BTS SIO 2022

Support et mise à disposition de services informatiques (E4)

PAGE DE PRÉSENTATION DU DOSSIER

N° de candidat : |0|2|1|4|6|7|1|7|4|7|3|

NOM : DUFOUR

Prénom : Thomas

Date de passage[1] : 24 / 05 /2022	**Heure de passage[1] : 9h30**

CATEGORIE CANDIDAT [2] (UNE CASE A COCHER)	
❑ Scolaire	❑ Ex-scolaire
☒ Apprenti	❑ Ex-apprenti
❑ Formation professionnelle continue	❑ Ex-formation professionnelle continue
❑ Expérience professionnelle 3 ans	

[1] *Informations communiquées sur votre convocation envoyée en mars-avril 2022*

[2] *Informations communiquées sur votre confirmation d'inscription*

Tampon de
l'établissement

BTS SIO – Dossier Etudiant

Justificatif d'acquisition des compétences

Rédacteur(s)	Version	Date	Nb pages
Heilig Fabian	1.1	15/03/2022	10

Epreuve E4

Support et mise à disposition de services informatiques

SOMMAIRE

1 INTRODUCTION 4
2 MISSION 5 CREATION D'UN BLOG EN SYMFONY 5
2.1 Cahier des charges 5
2.2 Etude et conception de la solution 6
2.3 Gestion de projet 6
2.4 Mise en œuvre 8
2.5 Bilan 14

1 Introduction

L'objectif de ce document est de vous présenter les missions professionnelles que j'ai effectué dans le cadre de ma formation BTS SIO à l'école IRIS de Strasbourg.

Ces missions peuvent être de trois types :

- Effectuées en entreprise durant une alternance
- Effectuées en stage en entreprise
- Effectuées à l'école (compte-rendu de TP, projet collaboratif)

Le type de la mission sera précisé dans chaque cahier des charges.

Ce document se compose des parties suivantes :

Chapitres	Contenu
Chapitres 2 à 8	Présentation des missions, avec pour chacune : - Le cahier des charges - La solution proposée - La gestion de projet - La mise en œuvre - Le bilan du projet

2 Mission 5 : Création d'un blog en symfony

2.1 Cahier des charges

Type de mission
Mission effectuée dans le cadre de mon développement professionnel
Contexte
Les technologies utilisées par la majorité des entreprises étant en constantes évolutions, il a fallu pour moi commencer à en apprendre de nouvelles. Au préalable à l'aise avec le langage PHP et avec une bonne connaissance de la programmation orienté objet, j'ai décidé de me lancer dans l'apprentissage du framework PHP Symfony.
Demande du client
La demande de cette mission est assez simple, il faut pouvoir ajouter des articles selon des catégories et pouvoir par la suite les éditer.
Expression du besoin
- Apprendre l'utilisation des requêtes en symfony - Apprendre à utiliser le moteur de rendu Twig - Découvrir le modèle MVC (Modèles Vues Contrôleurs)
Budget disponible
Aucun budget nécessaire pour cette tâche
Outils disponibles
- L'IDE PHP Storm - Composer installé - Symfony installé
Contraintes
La contrainte pour ce projet a été de comprendre le fonctionnement du Framework. Comprendre comment relier des contrôleurs à des vues. Comment fonctionnes les requêtes SQL au sein du Framework et surtout comment se comporte l'architecture de celui-ci.

2.2 Etude et conception de la solution

2.2.1 Les outils de Symfony

Symfony embarque avec lui tout un tas d'outils permettant son bon fonctionnement, du moteur de rendu jusqu'à l'ORM, Symfony utilise un tas d'outils permettant la création de site web avec plus de facilité.

2.2.1.1 Le moteur de template TWIG

Nous pouvons tout d'abord nous poser la question suivante : « Qu'est-ce qu'un moteur de template ?»

Un moteur de template est un outil de modèle structurel qui simplifie la syntaxe pour assurer une bonne maintenabilité de son projet web. Il permet de dissocier la partie présentation (HTML) de la partie programmation (PHP, JS…).

Twig est le moteur de template intégré dans le Framework Symfony, mais il peut être utilisé indépendamment de Symfony. On peut aussi l'utiliser au sein d'autres projets PHP.

Pour exécuter ce moteur de template, il faut un serveur web Apache avec PHP qui ne doit pas être antérieur à la version 5.2.7.

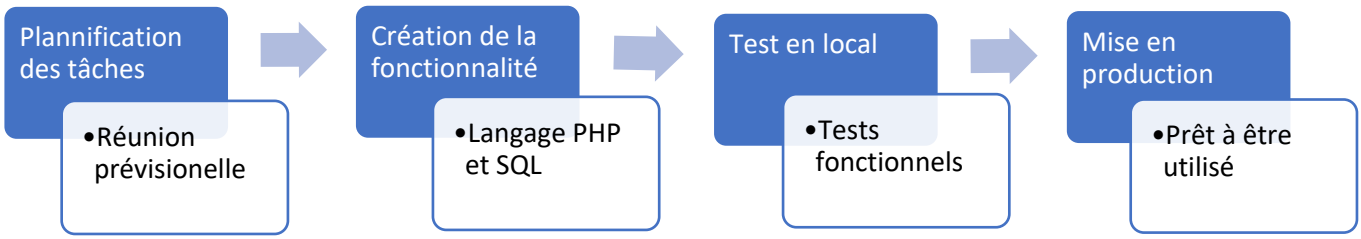
2.2.1.1 L'ORM doctrine

Un ORM (Object Relational Mapper) est un programme qui se place entre une application et une base de données relationnelle pour simuler une base de données orienté objet. Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif. Il pourrait être désigné comme une couche d'abstraction entre les schémas de la base de données et les classes du programme applicatif

Doctrine est l'ORM de Symfony embarqué avec l'installation d'un projet.

2.3 Gestion de projet

2.3.1 Planing de déploiement de la solution



2.3.2 Budget

Pas de budget alloué pour cette opération.

2.4 Mise en œuvre

2.4.1 Explication du Modèle Vue Contrôleur avec Symfony

I- Les routes et contrôleurs

Pour créer un contrôleur, on utilises une commande CLI qui est :

```
thomasdufour@macbook-pro-de-geoffroy blog % php bin/console make:controller
```

Ensuite, cette commande nous demande quel nom doit avoir le contrôleur puis le créer et y affecte un template :

```
Choose a name for your controller class (e.g. VictoriousElephantController):
> TestController

created: src/Controller/TestController.php
created: templates/test/index.html.twig


 Success!
```

Pour dire à un contrôleur vers quelle vue il doit rediriger sa logique, nous utilisons un système de routes auquel nous pouvons définir des options. En voici un exemple :

```php
    /**
     * @Route("/blog", name="app_blog")
     */
    public function index(ArticleRepository $articleRepository): Response
    {
        $articles = $articleRepository->findAll();
        return $this->render( view: 'blog/index.html.twig', [
            'controller_name' => 'BlogController',
            'articles' => $articles
        ]);
    }
```

Ici, la route est définie sous forme d'annotation dans un commentaire. Ce que nous dis ce commentaire, c'est que pour activer la logique contenu dans la méthode index(), il nous faut nous rendre sur « /blog ». Pour rendre une vue dans un contrôleur, on utilise la méthode render('cheminDeLaVue', [tableauDoptions]).

Dans ce controlleur par exemple, lorsque nous nous rendons sur l'url « /blog », il nous renvoi la vue dans le répertoire blog : index.html.twig

Nous pouvons aussi définir des routes dynamiques, comme sur cette capture par exemple :

```php
/**
 * @Route("/blog/{id}", name="blog_show")
 */
public function show(int $id, ArticleRepository $articleRepository)
{
    $article = $articleRepository->find($id);

    return $this->render( view: 'blog/show.html.twig',[
        'article' => $article
    ]);
}
```

Nous pouvons voir qu'il à été défini un URI dynamique avec « /{id} » qui nous permettra par la suite de chercher un article par son Identifiant.

Il a été défini un name dans la route qui nous permettra par la suite d'appeler le contrôleur facilement au sein du moteur de Template.

II- Les entités (Model)

Pour créer une entité, on utilises encore une fois une commande CLI qui est :

```
php bin/console make:entity
```

Tout comme la création de contrôleur, cette commande va nous poser tout un tas de questions. Tout d'abord le nom de l'entité :

```
Class name of the entity to create or update (e.g. VictoriousPopsicle):
> TestEntity

created: src/Entity/TestEntity.php
created: src/Repository/TestEntityRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.
```

Après avoir renseigné le nom de l'entité, la commande nous crée directement l'entité ainsi que son Repository associé qui nous servira à faire des requêtes par la suite.

```
New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> n

updated: src/Entity/TestEntity.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>
```

Pour la suite, on nous demandes certaines informations sur l'entité :

- Le titre de notre propriété, ici « Title
- Le type de la propriété (string, int, datetime…)
- La longueur du champ
- Si le champ peut être nul ou pas

Ensuite il suffit de dire si nous voulons une autre propriété ou non.

Lorsque l'entité est créée, les accesseurs et mutateurs sont automatiquement crées. Voici l'exemple d'une entité :

```php
namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Doctrine\ORM\Mapping as ORM;    DUFOUR, 13/04/2022 15:

/**
 * @ORM\Entity(repositoryClass=ArticleRepository::class)
 */
class Article
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $title;
```

```php
    public function getTitle(): ?string
    {
        return $this->title;
    }

    public function setTitle(string $title): self
    {
        $this->title = $title;

        return $this;
    }
```

Et enfin nous pouvons voir le Repository auprès duquel nous pourrons créer nos propres requêtes SQL :

```php
namespace App\Repository;

use ...

/**
 * @method Article|null find($id, $lockMode = null, $lockVersion = null)
 * @method Article|null findOneBy(array $criteria, array $orderBy = null)
 * @method Article[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class ArticleRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, entityClass: Article::class);    DUFOUR, 13/04/2022 15:31 • First c
    }

    public function findAll()
    {
        return $this->findBy(array(), array('id' => 'DESC'));
    }

    /**
     * @throws ORMException
     * @throws OptimisticLockException
     */
    public function add(Article $entity, bool $flush = true): void
    {
        $this->_em->persist($entity);
        if ($flush) {
            $this->_em->flush();
        }
    }
```

Suite à la création de ces deux fichiers, il nous faut taper la commande suivante :

```
php bin/console make:migration
```

Cette commande créer un fichier nous permettant d'instantanément récupérer la structure de la base de donnée avec la commande suivante :

```
php bin/console doctrine:migration:migrate
```

```
WARNING! You are about to execute a migration in database "blog_new" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
> yes

[OK] Already at the latest version ("DoctrineMigrations\Version20220427132520")
```

III- Les vues

Les vues en Symfony sont gérées par Twig. Pour envoyer des informations à la vue, il nous faut les définir dans le tableau d'option du contrôleur comme ici :

```php
/**
 * @Route("/blog", name="app_blog")
 */
public function index(ArticleRepository $articleRepository): Response
{
    $articles = $articleRepository->findAll();
    return $this->render( view: 'blog/index.html.twig', [
        'controller_name' => 'BlogController',
        'articles' => $articles
    ]);
}
```

Ici, on cherches tous les articles de la base de données avec la méthode findAll() que l'on passe dans une variable $articles.

Cette variable $article sera passé dans le tableau d'options du rendu pour la vue sous le nom articles. Ensuite avec twig, on passe une boucle for afin de parcourir le tableau d'articles et les afficher :

```twig
{% extends 'base.html.twig' %}

{% block title %}Hello BlogController!{% endblock %}

{% block body %}
<section class="article">
    {% for article in articles %}
    <article>
        <h2>{{ article.title}}</h2>
        <div class="metadata">Ecrit le {{ article.createdAt | date('d/m/y')}} à {{  article.createdAt | date('H:i') }}
        <div class="content"><img src="{{ article.image }}" alt="">
            <p>{{ article.content | raw}}</p>
            <a href="{{ path('blog_show', {'id': article.id}) }}" class="btn btn-primary">Lire la suite</a>
            <a href="{{ path('blog_update', {'id': article.id}) }}" class="btn btn-primary">Editer</a>
        </div>
    </article>
    {%  endfor %}
</section>
{% endblock %}
```

Pour afficher une valeur en particulier du tableau d'articles, on peut directement taper dans entre crochets ({{test}}) le nom de la ligne défini dans le for (ici article) suivi d'un « . » puis la propriété. Pour afficher le titre on tapera donc « article.title ».

Comme on peut le voir plus haut, twig fonctionne sur un système de block pour s'implémenter dans un template de base. Voici un exemple de template de base :

```
<div class="container">
    {% block body %}{% endblock %}
</div>
```

Dans la pratique, quand on défini le block {% extends 'base.html.twig' %} et qu'ensuite on défini de l'HTML dans un block body, tout ce qui a dans le block body sera affiché en plus de l'affichage du template de base.
Concerant l'affichage des variables, on peut aussi définir des filtres pour pouvoir formater des données. Par exemple « article.createdAt | date('d/m/y') » pour afficher la date au format jour/mois/année.

Concernant les chemin à définir dans les balises « a » par exemple on peut utiliser une méthode s'appelant path() afin d'y définir le name donné à un contrôleur au préalable.

2.5 Bilan

2.5.1 Validation des exigences point par point

- Apprentissage du modèle MVC de symfony : **FAIT**
- Utilisation des requêtes SQL avec les Repository : **FAIT**
- Utilisation du moteur de template TWIG : **FAIT**

2.5.2 Axes d'amélioration

Les améliorations pouvant être apportés à ce projet sont multiples, tout d'abord la possibilité de créer des rôles pour ne pouvoir ajoutes des articles qu'en étant administrateur du site. Un système de connexion nous permettant de poster des commentaires en dessous de chaque article est aussi envisageable afin de gérer la partie authentification.

2.5.3 Compétences acquises

Lors de la réalisation de ce projet, j'ai pu apprendre à utiliser les outils de symfony afin de créer une application de base faisant appel à des requêtes en base de données. J'ai aussi pu comprendre comment fonctionne le modèle MVC à travers des exemples simples et concrets. En outre, j'ai pu développer mes connaissances et et augmenter mon portefeuille de compétence. C'est un pas de plus pour moi dans l'apprentissage du Framework.