asynchronous_commands_producer
```

Please note that commands are only handled asynchronously when there is no regular handler defined for it. Instead of registering the handler using the tag `command_handler`, you should now register it using the tag `asynchronous_command_handler`:

```
services:
    my_asynchronous_command_handler:
        class: ...
        tags:
            { name: asynchronous_command_handler, handles: ... }
```

See also the documentation of SimpleBus/AsynchronousBundle.

To actually consume command messages, you need to start (and keep running):

```
php app/console rabbitmq:consume asynchronous_commands
```

Handling events asynchronously

If you want events to be handled asynchronously, you should first configure `OldSoundRabbitMqBundle`:

```
# in config.yml
old_sound_rabbit_mq:
    # don't forget to provide the connection details
    ...
    producers:
        ...
        asynchronous_events:
            connection:       default
            exchange_options: { name: 'asynchronous_events', type: direct }
    consumers:
        asynchronous_events:
            connection:       default
            exchange_options: { name: 'asynchronous_events', type: direct }
            queue_options:    { name: 'asynchronous_events' }
            # use the consumer provided by SimpleBusRabbitMQBundleBridgeBundle
            callback:         simple_bus.rabbit_mq_bundle_bridge.events_consumer
```

Now enable asynchronous event handling:

```
# in config.yml
simple_bus_rabbit_mq_bundle_bridge:
    events:
        # this producer service will be defined by OldSoundRabbitMqBundle,
        # its name is old_sound_rabbit_mq.%producer_name%_producer
        producer_service_id: old_sound_rabbit_mq.asynchronous_events_producer
```

Events are *always handled synchronously as well as asynchronously*. If you want an event subscriber to only be notified of an event asynchronously, instead of registering the subscriber using the tag `event_subscriber` tag, you should now use the `asynchronous_event_subscriber` tag:

```
services:
    my_asynchronous_event_subscriber:
        class: ...
        tags:
            { name: asynchronous_event_subscriber, subscribes_to: ... }
```

To actually consume event messages, you need to start (and keep running):

```
php app/console rabbitmq:consume asynchronous_events
```

Note: You are encouraged to tweak the exchange/queue options and make them right for your project. Read more about your options in the RabbitMQ documentation and in the documentation of OldSoundRabbitMQBundle.

Events

Failure during message consumption

When an exception is thrown while a `Message` is being consumed, the exception is not allowed to bubble up so it won't cause the consumer process to fail. That way, one `Message` that can't be processed is no danger to any other `Message`.

The AMQP message containing the `Message` that caused the failure will be logged, together with the `Exception` that was thrown.

If you want to implement some other error handling behaviour (e.g. storing the message to be published again later), you only need to implement an event subscriber (or listener if you want to) which subscribes to the event `simple_bus.rabbit_mq_bundle_bridge.message_consumption_failed`:

```
use SimpleBus\RabbitMQBundleBridge\Event\Events;
use SimpleBus\RabbitMQBundleBridge\Event\MessageConsumptionFailed;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class MyErrorHandler implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return [Events::MESSAGE_CONSUMPTION_FAILED => 'messageConsumptionFailed'];
    }

    public function messageConsumptionFailed(MessageConsumptionFailed $event)
    {
        $exception = $event->exception();
        $amqpMessage = $event->message();
```

```
        ...
    }
}
```

Don't forget to define a service for it and tag it as `kernel.event_subscriber`:

```
services:
    my_error_handler:
        class: MyErrorHandler
        tags:
            - { name: kernel.event_subscriber }
```

Successful message consumption

When a `Message` has been handled successfully you may want to perform some additional actions. You can do this by creating an event subscriber which subscribes to the `simple_bus.rabbit_mq_bundle_bridge.message_consumed` event:

```
use SimpleBus\RabbitMQBundleBridge\Event\Events;
use SimpleBus\RabbitMQBundleBridge\Event\MessageConsumed;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class MySuccessHandler implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return [Events::MESSAGE_CONSUMED => 'messageConsumed'];
    }

    public function messageConsumed(MessageConsumed $event)
    {
        $amqpMessage = $event->message();
        ...
    }
}
```

Don't forget to define a service for it and tag it as `kernel.event_subscriber`:

```
services:
    my_success_handler:
        class: MySuccessHandler
        tags:
            - { name: kernel.event_subscriber }
```

Routing

By default, this bundle assumes that you want to use "direct" exchanges and use one queue for all commands, and one queue for all events. If you want to use "topic" exchanges and selectively consume messages using a routing key, this bundle can generate routing keys automatically for you based on the class name of the `Message`. Just change the bundle configuration:

```
# in config.yml
simple_bus_rabbit_mq:
    # default value is "empty"
    routing_key_resolver: class_based
```

When for example a `Message` of class `Acme\Command\RegisterUser` is published to the queue, its routing key will be `Acme.Command.RegisterUser`. Now you can define consumers for specific messages, based on this routing key:

```
# in config.yml
old_sound_rabbit_mq:
    ...
    consumers:
        acme_commands:
            connection:       default
            exchange_options: { name: 'asynchronous_commands', type: topic }
            queue_options:    { name: 'asynchronous_commands', routing_keys: ['Acme.Command.#'] }
            callback:         simple_bus.rabbit_mq_bundle_bridge.events_consumer
```

Custom routing keys

If you want to define routing keys in a custom way (not based on the class of a message), create a class that implements `RoutingKeyResolver`:

```
use SimpleBus\RabbitMQBundleBridge\Routing\RoutingKeyResolver;

class MyCustomRoutingKeyResolver implements RoutingKeyResolver
{
    public function resolveRoutingKeyFor($message)
    {
        // determine the routing key for the given Message
        return ...;

        // if you don't want to use a specific routing key, return an empty string
    }
}
```

Now register this class as a service:

```
services:
    my_custom_routing_key_resolver:
        class: MyCustomRoutingKeyResolver
```

Finally, mention your routing key resolver service id in the bundle configuration:

```
# in config.yml
simple_bus_rabbit_mq_bundle_bridge:
    routing_key_resolver: my_custom_routing_key_resolver
```

Fair dispatching

If you are looking for a way to evenly distribute messages over several workers, you may not be better off using a "topic" exchange. Instead, you could just use a "direct" exchange, spin up several workers, and configure consumers to prefetch only one message at a time:

```
# in config.yml
old_sound_rabbit_mq:
    consumers:
        ...
        asynchronous_commands:
            ...
```

```
            qos_options:
                prefetch_count: 1
```

Note: See also Fair dispatching in the bundle's official documentation.

Additional properties

Besides the raw message and a *routing key* the RabbitMQ producer accepts several additional properties. You can determine them dynamically using additional property resolvers. Define your resolvers as a service and tag them as `simple_bus.additional_properties_resolver`:

```
services:
    your_additional_property_resolver:
        class: Your\AdditionalPropertyResolver
        tags:
            - { name: simple_bus.additional_properties_resolver }
```

Optionally you can provide a priority for the resolver. Resolvers with a higher priority will be called first, so if your resolver should have the final say, give it a very low (i.e. negative) priority.