Rendu projet serveur web

Introduction:

Pour ce projet qui consistait à créer un blog en version serveur web, nous avons décidé d'utiliser le langage php pour le côté serveur, mySql / phpMyAdmin pour le côté base de données.
Nous avions besoin d'un gestionnaire de dépendances pour gérer les sessions en utilisant les tokens, pour ce faire, nous avons utilisé composer, il nous a permis d'implémenter la librairie jwt token en php sans soucis.
Pour lancer le projet, nous avons créé un virtual host grâce à WampServer, puis dans l'url il suffit de mettre le nom du virtualhost créé. Dans le fichier connBdd.php sur la variable "$bdd", il faut mettre votre nom de base de donnée pour que le serveur ait accès à votre base de données. Pour créer les tables dans votre base de données, il suffit d'entrer dans le terminal la commande "php createBdd.php" et cela créera les tables nécessaires au fonctionnement du serveur. Pour pouvoir utiliser toutes les fonctionnalités sur blog, créer un compte admin avec comme rôle "Admin".

Architecture:

Un dossier *public* qui regroupe tous les sous-dossiers et les fichiers nécessaires au serveur pour fonctionner.
Un dossier *vendor* avec deux fichier composer (un .json et un .lock) pour utiliser composer et utiliser les tokens jwt, on ne rentrera pas dans les détails pour expliquer le gestionnaire de dépendances composer et nous allons plutôt expliquer le dossier public.

Comment est structuré le dossier *public*?

Nous avons opté pour la structure MVC pour faire marcher ce projet. Donc le dossier *public* est composé d'un dossier *controller* (un seul controller.php) qui va mettre en liaison les appels à la bdd (base de données) et les vues pour afficher ce que l'on souhaite.
Le dossier *model* va donc contenir un fichier model.php qui va avoir plusieurs fonctions pour faire des requêtes Sql sur notre base de données et récupérer les lignes que nous souhaitons exploiter, il a aussi un fichier model_inscription.php et un fichier modelLogin.php séparé du rest pour gérer l'inscription et la connexion au blog.
Le dossier *view* va donc regrouper tous les fichiers nécessaires à l'affichage du blog selon les pages demandées par l'utilisateur.

Exemple: On veut la page de tous les articles, le controller appel le fichier model.php puis la fonction getArticles() pour récupérer dans la bdd les articles puis il appel le fichier IndexView.php pour afficher en html les articles proprement.

Mais nous avons besoin de beaucoup de controller différents, évidemment nous n'allons pas faire un fichier controller pour chaque page spécifique. On a donc rajouté un routeur dans notre projet qui est simplement le fichier racine index.php qui va permettre d'appeler le bon controller pour chaque page demandée.

Notre structure a donc une couche supérieure par dessus notre MVC pour indiquer les bons chemins et avoir ce que l'on souhaite sur votre écran !

Bien sûr nous avons d'autres dossiers et fichiers dans le dossier *public* pour gérer un tas d'autres fonctionnalités sur notre blog, nous allons les expliquer ci-dessous.

Un affichage propre grâce à un template:

Tous nos fichiers du dossier *view* vont se ressembler, on ne va pas ouvrir une balise html sur chaque fichier de vue, on a donc créer un fichier template.php qui va être appliqué sur chaque fichier de vue. Nous faisons appel à un stylesheet de w3school pour avoir une vue plus esthétique étant donné que le projet se repose sur le côté serveur et non html/css.

Les sessions sans états:

Un dossier *token* apparaît dans le dossier *public*, vous vous en doutez, c'est la gestion d'authentification !
Tout d'abord, un fichier token_encode.php qui va permettre de créer un token jwt à chaque nouvelle connexion. La clef secrète aurait pu être appelée dans un fichier sécurisé mais pour ce petit projet, nous l'avons juste écrit simplement en code brut.
Le token prendra donc les informations suivantes:
une date de publication, une date d'expiration, l'id, le pseudo et le rôle de l'utilisateur (stocker dans un array data). Puis on encode la clef secrète au payload pour avoir notre token avec la fonction JWT:encode. On met le token dans un cookie avec une date d'expiration de 3600s.
Et voilà notre token est prêt à être décodé, pour ce faire on utilise un fichier token_decode.php avec un try catch pour regarder si le token est valide.
On extrait du token les informations de l'utilisateur qui était stockées dans un array data dans le payload du token.
Ce fichier token_decode.php va être utilisé partout où on aura besoin de session, on le place donc au début du routeur index.php pour tester les droits des utilisateurs et

d’autres fonctionnalités (tel que les affichages différents selon l’état de l’utilisateur (connecté / déconnecté).
Le dernier fichier toek_unset.php permet de détruire le cookie, il est utile pour se déconnecter du site.

La connexion à la base de données:

On a un dossier *bdd* avec un premier fichier connBdd.php pour se connecter à notre base de donnée en utilisant toutes les informations nécessaires (host, user, psw, port..). On applique aussi dans les options l’affichage des erreurs, puis par dessus un try catch si la connexion à la bdd échoue.
Le deuxième fichier createBdd.php est pratique mais optionnel, il permet de créer les tables dans la bdd pour faire fonctionner le site. On peut aussi se contenter de créer les tables à la main sur phpMyAdmin.

La gestion des erreurs:

Pour gérer les erreurs, on a utilisé une petite astuce.. Le but est de pouvoir arrêter le programme n’importe où dans n’importe quel fichier. Pourquoi ne pas placer un try catch tout en haut de notre projet? On se place donc sur la racine qui est le fichier que le serveur va appeler en premier: index.php. Sur notre fichier routeur (index.php), on à placé ce try catch par dessus tous les appels au controller. L’avantage énorme est le suivant: si une erreur arrive dans un de nos fichiers, vu que ce fichier est appelé par le routeur, il a juste a utiliser la fonction “throw new Exception” qui va être immédiatement récupéré par le routeur avant de continuer de faire des bêtises. Cette erreur va être jeté par un des fichier puis capturer par le routeur, par dessus on a rajouté une fonction dans le controller qui appel le fichier error.php pour afficher proprement l’erreur reçu, ce fichier error.php va donc pouvoir afficher toutes les erreurs que l’on a décidé de générer !

Et voilà, nous avons la description assez générale de notre architecture, bien sûr nous aurions pu rentrer dans le détail de chaque sous-dossier, mais cela aurait donné un rendu beaucoup trop long et détaillé..
Nous espérons avoir expliqué assez généralement pour que la compréhension soit claire sans être trop compliquée. Mais ce n’est pas encore fini, il y a une dernière partie qui va compléter ce projet..

La partie REST

Qu’est ce qu’un serveur web sans rest en 2021? c’est en regardant les autres sites connues qu’on se rend compte que rest est important, surtout en php quand on sait

que l'on ne peut faire que des GET et des POST, on se doit d'avoir un côté rest dans notre serveur voir n'avoir que du rest et laisser tomber la structure MVC !

Dans notre routeur index.php, on peut à partir de l'URL avoir accès à nos différentes pages et faire les requêtes que l'on souhaite, mais étant donné que l'on a des pages avec des formulaires pour envoyer nos données, cela devient vite ennuyant pour avoir accès à ces pages qu'à partir de l'URL..
Notre routeur se divise donc en deux parties, une partie MVC qui est appelée avec le mot "action" dans l'URL et une partie REST qui est appelée avec le mot "rest" dans l'URL.
On a donc rajouté un dossier *rest* dans le dossier *public* pour pouvoir avoir notre API Rest propre et dissocier du MVC.
Pour le rest, plus besoin de formulaire pour envoyer nos données et faire des requêtes, on utilise le Json et on affiche nos données en Json, aussi on gère les erreurs différemment avec des codes de retour Http et des messages en Json.
On peut donc appeler notre API avec un logiciel comme Postman ou Insomnia et au lieu d'utiliser des formulaires pour par exemple créer un article, on va créer un raw en Json avec des accolades pour envoyer le titre et le contenu de l'article.
On va aussi gérer les types des requêtes avec la variable globale $_SERVER et son indice "request_method" pour être sûr que l'utilisateur va demander du GET et non du PUT par exemple.
Par contre, nous avons gardé les cookies dans notre partie rest et non utilisé le header "authorization" pour les tokens.
Ainsi grâce au rest, on respecte bien les codes de retour Http que l'on avait pas implémenter dans la partie MVC étant donné que l'on affiche les erreurs avec une page spécifique.

Conclusion:

Comme dit précédemment, il y a sûrement des parties spécifiques que nous n'avons pas traité dans ce rendu pour ne pas alourdir davantage la taille du rendu et avoir une lecture plus agréable et globale de notre projet.
Par exemple on aurait pu expliquer la fonction "htmlspecialchars" qui permet d'éviter la faille XSS (on ne rentrera pas dans les détails encore une fois).
Notre projet est donc structuré avec un partie MVC et une partie REST qui sont appelées quand on le souhaite dans le routeur index.php. On peut utiliser notre site qu'en "cliquant" sur les différentes pages et cela générera bien dans l'URL les infos donc le serveur a besoin pour fonctionner, on peut aussi s'amuser à n'utiliser que l'URL et un logiciel comme Postman pour ajouter les formulaires aux requêtes et faire fonctionner la partie MVC sans "cliquer" sur les pages.
La partie REST quant à elle est assez pratique à utiliser avec Postman pour pouvoir envoyer tout ce dont le serveur a besoin avec du Json.