# Préambule

Jusqu’à présent nous avons développé un site avec :

* une page **/login** qui affiche un formulaire de connexion pour l’administrateur (à l’aide d’identifiants “en dur” dans le fichier constants.php de l’application)
* une page **/home** (la page par défaut de CodeIgniter)

Nous avons utilisé le mécanisme de redirection pour rediriger l’utilisateur qui a soumis les bons paramètres dans le formulaire de login vers **/home**

Mais nous avons encore bien des choses à faire :

* Notre site ne charge pas de style 😯
* Notre site ne permet pas aux abonnés de se connecter 🙁
* Notre site ne vérifie pas qu’on est connecté pour afficher la page **/home** 😨
* Notre site n’affiche pas de données dynamiques (provenant de la base de données)

Nous allons aujourd’hui pallier ces problèmes en utilisant les fonctionnalités de CodeIgniter prévues pour chacun de ces besoins.

# Chargement de styles : les ressources *statiques*

Une application web (telle que celle que nous développons avec le *framework* CodeIgniter) utilise des ressources dites *dynamiques* : en effet, chaque requête sur une URL va enclencher tout un processus :

* L’URL est comparée aux routes applicatives (définies dans **Routes.php**)
* La route qui correspond à l’URL / la méthode invoque une **méthode** d’une classe *Controller* (présente dans le dossier **Controllers**, et qui étend la classe **BaseController**
* La méthode du controller invoque un ensemble de vues (présentes dans le dossiers **Views**)

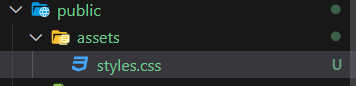
Ces ressources sont **recalculées dynamiquement** dès que l’on effectue une requête.

Cependant, un site web définit aussi (comme vu dans les TPs précédents) des ressources **qui ne changent pas lors du traitement d’une requête**, aussi appelées des ressources **statiques** :

* Des fichiers de style CSS
* Des scripts JavaScript

Ces ressources peuvent être invoquées par tout visiteur le souhaitant, sans restrictions ni contrôle particulier. On dit que ce sont des ressources **publiques**.

Afin qu’elles puissent être accessibles, CodeIgniter vous fournit un dossier **public** dans lequel vous pouvez créer de telles ressources :



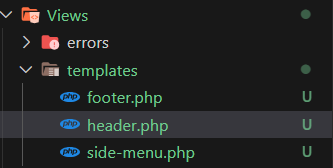
*Dans la copie d’écran, j’ai créé un dossier* ***assets*** *dans le répertoire* ***public****, qui contient mon fichier de style* ***styles.css***

Ces ressources peuvent donc être ajoutées dans les **vues** que l’on écrit dans le dossier **Views** du projet :

****

💡*Afin d’éviter de réécrire toute votre page HTML dans chacune des vues que vous utilisez (avec le <html>, <head>, <body>, etc…), vous devez (comme déjà vu dans le TP2) utiliser des fragments de vues HTML appelés* ***templates*** *(un template pour le début de la page qui définit le début du document HTML et l’en-tête de votre site, et un template qui définit le footer et ferme les balises <body> et <html>).*

*Pour ce faire, une bonne pratique est de créer un dossier* **templates** *qui va contenir ces fragments de vue qui permettent de générer votre page complète (et qui peuvent être utilisées par plusieurs controllers / méthodes !) :*

**

*De cette manière, vous* ***réutilisez un maximum votre code****, et* ***vous gagnez beaucoup de temps*** *dans la réalisation de vos pages !*

👉 *Définissez un fichier de styles et vérifiez que votre page l’interprète correctement, en créant un header et un footer affiché sur toutes les pages de votre application.*

# Utilisation des *Models*

💡 *Cette partie nécessite que votre base de données MySQL fonctionne, donc que votre Wamp soit actif !*

Afin de permettre aux abonnés de la bibliothèque de se connecter à notre application, nous avons besoin d’aller chercher dans notre base de données les abonnés et de vérifier que les informations saisies dans la page de login soient correctes. Pour ce faire, nous allons utiliser les classes *Models* que propose CodeIgniter.

👉 Pour rappel, dans une architecture logicielle MVC (**Model-View-Controller**), les classes **Models** sont les classes qui sont chargées de représenter les données que nous manipulons (ou les manières de les obtenir).

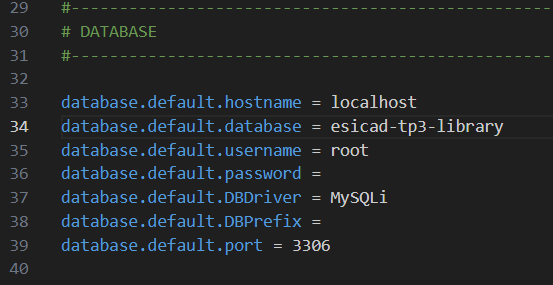
CodeIgniter propose [2 manières de gérer ces données](https://www.codeigniter.com/user_guide/models/index.html) :

* Une manière simple : créer des classes **Models** qui permettent d’écrire des méthodes allant chercher des données en simplifiant l’écriture de requêtes SQL
* Une manière plus élaborée : créer des classes **Entities** qui représentent une instance d’une entité d’un MLD

👉 *Dans un premier temps, nous allons utiliser les classes Models afin de charger et de manipuler des données. Nous complèterons notre application avec des Entities lors d’une prochaine séance*

**⇒ Voir documentation complète des Models :** [**Using CodeIgniter’s Model**](https://www.codeigniter.com/user_guide/models/model.html)

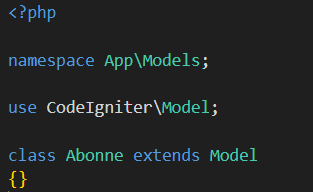
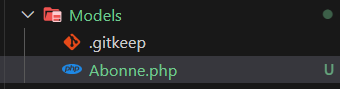
Afin de gérer la connexion d’un abonné, nous devons en premier lieu vérifier que nous avons configuré les bonnes informations relatives à la base de données dans les fichiers de configuration de CodeIgniter.  
La méthode **la plus simple** est d’utiliser le fichier .env (si vous ne l’avez pas, copiez le fichier env de votre projet et renommez-le en .env), puis de décommenter et remplir les propriétés suivantes :



👍 *De cette manière, CodeIgniter sera capable de gérer la connexion à la base de données de notre application tout seul !*

Une fois notre configuration de base de données définie, il nous faut définir des **Models** qui gèrent chacun une table de la base de données et permettent de récupérer des données selon ce dont on a besoin dans nos controllers.

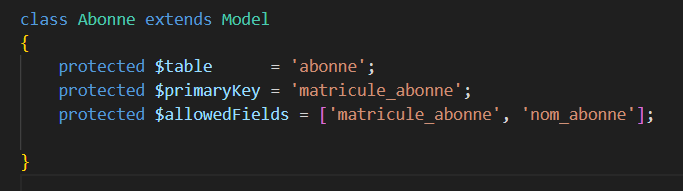
Puis nous allons devoir créer une classe **Abonne** dans le fichier **Abonne.php** dans le dossier **Models** : **Abonne.php** :



☣️ **ATTENTION : UTILISER LE MÊME   
NOM DE CLASSE ET DE FICHIER !**☣️  
*(demandez à Cédric pourquoi* 😹*)*

Cette classe Abonne **hérite** de la classe Model de CodeIgniter : cet héritage permet d’utiliser les méthodes de CodeIgniter permettant de faire des requêtes SQL simplement 👍

Nous allons définir dans notre classe Abonne le nom de la table, le nom de la clé primaire ainsi que les champs que notre application va vouloir manipuler (pour la connexion) :



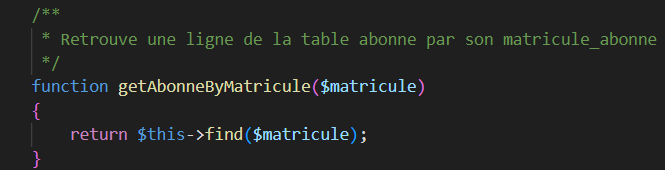
*On précise ainsi à CodeIgniter que les* ***Models*** *Abonne sont localisés dans la table abonne, que leur clé primaire est dans le champ* ***matricule\_abonne****, et que nous voulons pouvoir utiliser les champs* ***matricule\_abonne*** *et* ***nom\_abonne****.*

👉 *Il vous faudra modifier ce fichier lorsque vous voudrez utiliser les autres colonnes de la table* ***abonne*** *dans vos pages !*

Une fois la table, la clé primaire et les colonnes définies, nous allons pouvoir implémenter une méthode qui va chercher à retrouver un abonné selon la clé primaire, c’est-à-dire son numéro de matricule.

*Pour le moment, notre application ne gère pas de notions de login / mot de passe, mais nous pourrons les ajouter plus tard* 👍

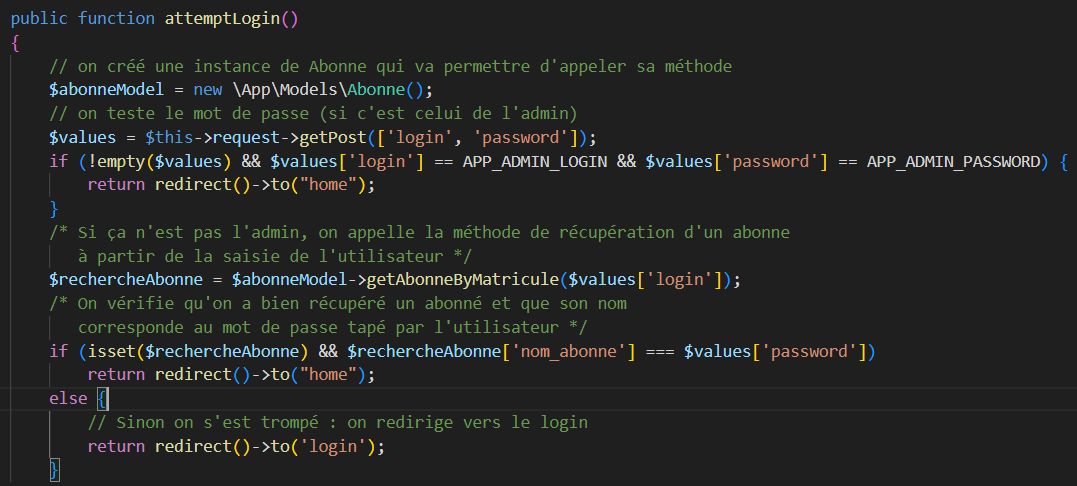
Écrivons une méthode **getAbonneByMatricule** qui va retrouver une ligne de la table abonne par son matricule :



👉 *Cette méthode utilise la méthode* [find](https://www.codeigniter.com/user_guide/models/model.html#find) *de CodeIgniter qui permet de retrouver une ligne de Model selon la clé primaire définie dans la propriété $primaryKey.*

👉 *CodeIgniter propose beaucoup d’autres méthodes pour rechercher / insérer / modifier / supprimer des lignes en base de données qu’il vous faudra utiliser pour la suite de votre projet. Il est important de lire la documentation des models* [*ici*](https://www.codeigniter.com/user_guide/models/model.html#finding-data)

Puis nous allons modifier **Connection::attemptLogin** en allant utiliser la méthode **getAbonneByMatricule**, afin de rechercher si la saisie de l’utilisateur correspond à un abonné enregistré en base :



👉 *Testez votre nouvelle implémentation de la fonction* ***attemptLogin*** *en vous connectant en tant qu’abonné. Pour ce faire, vous devez utiliser* [*PhpMyAdmin*](http://localhost/phpmyadmin/) *afin de créer une ligne dans la table des abonnés manuellement dans votre base de données !*

# 

# Filters

Maintenant que l’administrateur et que nos abonnés sont capables de se connecter, il faut que nous nous assurions que les requêtes vers les URLs de notre application qui nécessitent d’être connectés redirigent vers la page de login.

Pour ce faire, au lieu de devoir faire le contrôle dans chaque méthode liée à une route protégée, CodeIgniter propose une fonctionnalité appelée les **Filters** (documentation [ici](https://www.codeigniter.com/user_guide/incoming/filters.html)).

Les **Filters** permettent (entre autres) d’empêcher l’appel d’une méthode d’un Controller en *filtrant* l’appel à la méthode.

Un Filter est donc une classe qui implémente l’interface **FilterInterface** et qui doit définir 2 méthodes :

* **before**: est-ce que le Filter doit filter l’exécution **avant** l’appel de la méthode du controller
* **after**: est-ce que le Filter doit filter l’exécution **après** l’appel de la méthode du controller

Pour notre besoin, nous aimerions que les pages qui nécessitent d’être connecté redirigent vers le login si on n’est pas connecté avant que l’on affiche la page demandée initialement.

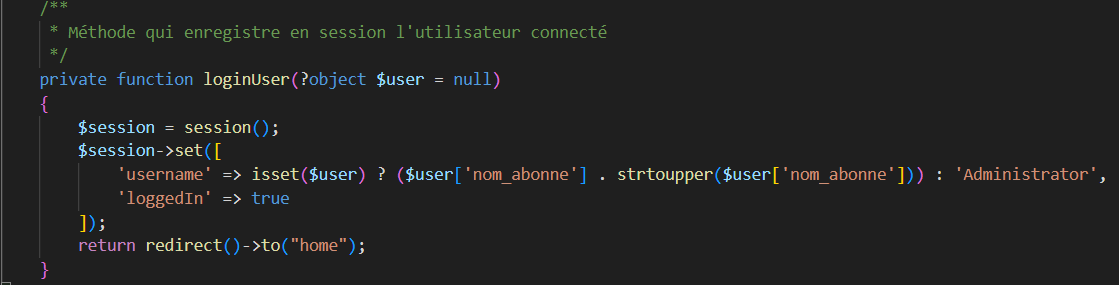
**👉 Il va donc falloir utiliser la méthode before pour renvoyer une redirection si on n’est pas connecté.**

Pour identifier si un utilisateur est connecté ou pas, nous allons utiliser le mécanisme de **session** proposé par PHP.

*Pour rappel*, la session est un mécanisme qui stocke des données sur le serveur et qui renvoie un **cookie 🍪** au client afin que chaque nouvelle requête puisse récupérer les données stockées.

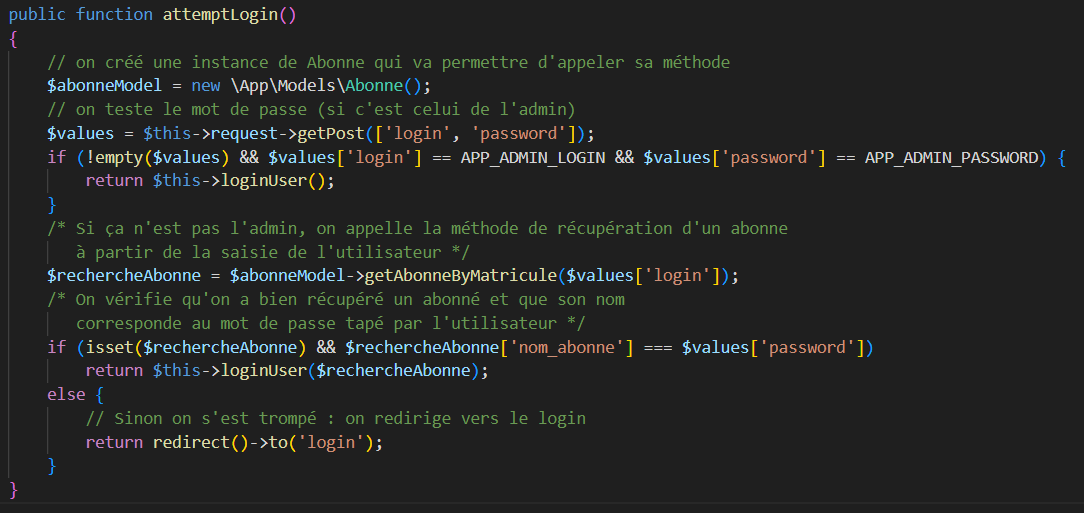
Le mécanisme de session est repris et simplifié par le framework CodeIgniter, en utilisant la fonction session() (documentation [ici](https://www.codeigniter.com/user_guide/libraries/sessions.html#initializing-a-session)), on va exploiter ce fonctionnement de la manière suivante :

1. Dans Connection.php, on écrit la méthode loginUser qui va **enregistrer en session l’utilisateur connecté** et effectuer la redirection vers la page /home



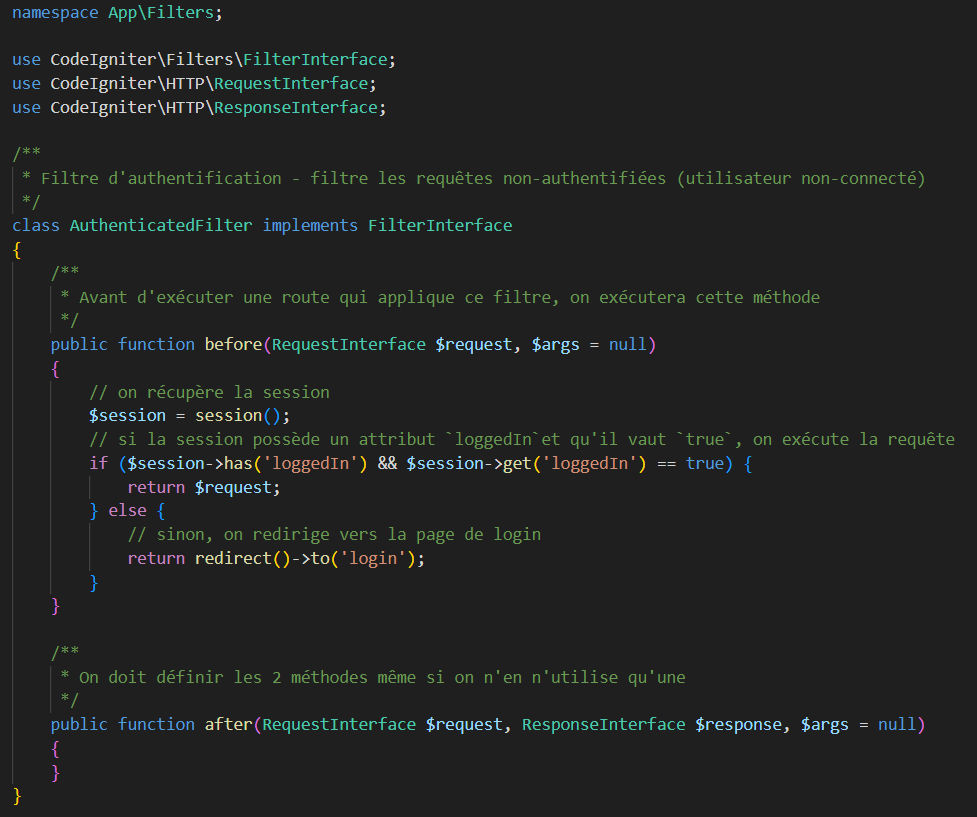
💡 *La méthode est définie avec une* ***visibilité privée****, ce qui empêche un appel par une méthode externe à la classe (comme l’enregistrement dans Routes.php, par exemple), car on ne veut pas que cette méthode soit appelable de depuis l’extérieur de la classe !*

1. On modifie la méthode attemptLogin pour appeler loginUser en cas de connexion réussie :



💡 *On peut voir que la méthode loginUser est appelée une fois avec un paramètre et une fois sans - c’est tout à fait possible car loginUser a été définie avec un paramètre ayant une* ***valeur par défaut*** *(null, en l’occurrence) et qui permet de gérer le fait que l’utilisateur est soit un abonné, soit l’administrateur* 👍

1. On crée un fichier AuthenticatedFilter.php dans le dossier Filters qui va définir notre Filter. On définit la classe correspondante qui implémente l’interface FilterInterface fournie par CodeIgniter :



1. On va modifier la définition de nos routes dans Routes.php afin de définir le filtre AuthenticatedFilter sur les routes que l’on veut protéger :



Ainsi, on précise que les accès aux routes ‘/’ et ‘/home’ doivent être effectués en tant qu’utilisateur connecté, et redirigeront vers ’/login’ le cas échéant !

👉 **Testez vos modifications** en ouvrant votre site en navigation privée, si vous essayez d’accéder à /home sans vous être connecté, vous êtes redirigés

⚡ **Attention : cela signifie que toutes les pages que vous rajouterez dans le fichier Routes.php doivent préciser si elles doivent utiliser le filtre développé ou non !**

💡 *Il est possible d’améliorer le filtre en ajoutant dans la requête de redirection la route que l’utilisateur connecté essayait d’utiliser, et en le redirigeant vers cette route en lieu et place de* /home *dans* Connection::loginUser.

# Views with Data

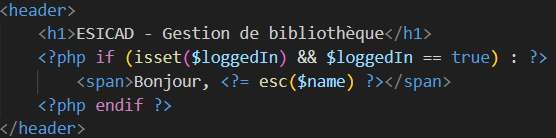
Pour le moment, les vues que nous avons défini n’utilisent pas de données dynamiques (elles affichent le code HTML que nous avons écrit sans injecter de données provenant du controller). Or, nous voulons pouvoir utiliser les données que nous chargeons depuis la base de données dans nos vues (comme vu dans le TP2).

Pour cela, CodeIgniter propose une mécanique qui permet d’injecter des données dynamiques dans les vues, cf. documentation [Adding Dynamic Data to the View](https://www.codeigniter.com/user_guide/outgoing/views.html#adding-dynamic-data-to-the-view)

Il s’agit de fournir en **second paramètre** de l’appel à la fonction view() un **tableau associatif** qui définit des clés et des valeurs associées.

👉 *Nous allons exploiter ce mécanisme pour injecter dans le header de notre site le nom / prénom de notre utilisateur connecté (et s’il s’agit de l’administrateur, nous afficherons seulement “Administrator”).*

1. Modifions notre template header.php pour ajouter des variables dynamiques à afficher :



👉 *Il est possible de faire des conditions et boucles dans les vues - voir syntaxe* [*ici*](https://www.codeigniter.com/user_guide/outgoing/alternative_php.html#alternative-control-structures)

1. Dans les controllers qui vont utiliser la view header, il va falloir injecter le second paramètre $data afin d’injecter les valeurs que la vue doit afficher. Par exemple, dans la classe Home, on va pouvoir écrire :



Lorsque la méthode **index** sera appelée, elle ira chercher dans la session les valeurs des paramètres loggedIn et username, et ira alimenter les variables utilisables dans la vue : 