OUDIN Mathis

Partiel : Partie théorique

1. Symfony est un framework à vocation full-stack php mais principalement utilisé pour du développement back-end en php. Par défaut, Symfony se base sur une architecture MVC (Model Vue Controller) permettant un développement structuré et évolutif. De part sa nature en php, on peut utiliser des containers pour nos applications symfony afin de permettre les meilleures performances au moment de l'hébergement. De plus, Symfony dispose de plusieurs extensions et packages officiels et également communautaires justifiant la pérennité du framework et de sa sécurité.

2. L'architecture MCV (Model Vue Controller) permet de séparer les responsabilités des composants de l'application.

   On retrouve dans un premier temps : le Modèle qui est notre Entity (entité) à savoir les colonnes des tables de notre base de données, par exemple : src/Entity/Product.php.

   Ensuite, on a la Vue qui représente les composants UI (Twig dans le cas de Symfony) pouvant être représenté par l'interaction d'un utilisateur via des boutons, formulaires… tout simplement ce que l'utilisateur voit. Par exemple : templates/product/addToCartButton.html.twig

   Enfin, il y a le Controller qui vient intercepter les requêtes et nous rediriger vers les bonnes ressources, il fait le pont entre la Vue/Model et les coordonne. exemple : src/Controller/Product/ProductController.php

3. Une route permet l'accès à une ressource en particulier et fait le lien entre URL et une méthode d'un Controller. Elle permet de capturer des paramètres dynamiques dans l'URL.

   Exemple de route dynamique : /product/{id}/

   Dans cet exemple, on est sur la route des produits et l'on passe le paramètre dynamique id ({id}). Au sein d'une application, chaque ligne de notre base de données possède un identifiant unique qui lui est propre, passer ce paramètre dans cette route nous permettrait d'obtenir les données d'un produit en particulier. En passant l'id du produit, notre Controller demandera à la couche de Service et de Repository d'aller chercher (via un findById ou findBy avec l'id en params) puis nous renverra le résultat.

4. Imaginons que nous ayons, sur une page produit, un titre (nom du produit), une description produit et un bouton d'achat. Et que nous voulons que ces derniers soient

des dérivés de notre charte graphique et de notre DA de manière générale, on peut très bien créer un fichier composant (par exemple : button.html.twig) que l'on va utiliser dans buttonAddToCart.html.twig comme base et que l'on va étendre pour notre cas d'utilisation avec par exemple, juste un fond de couleur qui change et un effet au hover particulier mais tout le reste étant le même. Cela permet d'éviter la répétition avec un code plus propre et plus maintenable et si un jour, on décide que les tous les boutons du site doivent être en police Roboto, le changement se ferra sur tous les boutons de notre application du moment qu'ils extends tous de ce composant mère et que l'on applique ce changement sur ce dernier et non les composants enfants. Cela s'applique pour tous les éléments de notre page produit à savoir le titre (title) qui être imaginons, comme ceux que l'on retrouve de partout sur notre site et de même pour le contenu (content) qui doit être un paragraphe classique de notre application mais juste en gras. On créera alors un fichier productContentDescription.html.twig qui extends de paragraph.html.twig mais qui surcharge la propriété font-weight.

5. Doctrine ORM est un ORM (Object-Relationnal Mapping ou mapping objet-relationnel).

   Doctrine permet de manipuler la base de données via des objets PHP plutôt que du SQL brut.

   Donc, dans un premier temps on définit nos modèles de données de nos entités via une commande CLI dans le terminal ou alors à la main via des classes PHP (Entity).

   Ensuite, on vient générer le fichier dit de migration qui est une classe PHP contenant le SQL nécessaire (Migration).

   Et pour terminer, on vient exécuter ce fichier de migration afin d'appliquer les changements en base de données (Migrate).

   Doctrine vient également avec des méthodes de repository permettant l'accès à nos données en passant ou non par des paramètres. On y retrouve, par exemple : find, findAll, findBy, findById, delete, update, create…

   Ces méthodes sont natives et permettent rapidement la création, la modification, la suppression d'éléments dans notre base de données mais également, on a des wrapper de méthode pour effectuer de la recherche au sein de notre BDD comme findAll qui vient récupérer sans paramètre, tous les éléments de notre table, ou alors findById qui vient récupérer les données d'un éléments et utilise l'id en paramètre pour savoir lequel ciblé. Par exemple : pour la route /product/{id}/ -> findById($id).

6. Le composant Security gère tout ce qui concerne la sécurité : identification des utilisateurs, contrôle d'accès, protection CSRF, hashage de mots de passe, etc.

L'authentification permet de vérifier qui est l'utilisateur connecté sur cette session en passant par exemple via le formulaire de login/register, on enregistre localement au sein d'un JWT sur la machine locale de l'utilisateur son identité.

L’autorisation permet de vérifier les droits d’accès à telle ou telle ressource sur notre application d’un utilisateur en utilisant son rôle (utilisateur, modérateur, administrateur…). Nos autorisations et accès peuvent varier suivant notre statut avec par exemple : le rôle ROLE_ADMIN pouvant voir, créer, modifier ou même supprimer un article de blog pendant qu’un simple utilisateur ROLE_USER ne pouvant que voir l’article mais peut également modifier ou supprimer ce dernier que s’il lui appartient (s’il en est l’auteur).
L’accès aux routes de suppression et de modification lui seront refusées s’il n’est pas l’auteur et les boutons (ui de manière générale) seront également différents.