Documentation technique du site de la médiathèque

Prénom : Gaël
Nom : Layer

Spécifications techniques

Configuration du serveur local MAMP sous Mac

- Apache
- PHP version 7.4.12

Modification du chemin dans le fichier **httpd-vhosts.conf** (utilisation en parallèle à des fins privés sur d'autres projets) :

```
<VirtualHost *:80>
    ServerName symfony.localhost
    DocumentRoot /Applications/MAMP/apps/mediatheque/public
 <Directory /Applications/MAMP/apps/mediatheque/public>
    AllowOverride All
    Allow from All
    FallbackResource /index.php
 </Directory>
</VirtualHost>
```

Front-end

- HTML 5
- CSS 3
- Bootstrap
- Javascript
- JQuery

Configuration Bootstrap

J'ai choisi d'utiliser le Framework Bootstrap car il propose un ensemble d fonctionnalités et de composants au design recherchés qui répondent aux besoins actuels. Il à également une rapide prise en main notamment concernant les breakpoints intégrés dans le Css.

Téléchargement du fichier source Bootstrap sur le site puis vérification avec le terminal de l'existence du préprocesseur **Sass**, du gestionnaire de dépendance **nodeJS** et **yarn** sur mon système.

Installation du bundle **webpack/encore,** configuration des fichiers à l'intérieur du nouveau répertoire assets pour interpréter du **scss**, modification également du fichier **webpack.config** puis ajout du module auprefixer dans le fichier postes.config.js pour exécuter le scss

Utilisation d yearn build pour venir compiler les fichiers compilés dont on aura besoin.
Installation de bootstrap avec la commande yarn add bootstrap puis ajout de dépendances dont bootstrap à besoin pour fonction yarn add jquery @popperjs/core --dev

---

Configuration CSS

Préférences > Frameword > Symfony > Chamgement répertoire web en public > téléchargement de la dépendance asset pour pouvoir lire mes fichiers asset composer req symfony/asset

Back-end

- PHP 7.4 sous PDO
- Symfony
- MySQL

Configuration Symfony

Il me semblait évident de devoir utiliser et découvrir le Framework Symfony au vu de sa popularité et de son ergonomie. J'ai également cru comprendre qu'il permettait grâce à son abstraction de proposer un niveau de sécurité assez élevé notamment durant les requêtes SQL.

Configuration de Symfony avec l'ajout des dépendances nécessaires à la réalisation du projet à l'aide de composer

Création d'un nouveau projet avec la commande :

- **composer create-project symfony/skeleton mediatheque**

Ajout des dépendances :

- **composer req orm (doctrine)**
- **composer req maker** : pour créer des entités, contrôleur … plus rapidement
- **composer req validator** : Pour valider les données d'un formulaire
- **composer req security**
- **composer req debug-pack / profiler-pack :** pour debugger l'application
- **composer req annotations :** qui vont permettre de mettre en place pour décrire nos entités doctrine, les routes..
- **composer req migrations :** Pour faire la migration de nos entités vers notre base de données
- **composer req twig :** pour le rendu HTML
- **composer req form :** pour la création des formulaires
- **composer require easycorp/easyadmin-bundle :** pour installer Easyadmin
- **composer require vich/uploader-bundle** : pour l'upload d'images
- **composer require webpack-encore** : pour l'utilisation de Sass avec Bootstrap

Configuration de la base de données dans le fichier .env :

```
DATABASE_URL="mysql://root:root@127.0.0.1:3306/mediatheque?serverVersion=5.7"
```

Système de gestion de versions

- Utilisation du Git pour le stockage local du projet

Création des branches à chaque nouvel technique / users stories à intégrer dans l'application

- GitHub pour le stockage en ligne du projet

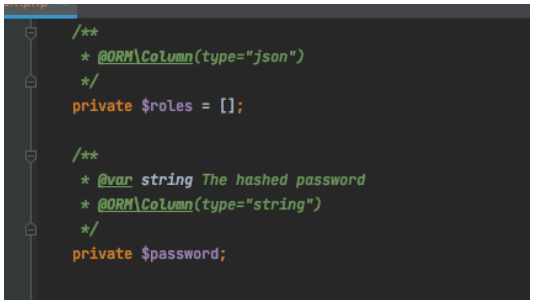
Sécurité

Pour la mise en place des différentes procédures de sécurité du site, j'ai dans un premier temps respecté le principe des contraintes de validation avec la dépendance **validator** pour protéger les valeurs des propriétés.

1.Gestion des utilisateurs

Authentification

Il est indiqué dans l'énoncé les différentes propriétés de la classe des inscrits (borrower) dont l'email qui normalement doit être pour l'authentification mais j'ai trouvé plus pertinent d'utiliser un nom d'utilisateur
Création des utilisateurs à l'aide la commande **make:user** qui permet de générer plusieurs contraintes automatiques comme l'implémentation **UserInterface** qui contient le champ unique de notre table ainsi que le hachage du mot de passe pour se connecter.



Connexion

Pour la connexion j'ai utilisé la commande **make:auth** qui permet de générer plusieurs classes dont le **Guard Authenticator** (récupération des données, correspondance, redirection…), une classe contrôleur avec l'aspect déconnexion et les routes qui y sont liées et le template Twig du formulaire de connexion.

Le fichier **security.yaml** a également été mis à jour avec la création de mes entités et des différentes propriétés utilisateur. J'ai également dû apporter quelques modifications à ce fichier comme l'ajout d'un utilisateur supplémentaire.

```yaml
providers:
    chain_provider:
        chain:
            providers: [ app_user_provider, app_staff_provider ]
    # used to reload user from session & other features (e.g. switc
    app_user_provider:
        entity:
            class: App\Entity\Borrower
            property: username
    app_staff_provider:
        entity:
            class: App\Entity\Staff
            property: username
```

Autorisation

Une fois mes utilisateurs créés, j'ai attribué dans le fichier **security.yaml** les différents rôles de mes utilisateurs sous forme de hiérarchie pour jouer sur les vues de mon interface ainsi que sur les accès autorisés dans la partie **access_control**

Staff.php | security.yaml

```yaml
security:
    role_hierarchy:
        ROLE_BORROWER: ROLE_USER
        ROLE_STAFF: ROLE_BORROWER
        ROLE_ADMIN: ROLE_STAFF
```

```yaml
    # Easy way to control access for large sections of your si
    # Note: Only the *first* access control that matches will
    access_control:
     - { path: ^/admin, roles: ROLE_BORROWER }
     - { path: ^/borrower, roles: ROLE_USER }
```

Validation du compte par l'administrateur

Je me suis inspiré d'une des fonctionnalités proposées par Symfony en créant une classe userChecker.

- J'ai dans un premier temps modifier ma classe Borrower pour y ajouter une propriété isChecked de type booléenne, non nulle avec comme valeur par défaut false
- J'ai ensuite modifier la classe userChecker pour y inclure mes conditions postAuth

```php
public function checkPreAuth(UserInterface $user): void
{

    if (!$user instanceof AppBorrower) {
        return;
    }

    //$dump = dump($user->getIsChecked());
    //$dump();
    if ($user->getIsChecked() === false) {
        throw new CustomUserMessageAccountStatusException("
        Votre compte n\'est pas encore activé" );
    }
```

- Côté administration j'ai ajouté le champs BooleanField pour permettre à l'administrateur de vérifier l'intégrité des données avant de valider le compte

---

2.Gestion du back office

J'ai découvert le bundle **EasyAdmin** qui permet d'avoir un contrôle total sur l'administration de son site, on peut y inclure toute forme d'action liée au CRUD (create, read, update, delete), je m'en suis servie pour la gestion du catalogue et la création d'employé.

Ce bundle permet d'ajouter une couche de sécurité en plus dans mon application car il est directement relié au système de sécurité proposé par Symfony.

Installation et configuration d'EasyAdmin

Pour installer le bundle j'ai suivie la documentation et pour la partie configuration je me suis aidé de tutoriels proposés sur le web.

- Tout d'abord la commande **make:admin:dashboard** génère le contrôleur et la vue du Dashboard.
- Extension du fichier dashboard.html.twig pour permettre une gestion de la vue du panneau de contrôle
- J'ai ensuite créé mes crudController à l'aide la commande **make:crud** sur les classes qui devront être manipulable

Je me suis ensuite concentré sur les différents champs qui seront affichés dans mon Dashboard à l'aide des crudControllers et notamment sur la partie upload d'image pour les livres

- Pour cela j'ai d'abord dû installer le bundle vich/uplaoder qui permet l'upload de fichier en base de données
- J'ai ensuite configuré les paramètres du fichier services.yaml pour indiquer le dossier dans lequel je souhaite que les images soient placées ainsi que le nouveau fichier vich_uploader.yaml pour la gestion du mapping

```
vich_uploader:
    db_driver: orm

    mappings:
        book_images:
            uri_prefix: '%book_images%'
            upload_destination: '%kernel.project_dir%/public%book_images%'
            namer : Vich\UploaderBundle\Naming\UniqidNamer
            #Pour éviter de supprimmer lors d'un update ou d'un déplacement
            #delete_on_update: false
            #delete_on_remove: false
```

- A l'aide de la documentation j'ai modifié mon entité Book pour ajouter les bonne propriétés à ma classe et j'ai également ajouté un eventSubscriber pour initialiser la date du jour lors de l'upload et ainsi ajouter l'image dans le fichier de destination

```
    public function setUpdateAt(BeforeEntityPersistedEvent $event)
    {
        $entity = $event->getEntityInstance();

        if (!($entity instanceof Book)) {
            return;
        }

        $now = new DateTime('now');
        $entity->setUpdatedAt($now);

    }
```

- Pour terminer j'ai configuré les bons champs dans mon fichier crudController

```
return [
    TextField::new('title'),
    TextEditorField::new('description')->hideOnIndex(),
    TextField::new('author'),
    TextField::new('genre'),
    DateTimeField::new('publication_date'),
    /*BooleanField::new('availability'),*/
    TextField::new('picture')->setFormType(VichImageType::class)->onlyWhenCreating(
    ImageField::new('filename')->setBasePath('/images/books')->onlyOnIndex(),

];
```

Note : En raison d'un manque de temps je n'ai pas pu faire l'action supprimer, j'ai donc ajouter la configuration dans les crud de mes entités pour cacher l'option delete. J'utilise à la place la commande SQL :

php bin/console doctrine:query:sql DELETE FROM table WHERE ? = '?'

Déploiement

- Tout d'abord initialisation d'un projet Heroku avec la commande heroku create
- Création du fichier procfile avec la commande echo 'web: heroku-php-apache2 public/' > Procfile pour indiquer à Heroku où pointer pour ouvrir le répertoire qui servira à ouvrir les fichiers
- Installation d'une base de donnée distante avec l'addons JawsDB mysql : heroku addons:create jawsdb:kitefin
- Paramètrage du fichier .env pour indiquer à Heroku que l'on souhaite qu'il manipule le projet toujours en mode prod
- Par défaut Heroku va jouer le composer install, il faut donc indiquer à heroku de faire les migrations de la base de données avec la commande définie dans le script.
- Modification du fichier .htacces qui va venir indiquer à apache comment réécrire les url avec le bundle composer require symfony/apache-pack

- Push vers le remote heroku avec la commande git push heroku main pour versionner mon projet vers le dépôt heroku

- Pour terminer j'ai effectuer un import puis export d embase de données vers la nouvelle base de JawsDB

mysqldump -h localhost -u root -proot mediatheque > backup.sql
mysql -h newest -u newsier -p new password new db name < backup.sql