PSX Documentation

Release 2.0

Christoph Kappestein

Mar 04, 2018

Contents

1 TOC **3**

1.1 Getting started 3

1.2 Design 13

1.3 Concept 20

2 Scope **23**

PSX is a framework written in PHP dedicated to build RESTful APIs. It is based on multiple components which cover many aspects of the API lifecycle. These components are independent of the framework and can also be used in another context. More informations at http://phpsx.org

This manual is seperated into three chapters. The *Getting started* chapter contains all information to quickly create an REST API with PSX. The *Design* chapter contains more technical details about the anatomy of PSX. The last chapter *Concept* covers various concepts to get a deeper understanding of PSX.

CHAPTER 1

TOC

1.1 Getting started

1.1.1 Installation

PSX uses composer as dependency manager. In order to install PSX composer must be installed on your system. More information how to install composer at https://getcomposer.org/.

The easiest way to start is to install the PSX sample project through composer:

```
$ php composer.phar create-project psx/sample .
```

This sample project contains a sample API and all classes which you need to start with PSX.

In case you cant install composer PSX has also a pre-packaged release which already includes all dependencies which you can download at https://github.com/apioo/psx/releases

Configuration

The main configuration is defined in the file `configuration.php` which is a simple php array with key value pairs. You must change the key "psx_url" so that it points to the psx public root. All other entries are optional.

```
<?php

/*
This is the configuration file of PSX. Every parameter can be used inside your
application or in the DI container. Which configuration file gets loaded depends
on the DI container parameter "config.file". See the container.php if you want
load a different configuration depending on the environment.
*/

return array(
```

```
    // The url to the psx public folder (i.e. http://127.0.0.1/psx/public,
    // http://localhost.com or //localhost.com)
    'psx_url'                => 'http://127.0.0.1/projects/psx/public',

    // The default timezone
    'psx_timezone'           => 'UTC',

    // Whether PSX runs in debug mode or not. If not error reporting is set to 0
    // Also several caches are used if the debug mode is false
    'psx_debug'              => true,

    // Database parameters which are used for the doctrine DBAL connection
    // http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/
↪configuration.html
    'psx_connection'         => getConnectionParams(getenv('DB')),

    // Path to the routing file
    'psx_routing'            => __DIR__ . '/routes.php',

    // Folder locations
    'psx_path_cache'         => __DIR__ . '/cache',
    'psx_path_library'       => __DIR__ . '/src',

    // Supported writers
    'psx_supported_writer'   => [
        \PSX\Data\Writer\Json::class,
        \PSX\Data\Writer\Jsonp::class,
        \PSX\Data\Writer\Jsonx::class,
    ],

    // Global middleware which are applied before and after every request. Must
    // bei either a classname, closure or PSX\Dispatch\FilterInterface instance
    //'psx_filter_pre'         => [],
    //'psx_filter_post'        => [],

    // A closure which returns a doctrine cache implementation. If null the
    // filesystem cache is used
    //'psx_cache_factory'      => null,

    // A closure which returns a monolog handler implementation. If null the
    // system handler is used
    //'psx_logger_factory'     => null,

    // Class name of the error controller
    //'psx_error_controller'   => null,

    // If you only want to change the appearance of the error page you can
    // specify a custom template
    //'psx_error_template'     => null,

);

function getConnectionParams($db)
{
    switch ($db) {
        case 'mysql':
            return [
                'dbname'   => 'psx',
```

```
                'user'     => 'root',
                'password' => '',
                'host'     => 'localhost',
                'driver'   => 'pdo_mysql',
            ];
            break;

        case 'pgsql':
            return [
                'dbname'   => 'psx',
                'user'     => 'postgres',
                'password' => '',
                'host'     => 'localhost',
                'driver'   => 'pdo_pgsql',
            ];
            break;

        default:
        case 'sqlite':
            return [
                'path'   => __DIR__ . '/cache/population.db',
                'driver' => 'pdo_sqlite',
            ];
            break;
    }
}
```

If your application needs database access you can enter the credentials in the `psx_connection` key. The connection service provides a Doctrine DBAL connection which you can use in your application.

Routing

In order to make a controller accessible you have to define a route in your `routes` file. If a request arrives at an endpoint PSX tries to autoload the provided class through composer. Here an example route entry:

```
GET|POST|PUT|DELETE /foo/bar Acme\Api\News\Endpoint
```

This would invoke the class *Acme\Api\News\Endpoint* if you visit the route */foo/bar*. All controller classes must extend the class *PSX\Framework\Controller\ControllerAbstract*

Webserver

If you dont have a local web server you can use the build in HTTP server of PHP. You can start the server with the following command:

```
php -S 127.0.0.1:8000 public/index.php
```

The configuration file should have then the following entries:

```
'psx_url'                => 'http://127.0.0.1:8000',
'psx_dispatch'           => '',
```

1.1.2 Controller

The controller receives the request and handles the response. To build an API endpoint we should tell PSX which request methods are available and how the incoming and outgoing data looks. Because of this PSX can automatically validate incoming data and it is possible to generate documentation or other schema formats like i.e. Swagger or RAML based from the controller. In the following we describe the available options.

Annotation

It is possible to provide all API information to the controller through annotations.

```
<?php

namespace PSX\Project;

use PSX\Framework\Controller\SchemaApiAbstract;
use PSX\Record\RecordInterface;

/**
 * @Title("Endpoint")
 * @PathParam(name="foo_id", type="integer")
 */
class Endpoint extends SchemaApiAbstract
{
    /**
     * @QueryParam(name="count", description="Count of comments")
     * @Outgoing(code=200, schema="PSX\Project\Model\Song")
     */
    protected function doGet()
    {
        $count = $this->queryParameters->getProperty('count');

        // @TODO return the result i.e. from a database

        return new Song('Wut ueber den verlorenen groschen', 'Beethoven');
    }

    /**
     * @Incoming(schema="PSX\Project\Model\Song")
     * @Outgoing(code=201, schema="PSX\Project\Model\Message")
     */
    protected function doPost($record)
    {
        // @TODO work with the record

        return new Message(true, 'Successful created');
    }
}
```

The schema must be either a simple POPO class which contains annotations to describe the schema or a json schema file. The model class *PSX\Project\Model\Song* could look like:

```
class Song
{
    /**
     * @Type("string")
     */
```

```
    protected $title;

    /**
     * @Type("string")
     */
    protected $artist;

    public function __construct($title = null, $artist = null)
    {
        $this->title  = $title;
        $this->artist = $artist;
    }

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

    public function setTitle($title)
    {
        $this->title = $title;
    }

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

    public function setArtist($artist)
    {
        $this->artist = $artist;
    }
}
```

It is also possible to generate a PHP class based on a json schema.

```
vendor/bin/psx schema schema.json php
```

More information at the psx schema (https://github.com/apioo/psx-schema) project. The following annotations are available for the controller:

Annotation	Target	Example
@Description	Class/Method	@Description("Bar")
@Exclude	Method	@Exclude
@Incoming	Method	@Incoming(schema="PSX\Project\Model\Song")
@Outgoing	Method	@Outgoing(code=200, schema="PSX\Project\Model\Song")
@PathParam	Class	@PathParam(name="foo", type="integer")
@QueryParam	Method	@QueryParam(name="bar", type="integer")
@Title	Class/Method	@Title("Foo")

RAML

Instead of annotations it is also possible to provide a schema file which describes the endpoint. At the moment we support the RAML (http://raml.org/) specification.

```php
<?php

namespace PSX\Project;

use PSX\Api\Parser\Raml;
use PSX\Framework\Controller\SchemaApiAbstract;
use PSX\Framework\Loader\Context;
use PSX\Record\RecordInterface;

class Endpoint extends SchemaApiAbstract
{
    public function getDocumentation($version = null)
    {
        return $this->apiManager->getApi(__DIR__ . '/endpoint.raml', $this->context->
↪get(Context::KEY_PATH));
    }

    protected function doGet()
    {
        $count = $this->queryParameters->getProperty('count');

        // @TODO return the result i.e. from a database

        return [
            'title'  => 'Wut ueber den verlorenen groschen',
            'artist' => 'Beethoven',
        ];
    }

    protected function doPost(RecordInterface $record)
    {
        // @TODO work with the record

        return [
            'success' => true,
            'message' => 'Successful created',
        ];
    }
}
```

RAML definition (endpoint.raml)

```
#%RAML 1.0
title: Endpoint
baseUri: http://example.phpsx.org
/endpoint/{foo_id}:
  uriParameters:
    foo_id:
      type: integer
  get:
    queryParameters:
      count:
        type: integer
    responses:
      200:
        body:
          application/json:
            type: !include schema/song.json
```

```
  post:
    body:
      application/json:
        schema: !include schema/song.json
    responses:
      201:
        body:
          application/json:
            type: !include schema/message.json
```

1.1.3 Service

In the previous chapter we have defined a controller which receives the request and produces a response. This chapter describes where we can define the business logic of our API endpoint.

Definition

> Put simply, a Service is any PHP object that performs some sort of "global" task. It's a purposefully-generic name used in computer science to describe an object that's created for a specific purpose (e.g. delivering emails). Each service is used throughout your application whenever you need the specific functionality it provides. You don't have to do anything special to make a service: simply write a PHP class with some code that accomplishes a specific task. Congratulations, you've just created a service!
>
> So what's the big deal then? The advantage of thinking about "services" is that you begin to think about separating each piece of functionality in your application into a series of services. Since each service does just one job, you can easily access each service and use its functionality wherever you need it. Each service can also be more easily tested and configured since it's separated from the other functionality in your application. This idea is called service-oriented architecture and is not unique to Symfony or even PHP. Structuring your application around a set of independent service classes is a well-known and trusted object-oriented best-practice. These skills are key to being a good developer in almost any language.

This description was copied from the symfony documentation[1]

Usage

It is recommended to place your API logic inside a service. In the controller you only need to call service methods which return the response data or handle the request data. The service should be independent of the HTTP request/response context. Only the controller needs to read the HTTP data and pass it to the service methods.

In order to use a service from the DI container you can use the *Inject* annotation to include a service into your controller. In the following we extend our previous controller with a service which fetches/inserts data on a table.

```
<?php

namespace PSX\Project;

use PSX\Framework\Controller\AnnotationApiAbstract;
use PSX\Record\RecordInterface;

/**
 * @Title("Endpoint")
 * @PathParam(name="foo_id", type="integer")
```

[1] http://symfony.com/doc/current/book/service_container.html

```
 */
class Endpoint extends AnnotationApiAbstract
{
    /**
     * @Inject
     * @var \Acme\NewsService
     */
    protected $news;

    /**
     * @QueryParam(name="count", description="Count of comments")
     * @Outgoing(code=200, schema="schema/song.json")
     */
    protected function doGet()
    {
        return $this->news->getSongById(
            $this->getUriFragment('foo_id'),
            $this->queryParameters->getProperty('count')
        );
    }

    /**
     * @Incoming(schema="schema/song.json")
     * @Outgoing(code=201, schema="schema/message.json")
     */
    protected function doPost($record)
    {
        $this->news->createSong(
            $record->title,
            $record->artist,
        );

        return [
            'success' => true,
            'message' => 'Successful created',
        ];
    }
}
```

Register

In order to add a new service to the DI container you have to add a method to the container class. In the sample porject the container class is located at *src/Sample/Dependency/Container.php*. In the following an example which creates a new service:

```
class Container extends DefaultContainer
{
    /**
     * @return \Acme\ServiceInterface
     */
    public function getAcmeService()
    {
        return new Acme\Serivce();
    }
}
```

This service can then be used in a controller.

```
<?php

class Endpoint extends SchemaApiAbstract
{
    /**
     * @Inject
     * @var \Acme\ServiceInterface
     */
    protected $acmeService;
}
```

Command

By default PSX comes with the following registered services which can be used inside a controller:

```
annotation_reader            \Doctrine\Common\Annotations\Reader
annotation_reader_controller \Doctrine\Common\Annotations\Reader
api_manager                  \PSX\Api\ApiManager
application_stack_factory    \PSX\Framework\Dispatch\ControllerFactoryInterface
cache                        \Psr\Cache\CacheItemPoolInterface
config                       \PSX\Framework\Config\Config
connection                   \Doctrine\DBAL\Connection
console                      \Symfony\Component\Console\Application
console_reader               \PSX\Framework\Console\ReaderInterface
controller_factory           \PSX\Framework\Dispatch\ControllerFactoryInterface
dispatch                     \PSX\Framework\Dispatch\Dispatch
dispatch_sender              \PSX\Framework\Dispatch\SenderInterface
event_dispatcher
↪\Symfony\Component\EventDispatcher\EventDispatcherInterface
exception_converter          \PSX\Framework\Exception\ConverterInterface
http_client                  \PSX\Http\ClientInterface
io                           \PSX\Data\Processor
loader                       \PSX\Framework\Loader\Loader
loader_callback_resolver     \PSX\Framework\Loader\CallbackResolverInterface
loader_location_finder       \PSX\Framework\Loader\LocationFinderInterface
logger                       \Psr\Log\LoggerInterface
object_builder               \PSX\Framework\Dependency\ObjectBuilderInterface
request_factory              \PSX\Framework\Dispatch\RequestFactoryInterface
resource_listing             \PSX\Api\ListingInterface
response_factory             \PSX\Framework\Dispatch\ResponseFactoryInterface
reverse_router               \PSX\Framework\Loader\ReverseRouter
routing_parser               \PSX\Framework\Loader\RoutingParserInterface
schema_manager               \PSX\Schema\SchemaManagerInterface
session                      \PSX\Framework\Session\Session
table_manager                \PSX\Sql\TableManagerInterface
template                     \PSX\Framework\Template\TemplateInterface
validate                     \PSX\Validate\Validate
```

A current list of services can also be generated with the following command.

```
vendor\bin\psx container
```

1.1.4 Tools

PSX comes by default with a set of controllers which help you to generate documentation or other data based on the defined APIs. The following chapter explains all available controllers. To setup such a controller you have to add a route entry.

Documentation

PSX provides a controller which can generate automatically a documentation for the defined APIs. The controller provides only an API which can be used by any client to display a documentation for end-users. You can use i.e. evid (https://github.com/k42b3/evid) which is a html/javascript app which uses the API to provide a clean API documentation. With evid it is also very easy to customize the documentation for your own needs by i.e. providing custom pages or change the style according to your needs.

```
GET /doc                PSX\Framework\Controller\Tool\DocumentationController::doIndex
GET /doc/:version/*path␣
↪PSX\Framework\Controller\Tool\DocumentationController::doDetail
```

Swagger

Generates a Swagger resource listing and definition.

```
GET /swagger/:version/*path PSX\Framework\Controller\Generator\SwaggerController
```

RAML

Generates a RAML representation for the given API.

```
GET /raml/:version/*path PSX\Framework\Controller\Generator\RamlController
```

Routing

Provides an API to publish all available API paths.

```
GET /routing PSX\Framework\Controller\Tool\RoutingController
```

1.1.5 CLI

PSX provides several commands which help to rapidly develop and debug PSX applications. In the following we will describe the available commands:

API

It is possible to automatically generate an API controller from a specification. The following command reads the */acme/endpoint* and generates the appropriated controller.

```
$ vendor/bin/psx api spec.raml php
```

Schema

Through the schema command it is possible to generate a POPO based on a jsonschema specification.

```
$ vendor/bin/psx schema schema.json php
```

1.2 Design

1.2.1 Request lifecycle

This chapter explains the request lifecycle of a PSX application.

Http request/response

PSX uses a standard HTTP request and response model. At the start of the application lifecycle a HTTP request and response object will be created.

```
interface RequestFactoryInterface
{
    /**
     * Returns the http request containing all values from the environment
     *
     * @return \PSX\Http\RequestInterface
     */
    public function createRequest();
}
```

```
interface ResponseFactoryInterface
{
    /**
     * Returns the http response containing default values and the body stream
     *
     * @return \PSX\Http\ResponseInterface
     */
    public function createResponse();
}
```

After the request and response objects are created the loader searches the fitting controller based on the routing file. The controller must implement the ApplicationStackInterface.

```
interface ApplicationStackInterface
{
    /**
     * Returns an array containing FilterInterface or callable. The request and
     * response object gets passed to each filter which then can produce the
     * response
     *
     * @return \PSX\Framework\Filter\FilterInterface[]|\Closure[]
     */
    public function getApplicationStack();
}
```

Then the loader receives the application stack from the controller which is an array containing callable or FilterInterface. Each middleware can then read from the request and write data to the response.

```
interface FilterInterface
{
    /**
     * @param \PSX\Http\RequestInterface $request
     * @param \PSX\Http\ResponseInterface $response
     * @param \PSX\Framework\Filter\FilterChainInterface $filterChain
     * @return void
     */
    public function handle(RequestInterface $request, ResponseInterface $response,␣
↪FilterChainInterface $filterChain);
}
```

After the stack was executed the response must be send to the client. This is done through a sender class which sends the header through the "header" function and outputs the response body via "echo".

```
interface SenderInterface
{
    /**
     * Method to send the response which was created to the browser
     *
     * @param \PSX\Http\ResponseInterface $response
     */
    public function send(ResponseInterface $response);
}
```

Events

Through the request lifecycle there are some places where PSX triggers an event. In the following a list of events with a short description.

Event name	Description
Event::REQUEST_INCOMING	Triggered when an request arrives
Event::ROUTE_MATCHED	Triggered when an route was found for the request
Event::CONTROLLER_EXECUTE	Triggered before an controller gets executed
Event::CONTROLLER_PROCESSED	Triggered after an controller was executed
Event::RESPONSE_SEND	Triggered before the response gets send to the client
Event::EXCEPTION_THROWN	Triggered when an exception occurs
Event::COMMAND_EXECUTE	Triggered before an command gets executed
Event::COMMAND_PROCESSED	Triggered after an command was executed

Middleware

In PSX a middleware must be either a FilterInterface or callable i.e. the most basic "hello world" example would be:

```
<?php

use PSX\Framework\Controller\ControllerAbstract;

class Controller extends ControllerAbstract
{
    public function getApplicationStack()
    {
        return [function($request, $response){
```

```
            $response->getBody()->write('Hello world');
        }];
    }
}
```

By default the controller returns the ControllerExecutor middleware which simply calls the on* methods and optional the method which was set in the routes file. In addition you could also overwrite the `getPreFilter` or `getPostFilter` method which are merged together to the application stack.

1.2.2 Dependency injection

Definition

The file `container.php` returns the main DI container of your application.

```
$container = new \PSX\Framework\Dependency\DefaultContainer();
$container->setParameter('config.file', __DIR__ . '/configuration.php');

return $container;
```

The DI container of PSX is a simple class where each method represents a service definition. Normally you would create your own container which extends the PSX default container. To add a service to the container you have to simply specifiy a method in the container which returns an object instance.

```
<?php

class CustomContainer extends DefaultContainer
{
    /**
     * @return Foo\Bar
     */
    public function getSomeBar()
    {
        return new Bar($this->get('entity_manager'));
    }

    /**
     * @return Foo\EntityManager
     */
    public function getEntityManager()
    {
        return new EntityManager();
    }
}
```

The id of the service is "some_bar" which can be used in an inject annotation. Note when your object has a dependency to another service use the $this->get('[service_id]') method to get only one instance of the service.

The returned container must be compatible with the symfony DI container. You could also use i.e. the symfony DI container

```
<?php

$container = new Symfony\Component\DependencyInjection\ContainerBuilder();
$loader = new Symfony\Component\DependencyInjection\Loader\XmlFileLoader($container,␣
↪new Symfony\Component\Config\FileLocator(__DIR__));
```

```
$loader->load('services.xml');

return $container;
```

Usage

The DI container is not directly accessible instead you have to specify properties in your controller with a specific inject annotation to use a service.

```
<?php

class FooController extends ControllerAbstract
{
    /**
     * @Inject
     * @var \PSX\Http\Client\ClientInterface
     */
    protected $httpClient;
}
```

When the controller is created PSX searches for properties with the "@Inject" annotation. If the inject annotation is available it takes the name of the property and searches in the DI container for a fitting service. You can also specify the id of the service which should be used i.e. *@Inject("entity_manager")*. We need to have the following method in our DI container in order to access the *httpClient* service from the code above.

```
<?php

class DefaultContainer extends Container
{
    /**
     * @return PSX\Http\Client\ClientInterface
     */
    public function getHttpClient()
    {
        return new Http\Client();
    }
}
```

This has the advantage that the DI container is completely invisible for our application. We only need to explicit specify the services which we need in our controller. So it is by design not possible to pass the DI container to any service in our application which decouples the code from the framework. Also it has the nice advantage that you can use code completion in your IDE.

1.2.3 Middleware

Abstract

This chapter gives a short overview how middlewares are implemented in PSX.

Usage

In PSX a controller is basically a class which is a traversable containing middleware instances. A middleware is either an instance of `FilterInterface` or `Closure`. The default `ControllerAbstract` merges the middlewares

from the `getPreFilter` and `getPostFilter` method with the controller it self since it is also a middleware:

```
<?php

class Controller extends ControllerAbstract
{
    public function getIterator()
    {
        return new \ArrayIterator(array_merge(
            $this->getPreFilter(),
            [$this],
            $this->getPostFilter()
        ));
    }
}
```

Also the global pre and post middlewares are merge to the middleware stack. The global middlewares are specified at the `configuration.php` file:

```
'psx_filter_pre'  => [],
'psx_filter_post' => [],
```

This complete stack of middlewares is then executed.

1.2.4 Routing

Routing is the process to delegate the HTTP request to the fitting controller. By default PSX uses a simple routing PHP file which contains all available routes.

Routing file

PSX parses the routing file and the route which matches first is taken. Lets take a look at the following example routing file

```
<?php

# example routing file

return [
    [["GET"], "/", "Foo\News\Application\Index"],
    [["GET"], "/foo", "~/"],
    [["GET", "POST"], "/api", "Foo\News\Application\Api"],
    [["GET"], "/news/:news_id", "Foo\News\Application\News\Detail"],
    [["GET"], "/news/archive/$year<[0-9]+>", "Foo\News\Application\News\Archive"],
    [["GET"], "/file/*path", "Foo\News\Application\File"],
];
```

in case we make the following HTTP request the *Foo\News\Application\Api* controller gets called

```
GET /api HTTP/1.1
```

The path */foo* is an alias for the path */*. That means in both cases we call controller *Foo\News\Application\Index*.

If we make a HTTP request to */news/2* the controller *Foo\News\Application\News\Detail* gets called. In our controller we can access the dynamic part of the path with:

```
<?php

$newsId = $this->context->getUriFragment('news_id')
```

The dynamic part contains all content to the next slash. You can also specifiy a regular expression to define which chars are allowed in the dynamic part. In our example the path /news/archive/$year<[0-9]+> is only matched if the last part contains numeric signs i.e. /news/archive/2014.

Also you can specifiy a wildcard * which matches all content i.e. /file/foo/bar/test.txt would invoke *Foo\News\Application\File*

Reverse routing

PSX contains a reverse router class with that it is possible to get the path or url to an existing controller method. If you want i.e. to link or redirect to a specific controller method

```
<?php

$path = $this->reverseRouter->getPath('Foo\News\Application\Index');
$url  = $this->reverseRouter->getUrl('Foo\News\Application\Index');
$path = $this->reverseRouter->getPath('Foo\News\Application\News\Archive', ['year' =>␣
↪2014]);
```

or in you template you can use

```
<a href="<?php echo $router->getPath('Foo\News\Application\Index'); ?>">Home</a>
```

this has the advantage that you can easily change your urls in your routing file and all links point automatically to the correct path.

Custom routing

If you want store your routes in another format or in a database you can write your own RoutingParser. The routing parser returns a RoutingCollection which contains all available routes.

```
interface RoutingParserInterface
{
    /**
     * @return \PSX\Framework\Loader\RoutingCollection
     */
    public function getCollection();
}
```

Your routing class has to implement this interface. Then you can overwrite the method `getRoutingParser` in your DI container. Note in case you have a really huge amount of routes you should probably consider to write your own location finder since the routing collection contains all available routes. A location finder has to implement the following interface.

```
interface LocationFinderInterface
{
    /**
     * Resolves the incoming request to an source. An source is an string which
     * can be resolved to an callback. The source must be added to the context.
     * If the request can not be resolved the method must return null else the
     * given request
```

```
     *
     * @param \PSX\Http\RequestInterface $request
     * @param \PSX\Framework\Loader\Context $context
     * @return \PSX\Http\RequestInterface|null
     */
    public function resolve(RequestInterface $request, Context $context);
}
```

1.2.5 Controller

Abstract

This chapter gives a short overview of the most important methods which you can use inside a controller.

Usage

```
<?php

namespace Foo\Bar;

use PSX\Framework\Controller\ControllerAbstract;
use PSX\Http\RequestInterface;
use PSX\Http\ResponseInterface;

class Controller extends ControllerAbstract
{
    public function onGet(RequestInterface $request, ResponseInterface $response)
    {
        // returns the request url as PSX\Uri\Uri
        $request->getUri();

        // returns a request header
        $request->getHeader('Content-Type');

        // returns a GET parameter
        $request->getUri()->getParameter('id');

        // returns the request body
        $body = $this->requestReader->getBody($request);

        // imports data from the request body into the model
        $model = $this->requestReader->getBodyAs($request, Model::class);

        // set data to the response body
        $this->responseWriter->setBody($response, [
            'foo' => 'bar'
        ]);
    }
}
```

1.3 Concept

1.3.1 Authentication

Abstract

This chapter shows how to implement authentication for a controller

Basic authentication

Basic authentication is the most simple authentication method where a user provides a username and password in the header. Note if you use basic authentication you should use https since the username and password is transported in plaintext over the wire. Add the following method to the controller in order to add basic authentication

```
<?php

use PSX\Http\Filter\BasicAuthentication;

...

public function getPreFilter()
{
    $auth = new BasicAuthentication(function($username, $password) {
        if ($username == '[username]' && $password == '[passsword]') {
            return true;
        }

        return false;
    });

    return [$auth];
}
```

Oauth authentication

Sample oauth authentication. This is only to illustrate what to return. Normally you have to check

- is the consumerKey valid
- does the token belongs to a valid request with a valid status
- is the token not expired

PSX calculates and compares the signature if you return an consumer. For more informations see RFC 5849.

```
<?php

use PSX\Framework\Filter\OauthAuthentication;
use PSX\Oauth\Consumer;

...

public function getPreFilter()
{
    $auth = new OauthAuthentication(function($consumerKey, $token) {
```

```
        if ($consumerKey == '[consumerKey]' && $token == '[token]') {
            return new Consumer('[consumerKey]', '[consumerSecret]', '[token]',
↪'[tokenSecret]');
        }

        return false;
    });

    return [$auth];
}
```

Oauth2 authentication

Sample oauth2 authentication. In the callback you have to check whether the provided *Bearer* access token is valid. For more informations see **RFC 6749**.

```
<?php

use PSX\Framework\Filter\Oauth2Authentication;

...

public function getPreFilter()
{
    $auth = new Oauth2Authentication(function($accessToken) {
        if ($accessToken == '[accessToken]') {
            return true;
        }

        return false;
    });

    return [$auth];
}
```

1.3.2 Value objects

Abstract

PSX uses in several places value objects. To unify the look and feel of those value objects we have a simple specification which describes the behaviour.

Specification

- The class consists of multiple components. A component must be either a scalar value or another value object
- The class must be immutable so it must be not possible to modify the state of a value object once it was created
- The class must have a constructor with two behaviours. If only one argument was passed to the constructor this value represents the string representation of the value object and must be parsed into its components. If more arguments are provided they must be assigned to the fitting components. It must be possible to pass every component to the constructor. I.e. the following two objects are equal:

```
$mediaType = new MediaType('text/plain; charset=UTF-8');

$mediaType = new MediaType('text', 'plain', ['charset' => 'UTF-8']);
```

- The class must have a method `toString()` which returns a string which contains the complete state of the object. The following operation should be always true:

```
$uri = new Uri('/foo');

$uri->toString() === (new Uri($uri->toString()))->toString();
```

  We use a custom `toString()` method since the exception handling inside the `__toString()` method is really bad. Nevertheless each object can have the magic `__toString()` method which calls the `toString()` method

- The class must have a protected property and getter method for each component

```
$mediaType = new MediaType('text/plain; charset=UTF-8');

$mediaType->getType();
$mediaType->getSubType();
```

- The class may have static factory methods `fromXXX` to create the object from other values

```
$time = Time::fromDateTime(new \DateTime())
```

- The class may have `withXXX` methods to create a new value object replacing only the given component

```
$uri = new Uri('/test?foo=bar');

$uri = $uri->withPath('/foo');
```

- The value object should provide a protected method `parse()` which parses the string representation. When parsing the value must be cast to a string so that in case it is another value object it returns its string representation. If there are components which need extra processing provide protected methods `parseXXX()` which are called inside the parse method. This has the advantage that value objects which extend this class can override these methods so they can choose whether they want to parse theses values or not

Implementations

The following value object implementations exist:

- PSX\Uri\Uri
- PSX\Uri\Url
- PSX\Uri\Urn
- PSX\Http\MediaType
- PSX\Http\Cookie
- PSX\DateTime\Date
- PSX\DateTime\Time
- PSX\DateTime\DateTime

CHAPTER 2

Scope

This manual covers the full stack framework. If you need informations about a specific component please take a look at the repository.

Index

R

RFC
    RFC 5849, 20
    RFC 6749, 21