DOCUMENTATION TECHNIQUE

Table des matières

[Les contrôleurs Symfony : 2](#_Toc66181114)

[Créer des routes : 2](#_Toc66181115)

[TWIG : 3](#_Toc66181116)

[SECURITY.YAML : 3](#_Toc66181117)

[Base de données : 4](#_Toc66181118)

[Utilisateurs : 6](#_Toc66181119)

[Création des formulaires 6](#_Toc66181120)

[Page d’accueil 7](#_Toc66181121)

[Modification des vues 8](#_Toc66181122)

# Les contrôleurs Symfony :

Un contrôleur est une fonction PHP que vous créez et qui récupère l'information depuis l'objet Request d'une requête HTTP, puis crée et retourne une réponse HTTP (un objet Response Symfony).

La réponse peut être une page HTML, un document XML un tableau JSON sérialisé, une image, une redirection, une erreur 404, ou toute autre chose que vous pouvez imaginer.

Le contrôleur exécute quelque logique arbitraire selon les besoins de votre application pour interpréter le contenu de la page.

Le but d'un contrôleur est toujours le même : créer et retourner un objet Response. Le long de son parcours, il pourrait lire l'information depuis la requête, charger une ressource de base de données, envoyer un mail, ou placer une information sur la session utilisateur. Mais dans tous les cas, le contrôleur va éventuellement retourner l'objet Response qu'il va fournir en retour au client.

Chaque requête gérée par un projet Symfony passe par le même cycle de vie simple. Le Framework s'occupe de toutes les tâches répétitives : vous avez juste à écrire votre code personnalisé dans les fonctions contrôleur :

* Chaque requête est gérée par un simple fichier contrôleur frontal (ex. : app.php ou app\_dev.php) qui amorce l'application ;
* le routeur lit les informations depuis la requête (ex. : l'URI), trouve une route qui correspond à l'information, et lit le paramètre du contrôleur depuis la route ;
* le contrôleur est exécuté depuis la route correspondante et le code interne de celui-ci crée et retourne un objet Response ;
* Les entêtes HTTP et le contenu de l'objet Response sont renvoyés au client. Le contrôleur correct de la route correspondante est exécuté et le code de celui-ci crée et retourne l'objet Response approprié ;
* Les entêtes HTTP et le contenu de l'objet Response sont renvoyés au client.

# Créer des routes :

Les itinéraires peuvent être configurés en YAML, XML, PHP ou en utilisant des attributs ou des annotations. Tous les formats offrent les mêmes fonctionnalités et performances, alors choisissez votre favori. Symfony recommande les attributs car il est pratique de placer la route et le contrôleur au même endroit. Créer des Routes comme Attributs ou Annotations

Sur PHP 8, vous pouvez utiliser des attributs natifs pour configurer les routes tout de suite. Sur PHP 7, où les attributs ne sont pas disponibles, vous pouvez utiliser des annotations à la place, fournies par la bibliothèque Doctrine Annotations. Si vous souhaitez utiliser des annotations au lieu d'attributs, exécutez cette commande une fois dans votre application pour les activer. Création de routes dans des fichiers YAML, XML ou PHP

Au lieu de définir des routes dans les classes de contrôleurs, vous pouvez les définir dans un fichier YAML, XML ou PHP distinct. Le principal avantage est qu'ils ne nécessitent aucune dépendance supplémentaire. Le principal inconvénient est que vous devez travailler avec plusieurs fichiers lors de la vérification du routage de certaines actions du contrôleur.

# TWIG :

Le langage PHP qui était un moteur de gabarit à ses débuts est maintenant devenu un langage complet capable de supporter la programmation objet, fonctionnelle et impérative.

L'intérêt principal d'un moteur de gabarit est de séparer la logique de sa représentation. En utilisant PHP, comment définir ce qui est de la logique et ce qui est de la représentation ?

Pourtant, nous avons toujours besoin d'un peu de code dynamique pour intégrer des pages web :

* pouvoir boucler sur une liste d'éléments ;
* pouvoir afficher une portion de code selon une condition ;
* ou formater une date en fonction de la date locale utilisée par le visiteur du site...

Voici pourquoi Twig est plus adapté que le PHP en tant que moteur de gabarit :

* il a une syntaxe beaucoup plus concise et claire ;
* par défaut, il supporte de nombreuses fonctionnalités utiles, telles que la notion d'héritage ;
* et il sécurise automatiquement vos variables.

Éléments de syntaxe :

Twig supporte nativement trois types de syntaxe :

* {{ ... }} permet l'affichage d'une expression ;
* {% ... %} exécute une action ;
* {# ... #} n'est jamais exécuté, c'est utilisé pour des commentaires.

# 

# SECURITY.YAML :

Le fichier security.yml bénéficie du mécanisme de configuration en cascade et peut inclure des constantes.

La configuration de l'application par défaut peut être substituée pour un module en créant un fichier security.yml dans le répertoire config/ du module. Les clés principales sont les noms des actions sans le préfixe execute (index pour la méthode executeIndex par exemple).

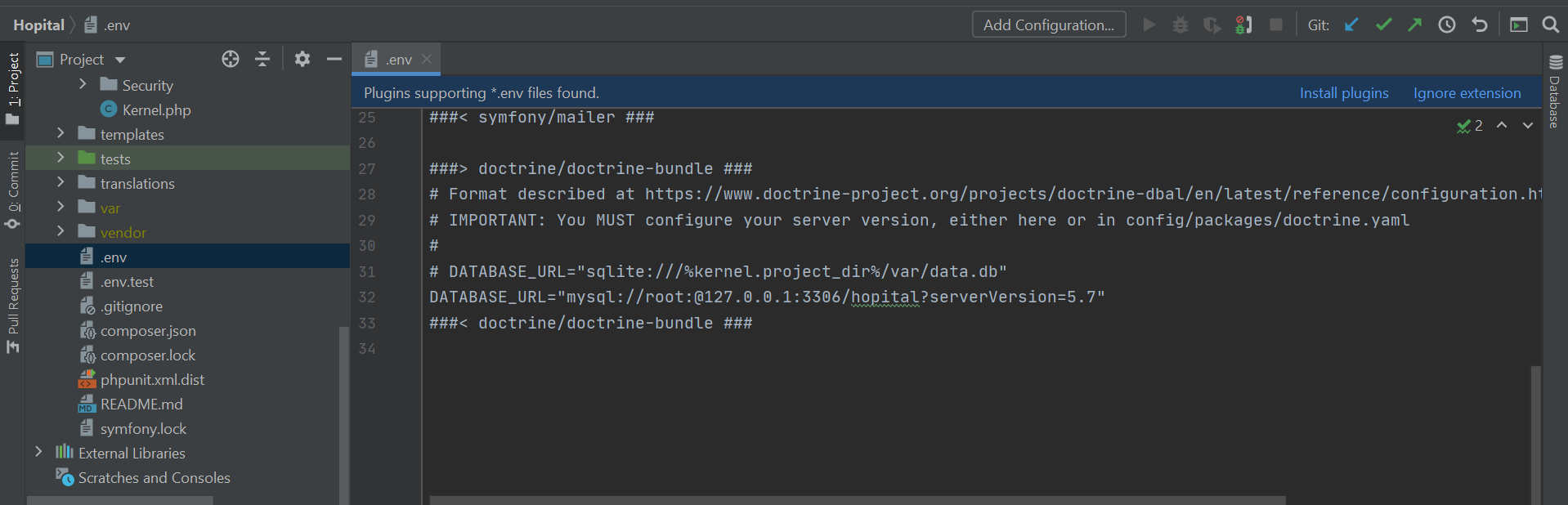
Pour déterminer si une action est sécurisé ou non, symfony regarde pour l'information dans l'ordre suivant :

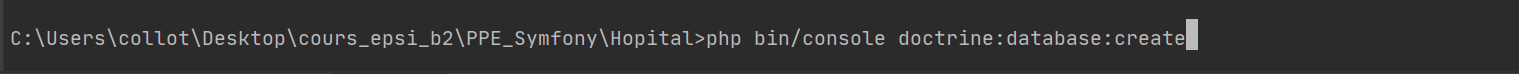
* une configuration pour l'action spécifique dans le fichier de configuration du module, si elle existe;
* une configuration pour l'ensemble du module dans le fichier de configuration du module si elle existe (sous la clé all); l'application de configuration par défaut (sous la clé default). Les mêmes règles de priorité sont utilisées pour déterminer les credentials nécessaires pour accéder à une action.

# Base de données :

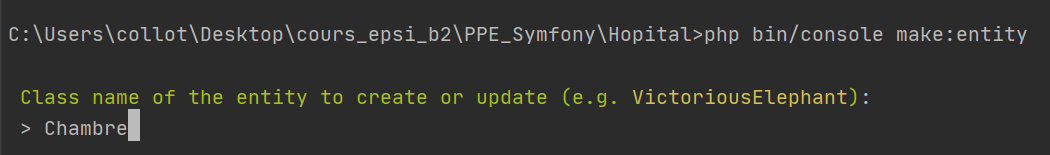
Pour commencer nous avons créé la base de données grâce au bundle doctrine.

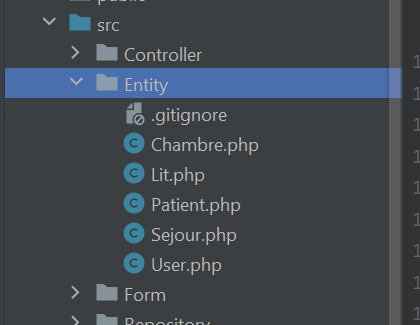
Nous avons créé la base de données grâce à la commande suivante qui nécessite la modification du fichier .env :



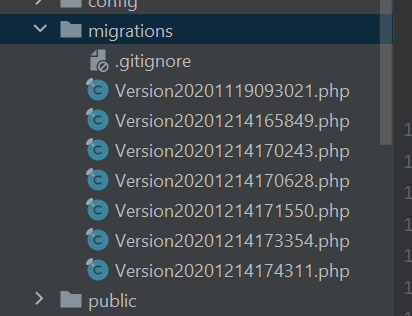


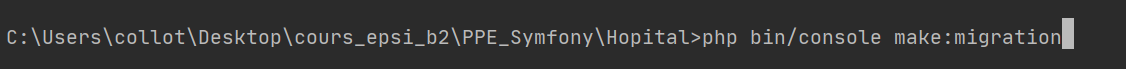
Avec le bundle maker nous avons créé des entités qui représente les différentes tables de la base de données et qui contienne les différents attributs ainsi que les méthodes permettent de les exploiter par la suite dans le Framework.



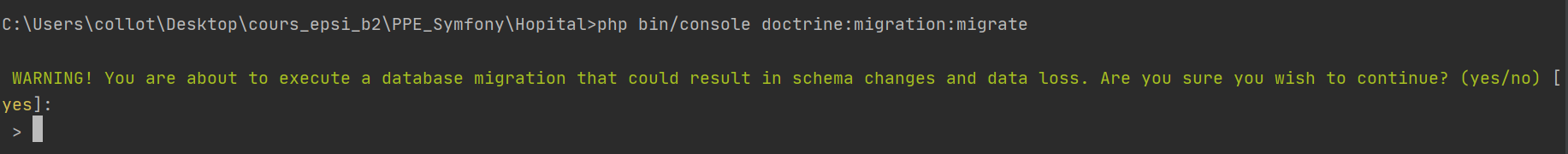


A chaque modification de la base de données nous avons créé des migrations également grâce à maker.



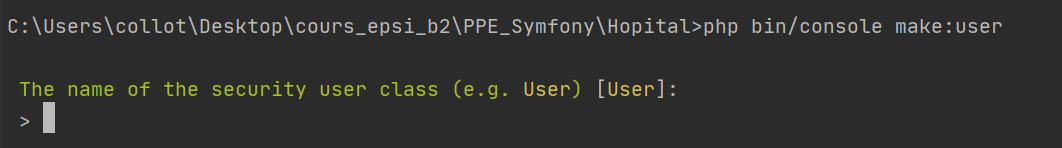


Une fois les migrations créées nous les envoyons vers la base de données grâce à la commande suivante :



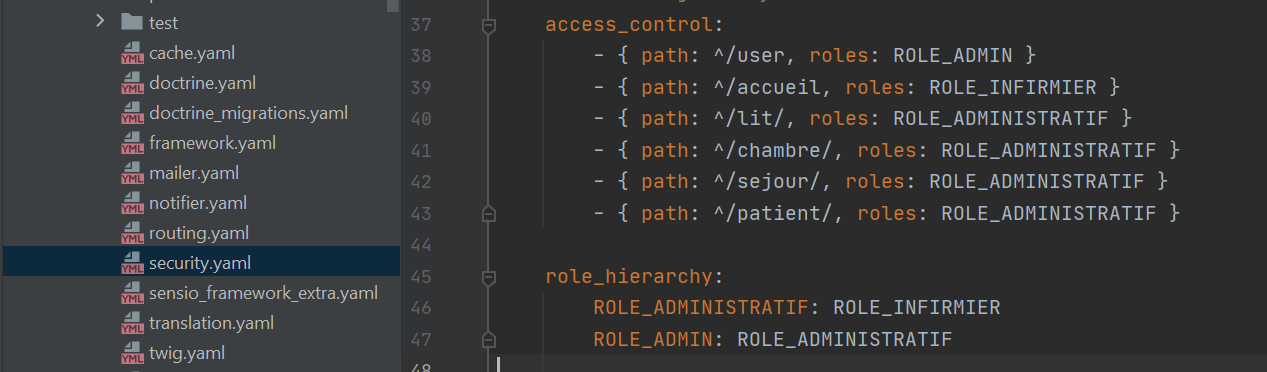
# Utilisateurs :

Pour créer les utilisateurs nous avons utilisé le bundle maker avec la commande suivante :



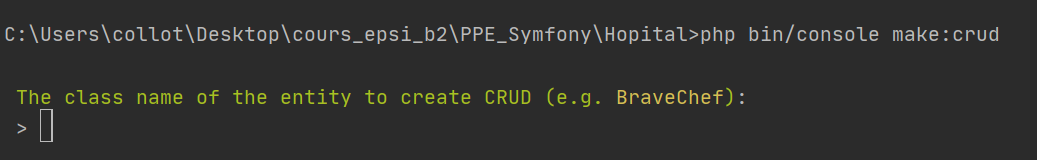
Cette commande créée une entité pour la base de données, un formulaire de connexion, un contrôleur pour ce formulaire et un fichier pour l’authentification.

Nous avons créé différents rôles selon les utilisateurs (infirmier, administratif et administrateur). Chaque rôle a accès à différentes fonctionnalités de l’application :

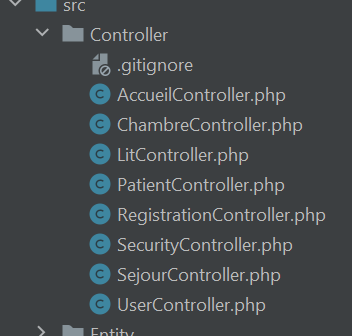


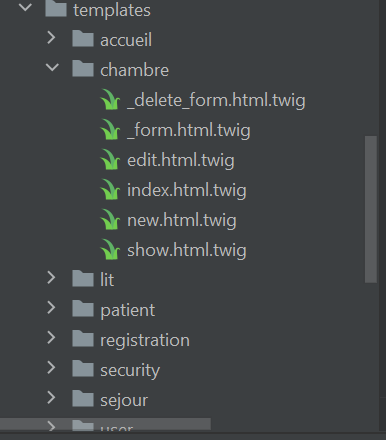
# Création des formulaires

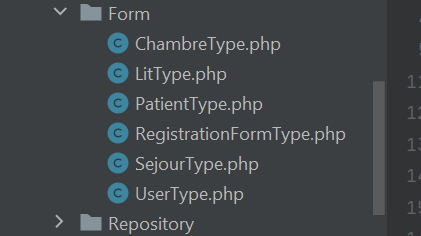
Nous avons créé les formulaires de gestion des différentes entités (ajouter, supprimer, modifier) avec la commande suivante :



Cette commande crée un contrôleur et le formulaire en relation avec l’entité voulu.

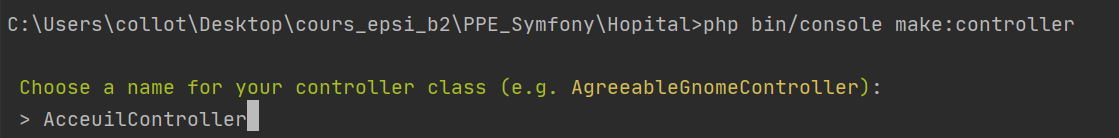






# Page d’accueil

Pour créer la page d’accueil nous avons créé un controller avec la commande suivante :



Le controllrer renvoi vers la vue de l’accueil que nous avons donc créé. C’est ici que l’on peut naviguer entre toutes les parties de l’application.

# Modification des vues

Nous avons utilisé un thème bootstrappe pour le dissigne l’application. Ainsi que le moteur de templates twig.

