

Introduction au développement **WEB**

TP4
SisTR

ÉTAPE 050.1 : INTÉGRATION DES MAQUETTES



- > Il est temps pour nous que nos maquettes HTML du sujet 1 servent à quelque chose.
- > Pour cela nous devons créer un ensemble de fichiers qui vont correspondre à la partie index de notre application (page d'accueil).
- > Créer le fichier `/application/templates/index.template.php`
 - ▶ Placer le namespace et la constante de sécurité
 - ▶ Y coller le code de `/maquette-accueil.html`
- > Créer le fichier `/application/views/index.view.php`
 - ▶ Placer le namespace et la constante de sécurité, ce sera tout pour l'instant
- > Créer fichier `/application/controllers/index.controller.php`
 - ▶ Placer le namespace et la constante de sécurité.
 - ▶ Créer la classe `IndexController` (penser à l'héritage)
 - ▶ Créer la méthode `indexAction()`
 - Désigner le template index
 - Désigner la vue index
 - ▶ Ajouter un constructeur pour désigner `indexAction()` comme action par défaut.
- > Se placer dans le fichier `/index.php`
 - ▶ Désigner `IndexController` comme contrôleur par défaut.
- > Tester en ne donnant aucun paramètre à l'URL.

ÉTAPE 050.2 : LA VUE INDEX



- > Nous allons maintenant transférer une partie du contenu du template pour constituer la vue.
- > Se placer dans le fichier `/application/templates/index.template.php`
- > Copier la partie qui va de la navigation (non comprise) au pied de page (non compris). En principe c'est le contenu de la balise `<main>`.
- > Coller ce code dans `/application/views/index.view.php`
- > Effacer la partie copiée du template
- > Inscrire `[%VIEW%]` à la place
- > Tester, rien ne doit avoir changé
- > Si vous avez un doute sur le fait que ça marche, testez en indiquant `vue2` dans `indexAction()`, si la cette vue apparaît dans votre template c'est que vous avez bien "éclaté votre maquette" en vue + modèle. Remettre la vue `index` après ce test.
- > Bizarrement, le formulaire de connexion ne fera pas partie de la vue. Ainsi va le découpage de notre code HTML
- > Il aurait été possible de faire en sorte que les balises du formulaire soient dans la vue et que l'apparence soit dans le coin supérieur gauche. La page d'`index` n'est pas celle que nous allons beaucoup plus travailler.

ÉTAPE 50.3 : TEMPLATE APPLICATION

> Nous allons faire pareil pour la maquette application.



> Créer le fichier `/application/templates/application.template.php`

- ▶ Placer le namespace et constante de sécurité
- ▶ Ouvrir le code source de `/maquette-application.html`
- ▶ Copier tout le code
- ▶ Coller le code dans `/application/templates/application.template.php`

> Créer le fichier `/application/views/sujet-liste.view.php`

- ▶ Placer le namespace et la constante de sécurité

> Créer le fichier `/application/controllers/sujet.controller.php`

- ▶ Créer la classe correspondant au contrôleur sujet (héritage).
- ▶ Ajouter la méthode `listAction()`
 - Sélectionner le template et la vue que nous venons de créer.
- ▶ Créer le constructeur et y régler l'action par défaut sur `listAction()`.

> Tester en entrant l'URL de l'action.

ÉTAPE 50.4 : VUE SUJET

> Nous allons procéder de la même manière que pour la page d'accueil.



> Se placer dans `/application/templates/application.template.php`

> Copier ce qui correspond à la zone de contenu sauf la partie gauche.

> Se placer dans `/application/views/sujet-liste.view.php`

> Coller le contenu

> Revenir au template, supprimer ce qui vient d'être copié et mettre à la place `[%VIEW%]`.

> Tester. Rien ne doit changer.

> Pour être sûr de changer de vue pour voir si ça fonctionne bien (vue2 par exemple).

ÉTAPE 050.5 : CONTRÔLEUR SUIVI



- > Nous allons prochainement mettre en route la navigation principale.
- > Pour que cela soit intéressant nous avons besoin de mettre en route un embryon de contrôleur suivi.

- > Créer le fichier `/application/controllers/suivi.controller.php`
- > Placer le namespace et la constante de sécurité.
- > Créer la classe `SuiviController` (héritage)
- > Créer la méthode `listAction()`
 - ▶ Choisir le template application et la vue `vue2`
- > Créer le constructeur et désigner `listAction()` comme action par défaut.
- > Tester.

- > J'espère que vous vous rendez compte au travers de toutes ces créations de contrôleurs, de template et de vue ce qu'ajouter des pages à notre application correspond à un schéma qui se précise.
- > Je tiens à préciser toutefois quelques points :
 - ▶ NON : il n'y a pas un contrôleur par item du menu principal, c'est une coïncidence.
 - ▶ NON : tous les contrôleurs n'ont pas une action qui s'appelle `listAction()`, c'est encore une coïncidence.
 - ▶ En réalité tout ceci dépend des besoins.
 - ▶ Après ce n'est pas entièrement faux, mais attention de ne pas ériger en principe ce qui est au départ une coïncidence. Tout dépend de ce que fait votre application. De plus ici nous n'avons qu'une navigation à un seul niveau.

ÉTAPE 051.1 : NAVIGATION PRINCIPALE

- > La navigation doit jouer un double jeu :
 - ▶ Permettre d'accéder aux différentes grandes parties de l'application.
 - ▶ Indiquer dans quelle partie on se trouve en la distinguant graphiquement des autres (ça aide).
- > Pour mettre en place la navigation principale :
 - ▶ Nous allons l'extraire et la placer dans un composant spécifique qui sera un Helper. Ce qui la rendra éventuellement intégrable dans plusieurs templates.
 - ▶ La rendre paramétrable de sorte à pouvoir rajouter facilement un autre item de menu.
- > Cela peut vous paraître beaucoup, mais ce c'est qu'un point de départ.
- > Pour commencer, nous allons doter notre classe `Application` de fonctionnalités nécessaires pour la navigation.
- > Se placer dans le fichier `/framework/application.php`
- > Ajouter la méthode `getControllerName()` : getter de `$controllerName`
- > Ajouter la méthode `getActionName()` : getter de `$actionName`
- > Ajouter la méthode `getCurrentLocation()` : le contrôleur et le nom de l'action dans un tableau [`'controller'=>'xxx', 'action'=>'yyy'`]
- > Faudra attendre pour tester.



ÉTAPE 051.2 : CRÉATION DE L'HELPER



- > Créer le fichier `/application/helpers/navigation.helper.php`
- > Placer le namespace et la constante de sécurité
- > Coder la classe dont le diagramme UML est ci-dessous.
- > Se rendre dans le fichier `/application/templates/application.template.php`
- > Copier le code HTML correspondant au menu principal (juste la partie Sujets/Suivis/Utilisateurs) en principe une liste ``.
- > Coller le code dans la méthode de classe `render()` du Helper.
- > Dans le template supprimer le code copié et le remplacer par un appel de la méthode `render()` du helper.
- > Tester, rien ne doit changer Je sais c'est frustrant.

<i>NavigationHelper</i>
+ <u>render()</u>

ÉTAPE 051.3 : PRÉPARER LA LISTE DES MENUS

- > Notre objectif maintenant va être de préparer la liste des menus et de générer automatiquement la liste `/`.
- > Le menu va être représenté par une variable de classe privée (static) `$menu`, c'est un tableau de tableaux qui va décrire chaque item du menu.
Les tableaux imbriqués doivent contenir 3 éléments (tableau indexé par des chaînes) :
 - ▶ `title` : le titre à afficher de l'item
 - ▶ `controller` : le contrôleur à faire figurer dans le lien
 - ▶ `action` : l'action à faire figurer dans le lien
- > Se placer dans le fichier `/application/helpers/navigation.helper.php`
- > Créer tout le contenu du tableau pour représenter le menu.
- > Créer une méthode de classe (static) privée : `itemRenderer($item)`
 - ▶ L'item reçu en paramètre correspond à un des tableaux (`title/controller/action`)
 - ▶ Générer le code d'une balise `` (avec le lien HTML) basée sur l'item reçu en paramètre.
- > Dans la méthode `render()`
 - ▶ Remplacer la série des `` par une boucle sur les éléments de la variable de classe `$menu` qui appelle la méthode de classe `itemRenderer()` pour chacun d'eux.
- > Tester. Le menu doit devenir actif avec des liens fonctionnels.



ÉTAPE 051.4 : LIEN ACTIF



- > Le lien actif (page courante) se traduira par une mise en forme spécifique.
- > Cette mise en forme s'obtiendra en rajoutant la classe 'active' à la balise du lien concerné.
- > Se placer dans `/application/helpers/navigation.helper.php`
- > Dans la méthode `render()`, avant tout autre code :
 - ▶ Récupérer l'instance de l'application.
 - ▶ Récupérer dans une variable le tableau contrôleur/action avec la méthode `getCurrentLocation()` de `Application`.
 - ▶ Transmettre cette donnée à la méthode `itemRenderer()`
- > Ajouter à `itemRenderer()` un paramètre : `$location`
- > Si `$location` correspond au contrôleur du lien à générer ajouter la classe active à la balise .
- > Tester.
- > C'est toujours délicat de générer un attribut conditionnel dans une balise.
- > Personnellement j'utilise une chaîne qui peut être vide ou coder "tout l'attribut" exemple ' `class="active"` '. Et je fait un echo systématique de cette variable.

ÉTAPE 051.5 : PETITE CORRECTION

> Histoire de rendre l'ensemble plus cohérent, il faut changer le template de la gestion des utilisateurs.



> Se placer dans `/application/controllers/utilisateur.controller.php`

> Dans chacune des actions de contrôleur, changer le template pour application.

> Tester.

> Même si un certains nombre d'éléments sont factices, les choses se dessinent petit à petit. Cela fait du bien de voir que nous efforts finissent par payer.

ÉTAPE 052.1 : FORMULAIRE DE LOGIN

> Petits rappels :

- ▶ Un formulaire c'est quoi / à qui / comment.
- ▶ Un formulaire c'est :
 - Un objet Form
 - Un code HTML
 - Une action de contrôleur de gestion

> Commençons par le code HTML.



- > Créer le fichier `/application/forms/html/login.form-html.php`
- > Placer le namespace et la constante de sécurité.
- > Recopier le code HTML du formulaire depuis `/maquette-accueil.html`
- > Remplacer l'attribut `action` de la balise `<form>` par un appel à la méthode `getAction()` (voir le code des autres formulaires).
- > Passer en méthode POST
- > Au niveau des champs, juste corriger l'id/name du mot de passe, remplacer `mot-de-passe` par `motdepasse`.
- > Ajouter la protection CSRF.

ÉTAPE 052.2 : CLASS FORM



- > Créer le fichier `/application/forms/login.form.php`
 - ▶ Placer le namespace et la constante de sécurité
 - ▶ Créer la classe `LoginForm`
 - ▶ Ajouter le constructeur (observer un autre formulaire pour exemple).
 - ▶ Ajouter les champs `login` et `motdepasse` en champs obligatoire sans valeur par défaut.

- > Se placer dans `/application/controllers/index.controller.php`
- > Au niveau de `indexAction()`, créer un objet `LoginForm` et le transmettre à la page (donner l'URL de `indexAction()` comme action).

- > Se placer dans `/application/templates/index.templates.php`
- > Remplacer le code HTML du formulaire par un affichage de celui que nous venons de créer.

- > Tester. C'est fou, une fois de plus rien ne change !

- > Sauf si vous validez le formulaire, l'URL, au lieu de se terminer par `/sistr/#` se termine par `/sistr/?controller=index&action=index`
- > Ouf, on a de quoi vérifier.

ÉTAPE 052.3 : MISE EN ROUTE

> J'avoue que je n'ai pas anticipé le fait d'afficher les messages de validation dans cet espace étriqué et que je ne vois pas comment les placer sans :

- ▶ Détruire votre labeur par des affichages disgracieux.
- ▶ Vous demander de replancher sur le HTML / CSS.
- ▶ Vous fournir une modification casse-gueule à mélanger à votre code.

> Mais j'ai une parade (et je vous assure que je vous la propose en live) !



- > Se placer dans `/application/templates/index.template.php`
- > Ajouter le code suivant au-dessus du rendu du formulaire

```
/application/templates/index.template.php
```

```
01 | <div id="validation-messages">
02 | [%MESSAGES%]
03 | </div>
```

- > Ajouter le code suivant à `/css/sistr.css` histoire de placer les messages "tout en haut"

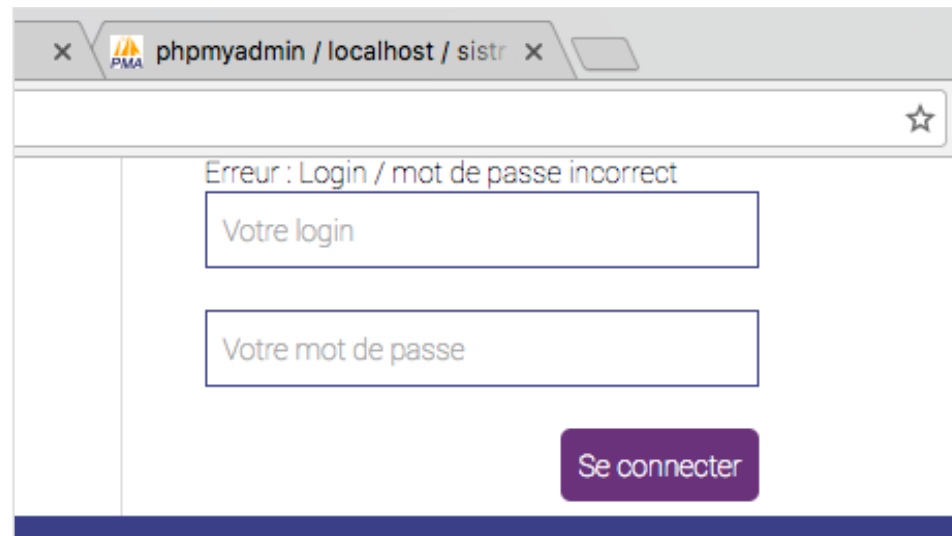
```
/css/sistr.css
```

```
01 | #validation-messages {
02 |     position: absolute;
03 |     top: 0px;
04 | }
```

- > Maintenant nous allons vraiment mettre en route le formulaire.


ÉTAPE 052.4 : MISE EN ROUTE

- > L'astuce que je propose joue du fait que c'est un "micro-formulaire".
 - > Il n'y a que deux champs. On n'est pas obligé de détailler pour chaque champ les erreurs (juste champ manquant).
 - > Du coup on peut se contenter de "Login / mot de passe incorrect" qui passe plutôt bien dans ce cas précis.
-
- > Se placer dans `/application/controllers/index.controller.php`
 - > Mettre en place le cheminement standard d'un formulaire :
 - ▶ Sortie de la méthode si le formulaire n'est pas soumis.
 - ▶ Chargement des données depuis POST.
 - ▶ Sortie avec message "Login / mot de passe incorrect" si le formulaire n'est pas valide.
 - > Tester.
-
- > Le message d'erreur doit apparaître si il manque une donnée (seule détection effectuée jusque-là).



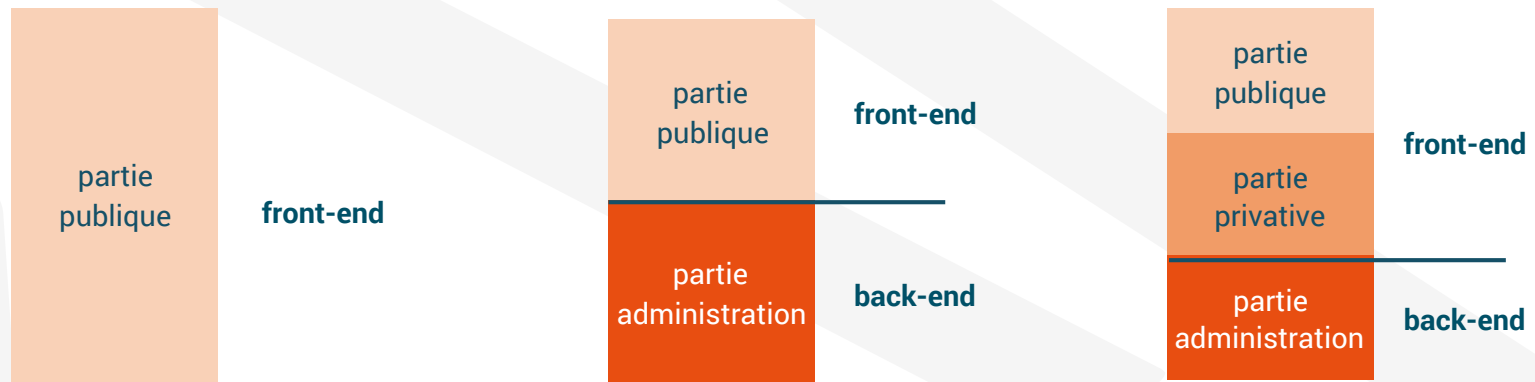
The screenshot shows a web browser window with the address bar displaying "phpmyadmin / localhost / sistr". The page content includes an error message "Erreur : Login / mot de passe incorrect" in red text. Below the message are two input fields: "Votre login" and "Votre mot de passe". At the bottom right of the form is a purple button labeled "Se connecter".

ÉTAPE 052.5 : RÉENTRANCE DES DONNÉES

- > C'est toujours mieux que le formulaire nous restitue les valeurs erronées pour pouvoir plus facilement les corriger.
 - > Sauf pour le mot de passe car c'est difficile dans ***** de savoir qu'on a fait une faute de frappe.
-
- 
- > Se placer dans `/application/forms/html/login.form-html.php`
 - > Remplacer les valeurs des attributs `id` et `name` par des appels de la méthode `fName()`.
 - > Ajouter l'attribut `value` et y afficher la valeur des champs avec `fValue()`.
 - > Tester
-
- > Notre formulaire est quasi actif, il reste à lui rajouter l'authentification pour être pleinement fonctionnel.
 - > Mais ça une histoire un peu plus longue.

ÉTAPE 053.1 : AUTHENTIFICATION

- > Nous avons mis en place une gestion des comptes utilisateurs, maintenant il est temps qu'elle entre véritablement en fonction.
- > La plupart des applications Web que vous utilisez nécessitent une authentification.
- > Comme c'est un élément récurrent des applications Web, cela doit naturellement faire partie du framework.
- > On peut trouver différents types d'utilisation de l'authentification.



- > On distingue principalement deux zones dont les dénominations varient :
 - ▶ front-end : sert plutôt à de la consultation
 - ▶ back-end : sert à administrer l'application Web, il y a nécessité de protéger cette zone par une authentification
- > Front-end et back-end ont des interfaces souvent très différentes.
- > Ensuite, le front-end peut avoir deux sous-parties :
 - ▶ Partie publique, sans authentifications, plus ou moins développée
 - ▶ Partie privative, pour permettre à certains utilisateurs enregistrés de bénéficier de fonctionnalités supplémentaires ou de contenus premium.
- > Généralement, l'authentification seule ne suffit pas, elle va de paire avec des droits utilisateurs ou ACL (Access Control List).
- > Je vous rassure, nous n'irons pas jusque-là avec SisTR, nous nous contenterons de l'authentification.

ÉTAPE 053.1 : DONNÉES UTILISATEURS

- > Classiquement, une authentification c'est un login et un mot de passe.
- > Ça tombe bien, nous avons prévu cela dans notre table Utilisateurs avec un modèle associé.
- > Cela rentre pleinement dans la logique que nous avons mise en place : tout dialogue avec une source de données doit passer par un modèle.
- > Les modèles sont dans la partie application.
- > Or, comme annoncé sur la page précédente, nous devons écrire un composant du framework.
- > Nous nous trouvons dans une situation inverse à celle que nous avons pratiquée jusque-là :
 - ▶ D'ordinaire les composants de l'application utilisent/dépendent des composants du framework.
 - ▶ Ici nous nous trouvons à devoir écrire un composant du framework qui va devoir s'appuyer sur un composant de l'application.
- > Pour être plus pragmatique :
 - ▶ Le composant authentification (du framework), qui va assurer la vérification du combo login/mot de passe, va devoir accéder aux données de la base de données.
 - ▶ Le point critique est de déterminer comment.
 - ▶ Clairement c'est un débat philosophique :
 - Si notre authentification dialogue directement avec la base de données, c'est contraire à notre logique MVC.
 - Si notre authentification dialogue directement avec la base de données, c'est supposer que la source de données de l'application est une base de données, ce n'est qu'un cas de figure possible et pas le seul (on fait beaucoup de webservices aujourd'hui).
 - Pire, si notre authentification a en dur dans son code les noms de table utilisateur et des colonnes utiles, cela reviendrait à dire que toute application développée avec F3iL et utilisant l'authentification devra avoir dans sa base de données cette table précise avec ces colonnes présentes. Cette contrainte est trop forte. Pour moi, une application codée en anglais devrait pouvoir avoir une table "Users" sans avoir à modifier une classe du framework.
- > Ce débat est proche de celui des parties du framework qui génèrent du code HTML. Chaque fois, nous avons fait en sorte qu'il soit possible de customiser ce code HTML par un composant intermédiaire.


ÉTAPE 053.1 : COMPOSANT INTERMÉDIAIRE

- > L'idée générale de ce débat est de vous montrer qu'on a tout intérêt à passer par un composant intermédiaire :
- > Je me doute que ce débat vous semble obscure. Je vais essayer de prendre un exemple plus concret.
- > Qu'est-ce qui interdit à une vue ou un contrôleur d'accéder directement à la base de données sans passer par un modèle ? Rien; si ce ne sont mes directives.
- > Absolument rien.
- > Alors pourquoi ne le fait-on pas ?
- > Parce que c'est mieux : le modèle joue le rôle de composant intermédiaire entre une source de données et un composant qui les exploite.
- > Entrons dans la fiction, demain j'arrive et de vous dit "la base de données c'est has-been, pour les utilisateurs on va travailler avec des fichiers XML". OK partons là-dessus (je suis le chef).
- > Parce que nous utilisons un composant intermédiaire, à condition de respecter les méthodes actuellement en place, nous pouvons passer de la base de données aux fichiers XML de façon transparente pour le reste de l'application.
- > Cela s'appelle de découplage.
- > Les contrôleurs ne savent pas qu'ils dialoguent avec une base de données, ils dialoguent avec un modèle qui leur propose une série d'opérations disponibles.
- > On peut substituer au modèle actuel, n'importe quelle classe qui aurait la même série d'opération.
- > Ce ne serait pas aussi simple si n'importe quel contrôleur ou n'importe quelle vue accédaient directement à la base de données.
- > C'est exactement ce que nous allons faire avec l'authentification : nous allons utiliser un composant intermédiaire pour dialoguer avec la source de données contenant les informations d'authentification.
- > Et ce composant sera un modèle "labélisé" fournisseur d'authentification.

ÉTAPE 053.1 : HÉRITAGE VS INTERFACE

- > Notre composant authentification va donc avoir besoin que le modèle sur lequel il s'appuie dispose de certaines fonctionnalités.
- > En POO le moyen que l'on utilise le plus souvent c'est l'héritage. C'est possible dans notre cas.
- > J'avoue quand je dis "c'est possible", il y a souvent un "attendez j'ai un truc à vous vendre".
- > L'héritage met en place une relation "est un".
 - ▶ Un chien est un mammifère et un caniche est un chien.
 - ▶ Dans cet exemple mammifère est la classe mère, dont chien hérite et caniche hérite de chien.
- > Il existe des cas où cette relation "est un" est trop forte par rapport à la réalité des choses, voir contraignante.
- > Il existe d'autres cas où une relation "se comporte comme" (a les mêmes fonctionnalités) serait suffisante. Ceci correspond à la notion d'interface.
- > En POO, une interface est une ensemble de méthodes abstraites (sans code donc) regroupée et dont on peut imposer à une classe d'en disposer. On parle souvent de protocole.
- > C'est précisément le "truc que j'avais à vous vendre" et que nous allons utiliser.
- > Nous allons ainsi construire notre mécanisme d'authentification en supposant l'existence d'un objet d'une classe dont on ne se soucie pas du nom mais à laquelle on imposera par une interface de disposer de méthodes précises.
- > C'est un mécanisme de délégation.
- > Notre classe modèle UtilisateursModel aura la responsabilité par délégation d'une partie du traitement.
- > Nous sommes dans des mécanismes proches de ce que l'on appelle l'injection de dépendance.
- > Assez de théorie passons à la pratique.

ÉTAPE 053.1 : MODIFICATION AUTOLOADER

- > Nous allons introduire dans notre système une interface.
 - > Or il se trouve que notre autoloader ne gère pour l'instant que des classes, ce qu'il fait plutôt bien.
 - > Même si une interface ressemble à une classe, la fonction `class_exists()` présente dans l'autoloader répondra toujours faux pour une interface. Pour mémoire, `class_exists()` fait comme son nom l'indique, elle vérifie que ce que la classe que l'on cherche est bien définie.
 - > Il existe aussi `interface_exists()` qui permet de tester si une interface est définie.
- 
- > Se rendre `/framework/autoloader.php` au niveau de `Autoloader::checkAndRequire()`
 - > Corriger le dernier `if()` pour ajouter un test avec `interface_exists()`.
 - > Tester. Si votre programme ne fonctionne plus après cette infime modification, c'est que le test que vous avez mis en place n'est pas le bon.
- > Nous allons très prochainement vérifier si cela fonctionne.

ÉTAPE 053.2 : INTERFACE AUTHENTICATION



- > Afin donc de décrire quelles méthodes le modèle qui sera délégué à l'authentification, nous allons utiliser une interface.
- > Je vous la fournis.
- > Créer le dossier /framework/interfaces
- > Mettre un fichier index.html de sécurité (copie)
- > Se rendre sur TOUSCOM dans le dossier correspondant à l'étape.
- > Copier le fichier.
- > Coller le fichier dans le dossier que nous venons de créer.
- > Il n'y a rien à tester, juste à regarder.
- > Petit rappel : dans une interface, nous ne pouvons que déclarer des méthodes abstraites.
- > Ici je les ai préfixées par "auth_", histoire de dans le code du modèle, on puisse visuellement savoir quelles sont les méthodes propres au modèle, celles qui existent déjà, et celle qui arrivent par le biais de l'interface.
- > Trois de ces méthodes se terminent par le mot Key. Ces trois méthodes ne sont là que pour savoir où chercher les informations dans un tableau PHP représentant les données d'un utilisateur.
- > Dans notre cas ce tableau PHP correspondra à une ligne de la base de données, mais encore une fois, ce n'est en rien une obligation. Du moment que ces données soient manipulables sous la forme d'un tableau PHP.
- > Les deux autres méthodes présentes pour l'instant sont justement des méthodes pour récupérer les données d'un utilisateur soit par rapport à son login, soit par rapport à son identifiant.

ÉTAPE 053.3 : IMPLÉMENTATION

> Nous allons maintenant mettre en route cette interface dans le modèle `UtilisateursModel`.



> Se placer dans le fichier `/application/models/utilisateurs.model.php`

> Ajouter sur la ligne de déclaration de la classe implements `\F3il\AuthenticationInterface`

> À partir de maintenant, la classe `UtilisateursModel` est tenue de définir les méthodes de l'interface.

> Pour ceux qui sont sous NetBeans, le nom de la classe est souligné en rouge pour indiquer ce problème (faire alt-entrée dessus pour que NetBeans les génère automatiquement).

> Pour les autres, il suffit de consulter dans le navigateur `UtilisateurController::listAction()` pour voir les dégâts.



> Coder les 3 getters de clés. Ils doivent juste retourner une chaîne de caractères, qui, dans notre cas, doivent correspondre aux noms des colonnes de notre table utilisateurs. Cette partie de l'étape nécessite une grande perspicacité et précision (humour).

> Pour la méthode `auth_getUserById()`, il suffit de retourner un appel de `lire()` qui est déjà présente.

> Pour `auth_getUserByLogin()`, c'est une méthode très similaire à `lire()`, sauf qu'on travaille avec le login au lieu de l'id. Quand on a dit ça, on a tout dit.

> Pour tester, pour tout le monde cette fois, il faut consulter `UtilisateurController::listAction()`.

> Si une Erreur comme quoi la classe `AuthenticationInterface` n'est pas trouvée, c'est qu'il y a soit une faute de frappe dans le nom, soit une erreur dans l'autoloader. À résoudre d'urgence.

> Nous n'avons fait que la première brique : la mise en place du délégué, maintenant il nous faut mettre en place celui qui délègue.

ÉTAPE 054.1 : AUTHENTICATION



- > Authentication (sans le mot interface cette fois), sera la classe du framework qui va gérer l'authentification.
- > C'est un singleton.
- > Pour ce premier jet, je vous donne la base de la classe, avec le PHPDoc, reste le code des méthodes à écrire.

- > Se rendre sur TOUSCOM dans le dossier correspondant à cette étape.
- > Copier le fichier authentication.php dans le dossier /framework
- > Pour le constructeur :
 - ▶ Affecter la propriété `$authenticationModel` avec le modèle passé en paramètre.
 - ▶ Appeler sur le modèle les méthodes de récupération de clés pour affecter les propriétés correspondantes (`$idKey`, `$loginKey` et `$passwordKey`).
- > Pour la méthode `getInstance()` :
 - ▶ Mettre en place le code de récupération d'instance d'un singleton. Celui de `Application` est un bon exemple.
 - ▶ Ajouter avant l'appel de `new` un test : lancer une erreur si `$model` vaut `null`. Ce test permet de s'assurer qu'au premier de `getInstance()` un objet d'une classe respectant l'interface soit fourni en paramètre.
- > Pour `checkAuthenticationKeys()`, le but est de s'assurer qu'on a bien un tableau qui contient les informations nécessaire pour l'authentification :
 - ▶ Retourner faux si (faire autant de `if()`):
 - `$data` n'est pas un tableau (`is_array()`).
 - `$data` ne contient pas un emplacement pour l'identifiant (dont la clé est la propriété `$idKey`).
 - `$data` ne contient pas un emplacement pour le login.
 - `$data` ne contient pas un emplacement pour le mot de passe.
 - ▶ Retourner vrai (parce qu'on a pas retourné faux avant).

ÉTAPE 054.1 : AUTHENTICATION



> Toujours dans /framework/authentication.php

> Pour la méthode login() :

▶ Voici le code :

/framework/authentication.php [Extrait]

```
01 public function login($login,$password) {  
02     $this->user = $this->authenticationModel->auth_getUserByLogin($login);  
03     if(!$this->user) return false;  
04  
05     if(!$this->checkAuthenticationKeys($this->user)){  
06         throw new Error('Modèle Authentification');  
07     }  
08  
09     if($this->user[$this->passwordKey]!=$password) return false;  
10  
11     return true;  
12 }
```

> Explications :

▶ Ligne 2 : on demande au modèle délégué de nous fournir les informations de l'utilisateur par son login.

▶ Ligne 3 : si aucun utilisateur ne correspond au login (false), l'authentification échoue.

▶ Ligne 5 : vérification que ce qui est retourné contient les informations attendues, sinon il y a tromperie sur la marchandise, c'est une erreur de codage, donc Error.

▶ Ligne 9 : si le mot de passe entré dans le formulaire de login (reçu en paramètre) et celui qui est en base sont différents, l'authentification échoue.

▶ Ligne 11 : (sinon implicite) tout semble OK, donc l'authentification réussit.

> Oui, il y aura des choses à dire par rapport aux mots de passe, mais patience.

ÉTAPE 054.2 : MISE EN ROUTE



- > Se placer dans le fichier `/framework/application.php`
- > Ajouter une méthode `setAuthenticationDelegate($className)`
 - ▶ Information : `$className` désigne forcément une classe de l'application et non du framework. Pour désigner `\Sistr\UtilisateursModels` nous recevrons la chaîne `"UtilisateursModel"`
 - ▶ Dans une variable, compléter `$className` par le namespace de l'application.
 - ▶ Créer un objet de cette classe et le passer en paramètre de `Authentication::getInstance()`.
- > Se placer dans `/index.php` avant l'appel de la méthode `run()`
- > Appeler la méthode `setAuthenticationDelegate()` en indiquant `"UtilisateursModel"` en paramètre.
- > Tester.
- > Nous avons une partie de l'authentification qui est opérationnelle. Nous pouvons aller jusqu'à la vérification du login/mot de passe (en clair).

ÉTAPE 055.1 : UTILISATION



> Nous pouvons avancer dans la gestion du formulaire d'authentification.

> Se placer dans le fichier `/application/controllers/index.controller.php` à la fin de `IndexController::indexAction()`

> Ajouter le code de l'algorithme suivant :

```
01 Récupérer l'instance de Authentication
02 Si(Non(Authentication réussie(login, mot de passe)){
03     Ajouter un message d'erreur ('Login / Mot de passe incorrect')
04     Sortir
05 }
06 Afficher "Authentication réussie"
07 Mettre fin au programme
```

> Tester avec :

- ▶ Login / Mot de passe corrects : le programme doit se terminer sur "Authentication réussie"
- ▶ Login incorrect / mot de passe correct : erreur
- ▶ Login correct / mot de passe incorrect : erreur
- ▶ Login vide / mot de passe correct : erreur
- ▶ Login correct / mot de passe vide : erreur
- ▶ Login et mot de passe vides : erreur

> Oui, j'avais envie d'une étape en une seule page. Ne cherchez pas de 55.2 il n'y en a jamais eu !

**NE JAMAIS
STOCKER DE
MOT DE PASSE**

ÉTAPE 056.1 : WHAT'S THE PROBLEM ?

- > Stocker un mot de passe est purement et simplement le top du top de la dangerosité.
- > J'ai une question simple : sur combien de sites utilisez-vous le même mot de passe ?
- > Vous avez confiance en ces sites ?
- > Si l'un d'entre eux se faisait hacker, genre vol de données clients (cela arrive) avec votre mot de passe. Ça ne serait pas terrible pour vous : il suffirait au hacker de vous pister sur la toile (pas trop dur) pour avoir accès à votre vie numérique. C'est moche.
- > Ça c'est la version client.
- > La version développeur (vous, dans un futur proche) est moins drôle, si en entreprise vous êtes celui qui a eu la bonne idée d'enregistrer des mots de passes en base de données, et que l'entreprise s'est fait dérober, je pense que vous êtes mal. C'est une faute grave. Je précise que je ne fais pas d'humour en disant cela.

- > OK on ne stockera jamais de mot de passe.
- > Mais que stocker alors ? Un mot de passe crypté ?
- > PERDU !!!!
- > En clair ou crypté cela revient strictement au même. Ce qui est crypté peut être décrypté (plus ou moins facilement). C'est donc pratiquement aussi dangereux.

- > Ok on ne stockera jamais de mot de passe, ni de mot de passe crypté.
- > Mais que stocker alors ?
- > Les sites stockent une empreinte du mot de passe.
- > Une quoi ?
- > Une empreinte.

ÉTAPE 056.1 : L'EMPREINTE

- > Une empreinte de mot de passe est une chaîne de caractères obtenue à partir du mot de passe via une fonction de hashage **NON RÉVERSIBLE**.
- > Le caractère non réversible est important, pour ne pas dire primordial.
- > C'est-à-dire qu'à partir de l'empreinte on ne peut pas retrouver le mot de passe qui a servi à la générer.
- > Au moment où un compte est créé, on génère l'empreinte du mot de passe, et cette empreinte est stockée en base de données.
- > Au moment de la connexion, on génère l'empreinte du mot de passe saisi, et on compare cette empreinte avec celle stockée dans la base de données.
- > Si les deux empreintes sont identiques, c'est que c'est le même mot de passe a été tapé les deux fois.
- > Bon ça c'est la théorie. (Aïe, quand je dis ça, généralement, la suite se complique).
- > Il y a quelques années parmi les fonctions hashage utilisées pour les mots de passe, on trouvait le MD5 ou SHA-1. Ces fonctions réalisent un calcul à partir d'une chaîne de caractères de taille quelconque et retournent une chaîne de respectivement 32 ou 40 caractères alphanumériques

Source	MD5	SHA-1
a	0cc175b9c0f1b6a831c399e269772661	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
abc	900150983cd24fb0d6963f7d28e17f72	a9993e364706816aba3e25717850c26c9cd0d89d
aBc	dbbbbe4975e026e04a687871f296a2b2	a792075841df16e50563ef4d3aab7424604a505b
une phrase un peu plus longue	3f3f187091f9d3d09649d0ff5cc23e7c	63e8946fd7a86e475e50bdb3a681b9b6847ac9ba
fichier de 3.102.231 octets (3,1 Mo)	ea8cdef04ace6620ae3c162eb0f7d666	c9f280da5afb5dc0a88209b52578cce91a3ced4e

- > On voit bien sur ces exemples qu'une chaîne source de 1 caractère ou un fichier de 3.1 Mo, le résultat fait exactement la même longueur.
- > Cela a une conséquence : il a nécessairement des collisions, soit deux chaînes sources différentes qui ont la même empreinte.

ÉTAPE 056.1 : FONCTIONS DE HASHAGE

- > Aujourd'hui utiliser les empreintes MD5 ou SHA-1 sans complément est considéré comme dangereux !
- > Alors pourquoi vous parler d'un truc dangereux ?
- > Parce qu'on trouve sur le Web des archives qui datent un peu et qui datent d'un temps où ces fonctions étaient suffisantes.
- > Pourquoi MD5 ou SHA-1 sans complément c'est dangereux ?
- > Des petits malins s'amuse à cumuler des empreintes MD5/SHA-1 et les mots de passe qui les ont générés dans des bases de données.
- > Ils ont commencé par passer à la moulinette tous les dictionnaires de la planète et les mots de passes triviaux comme "azerty".
- > Miser sur la fainéantise de l'être humain est toujours un pari gagnant.
- > Question indiscrete : prenez-vous du plaisir à choisir un mot de passe tel qu'on vous indique de le faire ? Vous savez un mot de passe avec des caractères spéciaux et des chiffres et des majuscules et des minuscules et des caractères qu'on sait d'où qui sortent ni à quoi ils servent en dehors des mots de passes qu'on sait même pas leur nom tellement on ne les connaît pas. C'est vachement plus simple à mémoriser que "loulou94", non ? (ironie).
- > Ces bases de données des mots de passes triviaux sont nommées les Rainbow Tables.
- > Évidemment, c'est disponible sur la toile.
- > Imaginons que mon mot de passe a pour MD5 "9612fe167e052bb7292202ccb7ce9fd1", faites un petit tour sur crackstation.net et entrez la clé. Bon en même temps, ce mot de passe était prévisible.
- > Les mots de passe en clair : vous oubliez.
- > Les mots de passe en MD5 ou SHA-1 : vous oubliez.

ÉTAPE 056.1 : LE SALAGE

- > Aujourd'hui une technique simple et qui n'est pas encore frappée d'obsolescence ne nomme le salage.
- > La technique peut reposer sur les fonctions MD5 ou SHA-1, même si on préconiser de passer à de meilleures fonctions.
- > Le principe est le suivant :
- > Au moment de la création du compte, et AVANT de générer l'empreinte, on fait la concaténation du mot de passe avec une autre chaîne de caractères que l'on nomme le sel.
- > Au moment de la connexion, on réalise la même concaténation (avec le même sel donc) avant génération de l'empreinte.
- > On vérifie toujours que les deux empreintes coïncident parfaitement.
- > Il se dit que même en disposant de l'empreinte et du sel, il est impossible de retrouver le mot de passe. C'est ce qui est le plus important.
- > Par rapport à mon mot de passe préféré de tout à l'heure, si j'utilise un sel même aussi trivial que "azerty", j'obtiens comme MD5 :
04c84a712afe8debc59aa7b172cd0cee et crackstation vous répondra qu'il ne trouve pas.
- > Je vous assure que mon mot de passe n'a pas changé !
- > Précision : cela ne change rien, il vaut toujours mieux avoir un mot de passe non trivial (caractères spéciaux,...), avoir des mots de passes différents pour chaque site.
- > Le plus paranoïaques utilisent un sel différent pour chaque mot de passe, et d'autres techniques comme plusieurs passages dans le générateur d'empreintes pour "s'éloigner" encore plus du mot de passe original.
- > Je me permets cette remarque : aux yeux de ce qui vient d'être vu, si quand vous avez perdu votre mot de passe, un site vous renvoie effectivement votre mot de passe, c'est que c'est un site dangereux !
- > Un bon site vous renverra un jeton avec une durée de vie très courte pour entrer un nouveau mot de passe à la place de l'ancien, mais jamais il ne doit vous restituer votre mot de passe (au pire vous en envoyer un nouveau).

ÉTAPE 056.1 : MOT DE PASSE SALÉ

- > Nous allons utiliser l'algorithme SHA256 comme base de hashage et comme sel, nous utiliserons l'empreinte SHA256 de l'heure de création du compte.



- > Se placer dans le fichier /framework/authentication.php
- > Ajouter la méthode hash() dont le code vous est donné ci-dessous.

/framework/authentication.php

```
01 /**
02  * Méthode d'encodage du mot de passe
03  *
04  * @param string $password : mot de passe
05  * @param string $salt : sel
06  * @return string : mot de passe encodé
07  */
08 public function hash($password,$salt) {
09     return hash('sha256',hash('sha256',$salt).$password);
10 }
```

- > Nous devons encoder les mots de passe des utilisateurs que nous avons créés jusqu'à présent.
- > Je vous ai préparé un fichier pour vous aider.



- > Se rendre sur TOUSCOM dans le dossier correspondant à cette étape.
- > Copier le fichier dans /temp
- > Consulter /temp/hash.php : vous devez voir l'empreinte des mots de passe de vos utilisateurs.
- > Recopier les empreintes obtenues dans la base de données (colonne motdepasse) avec phpMyAdmin.

ÉTAPE 056.2 : COMPARAISON

> Vu que nous nous basons sur des données stockées en base pour le sel, il faut "le faire remonter" jusqu'à l'authentification.

> Pour cela nous devons :

- ▶ Modifier l'interface, pour que cela fasse partie du mécanisme "standard"
- ▶ Modifier le modèle pour se conformer à cette nouvelle version de l'interface.



> Se placer dans le fichier `/framework/interfaces/authentication.delegate.php`

- ▶ Ajouter la méthode `auth_getSalt($user)` (public), pour le PHPDoc, cette fonction retourner une chaîne.

> Se placer dans le fichier `/application/models/utilisateurs.model.php`

- ▶ Ajouter la méthode `auth_getSalt($user)` (public)
- ▶ Retourner la date de création.

> Il nous faut maintenant adapter notre façon de vérifier la concordance de mot de passe.



> Se placer dans le fichier `/framework/authentication.php`

- > Corriger le test de comparaison de mot de passe de façon à comparer l'empreinte stockée en base de données avec l'empreinte de celui obtenu à la connexion.
- > Tester avec les différentes combinaisons.

ÉTAPE 056.3 : GESTION DES COMPTES

> Maintenant que notre vérification du mot de passe se base sur un mot de passe salé avec l'utilisation du SHA-256, il faut adapter notre création de compte afin que tous les nouveaux utilisateurs puissent se connecter.



> Se placer dans `/application/models/utilisateurs.model.php`

▶ Se placer au niveau de `UtilisateursModel::creer()`

○ Faire en sorte que la récupération de la date et l'heure passe par une variable (je l'avais fait instinctivement dans mon code, je ne sais pas si c'est votre cas). L'intérêt de passer par une variable c'est pour ne pas dépendre de deux demandes distinctes qui pourraient renvoyer une heure différente.

○ Récupérer l'instance de la classe `Authentication`.

○ Dans une variable stocker le mot de passe "hashé" en utilisant l'heure comme sel.

○ Corriger le `bindValue()` pour utiliser l'empreinte que nous venons de générer.

▶ Se placer au niveau de `UtilisateursModel::mettreAJour()`

○ Faire en sorte que l'enregistrement d'un nouveau mot de passe transite par la génération de son empreinte basée *ATTENTION* sur l'heure de création du compte (celle qui est déjà dans la base de données).

> Tester en créant un compte et essayer de se connecter puis en modifiant le mot de passe d'un compte.

ÉTAPE 057.1 : PARTIES PUBLIQUES/PRIVÉES

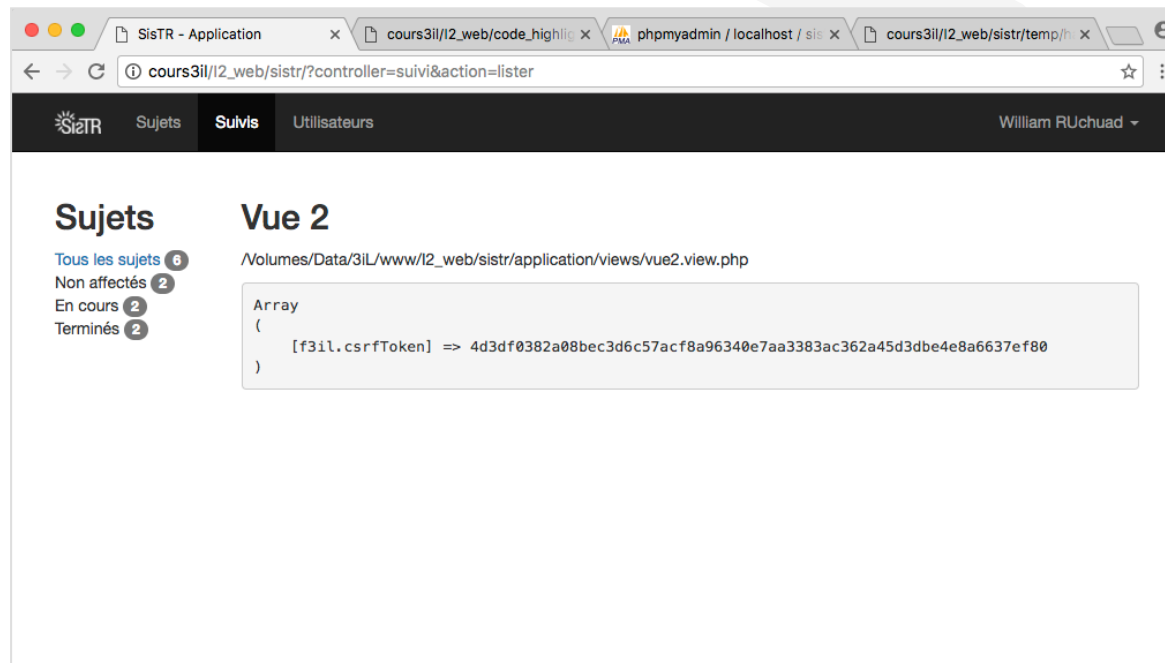
- > Nous avons une authentification fonctionnelle : elle permet de vérifier si un couple login/mot de passe correspond à un utilisateur figurant dans notre base de données.
- > En l'état, notre application Web expose toutes ses fonctionnalités à tout va. Il est temps de faire en sorte qu'il y ait une partie accessible sans être connecté, et le reste uniquement en étant connecté, sans quoi notre authentification ne serait pas vraiment crédible.
- > Pour faire simple : le contrôleur index sera d'accès libre.
- > Tous les autres contrôleurs seront d'accès protégé par le mot de passe.

- > Une fois connecté, nous aurons besoin que l'identification de l'utilisateur soit conservé jusqu'à sa déconnexion, nous utiliserons les sessions PHP pour cela (la clé du tableau sera la constante `SESSION_KEY`).
- > Nous ne stockerons dans la sessions que l'id de l'utilisateur connecté et pas plus. Par contre, notre classe Authentication chargera les données de l'utilisateur automatiquement. Cette distinction peut vous paraître subtile, mais il vaut mieux ne pas laisser trainer des données sensibles dans les sessions PHP. L'id de l'utilisateur connecté est donc largement suffisant.

- > Aussi bizarre que cela puisse paraître nous allons procéder à l'envers.
- > Je m'explique : manipuler les sessions / connexion peut entrainer des situations curieuses (mais pas bloquantes en principe).
- > En procédant à l'envers, donc, c'est-à-dire par les derniers éléments que l'on s'attendrait à mettre en place nous allons nous assurer de disposer de quoi garder la maîtrise de la situation, en tout cas je l'espère, vous êtes souvent surprenants.

ÉTAPE 057.1 : LA SESSION

- > Comme nous avons besoin de toucher aux sessions PHP pour finaliser notre authentification, autant se doter d'un moyen de voir facilement ce qu'il y a dedans.
- > Pour cela, nous allons mettre à profit un contrôleur et une vue qui ne nous servent pas encore : SuiviController et la vue vue2 (qui est juste une vue de test).
- > Se placer dans `/application/controllers/suivi.controller.php` au niveau de `SuiviController::listerAction()`
 - ▶ Faire en sorte que l'action choisisse le template application et la vue vue2. (Il y a de grande chance que cela soit déjà le cas).
- > Se placer dans le fichier `/application/views/vue2.view.php`
 - ▶ Ajouter dans une balise `<pre>` l'affichage avec `print_r()` du tableau `$_SESSION`.
- > Tester en consultant dans le navigateur l'action que nous venons de modifier.



ÉTAPE 057.2 : SE DÉCONNECTER

> Nous allons maintenant mettre en place de quoi se déconnecter. Pour cela il va falloir :

- ▶ Ajouter la méthode `Authentication::logout()`
- ▶ Ajouter l'action `UtilisateurController::deconnecter()`
- ▶ Ajouter un lien vers cette action dans le template application.



> Se placer dans `/framework/authentication.php`

- ▶ Ajouter la méthode `logout()` (public)
 - Affecter `null` à la propriété `$user` : ceci effacera les données de l'utilisateur dans l'application.
 - Avec `unset()` supprimer dans le tableau `$_SESSION` l'emplacement `SESSION_KEY` : pour effacer l'id utilisateur de la session.

> Se placer dans `/application/controllers/utilisateur.controller.php`

- ▶ Ajouter la méthode `deconnecterAction()` (public)
 - Récupérer l'instance d'`Authentication`.
 - Exécuter la méthode `logout()`.
 - Rediriger le navigateur vers `SuiviController::listerAction()` (c'est provisoire).

> Se placer dans `/application/templates/application.template.php`

- ▶ Mettre à jour le lien du menu "Déconnexion" pour désigner `UtilisateurController::deconnecterAction()`.

> Pour l'instant on peut tester le lien déconnexion : en allant sur la liste des utilisateurs, le menu déconnexion doit nous ramener sur `SuiviController::listerAction()`.

ÉTAPE 057.3 : FINIR LA CONNEXION



- > Pour terminer la connexion proprement dite, il faut stocker dans la session l'id de l'utilisateur connecté et rediriger après connexion vers un contrôleur de la partie privée.
- > Se placer dans `/framework/authentication.php` au niveau de `Authentication::login()`
 - ▶ À la fin de la méthode, **AVANT** de retourner `true`, stocker dans `$_SESSION`, à la clé `SESSION_KEY` l'id de l'utilisateur. Indice, toutes les données de l'utilisateur sont accessibles dans la propriété `$user`.
 - ▶ **ATTENTION**, très important : pour récupérer l'id de l'utilisateur dans `$user`, il ne faut pas écrire `'id'` mais passer par la propriété `$idKey` que nous avons justement initialisée avec les informations fournies par le modèle.
- > Se placer dans `/application/controllers/index.controller.php` au niveau de `IndexController::listAction()`
 - ▶ Supprimer le `die()` de fin de méthode.
 - ▶ Ajouter la redirection du navigateur vers `SuiviController::listAction()` (autant se faciliter la vie).
- > Tester.
 - ▶ Quand un utilisateur est connecté vous devez avoir son id dans `$_SESSION['f3il.authentication']`
 - ▶ Quand vous vous déconnectez, cet id doit disparaître.

ÉTAPE 058.1 : PARTIES PUBLIQUE / PRIVÉE

- > Pour que notre authentification tienne définitivement la route, nous devons faire en sorte que l'accès aux contrôleurs `UtilisateurController`, `SuiviController` et `SujetController` ne soient pas possible sans être authentifié.
- > De manière inverse, nous allons empêcher l'accès à `IndexController` en étant authentifié.
- > Ces "empêchements d'accès" se feront pas le biais de redirection du navigateur.
- > Se placer dans `/framework/authentication.php`
 - ▶ Ajouter la méthode `isLoggedIn()` (public)
 - Cette méthode renvoi un booléen qui indique si dans `$_SESSION` un emplacement existe pour la clé `SESSION_KEY` (tester avec `isset()`).
- > Se placer dans `/framework/controller.php`
 - ▶ Ajouter la méthode `redirectIfAuthenticated($redirect)` (public)
 - Cette méthode effectue une redirection vers `$redirect` si un utilisateur est connecté (test avec la méthode que nous venons d'ajouter à `Authentication`).
 - ▶ Ajouter la méthode `redirectIfUnauthenticated($redirect)` (public)
 - Fait la même chose que la méthode précédente mais si un utilisateur n'est pas connecté.
- > Pour l'instant il n'y a rien à tester sauf que tout doit continuer à fonctionner comme d'ordinaire.



ÉTAPE 058.2 : REDIRIGEONS !!



- > Il nous reste maintenant à utiliser ces méthodes de redirection.
- > Se placer dans le constructeur de `UtilisateurController`
 - ▶ En première ligne, utiliser la méthode `redirectIfUnauthenticated()` pour renvoyer vers `IndexController::indexAction()`.
- > Faire de même pour `SuiviController` et `SujetController`.
- > Se placer dans le constructeur de `IndexController`
 - ▶ En première ligne, utiliser la méthode `redirectIfAuthenticated()` pour renvoyer vers `SuiviController::listAction()`
- > Pour tester, il va falloir entrer des URLs dans la barre d'adresse :
 - ▶ Quand on n'est pas connecté, il faut tester `UtilisateurController::listAction()` (ou n'importe quelle autre action).
 - ▶ Quand on est connecté, il faut tester `IndexController::indexAction()`.
- > Nous avons traité le problème "globalement" au sein de nos contrôleurs, ce n'est pas une obligation. On pourrait dans un même contrôleur avoir des méthodes nécessitant d'être connecté ou pas. Dans ce cas, il faudrait mettre les tests dans nos actions de contrôleur et surtout pas dans le constructeur.

ÉTAPE 058.3 : LES FINISSIONS

> Nous en avons presque fini avec la mise en place de l'authentification !

> Il nous reste :

- ▶ À charger automatiquement les données de l'utilisateur connecté à la construction de Authentication.
- ▶ Permettre la récupération de ces données.
- ▶ Permettre la récupération de l'id de l'utilisateur connecté.
- ▶ Utiliser tout ça dans notre template application pour afficher le bon nom en haut à droite.

> Se placer dans `/framework/authentication.php` au niveau de `Authentication::__construct()`, à la fin de la méthode.

- ▶ Si un utilisateur est authentifié, charger ses données dans la propriété `$user` à partir du modèle délégué l'utilisateur connecté. Pour cela il faut utiliser :
 - `Authentication::isLoggedIn()`
 - `AuthenticationInterface::getUserById()`
- ▶ Effacer le mot de passe pour éviter qu'il traine en mémoire, simple précaution (utiliser `unset()`).

> Ajouter les méthodes :

- ▶ `getLoggedUser()` : getter pour la propriété `$user`
- ▶ `getLoggedUserId()` : getter pour l'id de l'utilisateur à lire dans `$_SESSION`.
- ▶ Pour ces deux méthodes, précéder le retour d'un test avec `isLoggedIn()` avec une `Error` si aucun utilisateur n'est connecté.



ÉTAPE 058.4 : UTILISATEUR CONNECTÉ

> Maintenant nous devons faire en sorte que le nom qui apparaît en haut à droite corresponde à l'utilisateur connecté.



> Se placer dans `/application/templates/application.template.php`,

> En PHP tout en haut du fichier, avant le `<!DOCTYPE>` :

▶ Récupérer l'instance d'authentification.

▶ Récupérer dans une variable l'utilisateur authentifié (regarder sur la page précédente pour la méthode à utiliser).

> Dans le menu, à l'emplacement du nom, afficher le prénom et nom de l'utilisateur.

> Tester avec différents utilisateurs connectés.

> Je ne peux m'empêcher de me poser la question si c'est bien au template de faire la récupération de l'utilisateur, puisqu'il est supposé ne faire que de l'affichage de données récupérées par une action de contrôleur. Il s'agit donc d'une entorse à ce principe.

> Si je ne l'avais pas fait ainsi, je l'aurais fait par un Helper qui nous aurait fourni le nom à afficher, car il est inconcevable que chaque action de contrôleur se voit confier la tâche de récupérer l'utilisateur connecté sur tous nos contrôleurs. Pas très DRY.

> Cette fois nous avons fini avec l'authentification, sauf oubli de ma part.

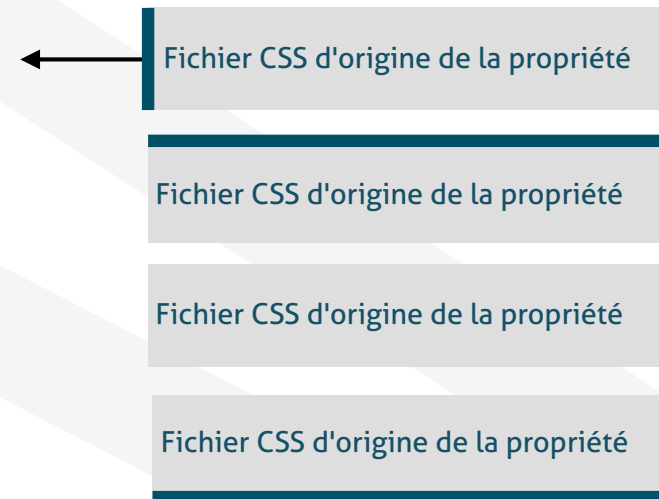


> Un dernier "truc" à faire : mettre les commentaires PHPDoc de toutes les méthodes que nous avons créées dans le framework ! Je ne vous l'ai pas dit, cela ne voulait pas dire qu'il ne fallait pas le faire ! (`authentication.php`, `controller.php`).

> Petit commentaire : cette authentification me paraît trop simple pour être honnête. Elle n'est pas sécurisée, c'est certain. Même si elle n'est pas parfaite, elle donne une idée de ce qu'il peut y avoir à faire pour en mettre une en place.

INTRODUCTION

- > Ça y est nous y sommes, nous voici en TP. Cette année le thème des TP est le développement d'une application que j'ai appelée Blimp.
- ▶ Le développement de Blimp va nous occuper les 8 séances prévues dans le planning, mais vous demandera de beaucoup travailler en



/index.html

Page

```
# $templateFile  
# $viewFile
```

```
+ setTemplate($templateName)  
+ setView($viewName)  
- insertView()  
+ render()
```