Fusio Documentation

Release 1.0

Christoph Kappestein

Feb 11, 2020

Basics

1 Overview 3
- 1.1 About 3
- 1.2 Why 3
- 1.3 Features 3
- 1.4 Architecture 4
- 1.5 Development 7
- 1.6 Backend 8
- 1.7 Apps 9
- 1.8 Use cases 9

2 Installation 11
- 2.1 Configuration 11
- 2.2 Docker 12
- 2.3 Web server 12
- 2.4 Apps 13
- 2.5 Updating 14

3 Get started 15
- 3.1 Build an API endpoint 15
- 3.2 Access a non-public API endpoint 16

4 Action 19
- 4.1 Engines 19
- 4.2 Examples 21

5 Connection 23
- 5.1 Sql 23
- 5.2 MongoDB 24
- 5.3 HTTP 24
- 5.4 AMQP 24
- 5.5 Beanstalk 25
- 5.6 Cassandra 25
- 5.7 Elasticsearch 25
- 5.8 Memcache 25
- 5.9 Neo4j 26
- 5.10 SOAP 26

6 Business logic 27
- 6.1 Library 27
- 6.2 Microservice 28
- 6.3 DI Container 29

7 Deploy 31
- 7.1 routes 31
- 7.2 schema 32
- 7.3 connection 33

8 Testing 35
- 8.1 Development 35

9 Registration 39
- 9.1 Register 39
- 9.2 Activate 39
- 9.3 Login 40
- 9.4 Provider 40

10 Integration 41
- 10.1 HTTP 41
- 10.2 RPC 41
- 10.3 Message-Queue 41
- 10.4 SQL 42
- 10.5 Include 42

11 SQL Builder 43
- 11.1 Action 43

12 Authorization 47
- 12.1 Simple 47
- 12.2 OAuth2 48

13 Cronjob 49
- 13.1 Installation 49

14 Migration 51
- 14.1 Usage 51
- 14.2 Commands 51

15 Monetization 53
- 15.1 Installation 53
- 15.2 Flow 53
- 15.3 Implementation 54

16 PubSub 55
- 16.1 Installation 55
- 16.2 Subscribe 55
- 16.3 Publish 55
- 16.4 Callback 56

17 Rate limit 57

18 Scope 59

19 Social Login **61**
19.1 Flow 61
19.2 Implementation 62

20 User Attributes **65**

21 SDK Generation **67**
21.1 Generation 67

22 JsonRPC **69**
22.1 Example 69

23 Backend **71**
23.1 Action 71
23.2 App 72
23.3 Config 72
23.4 Connection 73
23.5 Cronjob 73
23.6 Error 73
23.7 Event 73
23.8 Import 75
23.9 Plan 75
23.10 Rate 75
23.11 Routes 75
23.12 Schema 76
23.13 Scope 76
23.14 User 77

24 Consumer **79**
24.1 Authorization code 79
24.2 Implicit 80
24.3 Password 80

25 Documentation **81**

26 Swagger-UI **83**

27 API **85**

28 Adapter **87**
28.1 Provider 87
28.2 Testing 88
28.3 Schema 88

29 Serialization **89**
29.1 Custom writer 89

30 Event **91**
30.1 Implementation 91
30.2 Reference 91

Fusio is an open source API management platform which helps to build and manage RESTful APIs. The documentation covers the basic steps how to develop and maintain an API with Fusio.

CHAPTER 1

Overview

1.1 About

Fusio is an open source API management platform which helps to build and manage RESTful APIs. We think that there is a huge potential in the API economy. Whether you need an API to expose your business functionality, build micro services, develop SPAs or Mobile-Apps. Because of this we think that Fusio is a great tool to simplify building such APIs. More information on https://www.fusio-project.org/

1.2 Why

The originally idea of Fusio was to provide a tool which lets you easily build a great API beside an existing application. I.e. in case you have already a web application on a domain `acme.com` Fusio helps you to build the fitting API at `api.acme.com`. Beside this use case you can also use Fusio to build a new API from scratch or use it internally i.e. for micro services.

To build the API Fusio can connect to many different databases, message queue systems or internal web services. There are also many ways to integrate your business logic into the API of Fusio.

1.3 Features

Fusio covers all important aspects of the API lifecycle so you can concentrate on building the actual business logic of your API.

- **Versioning**

  It is possible to define different versions of your endpoint. A concrete version can be requested through the `Accept` header i.e. `application/vnd.acme.v1+json`

- **Documentation**

  Fusio generates automatically a documentation of the API endpoints based on the provided schema definitions.

- **Validation**

  Fusio uses the standard JSONSchema to validate incoming request data.

- **Authorization**

  Fusio uses OAuth2 for API authorization. Each app can be limited to scopes to request only specific endpoints of the API.

- **Analytics**

  Fusio monitors all API activities and shows them on a dashboard so you always know what is happening with your API.

- **Rate limiting**

  It is possible to limit the requests to a specific threshold.

- **Specifications**

  Fusio generates different specification formats for the defined API endpoints i.e. OpenAPI, Swagger, RAML.

- **Subscription**

  Fusio contains a subscription layer which helps to build pub/sub for your API.

- **User management**

  Fusio provides an API where new users can login or register a new account through GitHub, Google, Facebook or through normal email registration.

- **Logging**

  All errors which occur in your endpoint are logged and are visible at the backend including all information from the request.

- **Connection**

  Fusio provides an adapter system to connect to external services. By default we provide the HTTP and SQL connection type but there are many other types available i.e. MongoDB, Amqp, Cassandra.

- **Migration**

  Fusio has a migration system which allows you to change the database schema on deployment.

- **Testing**

  Fusio provides an api test case wherewith you can test every endpoint response without setting up a local web server.

Basically with Fusio you only have to define the schema (request/response) of your API endpoints and implement the business logic. All other aspects are covered by Fusio.

1.4 Architecture

The basic building block of Fusio is the concept of an action. An action is simply a PHP class which receives a request and returns a response. Around this action Fusio handles all common logic like Authentication, Rate-Limiting, Schema validation, Logging etc. The class has to implement the following signature:

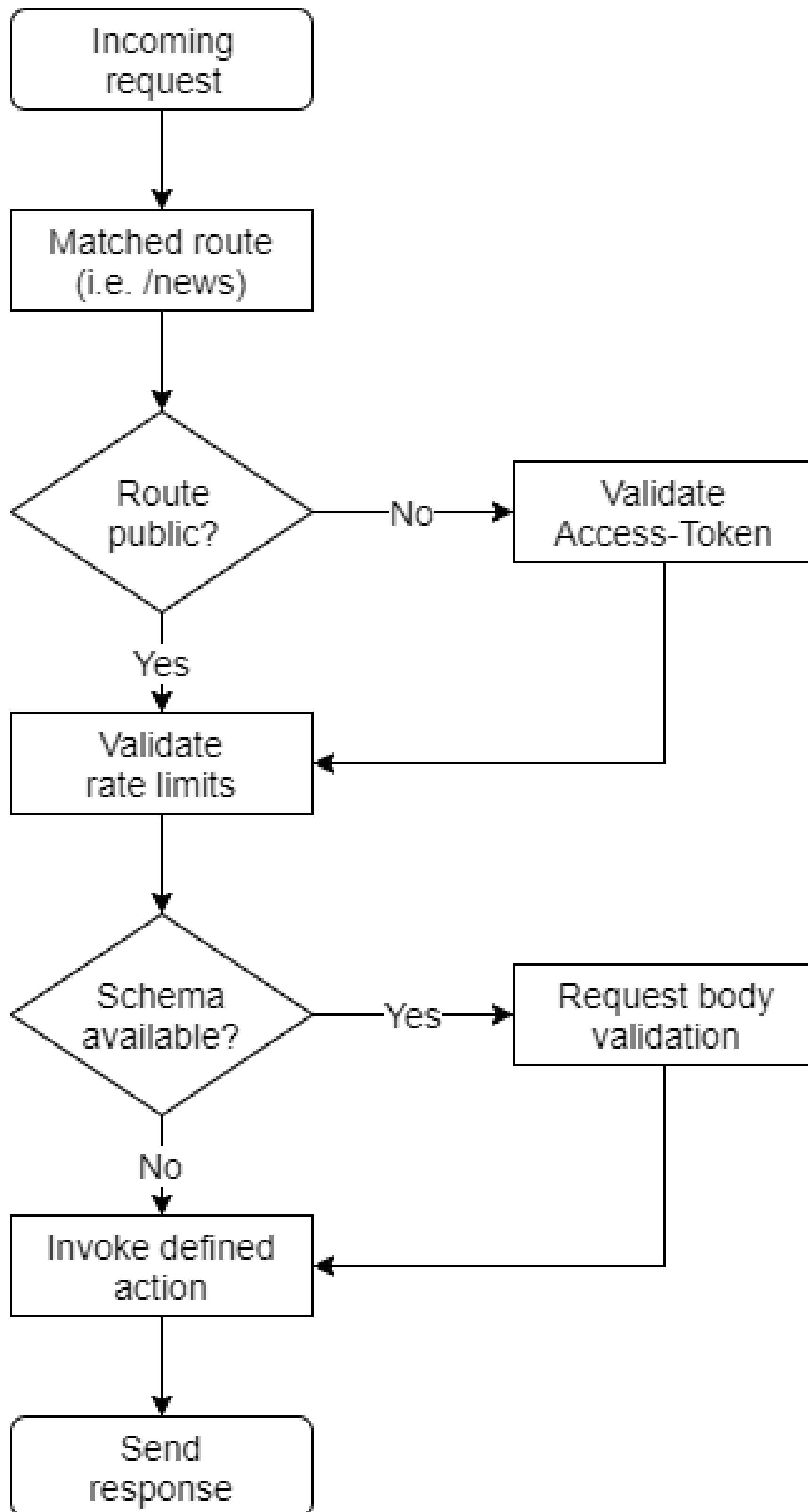
```
<?php

namespace App;

```

(continues on next page)

(continued from previous page)

```
use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;

class HelloWorld extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        return $this->response->build(200, [], [
            'hello' => 'world',
        ]);
    }
}
```

To give you a first overview, every request which arrives at such an action goes through the following lifecycle:



Fusio tries to assign the incoming request to a fitting route. The route contains all schema information about the incoming request and outgoing responses. Those schemas are also used at the documentation which is automatically available. If a request schema was provided the incoming request body gets validated after this schema. In case

everything is ok the action which is assigned to the route gets executed.

An action represents the code which handles an incoming request and produces a response. Each action can use connections to accomplish this task. A connection uses a library which helps to work with a remote service. I.e. the SQL connection uses the Doctrine DBAL library to work with a database (it returns a `Doctrine\DBAL\Connection` instance). A connection always returns a fully configured object so you never have to deal with any credentials in an action. There are already many different actions available which you can use i.e. to create an API based on a database table.

With Fusio we want to remove as many layers as possible so that you can work in your action directly with a specific library. Because of this Fusio has no model or entity system like many other frameworks, instead we recommend to write plain SQL in case you work with a relational database. We think that building API endpoints based on models/entities limits the way how you would design a response. You only need to describe the request and response in the JSON schema format. This schema is then the contract of your API endpoint, how you produce this response technically is secondary. Fusio provides the mentioned connections, which help you to create complete customized responses based on complicated SQL queries, message queue inserts or multiple remote HTTP calls.

1.5 Development

Fusio provides two ways to develop an API. The first way is to build API endpoints only through the backend interface by using all available actions. Through this you can solve already many tasks especially through the usage of the PHP-Sandbox or V8-Processor action.

The other way is to use the deploy mechanism. Through this you can use normal PHP files to implement your business logic and thus you can use the complete PHP ecosystem. Therefor you need to define a `.fusio.yml` deploy file which specifies the available routes and actions of the system. This file can be deployed with the following command:

```
php bin/fusio deploy
```

The action of each route contains the source which handles the business logic. This can be i.e. a php class, a simple php file or a url. More information in the `src/` folder. In the following an example action to build an API response from a database:

```
<?php

namespace App\Todo;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;

class Collection extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        /** @var \Doctrine\DBAL\Connection $connection */
        $connection = $this->connector->getConnection('System');

        $count   = $connection->fetchColumn('SELECT COUNT(*) FROM app_todo');
        $entries = $connection->fetchAll('SELECT * FROM app_todo WHERE status = 1␣
↪ORDER BY insertDate DESC LIMIT 16');

        return $this->response->build(200, [], [
```

(continues on next page)

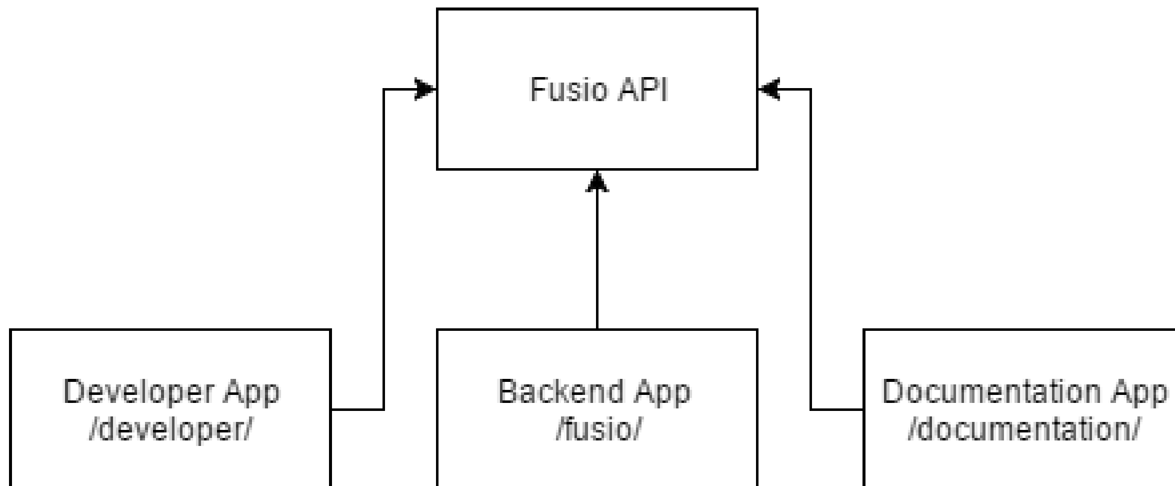
(continued from previous page)

```
            'totalResults' => $count,
            'entry' => $entries,
        ]);
    }
}
```

In the code we get the `System` connection which returns a `\Doctrine\DBAL\Connection` instance but we have already many adapters to connect to different services. Then we simply fire some queries and return the response.

1.6 Backend

Fusio provides several apps which work with the internal backend API. These apps can be used to manage and work with the API. This section gives a high level overview what the Fusio system provides and how the application is structured. Lets take a look at the components which are provided by Fusio:



1.6.1 API

If you install a Fusio system it setups the default API. Through the API it is possible to manage the complete system. Because of that Fusio has some reserved paths which are needed by the system.

- `/backend`

  Endpoints for the system configuration

- `/consumer`

  Endpoints for the consumer i.e. register new accounts or create new apps

- `/doc`

  Endpoints for the documentation

- `/authorization`

  Endpoints for the consumer to get i.e. information about the user itself and to revoke an obtained access token

- `/export`

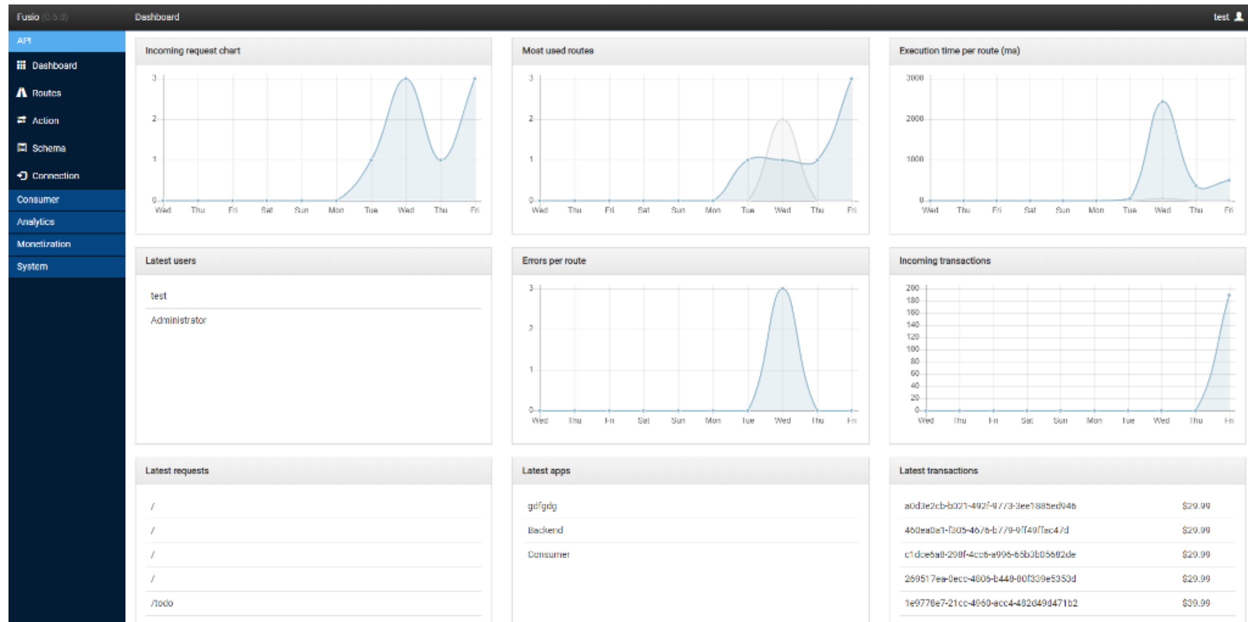
  Endpoints to export the documentation into other formats i.e. swagger

There is also a complete documentation about all internal API endpoints.

1.7 Apps

The following apps are working with the Fusio API.

1.7.1 Backend



The backend app is the app where the administrator can configure the system. The app is located at `/fusio/`.

1.7.2 Marketplace

Fusio has a marketplace which contains a variety of apps for specific use cases. Every app can be directly installed from the backend app under System / Marketplace.

1.8 Use cases

Today there are many use cases where you need a great documented REST API. In the following we list the most popular choices where Fusio comes in to play.

1.8.1 Business functionality

Exposing an API of your business functionality is a great way to extend your product. You enable customers to integrate it into other applications which gives the possibility to open up for new markets. With Fusio you can build such APIs and integrate them seamlessly into your product. We also see many companies which use the API itself as the core product.

1.8.2 Micro services

With Fusio you can simply build small micro services which solve a specific task in a complex system.

1.8.3 Javascript applications

Javascript frameworks like i.e. AngularJS or EmberJS becoming the standard. With Fusio you can easily build a backend for such applications. So you dont have to build the backend part by yourself.

1.8.4 Mobile apps

Almost all mobile apps need some form to interact with a remote service. This is mostly done through REST APIs. With Fusio you can easily build such APIs which then can also be used by other applications.

CHAPTER 2

Installation

It is possible to install Fusio either through composer or manually file download. Place the project into the www directory of the web server.

Composer

```
composer create-project fusio/fusio
```

Download

https://github.com/apioo/fusio/releases

2.1 Configuration

You can either manually install Fusio with the steps below or you can also use the browser based installer at `public/install.php`. Note because of security reasons it is highly recommended to remove the installer script after the installation.

- **Adjust the configuration file**

  Open the file `.env` in the Fusio directory and change the key `FUSIO_URL` to the domain pointing to the public folder. Also insert the database credentials to the `FUSIO_DB_*` keys.

- **Execute the installation command**

  The installation script inserts the Fusio database schema into the provided database. It can be executed with the following command `php bin/fusio install`.

- **Create administrator user**

  After the installation is complete you have to create a new administrator account. Therefor you can use the following command `php bin/fusio adduser`. Choose as account type "Administrator".

You can verify the installation by visiting the `psx_url` with a browser. You should see a API response that the installation was successful. The backend is available at `/fusio/`.

In case you want to install Fusio on a specific database you need to adjust the `driver` parameter at the `configuration.php` file:

- `pdo_mysql`: MySQL
- `pdo_sqlite`: SQLite
- `pdo_pgsql`: PostgreSQL
- `sqlsrv`: Microsoft SQL Server
- `oci8`: Oracle
- `sqlanywhere`: SAP Sybase SQL Anywhere

2.2 Docker

Alternatively it is also possible to setup a Fusio system through docker. This has the advantage that you automatically get a complete running Fusio system without configuration. This is especially great for testing and evaluation. To setup the container you have to checkout the repository and run the following command:

```
docker-compose up -d
```

This builds the Fusio system with a predefined backend account. The credentials are taken from the env variables `FUSIO_BACKEND_USER`, `FUSIO_BACKEND_EMAIL` and `FUSIO_BACKEND_PW` in the docker-compose.yml. If you are planing to run the container on the internet you MUST change these credentials.

2.3 Web server

It is recommended to setup a virtual host in your `sites-available` folder which points to the public folder of Fusio. After this you also have to change the configuration of the url i.e.:

```
'psx_url' => 'http://api.acme.com',
```

2.3.1 Apache

```
<VirtualHost *:80>
    ServerName api.acme.com
    DocumentRoot /var/www/html/fusio/public

    <Directory /var/www/html/fusio/public>
        Options FollowSymLinks
        AllowOverride All
        Require all granted

        # rewrite
        RewriteEngine On
        RewriteBase /

        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteRule (.*) /index.php/$1 [L]

```

(continues on next page)

(continued from previous page)

```
        RewriteCond %{HTTP:Authorization} ^(.*)
        RewriteRule .* - [e=HTTP_AUTHORIZATION:%1]
    </Directory>

    # log
    LogLevel warn
    ErrorLog ${APACHE_LOG_DIR}/fusio.error.log
    CustomLog ${APACHE_LOG_DIR}/fusio.access.log combined
</VirtualHost>
```

You should enable the module `mod_rewrite` so that the .htaccess file in the public folder is used. It is also possible to include the htaccess commands directly into the virtual host which also increases performance. The htaccess contains an important rule which redirects the `Authorization` header to Fusio which is otherwise removed. If the .htaccess file does not work please check whether the `AllowOverride` directive is set correctly i.e. to `All`.

2.3.2 Shared-Hosting

If you want to run Fusio on a shared-hosting environment it is possible but in general not recommended since you can not properly configure the web server and access the CLI. Therefore you can not use the deploy command which simplifies development. The biggest problem of a shared hosting environment is that you can not set the document root to the `public/` folder. If you place the following `.htaccess` file in the directory you can bypass this problem by redirecting all requests to the `public/` folder.

```
RewriteEngine On
RewriteBase /fusio/

RewriteCond %{THE_REQUEST} /public/([^\s?]*) [NC]
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ public/index.php/$1 [L,QSA]
```

While this may work many shared hosting provider have strict limitations of specific PHP functions which are maybe used by Fusio and which produce other errors.

2.3.3 cPanel

On cPanel you can create a new sub-domain and use the "base document path" option to point it at `/public`. Also you need to set the `psx_dispatch` key at the `configuration.php` to `''` and adjust the url at the `.env` file to your correct domain.

2.4 Apps

2.4.1 Backend

At the endpoint `fusio/index.html` you can login to the backend app. You should be able to login with the username (which you have entered for the `adduser` command) and the password which you have used. The following list covers the most login errors in case you are not able to login at the backend:

- **The javascript Backend-App uses the wrong API endpoint**

  This can be tested with the browser developer console. If you login at the backend with no credentials the app should make an request to the `/backend/token` endpoint which should return a JSON response i.e.:

```
{ "error": "invalid_request", "error_description": "Credentials not available" }
```

If this is the case your app is correctly configured. If this is not the case you need to adjust the endpoint url at `/public/fusio/index.htm` i.e.:

```
var fusioUrl = "http://localhost:8080/fusio/public/index.php/";
```

- **Apache module mod_rewrite is not activated**

  In case you use Apache as web server you must activate the module `mod_rewrite` so that the `public/.htaccess` file is used. Besides clean urls it contains an important rule which tells Apache to redirect the `Authorization` header to Fusio otherwise Apache will remove the header and Fusio can not authenticate the user

- **Fusio API returns an error**

  In this case Fusio can probably not write to the `cache/` folder. To fix the problem you have to change the folder permissions so that the user of the web server can write to the folder. If there is another error message it is maybe a bug. Please report the issue to GitHub.

2.4.2 Marketplace

Fusio has a marketplace which contains a variety of apps for specific use cases. Every app can be directly installed from the backend app under System / Marketplace.

2.5 Updating

There are two parts of Fusio which you can update. The backend system and the backend app. The backend app is the AngularJS application which connects to the backend api and where you configure the system. The backend system contains the actual backend code providing the backend API and the API which you create with the system.

2.5.1 Server

Fusio makes heavy use of composer. Because of that you can easily upgrade a Fusio system with the following composer command.

```
composer update fusio/impl
```

This has also the advantage that the version constraints of installed adapters are checked and in case something is incompatible composer will throw an error. It is also possible to simply replace the vendor folder with the folder from the new release. In either case you have to run the following command after you have updated the vendor folder:

```
php bin/fusio install
```

This gives Fusio the chance to adjust the database schema in case something has changed with a new release.

2.5.2 App

To update the backend app simply replace the javascript and css files from the new release:

- `public/fusio/`

CHAPTER 3

Get started

This chapter helps you to quickly create your first API with Fusio. In general there are two ways how you can use Fusio:

- You can use Fusio to build the complete API through the backend UI. This means you use it more like a CMS to build your API.
- You can use Fusio as framework to build your API. This means you can develop your endpoint logic in simple PHP classes and define your route meta data in simple YAML files.

In this example we build our first API using Fusio as a framework.

3.1 Build an API endpoint

Fusio provides a demo todo API which is ready for deployment. Take a look at the `.fusio.yml` file which contains the deployment configuration. The file contains several keys:

- **routes**

  Describes for each route the available request methods, whether the endpoint is public or private, the available request/response schema and also the action which should be executed:

  ```
  "/todo": !include resources/routes/todo/collection.yaml
  "/todo/:todo_id": !include resources/routes/todo/entity.yaml
  ```

- **schema**

  Contains the available request and response schema in the JSON-Schema format:

  ```
  Todo: !include resources/schema/todo/entity.json
  Todo-Collection: !include resources/schema/todo/collection.json
  Message: !include resources/schema/message.json
  ```

- **connection**

  Provides connections to a remote service i.e. mysql or mongodb. This connection can be used inside an action:

```
Default-Connection:
  class: Fusio\Adapter\Sql\Connection\SqlAdvanced
  config:
    url: "sqlite:///${dir.cache}/todo-app.db"
```

Through the command `php bin/fusio deploy` you can deploy the API. It is now possible to visit the API endpoint at: `/todo`.

3.2 Access a non-public API endpoint

The POST method of the todo API is not public, because of this you need an access token in order to send a POST request.

- **Assign the scope to your user**

  By default all routes are assigned to the `todo` scope. In order to use a scope, the scope must be assigned to your user account. Therefor go to the user panel click on the edit button and assign the `todo` scope to your user. It is also possible to set the default scopes for new users under settings `scopes_default`.

- **Request a JWT**

  Now you can obtain a JWT through a simple HTTP request to the `consumer/login` endpoint.

```
POST /consumer/login HTTP/1.1
Host: 127.0.0.1
Content-Type: application/json

{
  "username": "[username]",
  "password": "[password]"
}
```

  Which returns a token i.e.:

```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJzdWIiOiI5N2JkNDUzYjdlMDZlOWFlMDQxNi00YmY2MWFiYjg4MDJjZmRmOWZmN2UyNDg4OTNmNzYyYmU5Njc5MGUzYT
↪T49Af5wnPIFYbPer3rOn-KV5PcN0FLcBVykUMCIAuwI"
}
```

  Note this generates an OAuth2 token with contains all scopes from your user account. It is also possible to use the OAuth2 endpoint */authorization/token* to create an access token with specific assigned scopes.

- **Request the non-public API endpoint**

  Now we can use the JWT as Bearer token in the `Authorization` header to access the protected endpoint.

```
POST /todo HTTP/1.1
Host: 127.0.0.1
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJzdWIiOiI5N2JkNDUzYjdlMDZlOWFlMDQxNi00YmY2MWFiYjg4MDJjZmRmOWZmN2UyNDg4OTNmNzYyYmU5Njc5MGUzYT
↪T49Af5wnPIFYbPer3rOn-KV5PcN0FLcBVykUMCIAuwI
Content-Type: application/json

{
  "title": "lorem ipsum",
```

(continues on next page)

(continued from previous page)

```
  "content": "lorem ipsum"
}
```

CHAPTER 4

Action

The src/ folder contains the action code which is executed if a request arrives at an endpoint which was specified in the .fusio.yml deploy file. Fusio determines the engine based on the provided action string. The following engines are available:

4.1 Engines

4.1.1 PHP File

```
action: "${dir.src}/Todo/collection.php"
```

If the action points to a file with a php file extension Fusio simply includes this file. In the following an example implementation:

```
<?php
/**
 * @var \Fusio\Engine\ConnectorInterface $connector
 * @var \Fusio\Engine\ContextInterface $context
 * @var \Fusio\Engine\RequestInterface $request
 * @var \Fusio\Engine\Response\FactoryInterface $response
 * @var \Fusio\Engine\ProcessorInterface $processor
 * @var \Psr\Log\LoggerInterface $logger
 * @var \Psr\SimpleCache\CacheInterface $cache
 */

// @TODO handle request and return response

$response->build(200, [], [
    'message' => 'Hello World!',
]);
```

4.1.2 HTTP Url

```
action: "http://foo.bar"
```

If the action contains an `http` or `https` url the request gets forwarded to the defined endpoint. Fusio automatically adds some additional headers to the request which may be used by the endpoint i.e.:

```
X-Fusio-Route-Id: 72
X-Fusio-User-Anonymous: 1
X-Fusio-User-Id: 4
X-Fusio-App-Id: 3
X-Fusio-App-Key: 1ba7b2e5-fa1a-4153-8668-8a855902edda
X-Fusio-Remote-Ip: 127.0.0.1
```

4.1.3 Static file

```
action: "${dir.src}/static.json"
```

If the action points to a simple file Fusio will simply forward the content to the client. This is helpful if you want to build fast an sample API with dummy responses.

4.1.4 PHP Class

```
action: "App\\Todo\\CollectionAction"
```

If the action string is a PHP class Fusio tries to autoload this class through composer. The class must implement the `Fusio\Engine\ActionInterface`. This is the most advanced solution since it is also possible to access services from the DI container. In the following an example implementation:

```php
<?php

namespace App\Todo;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;

class CollectionAction extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        // @TODO handle request and return response

        return $this->response->build(200, [], [
            'message' => 'Hello World!',
        ]);
    }
}
```

4.2 Examples

Please take a look at the recipes section of our website, there we have example code to complete a specific task i.e. send a HTTP request or query data from a database.

CHAPTER 5

Connection

Inside an action you can use the `connector` to obtain an configured connection. The following list contains connection classes which you can use. Note some connections depend on PHP extensions or other client libraries, you have to install the fitting adapter in order to use the connection. Take a look at the adapter page for an overview of available adapters.

5.1 Sql

Connects to a SQL database using the doctrine DBAL library.

Class `Fusio\Adapter\Sql\Connection\Sql`

Return `Doctrine\DBAL\Connection`

Website http://www.doctrine-project.org/projects/dbal.html

API http://www.doctrine-project.org/api/dbal/2.5/class-Doctrine.DBAL.Connection.html

config

`type` The driver which is used to connect to the database

- `pdo_mysql` = MySQL
- `pdo_pgsql` = PostgreSQL
- `sqlsrv` = Microsoft SQL Server
- `oci8` = Oracle Database
- `sqlanywhere` = SAP Sybase SQL Anywhere

`host` The IP or hostname of the database server

`username` The name of the database user

`password` The password of the database user

`database` The name of the database which is used upon connection

5.2 MongoDB

Connects to a MongoDB using the official MongoDB library. Note this requires the PHP `mongodb` extension.

Class `Fusio\Adapter\Mongodb\Connection\MongoDB`

Return `MongoDB\Database`

Website https://github.com/mongodb/mongo-php-library

API https://docs.mongodb.com/php-library/master/reference/class/MongoDBDatabase/

config

`url` The url must have the following format `mongodb://[username:password@]host1[:port1][,host2[:port2:],...]/db`

`options` It is possible to provide option parameters. The options must be url encoded i.e. `connect=1&fsync=1`

`database` The name of the database which is used upon connection

5.3 HTTP

Uses the Guzzle library to send HTTP requests.

Class `Fusio\Adapter\Http\Connection\Http`

Return `GuzzleHttp\Client`

Website http://docs.guzzlephp.org/en/latest/

config

`url` HTTP base url

`username` Optional username for authentication

`password` Optional password for authentication

`proxy` Optional HTTP proxy

5.4 AMQP

Provides a client to send messages to a RabbitMQ.

Class `Fusio\Adapter\Amqp\Connection\Amqp`

Return `PhpAmqpLib\Connection\AMQPStreamConnection`

Website https://github.com/php-amqplib/php-amqplib

config

`host` The IP or hostname of the RabbitMQ server

`port` The port used to connect to the AMQP broker. The port default is 5672

`user` The login string used to authenticate with the AMQP broker

`password` The password string used to authenticate with the AMQP broker

`vhost` The virtual host to use on the AMQP broker

5.5 Beanstalk

Provides a client to send messages to a Beanstalkd.

Class `Fusio\Adapter\Beanstalk\Connection\Beanstalk`

Return `Pheanstalk\Pheanstalk`

Website https://github.com/pda/pheanstalk

config

`host` The IP or hostname of the Beanstalk server

`port` Optional the port of the Beanstalk server

5.6 Cassandra

Connects to a Cassandra database using the official PHP library. Requires the `cassandra` PHP extension.

Class `Fusio\Adapter\Cassandra\Connection\Cassandra`

Return `Cassandra\Session`

Website https://github.com/datastax/php-driver

API http://datastax.github.io/php-driver/api/Cassandra/interface.Session/

config

`host` Configures the initial endpoints. Note that the driver will automatically discover and connect to the rest of the cluster

`port` Specify a different port to be used when connecting to the cluster

`keyspace` Optional keyspace name

5.7 Elasticsearch

Connects to a Elasticsearch database using the official PHP library.

Class `Fusio\Adapter\Elasticsearch\Connection\Elasticsearch`

Return `Elasticsearch\Client`

Website https://github.com/elastic/elasticsearch-php

config

`host` Comma separated list of elasticsearch hosts i.e. `192.168.1.1:9200,192.168.1.2`

5.8 Memcache

Uses the native PHP `memcached` extension to connect to a memcache server.

Class `Fusio\Adapter\Memcache\Connection\Memcache`

Return `Memcached`

Website http://php.net/manual/de/book.memcached.php

config

host Comma seperated list of [ip]:[port] i.e. `192.168.2.18:11211,192.168.2.19:11211`

5.9 Neo4j

Connects to a Neo7j graph database using the official PHP library.

Class `Fusio\Adapter\Neo4j\Connection\Neo4j`

Return `GraphAware\Neo4j\Client\ClientInterface`

Website https://github.com/graphaware/neo4j-php-client

config

uri URI of the connection i.e. `http://neo4j:password@localhost:7474`

5.10 SOAP

Provides a client to send SOAP requests.

Class `Fusio\Adapter\Soap\Connection\Soap`

Return `SoapClient`

Website http://php.net/manual/de/class.soapclient.php

config

wsdl Location of the WSDL specification

location Required if no WSDL is available

uri Required if no WSDL is available

version Optional SOAP version

- `1` = SOAP 1.1
- `2` = SOAP 1.2

username Optional username for authentication

password Optional password for authentication

CHAPTER 6

Business logic

In case your API is not a simple CRUD app you probably need to execute some more complex business logic. This chapter shows options how you can organize this business logic for simple reuse.

In general an action should not contain any business logic it should simply forward the data to the fitting library or service. It is equivalent to a controller in a classical framework environment. If an action contains too many lines of code or you copy specific code snippets into different actions it is a smell to extract this logic into an external service. Inside an action you can then reuse this external service.

Fusio is designed to help you write framework independent code. That means that all services which you develop are complete free of Fusio specific code so you can simply reuse those components in another context.

6.1 Library

The simplest solution is to move business logic into a separate PHP class. This class can be autoloaded through composer. You can place this class either directly into the `src/` folder or develop a custom PHP package and require this package through composer.

In general a library should work with a specific connection. The following example shows a simple custom logger implementation which you could use in different actions.

```php
<?php

$connection = $connector->get('Mysql-1');

$myLogger = new MyLogger($connection);
$myLogger->log('A new log entry');
```

A simple implementation of the logger could look like:

```php
<?php

namespace Acme\MyLib;
```

(continues on next page)

(continued from previous page)

```
use Doctrine\DBAL\Connection;

class MyLogger
{
    /**
     * @var \Doctrine\DBAL\Connection
     */
    protected $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function log($message)
    {
        $this->connection->insert('my_table', [
            'message' => $message,
        ]);
    }
}
```

6.2 Microservice

If your business logic is more complex or has specific performance requirements you could also develop it as an external microservice. This has the advantage that service is completely decoupled from your app and it is also possible to use a complete different language. Usually you can talk to the micro service through HTTP. But it would be also possible to use a different protocol i.e. an AMQP connection to use a message queue.

```
<?php

$client = $connector->get('Http-1');

$myClient = new MyClient($client);
$myClient->send(['foo' => 'bar']);
```

A simple client implementation could look like:

```
<?php

namespace Acme\MyLib;

use GuzzleHttp\Client;

class MyClient
{
    /**
     * @var \GuzzleHttp\Client
     */
    protected $client;

    public function __construct(Client $client)
    {
```

(continues on next page)

(continued from previous page)

```
        $this->client = $client;
    }

    public function send($data)
    {
        $this->client->post('http://foo.bar/my_service', [
            'json' => $data
        ]);
    }
}
```

6.3 DI Container

Fusio uses a DI container to manage all internal services. You can also use this internal DI container in your action to access Fusio specific functions. It is also possible to extend the container with custom services. There for you need to add your service to the `container.php` file:

```
<?php

$container = new Fusio\Impl\Dependency\Container();
$container->setParameter('config.file', __DIR__ . '/configuration.php');

$container->set('my_service', function($c){
    return new MyService();
});

return $container;
```

To access this service in your action you need to use the following PHP action class. Note we do not recommend to rely heavily on the DI container instead use the technique describe in the chapter above to develop platform independent services which can be reused across multiple actions and applications.

```
<?php

namespace App;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;
use Fusio\Engine\Factory\ContainerAwareInterface;
use Psr\Container\ContainerInterface;

class Endpoint extends ActionAbstract implements ContainerAwareInterface
{
    protected $container;

    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        $myService = $this->container->get('my_service');

        $data = $myService->doSomething();
```

(continues on next page)

(continued from previous page)

```
        return $this->response->build(200, [], [
            'hello' => $data,
        ]);
    }

    public function setContainer(ContainerInterface $container)
    {
        $this->container = $container;
    }
}
```

This works only in case you use a PHP class as action. For normal PHP files and Javascript files it is not possible to access the DI container.

CHAPTER 7

Deploy

The `.fusio.yml` deploy file is the main configuration file to develop an API with Fusio. This chapter explains in detail the format.

7.1 routes

A route is the rule which redirects the incoming request to an action. If a request arrives the first route which matches is used. In order to be able to evolve an API it is possible to add multiple versions for the same route. For each version it is possible to specify the allowed request methods. Each method describes the request and response schema and the action which is executed upon request. If a request method is public it is possible to request the API endpoint without an access token.

```yaml
version: 1
methods:
  GET:
    public: true
    response: Todo-Collection
    action: "${dir.src}/Todo/collection.php"
  POST:
    public: false
    request: Todo
    response: Todo-Message
    action: "${dir.src}/Todo/insert.php"
```

The `request` and `response` key reference a schema name which was defined under the `schema` key. It is also possible to use the `Passthru` schema which simply redirects all data. The `action` key reference an action.

7.1.1 Path

The path can contain variable path fragments. It is possible to access these variable path fragments inside an action. The following list describes the syntax.

- `/news` No variable path fragment only the request to `/news` matches this route
- `/news/:news_id` Simple variable path fragment. This route matches to any value except a slash. I.e. `/news/foo` or `/news/12` matches this route
- `/news/$year<[0-9]+>` Variable path fragment with a regular expression. I.e. only `/news/2015` matches this route
- `/file/*path` Variable path fragment which matches all values. I.e. `/file/foo/bar` or `/file/12` matches this route

7.1.2 Status

Beside the `version` every route can also have a `status` field. By default the status is set to 4 (Development). If you change the status to 1 (Production) it is not longer possible to change the API endpoint through the backend. The following list describes each status

- `4 = Development` Used as first status to develop a new API endpoint. It adds a "Warning" header to each response that the API is in development mode.
- `1 = Production` Used if the API is ready for production use. If the API transitions from development to production all databases settings are copied into the route. That means changing a schema or action will not change the API endpoint.
- `2 = Deprecated` Used if you want to deprecate a specific version of the API. Adds a "Warning" header to each response that the API is deprecated.
- `3 = Closed` Used if you dont want to support a specific version anymore. Returns an error message with a `410 Gone` status code

7.2 schema

The schema defines the format of the request and response data. It uses the JsonSchema format. Inside a schema it is possible to refer to other schema definitions by using the `$ref` key and the `file` protocol i.e. `file:///[file]`.

```
{
    "id": "http://acme.com/schema",
    "type": "object",
    "title": "schema",
    "properties": {
        "name": {
            "type": "string"
        },
        "author": {
            "$ref": "file:///author.json"
        },
        "date": {
            "type": "string",
            "format": "date-time"
        }
    }
}
```

7.3 connection

A connection provides a class which helps to connect to another service.

```yaml
Acme-Mysql:
  class: Fusio\Adapter\Sql\Connection\Sql
  config:
    type: pdo_mysql
    host: localhost
    username: root
    password: test
    database: fusio
```

Please take a look at the *Connection* overview to see all available connection and return types.

CHAPTER 8

Testing

Fusio provides a complete Test-Setup for your API endpoints. For the test case we use an in-memory sqlite database which contains the schema defined in the `resources/migration` folder. In the `Fixture.php` class it is also possible to define fixture data which is inserted for every test case.

The idea is that each endpoint has a corresponding test case class which tests the GET, POST, PUT and DELETE method of the resource. Internally we can send an HTTP request to Fusio without the need to setup an HTTP server. This makes these tests very fast and efficient.

The Method `Fixture::getPhpUnitDataSet` returns the data set which is inserted for every test case. There we insert a fixed access token with the fitting rights so that we can call our protected API endpoints.

You can execute those tests inside the Fusio directory root with a simple phpunit command (because of the available `phpunit.xml` configuration):

```
phpunit
```

8.1 Development

Every test case should extend from the `ApiTestCase` class. The test case contains a test method for every HTTP method i.e. `testGet`, `testPost`, etc. Every test makes the appropriated call to the API endpoint. Then we assert the response body and if needed also the headers (for larger response bodies it is recommended to move the expected JSON payload to an external file which is then included i.e. through `file_get_contents`). Through this way we can simply assure that our API works as expected. The following shows a simple API test case from the example todo entity API endpoint:

```php
<?php

class EntityTest extends ApiTestCase
{
    public function testGet()
    {
        $response = $this->sendRequest('/todo/4', 'GET', [
```

(continues on next page)

(continued from previous page)

```
            'User-Agent'    => 'Fusio TestCase',
        ]);

        $actual = (string) $response->getBody();
        $actual = preg_replace('/\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}/', '0000-00-00␣
↪00:00:00', $actual);
        $expect = <<<'JSON'
{
    "id": "4",
    "status": "1",
    "title": "Task 4",
    "insertDate": "0000-00-00 00:00:00"
}
JSON;

        $this->assertEquals(200, $response->getStatusCode(), $actual);
        $this->assertJsonStringEqualsJsonString($expect, $actual, $actual);
    }

    public function testPost()
    {
        $response = $this->sendRequest('/todo/4', 'POST', [
            'User-Agent'    => 'Fusio TestCase',
        ]);

        $actual = (string) $response->getBody();
        $expect = <<<'JSON'
{
    "success": false,
    "title": "Internal Server Error",
    "message": "Given request method is not supported"
}
JSON;

        $this->assertEquals(405, $response->getStatusCode(), $actual);
        $this->assertJsonStringEqualsJsonString($expect, $actual, $actual);
    }

    public function testPut()
    {
        $response = $this->sendRequest('/todo/4', 'PUT', [
            'User-Agent'    => 'Fusio TestCase',
        ]);

        $actual = (string) $response->getBody();
        $expect = <<<'JSON'
{
    "success": false,
    "title": "Internal Server Error",
    "message": "Given request method is not supported"
}
JSON;

        $this->assertEquals(405, $response->getStatusCode(), $actual);
        $this->assertJsonStringEqualsJsonString($expect, $actual, $actual);
    }
```

(continues on next page)

(continued from previous page)

```
    public function testDelete()
    {
        $response = $this->sendRequest('/todo/4', 'DELETE', [
            'User-Agent'    => 'Fusio TestCase',
            'Authorization' => 'Bearer␣
↪da250526d583edabca8ac2f99e37ee39aa02a3c076c0edc6929095e20ca18dcf'
        ]);

        $actual = (string) $response->getBody();
        $expect = <<<'JSON'
{
    "success": true,
    "message": "Delete successful"
}
JSON;

        $this->assertEquals(200, $response->getStatusCode(), $actual);
        $this->assertJsonStringEqualsJsonString($expect, $actual, $actual);

        /** @var \Doctrine\DBAL\Connection $connection */
        $connection = Environment::getService('connector')->getConnection('Default-
↪Connection');
        $actual = $connection->fetchAssoc('SELECT id, status, title FROM app_todo␣
↪WHERE id = 4');
        $expect = [
            'id' => 4,
            'status' => 0,
            'title' => 'Task 4',
        ];

        $this->assertEquals($expect, $actual);
    }

    public function testDeleteWithoutAuthorization()
    {
        $response = $this->sendRequest('/todo/4', 'DELETE', [
            'User-Agent'    => 'Fusio TestCase',
        ]);

        $actual = (string) $response->getBody();
        $expect = <<<'JSON'
{
    "success": false,
    "title": "Internal Server Error",
    "message": "Missing authorization header"
}
JSON;

        $this->assertEquals(401, $response->getStatusCode(), $actual);
        $this->assertJsonStringEqualsJsonString($expect, $actual, $actual);
    }
}
```

CHAPTER 9

Registration

Fusio contains a registration system which can be used by you app to provide a secure registration and social login. This chapter explains how you can embed the registration system of Fusio in your app. In general it is important to note that Fusio provides only APIs so you have to create the UI (i.e. a javascript app) and call the fitting Fusio API endpoint.

9.1 Register

At the registration process the user needs to provide a name, email and password to create a new user account. In case you have configured a `RECAPTCHA_SECRET` at your `.env` file Fusio checks also the `captcha` value. The captcha secret must be a google recaptcha secret.

If the user has provided the values at your UI you need to call the /consumer/register endpoint. If everything is valid Fusio creates a new user account (which is deactivated) and sends a confirmation mail to the provided email address.

The content of the activation mail can be customized at the settings panel of the Fusio backend. There is a setting `mail_register_body` which can be changed. If you are using the deploy mechanism you can also modify the `resources/config.yaml` file.

We host also a sample developer app which contains a signup form.

9.2 Activate

The activation mail contains a link to activate the account. The link must point to your app, then your app needs to call the Fusio /consumer/activate endpoint to activate the account. In this way you can provide the user an UI which is in your look and feel.

The activation link contains the token from the url which you must pass to the endpoint. If everything is valid Fusio activates the user account.

9.3 Login

The login endpoint uses a username and password and returns a token which can be used for any subsequent API requests to authenticate the user. To login a user you need to call the /consumer/login endpoint.

Optional you can also provide a list of scopes so that the user can only access specific parts of your API.

9.4 Provider

Besides the normal registration it is also possible to use a remote provider i.e. Google or Github to handle registration. Through this way users dont need to create a separate account instead they can use an existing account to login.

To use such a social login you need to start the OAuth2 authentication flow and call the /consumer/provider/[provider] endpoint if the user comes back from the provider.

Fusio then calls the provider from the backend and checks whether this is a valid user and gets additional user information. If everything went well the method returns a token which can be used in any subsequent API calls. For more information how to implement your own provider please take a look at the *Social Login* chapter.

CHAPTER 10

Integration

Fusio is often used to create a REST API beside an existing web app. This chapter describes best practices how you can integrate your app without ignoring the business logic.

At first we should distinguish between read and write requests. A read request is a request which does not modify the state (database) of your app. For this case you can also connect directly to your app database. A write request modifies the state (database) of your app i.e. it creates a new record. In this case you most likely want to run the business logic of your app so that all data gets validated and all depending mechanisms are executed. For this case there are multiple ways to run your business logic:

10.1 HTTP

Your app provides an internal API which gets called by Fusio. In this case your action uses a HTTP connection to call the internal API of your app. The internal API also does not need to have a great design since the user only faces the Fusio endpoints. I.e. you could create a simple `api.php` script which bootstraps your app and invokes a specific method.

10.2 RPC

Your app provides an RPC service (i.e. Apache Thrift or GRPC) which can be called by Fusio. This has also the advantage that the performance is much better then an internal HTTP API because modern RPC services mostly serialize the data into an optimized binary format instead of JSON or XML.

10.3 Message-Queue

Your app provides a message queue which Fusio can use. This has also great performance but it is a unidirectional connection, this means the message queue can never return a response to Fusio. In most cases the response message must be defined in the action. Fusio has connections to connect to a AMQP or Beanstalk message queue.

10.4 SQL

In case you have no additional business logic which needs to be executed you can also directly connect to the database and insert a new entry.

10.5 Include

Another (but not recommended) solution is to include your app bootstrap code inside an action. This is possible but then you are mixing the context between your existing app and Fusio. In most cases it is recommended to use one of the approaches described above. But for some small apps it might be also feasible since this has basically no performance penalty.

CHAPTER 11

SQL Builder

Fusio contains a SQL builder which helps to create nested results based on flat SQL results. The following example shows a complex action from an internal project which uses the SQL builder to create a nested JSON response.

11.1 Action

```
<?php

namespace App\Storagesystem;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;
use PSX\Sql\Builder;
use PSX\Sql\Condition;

class Collection extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        /** @var \Doctrine\DBAL\Connection $connection */
        $connection = $this->connector->getConnection('System');

        $startIndex = (int) $request->getParameter('startIndex');
        $startIndex = $startIndex <= 0 ? 0 : $startIndex;
        $condition  = $this->getCondition($request);
        $builder    = new Builder($connection);

        $sql = 'SELECT article.id,
                       article.status,
                       article.articleNumber,
```

(continues on next page)

(continued from previous page)

```
                        article.articleStock,
                        article.description,
                        article.comment,
                        article.removeDate,
                        article.insertDate,
                        location.id AS locationId,
                        location.name AS locationName,
                        agroup.id AS groupId,
                        agroup.name AS groupName,
                        supplier.id AS supplierId,
                        supplier.name AS supplierName
                   FROM app_article article
             INNER JOIN app_location location
                     ON article.locationId = location.id
             INNER JOIN app_group agroup
                     ON article.groupId = agroup.id
             INNER JOIN app_supplier supplier
                     ON article.supplierId = supplier.id
                  WHERE article.status = 1
                    AND ' . $condition->getExpression($connection->
↪getDatabasePlatform()) . '
               ORDER BY article.insertDate DESC
                  LIMIT :startIndex, 32';

        $parameters = array_merge($condition->getValues(), ['startIndex' =>
↪$startIndex]);
        $definition = [
            'totalResults' => $builder->doValue('SELECT COUNT(*) AS cnt FROM app_
↪article WHERE status = 1', [], $builder->fieldInteger('cnt')),
            'startIndex' => $startIndex,
            'entries' => $builder->doCollection($sql, $parameters, [
                'id' => $builder->fieldInteger('id'),
                'location' => [
                    'id' => $builder->fieldInteger('locationId'),
                    'name' => 'locationName',
                ],
                'group' => [
                    'id' => $builder->fieldInteger('groupId'),
                    'name' => 'groupName',
                ],
                'supplier' => [
                    'id' => $builder->fieldInteger('supplierId'),
                    'name' => 'supplierName',
                ],
                'articleNumber' => 'articleNumber',
                'articleStock' => $builder->fieldInteger('articleStock'),
                'description' => 'description',
                'comment' => 'comment',
                'insertDate' => $builder->fieldDateTime('insertDate'),
                'links' => [
                    'self' => $builder->fieldReplace('/article/{id}'),
                ]
            ])
        ];

        return $this->response->build(200, [], $builder->build($definition));
    }
```

(continues on next page)

(continued from previous page)

```
    private function getCondition(RequestInterface $request)
    {
        $fields = [
            'location.name' => 'location',
            'agroup.name' => 'group',
            'supplier.name' => 'supplier',
            'article.articleNumber' => 'articleNumber',
            'article.description' => 'description',
            'article.serialNumber' => 'serialNumber',
            'article.comment' => 'comment',
        ];

        $condition = new Condition();
        foreach ($fields as $columnName => $parameterName) {
            $parameter = $request->getParameter($parameterName);
            if (!empty($parameter)) {
                $condition->like($columnName, '%' . $parameter . '%');
            }
        }

        return $condition;
    }
}
```

CHAPTER 12

Authorization

If your API exposes protected endpoints you need a way to authorize your call. At the core Fusio uses OAuth2 for authorization. This means you need to create an access token to be able to request the API. This access token has always an expire time adn can be revoked.

12.1 Simple

The most simple way to obtain an access token is to use the /consumer/login endpoint. If you need more control of your access token you should use the Oauth2 endpoint to obtain an access token.

12.1.1 Request

```
POST /consumer/login
Host: 127.0.0.1
Content-Type: application/json

{
  "username": "[username]",
  "password": "[password]"
}
```

12.1.2 Response

```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJzdWIiOiI5N2JkNDUzYjdlMDZlOWFlMDQxNi00YmY2MWFiYjg4MDJjZmRmOWZmN2UyNDg4OTNmNzYyYmU5Njc5MGUzYTk4NDQ
↪T49Af5wnPIFYbPer3rOn-KV5PcN0FLcBVykUMCIAuwI"
}
```

12.2 OAuth2

Fusio provides an OAuth2 endpoint to obtain an access token. The endpoint supports the following flows:

- Authorization Code
- Resource Owner Password Credentials
- Client Credentials

The following example shows how to obtain an access token using the client credentials grant. Which grant you should use always depends on whether your client is confidential or public. If your client is confidential this means you can securely store a client id and secret.

12.2.1 Request

```
POST /authorization/token
Host: 127.0.0.1
Authorization: Basic
↪NmM2MTM5NDUtOGQ1YS00YTBkLWI2NjAtMDlkZTVmYmRiNzUzOjMxZTA5M2Y5OGVhZDIyZWZjMjFiMzhhODdhMmE1YmQ3MWZjMTJ
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&scope=authorization,backend
```

12.2.2 Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjFcL3Byb2plY3RzXC9mdXNpb1wvcHVibGljIiwic3ViIjoiZTZjYTI4YWEtY2M4Ny0
↪9PYOaFkE0Qsnt5EUf-JF-73kBAiq8SVF495bjvo_eM0",
  "token_type": "bearer",
  "expires_in": 1553280735,
  "refresh_token": "65e95c8da122a0a5522f-
↪534b054a029019548036c8253d591309247d2899223a6a7b-907deae7ff",
  "scope": "authorization"
}
```

To extend an existing token you can use the refresh token grant i.e.:

12.2.3 Request

```
POST /authorization/token
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=65e95c8da122a0a5522f-
↪534b054a029019548036c8253d591309247d2899223a6a7b-907deae7ff&client_id=6c613945-8d5a-
↪4a0d-b660-09de5fbdb753&client_
↪secret=31e093f98ead22efc21b38a87a2a5bd71fc12bede3379f5eae6c6f7da9dabc5f
```

CHAPTER 13

Cronjob

When building an API it is sometimes required that specific code gets executed at a certain interval. For this purpose Fusio provides the cronjobs.

To execute those cronjobs Fusio uses the standard linux `cron` util. It can write a dedicated cron file which contains all needed cron entries. This file should be placed in the `cron.d` folder of your system.

13.1 Installation

To tell Fusio that it should write such a cron file you need to create the file where the `fusio_cron_file` setting points to i.e. `/etc/cron.d/fusio`. The web user needs also write access to this file since Fusio always updates the file in case you create or update an entry. You can change those settings at the `configuration.php` file.

CHAPTER 14

Migration

Fusio uses the Doctrine Migrations system. Fusio has basically two components which are using the migration system. At first Fusio it self uses it to install the internal schema and execute updates on a new release. Then it is also possible for the API developer to use the migration system on any remote SQL connection.

14.1 Usage

For example to install Fusio you execute the following command:

```
php /bin/fusio migration:migrate
```

which then executes all migrations on the database which was configured at the `.env` file. You may have also executed the `install` command but this is simply an alias to the `migrate` command.

To use the migration in your app you can provide a connection option which tells Fusio which connection should be used. This can be any SQL connection which you have configured at Fusio.

```
php /bin/fusio migration:migrate -connection=System
```

Fusio will look into the `src/Migrations/` folder and search for a folder which has the name of your connection. In this case it would be "System". Then it will execute all migrations which are not already executed.

14.2 Commands

Fusio supports the following migration commands:

```
migration:execute      Execute a single migration version up or down manually.
migration:generate     Generate a blank migration class.
migration:latest       Outputs the latest version number
migration:migrate      [install] Execute a migration to a specified version or the␣
↪latest available version.
```

(continues on next page)

(continued from previous page)

```
migration:status      View the status of a set of migrations.
migration:up-to-date  Tells you if your schema is up-to-date.
migration:version     Manually add and delete migration versions from the version␣
↪table.
```

CHAPTER 15

Monetization

Fusio helps you to monetize your API. It has a concept of points which each user can buy. A user can then spend those points by calling specific routes which cost a specific amount of points. The API developer can simply add a cost to every route and request method.

15.1 Installation

At first you need to create a plan at the Fusio backend. A plan has a name, a specific amount of points and a price assigned.

Then you need to configure a payment provider. For this you need to include i.e. the paypal adapter which configures paypal as payment provider.

```
composer require fusio/adapter-paypal
php bin/fusio system:register "Fusio\Adapter\Paypal\Adapter"
```

Then you need to create a new connection at the Fusio backend. This connection must be named "paypal" and you need to provide your app credentials.

15.2 Flow

If a user of your API wants to obtain points he has to use a configured payment provider. To start the payment process your app has to send a POST request to the `/consumer/transaction/prepare/paypal` endpoint (in this example we use paypal as provider) with the following payload:

```json
{
  "planId": 1,
  "returnUrl": "http://my-app.com/payment/return?transaction_id={transaction_id}"
}
```

The `planId` is the id of a plan which was configured at the backend. The return url is the url of your app where the user returns after the payment was completed. If everything is valid the endpoint returns an approval url of the payment provider:

```
{
  "approvalUrl": ""
}
```

Your app has to simply redirect the user to this approval url. Then the user authenticates at the payment provider and approves the payment. Then the user gets redirected to the `/consumer/transaction/execute/{transaction_id}` endpoint where Fusio checks whether the payment was accepted. If yes Fusio credits the amount of points to the user.

Then it redirects the user to the return url which was provided in the initial prepare call. You app can then lookup the status of the transaction and display a fitting message.

15.3 Implementation

It is also easy to implement a custom payment provider. It is important that the provider supports a redirect based flow. It is currently not possible to simply enter the credit card number. To create a new payment provider you need to create a class which implements the `Fusio\Engine\Payment\ProviderInterface`

CHAPTER 16

PubSub

Fusio has an event system which helps to build a pub/sub system. This means consumer of your API can subscribe to specific events. Inside your API endpoint you can then trigger such an event. Fusio will then send the payload to each subscriber in the background.

16.1 Installation

To enable Fusio to send messages in the background you need to setup a cronjob which executes the HTTP requests. The cronjob must execute the "event:execute" i.e.:

```
php /bin/fusio event:execute
```

16.2 Subscribe

To subscribe for such an event the user needs to send a HTTP POST request to the `/consumer/subscription` endpoint with the following payload:

```
{
  "event": "my_event",
  "endpoint": "http://my-app.com/callback"
}
```

16.3 Publish

To publish an event you need to use the dispatcher to create an event. I.e. the following action shows how to dispatch data to the event "my_event" which then will send the provided data JSON encoded to the subscribed endpoints.

```php
<?php

namespace App\Todo;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;

class Collection extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        // dispatch my event
        $this->dispatcher->dispatch('my_event', [
            'foo' => 'bar'
        ]);
    }
}
```

16.4 Callback

Fusio will send the following HTTP POST request to every subscribed endpoint. In case the endpoint returns a non successful status code Fusio will try to resend the message up to 3 times.

```
POST /callback HTTP/1.1
Host: my-app.com
Content-Type: application/json
User-Agent: Fusio/4.0.2@916a81045349cc0e149873b5b794777bb5f29a30

{"foo": "bar"}
```

CHAPTER 17

Rate limit

Through a rate it is possible to limit the amount of incoming requests to a threshold. If the threshold is reached the user receives a 429 HTTP status code. A rate can distinguish between authenticated and not authenticated calls. For authenticated calls the request count is based on the app for not authenticated calls it is based on the ip address.

CHAPTER 18

Scope

A scope describes the right to access specific routes and request methods. Each user account has assigned a set of allowed scopes. If a user creates an app he can only assign the scopes which are available for him.

CHAPTER 19

Social Login

Fusio provides a developer portal where consumers of your API can register and create their apps. Besides the traditional sign-up via email and password Fusio provides a system to allow 3rd party providers. By default Fusio supports:

- Facebook
- Google
- Github

But it is also easy possible to add other providers. The provider must support OAuth2 in order to work with Fusio.

19.1 Flow

The javascript app starts the authentication process by redirecting the user to the provider. I.e. the developer app uses the AngularJS satellizer module to start this process. If the user returns, your app needs to send a POST request to the endpoint `/consumer/provider/google` providing the following payload:

```
{
  "code": "",
  "clientId": ""
  "redirectUri": ""
}
```

Then on the server side Fusio will try to obtain an access token using the code and client id. Fusio knows also the client secret of the provider which you need to provide at the `.env` file. If this was successful Fusio tries to get some additional information about the user (this step depends always on the remote provider how you get information about the user).

If everything went fine Fusio creates a new "remote" user entry (if the id does not already exists) and returns directly an JWT which can be used in any subsequent API calls:

```
{
  "token": ""
}
```

19.2 Implementation

If you want to add a new provider you need to create a class which implements the `Fusio\Engine\User\ProviderInterface`. Then you need to register this class in your `provider.php` file. To give you an example how such a provider might look please take a look at our Google provider:

```
<?php

namespace Fusio\Impl\Provider\User;

use Fusio\Engine\Model\User;
use Fusio\Engine\User\ProviderAbstract;
use Fusio\Impl\Base;
use PSX\Http\Client\GetRequest;
use PSX\Http\Client\PostRequest;
use PSX\Json\Parser;
use PSX\Uri\Url;
use RuntimeException;

/**
 * Google
 */
class Google extends ProviderAbstract
{
    /**
     * @inheritdoc
     */
    public function getId()
    {
        return self::PROVIDER_GOOGLE;
    }

    /**
     * @inheritdoc
     */
    public function requestUser($code, $clientId, $redirectUri)
    {
        $accessToken = $this->getAccessToken($code, $clientId, $this->secret,
↪$redirectUri);

        if (!empty($accessToken)) {
            $url      = new Url('https://www.googleapis.com/plus/v1/people/me/
↪openIdConnect');
            $headers  = [
                'Authorization' => 'Bearer ' . $accessToken,
                'User-Agent'    => Base::getUserAgent()
            ];

            $response = $this->httpClient->request(new GetRequest($url, $headers));

            if ($response->getStatusCode() == 200) {
                $data  = Parser::decode($response->getBody());
                $id    = isset($data->sub) ? $data->sub : null;
                $name  = isset($data->name) ? $data->name : null;
                $email = isset($data->email) ? $data->email : null;

                if (!empty($id) && !empty($name)) {
```

(continues on next page)

(continued from previous page)

```
                    $user = new User();
                    $user->setId($id);
                    $user->setName($name);
                    $user->setEmail($email);

                    return $user;
                }
            }
        }

        return null;
    }

    protected function getAccessToken($code, $clientId, $clientSecret, $redirectUri)
    {
        if (empty($clientSecret)) {
            throw new RuntimeException('No secret provided');
        }

        $url = new Url('https://accounts.google.com/o/oauth2/token');

        $params = [
            'code'          => $code,
            'client_id'     => $clientId,
            'client_secret' => $clientSecret,
            'redirect_uri'  => $redirectUri,
            'grant_type'    => 'authorization_code'
        ];

        $headers = [
            'Accept'     => 'application/json',
            'User-Agent' => Base::getUserAgent()
        ];

        $response = $this->httpClient->request(new PostRequest($url, $headers,
↪$params));

        if ($response->getStatusCode() == 200) {
            $data = Parser::decode($response->getBody());
            if (isset($data->access_token)) {
                return $data->access_token;
            }
        }

        return null;
    }
}
```

CHAPTER 20

User Attributes

If you build an app and use the authorization system of Fusio you might want to extend the default user table with additional columns i.e. first- and last name (by default the user table contains only the most basic fields: username, email and password). Fusio has a system called user attributes which let you easily add arbitrary attributes to each user account. Therefor you need to define the allowed attributes in the file `configuration.php` i.e.:

```
'fusio_user_attributes'    => [
    'first_name',
    'last_name',
],
```

Then it is possible to update the defined properties for each user by sending an PUT request to the `/consumer/account` endpoint. This changes the attributes of the current authenticated user.

```
{
  "email": "foo@bar.com",
  "attributes": {
    "first_name": "Sebastian",
    "last_name": "Bach",
  }
}
```

This allows you to build a complete custom user profile page which contains all fields for a user account which you need.

CHAPTER 21

SDK Generation

You can help your users to build great apps based on your API by providing an SDK. Building such SDKs is always a time consuming task and they can get easily out dated if your write them by hand. Fuiso can help you to build a great SDK for your API by automatically generating code for your API based on the provided schema definitions. This chapter explains how you can generate such code. You can also take a look at the Fusio javascript SDK which also uses the same approach.

21.1 Generation

```
php bin/fusio api:generate -f typescript output
```

This command generates for every route a typescript file inside the output folder which contains a class to call the endpoint.

If you are not happy with the generated code, you can take a look at the psx-api and psx-schema libraries which are responsible for the code generation. There you can build your own custom generator or provide suggestions to existing code generators.

CHAPTER 22

JsonRPC

Fusio contains a JsonRPC endpoint at `/export/jsonrpc` which can be used to execute directly a specific method. The method name is identified by the "Operation-Id" of each route method. Through the endpoint it is possible to execute multiple methods within a single request.

In case your method needs authorization you need to add an authorization header i.e. `Authorization: Bearer [token]` to the RPC call.

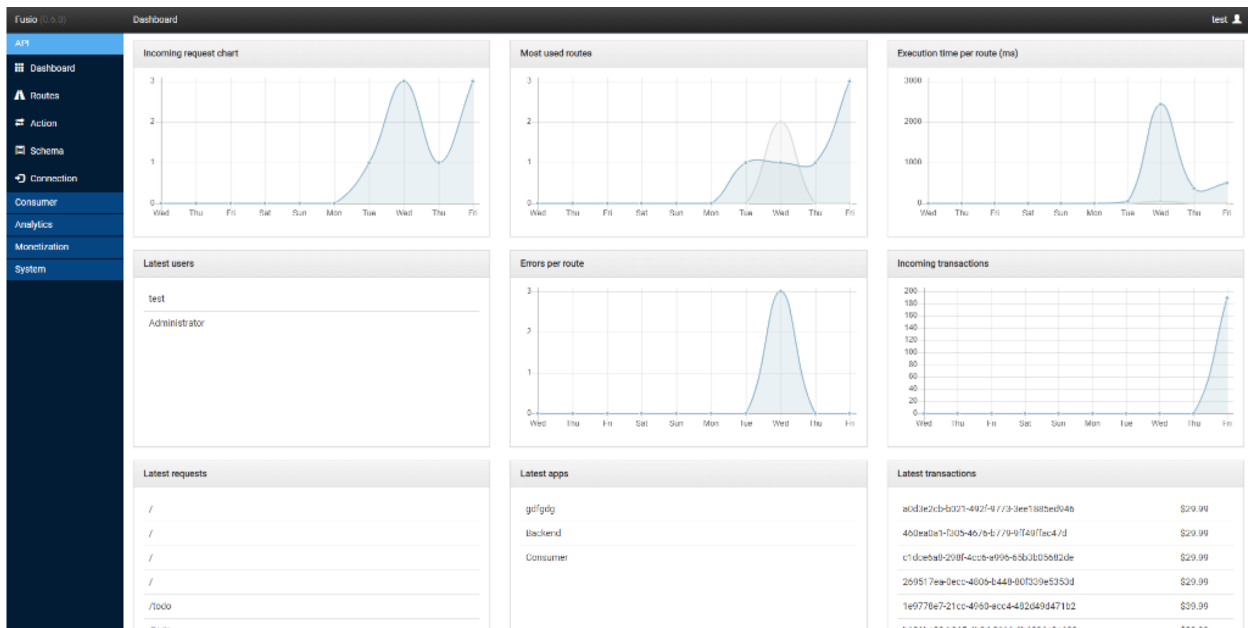
22.1 Example

The following code shows a simple example how to talk to the JsonRPC endpoint using a JsonRPC client

```php
<?php

require __DIR__ . '/vendor/autoload.php';

use Graze\GuzzleHttp\JsonRpc\Client;

$client = Client::factory('http://myapi.com/export/jsonrpc');

$responses = $client->sendAll([
    $client->request(1, 'my.operation.id', [
        'uriFragments'=> ['foo' => 'bar'],
        'parameters'=> ['foo' => 'bar'],
        'headers'=> ['foo' => 'bar'],
        'body'=> ['foo' => 'bar'],
    ]),
]);

foreach ($responses as $resp) {
    var_dump((string) $resp->getBody());
}
```

CHAPTER 23

Backend



This chapter contains information about the backend. The following pages are extracted from the online help which is also available inside the backend app:

23.1 Action

The action contains the logic to handle the request and produce a response. Each action is based on a class and can have specific parameters. Fusio contains already some actions for common tasks i.e. to execute database operations or push data to a message queue.

23.1.1 Development

To develop a custom action you need to create a class which implements the interface `Fusio\Engine\ActionInterface`. Then you can add this class to the `provider.php` file. Thus it is possible to select this action at the backend. Please take a look at the `src/` folder for more examples.

23.2 App

An app enables the consumer to request an access token through the app key and secret. With the access token it is possible to request protected API endpoints. There is a default consumer implementation located at `developer/` which enables a user to manage their apps. The consumer can use any OAuth2 client to request an access token. Fusio supports by default the `authorization_code`, `implicit` and `password` grant type. Please take a look at the OAuth2 RFC for more information about the flow.

23.2.1 Authorization code

At first you have to redirect the client to the consumer endpoint containing the app key, redirect uri and the needed scopes i.e.: `/developer/auth?response_type=code&client_id=[key]&redirect_uri=[url]&scope=foo,bar`. After the user has authenticated he approves or denies the access. If he accepts the user gets redirected to the provided `redirect_uri`. Note the `redirect_uri` must have the same host as the url which was provided for the app. The callback contains a GET parameter `code` which can be exchanged for an access token at the `/authorization/token` endpoint.

23.2.2 Implicit

Mostly used for javascript apps. Like in the authorization code flow the app redirects the user to the consumer endpoint i.e.: `/developer/auth?response_type=token&client_id=[key]&redirect_uri=[url]&scope=foo,bar` If the user has authenticated and approved the app the user gets redirected to the `redirect_uri`. The callback contains the access token in the fragment component. The access tokens which are issued through the implicit grant have usually a much shorter life time because they are more insecure. It is also possible to deactivate the implicit grant through the configuration.

23.2.3 Password

A user can use the password grant to obtain directly an access token with their username and password. Therefor he has to send a direct request to the `/authorization/token` endpoint.

23.3 Config

The config contains system wide settings. In the following we explain some important settings which you most likely need to configure.

- `mail_register_body`If a new user registers through the consumer app he receives an activation mail. Through this setting you can configure the text and adjust the activtaion url
- `recaptcha_secret`If provided the consumer registration can show an google recaptcha which prevents automatic registration. You also have to provide the recaptcha public key to the consumer app

- `scopes_default`Those are the scopes which are assigned by default if a new user registers
- `provider_facebook_secret provider_github_secret provider_google_secret`If provided a user can login through those remote providers. You also have to provide the app key to the consumer app

23.4 Connection

A connection enables Fusio to connect to other remote sources. This can be i.e. a database or message queue server. Please take a look at the adapter to see a list of all available connections. It is also easy possible to develop your own custom connection.

23.4.1 Development

To develop a custom connection you need to create a class which implements the interface `Fusio\Engine\ConnectionInterface`. Then you can add this class to the `provider.php` file. Thus it is possible to select this connection at the backend.

23.5 Cronjob

When building an API it is sometimes required that specific code gets executed at a certain interval. For this purpose Fusio provides the cronjobs.

To execute those cronjobs Fusio uses the standard linux `cron` util. It can write a dedicated cron file which contains all needed cron entries. This file should be placed in the `cron.d` folder of your system.

23.5.1 Installation

To tell Fusio that it should write such a cron file you need to create the file where the `fusio_cron_file` setting points to i.e. `/etc/cron.d/fusio`. The web user needs also write access to this file since Fusio always updates the file in case you create or update an entry. You can change those settings at the `configuration.php` file.

23.6 Error

Lists all errors which occurred on the endpoints. The detail view shows the message and the stack trace of the error.

23.7 Event

Fusio has an event system which helps to build a pub/sub system. This means consumer of your API can subscribe to specific events. Inside your API endpoint you can then trigger such an event. Fusio will then send the payload to each subscriber in the background.

23.7.1 Installation

To enable Fusio to send messages in the background you need to setup a cronjob which executes the HTTP requests. The cronjob must execute the `event:execute` command i.e.:

```
php /bin/fusio event:execute
```

23.7.2 Subscribe

To subscribe for such an event the user needs to send a HTTP POST request to the `/consumer/subscription` endpoint with the following payload:

```
{
  "event": "my_event",
  "endpoint": "http://my-app.com/callback"
}
```

23.7.3 Publish

To publish an event you need to use the dispatcher to create an event. I.e. the following action shows how to dispatch data to the event `my_event` which then will send the provided data JSON encoded to the subscribed endpoints.

```
<?php

namespace App\Todo;

use Fusio\Engine\ActionAbstract;
use Fusio\Engine\ContextInterface;
use Fusio\Engine\ParametersInterface;
use Fusio\Engine\RequestInterface;

class Collection extends ActionAbstract
{
    public function handle(RequestInterface $request, ParametersInterface
↪$configuration, ContextInterface $context)
    {
        // dispatch my event
        $this->dispatcher->dispatch('my_event', [
            'foo' => 'bar'
        ]);
    }
}
```

23.7.4 Callback

Fusio will send the following HTTP POST request to every subscribed endpoint. In case the endpoint returns a non successful status code Fusio will try to resend the message up to 3 times.

```
POST /callback HTTP/1.1
Host: my-app.com
Content-Type: application/json
User-Agent: Fusio/4.0.2@916a81045349cc0e149873b5b794777bb5f29a30
```

(continues on next page)

(continued from previous page)

```
{"foo": "bar"}
```

23.8 Import

The importer provides a way to import route and schema definitions. The data must be in the OpenAPI, RAML or Swagger format. The importer displays a preview what data is imported before any changes are made.

23.9 Plan

A plan can be purchased by a user to obtain points. Those points can then be used to call specific API endpoints. At the backend it is possible to set a specific cost for each route. Every plan has a name, description, price and points. The developer app contains already a possibility to buy those plans.

To enable payments you need to setup a payment provider (i.e. paypal). Through the payment provider the user can purchase such plans. Please take a look at the manual to see how to setup a payment provider:

23.10 Rate

Through a rate it is possible to limit the amount of incoming requests to a threshold. If the threshold is reached the user receives a 429 HTTP status code. A rate can distinguish between authenticated and not authenticated calls. For authenticated calls the request count is based on the app for not authenticated calls it is based on the ip address.

23.11 Routes

A route is the rule which redirects the incoming request to an action. If a request arrives the first route which matches is used. In order to be able to evolve an API it is possible to add multiple versions for the same route. For each version it is possible to specify the allowed request methods. Each method describes the request and response schema and the action which is executed upon request. If a request method is public it is possible to request the API endpoint without an access token.

23.11.1 Path

The path can contain variable path fragments. It is possible to access these variable path fragments inside an action. The following list describes the syntax.

- `/news` No variable path fragment only the request to `/news` matches this route
- `/news/:news_id` Simple variable path fragment. This route matches to any value except a slash. I.e. `/news/foo` or `/news/12` matches this route
- `/news/$year<[0-9]+>` Variable path fragment with a regular expression. I.e. only `/news/2015` matches this route
- `/file/*path` Variable path fragment which matches all values. I.e. `/file/foo/bar` or `/file/12` matches this route

23.11.2 Status

The status affects the behaviour of the API endpoint. The following list describes each status

- `Development` Used as first status to develop a new API endpoint. It adds a "Warning" header to each response that the API is in development mode.
- `Production` Used if the API is ready for production use. If the API transitions from development to production all databases settings are copied into the route. That means changing a schema or action will not change the API endpoint.
- `Deprecated` Used if you want to deprecate a specific version of the API. Adds a "Warning" header to each response that the API is deprecated.
- `Closed` Used if you dont want to support a specific version anymore. Returns an error message with a `410 Gone` status code

23.11.3 Action

The action contains the business logic of your API endpoint. It i.e. selects or inserts entries from a database or pushes a new entry to a message queue.

23.12 Schema

The schema defines the format of the request and response data. It uses the JsonSchema format. Inside a schema it is possible to refer to other schema definitions by using the $ref key and the schema protocol i.e. schema:///[schema-name]. More detailed information about the json schema format at the RFC.

23.12.1 Example

```
{
    "id": "http://acme.com/schema",
    "type": "object",
    "title": "schema",
    "properties": {
        "name": {
            "type": "string"
        },
        "author": {
            "$ref": "schema:///author"
        },
        "date": {
            "type": "string",
            "format": "date-time"
        }
    }
}
```

23.13 Scope

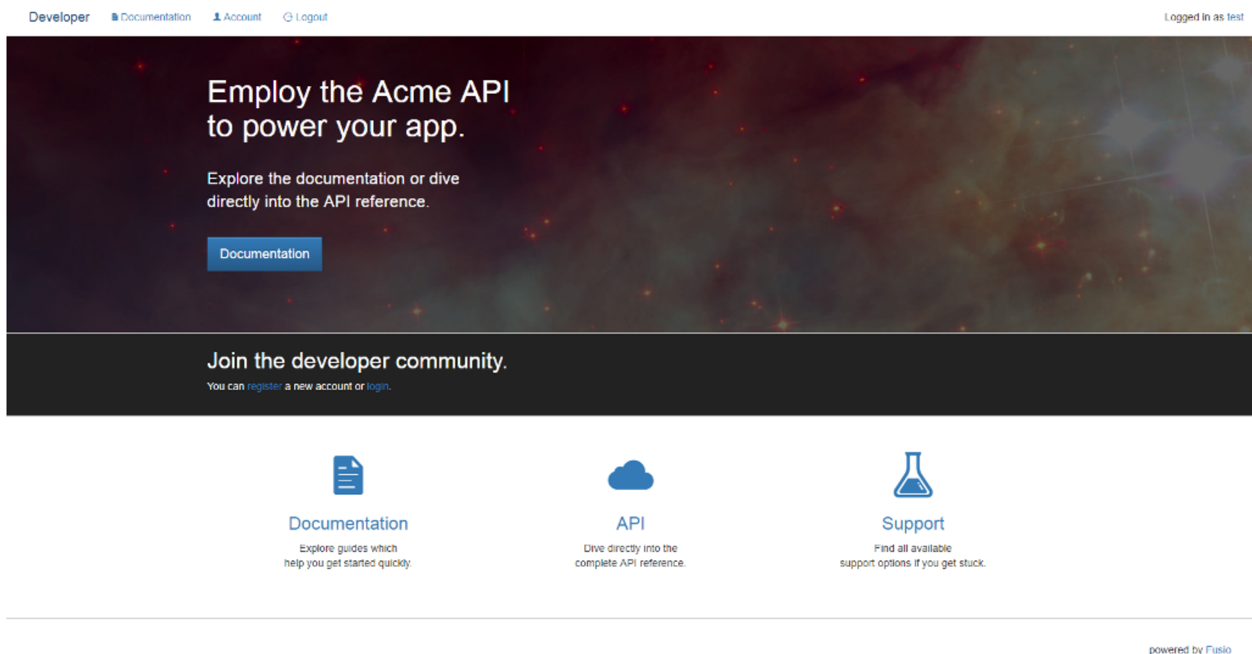
A scope describes the right to access specific routes and request methods. Each user account has assigned a set of allowed scopes. If a user creates an app he can only assign the scopes which are available for him.

23.14 User

A user is either a `Consumer` which uses the API or an `Administrator` which manages the API through the backend. An Administrator account can request an access token for the backend API. Fusio has a simple consumer app located at `/developer` where a user can manage all app settings.

CHAPTER 24

Consumer



Fusio provides a default consumer implementation located at `/developer` which provides a basic admin panel to manage and authorize apps. It is also possible to integrate it into an existing application. In the following an explanation how to authorize an app.

24.1 Authorization code

At first you have to redirect the client to the consumer endpoint containing the app key, redirect uri and the needed scopes i.e.: `/developer/auth?`

`response_type=code&client_id=[key]&redirect_uri=[url]&scope=foo,bar`. After the user has authenticated he approves or denies the access. If he accepts the user gets redirected to the provided `redirect_uri`. Note the `redirect_uri` must have the same host as the url which was provided for the app. The callback contains a GET parameter `code` which can be exchanged for an access token at the `/authorization/token` endpoint.

24.2 Implicit

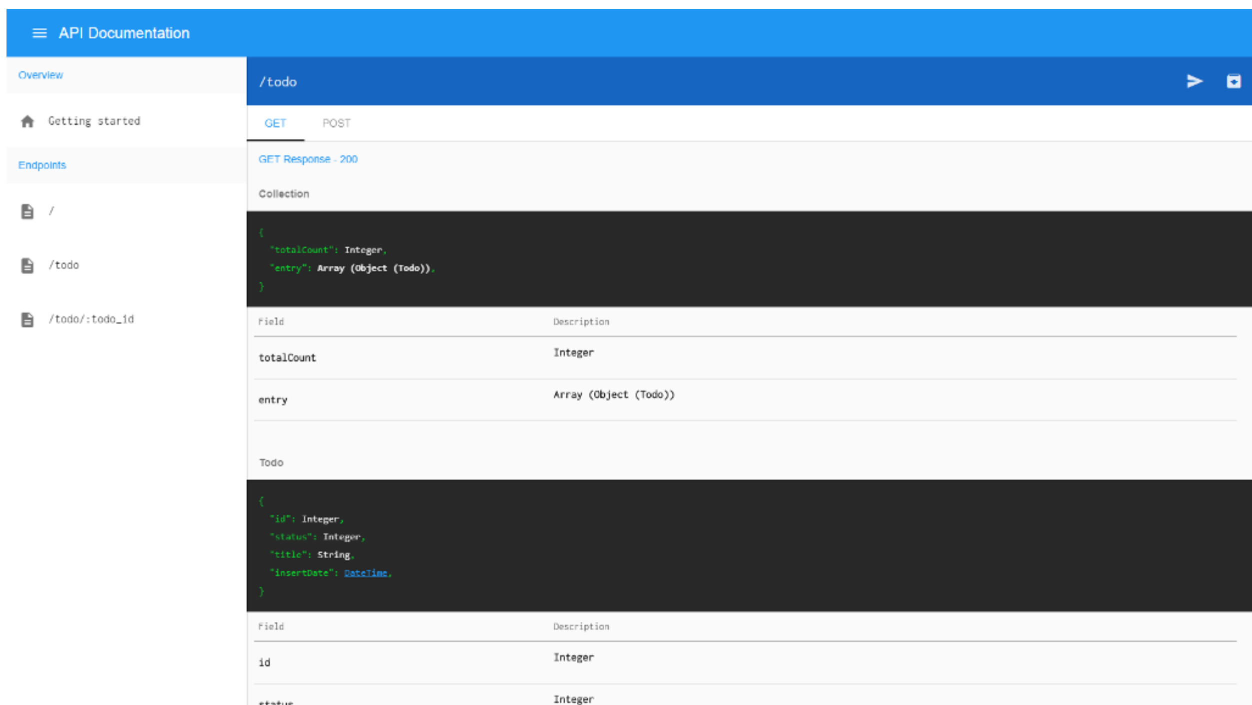
Mostly used for javascript apps. Like in the authorization code flow the app redirects the user to the consumer endpoint i.e.: `/developer/auth?response_type=token&client_id=[key]&redirect_uri=[url]&scope=foo,bar` If the user has authenticated and approved the app the user gets redirected to the `redirect_uri`. The callback contains the access token in the fragment component. The access tokens which are issued through the implicit grant have usually a much shorter life time because they are more insecure. It is also possible to deactivate the implicit grant through the configuration.

24.3 Password

A user can use the password grant to obtain directly an access token with their username and password. Therefor he has to send a direct request to the `/authorization/token` endpoint.

CHAPTER 25

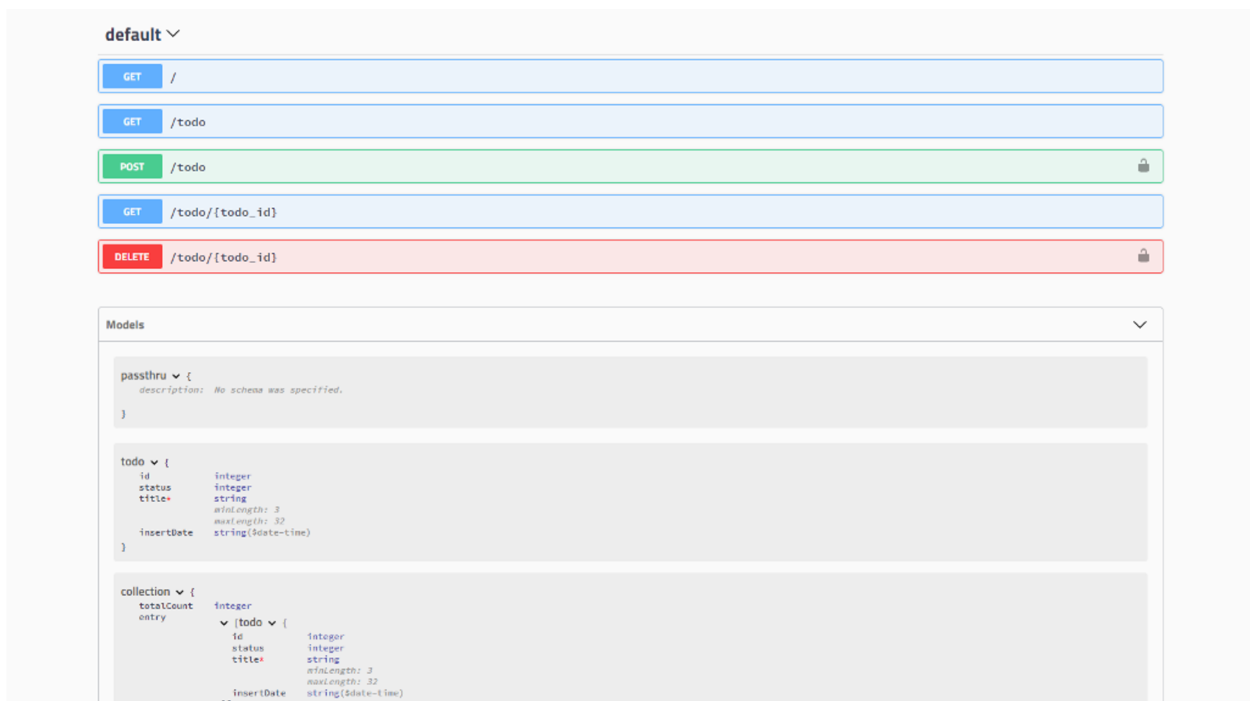
Documentation



The documentation app simply provides an overview of all available endpoints. It is possible to export the API definition into other schema formats like i.e. Swagger. The app is located at `/documentation/`.

CHAPTER 26

Swagger-UI



The swagger-ui app renders a documentation based on the OpenAPI specification. The app is located at `/swagger-ui/`.

CHAPTER 27

API

If you want to access or use the internal REST API of Fusio you can take a look at our internal API documentation. This documentation contains all available REST API endpoints.

CHAPTER 28

Adapter

An adapter is a composer package which provides classes to extend the functionality of Fusio. Through an adapter it is i.e. possible to provide custom action/connection classes or to install predefined routes for an existing system. Our website lists every available composer package which has the `fusio-adapter` keyword defined in the `composer.json` file.

The adapter needs to require the `fusio/engine` package and must have an adapter class which implements the `Fusio\Engine\AdapterInterface` interface. This interface has a method `getDefinition` which returns an absolute path to a `adapter.json` definition file. This definition contains all information for Fusio how to extend the system. The adapter can be installed through the register command:

```
php bin/fusio system:register "Acme\System\Adapter"
```

28.1 Provider

28.1.1 User

Interface: `Fusio\Engine\User\ProviderInterface`

Describes a remote identity provider which can be used to authorize an user through a remote system so that the developer dont need to create an account. Usually this is done through OAuth2, which has the following flow:

- The App redirects the user to the authorization endpoint of the remote provider (i.e. Google)
- The user authenticates and returns via redirect to the App
- The App calls the API endpoint and provides the fitting data to Fusio
- If everything is ok Fusio will get additional information and create a new account

Please take a look at the Github provider for an example implementation.

```
{
  "userClass": ["Acme\System\User\Provider"]
}
```

28.1.2 Payment

Interface: `Fusio\Engine\Payment\ProviderInterface`

Describes a payment provider which can be used to execute payments. Through the developer app the user has the possibility to buy points which can be used to call specific routes which cost points. To buy those points Fusio uses these payment providers to execute a payment. Usually the flow is:

- App calls the API endpoint to prepare a specific product, it provides an plan and a return url. The call returns an approval url
- App redirects the user to the approval url. The user has to approve the payment at the payment provider
- User returns to the App, the url contains the id of the transaction so the app can call the API endpoint to get details about the transaction
- If everything is ok Fusio will credit the points to the user so that he can start calling specific endpoints

Please take a look at the Paypal provider for an example implementation.

```
{
  "paymentClass": ["Acme\System\Payment\Provider"]
}
```

28.1.3 Routes

Interface: `Fusio\Engine\Routes\ProviderInterface`

Preconfigured route provider which helps to create automatically schemas, actions and routes for the user. This can be used to create complete applications.

Please take a look at the SQL-Table provider for an example implementation.

```
{
  "routesClass": ["Acme\System\Routes\Provider"]
}
```

28.2 Testing

If you build an adapter it is recommend to build a test case which extends the `Fusio\Engine\Test\AdapterTestCase` Test-Case. This test case checks whether the *definition.json* is valid and contains only plausible values.

28.3 Schema

Please take a look at the JsonSchema to see all options and to validate an existing definition.json file.

CHAPTER 29

Serialization

Fusio has a general mechanism to serialize the action response to the fitting format for a client. The serializer is used in case the action returns an array or stdClass. It is also possible to return a string as response body but in this case Fusio simply redirects this response.

Fusio respects also the HTTP `Accept` header. This means if the HTTP request contains i.e. the header `Accept: application/xml` Fusio uses the XML serializer to generate a XML response from the data. By default Fusio uses the JSON serializer. Besides the Accept header it is also possible to explicit define a serializer by using the `format` query parameter i.e. `?format=xml`

29.1 Custom writer

If you want to develop a custom serializer you can create a class which implements the `PSX\Data\WriterInterface`. This class receives the raw data and needs to return a string. To add your custom class you need to add the writer to the `io` service at the `container.php` file:

```
/** @var PSX\Data\Processor */
$container->set('io', function($c){
    $config = \PSX\Data\Configuration::createDefault(
        $c->get('annotation_reader'),
        $c->get('schema_manager'),
        $c->get('config')->get('psx_soap_namespace')
    );

    $phpWriter = new \App\Writer\Php();

    $config->getWriterFactory()->addWriter($phpWriter);

    return new \PSX\Data\Processor($config);
});
```

As example we develop a custom writer which serializes the response data using the native PHP *serialize* function. In reality this is not really useful since the format is not language independent but it shows the general mechanism:

```
namespace App\Writer;

class Php implements \PSX\Data\WriterInterface
{
    public function write($data)
    {
        return serialize($data);
    }

    public function isContentTypeSupported(\PSX\Http\MediaType $contentType)
    {
        return $contentType->getName() == 'application/php';
    }

    public function getContentType()
    {
        return 'application/php';
    }
}
```

Also at the `configuration.php` file you need to add the class to the allowed writer classes:

```
'psx_supported_writer'    => [
    \PSX\Data\Writer\Json::class,
    \PSX\Data\Writer\Jsonp::class,
    \PSX\Data\Writer\Jsonx::class,
    \App\Writer\Php::class,
],
```

Then it would be possible to use the writer by using the *Accept: application/php* header or the query parameter *?format=php*.

CHAPTER 30

Event

Fusio has an internal event system which can be used to extend Fusio. This chapter explains how to use those events and shows which events are available.

30.1 Implementation

To register a new event listener you can use the following code at the `container.php` file:

```
use Fusio\Impl\Event\Action;
use Fusio\Impl\Event\ActionEvents;

/** @var \Symfony\Component\EventDispatcher\EventDispatcher $eventDispatcher */
$eventDispatcher = $container->get('event_dispatcher');

$eventDispatcher->addListener(ActionEvents::CREATE, function(Action\CreatedEvent
↪$event){

    // @TODO action was created

});
```

30.2 Reference

Action-Events (`Fusio\Impl\Event\ActionEvents`)

action.create	CREATE	`Fusio\Impl\Event\Action\CreatedEvent`
action.delete	DELETE	`Fusio\Impl\Event\Action\DeletedEvent`
action.update	UPDATE	`Fusio\Impl\Event\Action\UpdatedEvent`

App-Events (`Fusio\Impl\Event\AppEvents`)

app.create	CREATE	Fusio\Impl\Event\App\CreatedEvent
app.delete	DELETE	Fusio\Impl\Event\App\DeletedEvent
app.generate_token	GENERATE_TOKEN	Fusio\Impl\Event\App\GeneratedTokenEvent
app.remove_token	REMOVE_TOKEN	Fusio\Impl\Event\App\RemovedTokenEvent
app.update	UPDATE	Fusio\Impl\Event\App\UpdatedEvent

Config-Events (Fusio\Impl\Event\ConfigEvents)

config.update	UPDATE	Fusio\Impl\Event\Config\UpdatedEvent

Connection-Events (Fusio\Impl\Event\ConnectionEvents)

connection.create	CREATE	Fusio\Impl\Event\Connection\CreatedEvent
connection.delete	DELETE	Fusio\Impl\Event\Connection\DeletedEvent
connection.update	UPDATE	Fusio\Impl\Event\Connection\UpdatedEvent

Cronjob-Events (Fusio\Impl\Event\CronjobEvents)

cronjob.create	CREATE	Fusio\Impl\Event\Cronjob\CreatedEvent
cronjob.delete	DELETE	Fusio\Impl\Event\Cronjob\DeletedEvent
cronjob.update	UPDATE	Fusio\Impl\Event\Cronjob\UpdatedEvent

Event-Events (Fusio\Impl\Event\EventEvents)

event.create	CREATE	Fusio\Impl\Event\Event\CreatedEvent
event.delete	DELETE	Fusio\Impl\Event\Event\DeletedEvent
event.subscribe	SUBSCRIBE	Fusio\Impl\Event\Event\SubscribedEvent
event.unsubscribe	UNSUBSCRIBE	Fusio\Impl\Event\Event\UnsubscribedEvent
event.update	UPDATE	Fusio\Impl\Event\Event\UpdatedEvent

Plan-Events (Fusio\Impl\Event\PlanEvents)

plan.create	CREATE	Fusio\Impl\Event\Plan\CreatedEvent
plan.credit	CREDIT	Fusio\Impl\Event\Plan\CreditedEvent
plan.delete	DELETE	Fusio\Impl\Event\Plan\DeletedEvent
plan.pay	PAY	Fusio\Impl\Event\Plan\PayedEvent
plan.update	UPDATE	Fusio\Impl\Event\Plan\UpdatedEvent

Rate-Events (Fusio\Impl\Event\RateEvents)

rate.create	CREATE	Fusio\Impl\Event\Rate\CreatedEvent
rate.delete	DELETE	Fusio\Impl\Event\Rate\DeletedEvent
rate.update	UPDATE	Fusio\Impl\Event\Rate\UpdatedEvent

Routes-Events (Fusio\Impl\Event\RoutesEvents)

routes.create	CREATE	Fusio\Impl\Event\Routes\CreatedEvent
routes.delete	DELETE	Fusio\Impl\Event\Routes\DeletedEvent
routes.deploy	DEPLOY	Fusio\Impl\Event\Routes\DeployedEvent
routes.update	UPDATE	Fusio\Impl\Event\Routes\UpdatedEvent

Schema-Events (`Fusio\Impl\Event\SchemaEvents`)

schema.create	CREATE	`Fusio\Impl\Event\Schema\CreatedEvent`
schema.delete	DELETE	`Fusio\Impl\Event\Schema\DeletedEvent`
schema.update	UPDATE	`Fusio\Impl\Event\Schema\UpdatedEvent`

Scope-Events (`Fusio\Impl\Event\ScopeEvents`)

scope.create	CREATE	`Fusio\Impl\Event\Scope\CreatedEvent`
scope.delete	DELETE	`Fusio\Impl\Event\Scope\DeletedEvent`
scope.update	UPDATE	`Fusio\Impl\Event\Scope\UpdatedEvent`

Transaction-Events (`Fusio\Impl\Event\TransactionEvents`)

transaction.execute	EXECUTE	`Fusio\Impl\Event\Transaction\ExecutedEvent`
transaction.prepare	PREPARE	`Fusio\Impl\Event\Transaction\PreparedEvent`

User-Events (`Fusio\Impl\Event\UserEvents`)

user.change_password	CHANGE_PASSWORD	`Fusio\Impl\Event\User\ChangedPasswordEvent`
user.change_status	CHANGE_STATUS	`Fusio\Impl\Event\User\ChangedStatusEvent`
user.create	CREATE	`Fusio\Impl\Event\User\CreatedEvent`
user.delete	DELETE	`Fusio\Impl\Event\User\DeletedEvent`
user.fail_authentication	FAIL_AUTHENTICATION	`Fusio\Impl\Event\User\FailedAuthenticationEvent`
user.update	UPDATE	`Fusio\Impl\Event\User\UpdatedEvent`