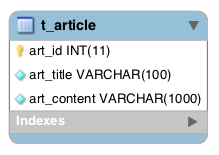
ITÉRATION 1 : AFFICHAGE DE LA LISTE DES ARTICLES

Le but de cette itération est d'écrire la toute première version de notre application web d'exemple. Conformément aux principes de l'agilité, nous commençons par réaliser ce qui apporte le plus de "valeur" à notre CMS : la page d'accueil qui affiche la liste de tous ses articles.

Création de la base de données

Le rôle principal de notre CMS est de gérer des articles. Tous les articles sont sauvegardés dans une base de données relationnelle nommée microcms. Le SGBDR utilisé est MySQL dans notre contexte. Un utilisateur MySQL nommé microcms\_user a tous les droits sur la base microcms. Il est utilisé pour se connecter à MySQL depuis notre application web.

Un article est caractérisé par son titre (art\_title) et son contenu (art\_content). On modélise un article sous la forme d'un enregistrement dans une table nommée t\_article. Le champ art\_id est la clé primaire de cette table.

Table t\_article

Un premier script database.sql va nous permettre de créer la base microcms qui hébergera toutes nos données persistantes, ainsi que l'utilisateur microcms\_user utilisé pour s'y connecter depuis notre code PHP.

1create database if not exists microcms character set utf8 collate utf8\_unicode\_ci;

2use microcms;

3

4grant all privileges on microcms.\* to 'microcms\_user'@'localhost' identified by 'secret';

Voici le script SQL structure.sql qui permet de créer la structure de la base.

1drop table if exists t\_article;

2

3create table t\_article (

4art\_id integer not null primary key auto\_increment,

5art\_title varchar(100) not null,

6art\_content varchar(2000) not null

7) engine=innodb character set utf8 collate utf8\_unicode\_ci;

On ajoute dans cette table un petit jeu de données de test contenant quelques articles. Voici le script SQLcontent.sql associé.

1insert into t\_article values

2(1, 'First article', 'Hi there! This is the very first article.');

3insert into t\_article values

4(2, 'Lorem ipsum', 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut hendrerit mauris ac porttitor accumsan. Nunc vitae pulvinar odio, auctor interdum dolor. Aenean sodales dui quis metus iaculis, hendrerit vulputate lorem vestibulum. Suspendisse pulvinar, purus at euismod semper, nulla orci pulvinar massa, ac placerat nisi urna eu tellus. Fusce dapibus rutrum diam et dictum. Sed tellus ipsum, ullamcorper at consectetur vitae, gravida vel sem. Vestibulum pellentesque tortor et elit posuere vulputate. Sed et volutpat nunc. Praesent nec accumsan nisi, in hendrerit nibh. In ipsum mi, fermentum et eleifend eget, eleifend vitae libero. Phasellus in magna tempor diam consequat posuere eu eget urna. Fusce varius nulla dolor, vel semper dui accumsan vitae. Sed eget risus neque.');

5insert into t\_article values

6(3, 'Lorem ipsum in french', "J’en dis autant de ceux qui, par mollesse d’esprit, c’est-à-dire par la crainte de la peine et de la douleur, manquent aux devoirs de la vie. Et il est très facile de rendre raison de ce que j’avance. Car, lorsque nous sommes tout à fait libres, et que rien ne nous empêche de faire ce qui peut nous donner le plus de plaisir, nous pouvons nous livrer entièrement à la volupté et chasser toute sorte de douleur ; mais, dans les temps destinés aux devoirs de la société ou à la nécessité des affaires, souvent il faut faire divorce avec la volupté, et ne se point refuser à la peine. La règle que suit en cela un homme sage, c’est de renoncer à de légères voluptés pour en avoir de plus grandes, et de savoir supporter des douleurs légères pour en éviter de plus fâcheuses.");

Démarrez les serveurs Apache et MySQL, puis importez successivement les scripts database.sql,  structure.sql et content.sql en utilisant votre outil d'administration MySQL favori (**phpMyAdmin**par exemple). Notre base de données est maintenant prête à être utilisée.

Affichage de la liste des articles

**Page d'accueil PHP**

À présent, écrivons le fichier source index.php qui affiche la liste de tous les articles. Nous allons travailler directement dans le répertoire servi par votre serveur web local. Son emplacement dépend de votre installation. Voici quelques emplacements possibles :

* c:\xampp\htdocs avec XAMPP sous Windows.
* /var/www sous Linux.
* /Applications/MAMP/htdocs avec MAMP sous Mac.

Créez dans ce répertoire un sous-répertoire nommé MicroCMS qui sera le répertoire racine de notre application web. Déplacez-vous dans ce répertoire puis créez un fichier texte nommé index.php. Ensuite, copiez-collez le code source ci-dessous dans ce fichier.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8" />

5    <link href="microcms.css" rel="stylesheet" />

6    <title>MicroCMS - Home</title>

7</head>

8<body>

9    <header>

10        <h1>MicroCMS</h1>

11    </header>

12    <?php

13    $bdd = new *PDO*('mysql:host=localhost;dbname=microcms;charset=utf8', 'microcms\_user', 'secret');

14    $articles = $bdd->query('select \* from t\_article order by art\_id desc');

15    foreach ($articles as $article): ?>

16    <article>

17        <h2><?php echo $article['art\_title'] ?></h2>

18        <p><?php echo $article['art\_content'] ?></p>

19    </article>

20    <?php endforeach ?>

21    <footer class="footer">

22        <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

23    </footer>

24</body>

25</html>

Ce code source mérite quelques commentaires :

* Il utilise la norme [HTML5](http://fr.wikipedia.org/wiki/HTML5) et emploie certaines nouvelles balises, comme <article> ;
* Il emploie la [syntaxe alternative](http://php.net/manual/fr/control-structures.alternative-syntax.php) pour la boucle foreach ;
* Il utilise l'extension [PDO](http://php.net/manual/fr/book.pdo.php) de PHP afin d'interagir avec la base de données.

Pour le reste, il s'agit d'un exemple assez classique d'utilisation de PHP pour construire une page dynamique affichée par le navigateur client.

**Feuille de style CSS**

Afin d'améliorer la présentation, nous utilisons une feuille de style nommée microcms.css.

1.footer {

2    *border-top*: 1px solid #ccc;

3    *padding-top*: 10px;

4    *text-align*: center;

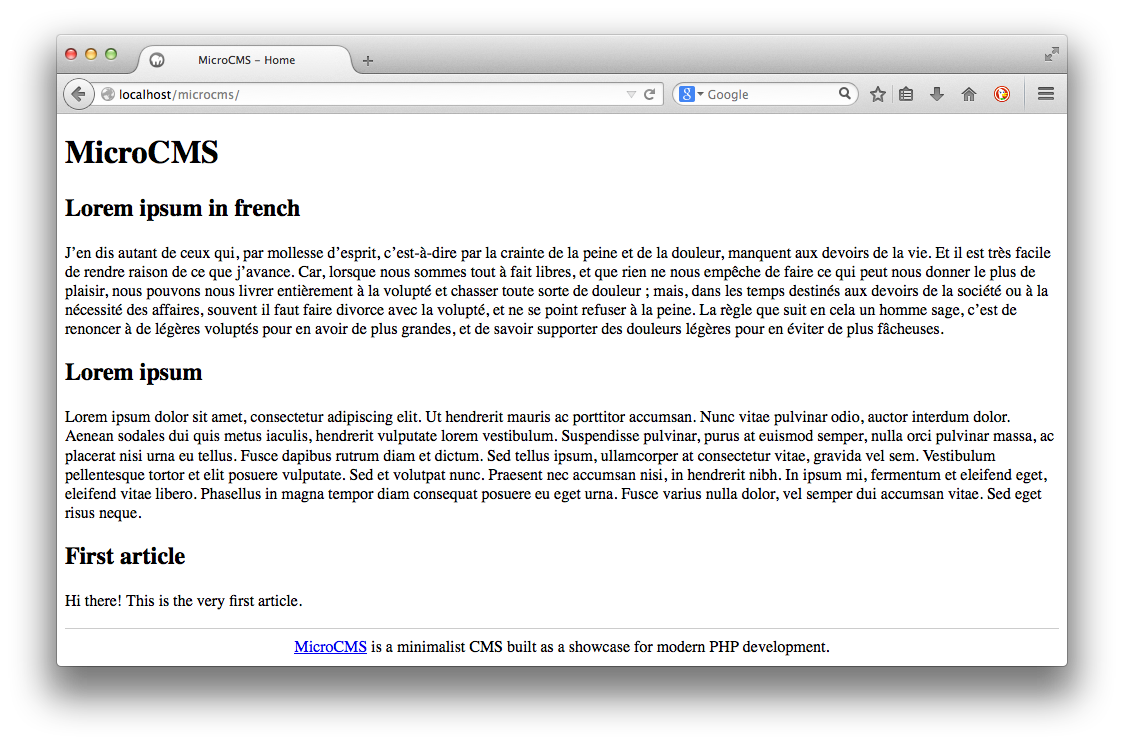
5}

Créez cette feuille de style dans le même répertoire que le fichier index.php.

Enfin, créez dans le répertoire MicroCMS un sous-répertoire db et copiez-y les trois scripts SQL utilisés plus haut.

**Affichage obtenu**

Après avoir vérifié que votre serveur web est bien démarré, lancez votre navigateur web favori et tapez l'URL <http://localhost/microcms>. Vous devez voir s'afficher la liste des articles.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-01).

Bilan

La toute première version (très simpliste) de notre application Web est maintenant opérationnelle. Elle nous a permis d'initialiser la structure de l'application. La prochaine itération consistera à améliorer son architecture.

ITÉRATION 2 : PASSAGE À UNE ARCHITECTURE MVC

Le but de cette itération est de modifier notre application web d'exemple en introduisant une architecture de type **MVC** (Modèle-Vue-Contrôleur).

Cette itération s'inspire en grande partie de l'article [Symfony versus PHP pur](http://symfony.com/fr/doc/current/book/from_flat_php_to_symfony2.html).

Critique de l'application existante

Les principaux défauts de notre application web actuelle sont les suivants :

* Elle mélange balises HTML et code PHP.
* Sa structure est monolithique, ce qui rend sa réutilisation difficile.

De manière très générale, tout logiciel doit gérer plusieurs problématiques :

* interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage. C'est la problématique de **présentation** ;
* opérations sur les données (calculs) en rapport avec les règles métier (« business logic »). C'est la problématique des **traitements** ;
* accès et stockage des informations qu'il manipule, notamment entre deux utilisations. C'est la problématique des **données**.

L'application web actuelle mélange code de présentation (les balises HTML) et accès aux données (requêtes SQL). Ceci est contraire au **principe de responsabilité unique** *(Single Responsibility Principle)*. Ce principe de conception logicielle est le suivant : afin de clarifier l'architecture et de faciliter les évolutions, chaque élément d'une application doit avoir un rôle et une responsabilité unique.

L'architecture actuelle montre ses limites dès que le contexte se complexifie. Le volume de code des pages PHP explose et la maintenabilité devient délicate. Il faut faire mieux.

Séparation des responsabilités

**Isolation de l'affichage**

Une première amélioration consiste à séparer le code d'accès aux données du code de présentation au sein du fichier index.php.

1<?php

2// Data access

3$bdd = new *PDO*('mysql:host=localhost;dbname=microcms;charset=utf8', 'microcms\_user', 'secret');

4$articles = $bdd->query('select \* from t\_article order by art\_id desc');

5?>

6<!doctype html>

7<html>

8<head>

9    <meta charset="utf-8" />

10    <link href="microcms.css" rel="stylesheet" />

11    <title>MicroCMS - Home</title>

12</head>

13<body>

14    <header>

15        <h1>MicroCMS</h1>

16    </header>

17    <?php foreach ($articles as $article): ?>

18        <article>

19            <h2><?php echo $article['art\_title'] ?></h2>

20            <p><?php echo $article['art\_content'] ?></p>

21        </article>

22    <?php endforeach ?>

23    <footer class="footer">

24        <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

25    </footer>

26</body>

27</html>

Le code est devenu plus lisible, mais les problématiques de présentation et d'accès aux données sont toujours gérées au sein d'un même fichier PHP. En plus de limiter la modularité, ceci est contraire aux bonnes pratiques de développement PHP (norme [PSR-1](http://www.php-fig.org/psr/psr-1/)).

On peut aller plus loin dans le découplage en regroupant le code d'affichage précédent dans un fichier dédié nommé view.php.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8" />

5    <link href="microcms.css" rel="stylesheet" />

6    <title>MicroCMS - Home</title>

7</head>

8<body>

9    <header>

10        <h1>MicroCMS</h1>

11    </header>

12    <?php foreach ($articles as $article): ?>

13    <article>

14        <h2><?php echo $article['art\_title'] ?></h2>

15        <p><?php echo $article['art\_content'] ?></p>

16    </article>

17    <?php endforeach ?>

18    <footer class="footer">

19        <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

20    </footer>

21</body>

22</html>

Le fichier index.php devient alors :

1<?php

2// Data access

3$bdd = new *PDO*('mysql:host=localhost;dbname=microcms;charset=utf8', 'microcms\_user', 'secret');

4$articles = $bdd->query('select \* from t\_article order by art\_id desc');

5

6// Data display

7require 'view.php';

La fonction PHP [require](http://php.net/manual/fr/function.require.php) fonctionne de manière similaire à [include](http://php.net/manual/fr/function.include.php) : elle inclut et exécute le fichier spécifié. En cas d'échec, include ne produit qu'un avertissement alors que require stoppe l'exécution du fichier source avec une erreur fatale.

La balise de fin de code PHP ?> est volontairement omise à la fin du fichier index.php. C'est une[bonne pratique](http://php.net/manual/fr/language.basic-syntax.phptags.php) pour les fichiers qui ne contiennent que du PHP. Elle permet d'éviter des problèmes lors d'inclusions de fichiers.

**Isolation de l'accès aux données**

Nous avons amélioré l'architecture de notre application, mais nous pourrions gagner en modularité en isolant le code d'accès aux données dans un fichier PHP dédié. Appelons ce fichier model.php.

1<?php

2

3// Return all articles

4function getArticles() {

5    $bdd = new *PDO*('mysql:host=localhost;dbname=microcms;charset=utf8', 'microcms\_user', 'secret');

6    $articles = $bdd->query('select \* from t\_article order by art\_id desc');

7    return $articles;

8}

Dans ce fichier, nous avons déplacé la récupération des articles du CMS à l'intérieur d'une fonction nommée getArticles.

Le code d'affichage (fichier view.php) ne change pas. Le lien entre accès aux données et présentation est effectué par le fichier index.php. Ce fichier est maintenant très simple.

1<?php

2

3require 'model.php';

4$articles = getArticles();

5require 'view.php';

Le résultat affiché reste bien entendu identique.

**Bilan**

Outre les scripts SQL et la feuille de style microcms.css, notre application est maintenant constituée de trois fichiers :

* model.php (code PHP uniquement) pour l'accès aux données ;
* view.php (code PHP et balises HTML) pour l'affichage des articles du CMS ;
* index.php (code PHP uniquement) pour faire le lien entre les deux pages précédentes.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-02).

Cette nouvelle structure est plus complexe, mais les responsabilités de chaque partie sont maintenant claires. En faisant ce travail de **refactorisation,** nous avons rendu notre exemple conforme à un modèle d'architecture très employé sur le Web : le modèle **MVC**.

Le modèle MVC

**Présentation**

Le modèle MVC décrit une manière d'architecturer une application informatique en la décomposant en trois sous-parties :

* la partie **Modèle** ;
* la partie **Vue**;
* la partie **Contrôleur**.

Ce **patron de conception***(design pattern)* a été imaginé à la fin des années 1970 pour le langage Smalltalk afin de bien séparer le code de l'interface graphique de la logique applicative. Il est utilisé dans de très nombreux langages : bibliothèques Swing et Model 2 (JSP) de Java, frameworks PHP, ASP.NET MVC, etc.

**Rôles des composants**

La partie **Modèle** d'une architecture MVC encapsule la logique métier ("*business logic*") ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet).

La partie **Vue** s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données.

La partie **Contrôleur** gère la dynamique de l'application. Elle fait le lien entre l'utilisateur et le reste de l'application.

**Interactions entre les composants**

Le diagramme ci-dessous résume les relations entre les composants d'une architecture MVC.

Extrait de la documentation du framework Symfony2

La demande de l'utilisateur (exemple : une requête HTTP) est reçue et interprétée par le Contrôleur. Celui-ci utilise les services du Modèle afin de préparer les données à afficher. Ensuite, le Contrôleur fournit ces données à la Vue, qui les présente à l'utilisateur (par exemple sous la forme d'une page HTML).

On peut trouver des variantes moins "pures" de cette architecture dans lesquelles la Vue interagit directement avec le Modèle afin de récupérer les données dont elle a besoin.

**Avantages et inconvénients**

L'architecture MVC offre une séparation claire des responsabilités au sein d'une application, en conformité avec les principes de conception déjà étudiés : responsabilité unique, couplage faible et cohésion forte. Le prix à payer est une augmentation de la complexité de l'architecture.

Dans le cas d'une application web, l'utilisation du modèle MVC permet aux pages HTML (qui constituent la partie Vue) de contenir le moins possible de code serveur, étant donné que le scripting est regroupé dans les deux autres parties de l'application.

**Différences avec une architecture en couches**

Attention à ne pas employer le terme de "couche" à propos d'une architecture MVC. Dans une architecture en couches, chaque couche ne peut communiquer qu'avec les couches adjacentes. Les parties Modèle, Vue et Contrôleur ne sont donc pas des couches au sens propre du mot.

ITÉRATION 3 : INTÉGRATION DU FRAMEWORK PHP SILEX

Le but de cette itération est d'intégrer un framework à notre application.

Avantages apportés par un framework

En informatique comme ailleurs, il est rarement utile de réinventer la roue. La plupart des applications web ont les mêmes besoins de base : accès au contenu de la requête HTTP reçue, création et renvoi de la réponse, gestions des sessions... Il existe une catégorie de logiciels dont le rôle est de gérer ces besoins communs : ce sont les **frameworks**.

Un framework fournit un ensemble de services de base, généralement sous la forme de classes en interaction. À condition de respecter l'architecture qu'il préconise (souvent une déclinaison du modèle MVC), un framework PHP libère le développeur de nombreuses tâches techniques comme le routage des requêtes, la sécurité, la gestion du cache, etc. Cela lui permet de se concentrer sur l'essentiel, c'est-à-dire ses tâches métier. Il existe une grande quantité de frameworks PHP. Parmi les plus connus, citons[Symfony](http://symfony.com/), [Zend Framework](http://framework.zend.com/) ou encore [Laravel](http://laravel.com/).

Choix du framework

Il est possible d'écrire soi-même son propre framework, puis de le réutiliser dans tous ses projets PHP. C'est une tâche intéressante et formatrice, mais il est difficile d'obtenir le niveau de qualité et de sûreté que peut offrir un framework professionnel du marché. Si la construction d'un framework PHP vous intéresse, consultez le tutoriel [Evoluer vers une architecture MVC en PHP](http://bpesquet.developpez.com/tutoriels/php/evoluer-architecture-mvc/). Ici, nous allons utiliser un framework existant plutôt que d'en construire un nous-même.

**Frameworks full stack contre micro-frameworks**

Au sein de l'écosystème des frameworks PHP, deux catégories coexistent. Les frameworks classiques comme Symfony ou Zend fournissent un très grand nombre de services (gestion avancée des formulaires,*mapping* objet-relationnel, etc.). Ces frameworks complexes imposent leur architecture au développeur. On les appelle des frameworks *full stack*.

L'autre catégorie de frameworks se concentre sur un ensemble réduit de services de base, et laisse au développeur beaucoup de liberté pour construire son application. On les appelle des **micro-frameworks**. Pour en découvrir certains, consultez [cet article](http://xybu.me/a-comparison-of-php-micro-frameworks/).

**Et le vainqueur est...**

Nous allons baser notre application web sur le micro-framework [Silex](http://silex.sensiolabs.org/). Ce choix d'un micro-framework plutôt que d'un framework *full stack* est bien entendu discutable. On pourra objecter qu'un framework comme Symfony est de plus en plus implanté en entreprise, et que les compétences associées sont de plus en plus recherchées. De plus, il existe beaucoup plus de documentation et d'exemples autour de Symfony qu'autour de Silex.

Malgré tout, j'ai choisi Silex pour les raisons suivantes :

* Il est plus facile à prendre en main et moins intimidant que Symfony pour un développeur PHP peu expérimenté.
* Il permet de bâtir petit à petit l'architecture de l'application en détaillant les changement à chaque étape.
* Il est *made in France* et basé sur les mêmes composants que Symfony, ce qui facilitera une éventuelle évolution vers ce dernier. J'irais jusqu'à dire que Silex représente une excellente solution pour s'autoformer progressivement à Symfony.

Premiers pas avec Silex

Intégrer un framework à une application nécessite de savoir l'utiliser. Le cours OpenClassrooms [Premiers pas avec le framework PHP Silex](http://openclassrooms.com/courses/premiers-pas-avec-le-framework-php-silex) décrit en détail l'installation de Silex et la création d'une application Web minimaliste avec ce framework.

Sauf si vous maîtrisez déjà Composer, Silex et la définition d'un hôte virtuel Apache, je vous incite fortement à suivre le cours ci-dessus. Il contient des informations importantes pour installer et configurer une application Silex. Ces informations sont nécessaires pour réaliser la suite de cette itération.

Réécriture de l'application avec Silex

**Récupération de Silex via Composer**

Afin de pouvoir récupérer les fichiers de Silex, créez dans le répertoire MicroCMS  servi par votre serveur web local un fichier texte nommé composer.json ayant le contenu suivant.

1{

2    "require": {

3        "silex/silex": "~1.2"

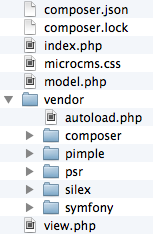
4    }

5}

Ouvrez une fenêtre de terminal et déplacez-vous dans le répertoire MicroCMS. Ensuite, lancez la commande ci-dessous :

composer install

Si tout va bien, Composer va télécharger les fichiers de Silex dans un sous-répertoire nommé vendor du répertoire courant. L'arborescence du projet est maintenant la suivante.

Arborescence du projet

Si vous utilisez Git, créez dans le répertoire MicroCMS un fichier nommé .gitignore  ayant le contenu suivant.

1vendor/

Cela permet d'exclure ce répertoire du contrôle de code source.

**Mise à jour de l'arborescence**

Créez dans MicroCMS  les sous-répertoires suivants :

* app, qui contiendra la configuration de l'application Silex ;
* db, qui contiendra les scripts SQL de création de la base de données ;
* src, qui contiendra les fichiers source PHP ;
* views, qui contiendra les vues de l'application ;
* web, qui contiendra les fichiers accessibles aux clients web.

Déplacez le fichier view.php dans le sous-répertoire views, puis déplacez le fichier model.php dans le sous-répertoire src. Déplacez ensuite les fichiers index.php et microcms.css dans le sous-répertoireweb. Enfin, créez ou déplacez dans le sous-répertoire db les scripts SQLdatabase.sql, structure.sql et content.sql.

Dans le sous-répertoire app, créez un fichier routes.php avec le contenu ci-dessous.

1<?php

2

3// Home page

4$app->get('/', function () {

5    require '../src/model.php';

6    $articles = getArticles();

7

8    ob\_start();             // start buffering HTML output

9    require '../views/view.php';

10    $view = ob\_get\_clean(); // assign HTML output to $view

11    return $view;

12});

Silex permet de définir des **routes**, c'est-à-dire des points d'entrée dans l'application. À chaque route est associée une réponse construite par notre code. La route ci-dessus correspond à l'URL racine de l'application (/). La [fonction anonyme](http://php.net/manual/fr/functions.anonymous.php) associée à cette route utilise la fonction getArticles définie dans le fichier model.php pour récupérer la liste des articles.

Une fonction qui gère une route est appelée un **contrôleur**.

Chaque contrôleur Silex doit renvoyer explicitement une réponse. Les fonctions PHP [ob\_start](http://php.net/manual/fr/function.ob-start.php) et[ob\_get\_clean](http://php.net/manual/fr/function.ob-get-clean.php) permettent de récupérer le résultat de l'appel à require '../views/view.php' (autrement dit la vue HTML générée) dans une variable nommée $view. Cette variable est renvoyée par le contrôleur.

Éditez index.php et remplacez son contenu par le code source ci-dessous.

1<?php

2

3require\_once \_\_DIR\_\_.'/../vendor/autoload.php';

4

5$app = new *Silex*\Application();

6

7require \_\_DIR\_\_.'/../app/routes.php';

8

9$app->run();

Ce fichier constitue le **contrôleur frontal** de notre application web. Il centralise la gestion des requêtes HTTP entrantes. Dans ce fichier, on instancie l'objet Silex principal $app puis on inclut la définition des routes de l'application (fichier routes.php).

Toujours dans web, créez un nouveau fichier texte nommé .htaccess contenant le texte ci-dessous. Ce fichier permet de configurer le serveur web Apache pour rediriger toutes les requêtes entrantes versindex.php.

1# Redirect incoming URLs to index.php

2<IfModule mod\_rewrite.c>

3    Options -MultiViews

4

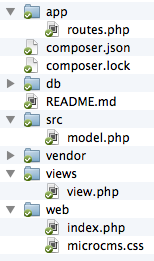
5    RewriteEngine On

6    RewriteCond %{REQUEST\_FILENAME} !-f

7    RewriteRule ^ index.php [QSA,L]

8</IfModule>

Voici l'arborescence obtenue après tous ces changements (les fichiers commençant par '.' ne sont pas affichés).

Arborescence du projet

**Définition d'un hôte virtuel**

Enfin, il nous reste à configurer un **hôte virtuel** afin que l'application puisse répondre à une URL de la forme [http://microcms](http://microcms/). En suivant l'exemple donné dans le cours [Premiers pas avec le framework PHP Silex](http://openclassrooms.com/courses/premiers-pas-avec-le-framework-php-silex), éditez le fichier de configuration Apache httpd-vhosts.conf et ajoutez le contenu ci-après en adaptant les lignes commençant par DocumentRoot et Directory à votre configuration locale. Voici un exemple de configuration avec XAMPP sous Windows.

1<VirtualHost \*:80>

2    DocumentRoot "C:\xampp\htdocs\MicroCMS\web"

3    ServerName microcms

4    <Directory "C:\xampp\htdocs\MicroCMS\web">

5        AllowOverride All

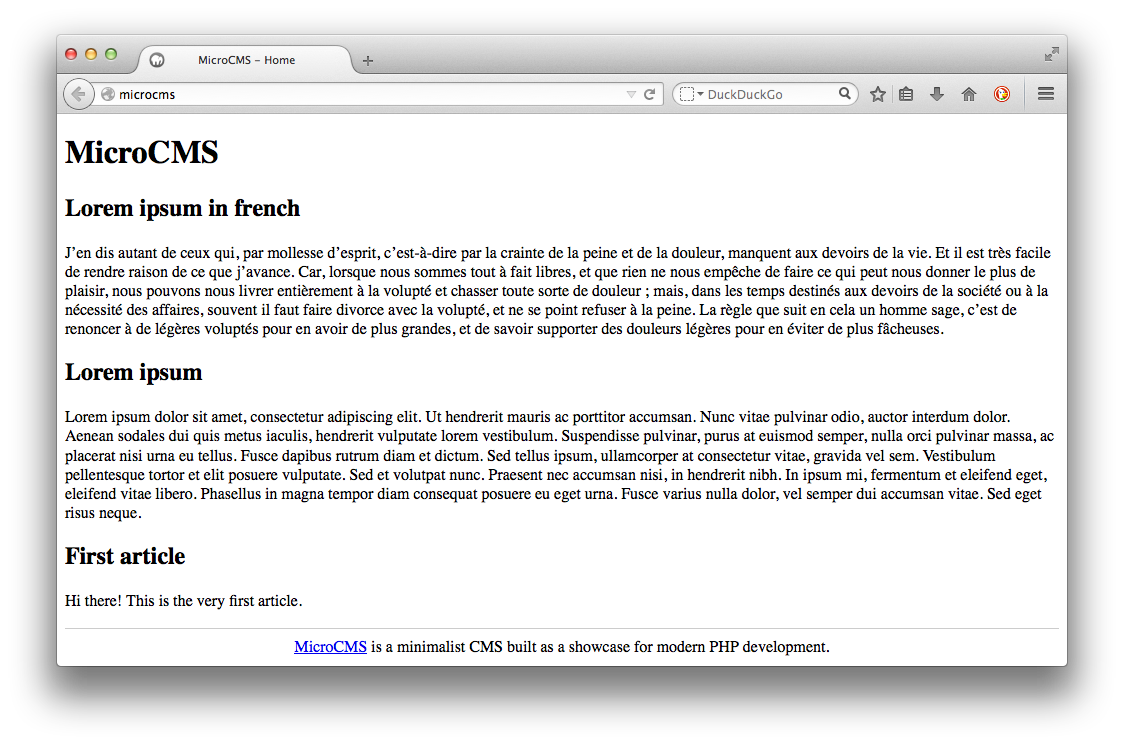
6    </Directory>

7</VirtualHost>

Placer la racine de l'hôte virtuel dans le sous-répertoire web plutôt qu'à la racine est une bonne pratique qui permet de protéger le reste de l'application (tout ce qui n'est pas dans web) des accès externes.

N'oubliez pas de définir également un hôte virtuel localhost si microcms est le premier hôte virtuel que vous ajoutez à Apache, faute de quoi l'URL [http://localhost](http://localhost/) ne fonctionnera plus.

Redémarrez le serveur web Apache pour terminer la configuration de l'hôte virtuel. À ce stade, une requête HTTP vers [http://microcms](http://microcms/) doit afficher correctement la liste des articles.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-03).

Bilan

Nous avons refactorisé notre application web pour intégrer le framework Silex et posé les bases d'une architecture robuste. Cependant, notre application profite encore peu des services que Silex peut fournir. Les prochaines itérations y remédieront.

## ITÉRATION 4 : MODÉLISATION OBJET DE L'ACCÈS AUX DONNÉES

Le but de cette itération est d'améliorer la partie Modèle de notre application web.

### Modélisation objet du domaine

Actuellement, la partie Modèle de notre application web est écrite de manière simpliste. Voici pour rappel le fichier source model.php.

1<?php

2

3// Return all articles

4function getArticles() {

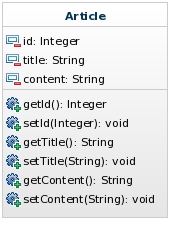
5    $bdd = new *PDO*('mysql:host=localhost;dbname=microcms;charset=utf8', 'microcms\_user', 'secret');

6    $articles = $bdd->query('select \* from t\_article order by art\_id desc');

7    return $articles;

8}

L'ajout futur de nouveaux services similaires risque de rendre la partie Modèle difficile à utiliser. Nous allons restructurer cette partie en introduisant une modélisation orientée objet des données métier. Pour l'instant, nos seules données métier sont les articles, qui se caractérisent par un identifiant, un titre et un contenu. Nous allons modéliser un article sous la forme d'une classe nommée Article dont voici le diagramme UML.

Diagramme UML de la classe Article

La classe Article  est ce qu'on appelle parfois (sans peur du ridicule) un POPO, ou \*Plain Old PHP Object\* par analogie avec les [POJO](http://fr.wikipedia.org/wiki/Plain_Old_Java_Object) du monde Java. Autrement dit, cette classe ne contient rien de complexe : uniquement les propriétés et accesseurs nécessaires pour représenter un article.

Écrivons maintenant cette classe en PHP.

1<?php

2

3namespace MicroCMS\Domain;

4

5class Article

6{

7    /\*\*

8     \* Article id.

9     \*

10     \* @var integer

11     \*/

12    private $id;

13

14    /\*\*

15     \* Article title.

16     \*

17     \* @var string

18     \*/

19    private $title;

20

21    /\*\*

22     \* Article content.

23     \*

24     \* @var string

25     \*/

26    private $content;

27

28    public function getId() {

29        return $this->id;

30    }

31

32    public function setId($id) {

33        $this->id = $id;

34    }

35

36    public function getTitle() {

37        return $this->title;

38    }

39

40    public function setTitle($title) {

41        $this->title = $title;

42    }

43

44    public function getContent() {

45        return $this->content;

46    }

47

48    public function setContent($content) {

49        $this->content = $content;

50    }

51}

Vous aurez peut-être remarqué que cette classe est définie dans un **espace de nommage** ou espace de noms (instruction PHP namespace au début du fichier). En PHP comme dans les autres langages qui les supportent, un espace de nommage permet d'éviter les conflits de nommage des éléments (exemple : deux classes portant le même nom) en regroupant ces éléments dans des espaces dédiés. Ainsi, le nom complet de notre classe Article est MicroCMS\Domain\Article. Pour plus d'informations sur les espaces de noms en PHP, consultez la [documentation du langage](http://php.net/manual/fr/language.namespaces.php) ou le [cours OpenClassrooms](http://fr.openclassrooms.com/informatique/cours/les-espaces-de-noms-en-php) sur ce thème.

Afin de permettre un chargement automatisé de la classe Article, le fichier source Article.php associé doit se trouver dans le répertoire correspondant à son espace de noms. Créez dans le répertoire src du projet le sous-répertoire Domain. Dans ce répertoire, créez le fichier Article.php et copiez-collez-y le contenu ci-dessus.

Afin d'éviter toute erreur future, veillez à bien respecter les distinctions majuscules/minuscules dans les noms des répertoires et des fichiers source.

### Remplacement de PDO par Doctrine DBAL

Il faut maintenant transformer notre partie Modèle afin qu'elle renvoie une liste d'objets de la classeArticle. Nous allons en profiter pour changer de technologie d'accès à la base de données en remplaçant PDO par Doctrine [DBAL](http://www.doctrine-project.org/projects/dbal.html).

Comme son nom l'indique, DBAL est une couche d'abstraction de base de données. Tout comme PDO, DBAL fournit des services de connexion et d'exécution de requêtes SQL indépendants du SGBDR utilisé : le même code pourra interagir avec MySQL, Oracle ou encore PostgreSQL. Par rapport à PDO, DBAL fournit des services additionnels comme la gestion avancée des transactions et des types. Le remplacement de PDO par DBAL va également permettre d'illustrer le mécanisme d'ajout de services proposé par le framework Silex.

#### Mise à jour de l'accès aux données

Écrivons le nouveau code d'accès aux données utilisant DBAL. Nous allons continuer à utiliser la POO en définissant ce code dans la classe ArticleDAO.

1<?php

2

3namespace MicroCMS\DAO;

4

5use Doctrine\DBAL\Connection;

6use MicroCMS\Domain\Article;

7

8class ArticleDAO

9{

10    /\*\*

11     \* Database connection

12     \*

13     \* @var \Doctrine\DBAL\Connection

14     \*/

15    private $db;

16

17    /\*\*

18     \* Constructor

19     \*

20     \* @param \Doctrine\DBAL\Connection The database connection object

21     \*/

22    public function \_\_construct(Connection $db) {

23        $this->db = $db;

24    }

25

26    /\*\*

27     \* Return a list of all articles, sorted by date (most recent first).

28     \*

29     \* @return array A list of all articles.

30     \*/

31    public function findAll() {

32        $sql = "select \* from t\_article order by art\_id desc";

33        $result = $this->db->fetchAll($sql);

34

35        // Convert query result to an array of domain objects

36        $articles = array();

37        foreach ($result as $row) {

38            $articleId = $row['art\_id'];

39            $articles[$articleId] = $this->buildArticle($row);

40        }

41        return $articles;

42    }

43

44    /\*\*

45     \* Creates an Article object based on a DB row.

46     \*

47     \* @param array $row The DB row containing Article data.

48     \* @return \MicroCMS\Domain\Article

49     \*/

50    private function buildArticle(array $row) {

51        $article = new *Article*();

52        $article->setId($row['art\_id']);

53        $article->setTitle($row['art\_title']);

54        $article->setContent($row['art\_content']);

55        return $article;

56    }

57}

Cette classe est définie dans l'espace de noms MicroCMS\DAO et utilise (instruction PHP use) les classesDoctrine\DBAL\Connection et MicroCMS\Domain\Article. Elle mémorise l'objet de connexion à la BD dans sa propriété $db, définie par son constructeur. L'ancienne fonction getArticles est remplacée par la méthode findAll qui renvoie la liste de tous les articles sous forme d'objets de la classe Article. La classe ArticleDAO dispose également d'une méthode interne buildArticle qui instancie un objet de la classe Article à partir d'une ligne de résultat SQL.

L'acronyme DAO signifie Data Access Object, ou objet d'accès aux données. Il s'agit d'un modèle de conception (design pattern) qui propose de regrouper le code d'accès aux données d'une application dans des classes dédiées. C'est le cas de notre classe ArticleDAO.

Créez le fichier source ArticleDAO.php dans le répertoire src/DAO (à créer) et ajoutez-lui le code source de la classe. Ensuite, supprimez le fichier src/model.php devenu inutile.

#### Mise à jour de l'application

Plusieurs étapes sont à réaliser avant de pouvoir utiliser le nouveau code d'accès aux données.

Tout d'abord, il faut indiquer à Composer que notre projet dépend maintenant de DBAL. Modifiez le fichiercomposer.json comme indiqué ci-dessous.

1{

2    "require": {

3        "silex/silex": "~1.2",

4        "doctrine/dbal": "~2.4"

5    },

6    "autoload": {

7        "psr-4": {"MicroCMS\\": "src"}

8    }

9}

La nouvelle entrée autoload dans ce fichier permet d'ajouter notre propre code source, défini dans le répertoire src, au mécanisme de chargement automatique (autoloading) géré par Composer. Finies les instructions require partout dans le code ! Pour que cela fonctionne, il faut que notre code source respecte le standard [PSR-4](http://www.php-fig.org/psr/psr-4/) – c'est le cas ici.

Dans une fenêtre de terminal positionnée dans le répertoire MicroCMS, lancez la commande ci-dessous afin que Composer récupère les fichiers du projet DBAL et mette à jour le fichier vendor/autoload.php.

composer update

Ensuite, créez dans le répertoire app du projet un nouveau fichier nommé app.php. Ce fichier va contenir le paramétrage de l'application Silex. Ajoutez dans ce fichier le code ci-dessous.

1<?php

2

3use Symfony\Component\Debug\ErrorHandler;

4use Symfony\Component\Debug\ExceptionHandler;

5

6// Register global error and exception handlers

7*ErrorHandler*::register();

8*ExceptionHandler*::register();

9

10// Register service providers.

11$app->register(new *Silex*\Provider\DoctrineServiceProvider());

12

13// Register services.

14$app['dao.article'] = $app->share(function ($app) {

15    return new *MicroCMS*\DAO\ArticleDAO($app['db']);

16});

La première partie de ce fichier configure Silex pour gérer les erreurs PHP qui pourraient se produire pendant l'exécution de l'application. Cela permet d'obtenir des messages d'erreur explicites. Vous trouverez plus d'explications dans la [documentation du framework](http://silex.sensiolabs.org/doc/cookbook/error_handler.html).

La deuxième partie du fichier enregistre le [fournisseur de services](http://silex.sensiolabs.org/doc/providers.html) associé à DBAL,[DoctrineServiceProvider](http://silex.sensiolabs.org/doc/providers/doctrine.html).

Enfin, la troisième partie enregistre un nouveau [service](http://silex.sensiolabs.org/doc/services.html) nommé dao.article sous la forme d'une instance partagée de la classe ArticleDAO. Une fois le service enregistré, l'appel$app['dao.article'] renverra cette instance.

Le service $app['db'] est défini automatiquement lors de l'enregistrement du fournisseurDoctrineServiceProvider.

Nous allons également améliorer la configuration de l'application. Pour cela, créez le fichierprod.php dans le sous-répertoire app/config du projet (à créer). Ce fichier contient les options de configuration liés à la mise en production de notre application. Ajoutez-lui le contenu ci-dessous.

1<?php

2

3// Doctrine (db)

4$app['db.options'] = array(

5    'driver'   => 'pdo\_mysql',

6    'charset'  => 'utf8',

7    'host'     => 'localhost',

8    'port'     => '3306',

9    'dbname'   => 'microcms',

10    'user'     => 'microcms\_user',

11    'password' => 'secret',

12);

Il s'agit du paramétrage de la connexion à la base de données via DBAL.

Créez le fichier dev.php dans le sous-répertoire app/config du projet. Ce fichier contient les options de configuration liés au développement de notre application. Ajoutez-lui le contenu ci-dessous.

1<?php

2

3// include the prod configuration

4require \_\_DIR\_\_.'/prod.php';

5

6// enable the debug mode

7$app['debug'] = true;

Ce fichier inclut la configuration de production puis paramètre Silex pour afficher des informations de débogage détaillées en cas d'erreur, ce qui est utile pendant la phase de développement.

À présent, modifiez le contrôleur défini dans le fichier app/routes.php comme indiqué ci-dessous.

1<?php

2

3// Home page

4$app->get('/', function () use ($app) {

5    $articles = $app['dao.article']->findAll();

6

7    ob\_start();             // start buffering HTML output

8    require '../views/view.php';

9    $view = ob\_get\_clean(); // assign HTML output to $view

10    return $view;

11});

L'appel à la fonction getArticles est remplacé par l'utilisation du service dao.article enregistré dansapp/app.php. L'appel à $app['dao.article'] renvoie un objet de la classe ArticleDAO dont on utilise ensuite la méthode findAll pour récupérer la liste des articles.

Ce contrôleur a maintenant besoin de l'objet application Silex $app, d'où l'ajout duuse($app) dans sa définition.

La variable $articles utilisée dans ce fichier source contient à présent un tableau d'objets de la classeArticle . Il faut donc modifier une partie de la vue views/view.php  en conséquence.

1// ...

2<?php foreach ($articles as $article): ?>

3<article>

4    <h2><?php echo $article->getTitle() ?></h2>

5    <p><?php echo $article->getContent() ?></p>

6</article>

7<?php endforeach ?>

8// ...

Chaque élément $article du tableau $articles est un objet de la classe Article, et non plus un tableau associatif. À l'intérieur de la boucle foreach qui parcourt la liste des articles, on utilise les méthodes getTitle et getContent de cette classe afin d'accéder aux données de l'article.

Il ne nous reste plus qu'à modifier le contrôleur frontal web/index.php afin d'inclure les fichiers de paramétrage de l'application.

1<?php

2

3require\_once \_\_DIR\_\_.'/../vendor/autoload.php';

4

5$app = new *Silex*\Application();

6

7require \_\_DIR\_\_.'/../app/config/dev.php';

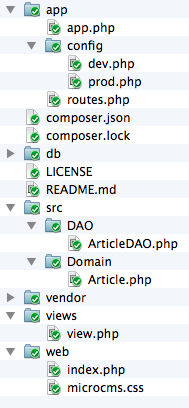
8require \_\_DIR\_\_.'/../app/app.php';

9require \_\_DIR\_\_.'/../app/routes.php';

10

11$app->run();

La structure de notre application est maintenant la suivante.

Arborescence du projet

C'est le moment de tester notre refactorisation en accédant à l'URL [http://microcms](http://microcms/). Si tout va bien, le résultat affiché reste le même.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-04).

### Bilan

Cette itération nous a permis d'introduire une modélisation orientée objet de l'accès aux données. Au passage, nous avons découvert comment Silex facilite l'inclusion de nouveaux services dans une application web. C'est l'un des avantages de l'utilisation d'un framework.

Dans cette itération, nous avons surtout travaillé dans les parties Modèle et Contrôleur de notre application. L'itération suivante s'intéressera à la partie Vue.

## ITÉRATION 5 : INTÉGRATION DU MOTEUR DE TEMPLATES TWIG

Le but de cette itération est d'améliorer la technologie d'affichage de notre application.

### Critique de l'application existante

La partie Présentation de notre application actuelle correspond au répertoire views, dans lequel on définit un fichier PHP/HTML par vue affichée. Voici pour mémoire le contenu du fichier view.php  affichant la liste des articles.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8" />

5    <link href="microcms.css" rel="stylesheet" />

6    <title>MicroCMS - Home</title>

7</head>

8<body>

9    <header>

10        <h1>MicroCMS</h1>

11    </header>

12    <?php foreach ($articles as $article): ?>

13    <article>

14        <h2><?php echo $article->getTitle() ?></h2>

15        <p><?php echo $article->getContent() ?></p>

16    </article>

17    <?php endforeach ?>

18    <footer class="footer">

19        <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

20    </footer>

21</body>

22</html>

L'évolution de l'application va impliquer l'écriture de nouvelles vues (détails sur un article, formulaire de connexion, etc.). Toutes les vues auront cependant des éléments communs (partie <head>, pied de page, etc.). On aimerait pouvoir définir un squelette de vue commun et ne définir dans chaque vue que ses éléments spécifiques.

Un autre problème de notre application concerne la sécurité. Actuellement, notre vue n'est pas protégée contre les [injections de code](http://fr.wikipedia.org/wiki/Injection_de_code_dans_les_applications_web) sur la page HTML générée. Ici, si l'instruction $article->getTitle()retourne un code malicieux (par exemple du JavaScript), la page HTML générée par cette vue exécutera ce code JavaScript non désiré.

Le mécanisme standard de défense contre les injections de code dans du HTML consiste à**échapper** (escape) toutes les données dynamiques incluses dans le code HTML. Avec PHP, la technique classique est de faire appel à la fonction [htmlspecialchars](http://php.net/manual/fr/function.htmlspecialchars.php) qui transforme certains caractères tels que <  ou> , ce qui désactive l'interprétation des balises comme <script>. Silex fournit ce mécanisme sous la forme de la méthode [escape](http://silex.sensiolabs.org/api/Silex/Application.html#method_escape). Cependant, il est fastidieux d'échapper systématiquement toutes les variables PHP dans toutes nos vues.

Afin de remédier à toutes ces limitations, nous allons utiliser une solution plus moderne en intégrant à notre application un moteur de templates.

### Choix du moteur de templates

Un **moteur de templates** est un logiciel spécialisé dans la générations de vues. Un template (que l'on peut traduire par "gabarit") est un fichier texte contenant des instructions spécifiques (variables, structures de contrôle, etc.). Lorsqu'un template est généré, les instructions spécifiques qu'il contient sont exécutées par le moteur pour aboutir au résultat désiré.

Quelque part, PHP est en lui-même un moteur de templates. Cependant, un moteur de templates dédié comme celui que nous allons utiliser fournit une syntaxe plus claire et des services plus avancés.

Il existe plusieurs moteurs de templates PHP, comme par exemple [Smarty](http://www.smarty.net/). Notre choix va se porter sur le moteur standard de Silex et de Symfony : [Twig](http://twig.sensiolabs.org/).

La prise en main de Twig n'est pas très difficile. Si vous souhaitez des informations détaillées, consultez sa[documentation](http://twig.sensiolabs.org/doc/templates.html) ou encore ce [cours OpenClassrooms](http://fr.openclassrooms.com/informatique/cours/utilisation-de-twig-un-moteur-de-templates).

### Intégration de Twig

L'intégration de Twig est facilitée par l'existence d'un fournisseur de services pour Silex. Nous allons donc employer la même technique que pour DBAL à l'itération précédente.

Commençons par ajouter une dépendance de notre projet envers Twig dans le fichier composer.json.

1{

2    "require": {

3        "silex/silex": "~1.2",

4        "doctrine/dbal": "~2.4",

5        "twig/twig": "~1.16"

6    },

7    "autoload": {

8        ...

9    }

10}

Mettez ensuite à jour les dépendances en tapant la ligne ci-dessous dans un terminal.

composer update

Ensuite, éditez le fichier app/app.php pour y ajouter l'enregistrement de Twig auprès de Silex.

1<?php

2

3use Symfony\Component\Debug\ErrorHandler;

4use Symfony\Component\Debug\ExceptionHandler;

5

6// Register global error and exception handlers

7*ErrorHandler*::register();

8*ExceptionHandler*::register();

9

10// Register service providers.

11$app->register(new *Silex*\Provider\DoctrineServiceProvider());

12$app->register(new *Silex*\Provider\TwigServiceProvider(), array(

13    'twig.path' => \_\_DIR\_\_.'/../views',

14));

15

16// Register services.

17$app['dao.article'] = $app->share(function ($app) {

18    return new *MicroCMS*\DAO\ArticleDAO($app['db']);

19});

Ici, Twig est configuré pour que le répertoire dans lequel nous stockerons nos templates soit le répertoireviews du projet.

Le service $app['twig'] est défini automatiquement lors de l'enregistrement du fournisseurTwigServiceProvider.

À présent, remplaçons la vue views/view.php par un template Twig pour afficher la liste des articles. Dans le répertoire views, créez le fichier index.html.twig avec le contenu ci-dessous.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8" />

5    <link href="microcms.css" rel="stylesheet" />

6    <title>MicroCMS - Home</title>

7</head>

8<body>

9    <header>

10        <h1>MicroCMS</h1>

11    </header>

12    {% for article in articles %}

13    <article>

14        <h2>{{ article.title }}</h2>

15        <p>{{ article.content }}</p>

16    </article>

17    {% endfor %}

18    <footer class="footer">

19        <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

20    </footer>

21</body>

22</html>

Comme vous le constatez, la syntaxe de Twig est assez proche de celle du PHP :

* la boucle for permet de parcourir le tableau $articles ;
* l'instruction article.title fait appel à la méthode getTitle de l'objet $article ([plus de précisions](http://twig.sensiolabs.org/doc/templates.html#variables)) ;

Par défaut, **l'échappement** des variables dynamiques qui sont insérées dans une vue est automatique.

Vous pouvez maintenant supprimer l'ancien fichier views/view.php devenu inutile.

Enfin, on modifie la route Silex dans le fichier app/routes.php pour générer la nouvelle vue.

1<?php

2

3// Home page

4$app->get('/', function () use ($app) {

5    $articles = $app['dao.article']->findAll();

6    return $app['twig']->render('index.html.twig', array('articles' => $articles));

7});

Ici, on demande au service Twig ($app['twig'] ) de générer le template index.html.twig en lui passant ses données dynamiques en paramètre. Ici, la seule donnée dynamique est une variable nomméearticles qui contient le tableau d'objets de la classe Article renvoyé par la partie Modèle.

Il est temps de vérifier l'intégration de Twig en accédant à [http://microcms](http://microcms/). Même si la technologie utilisée a changé, la liste des articles s'affiche exactement comme précédemment.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-05).

### Bilan

La partie Présentation de notre application web est maintenant gérée par le moteur de remplates Twig. Cependant, le rendu utilisateur n'a pas évolué depuis l'initialisation de l'application et il reste sommaire. La prochaine itération va améliorer cela.

## ITÉRATION 6 : INTÉGRATION DU FRAMEWORK WEB BOOTSTRAP

Le but de cette itération est de rendre l'affichage de notre application plus conforme aux standards actuels.

### Introduction au design web adaptatif

Commençons avec une image qui vaut mieux qu'un long discours.



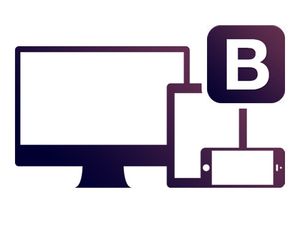
De nos jours, un site web a plus de chances d'être consulté depuis un terminal mobile (smartphone, tablette, etc.) que depuis un classique poste fixe. Cette (r)évolution doit absolument est prise en compte pour qu'un utilisateur de terminal mobile puisse consulter et utiliser le site dans de bonnes conditions.

Il existe plusieurs réponses complémentaires à cette problématique, comme par exemple la création d'une version mobile d'un site ou encore le développement d'applications dédiées à chaque écosystème mobile (iOS, Android, Windows Phone...). La solution la moins coûteuse consiste à adapter l'affichage en détectant les caractéristiques du terminal client. C'est ce qu'on appelle le **design web adaptatif**, traduction de l'anglais **responsive (web) design**. Pour une introduction générale à ce concept, consultez[Wikipedia](http://fr.wikipedia.org/wiki/Site_web_adaptatif).

Il existe plusieurs manières d'obtenir un design web adaptatif. On peut le réaliser "à la main" en utilisant des [media queries](http://www.alsacreations.com/article/lire/930-css3-media-queries.html) (fournies par la norme CSS3) pour adapter la mise en page à l'environnement détecté. Il existe également des frameworks qui facilitent la mise en page adaptative. Nous allons utiliser le plus populaire d'entre eux : Bootstrap.

### Présentation de Bootstrap

[Bootstrap](http://getbootstrap.com/) est un framework destiné à faciliter la création d'applications web. Il regroupe une collection d'outils fournis sous la forme de classes CSS et de librairies JavaScript. Bootstrap a été créé par deux développeurs du réseau social Twitter.



Il n'est pas nécessaire de connaître Bootstrap sur le bout des doigts pour suivre ce cours. Si vous n'avez jamais entendu parler de ce framework, je vous conseille tout de même de consulter le [cours OpenClassrooms](http://openclassrooms.com/courses/prenez-en-main-bootstrap) associé et de lire au minimum la partie 1 ("Premiers pas").

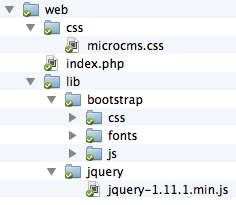
### Installation de Bootstrap et de jQuery

Les fichiers de Bootstrap sont nécessaires au navigateur client pour afficher les pages HTML de notre application. Nous allons donc les installer dans le répertoire web. Dans ce répertoire, créez un sous-répertoire lib puis un sous-répertoire bootstrap dans lib. Téléchargez Bootstrap sur [cette page](http://getbootstrap.com/getting-started/#download) puis décompressez le contenu de l'archive dans le répertoire bootstrap.

Pour fonctionner totalement, Bootstrap nécessite l'inclusion de la librairie JavaScript [jQuery](http://jquery.com/). Créez dans le répertoire web/lib un sous-répertoire jquery. Ensuite, téléchargez la version "production" de jQuery sur [cette page](http://jquery.com/download/) puis copiez le fichier JavaScript téléchargé dans le répertoire jquery.

Afin de clarifier l'organisation de web, créez dans ce répertoire un sous-répertoire css puis déplacez-y le fichier microcms.css.

Vous devez obtenir dans web une arborescence de la forme suivante.

Arborescence du répertoire web du projet

### Réécriture de la vue avec Bootstrap

Nous allons modifier la vue index.html.twig pour y intégrer Bootstrap. Au passage, nous allons en profiter pour ajouter à l'application une barre de navigation fixée en haut de la fenêtre.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8">

5    <meta http-equiv="X-UA-Compatible" content="IE=edge">

6    <meta name="viewport" content="width=device-width, initial-scale=1">

7    <link href="{{ app.request.basepath }}/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet">

8    <link href="{{ app.request.basepath }}/css/microcms.css" rel="stylesheet">

9    <title>MicroCMS - Home</title>

10</head>

11<body>

12    <div class="container">

13        <nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">

14            <div class="container">

15                <div class="navbar-header">

16                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#navbar-collapse-target">

17                        <span class="icon-bar"></span>

18                        <span class="icon-bar"></span>

19                        <span class="icon-bar"></span>

20                    </button>

21                    <a class="navbar-brand" href="/">MicroCMS</a>

22                </div>

23                <div class="collapse navbar-collapse" id="navbar-collapse-target">

24                </div>

25            </div><!-- /.container -->

26        </nav>

27        {% for article in articles %}

28        <article>

29            <h2>{{ article.title }}</h2>

30            <p>{{ article.content }}</p>

31        </article>

32        {% endfor %}

33        <footer class="footer">

34            <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

35        </footer>

36    </div>

37</body>

38</html>

Dans la partie <head> de cette page, nous avons ajouté les liens vers les feuilles de style de l'application et de Boostrap en préfixant ces liens par la variable app.request.basepath de Silex. Cela permet de gérer tous les scenarii possibles de déploiement de l'application ([plus de détails](http://silex.sensiolabs.org/doc/cookbook/assets.html)).

Dans le corps de cette page, nous avons introduit une barre de navigation (navbar) fixée en haut de la fenêtre (navbar-fixed-top). Tous les éléments du corps sont inclus dans un conteneur Boostrap (container).

Le fait de fixer la barre de navigation en haut de la fenêtre nécessite de décaler vers le bas la balise<body>. Il faut donc modifier la feuille style microcms.css ainsi :

1body {

2    *padding-top*: 40px;

3}

4

5.footer {

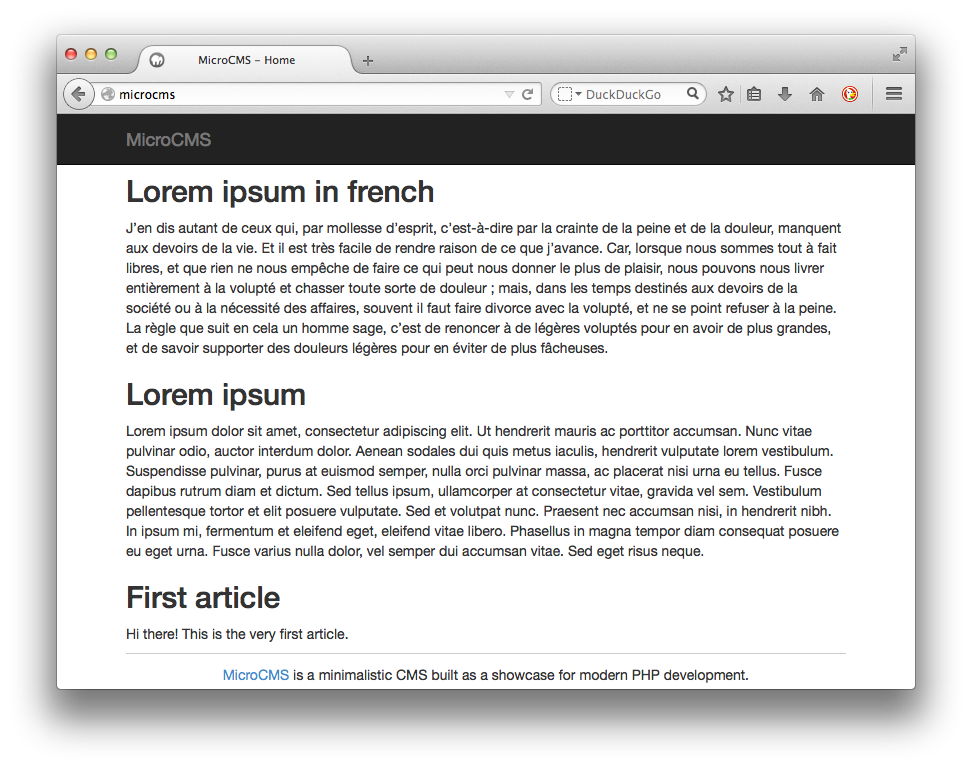
6    *border-top*: 1px solid #ccc;

7    *padding-top*: 10px;

8    *text-align*: center;

9}

Accédez à l'URL [http://microcms](http://microcms/) depuis votre navigateur Web. Vous obtenez à présent un résultat plus flatteur.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-06).

### Bilan

Grâce à l'intégration de Bootstrap, les vues gérant l'affichage de notre application ont un aspect plus actuel et peuvent être écrites de manière adaptative. Leur rendu sera optimal quel que soit le terminal client utilisé.

Cette itération et les précédentes ont consisté en des améliorations techniques de l'application. La prochaine itération va (enfin !) lui ajouter une nouvelle fonctionnalité métier.

ITÉRATION 7 : AFFICHAGE DES DÉTAILS SUR UN ARTICLE

Le but de cette itération est de permettre au visiteur de consulter les détails sur un article en cliquant sur son titre.

Mise à jour de la base de données

Notre base de données actuelle doit évoluer pour intégrer le stockage des commentaires sur les articles. Un commentaire se caractérise par son identifiant, son auteur, son contenu ainsi que l'article auquel il se rapporte.

Au cours de cette itération, nous allons gérer l'auteur comme une simple chaîne de caractères. Plus loin, nous découvrirons comment obliger les visiteurs à s'authentifier avant pouvoir de commenter un article. On crée donc une table t\_comment pour stocker les commentaires, en lui ajoutant les champs requis ainsi qu'une clé étrangère vers la table t\_article. On aboutit au contenu ci-dessous pour le fichierdb/structure.sql.

1drop table if exists t\_comment;

2drop table if exists t\_article;

3

4create table t\_article (

5    art\_id integer not null primary key auto\_increment,

6    art\_title varchar(100) not null,

7    art\_content varchar(2000) not null

8) engine=innodb character set utf8 collate utf8\_unicode\_ci;

9

10create table t\_comment (

11    com\_id integer not null primary key auto\_increment,

12    com\_author varchar(100) not null,

13    com\_content varchar(500) not null,

14    art\_id integer not null,

15    constraint fk\_com\_art foreign key(art\_id) references t\_article(art\_id)

16) engine=innodb character set utf8 collate utf8\_unicode\_ci;

On enrichit le jeu de données de test de l'application (fichier db/content.sql) en y ajoutant quelques commentaires.

1insert into t\_comment values

2(1, 'John Doe', 'Great! Keep up the good work.', 1);

3insert into t\_comment values

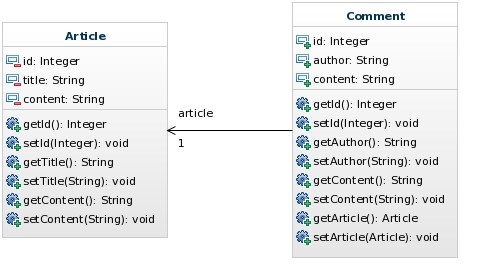
4(2, 'Ann Yone', "Thank you, I'll try my best.", 1);

Modifiez les scripts SQL comme indiqué ci-dessus, puis exécutez-les afin de mettre votre base de données à jour.

Mise à jour de l'application

**Partie Modèle**

En respectant les choix de modélisation objet effectués dans l'itération 4, on modélise un commentaire sous la forme d'une classe Comment dans l'espace de noms MicroCMS\Domain. Voici le diagramme UML associé.

Diagramme UML des classes Article et Comment

Ce diagramme modélise **l'association** entre un article et ses commentaires.

Créez le fichier source Comment.php dans le répertoire src/Domain, puis ajoutez-y le code source de la classe Comment.

1<?php

2

3namespace MicroCMS\Domain;

4

5class Comment

6{

7    /\*\*

8     \* Comment id.

9     \*

10     \* @var integer

11     \*/

12    private $id;

13

14    /\*\*

15     \* Comment author.

16     \*

17     \* @var string

18     \*/

19    private $author;

20

21    /\*\*

22     \* Comment content.

23     \*

24     \* @var integer

25     \*/

26    private $content;

27

28    /\*\*

29     \* Associated article.

30     \*

31     \* @var \MicroCMS\Domain\Article

32     \*/

33    private $article;

34

35    public function getId() {

36        return $this->id;

37    }

38

39    public function setId($id) {

40        $this->id = $id;

41    }

42

43    public function getAuthor() {

44        return $this->author;

45    }

46

47    public function setAuthor($author) {

48        $this->author = $author;

49    }

50

51    public function getContent() {

52        return $this->content;

53    }

54

55    public function setContent($content) {

56        $this->content = $content;

57    }

58

59    public function getArticle() {

60        return $this->article;

61    }

62

63    public function setArticle(Article $article) {

64        $this->article = $article;

65    }

66}

L'association avec un article se traduit dans le code source par la présence d'une propriété $article. Il ne s'agit pas d'un simple identifiant de type entier, mais bien d'un objet de la classe Article.

À présent, il faut créer la classe d'accès aux données pour les commentaires. Cette classe doit permettre de récupérer la liste des commentaires associés à un article donné. Elle est logiquement nomméeCommentDAO et se trouve dans l'espace de noms MicroCMS\DAO. Voici une première version de cette classe (à ne pas recopier pour l'instant).

1<?php

2

3namespace MicroCMS\DAO;

4

5use Doctrine\DBAL\Connection;

6use MicroCMS\Domain\Comment;

7

8class CommentDAO

9{

10    /\*\*

11     \* Database connection

12     \*

13     \* @var \Doctrine\DBAL\Connection

14     \*/

15    private $db;

16

17    /\*\*

18     \* Constructor

19     \*

20     \* @param \Doctrine\DBAL\Connection The database connection object

21     \*/

22    public function \_\_construct(Connection $db) {

23        $this->db = $db;

24    }

25

26    /\*\*

27     \* Return a list of all comments for an article, sorted by date (most recent first).

28     \*

29     \* @param integer $articleId The article id.

30     \*

31     \* @return array A list of all comments for the article.

32     \*/

33    public function findAllByArticle($articleId) {

34        // ...

35    }

36

37    /\*\*

38     \* Creates a Comment object based on a DB row.

39     \*

40     \* @param array $row The DB row containing Comment data.

41     \* @return \MicroCMS\Domain\Comment

42     \*/

43    private function buildComment($row) {

44        // ...

45    }

46}

On peut remarquer que la propriété $db ainsi que le constructeur sont exactement les mêmes que dans la classe ArticleDAO. Il s'agit d'une duplication de code.

La **duplication de code** est l'ennemie du bon développeur, et l'un des symptômes d'un design perfectible.

Nous allons profiter de cette itération pour refactoriser le code d'accès aux données afin de supprimer cette duplication de code. Commençons par identifier les besoins communs à toutes les classes d'accès aux données :

* la connexion à la base (propriété $db) ;
* la construction d'un objet du domaine à partir d'une ligne de résultat SQL (méthodesbuildArticle  et buildComment).

Nous factorisons ces besoin communs au sein d'une classe abstraite DAO dont **hériteront** toutes nos classes d'accès aux données. Si vous avez besoin de détails sur le concept d'héritage, consultez [ce cours](http://bpesquet.gitbooks.io/programmation-orientee-objet-csharp/content/chapters/02-principaux-concepts-objet.html). Voici le code source de la classe DAO.

1<?php

2

3namespace MicroCMS\DAO;

4

5use Doctrine\DBAL\Connection;

6

7abstract class DAO

8{

9    /\*\*

10     \* Database connection

11     \*

12     \* @var \Doctrine\DBAL\Connection

13     \*/

14    private $db;

15

16    /\*\*

17     \* Constructor

18     \*

19     \* @param \Doctrine\DBAL\Connection The database connection object

20     \*/

21    public function \_\_construct(Connection $db) {

22        $this->db = $db;

23    }

24

25    /\*\*

26     \* Grants access to the database connection object

27     \*

28     \* @return \Doctrine\DBAL\Connection The database connection object

29     \*/

30    protected function getDb() {

31        return $this->db;

32    }

33

34    /\*\*

35     \* Builds a domain object from a DB row.

36     \* Must be overridden by child classes.

37     \*/

38    protected abstract function buildDomainObject($row);

39}

La connexion à la base de données est encapsulée sous la forme d'une propriété privée $db et d'un accesseur protégé (donc accessible uniquement aux classes dérivées) getDb. La construction d'un objet du domaine est spécifique à chaque entité métier : on factorise donc uniquement la *déclaration* de ce service (méthode protégée buildDomainObject). Chaque classe d'accès aux données devra**redéfinir** cette méthode pour consstruire un objet du domaine particulier.

Créez le fichier source src/DAO.php, puis ajoutez-y le code source de la classe DAO.

L'existence de la classe abstraite DAO  nous permet de modifier la définition de la classeArticleDAO comme indiqué ci-dessous.

1<?php

2

3namespace MicroCMS\DAO;

4

5use MicroCMS\Domain\Article;

6

7class ArticleDAO extends DAO

8{

9    /\*\*

10     \* Return a list of all articles, sorted by date (most recent first).

11     \*

12     \* @return array A list of all articles.

13     \*/

14    public function findAll() {

15        $sql = "select \* from t\_article order by art\_id desc";

16        $result = $this->getDb()->fetchAll($sql);

17

18        // Convert query result to an array of domain objects

19        $articles = array();

20        foreach ($result as $row) {

21            $articleId = $row['art\_id'];

22            $articles[$articleId] = $this->buildDomainObject($row);

23        }

24        return $articles;

25    }

26

27    /\*\*

28     \* Creates an Article object based on a DB row.

29     \*

30     \* @param array $row The DB row containing Article data.

31     \* @return \MicroCMS\Domain\Article

32     \*/

33    protected function buildDomainObject($row) {

34        $article = new *Article*();

35        $article->setId($row['art\_id']);

36        $article->setTitle($row['art\_title']);

37        $article->setContent($row['art\_content']);

38        return $article;

39    }

40}

La classe ArticleDAO hérite (mot-clé extends) de la classe abstraite DAO. L'utilisation directe de la propriété $db est remplacée par l'appel de la méthode getDb définie dans  DAO. La méthodebuildArticle est remplacée par la méthode redéfinie buildDomainObject.

Nous aurions pu aller plus loin dans le redesign en factorisant dans la classe DAO la construction d'un objet du domaine générique. Cela aurait complexifié l'architecture au-delà du niveau de ce cours.

Voici maintenant le code source de la classe CommentDAO.

1<?php

2

3namespace MicroCMS\DAO;

4

5use MicroCMS\Domain\Comment;

6

7class CommentDAO extends DAO

8{

9    /\*\*

10     \* @var \MicroCMS\DAO\ArticleDAO

11     \*/

12    private $articleDAO;

13

14    public function setArticleDAO(ArticleDAO $articleDAO) {

15        $this->articleDAO = $articleDAO;

16    }

17

18    /\*\*

19     \* Return a list of all comments for an article, sorted by date (most recent last).

20     \*

21     \* @param integer $articleId The article id.

22     \*

23     \* @return array A list of all comments for the article.

24     \*/

25    public function findAllByArticle($articleId) {

26        // The associated article is retrieved only once

27        $article = $this->articleDAO->find($articleId);

28

29        // art\_id is not selected by the SQL query

30        // The article won't be retrieved during domain objet construction

31        $sql = "select com\_id, com\_content, com\_author from t\_comment where art\_id=? order by com\_id";

32        $result = $this->getDb()->fetchAll($sql, array($articleId));

33

34        // Convert query result to an array of domain objects

35        $comments = array();

36        foreach ($result as $row) {

37            $comId = $row['com\_id'];

38            $comment = $this->buildDomainObject($row);

39            // The associated article is defined for the constructed comment

40            $comment->setArticle($article);

41            $comments[$comId] = $comment;

42        }

43        return $comments;

44    }

45

46    /\*\*

47     \* Creates an Comment object based on a DB row.

48     \*

49     \* @param array $row The DB row containing Comment data.

50     \* @return \MicroCMS\Domain\Comment

51     \*/

52    protected function buildDomainObject($row) {

53        $comment = new *Comment*();

54        $comment->setId($row['com\_id']);

55        $comment->setContent($row['com\_content']);

56        $comment->setAuthor($row['com\_author']);

57

58        if (array\_key\_exists('art\_id', $row)) {

59            // Find and set the associated article

60            $articleId = $row['art\_id'];

61            $article = $this->articleDAO->find($articleId);

62            $comment->setArticle($article);

63        }

64

65        return $comment;

66    }

67}

Afin de pouvoir construire complètement une instance de la classe Comment, la classe CommentDAO  doit pouvoir récupérer un article à partir de son identifiant et construire une instance de la classe Article. Plutôt que d'ajouter cela dans le code source de CommentDAO, on ajoute dans la classe ArticleDAO une nouvelle méthode find définissant le service requis. La classe CommentDAO a besoin de ce service pour fonctionner : on dit qu'il existe une **dépendance** entre la classe CommentDAO et la classe ArticleDAO. Cette dépendance se traduit dans le code source de CommentDAO par la présence d'une propriété privée$articleDAO et d'un accesseur en écriture (mutateur) setArticleDAO.

Dans la méthode protégée buildDomainObject, on ne construit l'article associé au commentaire que si le champ art\_id est présent dans la ligne de résultat SQL. Cela permet  de ne construire l'objet Article qu'une seule fois dans la méthode findAllByArticles, ce qui limite le nombre de requêtes SQL.

Créez le fichier source src/DAO/CommentDAO.php et ajoutez-y le code de la classe CommentDAO. Ensuite, ajoutez au fichier ArticleDAO.php la méthode find ci-dessous.

1<?php

2

3// ...

4

5class ArticleDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Returns an article matching the supplied id.

11     \*

12     \* @param integer $id

13     \*

14     \* @return \MicroCMS\Domain\Article|throws an exception if no matching article is found

15     \*/

16    public function find($id) {

17        $sql = "select \* from t\_article where art\_id=?";

18        $row = $this->getDb()->fetchAssoc($sql, array($id));

19

20        if ($row)

21            return $this->buildDomainObject($row);

22        else

23            throw new \Exception("No article matching id " . $id);

24    }

25

26    // ...

Notre refactorisation de la partie Modèle est maintenant terminée.

**Partie Vue**

L'évolution de cette partie consiste à ajouter une vue affichant les détails sur un article : titre, contenu et liste des commentaires. Cela se traduit par l'ajout d'un nouveau template article.html.twig  dans le répertoire views. Voici une première version de ce nouveau template (à ne pas recopier pour l'instant).

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8">

5    <meta http-equiv="X-UA-Compatible" content="IE=edge">

6    <meta name="viewport" content="width=device-width, initial-scale=1">

7    <link href="{{ app.request.basepath }}/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet">

8    <link href="{{ app.request.basepath }}/css/microcms.css" rel="stylesheet">

9

10    <title>MicroCMS - {{ article.title }}</title>

11</head>

12<body>

13    <div class="container">

14        <nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">

15            <div class="container">

16                <div class="navbar-header">

17                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#navbar-collapse-target">

18                        <span class="icon-bar"></span>

19                        <span class="icon-bar"></span>

20                        <span class="icon-bar"></span>

21                    </button>

22                    <a class="navbar-brand" href="/">MicroCMS</a>

23                </div>

24                <div class="collapse navbar-collapse" id="navbar-collapse-target">

25                </div>

26            </div><!-- /.container -->

27        </nav>

28        <p>

29            <h2>{{ article.title }}</h2>

30            <p>{{ article.content }}</p>

31            <h3>Comments</h3>

32            {% for comment in comments %}

33                <strong>{{ comment.author }}</strong> said : {{ comment.content }}<br>

34            {% else %}

35                No comments yet.

36            {% endfor %}

37        </p>

38        <footer class="footer">

39            <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

40        </footer>

41    </div>

42</body>

43</html>

Ce template utilise la structure de contrôle Twig for associée à un else si aucune itération de boucle n'est réalisée (absence de tout commentaire).

Le template existant index.html.twig et le nouveau template article.html.twig partagent de nombreux éléments : partie <head>, barre de navigation, pied de page... Pour éviter la duplication de code, on aimerait centraliser la définition de ces éléments et les inclure dans nos templates.

Le moteur de templates Twig permet de faire encore mieux : il supporte le mécanisme d'[héritage de templates](http://twig.sensiolabs.org/doc/templates.html#template-inheritance). Cela permet de définir un template de base contenant les éléments communs, puis de créer chaque template spécifique par **héritage** du template commun.

Créez dans le répertoire views un fichier texte layout.html.twig qui sera notre template commun :

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8">

5    <meta http-equiv="X-UA-Compatible" content="IE=edge">

6    <meta name="viewport" content="width=device-width, initial-scale=1">

7    <link href="{{ app.request.basepath }}/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet">

8    <link href="{{ app.request.basepath }}/css/microcms.css" rel="stylesheet">

9    <title>MicroCMS - {% block title %}{% endblock %}</title>

10</head>

11<body>

12    <div class="container">

13        <nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">

14            <div class="container">

15                <div class="navbar-header">

16                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#navbar-collapse-target">

17                        <span class="icon-bar"></span>

18                        <span class="icon-bar"></span>

19                        <span class="icon-bar"></span>

20                    </button>

21                    <a class="navbar-brand" href="/">MicroCMS</a>

22                </div>

23                <div class="collapse navbar-collapse" id="navbar-collapse-target">

24                </div>

25            </div><!-- /.container -->

26        </nav>

27        <div id="content">{% block content %}{% endblock %}</div>

28        <footer class="footer">

29            <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

30        </footer>

31    </div>

32</body>

33</html>

Ce template définit deux blocs, appelés title et  content. Les templates dérivés redéfinissent ces blocs afin d'ajouter les parties spécifiques de chaque vue. Voici la nouvelle définition du fichierindex.html.twig qui affiche la liste des articles.

1{% extends "layout.html.twig" %}

2

3{% block title %}Home{% endblock %}

4

5{% block content %}

6{% for article in articles %}

7<article>

8    <h2><a class="articleTitle" href="/article/{{ article.id }}">{{ article.title }}</a></h2>

9    <p>{{ article.content }}</p>

10</article>

11{% endfor %}

12{% endblock %}

On constate l'utilisation du mot-clé extends  pour indiquer que index.html.twig hérite du template commun layout.html.twig. Le reste du template définit les valeurs des blocs title et content. Un lien (balise <a>) ajouté au titre de chaque article renvoie vers une URL du type /article/<identifiant de l'article>.

Le template article.html.twig suit le même modèle.

1{% extends "layout.html.twig" %}

2

3{% block title %}{{ article.title }}{% endblock %}

4

5{% block content %}

6<p>

7    <h2>{{ article.title }}</h2>

8    <p>{{ article.content }}</p>

9    <h3>Comments</h3>

10    {% for comment in comments %}

11        <strong>{{ comment.author }}</strong> said : {{ comment.content }}<br>

12    {% else %}

13        No comments yet.

14    {% endfor %}

15</p>

16{% endblock %}

La dernière modification de la partie Vue consiste à enrichir légèrement la feuille de styleweb/css/microcms.css pour améliorer la présentation des titres d'article cliquables. Ajoutez le code ci-dessous à la fin de ce fichier.

1.articleTitle:hover, .articleTitle:focus {

2    *text-decoration*: none;

3}

**Partie Contrôleur**

La partie Contrôleur de notre application fait le lien entre le Modèle et la Vue. On commence par mettre à jour le fichier app/app.php afin d'enregistrer le nouveau service d'accès aux commentaires. Ajoutez le code ci-dessous à la fin de ce fichier.

1<?php

2

3// ...

4

5$app['dao.comment'] = $app->share(function ($app) {

6    $commentDAO = new *MicroCMS*\DAO\CommentDAO($app['db']);

7    $commentDAO->setArticleDAO($app['dao.article']);

8    return $commentDAO;

9});

C'est dans ce fichier que la dépendance envers la classe ArticleDAO est **injectée** à l'instance deCommentDAO grâce au mutateur setArticleDAO.

Enfin, on fait évoluer le fichier app/routes.php afin d'ajouter une nouvelle route.

1<?php

2

3// ...

4

5// Detailed info about an article

6$app->get('/article/{id}', function ($id) use ($app) {

7    $article = $app['dao.article']->find($id);

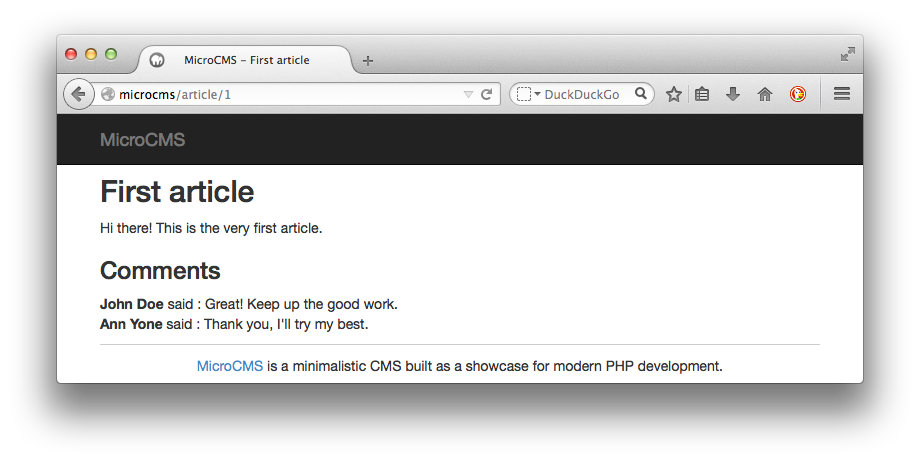
8    $comments = $app['dao.comment']->findAllByArticle($id);

9    return $app['twig']->render('article.html.twig', array('article' => $article, 'comments' => $comments));

10});

Le contrôleur associé à la nouvelle route génère le template article.html.twig en lui passant en paramètres les données nécessaires, récupérées depuis les services de la partie Modèle : l'article identifié par le paramètre $id présent dans l'URL et la liste des commentaires associés à cet article.

Cette longue itération touche à sa fin et il est temps de tester l'application. Ouvrez l'URL [http://microcms](http://microcms/)pour afficher la liste des articles, puis cliquez sur le titre de l'article du bas. Vous devriez obtenir l'affichage de ses détails.



Si vous cliquez sur le titre d'un article sans aucun commentaire, vous obtenez un résultat de la forme suivante.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-07).

Bilan

Au cours de cette itération, nous avons ajouté à l'application une fonctionnalité métier en nous appuyant sur les bases définies précédemment. Nous avons également saisi les occasions de refactoriser l'architecture afin qu'elle s'adapte aux nouveaux besoins.

Afin de réaliser de nouvelles fonctionnalités métier, l'itération suivante va s'intéresser à la sécurisation de l'application.

ITÉRATION 8 : GESTION DE LA SÉCURITÉ

Le but de cette itération est d'offrir aux visiteurs la possibilité de s'identifier afin d'être reconnus par l'application.

Contexte métier

Nous souhaitons ajouter à notre CMS une fonctionnalité d'ajout de commentaires à un article. Cependant, cet ajout ne doit être possible que pour les utilisateurs enregistrés dans l'application. Tout commentaire sera associé à son auteur, qui est nécessairement un utilisateur enregistré.

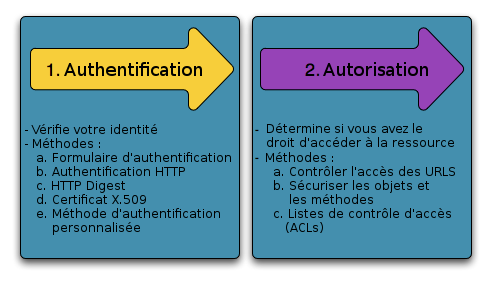
Afin de pouvoir réaliser cette fonctionnalité, il est nécessaire de sécuriser notre application web. Pour cela, nous allons tirer parti des possibilités du framework Silex, reprises de celles de son grand frère Symfony.

Symfony et la sécurité

La sécurisation est un besoin récurrent des applications web. Comme tous les frameworks majeurs, Symfony dispose de fonctionnalités avancées dans ce domaine. Ce paragraphe en fait un bref résumé. Pour plus de détails sur la sécurité avec Symfony/Silex, consultez les rubriques associées des documentations de [Symfony](http://symfony.com/doc/current/book/security.html) et de [Silex](http://silex.sensiolabs.org/doc/providers/security.html).

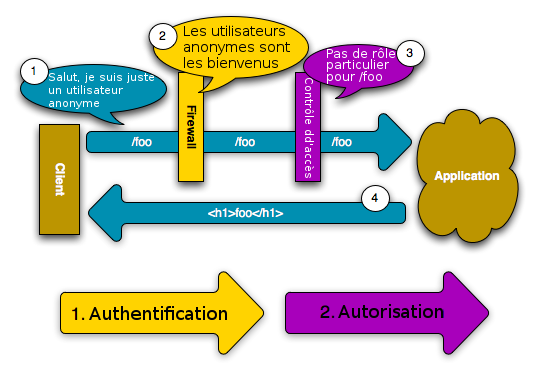
Symfony envisage la sécurité comme un processus en deux étapes :

1. **L'authentification**. Durant cette étape, l'utilisateur s'identifie auprès de l'application. Celle-ci tente ensuite de le reconnaître.
2. **L'autorisation**. Ici, l'application détermine si l'utilisateur reconnu a accès à la ressource qu'il demande.

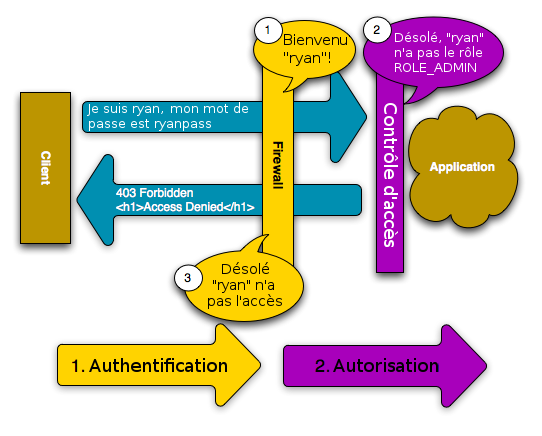


Ce schéma et les suivants sont extraits de la documentation du framework Symfony.

Symfony permet de sécuriser les ressources d'une application web en définissant un **pare-feu***(firewall)*. Lorsqu'un utilisateur fait une requête à une URL qui est protégée par un pare-feu, le système de sécurité de Symfony est activé. Le rôle du pare-feu est de déterminer si un utilisateur doit ou ne doit pas être authentifié (selon sa configuration, un pare-feu peut autoriser ou non les utilisateurs anonymes), et s'il doit l'être, de retourner une réponse à l'utilisateur afin d'entamer le processus d'authentification. Cette authentification peut prendre différentes formes : saisie d'un couple login/mot de passe dans un formulaire web (la plus courante), certificat, etc.

Accès d'un utilisateur anonyme à l'application

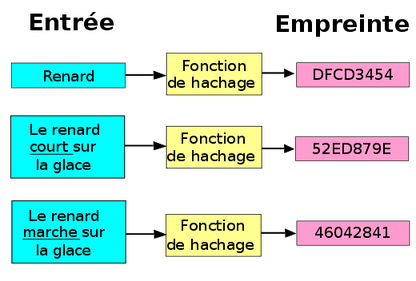
L'autorisation se base sur l'attribution de **rôles** aux utilisateurs reconnus. Dans la configuration du pare-feu, on peut soumettre l'accès à certaines ressources à la possession par l'utilisateur du rôle associé.

Refus d'accès à une ressource nécessitant le rôle d'administrateur

Gestion des mots de passe

La forme la plus courante d'authentification, celle que nous allons adopter, consiste à attribuer à chaque utilisateur un login et un mot de passe. Ces identifiants *(credentials)* permettent à l'application de reconnaître l'utilisateur.

Comme l'actualité nous le rappelle souvent, la gestion des mots de passe revêt une importante critique. Il est essentiel qu'une application web stocke et manipule ses mots de passe sous forme cryptée et non directement "en clair". Les mots de passe de nos utilisateurs seront donc stockés après application d'un[algorithme de hachage cryptographique](http://fr.wikipedia.org/wiki/Fonction_de_hachage). Par exemple, le mot Baptiste donne le résultat 04ce34a463c52d41c4d0c04c9afd0abe après application de l'algorithme de hachage [MD5](http://fr.wikipedia.org/wiki/MD5). Le résultat d'une opération de hachage est appelée empreinte.

Hachage de différents textes (source : Wikipedia)

Les algorithmes de hachage ont la particularité d'être **unidirectionnels** : il n'existe pas de moyen de revenir de l'empreinte obtenue au mot de passe en clair initial. Le mot de passe saisi par un utilisateur sera immédiatement haché avec le même algorithme, puis comparé avec la valeur cryptée stockée : si les deux résultats sont identiques, c'est que l'utilisateur a saisi le bon mot de passe.

Même si nos mots de passe sont stockés sous forme hachée, le risque subsiste qu'un individu mal intentionné arrive à s'introduire dans la base de données pour dérober ces mots de passe cryptés. Il pourrait ensuite utiliser une solution de type force brute pour hacher un très grand nombre de mots de passe et comparer le résultat avec les mots de passe volés jusqu'à trouver une correspondance.

Le [salage](http://fr.wikipedia.org/wiki/Salage_(cryptographie)) est une solution pour limiter ce risque. Cette technique consiste à ajouter plusieurs caractères au mot de passe juste avant de le hacher. Le résultat du hachage est différent de celui obtenu avec le mot de passe seul. Cela permet de protéger les mots de passe contre les attaques de type [dictionnaire](http://fr.wikipedia.org/wiki/Attaque_par_dictionnaire) (force brute avec utilisation d'une liste de mots de passe potentiels). Les données ajoutés au mot de passe sont appelées *salt*. Pour plus de sécurité, le *salt* doit être différent pour chaque mot de passe.

Pour plus de détails concernant les bonnes pratiques de cryptage des mots de passe, consultez [cet article](http://www.jasypt.org/howtoencryptuserpasswords.html).

Sécurisation de l'application

**Base de données**

La structure de notre base de données doit évoluer afin de refléter les nouveaux besoins métier : un commentaire est maintenant associé à un utilisateur de l'application. Voici le nouveau script SQLdb/structure.sql qui permet de créer la base.

1drop table if exists t\_comment;

2drop table if exists t\_user;

3drop table if exists t\_article;

4

5create table t\_article (

6    art\_id integer not null primary key auto\_increment,

7    art\_title varchar(100) not null,

8    art\_content varchar(2000) not null

9) engine=innodb character set utf8 collate utf8\_unicode\_ci;

10

11create table t\_user (

12    usr\_id integer not null primary key auto\_increment,

13    usr\_name varchar(50) not null,

14    usr\_password varchar(88) not null,

15    usr\_salt varchar(23) not null,

16    usr\_role varchar(50) not null

17) engine=innodb character set utf8 collate utf8\_unicode\_ci;

18

19create table t\_comment (

20    com\_id integer not null primary key auto\_increment,

21    com\_content varchar(500) not null,

22    art\_id integer not null,

23    usr\_id integer not null,

24    constraint fk\_com\_art foreign key(art\_id) references t\_article(art\_id),

25    constraint fk\_com\_usr foreign key(usr\_id) references t\_user(usr\_id)

26) engine=innodb character set utf8 collate utf8\_unicode\_ci;

Les utilisateurs sont stockés dans la table t\_user. Voici la description de ses champs.

* usr\_id est l'identifiant de l'utilisateur.
* usr\_name est le nom de l'utilisateur. Il sera utilisé commé login.
* usr\_password est son mot de passé, stockée sous forme hachée.
* usr\_salt est le salage utilisé pour hacher le mot de passe. Il est stocké en clair.
* usr\_role est le rôle attribué à l'utilisateur. Il sera utilisé lors de la phase d'autorisation.

On observe également que la table t\_comment contient maintenant une clé étrangère vers la tablet\_user , afin de matérialiser le lien entre un commentaire et son auteur.

Les données de test sont également mises à jour. Voici le script db/content.sql  associé.

1insert into t\_article values

2(1, 'First article', 'Hi there! This is the very first article.');

3insert into t\_article values

4(2, 'Lorem ipsum', 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut hendrerit mauris ac porttitor accumsan. Nunc vitae pulvinar odio, auctor interdum dolor. Aenean sodales dui quis metus iaculis, hendrerit vulputate lorem vestibulum. Suspendisse pulvinar, purus at euismod semper, nulla orci pulvinar massa, ac placerat nisi urna eu tellus. Fusce dapibus rutrum diam et dictum. Sed tellus ipsum, ullamcorper at consectetur vitae, gravida vel sem. Vestibulum pellentesque tortor et elit posuere vulputate. Sed et volutpat nunc. Praesent nec accumsan nisi, in hendrerit nibh. In ipsum mi, fermentum et eleifend eget, eleifend vitae libero. Phasellus in magna tempor diam consequat posuere eu eget urna. Fusce varius nulla dolor, vel semper dui accumsan vitae. Sed eget risus neque.');

5insert into t\_article values

6(3, 'Lorem ipsum in french', "J’en dis autant de ceux qui, par mollesse d’esprit, c’est-à-dire par la crainte de la peine et de la douleur, manquent aux devoirs de la vie. Et il est très facile de rendre raison de ce que j’avance. Car, lorsque nous sommes tout à fait libres, et que rien ne nous empêche de faire ce qui peut nous donner le plus de plaisir, nous pouvons nous livrer entièrement à la volupté et chasser toute sorte de douleur ; mais, dans les temps destinés aux devoirs de la société ou à la nécessité des affaires, souvent il faut faire divorce avec la volupté, et ne se point refuser à la peine. La règle que suit en cela un homme sage, c’est de renoncer à de légères voluptés pour en avoir de plus grandes, et de savoir supporter des douleurs légères pour en éviter de plus fâcheuses.");

7

8/\* raw password is 'john' \*/

9insert into t\_user values

10(1, 'JohnDoe', 'L2nNR5hIcinaJkKR+j4baYaZjcHS0c3WX2gjYF6Tmgl1Bs+C9Qbr+69X8eQwXDvw0vp73PrcSeT0bGEW5+T2hA==', 'YcM=A$nsYzkyeDVjEUa7W9K', 'ROLE\_USER');

11/\* raw password is 'jane' \*/

12insert into t\_user values

13(2, 'JaneDoe', 'EfakNLxyhHy2hVJlxDmVNl1pmgjUZl99gtQ+V3mxSeD8IjeZJ8abnFIpw9QNahwAlEaXBiQUBLXKWRzOmSr8HQ==', 'dhMTBkzwDKxnD;4KNs,4ENy', 'ROLE\_USER');

14

15insert into t\_comment values

16(1, 'Great! Keep up the good work.', 1, 1);

17insert into t\_comment values

18(2, "Thank you, I'll try my best.", 1, 2);

Ce jeu de données insère dans la base les utilisateurs 'JohnDoe' (mot de passe : 'john') et 'JaneDoe' (mot de passe : 'jane'). Pour chaque utilisateur, un salage a été généré aléatoirement puis l'algorithme de hachage par défaut de Symfony a été utilisé pour générer le mot de passe crypté. Celui-ci est stocké dans la base. On attribue à tous les utilisateurs le rôle ROLE\_USER (rôle par défaut pour Symfony).

Exécutez structure.sql puis content.sql afin de mettre à jour votre base de données.

**Composants Symfony**

Afin d'exploiter les fonctionnalités offertes par Symfony, nous devons récupérer les composants nécessaires. Il suffit pour cela de les déclarer dans le fichier de dépendances composer.json.

Le composant security regroupe les services de gestion de la sécurité. Le composant twig-bridge permet d'accéder à certains services Symfony depuis les templates Twig.

1"require": {

2    ...

3    "symfony/security": "~2.4",

4    "symfony/twig-bridge": "~2.4"

5}

6...

Une fois ce fichier modifié, on utilise Composer pour télécharger ces composants et leurs éventuelles dépendances.

composer update

**Partie Modèle**

La première étape de notre travail dans cette partie est de modéliser un utilisateur de l'application sous la forme d'une classe User située dans l'espace de noms MicroCMS\Domain. Voici son code source. Créez cette classe dans le fichier source src/Domain/User.php.

1<?php

2

3namespace MicroCMS\Domain;

4

5use Symfony\Component\Security\Core\User\UserInterface;

6

7class User implements UserInterface

8{

9    /\*\*

10     \* User id.

11     \*

12     \* @var integer

13     \*/

14    private $id;

15

16    /\*\*

17     \* User name.

18     \*

19     \* @var string

20     \*/

21    private $username;

22

23    /\*\*

24     \* User password.

25     \*

26     \* @var string

27     \*/

28    private $password;

29

30    /\*\*

31     \* Salt that was originally used to encode the password.

32     \*

33     \* @var string

34     \*/

35    private $salt;

36

37    /\*\*

38     \* Role.

39     \* Values : ROLE\_USER or ROLE\_ADMIN.

40     \*

41     \* @var string

42     \*/

43    private $role;

44

45    public function getId() {

46        return $this->id;

47    }

48

49    public function setId($id) {

50        $this->id = $id;

51    }

52

53    /\*\*

54     \* @inheritDoc

55     \*/

56    public function getUsername() {

57        return $this->username;

58    }

59

60    public function setUsername($username) {

61        $this->username = $username;

62    }

63

64    /\*\*

65     \* @inheritDoc

66     \*/

67    public function getPassword() {

68        return $this->password;

69    }

70

71    public function setPassword($password) {

72        $this->password = $password;

73    }

74

75    /\*\*

76     \* @inheritDoc

77     \*/

78    public function getSalt()

79    {

80        return $this->salt;

81    }

82

83    public function setSalt($salt)

84    {

85        $this->salt = $salt;

86    }

87

88    public function getRole()

89    {

90        return $this->role;

91    }

92

93    public function setRole($role) {

94        $this->role = $role;

95    }

96

97    /\*\*

98     \* @inheritDoc

99     \*/

100    public function getRoles()

101    {

102        return array($this->getRole());

103    }

104

105    /\*\*

106     \* @inheritDoc

107     \*/

108    public function eraseCredentials() {

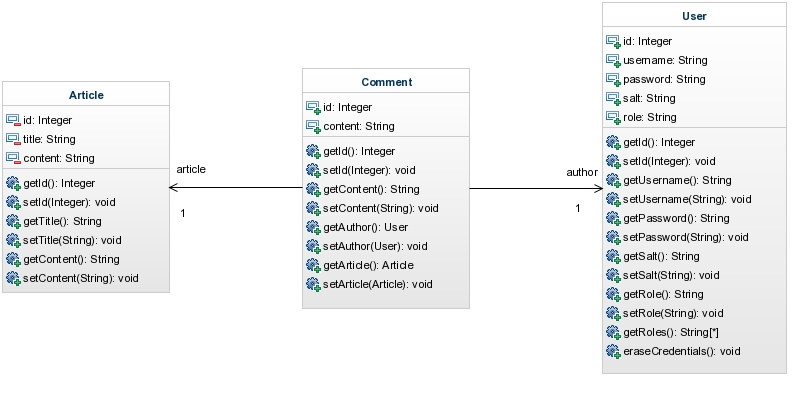
109        // Nothing to do here

110    }

111}

On constate une différence importante avec les classes métier Article et Comment : la classeUser implémente l'interface Symfony [UserInterface](http://api.symfony.com/2.6/Symfony/Component/Security/Core/User/UserInterface.html) et définit les méthodes présentes dans cette interface (ces méthodes sont identifiées par des @inheritDoc dans les commentaires de la classe User). Ces méthodes sont indispensables pour que l'utilisateur puisse être authentifié et autorisé par Symfony.

L'auteur d'un commentaire est maintenant un utilisateur enregistré. On modélise cela par une association entre les classes du domaine Comment et User.

Diagramme de classes UML du domaine

La classe Comment subit un changement mineur du point de vue du code source, mais important pour son utilisation : sa propriété $author n'est plus une chaîne de caractères, mais une instance de la classe User. On met à jour le commentaire et le mutateur associés pour refléter cette évolution.

1<?php

2

3namespace MicroCMS\Domain;

4

5class Comment

6{

7    // ...

8

9    /\*\*

10     \* Comment author.

11     \*

12     \* @var \MicroCMS\Domain\User

13     \*/

14    private $author;

15

16    // ...

17

18    public function setAuthor(User $author) {

19        $this->author = $author;

20    }

21

22    // ...

Nous devons également créer dans le fichier src/DAO/UserDAO.php la classe UserDAO qui gère l'accès aux utilisateurs.

1<?php

2

3namespace MicroCMS\DAO;

4

5use Symfony\Component\Security\Core\User\UserInterface;

6use Symfony\Component\Security\Core\User\UserProviderInterface;

7use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;

8use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

9use MicroCMS\Domain\User;

10

11class UserDAO extends DAO implements UserProviderInterface

12{

13    /\*\*

14     \* Returns a user matching the supplied id.

15     \*

16     \* @param integer $id The user id.

17     \*

18     \* @return \MicroCMS\Domain\User|throws an exception if no matching user is found

19     \*/

20    public function find($id) {

21        $sql = "select \* from t\_user where usr\_id=?";

22        $row = $this->getDb()->fetchAssoc($sql, array($id));

23

24        if ($row)

25            return $this->buildDomainObject($row);

26        else

27            throw new \Exception("No user matching id " . $id);

28    }

29

30    /\*\*

31     \* {@inheritDoc}

32     \*/

33    public function loadUserByUsername($username)

34    {

35        $sql = "select \* from t\_user where usr\_name=?";

36        $row = $this->getDb()->fetchAssoc($sql, array($username));

37

38        if ($row)

39            return $this->buildDomainObject($row);

40        else

41            throw new *UsernameNotFoundException*(sprintf('User "%s" not found.', $username));

42    }

43

44    /\*\*

45     \* {@inheritDoc}

46     \*/

47    public function refreshUser(UserInterface $user)

48    {

49        $class = get\_class($user);

50        if (!$this->supportsClass($class)) {

51            throw new *UnsupportedUserException*(sprintf('Instances of "%s" are not supported.', $class));

52        }

53        return $this->loadUserByUsername($user->getUsername());

54    }

55

56    /\*\*

57     \* {@inheritDoc}

58     \*/

59    public function supportsClass($class)

60    {

61        return 'MicroCMS\Domain\User' === $class;

62    }

63

64    /\*\*

65     \* Creates a User object based on a DB row.

66     \*

67     \* @param array $row The DB row containing User data.

68     \* @return \MicroCMS\Domain\User

69     \*/

70    protected function buildDomainObject($row) {

71        $user = new *User*();

72        $user->setId($row['usr\_id']);

73        $user->setUsername($row['usr\_name']);

74        $user->setPassword($row['usr\_password']);

75        $user->setSalt($row['usr\_salt']);

76        $user->setRole($row['usr\_role']);

77        return $user;

78    }

79}

Cette classe reprend la structure de nos classes DAO existantes et implémente l'interface Symfony[UserProviderInterface](http://api.symfony.com/2.6/Symfony/Component/Security/Core/User/UserProviderInterface.html). Cette interface contient les méthodes nécessaires pour qu'une classe puisse être utilisée comme **fournisseur de données utilisateur** par le composant de gestion de la sécurité de Symfony au cours du processus d'authentification.

La classe CommentDAO est mise à jour : elle dépend maintenant de la classe UserDAO pour construire un objet Comment complet à partir d'un résultat de requête SQL.

1<?php

2

3namespace MicroCMS\DAO;

4

5use MicroCMS\Domain\Comment;

6

7class CommentDAO extends DAO

8{

9    /\*\*

10     \* @var \MicroCMS\DAO\ArticleDAO

11     \*/

12    private $articleDAO;

13

14    /\*\*

15     \* @var \MicroCMS\DAO\UserDAO

16     \*/

17    private $userDAO;

18

19    public function setArticleDAO(ArticleDAO $articleDAO) {

20        $this->articleDAO = $articleDAO;

21    }

22

23    public function setUserDAO($userDAO) {

24        $this->userDAO = $userDAO;

25    }

26

27    /\*\*

28     \* Return a list of all comments for an article, sorted by date (most recent last).

29     \*

30     \* @param integer $articleId The article id.

31     \*

32     \* @return array A list of all comments for the article.

33     \*/

34    public function findAllByArticle($articleId) {

35        // The associated article is retrieved only once

36        $article = $this->articleDAO->find($articleId);

37

38        // art\_id is not selected by the SQL query

39        // The article won't be retrieved during domain objet construction

40        $sql = "select com\_id, com\_content, usr\_id from t\_comment where art\_id=? order by com\_id";

41        $result = $this->getDb()->fetchAll($sql, array($articleId));

42

43        // Convert query result to an array of domain objects

44        $comments = array();

45        foreach ($result as $row) {

46            $comId = $row['com\_id'];

47            $comment = $this->buildDomainObject($row);

48            // The associated article is defined for the constructed comment

49            $comment->setArticle($article);

50            $comments[$comId] = $comment;

51        }

52        return $comments;

53    }

54

55    /\*\*

56     \* Creates an Comment object based on a DB row.

57     \*

58     \* @param array $row The DB row containing Comment data.

59     \* @return \MicroCMS\Domain\Comment

60     \*/

61    protected function buildDomainObject($row) {

62        $comment = new *Comment*();

63        $comment->setId($row['com\_id']);

64        $comment->setContent($row['com\_content']);

65

66        if (array\_key\_exists('art\_id', $row)) {

67            // Find and set the associated article

68            $articleId = $row['art\_id'];

69            $article = $this->articleDAO->find($articleId);

70            $comment->setArticle($article);

71        }

72        if (array\_key\_exists('usr\_id', $row)) {

73            // Find and set the associated author

74            $userId = $row['usr\_id'];

75            $user = $this->userDAO->find($userId);

76            $comment->setAuthor($user);

77        }

78

79        return $comment;

80    }

81}

**Partie Contrôleur**

Le fichier de configuration de l'application Silex app/app.php est modifié pour intégrer les nouveaux services.

1<?php

2

3use Symfony\Component\Debug\ErrorHandler;

4use Symfony\Component\Debug\ExceptionHandler;

5

6// Register global error and exception handlers

7*ErrorHandler*::register();

8*ExceptionHandler*::register();

9

10// Register service providers

11$app->register(new *Silex*\Provider\DoctrineServiceProvider());

12$app->register(new *Silex*\Provider\TwigServiceProvider(), array(

13    'twig.path' => \_\_DIR\_\_.'/../views',

14));

15$app->register(new *Silex*\Provider\SessionServiceProvider());

16$app->register(new *Silex*\Provider\UrlGeneratorServiceProvider());

17$app->register(new *Silex*\Provider\SecurityServiceProvider(), array(

18    'security.firewalls' => array(

19        'secured' => array(

20            'pattern' => '^/',

21            'anonymous' => true,

22            'logout' => true,

23            'form' => array('login\_path' => '/login', 'check\_path' => '/login\_check'),

24            'users' => $app->share(function () use ($app) {

25                return new *MicroCMS*\DAO\UserDAO($app['db']);

26            }),

27        ),

28    ),

29));

30

31// Register services

32$app['dao.article'] = $app->share(function ($app) {

33    return new *MicroCMS*\DAO\ArticleDAO($app['db']);

34});

35$app['dao.user'] = $app->share(function ($app) {

36    return new *MicroCMS*\DAO\UserDAO($app['db']);

37});

38$app['dao.comment'] = $app->share(function ($app) {

39    $commentDAO = new *MicroCMS*\DAO\CommentDAO($app['db']);

40    $commentDAO->setArticleDAO($app['dao.article']);

41    $commentDAO->setUserDAO($app['dao.user']);

42    return $commentDAO;

43});

Il est important de bien comprendre les paramètres utilisés pour la définition du pare-feu *(firewall)* associé au fournisseur de services SecurityServiceProvider :

* pattern définit la partie sécurisée de l'application sous la forme d'une [expression rationnelle](http://fr.wikipedia.org/wiki/Expression_rationnelle). Ici, la valeur ^/ indique que le pare-feu sécurise l'intégralité de l'application ;
* anonymous précise qu'un utilisateur non authentifié peut tout de même accéder à la partie sécurisée. Il est nécessaire pour que les visiteurs anonymes puissent continuer à consulter les articles du CMS ;
* logout indique qu'il est possible pour les utilisateurs authentifiés de se déconnecter de l'application ;
* form permet d'utiliser un formulaire comme méthode d'authentification.
* login\_path définit le chemin vers le formulaire et check\_path le chemin d'authentification ;
* users définit le fournisseur de données utilisateur, autrement dit la source de données qui permet d'accéder aux utilisateurs de l'application. Ici, il s'agit logiquement d'une instance de la classeUserDAO créée précédemment.

L'enregistrement du fournisseur de services SessionServiceProvider démarre automatiquement la gestion des sessions PHP.

Il faut aussi ajouter une route pour afficher le formulaire d'authentification dans le fichierapp/routes.php.

1<?php

2

3use Symfony\Component\HttpFoundation\Request;

4

5// ...

6

7// Login form

8$app->get('/login', function(Request $request) use ($app) {

9    return $app['twig']->render('login.html.twig', array(

10        'error'         => $app['security.last\_error']($request),

11        'last\_username' => $app['session']->get('\_security.last\_username'),

12    ));

13})->bind('login');  // named route so that path('login') works in Twig templates

Le contrôleur associé utilise la classe Symfony Request pour afficher la vue login.html.twig en lui passant en paramètres l'éventuelle dernière erreur de sécurité (par exemple un utilisateur non reconnu) et le dernier nom d'utilisateur utilisé.

Cette nouvelle route est nommée login grâce à l'appel de la méthode bind. Sans cela, l'appel à la fonction path('login') dans un template Twig provoquera une erreur (voir plus loin).

**Partie Vue**

Dans la partie Vue, il faut tout d'abord créer la vue login.html.twig associée à la route d'authentification.

1{% extends 'layout.html.twig' %}

2

3{% block title %}User authentication{% endblock %}

4

5{% block content %}

6<h2 class="text-center">{{ block('title') }}</h2>

7{% if error %}

8<div class="alert alert-danger">

9    <strong>Login failed!</strong> {{ error }}

10</div>

11{% endif %}

12<div class="well">

13    <form class="form-signin form-horizontal" role="form" action="{{ path('login\_check') }}" method="post">

14        <div class="form-group">

15            <div class="col-sm-6 col-sm-offset-3 col-md-4 col-md-offset-4">

16            <input type="text" name="\_username" value="{{ last\_username }}" class="form-control" placeholder="Enter your username" required autofocus>

17            </div>

18        </div>

19        <div class="form-group">

20            <div class="col-sm-6 col-sm-offset-3 col-md-4 col-md-offset-4">

21                <input type="password" name="\_password" class="form-control" placeholder="Enter your password" required>

22            </div>

23        </div>

24        <div class="form-group">

25            <div class="col-sm-6 col-sm-offset-3 col-md-4 col-md-offset-4">

26                <button type="submit" class="btn btn-default btn-primary"><span class="glyphicon glyphicon-log-in"></span> Login</button>

27            </div>

28        </div>

29    </form>

30</div>

31{% endblock %}

Comme toutes nos vues, elle hérite de layout.html.twig afin d'intégrer les éléments d'interface communs (barre de navigation, pied de page, etc.). Elle définit un formulaire (balise <form>) contenant les champs \_username et \_password pour saisir le login et le mot de passe de l'utilisateur. L'action associée à ce formulaire utilise la fonction path fournie par le composant twig-bridge pour récupérer le chemin d'authentification défini lors du paramétrage du pare-feu. Le nom de ce chemin provient de la valeur du paramètre check\_path : les / sont remplacés par des \_ et le / initial est supprimé.

Ensuite, on modifie la vue article.html.twig pour obtenir un affichage adapté à la présence d'un utilisateur connecté.

1{% extends "layout.html.twig" %}

2

3{% block title %}{{ article.title }}{% endblock %}

4

5{% block content %}

6<p>

7    <h2>{{ article.title }}</h2>

8    <p>{{ article.content }}</p>

9

10    <h3>Comments</h3>

11    {% for comment in comments %}

12        <strong>{{ comment.author.username }}</strong> said : {{ comment.content }}<br>

13    {% else %}

14        No comments yet.

15    {% endfor %}

16

17    <h3>Add a comment</h3>

18    {% if is\_granted('IS\_AUTHENTICATED\_FULLY') %}

19        Soon!

20    {% else %}

21        <a href="{{ path('login') }} ">Log in</a> to add comments.

22    {% endif %}

23</p>

24{% endblock %}

Le nom de l'auteur du commentaire est maintenant accessible via la variable Twigcomment.author.username et non plus comment.author. L'appel à la fonction Twigis\_granted('IS\_AUTHENTICATED\_FULLY') permet de vérifier si la vue est affichée pour un utilisateur authentifié par l'application. Si c'est le cas, il faudrait lui offrir la possibilité d'ajouter un commentaire. Ce sera l'objet d'une prochaine itération. Sinon, on précise au visiteur anonyme qu'il doit se connecter pour pouvoir commenter l'article.

Enfin, on modifie la partie commune à toutes les vues (fichier layout.html.twig) afin d'ajouter à la barre de navigation un menu déroulant associé à l'éventuel utilisateur authentifié.

1<!doctype html>

2<html>

3<head>

4    <meta charset="utf-8">

5    <meta http-equiv="X-UA-Compatible" content="IE=edge">

6    <meta name="viewport" content="width=device-width, initial-scale=1">

7    <link href="{{ app.request.basepath }}/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet">

8    <link href="{{ app.request.basepath }}/css/microcms.css" rel="stylesheet">

9    <title>MicroCMS - {% block title %}{% endblock %}</title>

10</head>

11<body>

12    <div class="container">

13        <nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">

14            <div class="container">

15                <div class="navbar-header">

16                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#navbar-collapse-target">

17                        <span class="icon-bar"></span>

18                        <span class="icon-bar"></span>

19                        <span class="icon-bar"></span>

20                    </button>

21                    <a class="navbar-brand" href="/">MicroCMS</a>

22                </div>

23                <div class="collapse navbar-collapse" id="navbar-collapse-target">

24                    <ul class="nav navbar-nav navbar-right">

25                        {% if app.security.token and is\_granted('IS\_AUTHENTICATED\_FULLY') %}

26                            <li class="dropdown">

27                            <a href="#" class="dropdown-toggle" data-toggle="dropdown">

28                                <span class="glyphicon glyphicon-user"></span> Welcome, {{ app.security.token.user.username }} <b class="caret"></b></a>

29                                <ul class="dropdown-menu">

30                                    <li><a href="{{ path('logout') }}">Log out</a></li>

31                                </ul>

32                            </li>

33                        {% else %}

34                            <li class="dropdown">

35                            <a href="#" class="dropdown-toggle" data-toggle="dropdown">

36                                <span class="glyphicon glyphicon-user"></span> Not connected <b class="caret"></b></a>

37                                <ul class="dropdown-menu">

38                                    <li><a href="{{ path('login') }}">Log in</a></li>

39                                </ul>

40                            </li>

41                        {% endif %}

42                    </ul>

43                </div>

44            </div><!-- /.container -->

45        </nav>

46        <div id="content">{% block content %}{% endblock %}</div>

47        <footer class="footer">

48            <a href="https://github.com/bpesquet/OC-MicroCMS">MicroCMS</a> is a minimalistic CMS built as a showcase for modern PHP development.

49        </footer>

50    </div>

51

52    <!-- jQuery -->

53    <script src="{{ app.request.basepath }}/lib/jquery/jquery-1.11.1.min.js"></script>

54    <!-- JavaScript Boostrap plugin -->

55    <script src="{{ app.request.basepath }}/lib/bootstrap/js/bootstrap.min.js"></script>

56</body>

57</html>

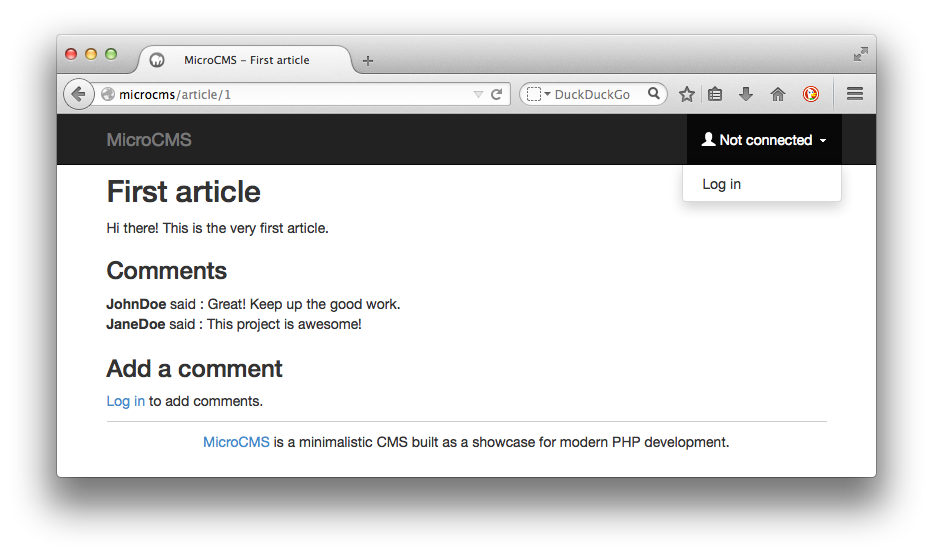
Si l'utilisateur est authentifié, le menu affiche son nom (accessible via la variable Twigapp.security.token.user.username) et lui permet de se déconnecter (lien vers path('logout')). Sinon, il affiche un message et un lien vers le formulaire d'authentification (lien vers path('login')). On ajoute également des liens vers jQuery et le plugin JavaScript de Bootstrap afin de faire fonctionner le menu déroulant.

L'ajout des liens JavaScript en fin de fichier plutôt qu'au début est une bonne pratique qui permet d'optimiser le temps de chargement des pages.

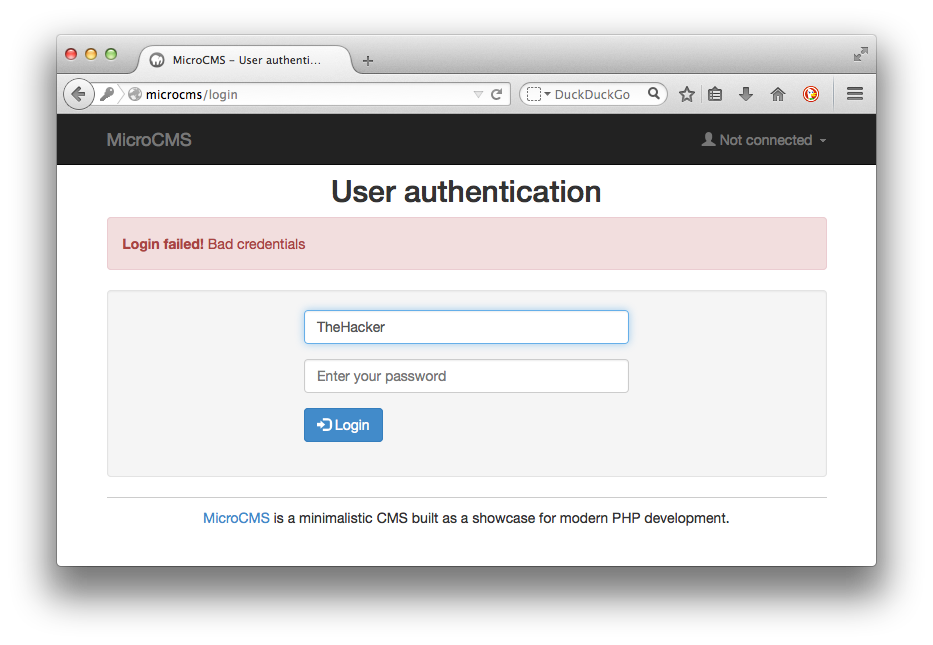
Le lien vers jQuery est à adapter à la version que vous avez téléchargée pendant l'itération 6.

**Résultat obtenu**

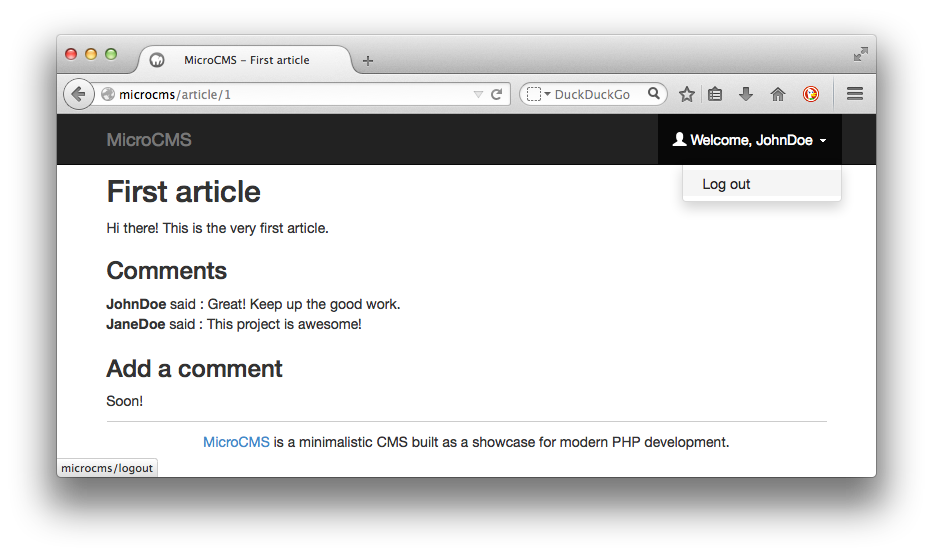
La sécurisation de notre application est terminée. Ouvrez l'URL [http://microcms](http://microcms/) pour afficher la page d'accueil de l'application. Elle doit maintenant disposer d'un menu déroulant en haut à droite. Lorsqu'aucun utilisateur ne s'est authentifié, ce menu comporte une entrée invitant l'utilisateur à le faire.



Le formulaire d'authentification permet à l'utilisateur de saisir son nom et son mot de passe. Les éventuelles erreurs d'authentification sont affichées.



Lorsque l'authentification réussit, le menu déroulant de la barre de navigation est mis à jour, ainsi que l'affichage d'un article.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-08).

Bilan

Cette itération nous a permis de sécuriser notre application web en exploitant les possibilités offertes par Symfony (et donc Silex). On constate combien l'intégration d'un framework, même si elle nécessite un travail initial d'adaptation, facilite l'ajout de fonctionnalités complexes à une application web.

La prochaine itération permettra aux utilisateurs authentifiés d'interagir avec l'application.

ITÉRATION 9 : AJOUT DE COMMENTAIRES À UN ARTICLE

Le but de cette itération est de permettre aux utilisateurs authentifiés d'ajouter des commentaires à un article.

Symfony et les formulaires

La saisie de commentaires par les utilisateurs de l'application implique l'ajout d'un formulaire dans la vue qui affiche les détails sur un article. Il serait possible de gérer manuellement la définition de ce formulaire et la récupération des données saisies. Nous allons plutôt en profiter pour découvrir une autre des fonctionnalités offertes par le framework Symfony : la gestion des formulaires.

Grâce au composant [Form](http://symfony.com/doc/current/book/forms.html), Symfony permet de créer un formulaire web à partir d'un objet. Les champs du formulaire sont associés aux propriétés de cet objet. Cette association est bidirectionnelle : le formulaire affiche les valeurs des propriétés de l'objet, et sa soumission *(submit)* met à jour les propriétés de l'objet. Le composant Form permet encore bien d'autres choses, comme nous allons le voir à présent.

Mise à jour de l'application

**Composants Symfony**

Commençons par récupérer les composants Symfony nécessaires en les ajoutant dans le fichier de dépendances composer.json.

1"require": {

2    ...

3    "symfony/form": "~2.4",

4    "symfony/translation": "~2.4"

5}

6...

Le composant form regroupe les services de gestion des formulaires. Le composant translation offre des services de traduction nécessaires pour utiliser le composant form.

Une fois ce fichier modifié, on utilise Composer pour télécharger ces composants et leurs éventuelles dépendances.

composer update

**Partie Modèle**

Le formulaire que nous devons créer permettra à un utilisateur connecté de saisir un nouveau commentaire. Ce commentaire sera associé à un article et à un utilisateur existants. Son identifiant sera défini au moment de son insertion dans la base de données. Le seul champ que notre formulaire pemettra de saisir sera donc le contenu du commentaire.

Symfony offre plusieurs manières de créer un formulaire. Nous allons employer la technique recommandée par les [bonnes pratiques](http://symfony.com/doc/current/best_practices/forms.html), qui consiste à le définir dans sa propre classe, afin de le rendre réutilisable dans toute l'application. Dans le répertoire src/Form/Type (à créer lui aussi), ajoutez un nouveau fichier source nommé CommentType.php. Placez-y le code source suivant.

1<?php

2

3namespace MicroCMS\Form\Type;

4

5use Symfony\Component\Form\AbstractType;

6use Symfony\Component\Form\FormBuilderInterface;

7

8class CommentType extends AbstractType

9{

10    public function buildForm(FormBuilderInterface $builder, array $options)

11    {

12        $builder->add('content', 'textarea');

13    }

14

15    public function getName()

16    {

17        return 'comment';

18    }

19}

Notre classe hérite de la classe Symfony [AbstractType](http://api.symfony.com/2.6/Symfony/Component/Form/AbstractType.html) et redéfinit sa méthode buildForm qui, comme son nom l'indique, permet de construire un formulaire. Ici, le formulaire aura une zone de texte associée au contenu du commentaire (champ content de type textarea).

Le nom de la zone de texte ("content") n'est pas choisi au hasard : il correspond exactement à la propriété content de la classe métier Comment. C'est indispensable pour que Symfony puisse associer notre formulaire à une instance de Comment.

Nous devons modifier la classe CommentDAO pour la rendre capable d'insérer des commentaires dans la base de données. Pour cela, une méthode save  est ajoutée à cette classe.

1<?php

2

3namespace MicroCMS\DAO;

4

5use MicroCMS\Domain\Comment;

6

7class CommentDAO extends DAO

8{

9    // ...

10

11    /\*\*

12     \* Saves a comment into the database.

13     \*

14     \* @param \MicroCMS\Domain\Comment $comment The comment to save

15     \*/

16    public function save(Comment $comment) {

17        $commentData = array(

18            'art\_id' => $comment->getArticle()->getId(),

19            'usr\_id' => $comment->getAuthor()->getId(),

20            'com\_content' => $comment->getContent()

21            );

22

23        if ($comment->getId()) {

24            // The comment has already been saved : update it

25            $this->getDb()->update('t\_comment', $commentData, array('com\_id' => $comment->getId()));

26        } else {

27            // The comment has never been saved : insert it

28            $this->getDb()->insert('t\_comment', $commentData);

29            // Get the id of the newly created comment and set it on the entity.

30            $id = $this->getDb()->lastInsertId();

31            $comment->setId($id);

32        }

33    }

34

35    // ...

Cette méthode rassemble les valeurs BD dans le tableau $commentData, puis vérifie s'il faut insérer ou mettre à jour le commentaire en se basant sur l'existence d'une valeur pour l'identifiant du commentaire. Ensuite, elle utilise la méthode DBAL appropriée pour effectuer l'opération dans la table t\_comment.

**Partie Contrôleur**

Nous devons tout d'abord enregistrer les nouveaux fournisseurs de services utilisés dans le fichierapp/app.php.

1<?php

2

3// ...

4

5// Register service providers

6// ...

7$app->register(new *Silex*\Provider\FormServiceProvider());

8$app->register(new *Silex*\Provider\TranslationServiceProvider());

9

10// ...

Ensuite, il faut mettre à jour le fichier app/routes.php  pour créer le formulaire d'ajout d'un commentaire avant de générer la vue qui affiche les détails sur un article. Voici le nouveau contenu de ce fichier.

1<?php

2

3use Symfony\Component\HttpFoundation\Request;

4use MicroCMS\Domain\Comment;

5use MicroCMS\Form\Type\CommentType;

6

7// Home page

8$app->get('/', function () use ($app) {

9    $articles = $app['dao.article']->findAll();

10    return $app['twig']->render('index.html.twig', array('articles' => $articles));

11});

12

13// Article details with comments

14$app->match('/article/{id}', function ($id, Request $request) use ($app) {

15    $article = $app['dao.article']->find($id);

16    $user = $app['security']->getToken()->getUser();

17    $commentFormView = null;

18    if ($app['security']->isGranted('IS\_AUTHENTICATED\_FULLY')) {

19        // A user is fully authenticated : he can add comments

20        $comment = new *Comment*();

21        $comment->setArticle($article);

22        $comment->setAuthor($user);

23        $commentForm = $app['form.factory']->create(new *CommentType*(), $comment);

24        $commentForm->handleRequest($request);

25        if ($commentForm->isSubmitted() && $commentForm->isValid()) {

26            $app['dao.comment']->save($comment);

27            $app['session']->getFlashBag()->add('success', 'Your comment was succesfully added.');

28        }

29        $commentFormView = $commentForm->createView();

30    }

31    $comments = $app['dao.comment']->findAllByArticle($id);

32    return $app['twig']->render('article.html.twig', array(

33        'article' => $article,

34        'comments' => $comments,

35        'commentForm' => $commentFormView));

36});

37

38// Login form

39$app->get('/login', function(Request $request) use ($app) {

40    return $app['twig']->render('login.html.twig', array(

41        'error'         => $app['security.last\_error']($request),

42        'last\_username' => $app['session']->get('\_security.last\_username'),

43    ));

44})->bind('login');  // named route so that path('login') works in Twig templates

Le contrôleur de la route /article/{id} est modifié : nous avons remplacé $app->get par $app->match afin de gérer à la fois l'accès à cette route via les commandes HTTP GET et POST ($app->get ne gère que la commande GET).

À l'intérieur de ce contrôleur, on récupère l'article (via son identifiant passé dans l'URL) puis l'utilisateur connecté (en utilisant le service security de Symfony). S'il y a bien un utilisateur connecté, on crée un nouveau commentaire puis le formulaire associé à ce commentaire (appel à $app['form.factory']->create). Ensuite, la méthode handleRequest gère la soumission du formulaire ([plus de détails](http://symfony.com/doc/current/book/forms.html#handling-form-submissions)). Si le formulaire a été soumis et que les données reçues sont valides, on fait appel au DAO pour sauvegarder le nouveau commentaire et on crée un message de succès.

Dans tous les cas, on ajoute aux données dynamiques envoyées à la vue le formulaire d'ajout d'un commentaire (variable $commentFormView). Si aucun utilisateur n'est connecté, cette variable vautnull.

**Partie Vue**

Il nous reste à afficher le formulaire créé dans la partie Contrôleur. Pour cela, on modifie le templatearticle.html.twig de la manière suivante.

1{% extends "layout.html.twig" %}

2

3{% block title %}{{ article.title }}{% endblock %}

4

5{% block content %}

6<p>

7    <h2>{{ article.title }}</h2>

8    <p>{{ article.content }}</p>

9

10    <h3>Comments</h3>

11    {% for comment in comments %}

12        <strong>{{ comment.author.username }}</strong> said : {{ comment.content }}<br>

13    {% else %}

14        No comments yet.

15    {% endfor %}

16

17    <h3>Add a comment</h3>

18    {% if commentForm %}

19        {{ form\_start(commentForm) }}

20            <div class="form-group">

21                {{ form\_errors(commentForm.content) }}

22                {{ form\_widget(commentForm.content, { 'attr':  {

23                    'rows': '4',

24                    'class': 'form-control',

25                    'placeholder': 'Enter your comment'

26                }}) }}

27            </div>

28            <div class="form-group">

29                <input type="submit" class="btn btn-primary" value="Publish comment" />

30            </div>

31        {{ form\_end(commentForm) }}

32        {% for flashMessage in app.session.flashbag.get('success') %}

33            <div class="alert alert-success">

34                {{ flashMessage }}

35            </div>

36        {% endfor %}

37    {% else %}

38        <a href="{{ path('login') }} ">Log in</a> to add comments.

39    {% endif %}

40</p>

41{% endblock %}

Dans ce template, on vérifie si le formulaire commentForm existe puis (si c'est le cas) on utilise des fonctions Twig pour générer le code HTML associé à chaque partie du formulaire.

* form\_start génère le début du formulaire (balise HTML <form>).
* form\_widget génère un champ de formulaire.

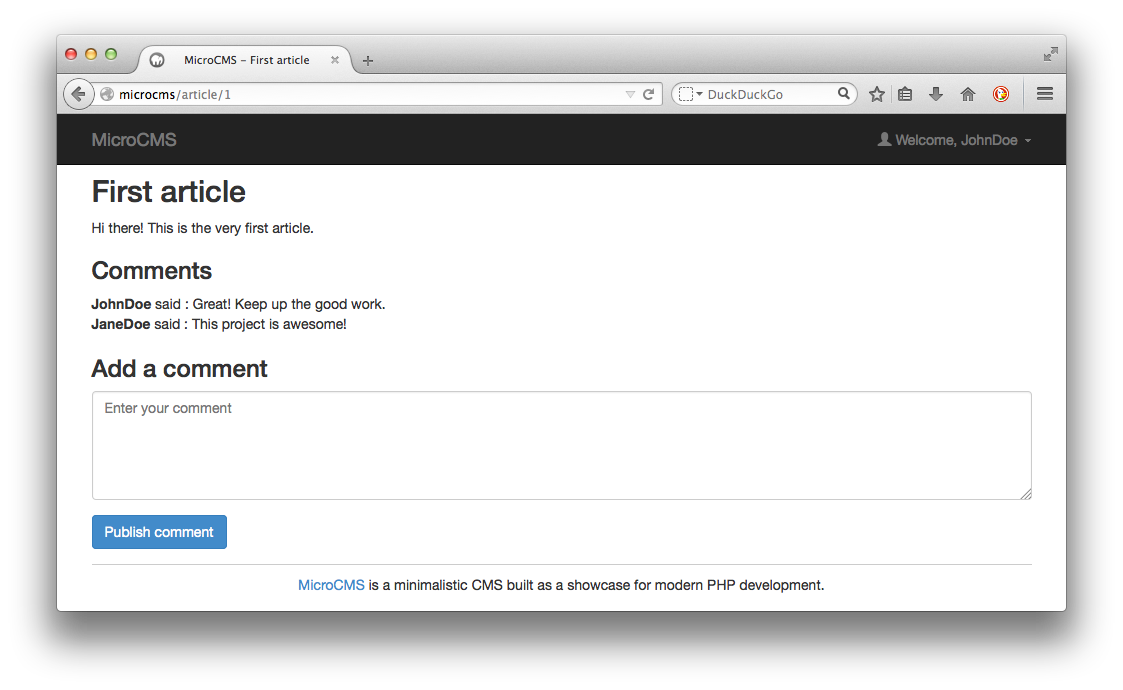
Pour plus de précisions, consultez la [documentation détaillée](http://symfony.com/doc/current/reference/forms/twig_reference.html) de ces fonctions.

Les champs générés par un formulaire Symfony sont par défaut obligatoires (attribut HTML5required), ce qui pourra déclencher une validation côté navigateur.

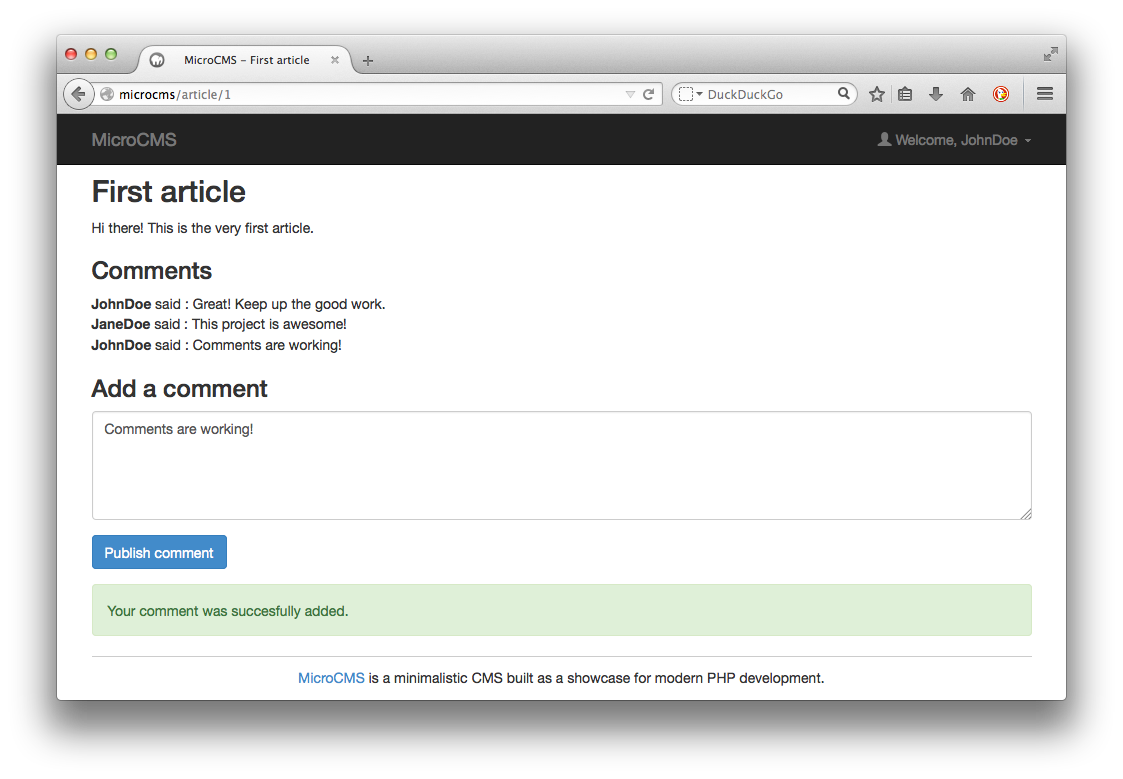
On associe à chaque champ généré des classes Bootstrap (form-group  et form-control) afin d'améliorer la présentation. Les éventuels messages de succès sont également affichés via Bootstrap.

**Résultat obtenu**

Lorsqu'un utilisateur connecté clique sur le titre d'un article, la vue qui affiche son détail lui permet à présent d'ajouter un commentaire.



Une fois le commentaire saisi et publié, l'affichage de l'article intègre le nouveau commentaire ainsi qu'un message de succès.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-09).

Bilan

Cette itération a fourni l'occasion d'intégrer à notre application la gestion des formulaires. Le composant Symfony associé permet de simplifier grandement ce processus. Ce composant est très puissant et dispose de nombreuses autres fonctionnalités que celles présentées ici.

ITÉRATION 10 : BACK-OFFICE D'ADMINISTRATION

Le but de cette itération est de pouvoir administrer l'application via un **back-office** dédié.

Cette itération est plus longue et un peu plus complexe que les précédentes. Concentrez-vous bien !

Objectifs du back-office

Le **back-office** d'une application Web désigne "la partie du site internet qui n'est visible que par l'administrateur du site et qui permet de gérer le contenu, les fonctionnalités..." ([source](http://aide.meabilis.fr/glossaire/b/definition-back-office.html)).

L'accès au back-office sera réservé aux administrateurs de l'application. Il leur offrira les fonctionnalités suivantes :

* Affichage des articles, commentaires et utilisateurs de l'application.
* Modification et suppression d'un article, d'un commentaire ou d'un utilisateur.
* Ajout d'un nouvel article ou d'un nouvel utilisateur (l'ajout de commentaires étant déjà permis par l'application).

Gestion des rôles par Symfony

Pour pouvoir réserver l'accès au back-office aux administrateurs, il faut pouvoir définir si un utilisateur connecté possède ou non le droit d'administration. Pour cela, nous allons utiliser la notion de **rôle** mise en œuvre dans Symfony. Ce framework permet d'associer aux utilisateurs un ou plusieurs rôle(s). L'accès aux ressources de l'application est ensuite conditionné à la possession de rôles particuliers. On peut relier les rôles par une hiérarchie, la possesion d'un rôle donnant automatiquement les droits associés à un autre rôle. Pour plus de détails sur les rôles, consultez la [documentation Symfony.](http://symfony.com/fr/doc/current/book/security.html#les-roles)

Notre application distingue deux types d'utilisateurs :

* les utilisateurs simples, qui ne peuvent qu'ajouter des commentaires aux articles ;
* les administrateurs, qui ont en plus l'accès complet au back-office.

Dans l'itération 8, nous avions donné à nos utilisateurs le rôle ROLE\_USER : c'est le rôle par défaut pour Symfony. Nous définissons donc un second rôle ROLE\_ADMIN associé aux administrateurs.

Commençons par ajouter un administrateur à l'application en modifiant le fichier db/content.sql.

1/\* ... \*/

2

3/\* raw password is '@dm1n' \*/

4insert into t\_user values

5(3, 'admin', 'gqeuP4YJ8hU3ZqGwGikB6+rcZBqefVy+7hTLQkOD+jwVkp4fkS7/gr1rAQfn9VUKWc7bvOD7OsXrQQN5KGHbfg==', 'EDDsl&fBCJB|a5XUtAlnQN8', 'ROLE\_ADMIN');

6

7/\* ... \*/

Vous pouvez ajouter temporairement un contrôleur similaire à [celui-ci](https://gist.github.com/bpesquet/501c789f01e5bdeda90d) pour obtenir le hachage d'un mot de passe.

Modifiez le fichier db/content.sql comme indiqué ci-dessus puis exécutez successivementdb/structure.sql  et db/content.sql pour mettre à jour votre base de données.

Il faut ensuite mettre à jour la configuration de la sécurité pour soumettre l'accès au back-office (zone /admin) à la possession du rôle ROLE\_ADMIN. Voici les modifications à intégrer au fichier app/app.php.

1<?php

2

3// ...

4

5$app->register(new *Silex*\Provider\SecurityServiceProvider(), array(

6    'security.firewalls' => array(

7        // ...

8    ),

9    'security.role\_hierarchy' => array(

10        'ROLE\_ADMIN' => array('ROLE\_USER'),

11    ),

12    'security.access\_rules' => array(

13        array('^/admin', 'ROLE\_ADMIN'),

14    ),

15));

16

17// ...

Comme indiqué plus haut, nous modifions la configuration du pare-feu pour définir une hiérarchie entreROLE\_ADMIN et ROLE\_USER, puis pour protéger spécifiquement la zone /admin.

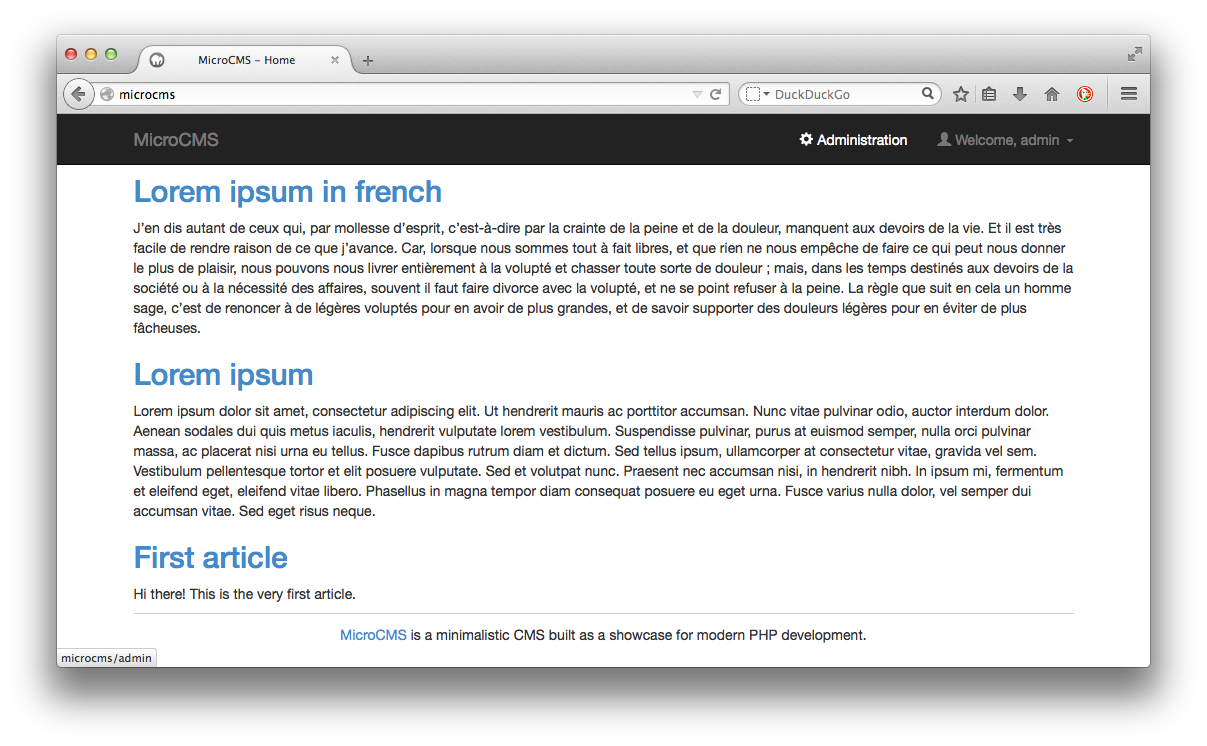
Enfin, il faut signaler visuellement à l'administrateur connecté qu'il a accès au back-office d'administration. Pour cela, ajoutez le code ci-dessous à la ligne 25 du fichierviews/layout.html.twig.

1{% if app.security.token and is\_granted('ROLE\_ADMIN') %}

2    <li class="{% if adminMenu is defined %}active{% endif %}"><a href="/admin"><span class="glyphicon glyphicon-cog"></span> Administration</a></li>

3{% endif %}

À présent, ouvrez l'URL http://microcms et tentez de vous connecter en tant qu'administrateur (nom d'utilisateur : "admin", mot de passe : "@dm1n"). Vous devriez obtenir l'affichage d'une nouvelle entrée "Administration" dans le barre de navigation de l'application.



Bien entendu, le clic sur ce menu provoquer une erreur : il nous reste à écrire tout le code d'administration.

Page d'accueil du back-office

L'accueil du back-office (route /admin) doit afficher à l'administrateur l'ensemble des données de l'application. On ajoute pour cela une nouvelle route à la fin du fichier app/routes.php.

1<?php

2

3// ...

4

5// Admin home page

6$app->get('/admin', function() use ($app) {

7    $articles = $app['dao.article']->findAll();

8    $comments = $app['dao.comment']->findAll();

9    $users = $app['dao.user']->findAll();

10    return $app['twig']->render('admin.html.twig', array(

11        'articles' => $articles,

12        'comments' => $comments,

13        'users' => $users));

14});

Le contrôleur associé génère la vue admin.html.twig en lui fournissant les listes des articles, des commentaires et des utilisateurs. La méthode findAll existe déjà dans la classe ArticleDAO. Il faut créer les deux autres.

Modifiez le fichier src/DAO/CommentDAO.php pour ajouter la méthode findAll comme indiqué ci-dessous.

1<?php

2

3// ...

4

5class CommentDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Returns a list of all comments, sorted by date (most recent first).

11     \*

12     \* @return array A list of all comments.

13     \*/

14    public function findAll() {

15        $sql = "select \* from t\_comment order by com\_id desc";

16        $result = $this->getDb()->fetchAll($sql);

17

18        // Convert query result to an array of domain objects

19        $entities = array();

20        foreach ($result as $row) {

21            $id = $row['com\_id'];

22            $entities[$id] = $this->buildDomainObject($row);

23        }

24        return $entities;

25    }

26

27    // ...

Modifiez le fichier src/DAO/UserDAO.php pour ajouter la méthode findAll comme indiqué ci-dessous.

1<?php

2

3// ...

4

5class UserDAO extends DAO implements UserProviderInterface

6{

7    /\*\*

8     \* Returns a list of all users, sorted by role and name.

9     \*

10     \* @return array A list of all users.

11     \*/

12    public function findAll() {

13        $sql = "select \* from t\_user order by usr\_role, usr\_name";

14        $result = $this->getDb()->fetchAll($sql);

15

16        // Convert query result to an array of domain objects

17        $entities = array();

18        foreach ($result as $row) {

19            $id = $row['usr\_id'];

20            $entities[$id] = $this->buildDomainObject($row);

21        }

22        return $entities;

23    }

24

25    // ...

Modifiez la feuille de style web/css/microcms.css pour y ajouter le contenu ci-dessous, destiné à améliorer la présentation de la page d'accueil.

1.adminTable {

2    *margin-top*: 20px;

3    *margin-bottom*: 20px;

4}

Afin de limiter la taille du contenu des articles dans la page d'accueil du back-office, nous allons utiliser la fonction truncate fournie par l'extension Text de Twig. Au passage, nous allons intégrer également les composants Symfony nécessaires pour valider des formulaires.

Pour cela, modifiez votre fichier composer.json comme indiqué ci-dessous.

1"require": {

2    ...

3    "twig/extensions": "~1.2",

4    "symfony/validator": "~2.4",

5    "symfony/config": "~2.4"

6},

7...

Récupérez ces nouveaux composants grâce à la commande habituelle :

composer update

Il faut ensuite modifier le fichier de configuration de l'application app/app.php  pour intégrer la nouvelle extension et les nouveaux composants. Ajoutez les lignes suivantes juste après avoir enregistréTwigServiceProvider (ligne 15).

1$app['twig'] = $app->share($app->extend('twig', function(Twig\_Environment $twig, $app) {

2    $twig->addExtension(new Twig\_Extensions\_Extension\_Text());

3    return $twig;

4}));

5$app->register(new Silex\Provider\ValidatorServiceProvider());

Enfin, créez le fichier views/admin.html.twig en lui donnant le contenu ci-dessous.

1{% extends "layout.html.twig" %}

2{% set adminMenu = true %}

3

4{% block title %}Administration{% endblock %}

5

6{% block content %}

7<h2 class="text-center">{{ block('title') }}</h2>

8{% for flashMessage in app.session.flashbag.get('success') %}

9<div class="alert alert-success">

10    {{ flashMessage }}

11</div>

12{% endfor %}

13<div class="row">

14    <div class="col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3">

15        <ul class="nav nav-tabs nav-justified">

16            <li class="active"><a href="#articles" data-toggle="tab">Articles</a></li>

17            <li><a href="#comments" data-toggle="tab">Comments</a></li>

18            <li><a href="#users" data-toggle="tab">Users</a></li>

19        </ul>

20    </div>

21</div>

22<div class="tab-content">

23    <div class="tab-pane fade in active adminTable" id="articles">

24        {% if articles %}

25        <div class="table-responsive">

26            <table class="table table-hover table-condensed">

27                <thead>

28                    <tr>

29                        <th>Title</th>

30                        <th>Content</th>

31                        <th></th>  <!-- Actions column -->

32                    </tr>

33                </thead>

34                {% for article in articles %}

35                <tr>

36                    <td><a class="articleTitle" href="/article/{{ article.id }}">{{ article.title }}</a></td>

37                    <td>{{ article.content | truncate(60) }}</td>

38                    <td>

39                        <a href="/admin/article/{{ article.id }}/edit" class="btn btn-info btn-xs" title="Edit"><span class="glyphicon glyphicon-edit"></span></a>

40                        <button type="button" class="btn btn-danger btn-xs" title="Delete" data-toggle="modal" data-target="#articleDialog{{ article.id }}"><span class="glyphicon glyphicon-remove"></span>

41                        </button>

42                        <div class="modal fade" id="articleDialog{{ article.id }}" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">

43                            <div class="modal-dialog">

44                                <div class="modal-content">

45                                    <div class="modal-header">

46                                        <button type="button" class="close" data-dismiss="modal" aria-hidden="true">&times;</button>

47                                        <h4 class="modal-title" id="myModalLabel">Confirmation needed</h4>

48                                    </div>

49                                    <div class="modal-body">

50                                        Do you really want to delete this article ?

51                                    </div>

52                                    <div class="modal-footer">

53                                        <button type="button" class="btn btn-default" data-dismiss="modal">Cancel</button>

54                                        <a href="/admin/article/{{ article.id }}/delete" class="btn btn-danger">Confirm</a>

55                                    </div>

56                                </div><!-- /.modal-content -->

57                            </div><!-- /.modal-dialog -->

58                        </div><!-- /.modal -->

59                    </td>

60                </tr>

61                {% endfor %}

62            </table>

63        </div>

64        {% else %}

65        <div class="alert alert-warning">No articles found.</div>

66        {% endif %}

67        <a href="/admin/article/add"><button type="button" class="btn btn-primary"><span class="glyphicon glyphicon-plus"></span> Add article</button></a>

68    </div>

69    <div class="tab-pane fade adminTable" id="comments">

70        {% if comments %}

71        <div class="table-responsive">

72            <table class="table table-hover table-condensed">

73                <thead>

74                    <tr>

75                        <th>Article</th>

76                        <th>Author</th>

77                        <th>Content</th>

78                        <th></th>  <!-- Actions column -->

79                    </tr>

80                </thead>

81                {% for comment in comments %}

82                <tr>

83                    <td><a class="articleTitle" href="/article/{{ comment.article.id }}">{{ comment.article.title }}</a></td>

84                    <td>{{ comment.author.username }}</td>

85                    <td>{{ comment.content | truncate(60) }}</td>

86                    <td>

87                        <a href="/admin/comment/{{ comment.id }}/edit" class="btn btn-info btn-xs" title="Edit"><span class="glyphicon glyphicon-edit"></span></a>

88                        <button type="button" class="btn btn-danger btn-xs" title="Delete" data-toggle="modal" data-target="#commentDialog{{ comment.id }}"><span class="glyphicon glyphicon-remove"></span>

89                        </button>

90                        <div class="modal fade" id="commentDialog{{ comment.id }}" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">

91                            <div class="modal-dialog">

92                                <div class="modal-content">

93                                    <div class="modal-header">

94                                        <button type="button" class="close" data-dismiss="modal" aria-hidden="true">&times;</button>

95                                        <h4 class="modal-title" id="myModalLabel">Confirmation needed</h4>

96                                    </div>

97                                    <div class="modal-body">

98                                        Do you really want to delete this comment ?

99                                    </div>

100                                    <div class="modal-footer">

101                                        <button type="button" class="btn btn-default" data-dismiss="modal">Cancel</button>

102                                        <a href="/admin/comment/{{ comment.id }}/delete" class="btn btn-danger">Confirm</a>

103                                    </div>

104                                </div><!-- /.modal-content -->

105                            </div><!-- /.modal-dialog -->

106                        </div><!-- /.modal -->

107                    </td>

108                </tr>

109                {% endfor %}

110            </table>

111        </div>

112        {% else %}

113        <div class="alert alert-warning">No comments found.</div>

114        {% endif %}

115    </div>

116    <div class="tab-pane fade adminTable" id="users">

117        {% if users %}

118        <div class="table-responsive">

119            <table class="table table-hover table-condensed">

120                <thead>

121                    <tr>

122                        <th>Name</th>

123                        <th>Role</th>

124                        <th></th>  <!-- Actions column -->

125                    </tr>

126                </thead>

127                {% for user in users %}

128                <tr>

129                    <td>{{ user.username }}</a></td>

130                    <td>

131                        {% if user.role == 'ROLE\_ADMIN' %}

132                            Admin

133                        {% else %}

134                            User

135                        {% endif %}

136                    </td>

137                    <td>

138                        <a href="/admin/user/{{ user.id }}/edit" class="btn btn-info btn-xs" title="Edit"><span class="glyphicon glyphicon-edit"></span></a>

139                        <button type="button" class="btn btn-danger btn-xs" title="Delete" data-toggle="modal" data-target="#userDialog{{ user.id }}"><span class="glyphicon glyphicon-remove"></span>

140                        </button>

141                        <div class="modal fade" id="userDialog{{ user.id }}" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">

142                            <div class="modal-dialog">

143                                <div class="modal-content">

144                                    <div class="modal-header">

145                                        <button type="button" class="close" data-dismiss="modal" aria-hidden="true">&times;</button>

146                                        <h4 class="modal-title" id="myModalLabel">Confirmation needed</h4>

147                                    </div>

148                                    <div class="modal-body">

149                                        Do you really want to delete this user ?

150                                    </div>

151                                    <div class="modal-footer">

152                                        <button type="button" class="btn btn-default" data-dismiss="modal">Cancel</button>

153                                        <a href="/admin/user/{{ user.id }}/delete" class="btn btn-danger">Confirm</a>

154                                    </div>

155                                </div><!-- /.modal-content -->

156                            </div><!-- /.modal-dialog -->

157                        </div><!-- /.modal -->

158                    </td>

159                </tr>

160                {% endfor %}

161            </table>

162        </div>

163        {% else %}

164        <div class="alert alert-warning">No users found.</div>

165        {% endif %}

166        <a href="/admin/user/add"><button type="button" class="btn btn-primary"><span class="glyphicon glyphicon-plus"></span> Add user</button></a>

167    </div>

168</div>

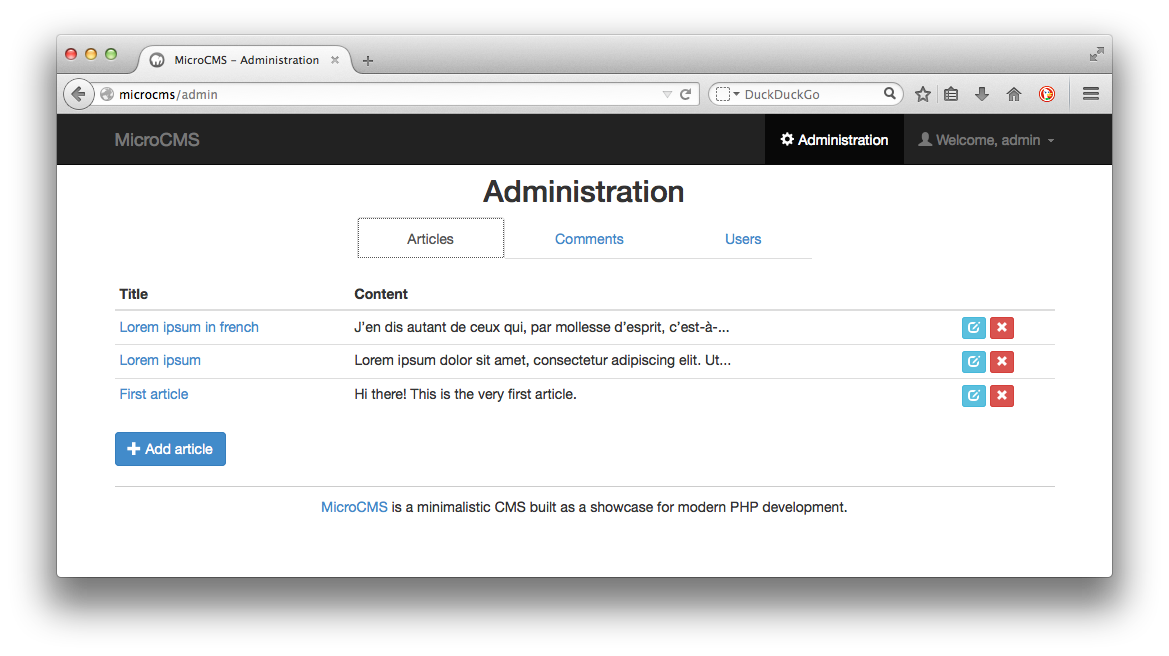
169{% endblock %}

Cette vue comporte beaucoup de code Bootstrap et peut vous paraître complexe si vous connaissez peu ce framework. Elle affiche les données de l'application dans trois onglets **Articles**, **Comments** et **Users**(classe Bootstrap tab-pane). Grâce au code JavaScript inclus dans Bootstrap, le clic sur un onglet déclenche automatiquement l'affichage du contenu de celui-ci.

À chaque ligne de donnée sont associés deux actions matérialisés par des boutons :

* la modification ("Edit") ;
* la suppression ("Delete"). Lors du clic sur ce bouton, une [fenêtre modale](http://fr.wikipedia.org/wiki/Fen%C3%AAtre_modale) permet de demander confirmation à l'utilisateur avant de lancer l'opération de suppression.

Pour mieux comprendre le fonctionnement de cette vue, connectez-vous à l'application en tant qu'administrateur puis accédez à la zone d'administration. Vous obtenez l'affichage suivant.



Vous pouvez cliquez sur les onglets pour afficher les données associées. Vous pouvez également cliquer sur les boutons rouge "Delete" pour voir apparaître les fenêtres modales de confirmation. Annulez ensuite l'opération : il nous reste à écrire son code.

Gestion des articles

Ajouter un nouvel article nécessite de saisir ses caractéristiques dans un formulaire. En suivant la méthode préconisée par Symfony, on définit ce formulaire dans une classe ArticleType créée dans le fichiersrc/Form/Type/ArticleType.php.

1<?php

2

3namespace MicroCMS\Form\Type;

4

5use Symfony\Component\Form\AbstractType;

6use Symfony\Component\Form\FormBuilderInterface;

7

8class ArticleType extends AbstractType

9{

10    public function buildForm(FormBuilderInterface $builder, array $options)

11    {

12        $builder

13            ->add('title', 'text')

14            ->add('content', 'textarea');

15    }

16

17    public function getName()

18    {

19        return 'article';

20    }

21}

Les deux champs du formulaire correspondent aux propriétés d'un article. Ce formulaire est affiché dans la vue views/article\_form.html.twig. Créez ce fichier avec le contenu suivant.

1{% extends 'layout.html.twig' %}

2{% set adminMenu = true %}

3

4{% block title %}{{ title }}{% endblock %}

5

6{% block content %}

7<h2 class="text-center">{{ block('title') }}</h2>

8{% for flashMessage in app.session.flashbag.get('success') %}

9<div class="alert alert-success">

10    {{ flashMessage }}

11</div>

12{% endfor %}

13

14<div class="well">

15{{ form\_start(articleForm, { 'attr': {'class': 'form-horizontal'} }) }}

16    <div class="form-group">

17        {{ form\_label(articleForm.title, null, { 'label\_attr':  {

18            'class': 'col-sm-4 control-label'

19        }}) }}

20        <div class="col-sm-6">

21            {{ form\_errors(articleForm.title) }}

22            {{ form\_widget(articleForm.title, { 'attr':  {

23                'class': 'form-control'

24            }}) }}

25        </div>

26    </div>

27    <div class="form-group">

28        {{ form\_label(articleForm.content, null, { 'label\_attr':  {

29            'class': 'col-sm-4 control-label'

30        }}) }}

31        <div class="col-sm-6">

32            {{ form\_errors(articleForm.content) }}

33            {{ form\_widget(articleForm.content, { 'attr':  {

34                'class': 'form-control',

35                'rows': '8'

36            }}) }}

37        </div>

38    </div>

39    <div class="form-group">

40        <div class="col-sm-offset-4 col-sm-3">

41            <input type="submit" class="btn btn-primary" value="Save" />

42        </div>

43    </div>

44{{ form\_end(articleForm) }}

45</div>

46{% endblock %}

Cette vue affiche les propriétés d'un article, ainsi qu'un bouton de validation.

À présent, modifiez le fichier src/DAO/ArticleDAO.php pour y ajouter les méthodes de sauvegarde et de suppression d'un article, comme indiqué ci-dessous.

1<?php

2

3// ...

4

5class ArticleDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Saves an article into the database.

11     \*

12     \* @param \MicroCMS\Domain\Article $article The article to save

13     \*/

14    public function save(Article $article) {

15        $articleData = array(

16            'art\_title' => $article->getTitle(),

17            'art\_content' => $article->getContent(),

18            );

19

20        if ($article->getId()) {

21            // The article has already been saved : update it

22            $this->getDb()->update('t\_article', $articleData, array('art\_id' => $article->getId()));

23        } else {

24            // The article has never been saved : insert it

25            $this->getDb()->insert('t\_article', $articleData);

26            // Get the id of the newly created article and set it on the entity.

27            $id = $this->getDb()->lastInsertId();

28            $article->setId($id);

29        }

30    }

31

32    /\*\*

33     \* Removes an article from the database.

34     \*

35     \* @param integer $id The article id.

36     \*/

37    public function delete($id) {

38        // Delete the article

39        $this->getDb()->delete('t\_article', array('art\_id' => $id));

40    }

41

42    // ...

43}

La suppression d’un article entraînant celle de tous ses commentaires, modifiez également le fichiersrc/DAO/CommentDAO.php pour ajouter la méthode suivante.

1<?php

2

3// ...

4

5class CommentDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Removes all comments for an article

11     \*

12     \* @param $articleId The id of the article

13     \*/

14    public function deleteAllByArticle($articleId) {

15        $this->getDb()->delete('t\_comment', array('art\_id' => $articleId));

16    }

17

18    // ...

19}

On aurait pu obtenir la suppression automatique des commentaires d'un article en utilisant des contraintes SQL ON DELETE CASCADE.

Il ne reste plus qu'à ajouter dans le fichier app/routes.php les routes permettant la création, la modification et la suppresion d'un article. Ajoutez les directives use et les contrôleurs ci-dessous à ce fichier.

1<?php

2

3use Symfony\Component\HttpFoundation\Request;

4use MicroCMS\Domain\Comment;

5use MicroCMS\Domain\Article;

6use MicroCMS\Form\Type\CommentType;

7use MicroCMS\Form\Type\ArticleType;

8

9// ...

10

11// Add a new article

12$app->match('/admin/article/add', function(Request $request) use ($app) {

13    $article = new *Article*();

14    $articleForm = $app['form.factory']->create(new *ArticleType*(), $article);

15    $articleForm->handleRequest($request);

16    if ($articleForm->isSubmitted() && $articleForm->isValid()) {

17        $app['dao.article']->save($article);

18        $app['session']->getFlashBag()->add('success', 'The article was successfully created.');

19    }

20    return $app['twig']->render('article\_form.html.twig', array(

21        'title' => 'New article',

22        'articleForm' => $articleForm->createView()));

23});

24

25// Edit an existing article

26$app->match('/admin/article/{id}/edit', function($id, Request $request) use ($app) {

27    $article = $app['dao.article']->find($id);

28    $articleForm = $app['form.factory']->create(new *ArticleType*(), $article);

29    $articleForm->handleRequest($request);

30    if ($articleForm->isSubmitted() && $articleForm->isValid()) {

31        $app['dao.article']->save($article);

32        $app['session']->getFlashBag()->add('success', 'The article was succesfully updated.');

33    }

34    return $app['twig']->render('article\_form.html.twig', array(

35        'title' => 'Edit article',

36        'articleForm' => $articleForm->createView()));

37});

38

39// Remove an article

40$app->get('/admin/article/{id}/delete', function($id, Request $request) use ($app) {

41    // Delete all associated comments

42    $app['dao.comment']->deleteAllByArticle($id);

43    // Delete the article

44    $app['dao.article']->delete($id);

45    $app['session']->getFlashBag()->add('success', 'The article was succesfully removed.');

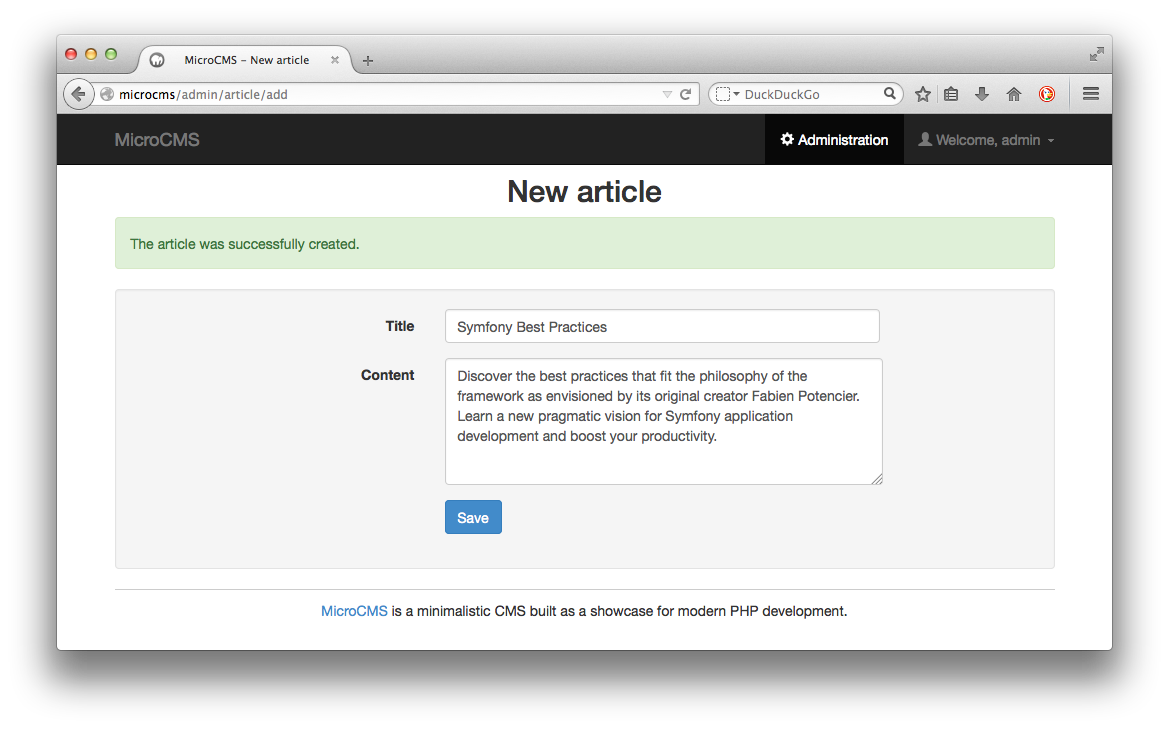
46    return $app->redirect('/admin');

47});

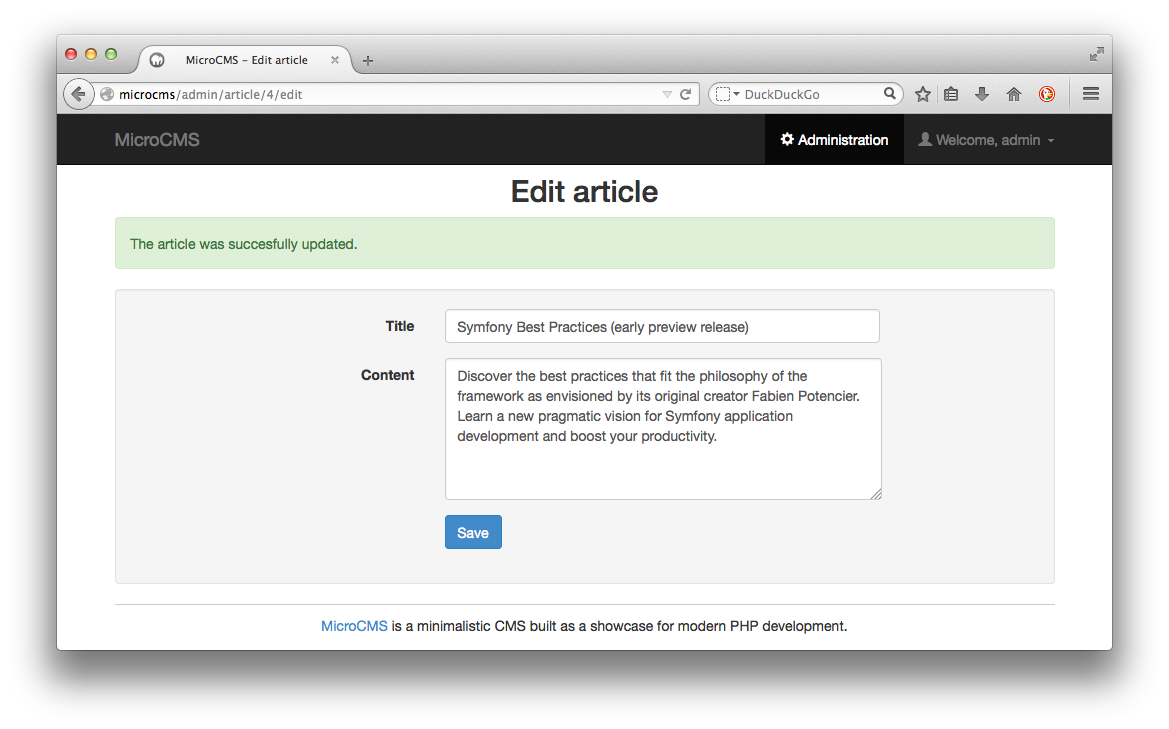
Les contrôleurs de création et de modification sont similaires. L'un crée un nouvel article, alors que l'autre le récupère dans la base de données à partir de l'identifiant passé en paramètre dans l'URL. Tous deux utilisent le formulaire ArticleType et la vue article\_form.html.twig définis précédemment.

Le contrôleur de suppression détruit l'article passé en paramètre de l'URL après avoir supprimé les commentaires associés. Il redirige ensuite le client vers la page d'accueil du back-office.

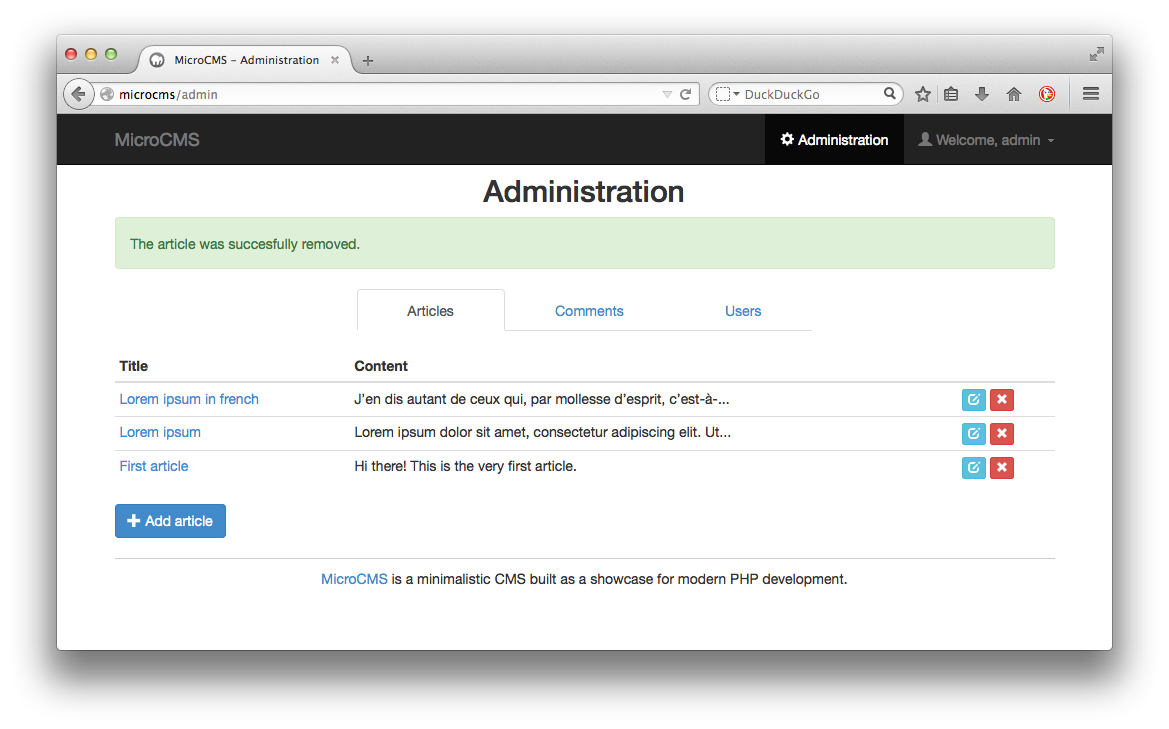
Il est temps de tester nos modifications. En tant qu'administrateur, essayez d'ajouter un nouvel article.



Vous pouvez ensuite le modifier...



Et enfin le supprimer.



Gestion des commentaires

La gestion des commentaires suit le même modèle que celles des articles. Le formulaireCommentType existe déjà : nous l'avions créé lors de l'itération 9, pour ajouter un commentaire à un article. Ce formulaire sera affiché par la vue existante views/article.html.twig et par la nouvelle vueviews/comment\_form.html.twig : on constate ici l'intérêt d'isoler la définition des formulaires dans des classes dédiées.

Créez le fichier views/comment\_form.html.twig et donnez-lui le contenu suivant.

1{% extends 'layout.html.twig' %}

2{% set adminMenu = true %}

3

4{% block title %}{{ title }}{% endblock %}

5

6{% block content %}

7<h2 class="text-center">{{ block('title') }}</h2>

8{% for flashMessage in app.session.flashbag.get('success') %}

9<div class="alert alert-success">

10    {{ flashMessage }}

11</div>

12{% endfor %}

13

14<div class="well">

15{{ form\_start(commentForm, { 'attr': {'class': 'form-horizontal'} }) }}

16    <div class="form-group">

17        {{ form\_label(commentForm.content, null, { 'label\_attr':  {

18            'class': 'col-sm-4 control-label'

19        }}) }}

20        <div class="col-sm-6">

21            {{ form\_errors(commentForm.content) }}

22            {{ form\_widget(commentForm.content, { 'attr':  {

23                'class': 'form-control',

24                'rows': '4'

25            }}) }}

26        </div>

27    </div>

28    <div class="form-group">

29        <div class="col-sm-offset-4 col-sm-3">

30            <input type="submit" class="btn btn-primary" value="Save" />

31        </div>

32    </div>

33{{ form\_end(commentForm) }}

34</div>

35{% endblock %}

La méthode de sauvegarde d'un commentaire existe déjà dans la classe CommentDAO. Modifiez le fichiersrc/DAO/CommentDAO.php pour y ajouter la méthode de recherche d'un commentaire et la méthode de suppression d'un commentaire définies ci-dessous.

1<?php

2

3// ...

4

5class CommentDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Returns a comment matching the supplied id.

11     \*

12     \* @param integer $id The comment id

13     \*

14     \* @return \MicroCMS\Domain\Comment|throws an exception if no matching comment is found

15     \*/

16    public function find($id) {

17        $sql = "select \* from t\_comment where com\_id=?";

18        $row = $this->getDb()->fetchAssoc($sql, array($id));

19

20        if ($row)

21            return $this->buildDomainObject($row);

22        else

23            throw new \Exception("No comment matching id " . $id);

24    }

25

26    // ...

27

28    /\*\*

29     \* Removes a comment from the database.

30     \*

31     \* @param @param integer $id The comment id

32     \*/

33    public function delete($id) {

34        // Delete the comment

35        $this->getDb()->delete('t\_comment', array('com\_id' => $id));

36    }

37

38    // ...

39}

Enfin, ajoutez les contrôleurs ci-dessous à la fin du fichier app/routes.php.

1<?php

2

3// ...

4

5// Edit an existing comment

6$app->match('/admin/comment/{id}/edit', function($id, Request $request) use ($app) {

7    $comment = $app['dao.comment']->find($id);

8    $commentForm = $app['form.factory']->create(new *CommentType*(), $comment);

9    $commentForm->handleRequest($request);

10    if ($commentForm->isSubmitted() && $commentForm->isValid()) {

11        $app['dao.comment']->save($comment);

12        $app['session']->getFlashBag()->add('success', 'The comment was succesfully updated.');

13    }

14    return $app['twig']->render('comment\_form.html.twig', array(

15        'title' => 'Edit comment',

16        'commentForm' => $commentForm->createView()));

17});

18

19// Remove a comment

20$app->get('/admin/comment/{id}/delete', function($id, Request $request) use ($app) {

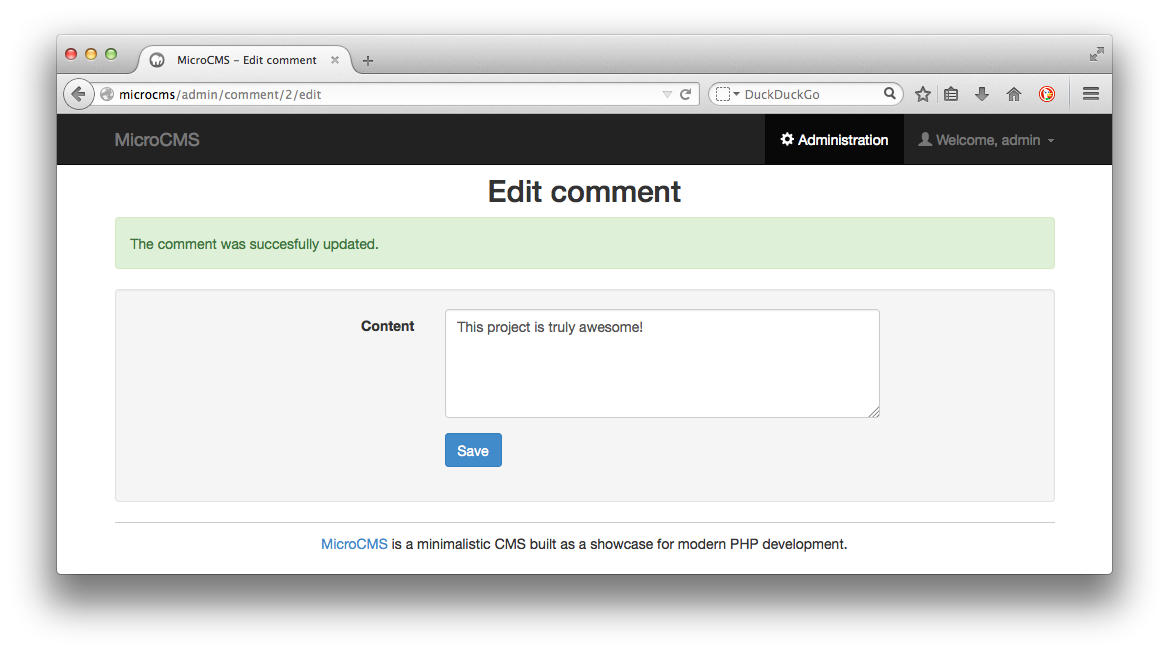
21    $app['dao.comment']->delete($id);

22    $app['session']->getFlashBag()->add('success', 'The comment was succesfully removed.');

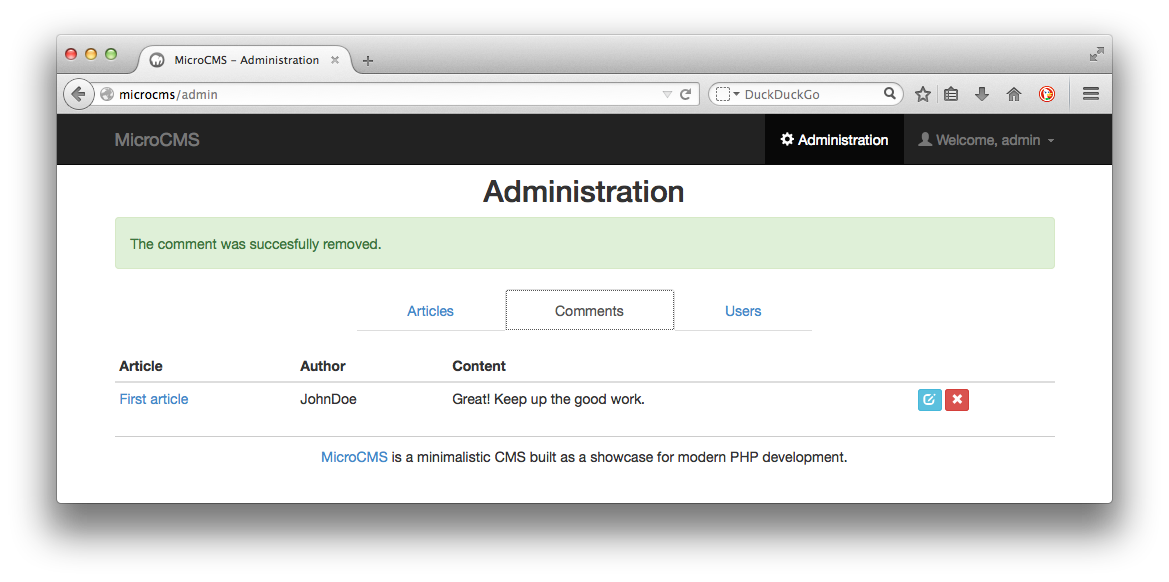
23    return $app->redirect('/admin');

24});

Vous pouvez à présent tester, en tant qu'administrateur, la modification d'un commentaire.



Vous pouvez également supprimer un commentaire existant.



Gestion des utilisateurs

Il ne nous reste plus qu'à implémenter la gestion des utilisateurs pour finaliser le back-office. Commençons par définir le formulaire associé à un utilisateur dans le fichiersrc/Form/Type/UserType.php.

1<?php

2

3namespace MicroCMS\Form\Type;

4

5use Symfony\Component\Form\AbstractType;

6use Symfony\Component\Form\FormBuilderInterface;

7

8class UserType extends AbstractType

9{

10    public function buildForm(FormBuilderInterface $builder, array $options)

11    {

12        $builder

13            ->add('username', 'text')

14            ->add('password', 'repeated', array(

15                'type'            => 'password',

16                'invalid\_message' => 'The password fields must match.',

17                'options'         => array('required' => true),

18                'first\_options'   => array('label' => 'Password'),

19                'second\_options'  => array('label' => 'Repeat password'),

20            ))

21            ->add('role', 'choice', array(

22                'choices' => array('ROLE\_ADMIN' => 'Admin', 'ROLE\_USER' => 'User')

23            ));

24    }

25

26    public function getName()

27    {

28        return 'user';

29    }

30}

Dans la plupart des applications web, les mots de passe des utilisateurs sont saisis deux fois pour éviter les erreurs de saisie. Symfony supporte cette fonctionnalité : le type de champ repeated permet de faire saisir un champ deux fois et effectue automatiquement la comparaison des deux valeurs. Le message défini par invalid\_message apparaîtra si les deux valeurs sont différentes. Nous définissons le champrole sous la forme d'une liste (choice). Les deux valeurs possibles sont ROLE\_ADMIN et ROLE\_USER. Nous observerons plus loin comment Symfony affiche et valide ce type de champ.

Le type de champ repeated implique l'utilisation du composant Symfony validator que nous avons intégré à l'application au début de cette itération.

Ce formulaire est utilisé par la vue views/user\_form.html.twig que voici.

1{% extends 'layout.html.twig' %}

2{% set adminMenu = true %}

3

4{% block title %}{{ title }}{% endblock %}

5

6{% block content %}

7<h2 class="text-center">{{ block('title') }}</h2>

8{% for flashMessage in app.session.flashbag.get('success') %}

9<div class="alert alert-success">

10    {{ flashMessage }}

11</div>

12{% endfor %}

13{% if form\_errors(userForm.password.first) %}

14<div class="alert alert-danger">

15    {{ form\_errors(userForm.password.first) }}

16</div>

17{% endif %}

18

19<div class="well">

20{{ form\_start(userForm, { 'attr': {'class': 'form-horizontal'} }) }}

21    <div class="form-group">

22        {{ form\_label(userForm.username, null, { 'label\_attr':  {

23            'class': 'col-sm-5 control-label'

24        }}) }}

25        <div class="col-sm-4">

26            {{ form\_errors(userForm.username) }}

27            {{ form\_widget(userForm.username, { 'attr':  {

28                'class': 'form-control'

29            }}) }}

30        </div>

31    </div>

32    <div class="form-group">

33        {{ form\_label(userForm.password.first, null, { 'label\_attr':  {

34            'class': 'col-sm-5 control-label'

35        }}) }}

36        <div class="col-sm-4">

37            {{ form\_widget(userForm.password.first, { 'attr':  {

38                'class': 'form-control'

39            }}) }}

40        </div>

41    </div>

42    <div class="form-group">

43        {{ form\_label(userForm.password.second, null, { 'label\_attr':  {

44            'class': 'col-sm-5 control-label'

45        }}) }}

46        <div class="col-sm-4">

47            {{ form\_widget(userForm.password.second, { 'attr':  {

48                'class': 'form-control'

49            }}) }}

50        </div>

51    </div>

52    <div class="form-group">

53        {{ form\_label(userForm.role, null, { 'label\_attr':  {

54            'class': 'col-sm-5 control-label'

55        }}) }}

56        <div class="col-sm-2">

57            {{ form\_errors(userForm.role) }}

58            {{ form\_widget(userForm.role, { 'attr':  {

59                'class': 'form-control'

60            }}) }}

61        </div>

62    </div>

63    <div class="form-group">

64        <div class="col-sm-offset-5 col-sm-3">

65            <input type="submit" class="btn btn-primary" value="Save" />

66        </div>

67    </div>

68{{ form\_end(userForm) }}

69</div>

70{% endblock %}

Cette vue affiche les champs du formulaire UserType. Le mot de passe est affiché sous la forme de deux champs : userForm.password.first et userForm.password.second.

Comme pour les articles, il faut ajouter les méthodes de modification et de suppression d'un utilisateur dans la classe UserDAO (fichier src/DAO/UserDAO.php).

1<?php

2

3// ...

4

5class UserDAO extends DAO implements UserProviderInterface

6{

7    // ...

8

9    /\*\*

10     \* Saves a user into the database.

11     \*

12     \* @param \MicroCMS\Domain\User $user The user to save

13     \*/

14    public function save(User $user) {

15        $userData = array(

16            'usr\_name' => $user->getUsername(),

17            'usr\_salt' => $user->getSalt(),

18            'usr\_password' => $user->getPassword(),

19            'usr\_role' => $user->getRole()

20            );

21

22        if ($user->getId()) {

23            // The user has already been saved : update it

24            $this->getDb()->update('t\_user', $userData, array('usr\_id' => $user->getId()));

25        } else {

26            // The user has never been saved : insert it

27            $this->getDb()->insert('t\_user', $userData);

28            // Get the id of the newly created user and set it on the entity.

29            $id = $this->getDb()->lastInsertId();

30            $user->setId($id);

31        }

32    }

33

34    /\*\*

35     \* Removes a user from the database.

36     \*

37     \* @param @param integer $id The user id.

38     \*/

39    public function delete($id) {

40        // Delete the user

41        $this->getDb()->delete('t\_user', array('usr\_id' => $id));

42    }

43

44    // ...

45}

Il faut également ajouter dans le fichier src/DAO/CommentDAO.php la possibilité de supprimer tous les commentaires associés à un utilisateur.

1<?php

2

3// ...

4

5class CommentDAO extends DAO

6{

7    // ...

8

9    /\*\*

10     \* Removes all comments for a user

11     \*

12     \* @param integer $userId The id of the user

13     \*/

14    public function deleteAllByUser($userId) {

15        $this->getDb()->delete('t\_comment', array('usr\_id' => $userId));

16    }

17

18    // ...

19}

Pour terminer, ajoutez les directives use et les contrôleurs nécessaires à la fin du fichierapp/routes.php.

1<?php

2

3use Symfony\Component\HttpFoundation\Request;

4use MicroCMS\Domain\Comment;

5use MicroCMS\Domain\Article;

6use MicroCMS\Domain\User;

7use MicroCMS\Form\Type\CommentType;

8use MicroCMS\Form\Type\ArticleType;

9use MicroCMS\Form\Type\UserType;

10

11// ...

12

13// Add a user

14$app->match('/admin/user/add', function(Request $request) use ($app) {

15    $user = new *User*();

16    $userForm = $app['form.factory']->create(new *UserType*(), $user);

17    $userForm->handleRequest($request);

18    if ($userForm->isSubmitted() && $userForm->isValid()) {

19        // generate a random salt value

20        $salt = substr(md5(time()), 0, 23);

21        $user->setSalt($salt);

22        $plainPassword = $user->getPassword();

23        // find the default encoder

24        $encoder = $app['security.encoder.digest'];

25        // compute the encoded password

26        $password = $encoder->encodePassword($plainPassword, $user->getSalt());

27        $user->setPassword($password);

28        $app['dao.user']->save($user);

29        $app['session']->getFlashBag()->add('success', 'The user was successfully created.');

30    }

31    return $app['twig']->render('user\_form.html.twig', array(

32        'title' => 'New user',

33        'userForm' => $userForm->createView()));

34});

35

36// Edit an existing user

37$app->match('/admin/user/{id}/edit', function($id, Request $request) use ($app) {

38    $user = $app['dao.user']->find($id);

39    $userForm = $app['form.factory']->create(new *UserType*(), $user);

40    $userForm->handleRequest($request);

41    if ($userForm->isSubmitted() && $userForm->isValid()) {

42        $plainPassword = $user->getPassword();

43        // find the encoder for the user

44        $encoder = $app['security.encoder\_factory']->getEncoder($user);

45        // compute the encoded password

46        $password = $encoder->encodePassword($plainPassword, $user->getSalt());

47        $user->setPassword($password);

48        $app['dao.user']->save($user);

49        $app['session']->getFlashBag()->add('success', 'The user was succesfully updated.');

50    }

51    return $app['twig']->render('user\_form.html.twig', array(

52        'title' => 'Edit user',

53        'userForm' => $userForm->createView()));

54});

55

56// Remove a user

57$app->get('/admin/user/{id}/delete', function($id, Request $request) use ($app) {

58    // Delete all associated comments

59    $app['dao.comment']->deleteAllByUser($id);

60    // Delete the user

61    $app['dao.user']->delete($id);

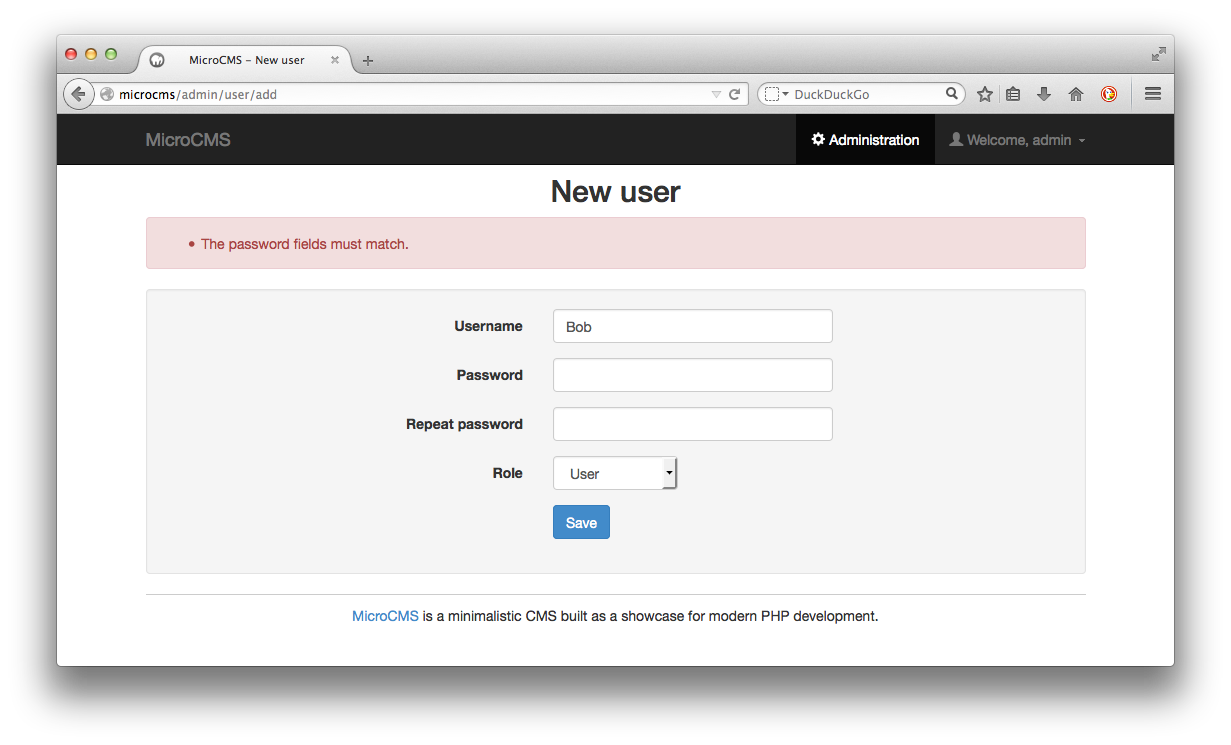
62    $app['session']->getFlashBag()->add('success', 'The user was succesfully removed.');

63    return $app->redirect('/admin');

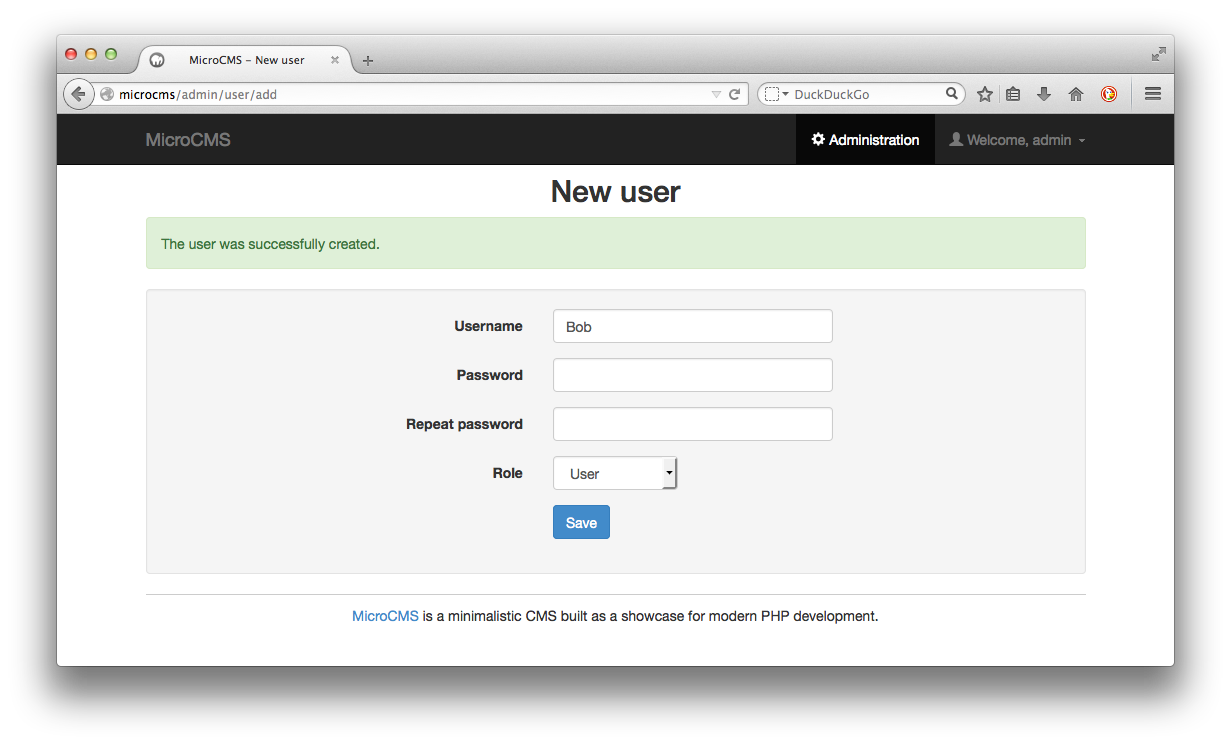
64});

Les contrôleurs de création et de modification doivent hacher le mot de passe saisi par l'utilisateur avant de le sauvegarder dans la base de données. Ils utilisent pour cela les services$app['security.encoder.digest'] et $app['security.encoder\_factory']. Le salage (variable $salt) est généré aléatoirement grâce à la fonction PHP [md5](http://php.net/manual/fr/function.md5.php).

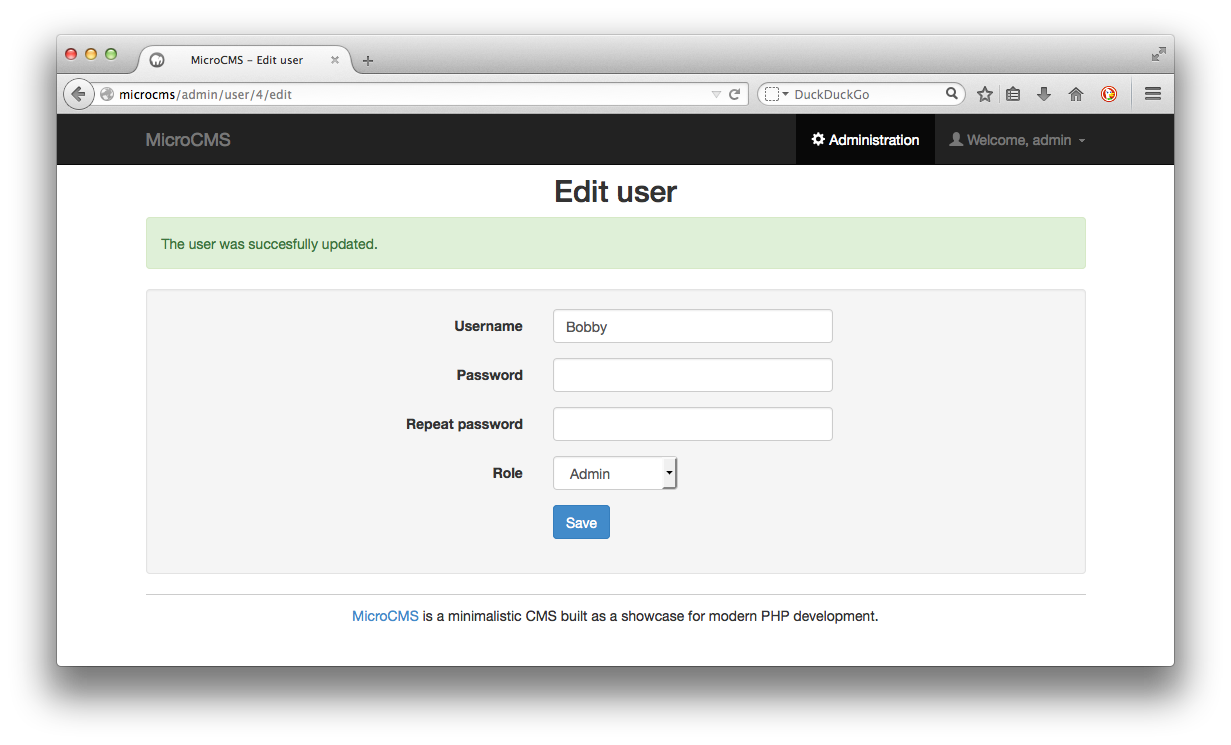
Le back-office est maintenant terminé. Vous pouvez le tester en vous connectant à l'application en tant qu'administrateur, puis en tentant d'ajouter un nouvel utilisateur. Commencez par saisir deux mots de passe différents : l'application affiche le message d'erreur approprié. Merci Symfony !



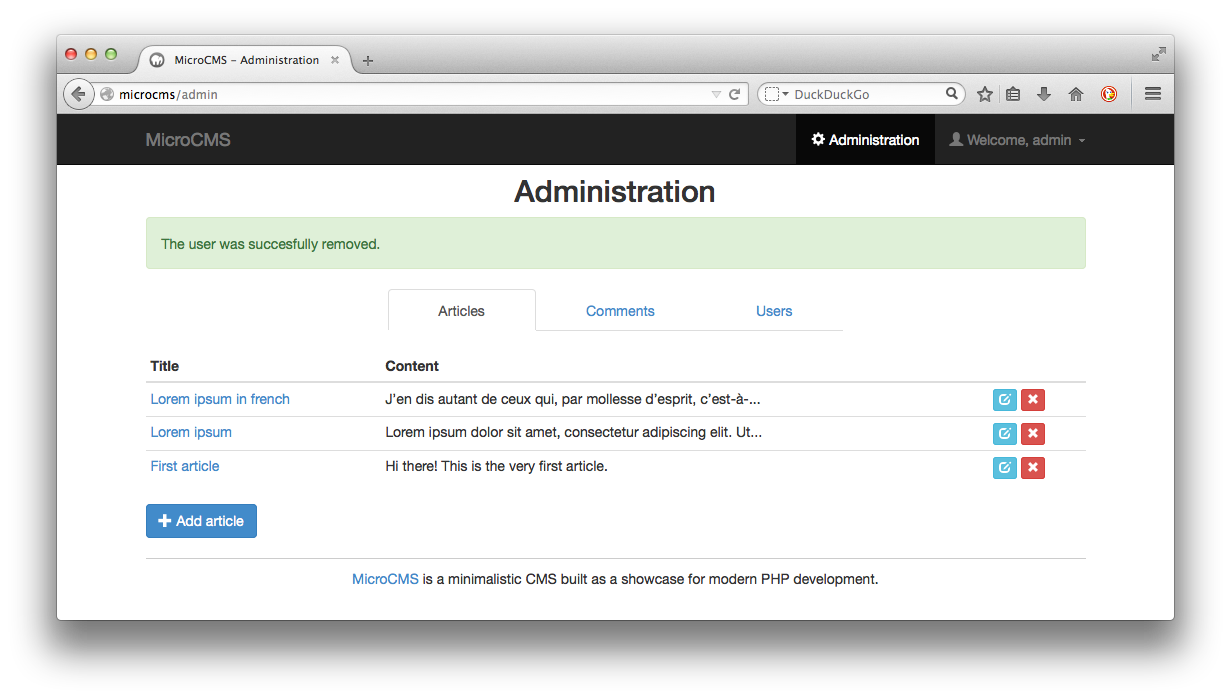
On remarque également que le rôle (type de champ choice) est affiché sous la forme d'une liste déroulante. Saisissez ensuite le même mot de passe deux fois : l'utilisateur est cette fois-ci bien créé.



Vous pouvez ensuite modifier ses propriétés...



Et enfin le supprimer.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-10).

Bilan

Cette longue itération nous a permis d'intégrer à l'application des fonctionnalités d'administration. Les utilisateurs disposant du rôle d'administrateur peuvent à présent ajouter, modifier et supprimer des données dans la base. Pour cela, nous avons exploité les possibilités du framework Bootstrap pour obtenir un affichage moderne. Nous avons également découvert certaines fonctionnalités de Symfony en matière de sécurité (gestion des rôles) et de validation des formulaires (comparaison des mots de passe).

ITÉRATION 11 : PRÉPARATION POUR LA PRODUCTION

Le but de cette itération est d'intégrer à l'application plusieurs améliorations dans l'objectif d'une future mise en production.

Ajout de tests fonctionnels

**Pourquoi tester ?**

La problématique des tests est souvent considérée comme secondaire et négligée par les développeurs débutants. C'est une erreur : lorsqu'on livre une application et qu'elle est placée en **production** (offerte à ses utilisateurs), il est essentiel d'avoir un maximum de garanties sur son bon fonctionnement afin d'éviter au maximum de coûteuses mauvaises surprises.

Le test d'une application peut être manuel. Dans ce cas, une personne effectue sur l'application une suite d'opérations prévue à l'avance (navigation, connexion, envoi d'informations...) pour vérifier qu'elle possède bien le comportement attendu. C'est un processus coûteux en temps et sujets aux erreurs (oublis, négligences, etc.).

En complément de ces tests manuels, on a tout intérêt à intégrer à un projet logiciel des tests automatisés qui pourront être lancés aussi souvent que nécessaire. Ceci est d'autant plus vrai pour les méthodologies agiles basées sur un développement itératif et des livraisons fréquentes, ou bien lorsque l'on met en place une [intégration continue](http://fr.wikipedia.org/wiki/Int%C3%A9gration_continue).

**Comment tester ?**

On peut employer différentes stratégies pour automatiser le test d'une application. Parmi les types de test possibles, citons les [tests unitaires](http://fr.wikipedia.org/wiki/Test_unitaire) qui testent individuellement chaque élément de l'application (un composant, une classe, etc) et les **tests fonctionnels** qui vérifient le fonctionnement global de l'application.

Quelle que soit la stratégie choisie, l'écriture de tests est une activité chronophage et parfois délicate. En suivant les recommandations des [bonnes pratiques Symfony](http://symfony.com/doc/current/best_practices/tests.html), nous allons nous contenter d'écrire des tests fonctionnels simples. Ces tests vérifieront uniquement que l'application répond sans erreur aux différentes routes possibles.

**Composants nécessaires**

Nous allons créer nos tests à l'aide de [PHPUnit](https://phpunit.de/), l'outil le plus fréquemment employé pour écrire des tests en PHP. Pour récupérer PHPUnit ainsi que les composants Symfony nécessaires, il faut modifier le fichiercomposer.json. Éditez ce fichier pour qu'il ait le contenu suivant.

1{

2    "require": {

3        ...

4    },

5    "require-dev": {

6        "phpunit/phpunit": "~4.3",

7        "symfony/browser-kit": "~2.4",

8        "symfony/css-selector": "~2.4"

9    },

10    "autoload": {

11        ...

12    }

13}

On utilise ici require-dev pour définir les dépendances nécessaires uniquement pendant le développement. Lors de la mise en production de l'application, on utilisera Composer avec l'option --no-dev pour ne pas installer ces dépendances.

Il ne reste plus qu'à récupérer ces composants via la commande habituelle :

composer update

**Classe de test**

Le framework Silex permet d'écrire des tests PHPUnit sous la forme de classes dérivées de [WebTestCase](https://github.com/silexphp/Silex/blob/master/src/Silex/WebTestCase.php).

Dans le répertoire racine de l'application, créez l'arborescence de répertoires tests/Tests. Dans ce dernier répertoire, ajoutez un fichier nommé AppTest.php ayant le contenu suivant.

1<?php

2

3namespace MicroCMS\Tests;

4

5require\_once \_\_DIR\_\_.'/../../vendor/autoload.php';

6

7use Silex\WebTestCase;

8

9class AppTest extends WebTestCase

10{

11    /\*\*

12     \* Basic, application-wide functional test inspired by Symfony best practices.

13     \* Simply checks that all application URLs load successfully.

14     \* During test execution, this method is called for each URL returned by the provideUrls method.

15     \*

16     \* @dataProvider provideUrls

17     \*/

18    public function testPageIsSuccessful($url)

19    {

20        $client = $this->createClient();

21        $client->request('GET', $url);

22

23        $this->assertTrue($client->getResponse()->isSuccessful());

24    }

25

26    /\*\*

27     \* {@inheritDoc}

28     \*/

29    public function createApplication()

30    {

31        $app = new \Silex\Application();

32

33        require \_\_DIR\_\_.'/../../app/config/dev.php';

34        require \_\_DIR\_\_.'/../../app/app.php';

35        require \_\_DIR\_\_.'/../../app/routes.php';

36

37        // Generate raw exceptions instead of HTML pages if errors occur

38        $app['exception\_handler']->disable();

39        // Simulate sessions for testing

40        $app['session.test'] = true;

41        // Enable anonymous access to admin zone

42        $app['security.access\_rules'] = array();

43

44        return $app;

45    }

46

47    /\*\*

48     \* Provides all valid application URLs.

49     \*

50     \* @return array The list of all valid application URLs.

51     \*/

52    public function provideUrls()

53    {

54        return array(

55            array('/'),

56            array('/article/1'),

57            array('/login'),

58            array('/admin'),

59            array('/admin/article/add'),

60            array('/admin/article/1/edit'),

61            array('/admin/comment/1/edit'),

62            array('/admin/user/add'),

63            array('/admin/user/1/edit'),

64            );

65    }

66}

Ce fichier définit une classe AppTest dérivée de WebTestCase. Sa méthodecreateApplication instancie, configure et renvoie notre application Silex. Sa méthodeprovideUrls définit toutes les URL à tester : elles correspondent aux routes de notre application accessibles via la commande HTTP GET. Les routes testées ici correspondent à la page d'accueil, l'affichage de l'article ayant l'identifiant 1, le formulaire de connexion et les différentes pages du back-office.

Enfin, la méthode testPageIsSuccessful de la classe AppTest instancie un client et vérifie (méthode PHPUnit AssertTrue) que la réponse HTTP renvoyée pour chaque URL à tester indique un succès. Dans le cas contraire, le test échouera.

Lors du lancement d'un test, PHPUnit exécute automatiquement les méthodes commençant partest.

**Configuration pour les tests**

Afin de faciliter le lancement des tests, on ajoute à l'application un fichier de configuration PHPUnit nommé phpunit.xml.dist. Créez ce fichier dans le répertoire racine et ajoutez-lui le contenu suivant.

1<?xml version="1.0" encoding="UTF-8"?>

2

3<!-- http://www.phpunit.de/manual/current/en/appendixes.configuration.html -->

4<phpunit

5    backupGlobals               = "false"

6    backupStaticAttributes      = "false"

7    colors                      = "true"

8    convertErrorsToExceptions   = "true"

9    convertNoticesToExceptions  = "true"

10    convertWarningsToExceptions = "true"

11    processIsolation            = "false"

12    stopOnFailure               = "false"

13    syntaxCheck                 = "false"

14    bootstrap                   = "vendor/autoload.php">

15

16    <testsuites>

17        <testsuite name="MicroCMS">

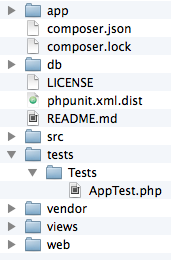
18            <directory>tests</directory>

19        </testsuite>

20    </testsuites>

21</phpunit>

Grâce à ce fichier, PHPUnit utilisera toutes les classes de test se trouvant dans le répertoire tests ainsi que ses sous-répertoires. L'arborescence de notre projet est maintenant la suivante.

Arborescence des tests du projet

Modifiez ensuite le fichier app/config/dev.php comme indiqué ci-dessous.

1<?php

2

3// Doctrine (db)

4$app['db.options'] = array(

5    'driver'   => 'pdo\_mysql',

6    'charset'  => 'utf8',

7    'host'     => '127.0.0.1',  // Mandatory for PHPUnit testing

8    'port'     => '3306',

9    'dbname'   => 'microcms',

10    'user'     => 'microcms\_user',

11    'password' => 'secret',

12);

13

14// enable the debug mode

15$app['debug'] = true;

Plutôt que d'importer comme auparavant le contenu de app/config/prod.php, on définit les paramètres de connexion à la base de données en utilisant 127.0.0.1 plutôt que localhost. Cela permet d'éviter un problème de connexion à MySQL pendant l'exécution des tests.

**Exécution des tests**

Si vous avez supprimé l'article ou le commentaire portant l'identifiant 1 en utilisant le back-office, certains tests échoueront. Dans ce cas, relancez les scripts db/structure.sql etdb/content.sql pour réinitialiser le contenu de votre base de données avant d'exécuter les tests.

Il est temps de vérifier le fonctionnement de l'application ! Dans une fenêtre de terminal, déplacez-vous dans le répertoire racine MicroCMS puis lancez la commande qui démarre l'exécution des tests. Celle-ci dépend de votre système d'exploitation.

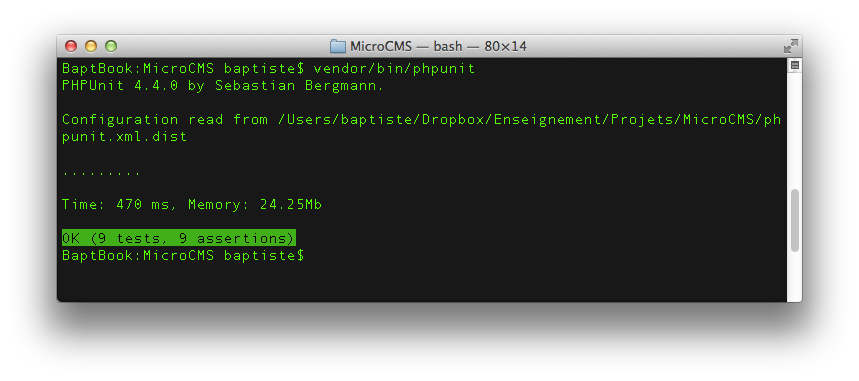
Sous Mac OS et Linux, tapez la commande ci-dessous :

vendor/bin/phpunit

Sous Windows, vous devez taper :

vendor\bin\phpunit.bat

Vous devriez obtenir un résultat similaire au suivant.



Ce résultat indique que 9 tests sur 9 ont réussi. Cela signifie que notre application répond avec succès aux requêtes vers les URL définies plus haut dans la méthode provideUrls. Même si ces tests ne couvrent pas tous les cas de figure et ne vérifient pas l'intégralité du code source, ils permettront d'éviter les erreurs majeures. On parle en anglais de [smoke testing](http://en.wikipedia.org/wiki/Smoke_testing_%5C(software%5C)).

Pour que ces tests gardent leur pertinence lorsque l'application évoluera, il faudra penser à modifier la méthode provideUrls de la classe AppTest pour ajouter les nouvelles URL à tester.

Journalisation et débogage

La **journalisation** consiste à mémoriser, le plus souvent dans un ou plusieurs fichiers textes, les événements qui se produisent lors du fonctionnement d'une application.

La journalisation peut fournir une aide bienvenue lors des phases de débogage. Elle est de plus très facile à mettre en oeuvre grâce à l'intégration par Silex de la librairie [Monolog](https://github.com/Seldaek/monolog). Pour le débogage, Symfony fournit une barre d'outils très utile qui peut également être intégrée avec Silex grâce à un [fournisseur de services](https://github.com/silexphp/Silex-WebProfiler)dédié.

Comme d'habitude, nous commençons par déclarer les dépendances nécessaires dans le fichiercomposer.json.

1{

2    "require": {

3        ...,

4        "symfony/monolog-bridge": "~2.4",

5        "silex/web-profiler": "~1.0"

6    },

7    ...

8}

Ensuite, on récupère les composants monolog-bridge et web-profiler avec Composer.

composer update

Il faut maintenant configurer notre application Silex pour qu'elle utilise ces composants. Le comportement souhaité dépend du contexte :

* En phase de développement, l'application doit produire plus d'événements de journalisation (pour aider au débogage) et la barre d'outils Symfony doit être activée.
* En phase de production, le nombre d'événements de journalisation doit être réduit pour limiter les ralentissements et la barre d'outils Symfony doit être désactivée.

Pour obtenir de comportement, ajoutez les lignes suivantes à la fin du fichier app/config/prod.php.

1<?php

2

3// ...

4

5// define log level

6$app['monolog.level'] = 'WARNING';

La journalisation des événements se fera avec le niveau WARNING. Seuls les avertissements et les erreurs seront journalisés.

Ajoutez ensuite les lignes suivantes à la fin du fichier app/config/dev.php.

1<?php

2

3// ...

4

5// define log level

6$app['monolog.level'] = 'INFO';

On définit le niveau de journalisation à INFO, ce qui permet d'enregistrer les informations en plus des avertissements et des erreurs.

À présent, modifiez le fichier app/app.php en ajoutant les lignes ci-dessous à la fin de l'enregistrement des fournisseurs de services.

1<?php

2

3// ...

4

5// Register service providers

6// ...

7$app->register(new *Silex*\Provider\MonologServiceProvider(), array(

8    'monolog.logfile' => \_\_DIR\_\_.'/../var/logs/microcms.log',

9    'monolog.name' => 'MicroCMS',

10    'monolog.level' => $app['monolog.level']

11));

12$app->register(new *Silex*\Provider\ServiceControllerServiceProvider());

13if (isset($app['debug']) && $app['debug']) {

14    $app->register(new *Silex*\Provider\WebProfilerServiceProvider(), array(

15        'profiler.cache\_dir' => \_\_DIR\_\_.'/../var/cache/profiler'

16    ));

17}

18

19// ...

Monolog est configuré avec le niveau de journalisation défini plus haut. Les événements de journalisation seront enregistrés dans le fichier var/logs/microcms.log. La barre d'outils Symfony n'est activée que lorsque l'application est configurée pour le débogage (uniquement lorsqu'on utiliseapp/config/dev.php).

Enfin, créez dans le répertoire racine les sous-répertoires var/logs et var/cache.

Si vous omettez de créer le répertoire var/logs, l'application ne pourra pas créer le fichier de journalisation et vous obtiendrez une erreur du type "*failed to open stream*".

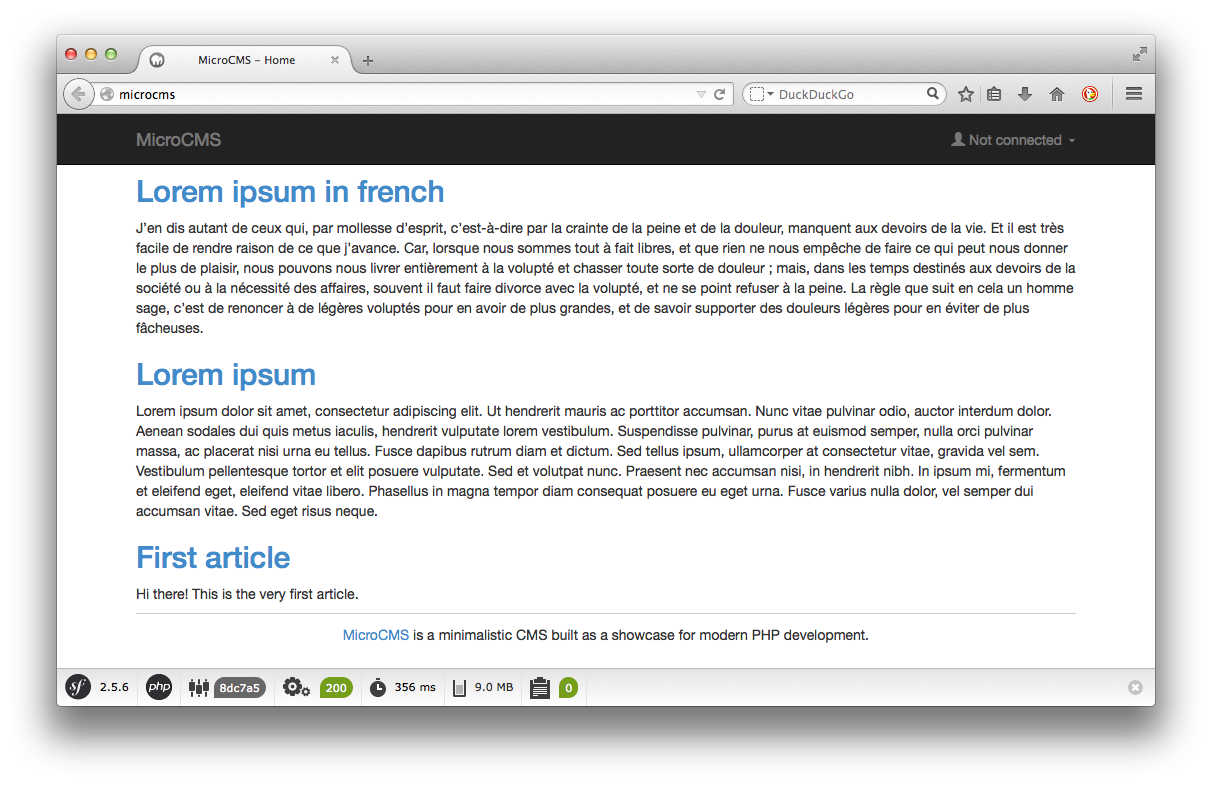
Si vous utilisez Git, ajoutez dans ces deux répertoires un fichier .gitignore avec le contenu suivant.

1\*

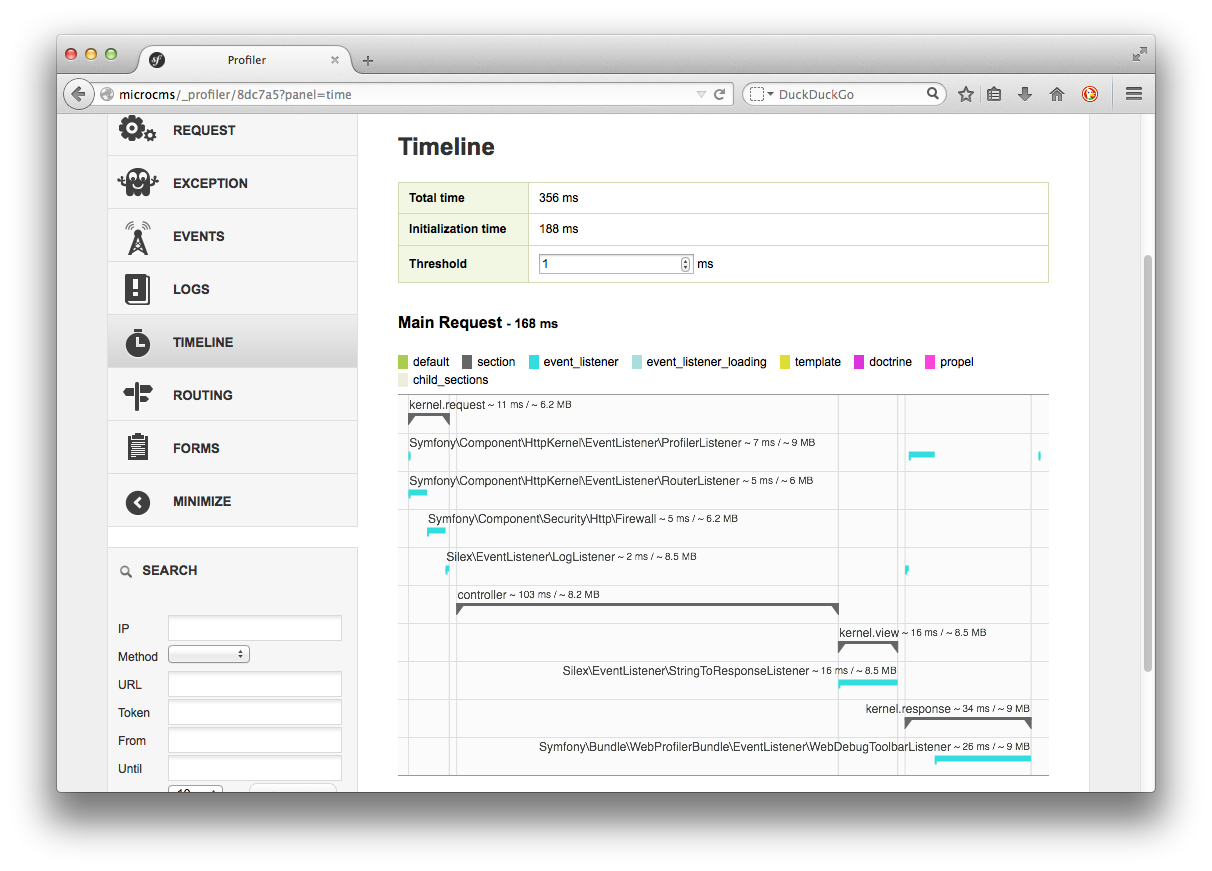
2!.gitignore

Cela permettra d'ajouter ces répertoires au contrôle de code source, tout en excluant leur contenu.

À présent, la page d'accueil de l'application [http://microcms](http://microcms/) intègre au bas de l'écran la barre d'outils Symfony.



Cette barre fournit de nombreux services, telle la *timeline* qui permet de visualiser précisément la chronologie d'une requête.



Le fichier var/logs/microcms.log enregistre les principaux événements de l'application. En voici un extrait.

1[2014-11-10 17:16:46] MicroCMS.INFO: Matched route "GET\_" (parameters: "\_controller": "{}", "\_route": "GET\_") [] []

2[2014-11-10 17:16:46] MicroCMS.INFO: > GET / [] []

3[2014-11-10 17:16:46] MicroCMS.INFO: < 200 [] []

4[2014-11-10 17:16:46] MicroCMS.INFO: Matched route "\_wdt" (parameters: "\_controller": "web\_profiler.controller. profiler:toolbarAction", "token": "c58b8c", "\_route": "\_wdt") [] []

5[2014-11-10 17:16:46] MicroCMS.INFO: > GET /\_profiler/wdt/c58b8c [] []

6[2014-11-10 17:16:47] MicroCMS.INFO: < 200 [] []

7[2014-11-10 17:16:56] MicroCMS.INFO: Matched route "\_article\_id" (parameters: "\_controller": "{}", "id": "1", "\_ route": "\_article\_id") [] []

8[2014-11-10 17:16:56] MicroCMS.INFO: > GET /article/1 [] []

9[2014-11-10 17:16:57] MicroCMS.INFO: < 200 [] []

10[2014-11-10 17:16:57] MicroCMS.INFO: Matched route "\_wdt" (parameters: "\_controller": "web\_profiler.controller. profiler:toolbarAction", "token": "c47a8b", "\_route": "\_wdt") [] []

11[2014-11-10 17:16:57] MicroCMS.INFO: > GET /\_profiler/wdt/c47a8b [] []

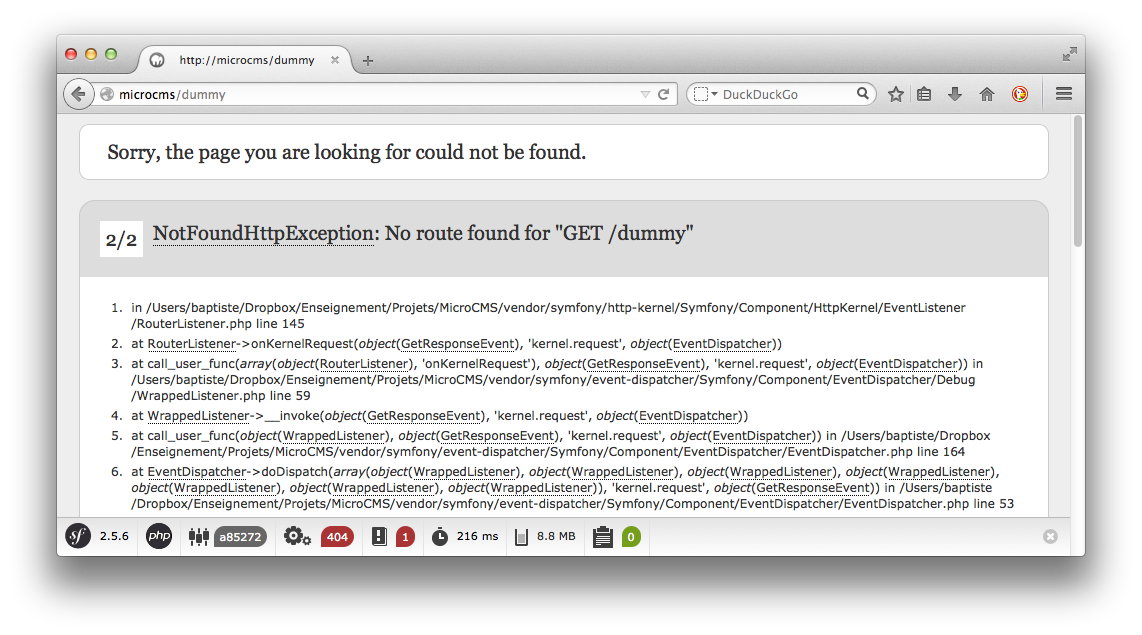
12[2014-11-10 17:16:58] MicroCMS.INFO: < 200 [] []

Pour l'instant, seuls les composants Silex/Symfony ajoutent des événements dans ce fichier. Pour que notre application en crée aussi, il suffirait de rajouter aux endroits appropriés de notre code source des appels de la forme :

1$app['monolog']->addInfo("Ceci est un évènement de test");

Gestion des erreurs

En cas d'apparition d'une erreur (ressource non trouvée, problème de connexion à la base, etc.), l'application actuelle affiche directement le message associé, ainsi que la pile des appels *(stack trace)* en configuration de débogage. Voici par exemple ce qui se produit lorsqu'on tente d'accéder à l'URL<http://microcms/dummy>, non gérée par l'application.



Ce comportement est acceptable et même pratique en phase de développement, mais pas en production. Il faudrait que l'affichage des erreurs soit homogène avec le reste des vues. Pour obtenir ce comportement, nous allons ajouter à l'application un gestionnaire d'erreurs personnalisé.

Modifiez le fichier app/app.php pour y ajouter le code ci-dessous en fin de fichier.

1<?php

2

3// ...

4

5// Register error handler

6$app->error(function (\Exception $e, $code) use ($app) {

7    switch ($code) {

8        case 403:

9            $message = 'Access denied.';

10            break;

11        case 404:

12            $message = 'The requested resource could not be found.';

13            break;

14        default:

15            $message = "Something went wrong.";

16    }

17    return $app['twig']->render('error.html.twig', array('message' => $message));

18});

Ce gestionnaire d'erreurs construit un message en fonction du code de l'erreur, puis génère la vueerror.html.twig en lui passant ce message en paramètre.

Créez le fichier views/error.html.twig avec le contenu ci-dessous.

1{% extends "layout.html.twig" %}

2

3{% block title %}Error!{% endblock %}

4

5{% block content %}

6<div class="row" id="errorPanel">

7    <div class="col-xs-5">

8        <img class="img-responsive pull-right" src="{{ app.request.basepath }}/images/404-ghost.png" alt="Error ghost"/>

9    </div>

10    <div class="col-xs-6">

11        <h1>Whoops...<br><small>{{ message }}</small></h1>

12    </div>

13</div>

14{% endblock %}

L'image 404-ghost.png fait partie du code source de la plate-forme de blogging [Ghost](https://ghost.org/). Vous pouvez la télécharger [ici](https://github.com/TryGhost/Ghost/blob/master/core/client/assets/img/404-ghost.png) puis la copier dans le répertoire web/images (à créer).

Modifiez ensuite le fichier web/css/microcms.css pour y ajouter le contenu ci-dessous, qui permet de placer correctement le message d'erreur dans la vue.

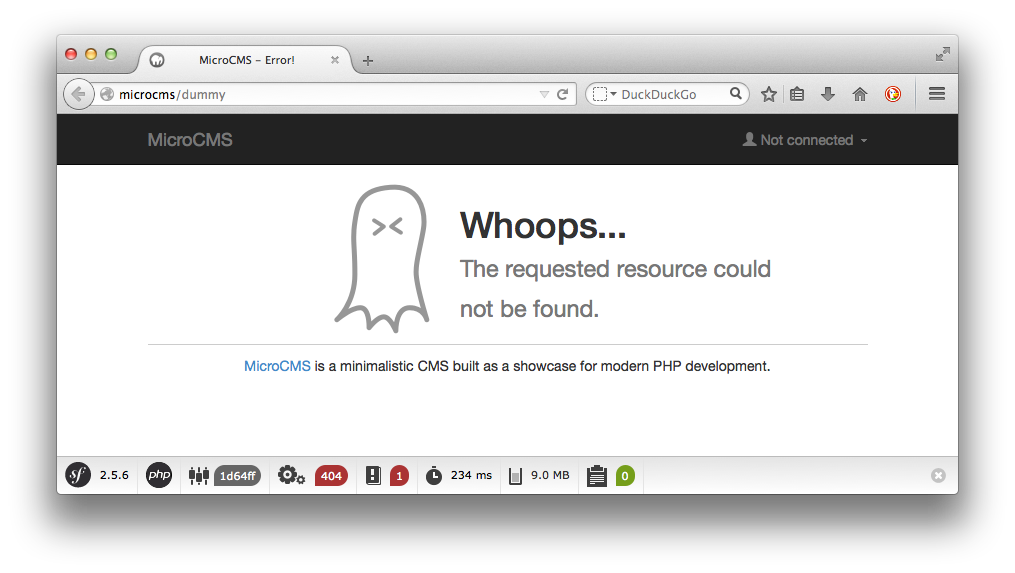
1#errorPanel {

2    *padding-top*: 30px;

3    *padding-bottom*: 10px;

4}

À présent, déclenchons volontairement une erreur en accédant à l'URL <http://microcms/dummy>. Voici le résultat obtenu.



Les informations détaillées sur l'erreur restent accessibles dans le fichier de journalisation ou (en configuration de débogage) grâce à la barre d'outils Symfony.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-11).

Mise en production

Lorsque l'application sera mise en production, il faudra utiliser le fichier app/config/prod.php à la place de app/config/dev.php dans le contrôleur frontal web/index.php.

1<?php

2

3require\_once \_\_DIR\_\_.'/../vendor/autoload.php';

4

5$app = new *Silex*\Application();

6

7require \_\_DIR\_\_.'/../app/config/prod.php'; // Config for production

8require \_\_DIR\_\_.'/../app/app.php';

9require \_\_DIR\_\_.'/../app/routes.php';

10

11$app->run();

Sur le serveur de production, il faudra installer les dépendances avec l'option --no-dev.

composer install --no-dev

Afin bénéficier de la barre d'outils Symfony, je vous conseille de continuer à utiliserapp/config/dev.php jusqu'à la fin de ce cours.

 Bilan

Au cours de cette itération, nous avons ajouté à l'application des tests automatisés qui, malgré leur simplicité, permettront d'augmenter la confiance dans son fonctionnement. Nous avons également facilité la mise au point de l'application grâce à la journalisation et à l'intégration de la barre de débogage Symfony. Enfin, l'apparition d'une erreur ne dégrade plus l'affichage.

La prochaine itération va permettre d'interagir avec notre application par le biais d'une API.

ITÉRATION 12 : AJOUT D'UNE API JSON

Le but de cette itération est d'intégrer à l'application une API utilisant le format JSON.

Fonctionnalités de l'API

Une [API](http://fr.wikipedia.org/wiki/Interface_de_programmation)*(Application Programming Interface)* est une interface offerte par une application à destination d'autres applications. Une API exposée via le protocole HTTP(S) est appelée un **service web**.

Ces dernières années, le format [JSON](http://fr.wikipedia.org/wiki/JavaScript_Object_Notation)*(JavaScript Object Notation)* a détroné XML et s'est imposé comme le nouveau standard pour les échanges de données entre applications via des services web.

Nous allons ajouter à notre application une API offrant les fonctionnalités suivantes :

* consultation de la liste des articles ;
* consultation d'un article à partir de son identifiant ;
* ajout d'un nouvel article ;
* suppression d'un article.

Création de l'API de consultation

Notre application offre déjà, via les classes DAO, les services d'accès aux données nécessaires. Il ne nous reste qu'à exposer ces services via JSON. Pour cela, ajoutez les lignes ci-dessous à la fin du fichierapp/routes.php.

1<?php

2

3// ...

4

5// API : get all articles

6$app->get('/api/articles', function() use ($app) {

7    $articles = $app['dao.article']->findAll();

8    // Convert an array of objects ($articles) into an array of associative arrays ($responseData)

9    $responseData = array();

10    foreach ($articles as $article) {

11        $responseData[] = array(

12            'id' => $article->getId(),

13            'title' => $article->getTitle(),

14            'content' => $article->getContent()

15            );

16    }

17    // Create and return a JSON response

18    return $app->json($responseData);

19});

20

21// API : get an article

22$app->get('/api/article/{id}', function($id) use ($app) {

23    $article = $app['dao.article']->find($id);

24    // Convert an object ($article) into an associative array ($responseData)

25    $responseData = array(

26        'id' => $article->getId(),

27        'title' => $article->getTitle(),

28        'content' => $article->getContent()

29        );

30    // Create and return a JSON response

31    return $app->json($responseData);

32});

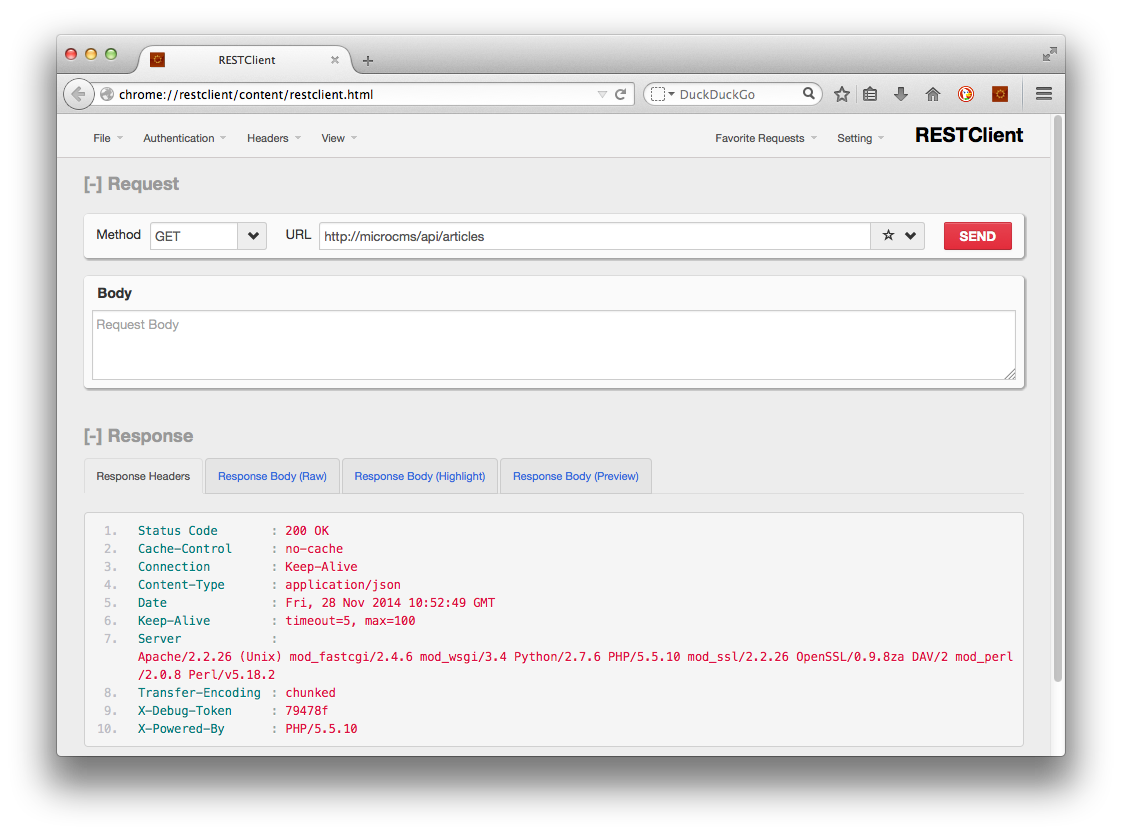
On définit ici les contrôleurs associés aux routes /api/articles et /api/article/{id}. Ces deux contrôleurs fonctionnent de manière similaire : ils utilisent le service d'accès aux articles pour récupérer les données nécessaires, puis convertissent ces données sous forme de tableaux associatifs champs/valeurs. Enfin, ils utilisent la méthode Silex $app->json pour renvoyer la réponse au format JSON.

Essayons sans tarder de vérifier si notre API fonctionne. Pour cela, il suffit d'envoyer une requête HTTP vers l'URL <http://microcms/api/articles> puis d'examiner la réponse reçue.



Nous constatons que les données sont bien renvoyées. Pour obtenir une meilleure présentation du résultat, je vous conseille d'installer une extension de navigateur comme par exemple [RESTClient](https://addons.mozilla.org/fr/firefox/addon/restclient/) pour Mozilla Firefox.

Une fois cette extension installée, on peut l'utiliser pour envoyer une requête HTTP GET vers<http://microcms/api/articles> et analyser en détail le résultat.



Les en-têtes de la réponse HTTP indiquent un succès de la requête (code 200) ainsi que le format de la réponse : application/json.



Les données reçues correspondent aux articles contenus dans notre base de données.

Création de l'API de modification

Pour créer l'API de modification (création et suppression d'un article), ajoutez les lignes suivantes à la fin du fichier app/routes.php.

1<?php

2

3// ...

4

5// API : create a new article

6$app->post('/api/article', function(Request $request) use ($app) {

7    // Check request parameters

8    if (!$request->request->has('title')) {

9        return $app->json('Missing required parameter: title', 400);

10    }

11    if (!$request->request->has('content')) {

12        return $app->json('Missing required parameter: content', 400);

13    }

14    // Build and save the new article

15    $article = new *Article*();

16    $article->setTitle($request->request->get('title'));

17    $article->setContent($request->request->get('content'));

18    $app['dao.article']->save($article);

19    // Convert an object ($article) into an associative array ($responseData)

20    $responseData = array(

21        'id' => $article->getId(),

22        'title' => $article->getTitle(),

23        'content' => $article->getContent()

24        );

25    return $app->json($responseData, 201);  // 201 = Created

26});

27

28// API : delete an existing article

29$app->delete('/api/article/{id}', function ($id, Request $request) use ($app) {

30    // Delete all associated comments

31    $app['dao.comment']->deleteAllByArticle($id);

32    // Delete the article

33    $app['dao.article']->delete($id);

34    return $app->json('No Content', 204);  // 204 = No content

35});

Le premier contrôleur répond à une requête HTTP POST vers la route /api/article. Il construit un nouvel objet de la classe Article à partir des champs title et content extraits de la requête HTTP. Le nouvel article est sauvegardé dans la base de données puis renvoyé comme résultat JSON de la requête.

La second contrôleur répond à une requête HTTP DELETE vers la route /api/article/{id}. Il supprime l'article associé par l'identifiant présent dans l'URL de la requête puis renvoie une réponse indiquant que l'article n'existe plus.

Pour pouvoir créer un nouvel article en utilisant la nouvelle API de création, il faut envoyer une requête HTTP POST contenant les données de l'article (titre et contenu) au format JSON. Voici un exemple de contenu JSON.

1{

2    "title": "Another article",

3    "content": "This article was created using the JSON API. How cool is that?"

4}

Le contenu JSON reçu doit être décodé avant de pouvoir être exploité dans le contrôleur. Pour cela, ajoutez les lignes ci-dessous au fichier app/app.php.

1<?php

2

3// ...

4use Symfony\Component\HttpFoundation\Request;

5

6// ...

7

8// Register JSON data decoder for JSON requests

9$app->before(function (Request $request) {

10    if (0 === strpos($request->headers->get('Content-Type'), 'application/json')) {

11        $data = json\_decode($request->getContent(), true);

12        $request->request->replace(is\_array($data) ? $data : array());

13    }

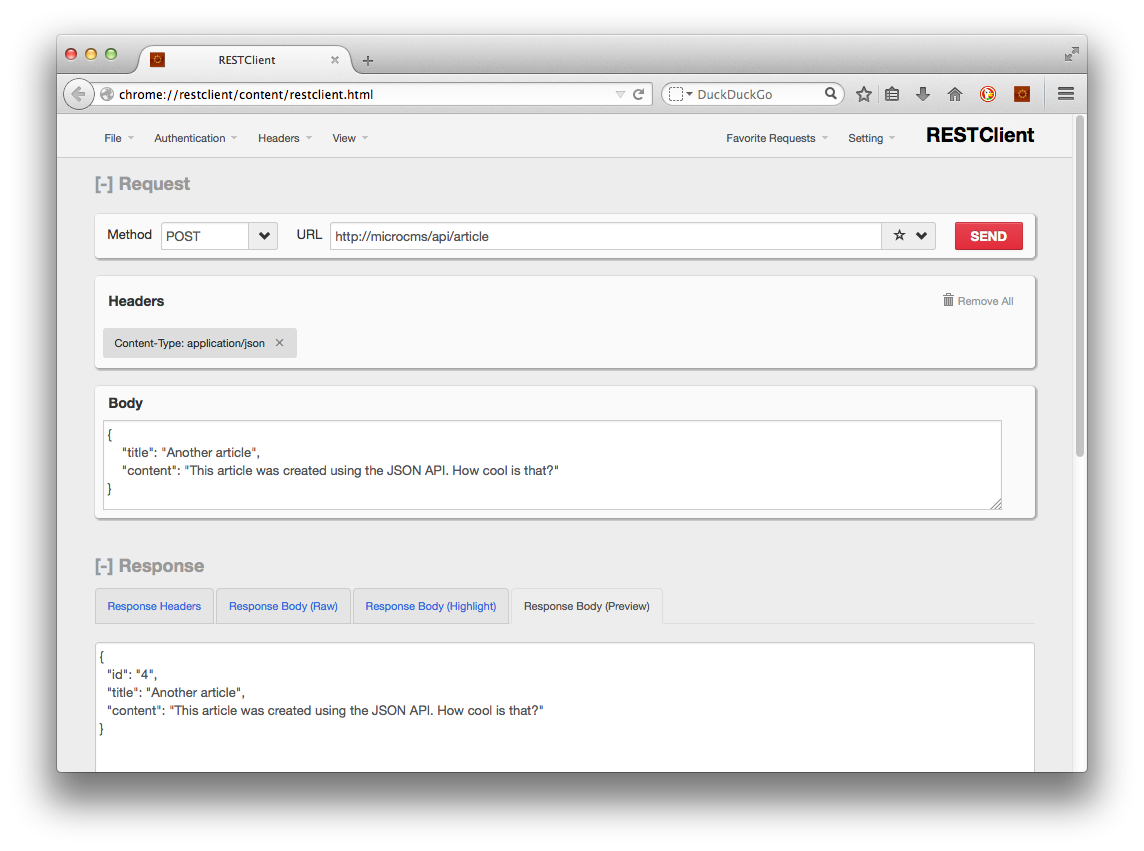
14});

Ce code intercepte toutes les requêtes reçues et, si son contenu est au format JSON, décode ce contenu et la définit comme paramètres de la requête. Ainsi, les méthodes $request->request->get('title') et $request->request->get('content') de notre contrôleur permettront d'accéder aux données JSON.

Il est temps de tester notre nouvelle API. Avant d'envoyer des requêtes avec RESTClient, il faut paramétrer l'extension pour définir application/json comme format des données envoyées. Pour cela, allez dans le menu **Headers** et choisissez l'option **Custom Header**.

Définition du format des requêtes envoyées avec RESTClient

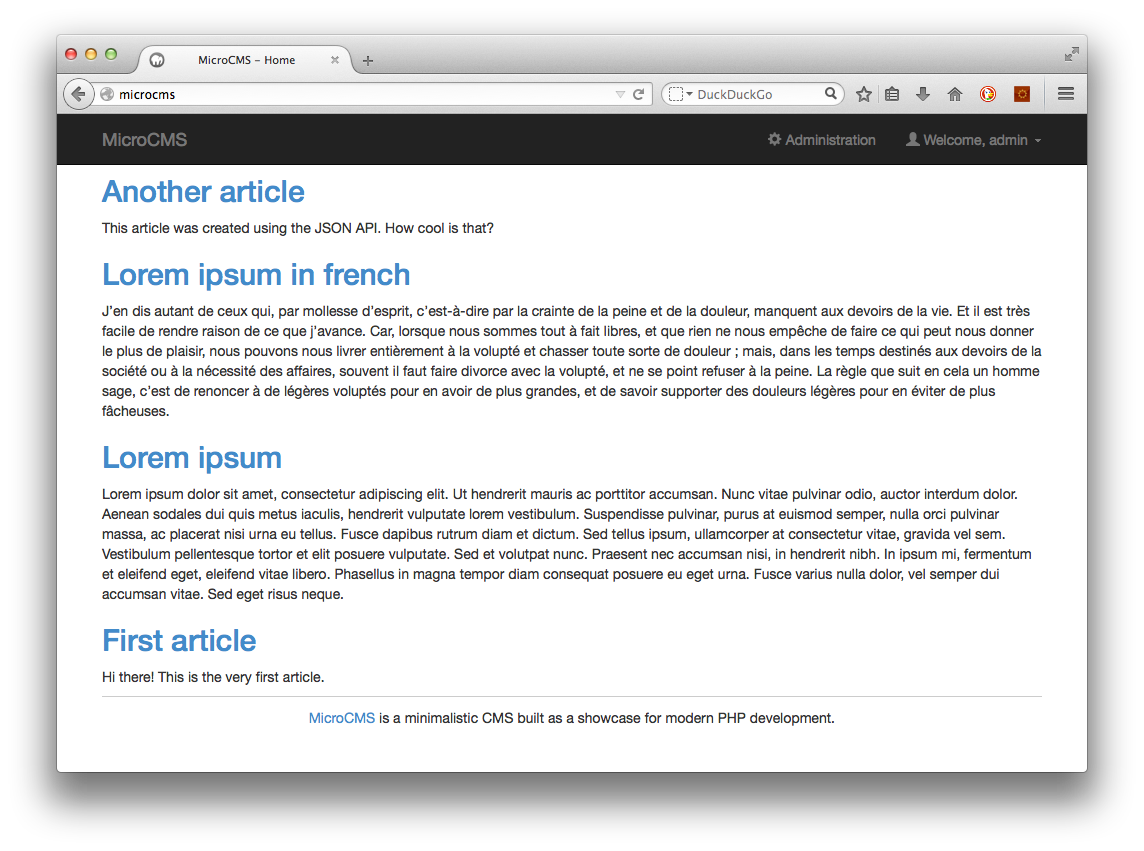
Ensuite, envoyez vers <http://microcms/api/article> une requête HTTP POST dont le corps *(body)*correspond au contenu JSON défini plus haut.



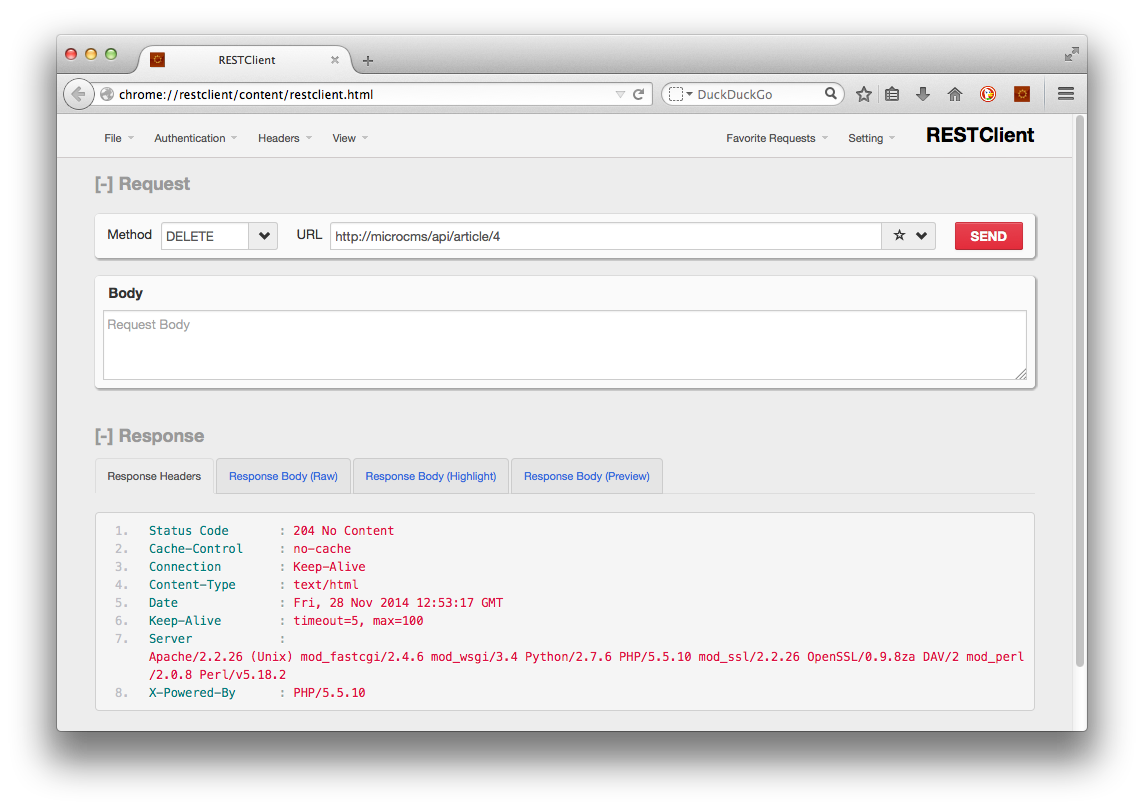
Le serveur renvoie une réponse avec le code HTTP 201, indiquant le succès de la création. Le corps de la réponse contient les données du nouvel article : son identifiant (ici : 4) a été défini au moment de l'insertion dans la base de données.

L'identifiant du nouvel article peut être différent si vous avez auparavant ajouté des articles avec le back-office.

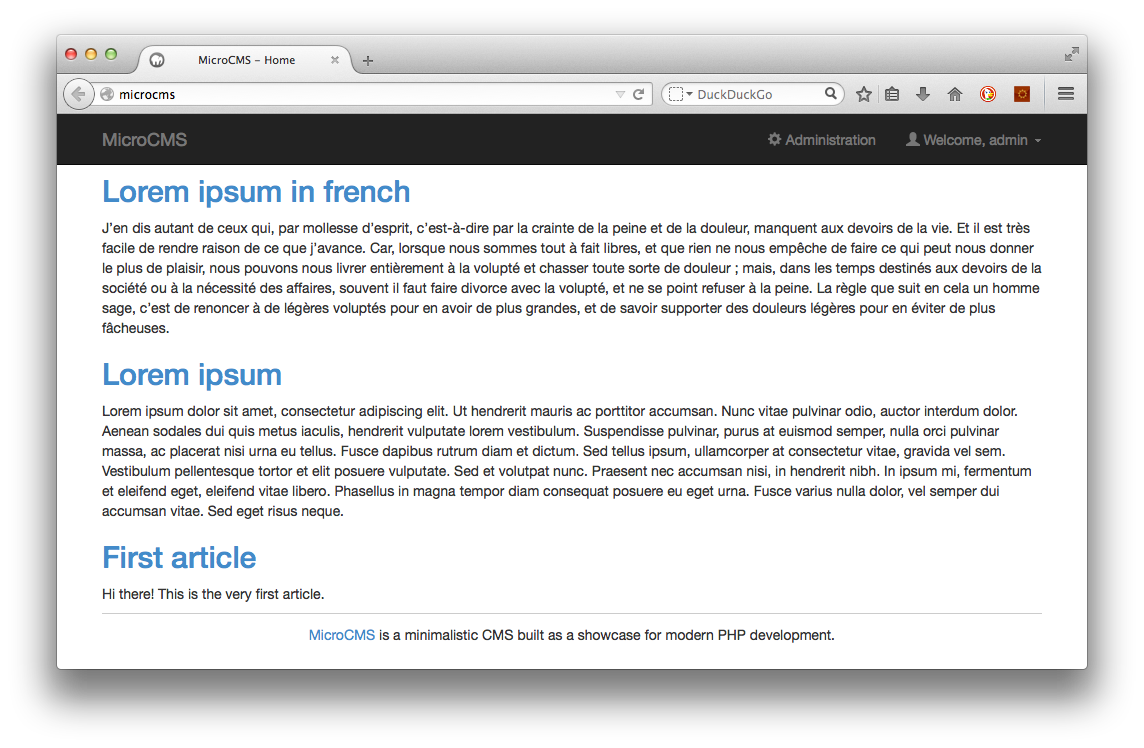
On peut vérifier en accédant à [http://microcms](http://microcms/) avec un navigateur que le nouvel article est présent.



Enfin, vérifions que notre API de suppression fonctionne en envoyant la requête HTTP DELETE vers l'URL<http://microcms/api/article/4>.



Le code 204 de la réponse HTTP indique que l'élément en question n'existe plus. On le vérifie également en accédant à [http://microcms](http://microcms/).



Mise à jour des tests fonctionnels

Enfin, il faut penser à mettre à jour nos tests fonctionnels pour ajouter les URL /api/articles et/api/article/1 permettant de tester nos API de consultation. Voici la nouvelle définition de la méthodeprovideUrls de la classe AppTest, située dans le fichier tests/MicroCMS/Tests/AppTest.php.

1<?php

2

3// ...

4

5class AppTest extends WebTestCase {

6

7    // ...

8

9    /\*\*

10     \* Provides all valid application URLs.

11     \*

12     \* @return array The list of all valid application URLs.

13     \*/

14    public function provideUrls()

15    {

16        return array(

17            array('/'),

18            array('/article/1'),

19            array('/login'),

20            array('/admin'),

21            array('/admin/article/add'),

22            array('/admin/article/1/edit'),

23            array('/admin/comment/1/edit'),

24            array('/admin/user/add'),

25            array('/admin/user/1/edit'),

26            array('/api/articles'),

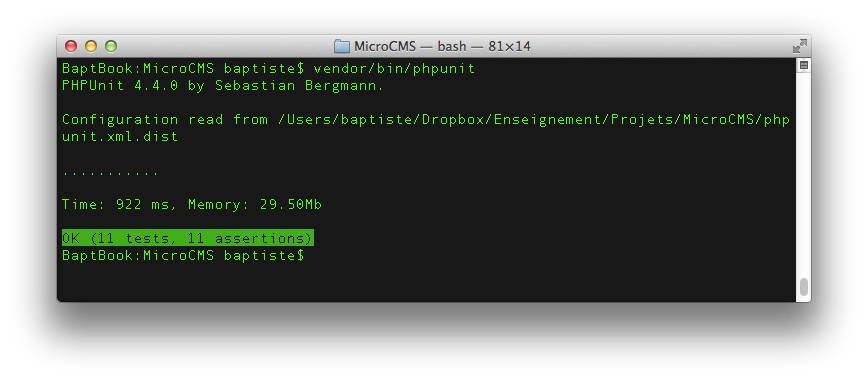
27            array('/api/article/1'),

28            );

29    }

30}

L'exécution des tests nous prouve que l'application fonctionne toujours correctement.



Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS/tree/iteration-12).

Bilan

Au cours de cette itération, nous avons découvert comment exposer certaines fonctionnalités de notre application aux développeurs extérieurs par le biais d'une API utilisant le format JSON. L'ajout de cette API a été facilitée par l'existence des services d'accès aux données (DAO) nécessaires, et le support du format JSON par Silex.

La prochaine itération va introduire ce qui sera l'architecture finale de notre application.

## ITÉRATION 13 : RÉORGANISATION DES CONTRÔLEURS

Le but de cette itération est de réorganiser les contrôleurs de notre application.

### Justification de la réorganisation

Jusqu'à présent, nous avons centralisé la définition de tous les contrôleurs dans le fichierapp/routes.php. Cette organisation a le mérite de la simplicité mais commence à montrer ses limites. Chaque nouvelle route fait augmenter la taille de ce fichier, jusqu'à rendre sa lecture difficile. Il est temps de réorganiser nos contrôleurs d'une manière plus lisible et plus logique. Ils vont devenir des **méthodes**définies dans des **classes**.

### Contrôleur d'accueil

Créez le répertoire src/Controller qui va regrouper tous nos contrôleurs. Créez ensuite le fichierHomeController.php dans ce répertoire. Donnez à ce fichier le contenu suivant.

1<?php

2

3namespace MicroCMS\Controller;

4

5use Silex\Application;

6use Symfony\Component\HttpFoundation\Request;

7use MicroCMS\Domain\Comment;

8use MicroCMS\Form\Type\CommentType;

9

10class HomeController {

11

12    /\*\*

13     \* Home page controller.

14     \*

15     \* @param Application $app Silex application

16     \*/

17    public function indexAction(Application $app) {

18        $articles = $app['dao.article']->findAll();

19        return $app['twig']->render('index.html.twig', array('articles' => $articles));

20    }

21

22    /\*\*

23     \* Article details controller.

24     \*

25     \* @param integer $id Article id

26     \* @param Request $request Incoming request

27     \* @param Application $app Silex application

28     \*/

29    public function articleAction($id, Request $request, Application $app) {

30        $article = $app['dao.article']->find($id);

31        $user = $app['security']->getToken()->getUser();

32        $commentFormView = null;

33        if ($app['security']->isGranted('IS\_AUTHENTICATED\_FULLY')) {

34        // A user is fully authenticated : he can add comments

35            $comment = new *Comment*();

36            $comment->setArticle($article);

37            $comment->setAuthor($user);

38            $commentForm = $app['form.factory']->create(new *CommentType*(), $comment);

39            $commentForm->handleRequest($request);

40            if ($commentForm->isSubmitted() && $commentForm->isValid()) {

41                $app['dao.comment']->save($comment);

42                $app['session']->getFlashBag()->add('success', 'Your comment was succesfully added.');

43            }

44            $commentFormView = $commentForm->createView();

45        }

46        $comments = $app['dao.comment']->findAllByArticle($id);

47        return $app['twig']->render('article.html.twig', array(

48            'article' => $article,

49            'comments' => $comments,

50            'commentForm' => $commentFormView));

51    }

52

53    /\*\*

54     \* User login controller.

55     \*

56     \* @param Request $request Incoming request

57     \* @param Application $app Silex application

58     \*/

59    public function loginAction(Request $request, Application $app) {

60        return $app['twig']->render('login.html.twig', array(

61            'error'         => $app['security.last\_error']($request),

62            'last\_username' => $app['session']->get('\_security.last\_username'),

63            ));

64    }

65}

Nous avons simplement déplacé dans les méthodes indexAction, articleAction et loginAction le code des contrôleurs des routes /, /article/{id} et /login.

Vous aurez peut-être remarqué que toutes nos méthodes contrôleurs se terminent par action. C'est une convention que l'on retrouve chez Symfony et dans d'autres frameworks.

### Contrôleur d'administration

Dans le répertoire src/Controller, créez le fichier AdminController.php avec le contenu suivant.

1<?php

2

3namespace MicroCMS\Controller;

4

5use Silex\Application;

6use Symfony\Component\HttpFoundation\Request;

7use MicroCMS\Domain\Article;

8use MicroCMS\Domain\User;

9use MicroCMS\Form\Type\ArticleType;

10use MicroCMS\Form\Type\CommentType;

11use MicroCMS\Form\Type\UserType;

12

13class AdminController {

14

15    /\*\*

16     \* Admin home page controller.

17     \*

18     \* @param Application $app Silex application

19     \*/

20    public function indexAction(Application $app) {

21        $articles = $app['dao.article']->findAll();

22        $comments = $app['dao.comment']->findAll();

23        $users = $app['dao.user']->findAll();

24        return $app['twig']->render('admin.html.twig', array(

25            'articles' => $articles,

26            'comments' => $comments,

27            'users' => $users));

28    }

29

30    /\*\*

31     \* Add article controller.

32     \*

33     \* @param Request $request Incoming request

34     \* @param Application $app Silex application

35     \*/

36    public function addArticleAction(Request $request, Application $app) {

37        $article = new *Article*();

38        $articleForm = $app['form.factory']->create(new *ArticleType*(), $article);

39        $articleForm->handleRequest($request);

40        if ($articleForm->isSubmitted() && $articleForm->isValid()) {

41            $app['dao.article']->save($article);

42            $app['session']->getFlashBag()->add('success', 'The article was successfully created.');

43        }

44        return $app['twig']->render('article\_form.html.twig', array(

45            'title' => 'New article',

46            'articleForm' => $articleForm->createView()));

47    }

48

49    /\*\*

50     \* Edit article controller.

51     \*

52     \* @param integer $id Article id

53     \* @param Request $request Incoming request

54     \* @param Application $app Silex application

55     \*/

56    public function editArticleAction($id, Request $request, Application $app) {

57        $article = $app['dao.article']->find($id);

58        $articleForm = $app['form.factory']->create(new *ArticleType*(), $article);

59        $articleForm->handleRequest($request);

60        if ($articleForm->isSubmitted() && $articleForm->isValid()) {

61            $app['dao.article']->save($article);

62            $app['session']->getFlashBag()->add('success', 'The article was succesfully updated.');

63        }

64        return $app['twig']->render('article\_form.html.twig', array(

65            'title' => 'Edit article',

66            'articleForm' => $articleForm->createView()));

67    }

68

69    /\*\*

70     \* Delete article controller.

71     \*

72     \* @param integer $id Article id

73     \* @param Application $app Silex application

74     \*/

75    public function deleteArticleAction($id, Application $app) {

76        // Delete all associated comments

77        $app['dao.comment']->deleteAllByArticle($id);

78        // Delete the article

79        $app['dao.article']->delete($id);

80        $app['session']->getFlashBag()->add('success', 'The article was succesfully removed.');

81        return $app->redirect('/admin');

82    }

83

84    /\*\*

85     \* Edit comment controller.

86     \*

87     \* @param integer $id Comment id

88     \* @param Request $request Incoming request

89     \* @param Application $app Silex application

90     \*/

91    public function editCommentAction($id, Request $request, Application $app) {

92        $comment = $app['dao.comment']->find($id);

93        $commentForm = $app['form.factory']->create(new *CommentType*(), $comment);

94        $commentForm->handleRequest($request);

95        if ($commentForm->isSubmitted() && $commentForm->isValid()) {

96            $app['dao.comment']->save($comment);

97            $app['session']->getFlashBag()->add('success', 'The comment was succesfully updated.');

98        }

99        return $app['twig']->render('comment\_form.html.twig', array(

100            'title' => 'Edit comment',

101            'commentForm' => $commentForm->createView()));

102    }

103

104    /\*\*

105     \* Delete comment controller.

106     \*

107     \* @param integer $id Comment id

108     \* @param Application $app Silex application

109     \*/

110    public function deleteCommentAction($id, Application $app) {

111        $app['dao.comment']->delete($id);

112        $app['session']->getFlashBag()->add('success', 'The comment was succesfully removed.');

113        return $app->redirect('/admin');

114    }

115

116    /\*\*

117     \* Add user controller.

118     \*

119     \* @param Request $request Incoming request

120     \* @param Application $app Silex application

121     \*/

122    public function addUserAction(Request $request, Application $app) {

123        $user = new *User*();

124        $userForm = $app['form.factory']->create(new *UserType*(), $user);

125        $userForm->handleRequest($request);

126        if ($userForm->isSubmitted() && $userForm->isValid()) {

127            // generate a random salt value

128            $salt = substr(md5(time()), 0, 23);

129            $user->setSalt($salt);

130            $plainPassword = $user->getPassword();

131            // find the default encoder

132            $encoder = $app['security.encoder.digest'];

133            // compute the encoded password

134            $password = $encoder->encodePassword($plainPassword, $user->getSalt());

135            $user->setPassword($password);

136            $app['dao.user']->save($user);

137            $app['session']->getFlashBag()->add('success', 'The user was successfully created.');

138        }

139        return $app['twig']->render('user\_form.html.twig', array(

140            'title' => 'New user',

141            'userForm' => $userForm->createView()));

142    }

143

144    /\*\*

145     \* Edit user controller.

146     \*

147     \* @param integer $id User id

148     \* @param Request $request Incoming request

149     \* @param Application $app Silex application

150     \*/

151    public function editUserAction($id, Request $request, Application $app) {

152        $user = $app['dao.user']->find($id);

153        $userForm = $app['form.factory']->create(new *UserType*(), $user);

154        $userForm->handleRequest($request);

155        if ($userForm->isSubmitted() && $userForm->isValid()) {

156            $plainPassword = $user->getPassword();

157            // find the encoder for the user

158            $encoder = $app['security.encoder\_factory']->getEncoder($user);

159            // compute the encoded password

160            $password = $encoder->encodePassword($plainPassword, $user->getSalt());

161            $user->setPassword($password);

162            $app['dao.user']->save($user);

163            $app['session']->getFlashBag()->add('success', 'The user was succesfully updated.');

164        }

165        return $app['twig']->render('user\_form.html.twig', array(

166            'title' => 'Edit user',

167            'userForm' => $userForm->createView()));

168    }

169

170    /\*\*

171     \* Delete user controller.

172     \*

173     \* @param integer $id User id

174     \* @param Application $app Silex application

175     \*/

176    public function deleteUserAction($id, Application $app) {

177        // Delete all associated comments

178        $app['dao.comment']->deleteAllByUser($id);

179        // Delete the user

180        $app['dao.user']->delete($id);

181        $app['session']->getFlashBag()->add('success', 'The user was succesfully removed.');

182        return $app->redirect('/admin');

183    }

184}

Comme son nom l'indique, cette classe centralise tous les contrôleurs d'administration.

### Contrôleur de l'API

Dans le répertoire src/Controller, créez le fichier ApiController.php avec le contenu suivant.

1<?php

2

3namespace MicroCMS\Controller;

4

5use Silex\Application;

6use Symfony\Component\HttpFoundation\Request;

7use MicroCMS\Domain\Article;

8

9class ApiController {

10

11    /\*\*

12     \* API articles controller.

13     \*

14     \* @param Application $app Silex application

15     \*

16     \* @return All articles in JSON format

17     \*/

18    public function getArticlesAction(Application $app) {

19        $articles = $app['dao.article']->findAll();

20        // Convert an array of objects ($articles) into an array of associative arrays ($responseData)

21        $responseData = array();

22        foreach ($articles as $article) {

23            $responseData[] = $this->buildArticleArray($article);

24        }

25        // Create and return a JSON response

26        return $app->json($responseData);

27    }

28

29    /\*\*

30     \* API article details controller.

31     \*

32     \* @param integer $id Article id

33     \* @param Application $app Silex application

34     \*

35     \* @return Article details in JSON format

36     \*/

37    public function getArticleAction($id, Application $app) {

38        $article = $app['dao.article']->find($id);

39        $responseData = $this->buildArticleArray($article);

40        // Create and return a JSON response

41        return $app->json($responseData);

42    }

43

44    /\*\*

45     \* API create article controller.

46     \*

47     \* @param Request $request Incoming request

48     \* @param Application $app Silex application

49     \*

50     \* @return Article details in JSON format

51     \*/

52    public function addArticleAction(Request $request, Application $app) {

53        // Check request parameters

54        if (!$request->request->has('title')) {

55            return $app->json('Missing required parameter: title', 400);

56        }

57        if (!$request->request->has('content')) {

58            return $app->json('Missing required parameter: content', 400);

59        }

60        // Build and save the new article

61        $article = new *Article*();

62        $article->setTitle($request->request->get('title'));

63        $article->setContent($request->request->get('content'));

64        $app['dao.article']->save($article);

65        $responseData = $this->buildArticleArray($article);

66        return $app->json($responseData, 201);  // 201 = Created

67    }

68

69    /\*\*

70     \* API delete article controller.

71     \*

72     \* @param integer $id Article id

73     \* @param Application $app Silex application

74     \*/

75    public function deleteArticleAction($id, Application $app) {

76        // Delete all associated comments

77        $app['dao.comment']->deleteAllByArticle($id);

78        // Delete the article

79        $app['dao.article']->delete($id);

80        return $app->json('No Content', 204);  // 204 = No content

81    }

82

83    /\*\*

84     \* Converts an Article object into an associative array for JSON encoding

85     \*

86     \* @param Article $article Article object

87     \*

88     \* @return array Associative array whose fields are the article properties.

89     \*/

90    private function buildArticleArray(Article $article) {

91        $data  = array(

92            'id' => $article->getId(),

93            'title' => $article->getTitle(),

94            'content' => $article->getContent()

95            );

96        return $data;

97    }

98}

Nous déplaçons dans ce fichier les contrôleurs de notre API. Au passage, l'ajout de la méthode privée buildArticleArray permet d'éliminer la duplication de code entre les contrôleurs.

Nous aurions éventuellement pu aller encore plus loin dans la refactorisation en définissant nos contrôleurs en tant que [services](http://silex.sensiolabs.org/doc/providers/service_controller.html).

### Réécriture des routes

Pour terminer, le fichier app/routes.php est profondément simplifié puisqu'il suffit à présent d'associer à chaque route la méthode appropriée des classes contrôleurs. Voici maintenant le contenu de ce fichier.

1<?php

2

3// Home page

4$app->get('/', "MicroCMS\Controller\HomeController::indexAction");

5

6// Detailed info about an article

7$app->match('/article/{id}', "MicroCMS\Controller\HomeController::articleAction");

8

9// Login form

10$app->get('/login', "MicroCMS\Controller\HomeController::loginAction")

11->bind('login');  // named route so that path('login') works in Twig templates

12

13// Admin zone

14$app->get('/admin', "MicroCMS\Controller\AdminController::indexAction");

15

16// Add a new article

17$app->match('/admin/article/add', "MicroCMS\Controller\AdminController::addArticleAction");

18

19// Edit an existing article

20$app->match('/admin/article/{id}/edit', "MicroCMS\Controller\AdminController::editArticleAction");

21

22// Remove an article

23$app->get('/admin/article/{id}/delete', "MicroCMS\Controller\AdminController::deleteArticleAction");

24

25// Edit an existing comment

26$app->match('/admin/comment/{id}/edit', "MicroCMS\Controller\AdminController::editCommentAction");

27

28// Remove a comment

29$app->get('/admin/comment/{id}/delete', "MicroCMS\Controller\AdminController::deleteCommentAction");

30

31// Add a user

32$app->match('/admin/user/add', "MicroCMS\Controller\AdminController::addUserAction");

33

34// Edit an existing user

35$app->match('/admin/user/{id}/edit', "MicroCMS\Controller\AdminController::editUserAction");

36

37// Remove a user

38$app->get('/admin/user/{id}/delete', "MicroCMS\Controller\AdminController::deleteUserAction");

39

40// API : get all articles

41$app->get('/api/articles', "MicroCMS\Controller\ApiController::getArticlesAction");

42

43// API : get an article

44$app->get('/api/article/{id}', "MicroCMS\Controller\ApiController::getArticleAction");

45

46// API : create an article

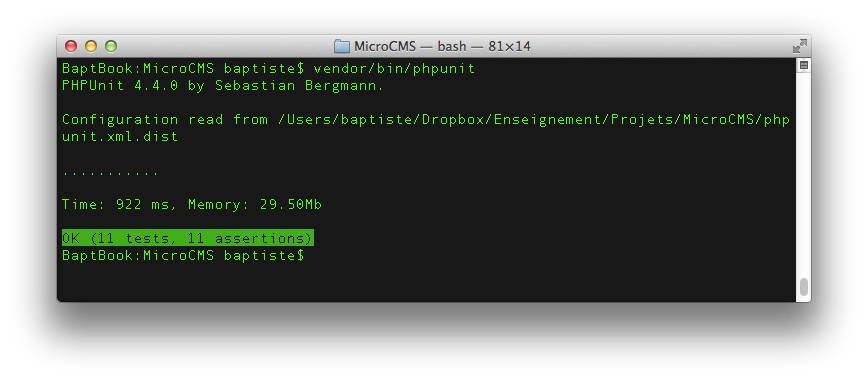
47$app->post('/api/article', "MicroCMS\Controller\ApiController::addArticleAction");

48

49// API : remove an article

50$app->delete('/api/article/{id}', "MicroCMS\Controller\ApiController::deleteArticleAction");

Notre réorganisation est terminée : lançons nos tests fonctionnels pour contrôler le fonctionnement de notre application.



Leur résultat positif prouve que la refactorisation des contrôleurs n'a introduit aucune perturbation.

Le code source associé à cette itération est disponible sur une [branche du dépôt GitHub](https://github.com/bpesquet/OC-MicroCMS-LMD/tree/iteration-13).

### Bilan

Cette itération nous a permis de réécrire nos contrôleurs sous la forme de méthodes plutôt que de fonctions anonymes, et de les répartir de façon logique dans trois classes dédiées. Lors des évolutions futures de l'application, on pourra ajouter de nouvelles méthodes dans ces classes, ou bien créer de nouvelles classes si le besoin s'en fait sentir.