Guide Symfiony pour creer un Projet de base

Etapes:

- Installation de Symfony et Maker Bundle

```
# Telechargement de l'installateur
wget https://get.symfony.com/cli/installer -O - | bash

# Bouger l'executable dans le dossier global d'executables
mv /home/utilisateur/.symfony/bin/symfony /usr/local/bin/symfony

# Creation d'un projet Symfony (se placer dans le repertoire ou on veut le
projet)
symfony new MonProjet

# Installation du composant 'Maker Bundle' afin d'avoir acces a des
bibliotheques symfony
composer require symfony/maker-bundle --dev

# explication: composer est le gestionnaire et installateur de bibliotheques,
require est l'action a executer, symfony/maker-bundle est la bibliothque a
installer, l'option --dev precise qu'il faut sauvegarder la bibliotheque en
tant que dependance de developpement (elle ne sera pas necessaire a installer
en environemment de Production)
```

- Creation et configuration de la BDD

1. Installation de Doctrine ORM

```
composer require symfony/orm-pack

# l'installation de l'ORM va nous generer quelques fichiers et va notamment
ecrire une ligne de config dans notre fichier .env, elle ressemble a:
DATABASE_URL="postgresql://postgres:@127.0.0.1:5432/db_name?
serverVersion=13.3&charset=utf8"
```

2. Creation de la BDD

```
symfony console doctrine:database:create

# cette commande va indiquer a Doctrine ORM de creer une base de donnees avec
les informations renseignees dans la ligne de de config ci dessus, en gros:
```

```
"{ENGIN_BDD}://{UTILISATEUR}:{MOT_DE_PASSE}@{ADDRESSE}:{PORT}/{NOM_BDD}?
serverVersion=13.3&charset=utf8"

# dans notre cas de figure, on creera la BDD avec PostgreSql, utilisateur
postgres, mot de passe vide, dans localhost, port par defaut 5432 d'ou:
"postgresql://postgres:@127.0.0.1:5432/db_name?serverVersion=13.3&charset=utf8"
```

Si la commande precedente est executee avec succes, on peut passer a la prochaine etape.

3. Creation des entites

Grace a l'interface de ligne de commande de php maker-bundle

```
symfony console make:entity

# se laisser guider par le programme, saisir d'abord le nom souhaite pour la
table, les noms des colonnes, ses types (STR, VARCHAR, INT...) a savoir qu'une
colonne ID AUTO_INCREMENT est cree toute seule, pas besoin de la saisir
manuellement
```

- Installer le bundle security afin d'avoir une entite 'utilisateur' a partir de laquelle on construit la notre

```
composer require security

symfony console make:user

# Note: on peut bien sur creer notre propre modele d'utilisateur a partir de 0,
cependant heriter depuis une entite prefabriquee a ses avantages, notamment:
# 1.) fonctionnalite de hachage du mot de passe inclue par defaut, c.a.d que le
MDP stocke en BDD sera une chaine de caracteres "hachee" donc illisible sans la
bonne cle de chiffrage (seul a travers l'ORM aura t-on acces a la lecture de
celui ci)
# 2.) Identifiants uniques par defaut (au choix: email, nom d'utilisateur).
cela nous permet d'avoir des contraintes d'unicite sur une colonne sans avoir a
le faire nous meme
# 3.) systeme de roles predefini et inclu par defaut (si jamais on souhaite
affecter des roles avec des privileges superieurs a un utilisateur (ex. admin)
on aura la possibilite sans ajouter de la logique supplementaire)
```

Faire ensuite les autres entites

4. Mettre a jour les entites

Ajouter les relations

(utilisateur->post par exemple)

```
symfony console make:entity

# on utilise la meme commande pour mettre a jour une entite, lorsque
l'interface nous demandera quelle entite on souhaite creer, si on saisit une
entite existante, il la reconnaitra automatiquement et il vous proposera de
rajouter des champs. pour ajouter un champ de type relation il suffit de le
preciser ainsi:

symfony console make:entity

 Class name of the entity to create or update (e.g. OrangeGnome):
 > User

Your entity already exists! So lets add some new fields!

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

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

 Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
 > OneToMany

 # l'outil nous proposera aussi par defaut la possibilite d'ajouter la relation
inverse. ici par exemple on obtiendrait post->user. Cela est pratique mais pas
toujours souhaitable
```

Une fois ces etapes realisees avec succes, deux fichiers sont generees et mis a jour dans `src/Entity/`. Ces fichiers correspondent aux declaration de Classes PHP regroupant les proprietes des entites qu'on a declare lors de la saisie en ligne de commande.

Note sur le fonctionnement et la structure d'une Entite Doctrine

Si on va dans `src/Entity/User.php` par exemple, on retrouve les proprietes qui seront converties par l'ORM en colonnes BDD. (lignes precedees d'une annotation `@ORM\Column`)

```
    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $email;
```

explication:

- `private`: mot cle propre a la programmation oriente objet. il precede une propriete qu'on souhaite rendre inaccessible en dehors du contexte de l'objet, on pourra la requeter seulement a travers son `getter`

  en effet, plus en bas du fichier on trouve:

  ```
  public function getEmail(): ?string
  {
      return $this->email;
  }
  ```

- `$email` le nom de la propriete
- `@` une annotation (ou decorateur) en programmation orientee objet, sert a associer des informations ou de la fonctionnalite **complementaire** a une **propriete** ou une **methode**. A ne pas confondre avec un **commentaire de documentation**, la difference etant que ce dernier n'aura aucune incidence sur le comportement de la propriete qu'il decore!
  - dans notre cas de figure

  ```
    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */

    // On est en train d'indiquer que la propriete qui suit, sera un Objet Colonne ORM, que son contenu sera de type chaine de caracteres, sa longueur maximale de 180 et qu'il sera unique! c.a.d pas deux objets User avec le meme email
  ```

Fin de la parenthese sur les entites

---

6. Creer et executer les migrations

Une fois que nos entites ont ete crees et qu'on est satisfaits avec, on peut proceder a la generation des migrations. Une migration est une serie d'instructions a transmettre a l'engin de la BDD afin qu'il convertisse nos Classes PHP en commandes SQL pour definir le 'schema' soit les tables, les colonnes, les relations etc.

Pour cela on genere d'abord les fichiers migration:

```
symfony console make:migration

# ceci generera des fichiers dans ./migrations
```

Ensuite, si on a pas eu d'erreur, on procede a les executer:

```
symfony console doctrine:migration:migrate
```

Si ces deux commandes ont abouti sans erreurs, notre BDD a ete cree et schematise selon la logique de nos entites. La BDD est donc prete a etre requetee en toute securite.

7. Convertir nos entites en ressources API

Introduction a API Platform

API Platform est un bundle Symfony qui nous permet de serialiser ou "traduire" nos entites PHP (qui en l'ocurrence sont attachees a des Tables SQL) en **Resources d'API REST-Conforme** prets a etre solicites via HTTP.

En ajoutant quelques lignes de code a nos entites, API Platform va generer toute la logique necessaire pour cela. (controleurs, router, endpoints et les methodes acceptees, serialiseurs JSON, etc) de maniere automatique.

pour cela, il suffit de se rendre dans nos classes d'entites (`src/Entity/NomDeLentite.php`) et de les decorer (annoter) avec la ligne `@ApiResource()`:

```
use ApiPlatform\Core\Annotation\ApiResource;
// Ne pas oublier de faire l'import du decorateur ci dessus !!

/**
 * @ApiResource()
 * @ORM\Entity(repositoryClass=MonEntiteRepository::class)
 */
class MonEntite
{
    ...
}
```

Auhtentication par JWT

Avant propos: Authentication avec un mot de passe hache

Etant donne que notre classe User admet une propriete `password` qui sera chiffree en BDD, il nous faut bien un moyen de pouvoir la mettre en place pour la premiere fois, une initialisation en quelque sorte. Pour cela, on ajoutera une colonne `plainPassword`

```
$ symfony console make:entity
```

```
>User
>plainPassword
>string (180)
>nullable? [yes] # lors de l'initialisation, (creation de compte) on viendra
affecter la valeur null en BDD a ce plainPassword et on utilisera dorenavant le
password chiffre

$ symfony console make:migration
$ symfony console doctrine:migration:migrate
```

Maintenant qu'on a un `plainPassword` qui va nous servir d'intermediaire, on doit creer un `DataPersister` pour l'entite User.

Un `DataPersister` est une classe qui nous permet d'ajouter de la logique supplementaire lors du process de `deserialization` avant d'ecrire definitivement en BDD.

Dans notre cas de figure, la logique a implementer est la suivante:

- On recoit un MDP 'plain' via requette `HTTP POST`
- On le recupere et on affecte sa valeur au MDP definitif
- On encode le MDP defintif
- On supprime le MDP provisoire
- On ecrit et on **persiste** en BDD

Un exemple de classe `UserDataPersister` ci dessous

```php
namespace App\DataPersister;
use ApiPlatform\Core\DataPersister\DataPersisterInterface;
use App\Entity\User;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class UserDataPersister implements DataPersisterInterface
{
    /**
     * @var UserPasswordHasherInterface
     */
    private $encoder;

    /**
     * @var EntityManagerInterface
     */
    private $manager;

    /**
     * UserPersister constructor.
     * @param UserPasswordHasherInterface $encoder
     * @param EntityManagerInterface $manager
     */
    public function __construct(
        UserPasswordHasherInterface $encoder,
```

```
        EntityManagerInterface $manager
    )
    {
        $this->encoder = $encoder;
        $this->manager = $manager;
    }

    /**
     ******************************************
     * Détermine si oui ou non notre objet $data
     * est une instance de App/Entity/User.
     ******************************************
     * @param $data
     * @return bool
     */
    public function supports($data): bool
    {
        return $data instanceof User;
    }

    /**
     ****************************************************************
     * Méthode venant encoder le  plain password de l'utilisateur
     * avant de pousser ses informations en BDD.
     ****************************************************************
     * @param $data
     * @return object|void
     */
    public function persist($data)
    {
        if($data->getPlainPassword()){

            $data->setPassword(
                $this->encoder->hashPassword(
                $data, $data->getPlainPassword()
            ));
            $data->eraseCredentials();

        }

        $this->manager->persist($data);
        $this->manager->flush();
    }

    /**
     ******************************************
     * Cette méthode précise quoi faire au moment
     * de la suppression de cet objet $data.
     ******************************************
     * @param $data
     */
    public function remove($data)
    {
        $this->manager->remove($data);
```

```
            $this->manager->flush();
        }
    }
```

Le fonctionnement dans le detail d'un `DataPersister` est en dehors du cadre de ce projet, il faut simplement garder en tete que c'est un moyen utile d'ajouter de la logique lors de l'ecriture en BDD.

Un autre cas dans lequel un Persister serait utile, c'est pour changer le nom d'un fichier d'image soumis par l'utilisateur, en quelque chose d'unique et de plus structure. Par ex: On recoit en entree un nom de fichier `photo1.jpeg` et on le transforme en `media/users/1/books/qw5ehsa52313j2.jpg`

Maintenant qu'on a toute la logique necessaire, on passe a la generation et la configuration de tokens JWT

Installation

```
$ composer require jwt
```

Generation de paire Cle publique - Cle privee

```
$ symfony console lexik:jwt:generate-keypair
```

Explication sans trop rentrer dans le detail: La cle publique servira a decoder une token entrante, tandis que la cle privee servira a generer des nouvelles token. Evidement toutes les 2 sont a garder imperativement de maniere securisee. **Veiller a ne pas les versionner** (dans .gitignore, cibler le fichier des cles `.pem` contenues dans `/config/jwt`)

Parametrer les fonctionnalites de login, declarer ses parefeus, et securiser les routes dans le fichier `security.yaml`

exemple de fichier `security.yaml`

```
security:
    enable_authenticator_manager: true
    password_hashers:
        App\Entity\User:
            algorithm: auto

    providers:
        # Dans notre projet, le fournisseur de la logique
        # utilisateur sera notre entite User
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

```
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        login:
            # 'patron' du chemin a controler
            pattern: ^/api/login

            # stateless: chaque requete vers login,
            # est independante de la derniere
            stateless: true

            # quel fournisseur de logique utlisateur
            provider: app_user_provider

            # etant donne que notre client distant
            # communique en JSON uniquement, on declare
            # le type de login a json_login (autre possibilite
            # ce serait du form_login si on avait des templates
            # genere cote serveur)
            json_login:

                # ici vu que la seule maniere de s'authentifier
                # est via JSON, alors le chemin est le meme
                check_path: /api/login

                # par defaut, le bundle lexik:jwt verifiera dans
                # le corps de la requete
                # une cle 'username' et une cle 'password',
                # or, nous on souhaite qu'au lieu de verifier le pseudo,
                # ce soit l'adresse mail qui serve d'identifiant unique
                username_path: email
                password_path: password

                # gestionnaires de succes et d'echec
                # (il convient de laisser les valeurs par defaut)
                success_handler:
lexik_jwt_authentication.handler.authentication_success
                failure_handler:
lexik_jwt_authentication.handler.authentication_failure

        # ce parefeu indique a Lexik:jwt que toutes les routes
        # commencant par /api pourront aussi etre 'controlees'
        api:
            pattern: ^/api
            stateless: true
            provider: app_user_provider
            guard:
                authenticators:
                    - lexik_jwt_authentication.jwt_token_authenticator

    # Controle d'acces personalise vers les routes qu'on souhaite proteger.
    # Il convient de commencer par les 'exceptions' en premier
```

```
    # et ensuite les cas generaux
    access_control:

        # la route login sera evidement accesible a tout le monde
        # afin qu'on puisse s'identifier
        - { path: ^/api/login, roles: IS_AUTHENTICATED_ANONYMOUSLY}

        # la route /api/users sera aussi accessible en mode anonyme
        # afin que l'on puisse creer un compte sans etre authentifie,
        # ce qui est logique.
        # D'ou l'autorisation vers la route en methode POST uniquement
        # (on souhaite pas qu'on puisse recuperer
        # des infos utilisateurs de maniere anonyme)
        - { path: ^/api/users, method: [POST], roles:
IS_AUTHENTICATED_ANONYMOUSLY }

        # pour le reste des routes commencant par /api,
        # il faudra etre authentifie!
        # c.a.d soit il y a une token valide en en-tete de requete
        # soit une reponse 401 (non autorise) sera retournee
        - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```