

TD Introduction à SLIM Framework

Ce TD vise à vous faire découvrir le framework SLIM qui permet une première approche de l'utilisation d'un framework « pro » avec le langage PHP. Il vous permettra par la suite d'appréhender des frameworks plus lourds comme Symfony ou encore le Zend Framework.

Partie 1 : premier contact avec Slim

Installation :

Il va falloir installer quelques petites choses afin de bien démarrer.

Commençons par installer composer sur votre machine Debian (placez vous dans le dossier /home pour ces manipulations) :

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fd586475ca9813a858088ffbc
1f233e9b180f061') { echo 'Installer verified'; } else { echo 'Installer corrupt';
unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Composer est un outil permettant de gérer les installations de composants. Il vous facilitera la tâche en automatisant les installations.

- Créez un nouveau répertoire (mySlimApp par exemple) pour l'application dans le dossier /var/www/html, un sous dossier « src » dans lequel vous vous placerez et exécutez y la commande suivante :

```
php /home/composer.phar require slim/slim "^3.0"
```

Ceci va installer le framework et ses dépendances. Attendez la fin du processus et vérifiez la présence de nouveaux fichiers dans votre dossier.

- Créez un dossier public dans le dossier src, au même niveau que le dossier vendor généré par la précédente commande.
- Créez un fichier index.php dans ce nouveau répertoire qui fera un classique HelloWorld. Le code de ce fichier est le suivant : (nous y reviendrons)

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;
require '../vendor/autoload.php';

$app = new \Slim\App;

$app->get('/hello/{name}', function (Request $request, Response $response, array $args) {
    $name = $args['name'];
    $response->getBody()->write("Hello, $name");
    return $response;
});

$app->run();
```

- Dans le dossier public, créez un fichier .htaccess avec le contenu suivant (on redirige toutes les requêtes vers index.php)

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule . index.php [L]
```

Ces règles de redirection nécessitent qu'un module d'Apache spécifique soit activé : le module mod_rewrite (réécriture d'URL). Pour cela :

- Vérifiez la présence du module dans les librairies Apache :

```
ls -l /usr/lib/apache2/modules/
```

- Activez le module par la commande :

```
a2enmod rewrite
```

- Editez le fichier de configuration d'Apache 2 :

```
nano /etc/apache2/apache2.conf
```

- Ajoutez en fin de fichier :

```
<ifModule mod_rewrite.c>
RewriteEngine On
</ifModule>
```

- Redémarrez Apache

Base de données MySQL :

Nous mettrons en place une base de données MySQL avec une table utilisateur qui contiendra notamment des identifiants de connexions.

- Créez une nouvelle base slimTDBDD et ajoutez-y la table suivante :

```
CREATE TABLE IF NOT EXISTS `user` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `tel` varchar(50) DEFAULT '0',
  `nom` varchar(50) DEFAULT '0',
  `prenom` varchar(50) DEFAULT '0',
  `mail` varchar(100) DEFAULT '0',
  `mdp` varchar(100) DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

- Ajoutez quelques enregistrements dans cette table

Nous allons tout de suite ajouter les paramètres de connexion à notre BDD dans la configuration. Pour cela, ajoutez les lignes suivantes dans votre fichier index.php :

```
$config['displayErrorDetails'] = true;
$config['addContentLengthHeader'] = false;

$config['db']['host'] = 'localhost';
$config['db']['user'] = 'user';
$config['db']['pass'] = 'password';
$config['db']['dbname'] = 'exampleapp';
```

Modifiez la création de l'objet App en lui passant ces paramètres :

```
$app = new \Slim\App(['settings' => $config]);
```

Chargement de nos classes personnelles :

Un mécanisme de chargement automatique des classes existe. Celui-ci parvient à charger les classes du dossier vendor mais nous pouvons également faire de même pour nos propres classes que nous stockerons par exemple dans un dossier « classes ». Pour indiquer au système l'endroit où ces classes se trouvent, modifier le fichier composer.json de la manière suivante :

```
1 {
2   "require": {
3     "slim/slim": "^3.0"
4   },
5   "autoload": {
6     "psr-4": {
7       "": "classes/"
8     }
9   }
10 }
```

Un appel à la commande de mise à jour est ensuite nécessaire :

```
php composer.phar update
```

Injection de dépendances avec le container :

Les frameworks modernes comme SLIM mettent en œuvre un principe que l'on appelle l'injection de dépendances. Celui-ci va permettre d'injecter à la volée des services (des objets créés) lorsque nous en aurons besoin. Cela peut être par exemple des services de logs, une connexion à une base ...

C'est la classe Container qui va permettre cela dans Slim et on peut obtenir cet objet directement sur l'instance de l'application avec cette instruction :

```
$container = $app->getContainer();
```

Nous allons voir un exemple concret d'utilisation en ajoutant une librairie de gestion de logs à notre application. Celle-ci se base sur la librairie Monolog que vous allez installer avec composer :

```
php composer.phar require monolog/monolog
```

- Créez un dossier logs dans le dossier src de votre application et donnez-y tous les droits avec un chmod 777.
- Ajoutez ensuite ces instructions dans votre index.php :

```
$container['logger'] = function($c) {  
    $logger = new \Monolog\Logger('my_logger');  
    $file_handler = new \Monolog\Handler\StreamHandler('../logs/app.log');  
    $logger->pushHandler($file_handler);  
    return $logger;  
};
```

- Ajoutez ensuite ce code dans la route hello par défaut :

```
$this->logger->addInfo('Something interesting happened');
```

- Appelez votre application dans le navigateur et allez constater que votre fichier de logs a bien été modifié.

Nous allons faire de même pour la connexion à la base de données en préparant une connexion PDO qui pourra être utilisée par notre application.

Ajoutez donc le code suivant à votre index.php :

```
$container['db'] = function ($c) {  
    $db = $c['settings']['db'];  
    $pdo = new PDO('mysql:host=' . $db['host'] . ';dbname=' . $db['dbname'],  
        $db['user'], $db['pass']);  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
    $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);  
    return $pdo;  
};
```

Création de nouvelles routes :

Nous allons utiliser notre service d'accès à la BDD pour faire les opérations élémentaires sur notre table user : lire par identifiant, lire tous, ajouter un user ... Ajoutez donc les routes suivantes à votre index.php :

```
$app->get('/allUsers/', function (Request $request, Response $response, array $args) {
    $daoUser = new DAOUser($this->db);
    $lesUsers = $daoUser->findAll();
    $response->getBody()->write("Liste des users :". print_r($lesUsers));
    $this->logger->addInfo('Something interesting happened');
    return $response;
});

$app->get('/anUser/{id}', function (Request $request, Response $response, array $args) {
    $id = $args['id'];
    $daoUser = new DAOUser($this->db);
    $leUser = $daoUser->find($id);
    $response->getBody()->write("Récupéré :". print_r($leUser));
    $this->logger->addInfo('Something interesting happened');
    return $response;
});

$app->get('/login/{id}/{mdp}', function (Request $request, Response $response, array $args) {
    $params = $request->getQueryParams();

    $response->getBody()->write("Paramètres passés :". var_dump($params));
    $this->logger->addInfo('Something interesting happened');
    return $response;
});

$app->post('/login/new', function (Request $request, Response $response) {
    $data = $request->getParsedBody();
    $donnees = [];
    $donnees['nom'] = filter_var($data['nom'], FILTER_SANITIZE_STRING);
    $donnees['prenom'] = filter_var($data['prenom'], FILTER_SANITIZE_STRING);

    $response->getBody()->write("Paramètres POST reçus :". var_dump($donnees));
    $this->logger->addInfo('Something interesting happened');
    return $response;
});
```

- Testez ces nouvelles routes en les appelant dans votre navigateur.
- Pour la méthode POST, vous devrez tester avec un outil comme postman qui vous permettra de simuler un appel POST à votre url en lui fournissant les 2 paramètres attendus nom et prénom.

L'écran ci après montre la requête post appelée dans postman et on constate que les données passées en POST sont bien récupérées par la route.

POST

http://193.252.48.172:8204/mySlimApp/src/public/index.php/login/new

Params

Send

Save

Key	Value	Description	...	Bulk Edit
New key	Value	Description		

AuthorizationHeaders (2)BodyPre-request ScriptTestsCode

☒ form-data☐ x-www-form-urlencoded☐ raw☐ binary

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> nom	Doe			
<input checked="" type="checkbox"/> prenom	John			
New key	Value	Description		

BodyCookiesHeaders (8)Test ResultsStatus: 200 OKTime: 165 ms

PrettyRawPreviewHTML

```
1 Paramètres POST reçus :array(2) {
2   ["nom"]=>
3   string(3) "Doe"
4   ["prenom"]=>
5   string(4) "John"
6 }
7
```

A vous :

Ajoutez les routes pour supprimer un utilisateur et pour authentifier un utilisateur d'après son login et mot de passe.

Partie 2 : le templating de vues

Cette notion de templating consiste à utiliser un framework dont le rôle est de faciliter la gestion des vues en créant des templates (des modèles) que les contrôleurs appelleront en leur fournissant les données à afficher.

Nous pouvons très bien coder notre contenu HTML depuis les routes directement mais vous allez voir qu'il est plus simple de séparer cela dans des templates et que se faire assister par un framework dédié à cela présente des avantages intéressants.

Nous allons utiliser une solution parmi d'autres qui se nomme TWIG.

- Placez vous dans le dossier src de votre projet et installez twig avec cette commande :

```
composer require slim/twig-view
```

- Il faut ensuite enregistrer un nouveau composant twig pour le container de notre application (comme nous l'avons fait pour les log et pour la base de données) :

```
// Register component on container
$container['view'] = function ($container) {
    $view = new \Slim\Views\Twig('path/to/templates', [
        'cache' => 'path/to/cache'
    ]);

    // Instantiate and add Slim specific extension
    $basePath = rtrim(str_ireplace('index.php', '', $container['request']->getUri()->getBasePath()), '/');
    $view->addExtension(new \Slim\Views\TwigExtension($container['router'],
    $basePath));

    return $view;
};
```

Il faudra créer un nouveau dossier « vues » dans le dossier src afin d'y stocker nos templates. Vous indiquerez ce dossier à la place de « path/to/template » dans le code ci-dessus.

De même pour la mise en cache des pages, créer un dossier « cache » en lui octroyant tous les droits et modifiez l'emplacement dans le code.

- Créons ensuite une première page « login.html » très simple qui contiendra le code suivant :

```
<html>

<head>
<title>Login </title>
</head>

<body>

<h1>Connexion</h1>

<p>
Connectez vous à l'application avec votre login et mot de passe.
</p>

</body>

</html>
```

- Ajoutons une nouvelle route dans le fichier index.php :

```
$app->get('/login', function ($request, $response) {
    return $this->view->render($response, 'login.html', [
        'message' => "Bienvenue"
    ]);
})->setName('login');
```

Ici, on demande au gestionnaire twig de nous rendre la vue « login.html » et on lui passe un paramètre que la vue pourra exploiter « message » qui contient la chaîne de caractères « Bienvenue ».

Remarquez la fonction set Name en toute fin de route qui sert à donner un nom à notre route afin de pouvoir l'appeler plus facilement depuis le code de nos vues (nous y reviendrons).

- Modifiez votre template pour qu'il utilise cette variable « message » en modifiant le code :

```
<html>
<head>
<title>Login </title>
</head>
<body>

<h1>{{ message }}</h1>

<p>
Connectez vous à l'application avec votre login et mot de passe.
</p>

</body>

</html>
```

Ici, vous venez de voir comment passer des variables à des vues facilement depuis un contrôleur. C'est très important de maîtriser cela pour la suite.

Vous constatez que le fait d'utiliser un template permet de bien séparer l'architecture du code en couches distincts (principe de MVC rappelons le). Ici, la vue ne se mélange pas au code du contrôleur et on ne lui fournit que les variables dont on sait qu'elle aura besoin (le message dans notre exemple).

L'héritage de templates :

Dans vos sites, vous êtes habitués à créer des pages et à inclure des parties communes sur chacune comme des bannières ou des menus répétitifs. L'utilisation de TWIG va vous permettre d'aller beaucoup plus loin avec l'héritage de templates. Comme pour les classes en POO, on va pouvoir définir un modèle de page de base et l'étendre sur d'autres pages pour le modifier tout en gardant des parties de code communes. Voyons cela en modifiant notre application ...

- Créez un fichier de base que nous appellerons « layout.html » :

```
<!DOCTYPE HTML">
<html>
<head>
    <meta charset="utf-8" />
    {% block head %}
    <link rel="stylesheet" href="../../css/styles.css" />
    <title>{% block title %}{% endblock %} - Application SIO Slim</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2018.
        {% endblock %}
    </div>
</body>
</html>
```

Ici on définit la structure de base et on crée des blocs avec des zones dont le contenu pourra varier dans les vues qui en hériteront.

- Remplaçons le contenu du fichier login.html par celui-ci :

```
{% extends "layout.html" %}

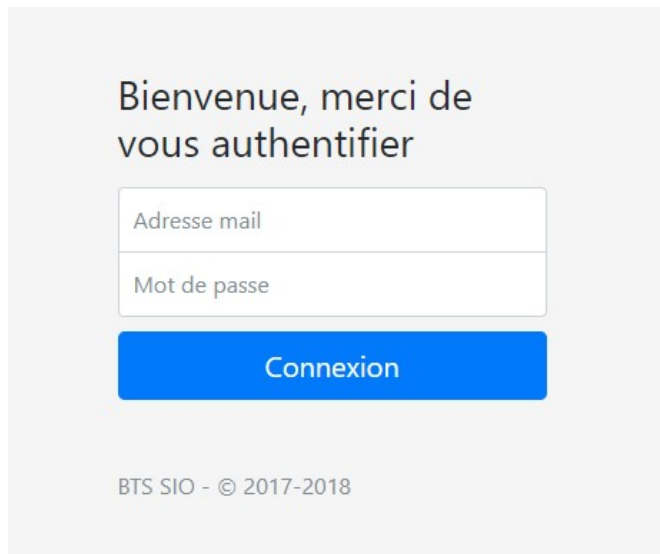
{% block title %}Login page{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Login </h1>
    <p class="important">
        {{ message }}
    </p>
{% endblock %}
```

Dans ce fichier, nous héritons du template de base « layout.html ». Nous redéfinissons son titre (avec le bloc title), puis la partie head en prenant soin de rappeler le code du fichier parent. Nous définissons ensuite le bloc content pour le contenu de notre page où l'on affiche le message passé en paramètre de la vue.

A vous :

Améliorez la page Login en ajoutant les 2 champs login et mot de passe pour se connecter et intégrez y le framework bootstrap pour une présentation plus moderne et responsive.

Nous obtenons une interface un peu plus agréable :



Bienvenue, merci de vous authentifier

Adresse mail

Mot de passe

Connexion

BTS SIO - © 2017-2018

Partie 3 : organisation du code

Nous avons jusqu'ici travaillé en MVC avec nos classes d'accès aux données qui sont notre couche Model, nos vues qui utilisent le moteur de templates Twig et notre contrôleur principal index.php.

Lorsque les applications grandissent, il est toutefois conseillé de ne pas tout centraliser dans un seul fichier index. La création de contrôleurs dédiés à telle ou telle partie des fonctionnalités est plus appropriée. Nous allons modifier notre structure afin de mieux séparer les responsabilités dans notre code.

Pour commencer ce travail, vous allez ouvrir le code fourni correspondant à la seconde version de notre ébauche d'application. Déployez le code sur votre serveur puis parcourez son arborescence.

A vous : identifiez les modifications effectuées par rapport à la première version construite. (nous en parlerons ensemble ensuite)

Nous venons de voir que le fichier index.php est réduit à son strict minimum. Les paramètres de configuration ont été délocalisés dans un fichier séparé (settings.php). Les services injectés dans l'application sont définis dans le fichier dependencies.php et les différentes routes dans le fichier routes.php. Ce dernier est assimilable à un contrôleur principal où toute l'orchestration des traitements est organisée.

Concernant nos actions gérées par le contrôleur, elles suivent toutes le schéma suivant :

```
$app->get('/users', function($request, $response, $args){  
    // On effectue une ou plusieurs requêtes  
  
    // On effectue des tests et on valide des données  
  
    // On effectue des traitements métiers spécifiques  
  
    // On met en place les données pour les passer à une vue  
  
    //...  
  
    // On appelle l'affichage d'une vue  
});
```

Imaginons une application complexe, on pourra alors vite se retrouver avec un code gérant nos routes extrêmement important ce qui le rendra difficile à maintenir pour le faire évoluer par la suite.

Il est possible de délocaliser le code de ces actions dans des classes spécifiques. Nous allons voir comment.

Le schéma de nos routes deviendra quelque chose du type :

```
$app->get('/some/path', 'ClassName:method');
```

En appliquant ces principes, nous arrivons aux fichiers suivants :

- **Le fichier *routes.php* :**

```
<?php

// Creating routes
// Psr-7 Request and Response interfaces
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

use classes\DAOUser;
use controllers\UserController;

$app->get('/', 'UserController:connect')->setName('home');
$app->post('/login', 'UserController:login')->setName('validLogin');
$app->get('/accueil', 'UserController:accueil')->setName('accueil');
```

- **Le nouveau fichier *UserController* (créé dans le répertoire *controllers*, au même niveau que le dossier *classes*) :**

```
<?php
namespace controllers;

use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

use classes\DAOUser;

class UserController{

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

    public function connect(Request $request, Response $response, $args)
    {
        $params = $request->getQueryParams();
        $message="";

        if(isset($params['error'])){
            $message = "Echec d'authentification";
        }
        return $this->container->view->render($response, 'login.html', [
            'message' => $message
        ]);
        //return $this->container->view->render($response, 'login.html');
    }

    public function login(Request $request, Response $response, $args)
    {
        $data = $request->getParsedBody();
        $donnees = [];
        $donnees['login'] = filter_var($data['login'],
FILTER_SANITIZE_STRING);
        $donnees['mdp'] = filter_var($data['mdp'], FILTER_SANITIZE_STRING);
```

```
$daoUser = new DAOUser($this->container->db);
$leUser = $daoUser->login($donnees['login'], $donnees['mdp']);

if($leUser){
    return $response->withRedirect('./accueil');
}else{
    return $response->withRedirect('./?error=invalidCredentials');
}
}

public function accueil(Request $request, Response $response, $args)
{
    return $this->container->view->render($response, 'accueil.html');
}
```

- *Le fichier dependencies.php à compléter avec :*

```
<?php
$container = $app->getContainer();

$container['UserController'] = function($c) {
    return new controllers\UserController($c);
};
```

A vous : notre code est correctement réorganisé mais vous remarquerez que la route accueil permet d'accéder à la page sans même être authentifié. Modifiez cela pour que seul un utilisateur correctement identifié puisse accéder à cette page.