Support de cours
Slim3

Support de cours Slim3

Jérôme AMBROISE

“

Ce document n’est pas un support de cours exhaustif.

C’est un guide proposant différentes étapes pour créer un projet PHP avec le micro-framework Slim.

Qu’est-ce qu’un micro-framework ?
Qu’est-ce que Slim ?

1.

Présentation générale

Présentation générale
Framework PHP

Un framework est un ensemble d'outils permettant de donner un cadre au projet. Le Framework a pour but d'accélérer le développement d'une application en proposant des implémentations aux éléments répétitifs d'un projet (structure, MVC, ...).

Présentation générale

Framework : Full-stack VS Micro

On distingue 2 types de Framework :

Micro (Slim, CodeIgniter, …) : ils proposent la mise en place des outils nécessaires au lancement du projet (PSR7, routage, conteneur).

Ils se distinguent par leur légèreté et leur flexibilité.

Présentation générale

Framework : Full-stack VS Micro

Full-stack (Symfony, Laravel, …) : en plus des outils nécessaire au lancement du projet, ils proposent des outils permettant d'accélérer le développement d'autres composants (ORM, authentification, classe abstraites, …).

Ils se distinguent par leur robustesse, leur communauté active mais aussi par leur niveau d'exigence au respect de leurs "standards".

Présentation générale
Slim Framework

Slim est un micro-framework PHP qui implemente :

- Le PSR7 : Request, Response (gestion des middlewares)
- Le routage
- Le conteneur d'injection de dépendances.

Comment créer une page web simple avec Slim Framework ?

2.

Création d’une page web

Création d'une page web
Installation

Pour utiliser Slim, nous avons besoin :

- D'un **serveur web** : embarqué, apache ou nginx
- De **composer** : gestionnaire de paquets PHP

Création d'une page web
Récupération de Slim

Documentation officielle : Slim Documentation

Pour mettre en place un projet, il faut récupérer Slim.
Deux méthodes s'offrent à nous :

- Récupérer seulement Slim
  - Création d'un dossier
  - *$ composer init*
  - *$ composer require slim/slim*
- Obtenir le squelette d'une application
  - *$ composer create-project slim/slim-skeleton project002*

Création d'une page web
Les différentes étapes

En utilisation seulement la librairie, nous devons créer l'application depuis zéro. Pour cela :

- Création d'un index où pointeront nos pages
  - /public.index.php

Dans cet index, il nous faudra :

- Créer l'application Slim
- Créer une route avec un traitement
- Lancer l'application

Création d'une page web

Créer l'application Slim

Pour créer l'application, nous allons utiliser la classe "App" de Slim, elle se trouve dans le **namespace** "Slim".

```
// Créer l'application Slim
$app = new \Slim\App();
```

Pour pouvoir utiliser une classe dans un namespace spécifique, nous allons utiliser l'**autoloader** de composer :

```
require "../vendor/autoload.php";
```

Créer une application Slim

Emplacement : “/public/index.php”

```php
<?php

require "../vendor/autoload.php";

// Créer l’application Slim
$app = new \Slim\App();
```

Création d'une page web

Créer une route avec un traitement

Slim nous donne accès à un système de routage permettant d'associer une **URL et sa méthode** à un **traitement**.

Forme générale du routage :

```
$app->#methode#(#url#, #traitement#);
```

- **Méthode** : get, post, put, delete, …
- **URL** : chaîne de caractères
- **Traitement** : callable (closure, classe, …)

Création d'une page web

Créer une route avec un traitement

Prenons un exemple :

- **Méthode** : get
- **URL** : "/hello"

```
$app->get("/hello", #traitement#);
```

Création d'une page web

Créer une route avec un traitement

Traitement : le nombre et l'ordre des arguments est défini de la façon suivante :

- Argument 1 : **La requête**
  - Correspondant à RequestInterface du PSR7
  - Implémenter par Slim (/Slim/Http)
- Argument 2 : **La réponse**
  - Correspondant à ResponseInterface du PSR7
  - Implémenter par Slim (/Slim/Http)
- Argument 3 : **Les arguments** (facultatif)
  - Contient les arguments de la requête

Création d'une page web

Créer une route avec un traitement

Traitement : l'objectif d'un traitement est de retourner une réponse (texte, HTML, JSON, …)

Dans notre exemple, nous allons "écrire" dans le corps de la réponse du texte.

- Utilisation de la Response : **$response**
- Récupération du body: **->getBody()**
- Ecriture dans le body : **->write(#chaine#)**

Exemple :

```
$response->getBody()->write('<h1>Hello !</h1>');
```

Créer une route avec un traitement

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/hello', function (RequestInterface $request, ResponseInterface $response, $args) {
   $response->getBody()->write('<h1>Hello !</h1>');
});
```

Création d'une page web
Lancer l'application

Une fois l'application créée et le traitement définie, il faut lancer l'application en renvoyant la réponse au navigateur.

Pour cela nous utilisons la méthode "run()" de l'application.

Exemple :

```
$app->run();
```

Nous pouvons alors accéder à l'URL défini pour obtenir la réponse dans le navigateur.

Lancer l'application

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/hello', function (RequestInterface $request, ResponseInterface $response, $args) {
   $response->getBody()->write('<h1>Hello !</h1>');
});

// Lancer l'application
$app->run();
```

3.

L’objet Request

Que représente l’objet Request ?
Que contient l’objet Request ?

L’objet Request
Présentation

L’objet Request représente sous forme d’objet la requête HTTP reçue par le serveur web.

Elle implémente le ServerRequestInterface du PSR7 et permet d’accéder :

- La méthode de la requête
- L’URL de la requête
- Les différents headers
- Le body de la requête

L'objet Request

La méthode de la requête

Les méthodes HTTP : GET, POST, PUT, DELETE, HEAD, PATCH, OPTION

- Pour récupérer la méthode de la requête :

```
$method = $request->getMethod();
```

- Pour tester unitairement une méthode :
  - $request->isGet()
  - $request->isPost()
  - $request->isPut()
  - $request->isDelete()
  - …

L'objet Request
L'URL de la requête

- Pour récupérer la chaine de caractère de l'URL

```
$uri = $request->getUri();
```

- Pour récupérer une partie de l'URL :
  - Scheme (e.g. http or https) : getScheme()
  - Host (e.g. example.com) : getHost()
  - Port (e.g. 80 or 443) : getPort()
  - Path (e.g. /users/1) : getPath()
  - Query string (e.g. sort=created&dir=asc): getQuery()

L'objet Request

Les différents headers

- Pour obtenir l'ensemble des headers :

```
$headers = $request->getHeaders();
```

- Pour obtenir un header particulier :

```
$headerValueArray = $request->getHeader('Accept');
$headerValueString = $request->getHeaderLine('Accept');
```

- Pour tester un header :

```
if ($request->hasHeader('Accept')) {
  // Do something
}
```

L'objet Request

Le corps de la requête

- Pour récupérer un contenu JSON, XML ou URL-encoded :

```
$parsedBody = $request->getParsedBody();
```

- Pour récupérer le corps sous forme de StreamInterface (contenu non maitrisé)

```
$body = $request->getBody();
```

- Pour récupérer les fichiers uploadés sous forme de UploadedFileInterface :

```
$files = $request->getUploadedFiles();
```

L'objet Request

Les paramètres GET et POST

- Récupérer les paramètres GET

```
$getParams = $request->getQueryParams();
```

- Pour récupérer les paramètres POST

```
$postParams = $request->getParsedBody();
```

4.

L’objet Response

Qu’est-ce qu’une Response ?
Comment modifier une Response ?

L'objet Response
Présentation

L'objet Response représente sous forme d'objet la réponse renvoyée par le serveur web suite à une requête.

Elle implémente le ResponseInterface du PSR7. La réponse peut être de plusieurs types :

- Du texte
- Du HTML
- Du JSON (XML, …)
- Une redirection
- …

L'objet Response

Le statut code

Chaque réponse HTTP possède un statut code. C'est un numéro de 3 chiffres qui permet d'identifier l'état de la réponse :

- Code de succès : >= 200 et < 300
- Code de redirection : >= 300 et < 400
- Erreurs du client : >= 400 et < 500
- Erreurs du serveur : >= 500 et < 600

L'objet Response

Le statut code

- Accéder au statut code :

```
$status = $response->getStatusCode();
```

- Modifier le statut code :

```
$newResponse = $response->withStatus(302);
```

L'objet Response

Le corps

- Accéder au corps de la réponse :

```
$body = $response->getBody();
```

- Ecrire dans le corps :

```
$body->write('Hello');
```

- Il existe d'autres méthodes pour accéder et modifier le corps :

Se renseigner sur la StreamInterface et son implémentation par Slim Stream.

L'objet Response
Retourner du JSON

- Pour renvoyer un tableau associatif en tant que réponse JSON :

```
$data = array('name' => 'Bob', 'age' => 40);
$newResponse = $oldResponse->withJson($data);
```

Remarque : Le header "Content-type" de la réponse est automatiquement adapté avec cette méthode (application/json;charset=utf-8.)

L’objet Response
Retourner une redirection

- Pour effectuer une redirection

```
return $response->withRedirect('/new-url', 301);
```

5.

Le routage

Qu’est-ce que le routage ?
Comment router différentes méthodes ?
Comment router des variables ?
Comment grouper des routes ?
Comment exporter le traitement d’une route ?

Le routage
Présentation

Le routage permet d'associer une URL à un traitement spécifique.

Nous allons voir les différentes façons "d'attraper" une URL et comment grouper ou externaliser le (ou les) traitement associés.

Le routage
Présentation

Comme nous l'avons vu plus tôt, pour afficher une page, nous avons besoin de 3 informations :

- **Méthode** : get, post, put, delete, …
- **URL** : chaîne de caractères
- **Traitement** : callable (closure, classe, …)

Intéressons-nous à ces 3 points, en commançant par la méthode.

Le routage
La méthode

Exemple précédent :

```
$app->get('/hello', function (RequestInterface $request, ResponseInterface $response, $args) {
  $response->getBody()->write('<h1>Hello !</h1>');
  return $response;
});
```

Nous interceptons la requête possédant la méthode “GET”. Il existe aussi des méthodes (fonctions) pour les autres méthodes HTTP :

- post(), put(), delete(), options(), patch()

Les arguments de ces méthodes sont identiques.

Le routage
La méthode

Nous pouvons aussi définir plusieurs méthodes acceptées en utilisant la fonction “map()”.

Nous devons allons passer en 1er argument un tableau contenant les méthodes HTTP autorisées.

Remarque : vous pouvez tester les différentes méthodes HTTP avec Postman.

Plusieurs méthodes HTTP dans une route

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->map(['GET', 'POST'],'/hello', function (RequestInterface $request, ResponseInterface $response, $args) {
  $response->getBody()->write('<h1>Hello !</h1>');
  return $response;
});

// Lancer l'application
$app->run();
```

Le routage
La méthode

Il est aussi possible d'accepter "toutes" les méthodes HTTP avec la fonction "any".

Exemple :

```
// Créer une route avec un traitement
$app->any('/hello', function (RequestInterface $request, ResponseInterface $response, $args) {
  $response->getBody()->write('<h1>Hello !</h1>');
  return $response;
});
```

Le routage
L'URL de la route

Jusqu'à lors nous avons vu comment définir l'URL d'une route avec une chaîne de caractères.

Nous pouvons aussi utiliser des variables.

Cas d'utilisation possible :

"afficher le détail d'un produit"

Les URL seront alors de la forme :

- /produit/1
- /produit/36
- /produit/144

Le routage
L'URL de la route

Les produits que nous souhaitons afficher sont stockés en base de données, nous ne savons donc pas combien il y en a lors de la création de l'application (ce nombre évolue).

- Nous ne pouvons donc pas créer autant de routes qu'il y a d'articles.

Nous allons donc définir une route unique et "variabiliser" le nombre se situant à la fin de l'URL.

L'URL sera de la forme : "/produit/#numero#"

Le routage
L’URL de la route

Pour “variabiliser” une partie de l’URL, nous utiliserons les accolades pour entourer notre variable :

“/produit/{index}”

Utiliser une variable dans une route

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/produit/{index}', function (RequestInterface $request, ResponseInterface $response, $args) {
   $response->getBody()->write('<h1>Détail d\'un produit !</h1>');
   return $response;
});

// Lancer l'application
$app->run();
```

Le routage
L’URL de la route

Remarque sur les variables dans les routes :

- Les variables de routes peuvent être multiples
- Chaque variable facultative devra être entourée de crochet ([ ])
- Les variables de routes sont stockées dans le 3ème argument de la closure “args”

Récupérer la valeur d'une variable de route

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/liste/page/[{index}]', function (RequestInterface $request, ResponseInterface $response, $args) {
    // Récupération du body
    $body = $response->getBody();
    $index = isset($args['index']) ? $args['index'] : null; // On récupère l'éventuel index

    if($index) {
        $body->write('Bienvenue sur la page ' . $index);
    } else {
        $body->write('Bienvenue sur la page 1 (par défaut)');
    }
    return $response;
});

// Lancer l'application
$app->run();
```

“

Nous attendons un index dans l’URL. Que se passe-t-il si nous entrons une chaîne de caractères contenant autre chose que des chiffres ?

Le routage
L'URL de la route

Il est possible d'appliquer des contraintes sur nos variables d'URL en utilisant les expressions régulières.

Pour cela nous suffisons notre variable avec les deux points (:) suivi de notre RegEx.

Exemple :

```
'/liste/page/[{index:\d+}]'
```

Remarque : si l'URL ne correspond pas à notre expression, le routeur cherchera une autre route. Si aucune route n'est trouvée, nous tombons sur une page 404.

Appliquer une RegEx sur une variable de route

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/liste/page/[{index:\d+}]', function (RequestInterface $request, ResponseInterface $response, $args) {
   // Récupération du body
   $body = $response->getBody();
   $index = isset($args['index']) ? $args['index'] : null; // On récupère l'éventuel index

   if($index) {
      $body->write('Bienvenue sur la page ' . $index);
   } else {
      $body->write('Bienvenue sur la page 1 (par défaut)');
   }
   return $response;
});

// Lancer l'application
$app->run();
```

Le routage

Nommer une route

Il est possible de nommer une route afin de pouvoir la retrouver par la suite.

- Cela nous servira pour créer des liens entre les différentes pages.
- L'intérêt est de pouvoir changer l'URL de la route sans changer son nom.

On utilise la méthode "setName(#name#)" après le définition de la route.

Nommer une route

Emplacement : “/public/index.php”

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l’application Slim
$app = new \Slim\App();

// Créer une route avec un traitement
$app->get('/produit/{index:\d+}', function (RequestInterface $request, ResponseInterface $response, $args) {

   $response->getBody()->write('Détail du produit ' . $args['index']);
   return $response;
})->setName('product_show');

// Lancer l’application
$app->run();
```

Le routage

Grouper des routes

Nous pouvons grouper des routes car elles peuvent partager :

- **Le début de leur URL** : on peut alors regrouper les routes afin d'y voir plus claire
- **Un traitement particulier** : on peut alors utiliser un ou plusieurs middlewares sur l'ensemble du groupe (nous le verrons par la suite)

Pour cela nous utilisons la méthode "group()".

Grouper des routes

Emplacement : "/public/index.php"

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer un groupe de routes
$app->group('/produit', function() {
   // Liste des produits
   $this->get('/liste', function(RequestInterface $request, ResponseInterface $response, $args){
      $response->getBody()->write('Liste des produits');
      return $response;
   })->setName('product_list');

   // Détail d'un produit
   $this->get('/{index:\d+}', function (RequestInterface $request, ResponseInterface $response, $args) {
      $response->getBody()->write('Détail du produit ' . $args['index']);
      return $response;
   })->setName('product_show');
});

// Lancer l'application
$app->run();
```

Le routage

Externaliser le traitement d'une route

Définir l'ensemble des traitements dans le fichier "index.php" va s'avérer très lourd.

Nous pouvons externaliser le traitement des routes dans une couche "Contrôleur".

La couche contrôleur est un ensemble de classes (contrôleur) possédant des traitements liés à des routes (des actions de contrôleur).

Le routage

Externaliser le traitement d'une route

Pour se faire, nous allons

- Créer un dossier "/app"
- Créer un dossier "Controller" dans "/app"
- Ajouter un namespace dans l'autoload de composer afin de pouvoir charger nos classes
- Créer un fichier "ProductController.php" : ce sera notre contrôleur
- Créer des méthodes dans le contrôleur :
  - list() : action traitant la liste des produits
  - show() : action traitant le détail du produit

Le routage

Création d'un espace de nom

La création d'un namespace passe par la création d'un dossier lié à un espace de nom.

Nous allons créer un dossier "/app" et le lier au namespace "App".

Nous pourrons alors le renseigner à composer qui chargera nos classes au besoin.

Renseigner un namespace

Emplacement : "/composer.json"

```
{
  "name": "demodoranco/slim3",
  "description": "Projet exposant les fonctionnalités du micro-framework Slim dans sa version 3",
  "type": "project",
  "authors": [
    {
      "name": "Jerome AMBROISE",
      "email": "ambroise.jerome.formation@gmail.com"
    }
  ],
  "require": {
    "slim/slim": "^3.11"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app/"
    }
  }
}
```

Le routage

Création d'un espace de nom

Pour que composer puisse charger les classes issues du nouveau namespace, il faut exécuter une commande :

$ composer dump-autoload

Nous pouvons alors créer et appeler des classes dans ce namespace.

Le routage
Création d'un contrôleur

Créons notre contrôleur "ProductController.php" dans le dossier "/app/Controller".

On ajoute alors les traitements précédemment créés "/public/index.php" :

- list() : action traitant la liste des produits
- show() : action traitant le détail du produit

Création d'un contrôleur de produits (liste, détail)

Emplacement : "/app/Controller/ProductController.php"

```php
<?php

namespace App\Controller;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

/**
* Class ProductController : contrôleur traitant des produits
*/
class ProductController {
   /**
    * Affiche la liste des produits
    */
   public function liste(RequestInterface $request, ResponseInterface $response){
      $response->getBody()->write('Liste des produits');
      return $response;
   }

   /**
    * Affiche le détail d'un produit par rapport à son "index"
    */
   public function show(RequestInterface $request, ResponseInterface $response, $args) {
      $response->getBody()->write('Détail du produit ' . $args['index']);
      return $response;
   }
}
```

Le routage

Création d’un contrôleur

Maintenant que notre contrôleur est créé, il faut lier les routes à ses méthodes.

Pour cela nous allons modifier “/public/index.php”

Lier une route à une action de contrôleur

Emplacement : "/public/index.php"

```php
<?php

use App\Controller\ProductController;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Créer un groupe de routes
$app->group('/produit', function() {
    // Liste des produits
    $this->get('/liste', ProductController::class . ':liste')->setName('product_list');

    // Détail d'un produit
    $this->get('/{index:\d+}', ProductController::class . ':show')->setName('product_show');
});

// Lancer l'application
$app->run();
```

6.

Les vues avec le moteur de template TWIG

Qu’est-ce que la couche vue ?
Qu’est-ce que TWIG ?
Comment mettre en place TWIG avec Slim ?
Qu’est-ce qu’un conteneur d’injection de dépendances ?

Les vues avec TWIG

La couche vue

La couche vue représente la couche de présentation HTML.

Elle combine du PHP et du HTML pour finalement retourner seulement du HTML.

Les moteurs de templates nous permettent :

- D'alléger la syntaxe PHP-HTML
- Mettre en place des layouts
- …

Les vues avec TWIG

La couche vue

En tant que Micro-Framework, Slim n'intègre pas nativement de moteur de templates.

Il convient d'en choisir un (si besoin) et de l'intégrer au projet.

Nous allons utiliser TWIG : documentation officielle de TWIG.

Les vues avec TWIG

Présentation de TWIG

TWIG est un moteur de template PHP.

- **Rapide** : la surcouche est limitée au maximum
- **Sécurisé** : échappement automatique
- **Extensible** : possibilité d'ajouter de nouvelles fonctions, filtres, …

La syntaxe de TWIG ne veut **simple**.

Il nous permettra aussi de mettre en place un système de **layout** facilement.

Les vues avec TWIG

Intégration de TWIG

Pour intégrer TWIG dans Slim, il existe un package qui va nous faciliter les choses : slim/twig-view.

Installons ce package dans notre projet :

$ composer require slim/twig-view

Les vues avec TWIG

Intégration de TWIG

TWIG va nous permettre de rendre des vues au navigateur.

Nous aurons donc besoin qu'il soit accessible un peu partout dans notre application.

Pour cela, nous allons l'enregistrer dans le **conteneur d'injection de dépendances** afin de pouvoir l'utiliser dans notre contrôleur.

Commençons par voir ce qu'est un **DIC**.

Les vues avec TWIG

Conteneur d'injection de dépendances

- Le conteneur d'injection de dépendances associe une clef à une valeur.
- La clef est une chaîne de caractères
- La valeur peut être de différents types (chaîne de caractères, fonctions, classes, …)

L'objectif est de pouvoir récupérer les valeurs définies grâce à leurs clefs.

A quoi cela sert-il ? Faire de l'injection de dépendances !

Les vues avec TWIG

Les injections de dépendances

L'injection de dépendances prend effet quand une classe A en a besoin d'une autre (ou plusieurs).

Pour cela, la classe A met en la (ou les) classe(s) dont elle a besoin en tant qu'argument de son constructeur.

Prenons un exemple !

Les vues avec TWIG

Les injections de dépendances

Nous souhaitons mettre en place un “calculatrice”.

Pour cela nous créons 2 classes :

- **Calculator** : se chargera de l’affichage
- **Maths** : se chargera de faire les calculs

La Classe Calculator a donc besoin de la classe Maths pour fonctionner.

Nous allons donc créer un constructeur dans la classe Calculator qui prendra en argument une instance de la classe Maths.

Création d'une classe temporaire "Maths"

Emplacement : "/app/classes/Maths.php"

```php
<?php

namespace App\classes;


class Maths {
   public function addition($a, $b)
   {
      return $a + $b;
   }
}
```

Création d'une classe temporaire "Calculator"

Emplacement : "/app/classes/Calculator.php"

```php
<?php

namespace App\classes;

class Calculator {
    /**
     * @var Maths
     */
    private $maths;

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

    public function displayAddition(): int
    {
        return $this->maths->addition(35, 46);
    }
}
```

Les vues avec TWIG

Les injections de dépendances

Désormais si nous voulons utiliser la classe “Calculator”, nous devons :

- Instancier une classe Maths
- Instancier une classe Calculator en lui donnant l’instance de la classe Maths.

```
$maths = new Maths();
$calculator = new Calculator($maths);
$calculator->displayAddition();
```

Les vues avec TWIG

Les injections de dépendances

Si nous voulons utiliser la classe Calculator dans un contrôleur, nous allons devoir à chaque fois définir cette logique dans le constructeur.

Le conteneur d'injection de dépendances nous permet de définir cette logique une seule fois, et de l'utiliser autant de fois que nous voulons.

Nous allons donc :

- Modifier notre contrôleur pour utiliser Calculator
- Définir dans le conteneur la façon dont le système doit instancier Calculator

Utilisation de Calculator par ProductController

Emplacement : "/app/Controller/ProductController.php"

```php
<?php

namespace App\Controller;

use App\classes\Calculator;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

class ProductController {

  private $calculator;

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

  public function liste(RequestInterface $request, ResponseInterface $response){
     var_dump($this->calculator->displayAddition());
     $response->getBody()->write('Liste des produits');
     return $response;
  }

  public function show(RequestInterface $request, ResponseInterface $response, $args) { … }
}
```

Définition du conteneur d'injection de dépendances

Emplacement : "/public/index.php"

```php
<?php

use App\classes\Calculator;
use App\classes\Maths;
use App\Controller\ProductController;

require "../vendor/autoload.php";

// Créer l'application Slim
$app = new \Slim\App();

// Container
$container = $app->getContainer();

$container[Maths::class] = function () {
  return new Maths();
};
$container[Calculator::class] = function ($container) {
  return new Calculator($container->get(Maths::class));
};
$container[ProductController::class] = function ($container) {
  return new ProductController($container->get(Calculator::class));
};
// Créer un groupe de routes
...
// Lancer l'application
$app->run();
```

Les vues avec TWIG

Ré-organisation du projet

Nous pouvons alors appeler les 3 classes, le conteneur d'injection de dépendances les instanciera en conséquence.

Notre "index.php" commence à contenir beaucoup de logiques, nous allons isoler les logiques dans des "require" :

- Définition du conteneur d'injection de dépendances
- Définitions des routes

Externalisation de la configuration (½)

Emplacement : "/config/container.php"

```php
<?php

// Container
use App\classes\Calculator;
use App\classes\Maths;
use App\Controller\ProductController;

$container = $app->getContainer();

$container[Maths::class] = function () {
  return new Maths();
};
$container[Calculator::class] = function ($container) {
  return new Calculator($container->get(Maths::class));
};
$container[ProductController::class] = function ($container) {
  return new ProductController($container->get(Calculator::class));
};
```

Externalisation de la configuration (½)

Emplacement : "/config/routing.php"

```php
<?php

use App\Controller\ProductController;

// Créer un groupe de routes

$app->group('/produit', function() {
   // Liste des produits
   $this->get('/liste', ProductController::class . ':liste')->setName('product_list');

   // Détail d'un produit
   $this->get('/{index:\d+}', ProductController::class . ':show')->setName('product_show');
});
```

Externalisation de la configuration (½)

Emplacement : "/public/index.php"

```php
<?php

require "../vendor/autoload.php";

// Configuration de l'application
$config = [
    'settings' => [
        'displayErrorDetails' => true
    ]
];

// Créer l'application Slim
$app = new \Slim\App($config);

// Configuration du conteneur d'injection de dépendances
require '../config/container.php';

// Configuration des routes
require '../config/routing.php';

// Lancer l'application
$app->run();
```

Les vues avec TWIG

Ré-organisation du projet

Nous pouvons en faire de même pour la configuration de l’application.

Externalisation de la configuration (2/2)

Emplacement : "/app/config/app-config.php"

```php
<?php

// Configuration de l'application

return [
    'settings' => [
        'displayErrorDetails' => true
    ]
];
```

Externalisation de la configuration (2/2)

Emplacement : "/public/index.php"

```php
<?php

require "../vendor/autoload.php";

// Configuration de l'application
$config = require('../config/config-app.php');

// Créer l'application Slim
$app = new \Slim\App($config);

// Configuration du conteneur d'injection de dépendances
require '../config/container.php';

// Configuration des routes
require '../config/routing.php';

// Lancer l'application
$app->run();
```

Les vues avec TWIG

Intégration de TWIG

Nous pouvons revenir au sujet initial : TWIG.

- Nous allons modifier le conteneur d'injection de dépendances pour définir la manière "d'instancier" TWIG.
- Nous aurons besoin de sa méthode "render()" dans les contrôleurs pour rendre des vues TWIG.

Nous allons donc suivre la même logique qu'avec "Maths" et "Calculator"

Les vues avec TWIG

Intégration de TWIG

Modification du conteneur :

- Ajout d'une clef "view"
- La valeur de clé nous est donnée par la documentation officielle, nous devons tout de même renseigner certaines valeurs

Mise en place de TWIG - Conteneur

Emplacement : "/config/container.php"

```php
<?php

use App\Controller\ProductController;

$container = $app->getContainer();


// TWIG
$container['view'] = function ($container) {
    $view = new \Slim\Views\Twig(dirname(__DIR__) . '/templates', [
        //'cache' => 'path/to/cache'
        'cache' => false
    ]);

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

    return $view;
};

// Contrôleur des produits
$container[ProductController::class] = function ($container) {
    return new ProductController($container->get('view'));
};
```

Les vues avec TWIG

Intégration de TWIG

Création de la vue :

- Nouveau fichier : “/templates/index.twig”

Modification du contrôleur :

- On injecte TWIG dans le constructeur
- On appelle la méthode “render” de TWIG en lui passant :
  - La réponse
  - Le chemin de la vue

1ère vue TWIG

Emplacement : "/templates/index.twig"

```
<!doctype html>
<html lang="fr-FR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Liste des produits</h1>
</body>
</html>
```

Utilisation de TWIG dans un contrôleur

Emplacement : "/app/Controller/ProductController.php"

```php
<?php

namespace App\Controller;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Slim\Views\Twig;

class ProductController {

  /** @var Twig  */
  private $twig;

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

  /**
   * Affiche la liste des produits
   */
  public function liste(RequestInterface $request, ResponseInterface $response){
      $this->twig->render($response, 'index.twig');
  }

  /** Affiche le détail d'un produit par rapport à son "index"  */
  public function show(RequestInterface $request, ResponseInterface $response, $args) { … }
}
```

Les vues avec TWIG

Intégration de TWIG

Mettons en place le même système pour le détail d'un produit.

Problématique : comment passer une variable à la vue ?

La méthode "render" de TWIG accepte un 3ème paramètre qui est un tableau associatif pour passer des valeurs à la vue.

Les vues avec TWIG

Intégration de TWIG

Problématique : comment afficher une variable dans la vue ?

L'affichage de variables dans la vue se fait via les double accolades qui entourent la variable à afficher.

Exemple (variable "index") :

```
{{ index }}
```

Envoi d'une variable à la vue

Emplacement : "/app/Controller/ProductController.php"

```php
<?php

namespace App\Controller;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Slim\Views\Twig;

/**
* Class ProductController : contrôleur traitant des produits
*/
class ProductController {

  ...

  /**
   * Affiche le détail d'un produit par rapport à son "index"
   */
  public function show(RequestInterface $request, ResponseInterface $response, $args) {
      return $this->twig->render($response, 'product/show.twig', [
          "index" => $args['index']
      ]);
  }
}
```

Affichage d'une variable dans une vue TWIG

Emplacement : "/templates/index.twig"

```
<!doctype html>
<html lang="fr-FR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
      content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <h1>Détail du produit {{ index }}</h1>
</body>
</html>
```

Les vues avec TWIG
Syntaxe de TWIG

Voyons la syntaxe de TWIG

Syntaxe	Explication
{{ … }}	Affichage d'une variable ou le résultat d'une expression dans le modèle.
{% … %}	Contrôle la logique du modèle; Il est utilisé pour exécuter des instructions telles que des boucles for pour par exemple.
{# … #}	Permet d'inclure des commentaires (équivalent PHP : /* … */)

Les vues avec TWIG

Syntaxe de TWIG : les conditions

Utilisation du if...elseif…else :

```
{% if kenny.sick %}
   Kenny is sick.
{% elseif kenny.dead %}
   You killed Kenny! You bastard!!!
{% else %}
   Kenny looks okay --- so far
{% endif %}
```

Les vues avec TWIG

Syntaxe de TWIG : les conditions

Bouche équivalente au foreach :

```
<ul>
  {% for user in users %}
     <li>{{ user.username }}</li>
  {% endfor %}
</ul>
```

Les vues avec TWIG

Debugguer une variable

L'équivalent du var_dump :

```
{{ dump(index) }}
```

Remarque : il faut ajouter l'extension "debug" à TWIG. Pour cela, on modifie le conteneur.

Ajout de l'extension de debug de TWIG

Emplacement : "/config/container.php"

```php
<?php

use App\Controller\ProductController;

$container = $app->getContainer();

// TWIG
$container['view'] = function ($container) {
    $view = new \Slim\Views\Twig(dirname(__DIR__) . '/templates', [
        //'cache' => 'path/to/cache'
        'cache' => false,
        'debug' => true
    ]);

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

    return $view;
};

// Contrôleur des produits
$container[ProductController::class] = function ($container) {
    return new ProductController($container->get('view'));
};
```

Les vues avec TWIG

Autre fonctionnalités de TWIG

Il existe aussi des filtres et des fonctions spécifiques à TWIG, les fonctions sont documentées sur le site officiel, nous ferons le point sur les fonctions/filtres quand nous les utiliserons.
Si besoin, n'hésitez pas à me demander des précisions sur une fonction/filtre spécifique.

Nous allons désormais mettre en place un "layout" qui nous permettra de stocker des éléments répétitifs tels que les menus, footer, sidebar, … .

Les vues avec TWIG

Mise en place d'un layout

TWIG propose un système de "**block**" pour construire une page.
Le nom des blocks est libre et doit être défini par le développeur.

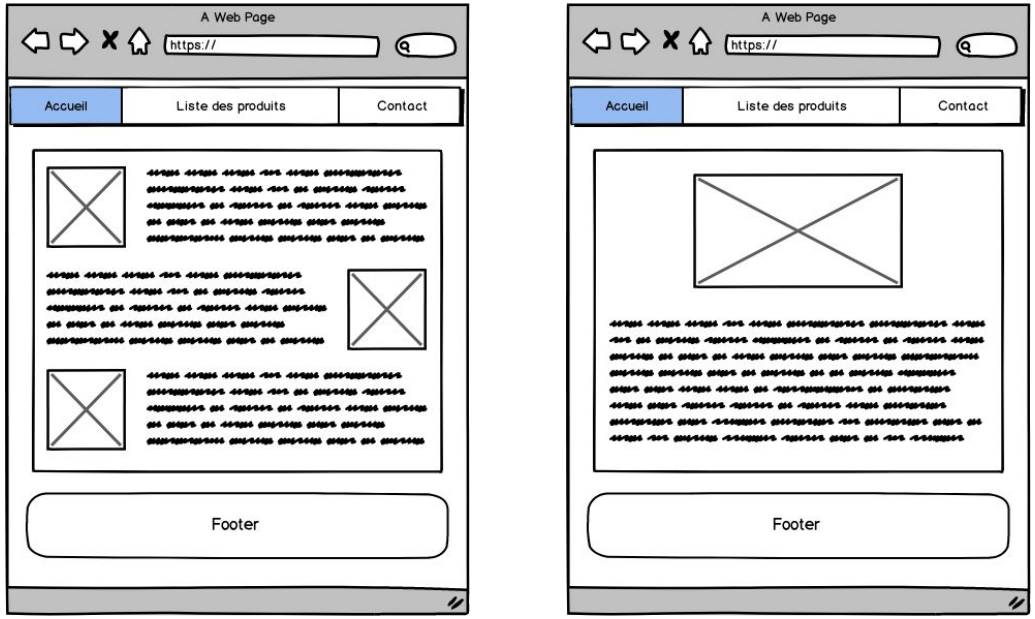
Qu'est-ce qu'un block ? Un ensemble PHP(TWIG) + HTML

Comment diviser une page en block ? De façon **logique** en déterminer les différentes "parties" de votre page, à vous de juger la granularité selon les besoins.

Les vues avec TWIG

Mise en place d'un layout

Imaginons un cas simple :



Les vues avec TWIG

Mise en place d'un layout

Dans cet exemple, nous distinguons 3 zones :

- Le menu (fixe)
- La zone principale (zone variable selon la page)
- Le footer (fixe)

Nous mettrons donc dans le layout générique, les deux zones fixes (menu et footer).

Nous utiliserons le layout dans nos 2 pages en personnalisant la zone variable.

Les vues avec TWIG

Création du layout

Créons donc un fichier “/templates/layout.twig”.
Nous allons créer les 3 blocks :

- Le block “menu” : contient le HTML du menu
- Le block “main” : vide (il sera rempli par la vue enfant)
- Le block “footer” : contient le HTML du footer

Pour créer un block nous utiliserons la syntaxe suivante :

```
{% block #name# %}#contenu#{% endblock %}
```

Création des blocks dans le layout TWIG

Emplacement : "/templates/layout.twig"

```
<!doctype html>
<html lang="fr-FR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
      content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  {% block menu %}{% endblock %}
  {% block main %}{% endblock %}
  {% block footer %}{% endblock %}
</body>
</html>
```

Les vues avec TWIG

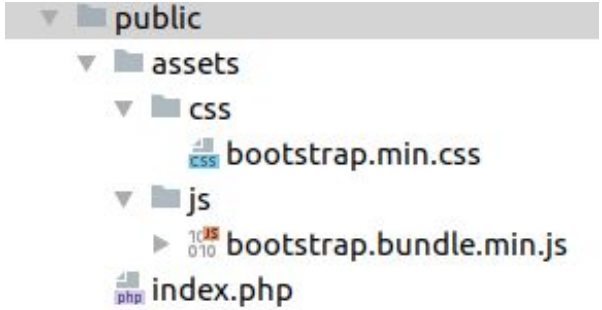
Création du layout

Nous allons remplir les blocks “menu” et “footer”.

Nous allons utiliser le framework front “Bootstrap” pour gagner du temps sur le CSS/JS.

Pour cela nous allons :

- télécharger sur le site officiel les fichiers CSS/JS compilés
- Intégrer les CSS/JS au projet
- Les intégrer dans le layout



Appel CSS/JS de Bootstrap dans le layout

Emplacement : "/templates/layout.twig"

```
<!doctype html>
<html lang="fr-FR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
      content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="/assets/css/bootstrap.min.css">
  <title>Document</title>
</head>
<body>
  {% block menu %}{% endblock %}
  {% block main %}{% endblock %}
  {% block footer %}{% endblock %}

  <script src="/assets/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Les vues avec TWIG

Création du layout

On peut désormais récupérer des exemples de Bootstrap pour créer le menu et le footer.

Intéressons-nous désormais à l'utilisation de ce layout par des vues enfants.

L'idée est "d'étendre" le layout :

- On récupère l'ensemble des blocks
- On peut personnaliser le contenu des blocks

Pour étendre une vue, on utilise le mot-clef "extends".

Utilisation d’un layout par une vue enfant

Emplacement : “/templates/product/index.twig”

```
{% extends 'layout.twig' %}

{% block main %}
   <h1>Liste des produits</h1>
{% endblock %}
```

8.

Utilisation de PDO

Comment gérer l'interfaçage avec une base de données ?
Qu’est-ce qu’un Repository ?
Comment mettre en place une requête de récupération et l’afficher dans la vue ?
Comment mettre en place une requête préparée et l’afficher dans la vue ?

Utilisation de PDO

Présentation

Nous allons nous interfacer avec une base de données avec PDO.

Nous allons voir comment organiser notre projet afin d'avoir une structure cohérente, maintenable et réutilisable.

Utilisation de PDO

Organisation

Nous allons d'abord créer une classe "Database" qui sera la classe qui consultera directement la base de données.

Ce sera la seule qui dépendra de PDO. De ce fait, on pourra imaginer de changer de source de données (PostgreSQL, MongoDB, …) en modifiant uniquement cette classe.

Utilisation de PDO
Organisation

Pour l'instant la classe "Database" nous servira à :

- établir une connexion avec notre base de données

Pour cela elle aura besoin des informations de connexion (host, dbname, user, pass).

Nous allons donc renseigner ses paramètres dans notre configuration.

Ajout des informations de BDD dans la configuration

Emplacement : "/config/config-app.php"

```php
<?php

// Configuration de l'application

return [
    'settings' => [
        'displayErrorDetails' => true,
        'database' => [
            'host'      => '#hote#',
            'dbname'    => '#dbname#',
            'user'      => '#user#',
            'password'  => '#pass#'
        ]
    ]
];
```

Utilisation de PDO

Création de la classe “Database"

Nous allons créer notre classe “Database” :

- Le constructeur prendra en entrée un tableau de paramètres (ceux de la connexion)
- Une propriété “pdo” permettra de stocker la connexion active
- Une méthode “connect()” permettra d’initialiser la connexion
- La méthode “connect()” sera appelée pendant la construction de l’objet

Classe de gestion de la BDD : connexion (1/2)

Emplacement : "/app/Generic/Database.php"

```php
<?php

namespace App\Generic;

use PDO;
use PDOException;

class Database
{
  /**  @var array : Tableau d'options (host, dbname, user, pass) */
  private $settingsBDD;
  /**   @var \PDO */
  private $pdo;

  public function __construct(array $settingsBDD)
  {
    $this->settingsBDD = $settingsBDD;
    $this->connect();
  }

  …

  public function __destruct()
  {
    $this->pdo = null;
  }
}
```

Méthode de connexion à la BDD

Emplacement : "/app/Generic/Database.php"

```php
/**
* Connexion à la BDD avec PDO
*/
public function connect(): void
{
   try {
      $this->pdo = new PDO('mysql:host='.
         $this->settingsBDD['host']
         .';dbname=' . $this->settingsBDD['dbname'],
         $this->settingsBDD['user'],
         $this->settingsBDD['password'],
         [
            PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC,
            PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"
         ]
      );
   } catch (PDOException $e) {
      print "Erreur !: " . $e->getMessage() . "<br/>";
      die();
   }
}
```

Utilisation de PDO

Création de la classe “Database"

Nous voudrions désormais utiliser cette classe dans notre contrôleur afin de récupérer les produits de la base de données.

Pour cela, nous devons renseigner au conteneur d’injection de dépendances comment instancier la classe “Database”.

Ajout de la classe “Database” au conteneur

Emplacement : “/config/container.php”

```
...

// Database
$container[Database::class] = function ($container) {
    return new Database($container['settings']['database']);
};
```

Utilisation de PDO

Présentation des Repository

La logique de récupération des données ne devraient pas être dans le contrôleur mais isolée dans une couche spécifique.

Nous allons donc utiliser des “Repository” pour mettre en place notre couche d’accès aux données.

L’intérêt est double :

- Alléger le contrôleur
- Pouvoir mettre en place des Repository génériques (CRUD, …) et de la spécialiser si besoin.

Processus d'accès aux données

Rôle :
Transformer une requête en réponse

ProductController
Demande tous les produits au ProductRepository

Rôle :
Gérer les requêtes SQL

ProductRepository
Créé la requête SQL correspondante et demande à Database de l'exécuter

Rôle :
Interroger la BDD

Database
Exécute la requête demandée

Utilisation de PDO

Création du Repository

Nous allons créer un Repository par table, nous trouvons ensuite une logique commune pour créer un Repository générique.

Pour l'instant notre "ProductRepository" :

- Prendra une instance de la "Database" à la construction
- Possédera une constante "tableName" qui stockera le nom de la table
- Possédera une méthode "findAll()" pour trouver tous les enregistrements des produits

Création du Repository des produits

Emplacement : "/app/Repository/ProductRepository.php"

```php
<?php

namespace App\Repository;

use App\Entity\Product;
use App\Generic\Database;

class ProductRepository
{
    /** @var string : Nom de la table en BDD */
    const tableName = 'product';

    /** @var Database */
    private $database;

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

    /** @return array|null */
    public function findAll(): ?array
    {
        ...
    }
}
```

Utilisation de PDO

Mise à jour du conteneur

Notre contrôleur utilisera le Repository pour interroger la BDD.

Nous allons devoir mettre à jour le conteneur :

- Instanciation du ProductRepository
  - Instance de Database
- Instanciation du ProductController
  - Instance de ProductRepository

Création du Repository des produits

Emplacement : "/app/Repository/ProductRepository.php"

```php
<?php

...

// Database
$container[Database::class] = function ($container) {
    return new Database($container['settings']['database']);
};

// Repository
$container[ProductRepository::class] = function ($container) {
    return new ProductRepository($container->get(Database::class));
};

// Contrôleur des produits
$container[ProductController::class] = function ($container) {
    return new ProductController(
        $container->get('view'),
        $container->get(ProductRepository::class)
    );
};
```

Utilisation de PDO

Gestion de PDO::query()

Nous pouvons désormais mettre en place la logique pour récupérer nos produits.

- Définition de la logique dans Database :
  - query()
    - Prend en paramètre la requête à exécuter
    - Prend en paramètre facultatif le nombre de la classe (pour le fetchMode)
    - Retourne le résultat

Méthode “query” de Database

Emplacement : “/app/Generic/Database.php”

```php
<?php

namespace App\Generic;

use App\Entity\Product;
use PDO;
use PDOException;

class Database
{
  …
   /**
    * Exécute la requête sur la BDD
    * @param string $query : requête à exécuter
    * @param string $tableName : la classe pour le fetch sinon retourne un tableau
    * @return array|null
    */
   public function query(string $query, string $tableName = null): ?array
   {
      $statement = $this->pdo->query($query);
      if($tableName) {
         $className = "App\Entity\\" . ucfirst($tableName);
         $statement->setFetchMode(PDO::FETCH_CLASS, $className);
      }
      return $statement->fetchAll();
   }
}
```

Utilisation de PDO

Repository : trouver les produits

Ensuite on définit la logique de ProductRepository

- findAll()
  - Ne prend pas de paramètres
  - Appelle la méthode “query” de Database
  - Retourne le résultat

Méthode “findAll” de ProductRepository

Emplacement : “/app/Repository/ProductRepository.php”

```php
<?php

namespace App\Repository;

use App\Entity\Product;
use App\Generic\Database;

class ProductRepository
{
    /**
     * @var string : Nom de la table en BDD
     */
    const tableName = 'product';

            …

    /**
     * @return array|null
     */
    public function findAll(): ?array
    {
        return $this->database->query("SELECT * FROM product", self::tableName);
    }
}
```

Utilisation de PDO

Utilisation du Repository dans le contrôleur

On peut alors appeler la méthode “findAll()” depuis le ProductController et retourner le résultat à la vue.

Appel de “findAll” depuis ProductController

Emplacement : “/app/Controller/ProductController.php”

```php
<?php

namespace App\Controller;

use App\Repository\ProductRepository;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Slim\Views\Twig;

class ProductController {
    /**
     * @var Twig
     */
    private $twig;

    /**
     * Affiche la liste des produits
     * @param RequestInterface $request
     * @param ResponseInterface $response
     * @return ResponseInterface
     */
    public function liste(RequestInterface $request, ResponseInterface $response){
        $products = $this->repository->findAll();
        return $this->twig->render($response, 'product/index.twig', [
            'products' => $products
        ]);
    }
}
```

Utilisation de PDO

Affichage des produits dans la vue

On peut enfin adapter la vue à nos besoins : on a désormais accès à une variable “products” qui est un tableau d’objet de Product.

On peut alors boucler dedans afficher les informations qui nous intéressent.

Affichage de la liste des produits

Emplacement : "/templates/product/index.twig"

```
{% extends 'layout.twig' %}

{% block main %}
  <main class="container">
    <h1>Liste des produits</h1>

    <div class="album py-5 bg-light">
      <div class="container">

        <div class="row">
          {% for product in products %}
            <div class="col-md-4">
              <div class="card mb-4 shadow-sm">
                <div class="card-body">
                    <h5 class="card-title">{{ product.name }}</h5>
                    <p class="card-text">{{ product.description }}</p>
                </div>
              </div>
            </div>
          {% endfor %}
        </div>
      </div>
    </div>
  </main>
{% endblock %}
```

Utilisation de PDO
Extensions TWIG

Lors de l'affichage des produits dans la liste, nous voyons apparaître l'ensemble de la description du produit.

Nous souhaiterions pouvoir afficher seulement les premiers caractères de la description.

Nous allons pour cela créer un filtre personnalisé qui contiendra la logique affichant le "résumé" d'une chaîne de caractères.

Utilisation de PDO
Filtre personnalisé TWIG

TWIG nous permet de créer des filtres.

Pour cela nous devons :

- Créer une classe qui étend la classe de base des extensions
- Créer la logique du filtre dans la classe
- Ajouter l'extension à l'initialisation de TWIG.

Création d'un filtre personnalisé

Emplacement : "/app/Twig/StringFilter.php"

```php
<?php

namespace App\Twig;

use Slim\Views\TwigExtension;

class StringFilter extends TwigExtension
{

  public function getFilters()
  {
     return [
        new \Twig_SimpleFilter('resume', [$this, 'resume'])
     ];
  }

  public function resume(string $str, $length = 150)
  {
     $strToReturn = substr($str, 0, $length);
     $lastSpacePosition = strrpos($strToReturn, '.');
     if($lastSpacePosition) {
        $strToReturn = substr($strToReturn, 0, $lastSpacePosition) . '...';
     }
     return $strToReturn;
  }
}
```

Ajout d'une extension TWIG lors de l'initialisation

Emplacement : "/config/container.php"

```php
<?php

use App\Controller\ProductController;
use App\Generic\Database;
use App\Repository\ProductRepository;

$container = $app->getContainer();

// TWIG
$container['view'] = function ($container) {
    $view = new \Slim\Views\Twig(dirname(__DIR__) . '/templates', [
        //'cache' => 'path/to/cache'
        'cache' => false,
        'debug' => true
    ]);

    // Extension dump
    $view->addExtension(new Twig_Extension_Debug());
    // Ajout d'extensions
    $basePath = rtrim(str_ireplace('index.php', '', $container->get('request')->getUri()->getBasePath()), '/');
    $view->addExtension(new Slim\Views\TwigExtension($container->get('router'), $basePath));
    $view->addExtension(new \App\Twig\StringFilter($container->get('router'), $basePath)); // resume

    return $view;
};
 …
```

Utilisation d'un filtre dans TWIG

Emplacement : "/templates/product/index.twig"

```
{% extends 'layout.twig' %}

{% block main %}
  <main class="container">
    <h1>Liste des produits</h1>

    <div class="album py-5 bg-light">
      <div class="container">

        <div class="row">
          {% for product in products %}
            <div class="col-md-4">
              <div class="card mb-4 shadow-sm">
                <div class="card-body">
                  <h5 class="card-title">{{ product.name }}</h5>
                  <p class="card-text">{{ product.description | resume }}</p>
                </div>
              </div>
            </div>
          {% endfor %}
        </div>
      </div>
    </div>
  </main>
{% endblock %}
```

Utilisation de PDO

Détail d'un produit

Nous allons désormais passer à l'affichage du détail d'un produit selon son "id" :

- Création d'une méthode dans la "Database" pour gérer les requêtes préparées
- Création d'une méthode dans le Repository pour définir la requête SQL
- Modification de la méthode du contrôleur gérant le détail du produit
- Modification de la vue du détail du produit

Méthode générique de requête préparées

Emplacement : "/app/Generic/Database.php"

```php
<?php
namespace App\Generic;

use App\Entity\Product;
use PDO;
use PDOException;

class Database
{
    …
    /**
     * @param string $query
     * @param string|null $tableName
     * @param array|null $params
     * @return mixed
     */
    public function prepared(string $query, string $tableName = null, array $params = [])
    {
        $statement = $this->pdo->prepare($query);
        if($tableName) {
            $className = "App\Entity\\" . ucfirst($tableName);
            $statement->setFetchMode(PDO::FETCH_CLASS, $className);
        }
        $statement->execute($params);
        return $statement->fetch();
    }
}
```

Méthode “findById” de ProductRepository

Emplacement : “/app/Repository/ProductRepository.php”

```php
<?php

namespace App\Repository;

use App\Entity\Product;
use App\Generic\Database;

class ProductRepository
{
    …

    /**
     * @param int $id : id du tuple à trouver
     * @return Product|null
     */
    public function findById(int $id): ?Product
    {
        $theoricProduct = $this->database->prepared(
            "SELECT * FROM product WHERE id = ?",
            ProductRepository::tableName,
            [$id]
        );
        return $theoricProduct ? $theoricProduct : null;
    }
}
```

Appel de “findById” depuis ProductController

Emplacement : “/app/Controller/ProductController.php”

```php
<?php

namespace App\Controller;

use App\Repository\ProductRepository;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Slim\Views\Twig;

class ProductController {
  …

  public function show(RequestInterface $request, ResponseInterface $response, $args) {
    // On récupère le produit
    $product = $this->repository->findById($args["index"]);

    // Si pas de produit => page 404
    if(!$product) {
      // ##todo : implémenter la 404 dans une classe générique
      throw new \Exception("Pas de produit trouvé");
    }

    return $this->twig->render($response, 'product/show.twig', [
      "product" => $product
    ]);
  }
}
```

Affichage du détail du produit

Emplacement : "/templates/product/show.twig"

```
{% extends 'layout.twig' %}

{% block main %}
  <main class="container">

    <h1>
      {{ product.name }}
      <small><span class="badge badge-primary">{{ product.price }}€</span></small>
    </h1>
    <p class="text-muted">Créé le {{ product.created_at | date('d/m/Y H:i') }}</p>

    <p>{{ product.description }}</p>

    <a class="btn btn-sm btn-secondary" href="{{ path_for('product_list') }}">
      Retour à la liste des produits
    </a>
  </main>

{% endblock %}
```

Merci de votre attention !

Et la suite ?

- Back-office (CRUD)
- Gestion des utilisateurs