SOUSSAN Chloé BTS SIO SLAM

Documentation technique : GSB-Visites

AP S3

Année 2021-2022

Lycée Turgot Paris

Sommaire

Introduction

1) Structure des données

2) Structure de l'application

3) Diagramme de classe

4) Présentation des classes

5) Moteur de Template TWIG

6) Authentification

7) Bundle

8) Les Routes

Conclusion

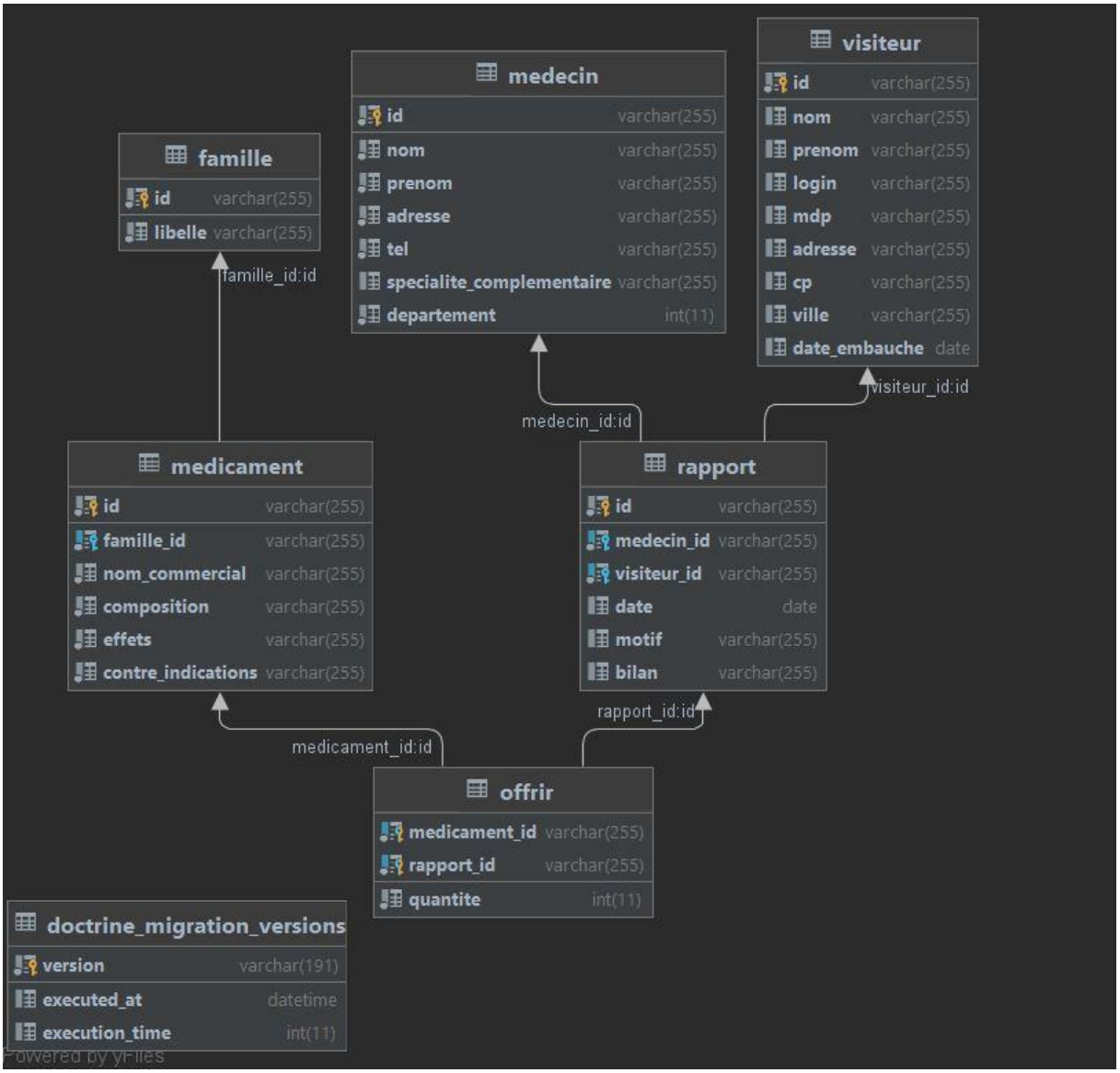
Introduction

Le projet d'application Web GSB gestions visites a été conçu avec le Framework PHP Symfony qui utilise la structure MVC.

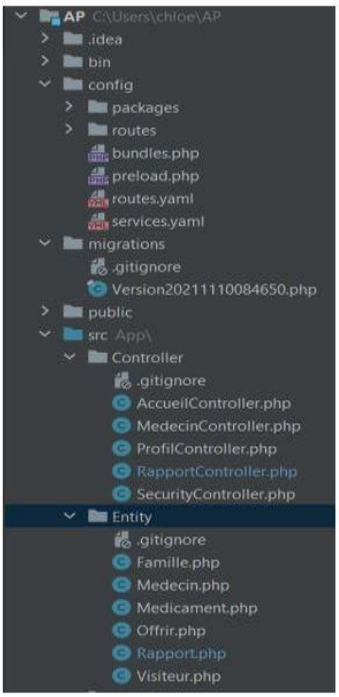
Le projet a été réaliser sur l'interface de développement IntelliJ.

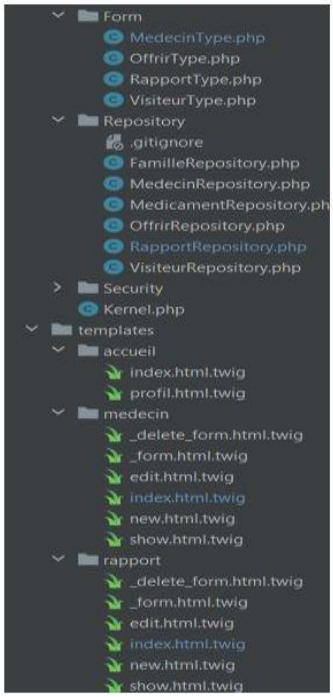
Le moteur de base de données est MariaDB.

1) Structure des données

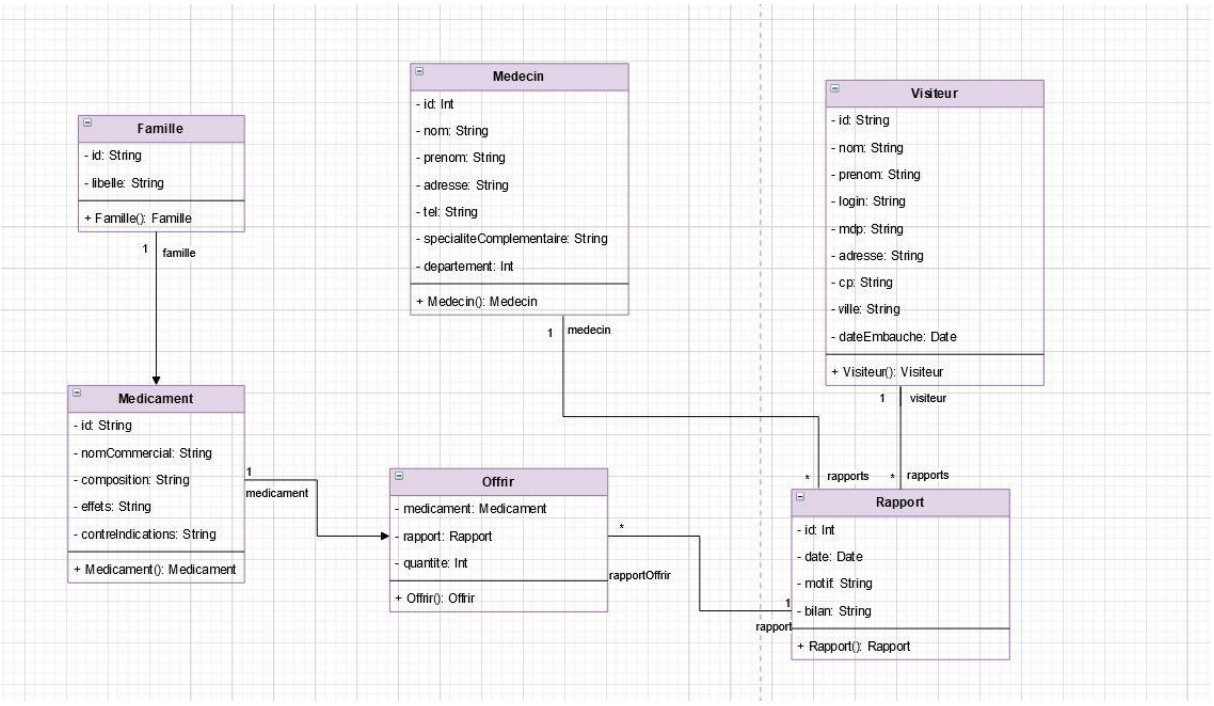


2) Structure de l'application



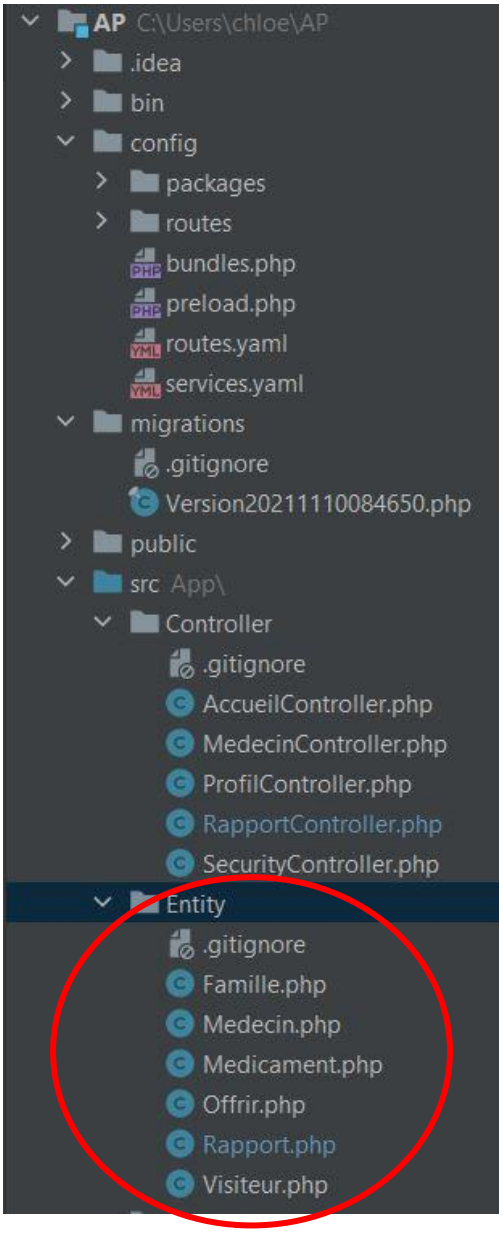


3) Diagramme de classe



4) Présentation des classes

Dans le dossier Entity : ce sont les objets métiers qui correspondent chacune à une table dans la base de données.



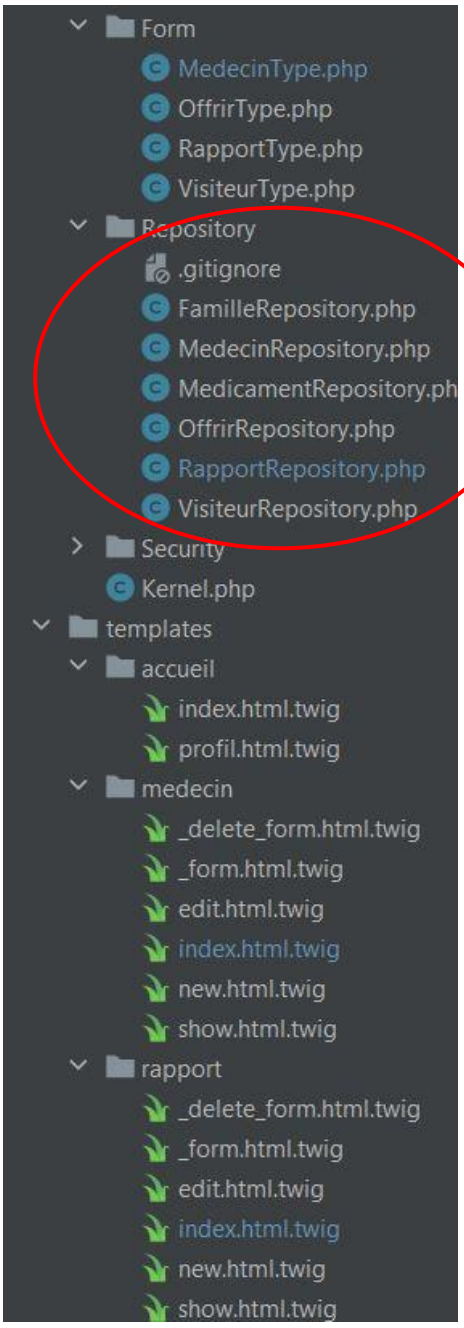
Exemple de l'entité Visiteur :

Des commentaires avec des « * » sont utilisées pour configurer les attributs/méthodes de nos entités et éviter les erreurs.

```php
/**
 * @ORM\Entity(repositoryClass=VisiteurRepository::class)

 */
class Visiteur implements UserInterface
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue()
     * @ORM\Column(type="string")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $prenom;

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

De plus, ces commentaires permettent de déterminer les relations entre chaque table et donc de générer nos clés étrangères dans la base de données selon la structure de données souhaitées.

Symfony utilise un ORM (mapping objet-relationnel) : **Doctrine** qui va se charger de gérer l'accès direct aux données ainsi que de formater la structure de la base dans le dossier repositories :



Ils permettent d'accéder aux données (DAO) en chargeant celles-ci depuis la base.

C'est ici qu'on réalise nos requête SQL. On va donc y construire des méthodes pour récupérer une entité par son id, pour récupérer une liste d'entités suivant un critère spécifique de l'entité correspondante.

Attention, il existe un repository par entité. Cela permet de bien organiser son code. Cela n'empêche pas qu'un repository utilise plusieurs types d'entité, dans le cas d'une jointure par exemple.

Les repositories utilisent en réalité directement l'EntityManager (une interface) pour fonctionner.

Méthode de récupération des entités :

Il existe deux moyens de récupérer les entités : en utilisant du DQL et en utilisant le QueryBuilder.

Le Doctrine Query Language (DQL)

Le DQL n'est rien d'autre que du SQL adapté à la vision par objets que Doctrine utilise.

Le QueryBuilder

Le QueryBuilder sert à construire une requête, par étape.

Autres méthodes de récupération :

Les repositories héritent de la classe Doctrine\ORM\EntityRepository, qui propose déjà quelques méthodes très utiles pour récupérer des entités qui sont effectués depuis un contrôleur : find(), findAll().

```php
/**
 * @method Rapport|null find($id, $lockMode = null, $lockVersion = null)
 * @method Rapport|null findOneBy(array $criteria, array $orderBy = null)
 * @method Rapport[]    findAll()
 * @method Rapport[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class RapportRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, entityClass: Rapport::class);
    }
    /**
     * @return Rapport[] Returns an array of Rapport objects
     */

    public function findRapportByDate(DateTime $date): array
    {
        $entityManager = $this->getEntityManager();

        $query = $entityManager->createQuery(
            dql: 'SELECT r
            FROM App\Entity\Rapport r
            WHERE r.date = :date
            '
        )->setParameter( key: 'date', $date);

        return $query->getResult();
    }
```

Dans le dossier Form :

Ce sont des classes qui vont générer nos formulaires.

Permet de créer et valider les formulaires (classes techniques).

```php
class RapportType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add( child: 'date')
            ->add( child: 'motif')
            ->add( child: 'bilan')
            ->add( child: 'medecin')
            //->add('visiteur')

            ->add( child: 'offrirs', type: CollectionType::class, [
                'entry_type' => OffrirType::class,
                'allow_add' => true,
                'allow_delete' => true,
                'prototype' => true,
                'by_reference' => false,
                'entry_options' => ['label' => false]
            ])


        //->add('valider', SubmitType::class);
        ;
    }
```

```php
public function configureOptions(OptionsResolver $resolver): void
{
    $resolver->setDefaults([
        'data_class' => Rapport::class,
    ]);
}
```

Ici la classe hérite de la classe AbstractType de Symfony qui définit 2 méthodes :

La première méthode buildForm() reçoit deux paramètres, dont le $builder qui va nous permettre de définir les champs de notre formulaire

La deuxième méthode va au fait nous permettre d'explicitement définir la classe à laquelle est rattachée ce formulaire.

Dans le dossier controller :

Ils permettent de contenir toute la logique de notre application Web.

C'est lui qui s'occupe de la liaison entre nos repository (le modèle) et la vue (Template).

Il retourne une réponse à partir de classe « Response » qui signifie d'instancier un objet Response et le retourne.

Symfony utilise 2 types d'objets qui vont nous permettre de construire une réponse en fonction de la requête : Request et Response.

Par exemple $request->query nous permet de récupérer les paramètres de l'URL passés en GET.

Par exemple, la réponse est contenue dans la vue, dans un Template via une méthode dans le contrôleur :

$this->render() : elle prend en paramètres le nom du Template et ses variables puis s'occupe de tout : créer la réponse, y passer le contenu de la Template et retourner la réponse.

```php
class RapportController extends AbstractController
{

    /**
     * @Route("/", name="rapport_index", methods={"GET"})
     */
    public function index(RapportRepository $rapportRepository): Response
    {
        $request = Request::createFromGlobals();
        $query = $request->query->get( key: 'date');
        $loggedUser = $this->getUser();


        if ($query != '' && $query != Null && strtotime($query)) {
            $rapports = $rapportRepository->findRapportByDate(date_create($query));
        } else {
            $rapports = $rapportRepository->findBy(['visiteur' => $loggedUser]);
        }
        return $this->render( view: 'rapport/index.html.twig', [
            'rapports' => $rapports
        ]);
    }
```

Les commentaires « * » permettent de définir les routes.

5) Moteur de Template TWIG

Les Template vont nous permettre de séparer le code PHP du code HTML/CSS /javascript.

Cependant, dans le cadre de cette application Web nous avons besoins de faire du code de présentation dynamique pour afficher tous les rapports d'un médecin par exemple.

C'est pour cela que Symfony utilise le moteur de Template TWIG qui facilite ce code dynamique.

Par exemple pour afficher une variable on utilise {{ }} au lieu d'un echo en PHP.

```
<tr>

    <th>Rapport </th>
    {% for rapport in medecin.rapports %}
        <td>
            {{ (rapport.medecin) }}
            <a class="btn btn-outline-dark" href="{{ path('rapport_show', {id: rapport.id})}}">Voir rapport</a>

        </td>
    {% endfor %}

</tr>
```

Le fonctionnement de la syntaxe {{ objet.attribut }} est un peu plus complexe. Elle ne fait pas seulement objet->getAttribut. En réalité, voici ce qu'elle fait exactement :

- Elle vérifie si objet est un tableau, et si attribut est un index valide. Si c'est le cas, elle affiche objet['attribut'].
- Sinon, et si objet est un objet, elle vérifie si attribut est un attribut valide (public). Si c'est le cas, elle affiche objet->attribut.
- Sinon, et si objet est un objet, elle vérifie si attribut() est une méthode valide (public). Si c'est le cas, elle affiche objet->attribut().
- Sinon, et si objet est un objet, elle vérifie si getAttribut() est une méthode valide. Si c'est le cas, elle affiche objet->getAttribut().
- Sinon, et si objet est un objet, elle vérifie si isAttribut() est une méthode valide. Si c'est le cas, elle affiche objet->isAttribut().
- Sinon, elle n'affiche rien et retourne null.

Twig enregistre des variables globales également et utilise des balises « bloc » pour définir chaque partie de notre code HTML (exemple : body)

6) Authentification

Pour réaliser une authentification nous utilisons SecurityBundle de Symfony. Il permet la connexion et la navigation de manière sécurisée.

Il suffit de l'installer avec composer :

```
composer require symfony/security-bundle
```

Nous générons l'authentification avec la commande :

```
php bin/console make:auth
```

Cette commande va modifier un fichier security.yaml qui va gérer les rôles utilisateurs ainsi que le hachage des mots de passes.

```yaml
    App\Entity\Visiteur:
        algorithm: plaintext

# https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\Visiteur
            property: login
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: lazy
        provider: app_user_provider
        guard:
            authenticators:
                - App\Security\VisiteurAuthenticator

        logout:
            path: app_logout

            # where to redirect after logout
            # target: app_any_route
```

Ajouter un controller, un Template et une classe héritière de AbstractFormLoginAuthentificator. Une classe de base pour faciliter l'authentification des formulaires de connexion.

7) Bundle

Utilisation de la console

Symfony intègre des commandes disponibles non pas via le navigateur, mais via l'invite de commandes (sous Windows) ou le terminal. Il existe pas mal de commandes qui vont nous servir assez souvent lors du développement comme vu précédemment.

Depuis cette console, on pourra par exemple créer une base de données, vider le cache, ajouter ou modifier des utilisateurs (sans passer par phpMyAdmin) Mais ici le plus intéressant c'est la **génération de code**.

En effet, pour créer un bundle, un modèle ou un formulaire, le code de départ est toujours le même. C'est ce code-là que le générateur va écrire pour nous.

Chaque partie du site est un bundle. Pour créer une page, il faut donc d'abord créer un premier bundle avec cette commande :

```
php app/console generate:bundle.
```

8) Les Routes

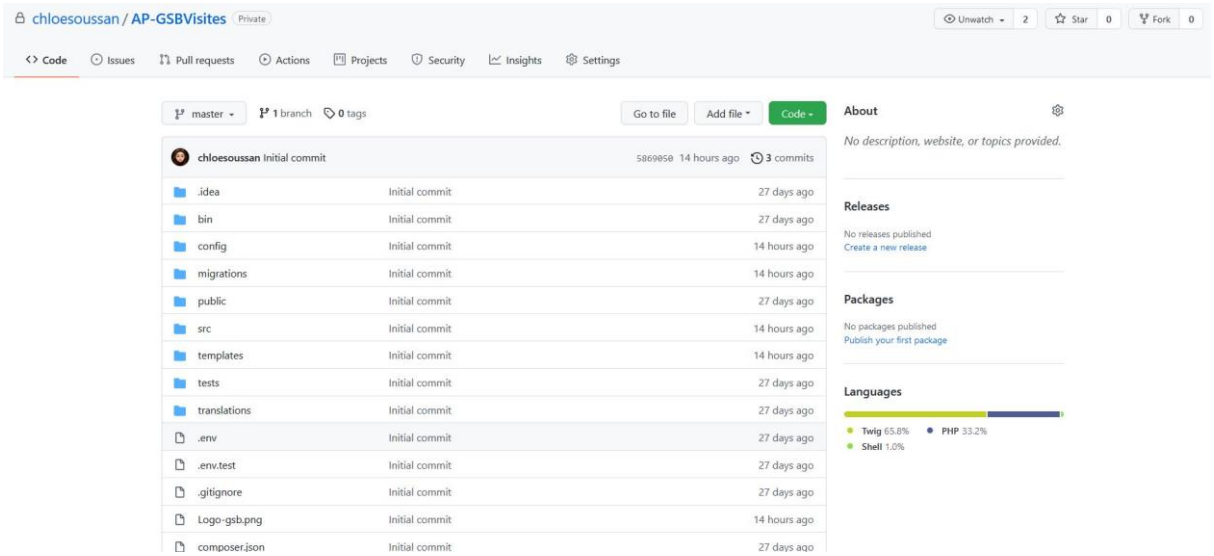
Le rôle du routeur est, à partir d'une URL, de déterminer quel contrôleur appeler et avec quels arguments. Cela permet de configurer l'application pour avoir le bon URL.

Pour cela on va pouvoir créer un fichier de *mapping* des routes (un fichier de correspondances). Ce fichier, situé dans Bundle/Resources/config/routing.yml, contient la définition des routes. Chaque route fait la correspondance entre une URL et le contrôleur à appeler.

Conclusion

Enfin, pour télécharger et compiler le projet suivre mon dépôt sur GitHub.

Lien téléchargeable : https://github.com/chloesoussan/AP-GSBVisites


chloesoussan / AP-GSBVisites
Private
Unwatch 2
Star 0
Fork 0
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
master
1 branch
0 tags
Go to file
Add file
Code
chloesoussan Initial commit
5869050 14 hours ago
3 commits
.idea Initial commit 27 days ago
bin Initial commit 27 days ago
config Initial commit 14 hours ago
migrations Initial commit 14 hours ago
public Initial commit 27 days ago
src Initial commit 14 hours ago
templates Initial commit 14 hours ago
tests Initial commit 27 days ago
translations Initial commit 27 days ago
.env Initial commit 27 days ago
.env.test Initial commit 27 days ago
.gitignore Initial commit 27 days ago
Logo-gsb.png Initial commit 14 hours ago
composer.json Initial commit 27 days ago
About
No description, website, or topics provided.
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Languages
Twig 65.8%
PHP 33.2%
Shell 1.0%