Chapitre 5 - authentification Web en PHP

L'objectif de ce cours est de découvrir comment soumettre l'accès à certaines zones d'un site à une authentification.

Table des matières

Webographie

Openclassrooms
CommentCaMarche

Evolution des besoins du contexte

Nous allons reprendre notre blog d'exemple, en souhaitant disposer d'une page d'administration qui affiche des statistiques simples sur le blog, à savoir le nombre de billets et de commentaires.

L'accès à cette page d'administration doit être réservé aux utilisateurs enregistrés du blog. Pour cela, une table T_UTILISATEUR est ajoutée à la base. Elle contient l'ensemble des utilisateurs. Par souci de simplicité, on ne distingue pas de hiérarchie entre les utilisateurs : ils ont tous le même droit d'accès à la page d'administration.

Le jeu de données de test comporte à présent la création d'un utilisateur nommé **admin**.

MonBlog.sql

```
...
insert into T_UTILISATEUR(UTIL_LOGIN, UTIL_MDP) values
('admin', 'secret');
```

Pour répondre à ce besoin, il va falloir ajouter au blog une zone dont l'accès est soumis à authentification. C'est un scénario typique d'utilisation des **sessions**.

Les sessions en PHP

Rappels sur le protocole HTTP

Pierre angulaire du Web, le protocole HTTP est un protocole **non connecté** (dit aussi sans état ou "*stateless*"). Cela signifie que chaque requête HTTP est indépendante des précédentes. Aucun mécanisme n'est prévu pour conserver un historique des requêtes. Ainsi, un serveur Web ne peut pas se "souvenir" qu'un ensemble de requêtes successives provient du même client, ce qui empêche des utilisations telles que le commerce électronique, pour lequel une mémorisation des achats de l'utilisateur à travers les différentes pages est nécessaire.

Rôle des sessions

Il est donc nécessaire de trouver un moyen de maintenir une certaine cohésion entre l'utilisateur et la requête, notamment :

- reconnaître les requêtes provenant du même utilisateur,
- associer un profil à l'utilisateur,
- connaître les paramètres de l'application pour l'utilisateur (nombre de produits vendus, ...).

PHP fournit cette fonctionnalité grâce aux **sessions**. Le support des sessions permet de stocker des données préservées d'une requête à l'autre. On peut ainsi reconnaître un utilisateur et lui fournir un contexte personnalisé (exemple : son panier d'achats).

Utilisation des sessions

Voici les étapes permettant d'utiliser les sessions en PHP :

1. Lorsqu'un visiteur arrive sur le site, on demande à créer une session pour lui.

2. Une fois la session générée, on peut créer une infinité de variables de session. Quelle que soit la page du site, on utilise la variable superglobale **$_SESSION** pour récupérer ou modifier les valeurs de ces variables. Par exemple, on peut créer une variable **$_SESSION['nom']** qui contient le nom du visiteur, **$_SESSION['prenom']** qui contient le prénom, etc.

3. Lorsque le visiteur se déconnecte du site, la session est fermée et PHP « oublie » alors toutes les variables de session créées. Il est souvent difficile de savoir précisément quand un visiteur quitte un site. Soit le visiteur clique sur un bouton de type « Déconnexion », soit on attend quelques minutes d'inactivité pour le déconnecter automatiquement : on parle alors de ***timeout***.

Fonctionnement interne des sessions

Lors de sa première visite sur un site, un client Web se voit attribuer un **identifiant de session** unique. Ce numéro est souvent écrit en hexadécimal, par exemple : a02bbffc6198e6e0cc2715047bc3766f.

Par défaut, cet identifiant est stocké dans un *cookie* nommé **PHPSESSID** ajouté aux échanges (requêtes HTTP) entre serveur et client. Si l'identifiant de session est présent dans la requête HTTP du client, PHP récupère les données associées à cette session. L'absence d'un identifiant ou d'un cookie de session dans une requête issue d'un client indique à PHP de créer une nouvelle session, et génère ainsi un nouvel identifiant de session.

Par défaut, toutes les données relatives à une session particulière seront stockées sur le **serveur**, dans un fichier du répertoire spécifié par le paramètre **session.save_path** dans les options du fichier **php.ini**. Un fichier sera créé pour chaque session. Sous XAMPP, les sessions sont stockées par défaut dans le répertoire **c:\xampp\tmp**.

Les sessions et le framework MVC

Utilisation traditionnelle des sessions

L'utilisation des sessions en PHP classique implique l'appel de plusieurs fonctions et l'emploi de la variable superglobale **$_SESSION**.

Création d'une session

La création d'une nouvelle session s'effectue en appelant la fonction `session_start`.

```
<?php
session_start();

...
```

Cette fonction doit **obligatoirement** être appelée au tout début de la page, avant tout code HTML.

Définition de variables de session

Une fois la session créée, on peut y ajouter de nouvelles variables identifiées par leur nom.

```
$_SESSION['prenom'] = 'Baptiste';
$_SESSION['age'] = 36;
```

Utilisation de variables de session

Dans n'importe quelle page où la fonction **session_start** a été appelée, on peut utiliser **$_SESSION** pour accéder aux valeurs des variables de session.

Il est prudent de vérifier au préalable que la variable existe, grâce à la fonction `isset`.

```
if (isset($_SESSION['prenom']) && isset($_SESSION['age'])) {
    echo 'Je te connais toujours ! Tu es ' . $_SESSION['prenom'] .
        ' et tu as ' . $_SESSION['age'] . ' ans.';
}
else {
    echo "Je ne te connais pas...";
}
```

Destruction d'une session

En fin de visite, la destruction explicite d'une session se fait grâce à la fonction `session_destroy`.

```
session_destroy();
```

Après l'appel, la variable **$_SESSION** n'est plus utilisable.

Rappel : cette destruction est automatique au bout d'un certain temps d'inactivité.

Intégration de la gestion des sessions dans le framework

La gestion des sessions est une problématique uniquement technique, indépendante de tout contexte métier. Elle a donc vocation à être intégrée dans notre *framework*.

Pour cela, nous allons lui ajouter une nouvelle classe nommée **Session** qui modélise une session.

Session.php

```php
<?php

/**
 * Classe modélisant la session.
 * Encapsule la superglobale PHP $_SESSION.
 *
 * @author Baptiste Pesquet
 */
class Session
{
    /**
     * Constructeur.
     * Démarre ou restaure la session
     */
    public function __construct() {
        session_start();
    }

    /**
     * Détruit la session actuelle
     */
    public function detruire() {
        session_destroy();
    }

    /**
     * Ajoute un attribut à la session
     *
     * @param string $nom Nom de l'attribut
     * @param string $valeur Valeur de l'attribut
     */
    public function setAttribut($nom, $valeur) {
        $_SESSION[$nom] = $valeur;
    }

    /**
     * Renvoie vrai si l'attribut existe dans la session
     *
     * @param string $nom Nom de l'attribut
     * @return bool Vrai si l'attribut existe et sa valeur n'est pas vide
     */
    public function existeAttribut($nom) {
        return (isset($_SESSION[$nom]) && $_SESSION[$nom] != "");
    }

    /**
     * Renvoie la valeur de l'attribut demandé
     *
     * @param string $nom Nom de l'attribut
     * @return string Valeur de l'attribut
     * @throws Exception Si l'attribut n'existe pas dans la session
     */
    public function getAttribut($nom) {
        if ($this->existeAttribut($nom)) {
            return $_SESSION[$nom];
        }
        else {
            throw new Exception("Attribut '$nom' absent de la session");
        }
    }
}
```

On remarque que son constructeur déclenche l'appel à **session_start**(). Toute instanciation de cette classe permet donc de démarrer la gestion des sessions.

La classe **Session** dispose de méthodes qui encapsulent l'accès à **$_SESSION** afin de pouvoir lire, écrire et vérifier l'existence de variables de session.

Une instance de la classe **Session** est ajoutée en tant qu'attribut de la classe **Requete**, qui modélise une requête HTTP dans le *framework*.

Requete.php

```
<?php

require_once 'Session.php';

/**
 * Classe modélisant une requête HTTP entrante.
 *
 * @author Baptiste Pesquet
 */
class Requete
{
    ...

    /** Objet session associé à la requête */
    private $session;

    /**
     * Constructeur
     *
     * @param array $parametres Paramètres de la requête
     */
    public function __construct($parametres) {
        $this->parametres = $parametres;
        $this->session = new Session();
    }

    /**
     * Renvoie l'objet session associé à la requête
     *
     * @return Session Objet session
     */
    public function getSession() {
        return $this->session;
    }

...
```

Le reste du *framework* n'est pas modifié. L'instanciation d'un objet **Requete** dans la méthode principale **routerRequete**() du routeur déclenche l'instanciation d'un objet **Session**, ce qui démarre la gestion des sessions.

Routeur.php

```
...
    /**
     * Méthode principale appelée par le contrôleur frontal
     * Examine la requête et exécute l'action appropriée
     */
    public function routerRequete() {
        try {
            // Fusion des paramètres GET et POST de la requête
            // Permet de gérer uniformément ces deux types de requête HTTP
            $requete = new Requete(array_merge($_GET, $_POST));
...
```

Application : ajout au blog d'une page d'administration

Mise en oeuvre des sessions pour répondre au besoin

Nous allons mettre en oeuvre les sessions afin de restreindre l'accès à la zone d'administration du site aux seuls utilisateurs authentifiés. Pour cela, nous allons ajouter un attribut (l'identifiant de l'utilisateur) à la session actuelle uniquement si cet utilisateur a réussi à s'authentifier.

L'accès à la zone d'administration sera donc refusé si la session est vide. Dans ce cas, le visiteur du site sera redirigé vers la page de connexion.

Ajout de l'action d'administration

Nous allons ajouter au site un nouveau contrôleur chargé de gérer les actions d'administration. Pour l'instant, notre besoin se limite à une seule action : l'affichage de statistiques simples. Ce sera donc l'action par défaut **index**() du contrôleur nommé **ControleurAdmin**.

On ajoute aux classes du modèle **Billet** et **Commentaire** les méthodes nécessaires pour calculer le nombre de billets et de commentaires.

Billet.php

```
...
    /**
     * Renvoie le nombre total de billets
     *
     * @return int Le nombre de billets
     */
    public function getNombreBillets() {
        $sql = 'select count(*) as nbBillets from T_BILLET';
        $resultat = $this->executerRequete($sql);
        $ligne = $resultat->fetch();  // Le résultat comporte toujours 1 ligne
        return $ligne['nbBillets'];
    }
}
```

Commentaire.php

```
...
    /**
```

```
     * Renvoie le nombre total de commentaires
     *
     * @return int Le nombre de commentaires
     */
    public function getNombreCommentaires()
    {
        $sql = 'select count(*) as nbCommentaires from T_COMMENTAIRE';
        $resultat = $this->executerRequete($sql);
        $ligne = $resultat->fetch();  // Le résultat comporte toujours 1 ligne
        return $ligne['nbCommentaires'];
    }
}
```

On écrit ensuite l'action qui affiche les statistiques dans le nouveau contrôleur d'administration.

ControleurAdmin.php

```
<?php

require_once 'ControleurSecurise.php';
require_once 'Modele/Billet.php';
require_once 'Modele/Commentaire.php';

/**
 * Contrôleur des actions d'administration
 *
 * @author Baptiste Pesquet
 */
class ControleurAdmin extends ControleurSecurise
{
    private $billet;
    private $commentaire;

    /**
     * Constructeur
     */
    public function __construct()
    {
        $this->billet = new Billet();
        $this->commentaire = new Commentaire();
    }

    public function index()
    {
        $nbBillets = $this->billet->getNombreBillets();
        $nbCommentaires = $this->commentaire->getNombreCommentaires();
        $login = $this->requete->getSession()->getAttribut("login");
        $this->genererVue(array('nbBillets' => $nbBillets,
            'nbCommentaires' => $nbCommentaires, 'login' => $login));
    }
}
```

On remarque que cette action récupère le `login` de l'utilisateur dans la session.

Enfin, on définit la vue qui affiche les données demandées. Le framework impose l'emplacement et le nommage du fichier associé : **Vue/Admin/index.php**.

Vue/Admin/index.php

```
<?php $this->titre = "Mon Blog - Administration" ?>

<h2>Administration</h2>

Bienvenue, <?= $this->nettoyer($login) ?> !
Ce blog comporte <?= $this->nettoyer($nbBillets) ?> billet(s) et
  <?= $this->nettoyer($nbCommentaires) ?> commentaire(s).
```

Sécurisation de l'action d'administration

L'action **index**() du contrôleur d'administration ne doit être accessible qu'après authentification de l'utilisateur. Au tout début de l'action, on doit donc vérifier que la session contient l'identifiant de l'utilisateur. Si c'est le cas, l'action continue à se dérouler. Sinon, le visiteur du site est redirigé vers la page de connexion.

Le comportement ci-dessus sera le même pour toutes les actions soumises à authentification. Plutôt que de dupliquer le code associé dans toutes les méthodes d'action, nous allons le centraliser en créant une superclasse abstraite nommée **ControleurSecurise**.

ControleurSecurise.php

```
<?php

require_once 'Framework/Controleur.php';

/**
 * Classe parente des contrôleurs soumis à authentification
 *
 * @author Baptiste Pesquet
 */
abstract class ControleurSecurise extends Controleur
{

    public function executerAction($action)
    {
        // Vérifie si les informations utilisateur sont présents dans la session
        // Si oui, l'utilisateur s'est déjà authentifié : l'exécution de l'action
        // continue normalement
        // Si non, l'utilisateur est renvoyé vers le contrôleur de connexion
        if ($this->requete->getSession()->existeAttribut("idUtilisateur")) {
            parent::executerAction($action);
        }
        else {
            $this->rediriger("connexion");
        }
    }

}
```

Cette classe abstraite redéfinit la méthode **executerAction**() de la classe **Controleur** dont elle hérite. Cette redéfinition permet de vérifier l'existence de l'identifiant de l'utilisateur dans la session. S'il existe, c'est que l'utilisateur s'est déjà connecté et l'action continue alors normalement. Sinon, le visiteur est redirigé vers le contrôleur de connexion.

Au passage, une méthode **rediriger**() doit être ajoutée à la classe **Controleur** du *framework*.

Controleur.php

```
...
    /**
     * Effectue une redirection vers un contrôleur et une action spécifiques
     *
     * @param string $controleur Contrôleur
     * @param type $action Action Action
     */
    protected function rediriger($controleur, $action = null)
    {
        $racineWeb = Configuration::get("racineWeb", "/");
        // Redirection vers l'URL racine_site/controleur/action
        header("Location:" . $racineWeb . $controleur . "/" . $action);
    }
}
```

Elle utilise la fonction PHP header afin de rediriger le navigateur client vers l'action d'un autre contrôleur.

Comment faire pour que l'action du contrôleur d'administration soit sécurisée et soumise à authentification ? Il suffit de le faire hériter non plus de `Controleur`, mais de `ControleurSecurise`.

ControleurAdmin.php

```
<?php

require_once 'ControleurSecurise.php';
require_once 'Modele/Billet.php';
require_once 'Modele/Commentaire.php';

/**
 * Contrôleur des actions d'administration
 *
 * @author Baptiste Pesquet
 */
class ControleurAdmin extends ControleurSecurise
{
...
```

On constate ici l'intérêt du mécanisme d'héritage de la POO : toutes les actions des contrôleurs qui hériteront de **ControleurSecurise** seront automatiquement sécurisées, sans aucun code spécifique à écrire dans ces actions !

Authentification de l'utilisateur

Pour finir, il nous faut gérer la connexion de l'utilisateur, ainsi que sa déconnexion. On va logiquement définir les actions associées dans un nouveau contrôleur nommé **ControleurConnexion**.

Ce contrôleur a besoin de vérifier si un utilisateur existe dans la base de données à partir de son login et de son mot de passe. Pour cela, il utilise une classe **Utilisateur** ajoutée au modèle

Utilisateur.php

```
<?php

require_once 'Framework/Modele.php';

/**
 * Modélise un utilisateur du blog
 *
 * @author Baptiste Pesquet
 */
class Utilisateur extends Modele {

  /**
   * Vérifie qu'un utilisateur existe dans la BD
   *
   * @param string $login Le login
   * @param string $mdp Le mot de passe
   * @return boolean Vrai si l'utilisateur existe, faux sinon
   */
  public function connecter($login, $mdp)
  {
    $sql = "select UTIL_ID from T_UTILISATEUR where UTIL_LOGIN=? and UTIL_MDP=?";
    $utilisateur = $this->executerRequete($sql, array($login, $mdp));
    return ($utilisateur->rowCount() == 1);
  }

  /**
   * Renvoie un utilisateur existant dans la BD
   *
   * @param string $login Le login
   * @param string $mdp Le mot de passe
   * @return mixed L'utilisateur
   * @throws Exception Si aucun utilisateur ne correspond aux paramètres
   */
  public function getUtilisateur($login, $mdp)
  {
    $sql = "select UTIL_ID as idUtilisateur, UTIL_LOGIN as login, UTIL_MDP as mdp
      from T_UTILISATEUR where UTIL_LOGIN=? and UTIL_MDP=?";
    $utilisateur = $this->executerRequete($sql, array($login, $mdp));
    if ($utilisateur->rowCount() == 1)
      return $utilisateur->fetch();  // Accès à la première ligne de résultat
    else
      throw new Exception("Aucun utilisateur ne correspond aux identifiants
        fournis");
    }
}
```

Les méthodes de la classe **Utilisateur** permettent de vérifier qu'un utilisateur existe et de récupérer ses propriétés (identifiant, login et mot de passe).

Le contrôleur de connexion défini ci-dessous utilise cette classe. Son action par défaut **index**() affiche le formulaire de connexion. Les actions **connecter**( ) et **deconnecter**() permettent respectivement de gérer la connexion (gestion du login et du mot de passe saisis par l'utilisateur) et la déconnexion d'un utilisateur déjà connecté.

ControleurConnexion.php

```
<?php

require_once 'Framework/Controleur.php';
require_once 'Modele/Utilisateur.php';

class ControleurConnexion extends Controleur
{
    private $utilisateur;

    public function __construct()
    {
        $this->utilisateur = new Utilisateur();
    }

    public function index()
    {
        $this->genererVue();
    }

    public function connecter()
    {
        if ($this->requete->existeParametre("login") &&
          $this->requete->existeParametre("mdp")) {
            $login = $this->requete->getParametre("login");
            $mdp = $this->requete->getParametre("mdp");
            if ($this->utilisateur->connecter($login, $mdp)) {
                $utilisateur = $this->utilisateur->getUtilisateur($login, $mdp);
                $this->requete->getSession()->setAttribut("idUtilisateur",
                        $utilisateur['idUtilisateur']);
                $this->requete->getSession()->setAttribut("login",
                        $utilisateur['login']);
                $this->rediriger("admin");
            }
            else
                $this->genererVue(array('msgErreur' =>
                  'Login ou mot de passe incorrects'), "index");
        }
        else
            throw new Exception(
              "Action impossible : login ou mot de passe non défini");
    }

    public function deconnecter()
    {
        $this->requete->getSession()->detruire();
        $this->rediriger("accueil");
    }

}
```

Si le couple login/mot de passe saisi correspond bien à un utilisateur existant, son idenfiant ainsi que son login sont ajoutés dans la session, puis l'utilisateur (maintenant authentifié) est redirigé vers le contrôleur d'administration.

Lorsque les informations saisies par l'utilisateur sont incorrectes, le contrôleur de connexion affiche à nouveau le formulaire de connexion (qui est la vue par défaut), mais en ajoutant aux données de la vue un message d'erreur (paramètre `msgErreur`).

ControleurConnexion doit hériter de **Controleur** et non de **ControleurSecurise**, sinon l'action de connexion sera toujours inaccessible : il faudrait être authentifié pour pouvoir s'authentifier...

Enfin, la vue par défaut **Vue/Connexion/index.php** associée au contrôleur de connexion affiche à l'utilisateur le formulaire permettant la saisie de son login et de son mot de passe.

Vue/Connexion/index.php

```
<?php $this->titre = "Mon Blog - Connexion" ?>

<p>Vous devez être connecté pour accéder à cette zone.</p>
<form action="connexion/connecter" method="post">
  <input name="login" type="text" placeholder="Entrez votre login" required
    autofocus>
  <input name="mdp" type="password" placeholder="Entrez votre mot de passe"
    required>
  <button type="submit">Connexion</button>
</form>
<?php if (isset($msgErreur)): ?>
  <p><?= $msgErreur ?></p>
<?php endif; ?>
```

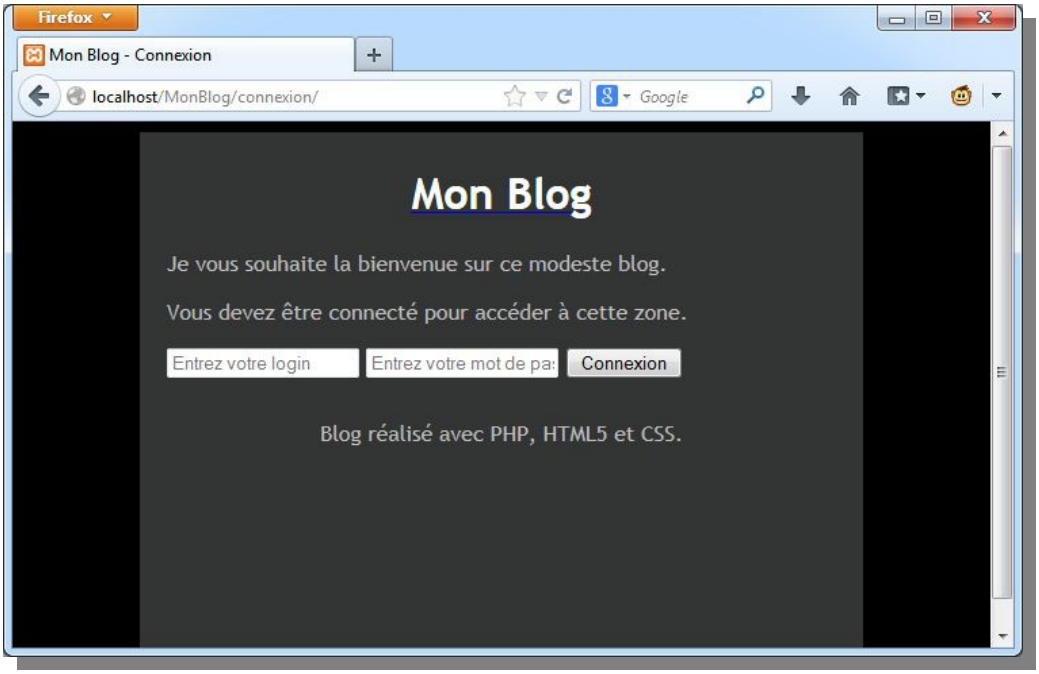
On ajoute à la vue d'administration un lien permettant à un utilisateur de se déconnecter.

Vue/Admin/index.php

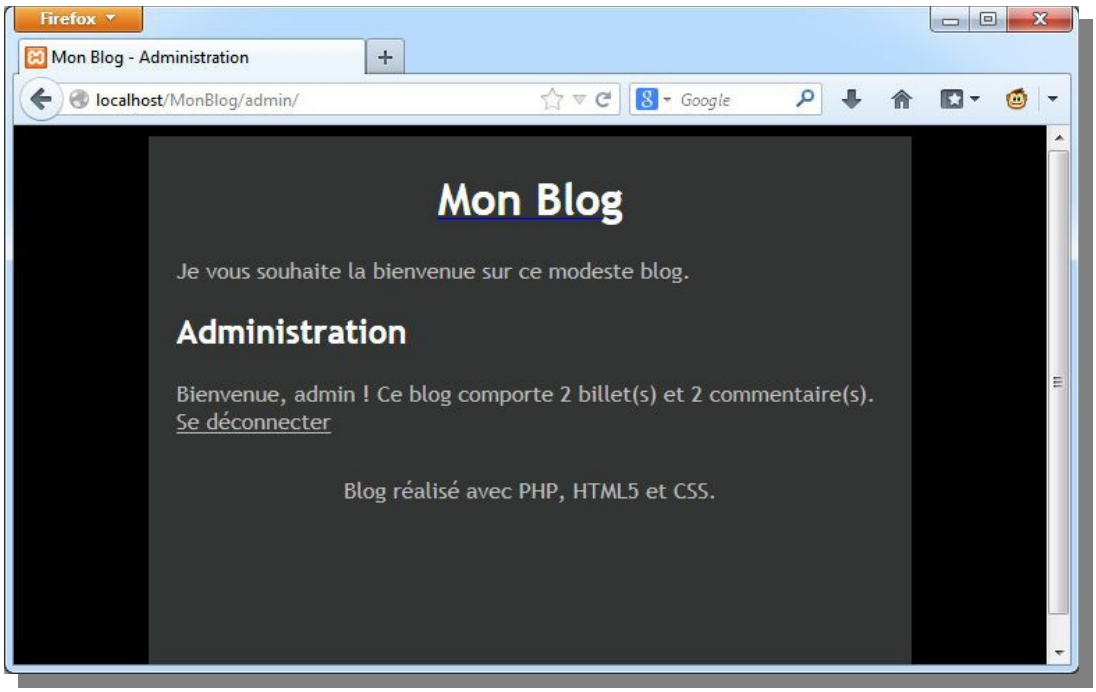
```
...
<a id="lienDeco" href="connexion/deconnecter">Se déconnecter</a>
```

Résultat obtenu

Lors d'une première visite, toute tentative d'accès à la zone d'administration du blog (**/admin/**) provoque la redirection vers le formulaire de connexion (**/connexion/**).



Lorsque l'authentification réussit, la page d'administration permet à l'utilisateur de consulter les statistiques du blog et de se déconnecter.



Un échec d'authentification déclenche l'affichage d'un message d'erreur sur la page de connexion.

Conclusion

Grâce aux fonctionnalités ajoutées à notre *framework*, il est facile de soumettre certaines actions d'un site Web à authentification préalable de l'utilisateur. Il suffit de définir un contrôleur sécurisé (classe abstraite) dont hériteront les contrôleurs à protéger, ainsi qu'un autre contrôleur (non sécurisé) gérant les actions de connexion/déconnexion.