Réalisez votre blog avec le framework CodeIgniter 3

Sébastien Adam

8 août 2016

Table des matières

Chapitre 1

Introduction

Bonjour,

Bienvenue sur ce tutoriel « Réalisez votre blog avec le framework CodeIgniter 3 ».

Cela fait déjà quelques années maintenant que je développe des sites web avec différents langages (Perl, PHP, C#...) et en utilisant différents outils (des gestionnaires de contenu, des frameworks...).

Lorsque j'ai découvert le framework PHP CodeIgniter, j'ai été séduit, et surpris, par sa simplicité et sa facilité d'apprentissage. Là où d'autres framework se comportent parfois comme de vraies « usines à gaz », CodeIgniter va à l'essentiel. Et j'espère, grâce à ce tutoriel, vous faire partager mon enthousiasme.

Nous allons ici passer en revue les différentes fonctionnalités d'un site web, et voir comment les réaliser avec CodeIgniter. Nous allons ainsi envisager différents aspects comme l'affichage d'une page statique, la création d'un formulaire de contact, les sessions, l'authentification et la création de contenu dynamique.

Que vous soyez un autodidacte éclairé, un jeune développeur, ou un développeur plus confirmé voulant évaluer CodeIgniter, vous êtes au bon endroit !

Entendons-nous bien, ceci n'est pas un cours de développement web. Je vais juste vous présenter un outil pour créer de sites. Ainsi, je suppose que vous possédez les pré-requis suivants :

- disposer d'un serveur web avec PHP 5.5+ ;
- disposer d'un serveur de base de données ;
- maîtriser les concepts de base de HTML, CSS et JavaScript ;
- maîtriser les concepts de base de la programmation orientée objet et plus spécifiquement avoir des bases en PHP ;
- maîtriser les concepts de base des bases de données ;
- maîtriser les concepts de base du patron d'architecture logicielle modèle-vue-contrôleur (MVC).

Bon, OK, ne vous laissez pas impressionner par la liste des pré-requis, vous ne devez pas être « BAC+5 » en réalisation de site web (de toute façon, je ne suis même pas sûr que cela existe). Vous avez seulement besoin de maîtriser les bases. Si vous êtes un peu débrouillard, vous vous en sortirez très bien. 🙂

Trêve de bavardages, passons au vif du sujet... Pour ce cours, j'ai utilisé mon serveur web de test tournant sous Debian 8.5 (stable). Les versions des logiciels sont les suivantes :

- Apache 2.4.10

- PHP 5.6.22
- MySQL 5.5.49
- CodeIgniter 3.0.6 [1]

Vous n'êtes pas obligé d'avoir votre propre serveur dédié. Il existe des tas de solutions sur internet vous permettant de faire tourner un serveur web sur votre machine comme XAMPP, et bien d'autres...

Je vous recommande de lire ce document avec attention, sinon, vous pourriez passer à côté de quelque chose d'important. Et si vous deviez rencontrer l'une ou l'autre erreur dans ce document, ou que vous vouliez proposer une amélioration, vous pouvez toujours me contacter à l'adresse sebastien.adam.webdev@gmail.com. Attention, je ne corrigerai pas vos travaux, ni ne vous donnerai de cours particulier (à moins de trouver un arrangement 😉).

1. J'ai commencé la rédaction de ce tutoriel avec la version 3.0.3. Il est donc possible que vous trouviez des références à des versions plus anciennes.

Chapitre 2

Concepts de base

2.1 Le patron MVC

Bon, en théorie, le « patron d'architecture logicielle modèle-vue-contrôleur » (plein de mots grandiloquents) doit faire partie de vos pré-requis. Mais comme c'est un concept important, je vais vite le passer en revue.

2.1.1 Qu'est-ce que le patron MVC ?

Alors, donc, le patron MVC est une manière d'organiser son code. L'idée est de séparer le code qui sert à accéder aux données de celui servant à la gestion de l'affichage et de celui destiné à gérer les requêtes des utilisateurs ainsi que les interactions entre l'affichage et les données. Pour cela, nous utiliserons respectivement des modèles, des vues et des contrôleurs.

Si vous faites des recherches sur internet à propos du patron MVC, vous allez trouver des tonnes de ressources traitant du sujet et vous risquez vite d'être noyé. Aussi, si vous vous lancez dans une discussion à ce propos, vous avez de fortes chances de vous retrouver pris entre les tirs croisés des différentes chapelles. Je vais vous présenter le point de vue qui, à mon sens, est le meilleur : le mien. 

OK, certains ne seront pas d'accord avec moi. Sachez seulement qu'il n'y a pas une chapelle qui soit meilleure que l'autre (sauf la mienne, bien entendu). Ce que je voudrais faire ici, c'est vous montrer comment CodeIgniter intègre cette philosophie de codage. Pour le reste, vous serez de toute façon seul(e)s devant vos PC, et vous ferez comme il vous plaira...

2.1.2 Modèle

Le modèle représente vos données. Il s'agit bien souvent d'une classe dont les méthodes permettent d'effectuer les actions de création, de lecture, de mise à jour et de suppression de vos données. Dans le jargon, on parle de « CRUD » (*Create*, *Read*, *Update*, *Delete*). En principe, ailleurs dans votre code, vous ne devriez jamais trouver d'instruction de connexion à une base de données, à un fichier, etc. (suivant le type de ressource que vous utilisez). Tout passe par le modèle.

Savoir ce que contient exactement le modèle est aussi un sujet qui peut enflammer les foules. Restons simple. Nous verrons au fur et à mesure comment l'implémenter avec CodeIgniter.

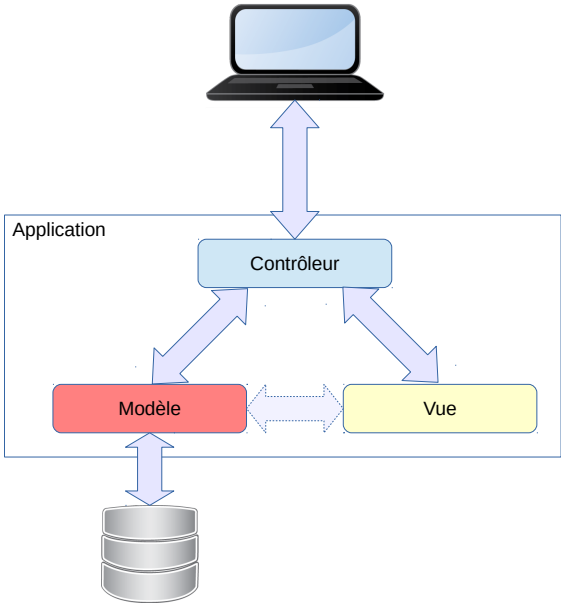

FIGURE 2.1 – Interactions entre les différents éléments du patron MVC

2.1.3 Vue

C'est dans la vue que nous allons définir ce qui doit être affiché et comment. En principe, pour une application web, c'est le seul endroit où nous pourrions retrouver du code HTML. Nous pouvons également trouver dans les vues des instructions concernant la programmation. C'est pratiquement inévitable. Cela permet de rendre les pages plus dynamiques.

2.1.4 Contrôleur

Le contrôleur va recevoir les requêtes de l'utilisateur, charger les modèles nécessaires ainsi que les vues adéquates et retourner le résultat. Avec une application web, les requêtes se font au travers des URL. La figure 2.1 à la page 4 montre les interactions entre les différents éléments.

Notez que la flèche qui unit le modèle et la vue est en pointillé. Certaines chapelles veulent que seule le contrôleur accède au modèle. D'autres acceptent que la vue communique directement avec le modèle. C'est plus facile, mais c'est moins propre. Encore une fois, c'est à vous de voir quelle est la meilleure pratique.

2.1.5 Conclusion

Voilà, j'ai très brièvement exposé le patron MVC. Vous avez maintenant en main le strict minimum nécessaire pour suivre le cours. Pour ceux qui ne maîtrisent pas encore bien le sujet, je vous invite à le creuser un peu plus, cela vous servira toujours.

Attention, ce que j'ai présenté ci-avant est vraiment le strict minimum à connaître. Si l'une ou l'autre chose n'est pas claire pour vous, revenez dessus, c'est important.

2.2 Framework

Mais qu'est-ce donc exactement un *framework* ? C'est un mot qui est difficilement traduisible en français (on pourrait parler de « cadre de travail »). Disons simplement qu'il s'agit d'une boîte à outil.

2.2.1 Qu'est-ce qu'un framework ?

Si vous avez déjà touché à la programmation, vous savez qu'on doit souvent répéter les mêmes choses dans chaque application. Pour cela, avant, nous développions nos propres outils. Et bien, aujourd'hui, ce n'est plus nécessaires, car il existe des « frameworks » pour à peu près tout.

L'idée des frameworks est de vous permettre de développer plus vite et à moindre coût. En effet, une partie des opérations répétitives sont fournies par le framework. Par exemple, l'architecture MVC est très largement utilisée actuellement dans le monde du développement web. Alors que le concept est assez « simple », si vous voulez l'implémenter vous-même, cela va vous prendre un certain temps (et même un temps certain ). L'usage du framework est donc une plus-value.

2.2.2 Présentation de CodeIgniter

Ainsi, CodeIgniter est un framework PHP qui va vous permettre de développer plus rapidement des applications (sites) web. Il fournit un ensemble varié d'outils qui vous permettra de réaliser les tâches les plus communes, tout en restant très simple. Il a en effet été développé avec pour objectif la simplicité et la rapidité.

CodeIgniter est un framework MVC. Il implémente les modèles, les vues et les contrôleurs. La gestion entre ces différents éléments est faite pour vous. Vous ne devez vous préoccuper de (presque) rien de cet aspect. Toutefois, il est très souple quant au concept de MVC. Il ne nécessite par exemple pas l'usage systématique d'un modèle comme d'autres frameworks.

Aussi, CodeIgniter est facilement extensible. Vous pouvez aisément adapter, ou remplacer, les fonctionnalités internes.

Les outils fournis par CodeIgniter sont les suivant :

Les *Helpers* Il s'agit de fichiers contenant différentes fonctions, comme en programmation procédurale. Celles-ci permettent d'ajouter des fonctionnalités à votre application qui sont accessibles partout. Par exemple, dans un helper, vous avez une fonction qui permet de générer le code HTML pour un lien.

Les librairies La différence entre les *helpers* et les librairies, c'est que ces dernières sont des classes. Nous avons par exemple une librairie permettant d'envoyer un e-mail.

Les *Drivers* Les *drivers* sont des librairies spéciales, plus élaborées et qui possèdent des enfants. Par exemple, nous avons un *driver* pour gérer les accès aux bases de données.

Vous n'avez pas besoin de connaître tous les outils dès le début. Mais lorsque vous aurez besoin d'une fonctionnalité, demandez-vous d'abord si elle n'existe pas déjà dans le framework avant de la développer. Pour cela, n'hésitez pas à explorer la documentation en ligne. Il serait bête de réinventer la roue. 

2.3 Installation

2.3.1 Installation standard

Basta avec la théorie maintenant, commençons la pratique ! Nous allons ici installer CodeIgniter pour pouvoir développer notre blog.

Rendez-vous sur le site de CodeIgniter à l'adresse `http://www.codeigniter.com/download` pour télécharger le framework. Je vous recommande la version courante (3.0.6 à l'écriture de ces

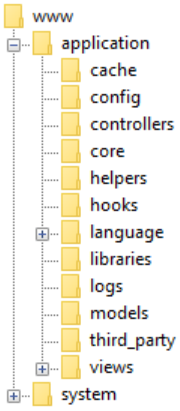

FIGURE 2.2 – Structure originale des répertoires

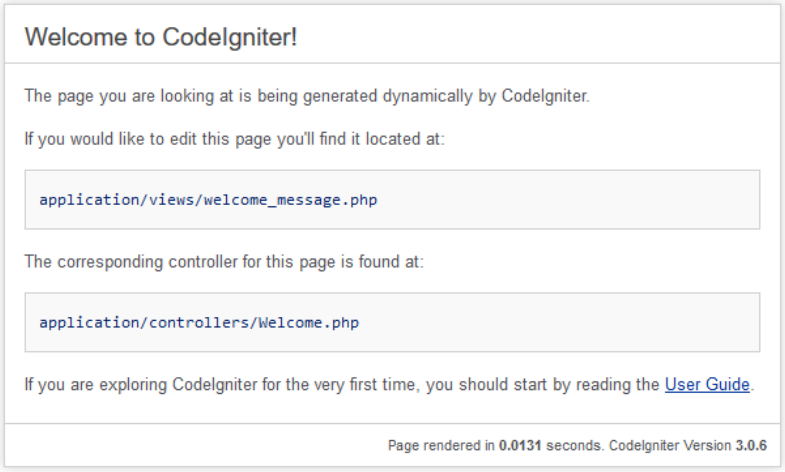

FIGURE 2.3 – Page d'accueil initiale

lignes, mais vous téléchargerez surement une version plus récente). Vous pouvez également télécharger les traductions si vous le voulez (cliquez sur « *Download System Message Translations* »).

L'installation est un processus particulièrement complexe : il faut décompressez le fichier d'archive que vous venez de télécharger dans la partie publique de votre serveur web.

La figure 2.2 à la page 6 donne la structure originale des répertoires de CodeIgniter. Voici leur utilité :

`application` C'est ici que se trouve votre application (site) web.

`application/config` **Les fichiers de configuration**.

`application/controllers` **Vos contrôleurs**.

`application/models` **Vos modèles**.

`application/views` **Vos vues**

`system` C'est le framework en lui-même. **Vous ne devez, en principe, rien modifier dans ce répertoire**.

J'ai supposé que vous aviez installé votre serveur web sur la machine où vous vous trouvez. Aisni, lancez votre navigateur et allez à l'adresse `http://localhost/`. Vous devriez avoir une page ressemblant à l'image 2.3 à la page 6. Si l'adresse de votre serveur est différentes, il suffira de changer « *localhost* » par le nom de votre serveur.

Si vous avez téléchargé les traductions, vous pouvez les également les installer. Depuis l'archive, copiez les répertoires contenant les langues qui vous intéressent se trouvant dans le répertoire `language` (par exemple `french`) vers le répertoire `application/language` de votre framework et le fichier `core/MY_Lang.php` de l'archive vers le répertoire `application/core`.

2.3.2 Configuration

C'est ici que les choses sérieuses commencent.

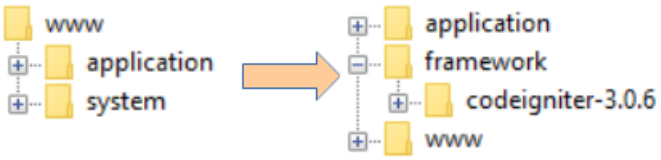

FIGURE 2.4 – Adaptation de l'arborescence

Tous les frameworks on cette chose en commun que la plupart des personnes négligent : les fichiers de configuration. Pour CodeIgniter, ils se trouvent tous dans le répertoire `application/config`. Vous constatez qu'il s'y trouve un certain nombre de fichiers. Chacun d'entre eux correspond à la configuration d'un domaine bien précis. Nous les explorerons au fur et à mesure de leur utilisation.

Pour l'instant, concentrons-nous sur `config.php`, le fichier de configuration principal. Nous devons définir la valeur du paramètre `$config['base_url']`. Comme nous l'avons installé sur notre machine locale, nous donnerons la valeur `'http://localhost/'`. Bien entendu, si vous avez installé votre site sur un autre serveur, c'est l'URL de ce serveur qui devra se trouver ici.

Un autre paramètre intéressant est `$config['index_page']`. Il donne le nom du fichier d'entrée du site. Ne le modifions pas tout de suite. Nous y reviendrons dans la partie « routage » (voir section 2.4.1 à la page 9).

Pour ceux qui ont installé les fichiers de la traduction française, vous pouvez assigner la valeur `'french'` au paramètre `$config['language']`.

Voici donc à quoi doit ressembler les paramètres dont nous venons de parler (attention, n'effacez pas les autres paramètres, sinon vous êtes dans la mouise) :

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3 // (...)
4 $config['base_url'] = 'http://localhost/';
5 $config['index_page'] = 'index.php';
6 $config['language'] = 'french';
```

Maintenant, si vous n'avez pas fait d'erreur, votre site doit toujours être opérationnel. Notez que l'installation des fichiers de traduction n'affecte pas la page d'accueil. Elle restera en anglais, c'est normal.

2.3.3 Installation personnalisée

Je ne sais pas pour vous, mais moi, mettre des éléments critiques, même cachés, dans la partie publique, cela ne me plaît pas de trop. Alors, je vais un peu réorganiser tout ça. Cette partie est optionnelle et si vous voulez, vous pouvez la passer. Aussi, cette réorganisation n'est peut-être pas compatible avec certains hébergeurs. Cela dit, elle apporte un plus à la sécurité de votre site.

La figure 2.4 à la page 7 nous montre l'avant et l'après. Pour faire simple, voici la liste des changements de répertoires. Nous supposons que la partie public se trouve dans le répertoire `www`.

`www/application` → `application`
`www/system` → `framework/codeigniter-3.0.6`

Vous vous doutez bien que maintenant, votre site web ne fonctionne plus. C'est normal. Mais ne vous inquiétez pas, avec CodeIgniter, tout est très simple. Il suffit juste de modifier deux variables dans le fichier `www/index.php` pour le remettre en route :

```php
<?php
// (...)
$system_path = '../framework/codeigniter-3.0.6';
// (...)
$application_folder = '../application';
// (...)
```

Pourquoi ai-je utiliser le répertoire `framework/codeigniter-3.0.6` et pas simplement `system` ? C'est simple, lorsque CodeIgniter produira sa prochaine version, vous pourrez la mettre à jour en créant un nouveau répertoire dans `framework` (par exemple `codeigniter-3.1.0`) et y copier le contenu du répertoire `system`. Ensuite, pour l'activer, il suffira de modifier le fichier `www/index.php`. Le temps d'inaccessibilité de votre site lors de la mise à jour sera fortement réduite. 😃

Il est temps maintenant de faire un peu de nettoyage. Vous pouvez supprimer (si, si, c'est vrai) tous les fichiers `index.html` et `.htaccess` se trouvant dans les répertoires `application` et `framework/codeigniter-3.0.6` (pas dans le répertoire `www`).

Tant qu'à parler de nettoyage, vous pouvez également supprimer les fichiers `composer.json`, `contributing.md`, `license.txt` et `readme.rst`. Pour ceux qui utilisent Git, en fait, j'espère que vous l'utilisez **TOUS** 😇, le fichier `.gitignore` peut être remonté d'un niveau et être édité comme suit :

```
.DS_Store

/application/cache/
/application/logs/

/www/user_guide/
/framework/

# IDE Files
#----------------------------
/nbproject/
.idea/*

## Sublime Text cache files
*.tmlanguage.cache
*.tmPreferences.cache
*.stTheme.cache
*.sublime-workspace
*.sublime-project
```

2.4 Première page

Cette section expliquera comment créer votre propre page d'accueil. Mais ce serait bête de faire tout ce travail, si on ne peut pas y accéder. Alors, voyons d'abord comment CodeIgniter gère l'accès aux pages.

2.4.1 Routage

D'après le principe du patron MVC, les pages sont générées à partir des méthodes des contrôleurs. Ainsi, si nous voulons faire afficher la page d'accueil de notre site, nous devons appeler la méthode correspondante du contrôleur adéquat. Et comme, avec CodeIgniter, tout est simple, c'est grâce à l'adresse que nous renseignons que nous allons pouvoir le faire (en fait, cette technique est utilisée par pratiquement tous les frameworks). Voici le format des adresses :

http://localhost/index.php/[controller-class]/[controller-method]/[arguments]

Ainsi, si pour afficher la page d'accueil, nous devons appeler la méthode `index()` du contrôler `site` (c'est d'ailleurs ce que nous ferons ), l'adresse sera celle ci-dessous. Nous n'avons que le contrôleur et la méthode, étant donné que nous n'avons aucun argument à passer.

http://localhost/index.php/site/index

Vous ne trouvez pas que cette adresse manque de charme ? Je voudrais que l'adresse de ma page d'accueil soit (je suis un nostalgique de l'HTML ) :

http://localhost/index.html

Comme toujours, avec CodeIgniter, tout est simple. Cette opération s'effectue en seulement quelques opérations...

Première étape, donner l'extension `.html` à toutes nos « pages ». Pour cela, nous devons éditer le fichier `application/config/config.php` (où ai-je bien pu entendre parler de ce fichier ?) et et assigner la valeur `.html` la variable `$config['url_suffix']` :

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3 // (...)
4 $config['url_suffix'] = '.html';
```

Seconde étape, « cacher » le `index.php` de l'adresse. Cela se fera en deux temps. Tout d'abord, dans le fichier `application/config/config.php`, assigner une chaîne vide à la variable `$config['index_page']`

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3 // (...)
4 $config['index_page'] = '';
```

Nous devons ensuite configurer notre configurer votre serveur web pour qu'il envoie de manière transparente les requêtes au fichier `index.php`. **Attention, si vous ne faites pas cela, vous ne pourrez plus accéder à vos pages...** Avec un serveur Apache, nous devons créer un fichier `.htaccess` à la racine de la partie publique contenant les quelques lignes ci-dessous. Assurez-vous que le module `rewrite_module` est activé, sinon, le serveur plante (pour de vrai !).

```
1 RewriteEngine On
2 RewriteCond %{REQUEST_FILENAME} !-f
3 RewriteCond %{REQUEST_FILENAME} !-d
4 RewriteRule ^(.*)$ index.php/$1 [L]
```

Comme dernière étape, nous allons définir quelques règles de routage. Et comme toujours, cela se fera de manière très simple, grâce au fichier `application/config/routes.php`. Dans ce dernier est créé une variable `$route`. Il s'agit d'un tableau associatif où les clés sont les URI que nous donnons, et les valeurs les appels qui seront effectivement effectués. Ce tableau est parcouru de manière séquentielle. Dès qu'une clé satisfait à l'adresse donnée, elle est utilisée. Ainsi, il est possible que des règles ne soient jamais atteintes. Si, sur votre site, pour une raison quelconque, ce ne sont pas les pages demandées qui s'affichent, c'est ici qu'il faut venir voir en premier.

Mais revenons à nos moutons. Pour faire afficher notre page d'accueil, nous allons définir un contrôleur `Site` avec une méthode `index()`. Nous allons ainsi définir les règles suivantes dans le fichier `application/config/routes.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
// (...)
$route['default_controller'] = 'site'; // Si aucun paramètre n'est donné, c'est
                                       // la route qui sera utilisée. Il n'est
                                       // pas nécessaire d'indiquer la méthode
                                       // 'index()' c'est celle qui est appelée
                                       // par défaut
$route['(:any)'] = 'site/$1'; // si un seul paramètre est donné, il sera utilisé
                              // comme méthode du contrôleur 'site'. Cela per-
                              // mettra de 'cacher' ce dernier dans les adresses
```

Croyez moi sur parole, lorsque vous aurez fini cette section et que vous taperez l'adresse `http://localhost/index.html` vous accéderez à la page d'accueil (si cela ne marche pas actuellement, c'est normal, nous n'avons pas encore fini). De toute façon, si j'avais fait une erreur à ce stade-ci, je serais revenu la corriger avant la publication de ce document. 

Ici, je ne vous ai montré que l'essentiel du sujet, ce que vous utiliserez à peu près tous les jours. Si vous voulez allez plus loin, je vous invite à consulter l'aide en ligne de CodeIgniter sur les URL et le routage.

2.4.2 Contrôleur

Pour rappel, dans la philosophie du patron MVC, nous utilisons un contrôleur pour charger les pages à afficher. Ces contrôleurs ne sont ni plus ni moins que des classes qui vont étendre `CI_Controller`. Tous les contrôleurs se trouveront dans le dossier `application/controllers`.

Maintenant, créons notre classe contrôleur, appelons-la `Site` et sauvons-la dans le fichier `application/controllers/Site.php` (respectez bien les majuscules et les minuscules). Dans cette classe, créons une méthode `index()`. Dans cette méthode, nous allons simplement faire appel à la librairie `Loader` pour charger notre vue.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  public function index() {
    $this->load->view('site/index');
  }
}
```

Vous allez me dire que vous ne connaissez pas la librairie `Loader` ? Ne vous inquiétez pas. Dites-vous simplement qu'elle est présente tout le temps et qu'elle sert à charger toutes les ressources dont nous avons besoin. Si vous voulez aller plus loin, vous pouvez consulter l'aide en ligne.

Pour charger la vue, nous passons son chemin à partir du répertoire `application/views`, sans l'extension. Ainsi, dans notre cas, notre vue sera le fichier `application/views/site/index.php`.

Attention, contrairement à la plupart des autres framework, CodeIgniter n'utilise pas les *layout*. Ainsi, lorsque vous chargez la vue `application/views/site/index.php`, vous n'aurez que le contenue de cette vue.

Vous me direz alors, mais comment faire pour avoir la même présentation pour toutes les pages ? Rassurez-vous, vous pouvez chargez plusieurs vues à la fois. Ainsi, vous pouvez définir une vue pour le haut de la page et une autre pour le bas.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  public function index() {
    $this->load->view('common/header');
    $this->load->view('site/index');
    $this->load->view('common/footer');
  }
}
```

Dans l'exemple ci-dessus, nous chargerons successivement les vues `application/views/common/header.php`, `application/views/site/index.php` et `application/views/common/footer.php`. Le résultat sera une seule page qui sera la concaténation des trois vues.

À ce stade, la page d'accueil ne peut toujours pas être affichée. C'est normal, mais on y est presque.

2.4.3 Vue

Voici (enfin) la dernière étape permettant l'affichage de notre page d'accueil. Nous allons ici créer nos trois vues. Je vous livre leur contenu tel quel, l'HTML ne faisant pas partie de ce cours.

La première vue, `application/views/common/header.php`, permettra d'afficher le haut de la page. Nous définirons l'entête HTML, le menu, etc.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title><?= $title ?></title>
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <link rel="stylesheet" href="css/bootstrap-theme.min.css">
    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media
       queries -->
    <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></
         script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script
         >
    <![endif]-->
  </head>
  <body>
    <nav class="navbar navbar-inverse">
```

```
      <div class="container-fluid">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle␣collapsed" data-toggle="
              collapse" data-target="#main_nav" aria-expanded="false">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" href="/index.html">Sébastien Adam</a>
        </div>
        <div class="collapse␣navbar-collapse" id="main_nav">
          <ul class="nav␣navbar-nav">
            <li><a href="/index.html">Accueil</a></li>
          </ul>
        </div>
      </div>
    </nav>
```

La seconde vue, `application/views/site/index.php`, permettra d'afficher le contenu de la page en elle-même.

```
<div class="jumbotron">
  <div class="container">
    <h1><?= $title ?></h1>
    <p>Ceci est ma page d'accueil</p>
  </div>
</div>
<div class="container">
  <p>Bla, bla, bla...</p>
  <p>Bla, bla, bla...</p>
  <p>Bla, bla, bla...</p>
</div>
```

La dernière vue, `application/views/common/footer.php`, affichera le bas de page et chargera les scripts.

```
    <div class="container">
      <hr>
      <footer>
        <p>&copy; Sébastien Adam 2016</p>
      </footer>
    </div>
    <script src="js/jquery-2.1.4.min.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

Je fois déjà les plus pressés d'entre vous qui ont essayé d'afficher la page et ce la ne marche toujours pas. Patience, il y a encore quelques concepts à vous expliquer...

Les plus perspicaces d'entre vous auront tout de suite vue que j'avais utilisé le framework Bootstrap pour le rendu de ma page. C'est un choix personnel, vous pouvez utiliser n'importe lequel, c'est à votre entière discrétion. L'important est d'installer tous les fichiers nécessaires dans la partie publique du site.

Vous constaterez également que dans l'en-tête et le contenu de la page, j'ai utilisé une variable appelée `$title`. Mais d'où vient-elle ? Je l'ai tout simplement définie dans le contrôleur (honte à moi, je vous ai laissé vous « planter » sans vergogne 😇).

FIGURE 2.5 – Page d'accueil initiale

Il est en effet possible de faire passer des informations depuis les contrôleurs vers les vues. C'est un des concepts-clés de la philosophie du patron MVC. Avec CodeIgniter, ce mécanisme est très simple, comme tout le reste. Il suffit de définir un tableau associatif dans le contrôleur et de le passer au `Loader`. Les clés du tableau associatif seront utilisées comme nom de variable dans la vue.

Voici ce qu'est devenue notre méthode :

```
1  <?php
2  defined('BASEPATH') OR exit('No direct script access allowed');
3
4  class Site extends CI_Controller {
5    public function index() {
6      $data["title"] = "Page d'accueil";
7
8      $this->load->view('common/header', $data);
9      $this->load->view('site/index', $data);
10     $this->load->view('common/footer', $data);
11   }
12 }
```

Dans notre exemple, la variable `$data["title"]` du contrôleur deviendra `$title` dans la vue. Vous pouvez donner le nom que vous voulez à votre tableau associatif.

Un dernier point, si vous travaillez déjà avec des frameworks PHP, vous avez certainement trouvé étrange que les vues ne contiennent « que » du code HTML et PHP. En effet, la plupart des framework utilisent leur propre système de génération dans les vues. Ce n'est pas l'option choisie par CodeIgniter. Ce dernier privilégie la rapidité et la simplicité. Toutefois, si vous tenez absolument à utiliser un langage propre à CodeIgniter, vous pouvez utiliser la librairie `Template Parser Class` (mais votre site sera plus lent).

Voilà, **maintenant**, vous pouvez accéder à votre page d'accueil. 😁 Rendez-vous à l'adresse http://localhost/index.html. Vous devriez avoir une page ressemblant à l'image 2.5 à la page

13.

2.4.4 URL Helper

Bon, notre page d'accueil fonctionne. Il a fallut le temps, mais on y est arrivé. Toutefois, nous avons fait une chose horrible : tous les liens sont en « dur ». Et c'est une chose qu'il ne faut jamais faire ! En effet, si nous déplaçons la structure de notre site, cela perturber fortement le fonctionnement du site.

Nous pourrions imaginer des tas de manières pour résoudre ce problème. Mais CodeIgniter l'a déjà fait pour nous (rappelez-vous du rôle d'un framework 🙂). Il existes le `URL Helper` qui va nous fournir toute une série de fonction permettant la gestion des URL. Je vais commencer par vous décrire deux de ces fonctions : `site_url()` et `base_url()`.

La première servira à pointer vers une « page ». Elle effectuera la concaténation de `$config['base_url']`, `$config['index_page']`, l'URI passé en paramètre et `$config['url_suffix']`.

La seconde servira à pointer vers un fichier. Elle effectuera la concaténation de `$config['base_url']` et l'URI passé en paramètre.

Sur base des paramètres que nous avons définis précédemment, le résultat de l'usage de ces fonctions sera le suivant :

`site_url("index")` → `http://localhost/index.html`
`base_url("js/jquery-2.1.4.min.js")` → `http://localhost/js/jquery-2.1.4.min.js`

Et voici un exemple d'utilisation de ces fonctions :

```
1 <script src="<?= base_url('js/jquery-2.1.4.min.js'); ?>"></script>
2 <a href="<?= site_url('index'); ?>">Accueil</a>
```

Une troisième fonction de ce *helper* qui peut s'avérer utile est `anchor()`. Elle permet de créer des liens, et Dieu sait combien de liens nous pouvons créer dans un site web (en fait, Dieu n'à rien à voir là dedans, mais c'est un autre débat 🙂). Elle prend comme premier paramètre l'URI du lien, tout comme les fonctions `site_url()` et `base_url()`, et comme second paramètre le texte qui sera entouré par les balises. Voici un exemple de l'utilisation de cette fonction :

```
1 echo anchor('index', "Accueil");
2 // affiche: <a href="http://localhost/index.html">Accueil</a>
```

Maintenant, je vous laisse mettre à jours vos vues en utilisant ces méthodes.

Hé, là ! Doucement ! Je ne vous ais pas encore dit d'essayer d'afficher votre page ! Cela ne marche pas encore ! Un *Helper*, il faut le charger avant de pouvoir l'utiliser. Nous pourrions faire un « *load* » à chaque fois qu'on en a besoin, c'est à dire tout le temps, et risquer de l'oublier l'une fois ou l'autre. Nous pouvons aussi demander à CodeIgniter de le charger automatiquement. Pour cela, nous utilisons le fichier `application/config/autoload.php` :

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3 // (...)
4 $autoload['helper'] = array('url');
```

Voilà, maintenant, l'affichage devrait marcher. Votre page d'accueil doit toujours ressembler à la figure 2.5 à la page 13. Et si vous examinez le code source de la page, vous devriez avoir vos liens en adresses absolues :

```
<link rel="stylesheet" href="http://localhost/css/bootstrap.min.css">
<a href="http://localhost/index.html">Accueil</a>
```

2.4.5 HTML Helper

Dans cette partie, nous allons explorer quelques fonctions du *HTML Helper*. Et dans cette partie, vous réserve une petite surprise... 😉

La première fonction que nous allons voir, c'est celle qui va générer le « *DOCTYPE* », c'est à dire la toute première balise de votre fichier HTML qui va indiquer à votre navigateur quelle est la version utilisée. Pour le HTML 5, cette balise est très simple. Mais pour les versions antérieures de HTML, trouver la bonne balise était parfois un vrai parcours du combattant. Cette fonction nous enlève toute cette problématique.

Ainsi, la fonction qui va nous générer notre « *DOCTYPE* » est `doctype()` . Voici une exemple pour générer celui de HTML 5 :

```
echo doctype('html5'); // affiche: <!DOCTYPE html>
```

La liste des versions valides se trouve dans le fichier `application/config/doctypes.php` . Je ne vous conseille pas d'y toucher. Toutefois, si vous utilisez des types qui ne s'y trouvent pas, vous pouvez les y ajouter.

La seconde fonction nous permettra de générer les balises de méta-données. Ce sont des balises permettant d'envoyer des informations concernant la page au navigateur et aux moteurs de recherches. En général, ces balises ont comme attribut soit `name` , soit `http-equiv` ainsi que l'attribut `content` . Nous allons utiliser la fonction `meta()` . Pour l'explication des paramètres à fournir, je vous renvoie à l'aide en ligne, c'est un peu complexe. Voici un exemple d'utilisation :

```
echo meta("X-UA-Compatible", "IE=edge", 'http-equiv');
// affiche: <meta http-equiv="X-UA-Compatible" content="IE=edge" />
echo meta("viewport", "width=device-width, initial-scale=1");
// affiche: <meta name="viewport" content="width=device-width, initial-scale=1"
   />
```

Cette fonction a toutefois un problème. HTML 5 a introduit l'attribut `charset` , et notre fonction ne permet pas de le générer. Vous me direz : « ce n'est pas grave, on n'a qu'à écrire la balise sans utiliser de fonction ! ». C'est vrai, vous avez raison, mais alors, on n'utilise pas ce *helper*. Et puis, nous, on ne se laisse pas démonter par un problème aussi minime.

Voici la petite surprise que je vous réservais : nous allons modifier le comportement de la fonction `meta()` . Nous pourrions directement modifier le fichier `html_helper.php` dans `system/helpers` . C'est vrai, mais si vous faites ça, je viens vous tire les oreilles. Et lorsque vous ferez la mise à jour de votre framework, ces modifications seront perdues.

Nous allons ainsi créer un fichier `application/helpers/My_html_helper.php` . Le nom de ce fichier doit être le même que celui du framework, mais préfixé de la valeur de la variable `$config['subclass_prefix']` (`MY_` par défaut). Ainsi, lorsque nous essayerons de charger le *helper*, c'est notre fichier qui sera chargé en premier, et c'est notre fonction adaptée qui sera utilisée et non celle du framework.

Voici le contenu de notre fichier :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

```

```php
if(!function_exists('meta')) {
  function meta($name = '', $content = '', $type = 'name', $newline = "\n") {
    if (!is_array($name)) {
      $name = array(
          array(
              'name' => $name,
              'content' => $content,
              'type' => $type,
              'newline' => $newline
          )
      );
    } elseif (isset($name['name'])) {
      $name = array($name);
    }
    $allowed_type = array('charset', 'http-equiv', 'name', 'property');
    $str = '';
    foreach ($name as $meta) {
      $meta['type'] = isset($meta['type']) ? (($meta['type'] === 'equiv') ? '
         http-equiv' : $meta['type']) : ''; // backward compatibility
      $type = in_array($meta['type'], $allowed_type) ? $meta['type'] : 'name';
      $name = isset($meta['name']) ? $meta['name'] : '';
      $content = isset($meta['content']) ? $meta['content'] : '';
      $newline = isset($meta['newline']) ? $meta['newline'] : "\n";
      $str .= '<meta␣' . $type . '="' . $name . (($type === 'charset') ? '' : '"
         ␣content="' . $content) . '"␣/>' . $newline;
    }
    return $str;
  }
}
```

J'ai fait un « copier - coller » de la fonction originale et j'y ai apporté quelques modifications. Maintenant, les attributs autorisés pour la balise `<meta>` sont `charset`, `http-equiv`, `name` et `property`. Tous ces attributs seront accompagnés de l'attribut `content`, sauf `charset` qui, lui, est seul. Si vous êtes un expert en HTML 5, vous savez que l'attribut `property` n'est pas standard. Je l'ai autorisé pour rendre la fonction compatible avec *Open Graph*

Maintenant, nous allons pouvoir générer notre balise pour l'encodage des caractères :

```php
echo meta("UTF-8", "", 'charset');    // affiche: <meta charset="UTF-8" />
```

Je ne sais pas pour vous, mais moi, je n'arrive jamais à retenir la syntaxe exacte pour l'insertion d'une feuille de style dans la page HTML. C'est pourtant une opération que l'on doit effectuer très régulièrement. Mais j'ai du mal avec cela. CodeIgniter a donc défini une fonction spécialement pour moi (enfin, elle existe et je l'aime beaucoup 🙂) : `link_tag()`. Voici un exemple de son utilisation :

```php
echo link_tag("css/style.css");
// affiche: <link href="http://localhost/css/style.css" rel="stylesheet" type="
    text/css" />
```

Outre les feuilles de style, cette fonction permet également de définir l'icône du site, les feuilles de style alternatives, etc. Pour plus de détails, voyez l'aide en ligne.

Une dernière petite fonction que je voudrais voir ici, c'est `heading()`. Elle permet de faire des titres :

```php
echo heading("Welcome", 3);    // affiche: <h3>Welcome</h3>
```

C'est une fonction toute simple. Le premier paramètre reçoit le texte du titre. Le deuxième paramètre reçoit le niveau du titre. S'il n'est pas donné, le niveau par défaut est « 1 ». La fonction

accepte un troisième paramètre qui permet de définir les attributs de la balise. Pour plus de détails, vous pouvez consulter l'aide en ligne.

Maintenant, je vous invite à modifier vos vues en utilisant les fonctions du « *HTML Helper* ». Le résultat obtenu doit être le même qu'avant. Vous vous demanderez alors quelle est l'utilité de ces fonctions ? C'est à vous de voir. Soit vous préférez taper votre code HTML vous-même, soit vous préférez utiliser les fonctions. On peut juste dire que le rendu des pages sera un poil plus rapide sans les fonctions, mais que ces dernières pourront vous aider si vous avez des doutes quant à la syntaxe de vos balises, ou à simplifier des tâches répétitives.

2.5 Page de contact

Dans cette section, nous allons créer une page de contact. Le but de cet exercice est, au travers d'un exemple, de voir comment gérer les formulaires.

2.5.1 Une nouvelle page

La première chose à faire, est de créer la « page » sur laquelle nous allons placer notre formulaire. Cela va se faire très rapidement.

Nous allons créer le fichier vide `application/views/site/contact.php`. Nous le remplirons plus tard, c'est promis. Pour une fois, je vais vous laisser afficher la page avant qu'elle ne soit terminée.

Ensuite, dans le contrôleur `application/controllers/Site.php`, nous allons créer la méthode `contact()`, comme ci-dessous. Pour l'instant, cette méthode va simplement définir le titre de la page et charger les vues. Nous la compléterons plus tard.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
    $data["title"] = "Contact";

    $this->load->view('common/header', $data);
    $this->load->view('site/contact', $data);
    $this->load->view('common/footer', $data);
  }
}
```

Maintenant, nous allons ajouter un lien dans le menu principal pour accéder à cette page de contact. Pour cela, nous allons éditer la vue `common/header.php` et rajouter une ligne dans le menu pour pointer vers la page de contact. Voici ce que nous devrions avoir après la modification (il ne s'agit que d'un extrait de la vue, tout le reste du code ne doit pas changer).

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
</div>
<!-- (...) -->
```

Maintenant, si nous allons à l'adresse `http://localhost/contact.html`, nous avons notre page avec seulement la barre de menu en haut et le copyright en bas. Vous pouvez cliquer sur « Accueil » pour aller à la page d'accueil et sur « Contact » pour revenir sur la page de contact. Notez l'utilisation de la fonction `anchor()` du *helper* `URL`.

Si vous n'avez pas le résultat attendu, revérifier les étapes précédentes.

2.5.2 Création du formulaire

Il est temps de réaliser le formulaire. Pour cela, CodeIgniter nous propose le Form Helper (qu'il faudra charger dans le contrôleur). Ce dernier va nous offrir toute une série de fonctions pour nous faciliter la vie. Encore une fois, l'utilisation de ces fonctions n'est pas obligatoire, mais ça aide.

Ci-dessous, notre contrôleur `application/controllers/Site.php` où l'on charge le Form Helper :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
    $this->load->helper("form");

    $data["title"] = "Contact";

    $this->load->view('common/header', $data);
    $this->load->view('site/contact', $data);
    $this->load->view('common/footer', $data);
  }
}
```

Ci-dessous, le contenue de notre vue `application/views/site/contact.php`. Ne vous effrayez pas, nous allons passer tous cela en revue.

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
  </div>
  <div class="row">
    <?= form_open('contact', ['class' => 'form-horizontal']); ?>
    <div class="form-group">
      <?= form_label("Votre nom :", "name", ['class' => "col-md-2 control-
          label"]) ?>
      <div class="col-md-10">
        <?= form_input(['name' => "name", 'id' => "name", 'class' => 'form-
            control']) ?>
      </div>
    </div>
    <div class="form-group">
      <?= form_label("Votre e-mail :", "email", ['class' => "col-md-2
          control-label"]) ?>
      <div class="col-md-10">
        <?= form_input(['name' => "email", 'id' => "email", 'type'=>'email', '
            class' => 'form-control']) ?>
      </div>
    </div>
    <div class="form-group">
```

```
20        <?= form_label("Titre :", "title", ['class' => "col-md-2 control-
              label"]) ?>
21        <div class="col-md-10">
22          <?= form_input(['name' => "title", 'id' => "title", 'class' => 'form-
                control']) ?>
23        </div>
24      </div>
25      <div class="form-group">
26        <?= form_label("Message :", "message", ['class' => "col-md-2 control-
              label"]) ?>
27        <div class="col-md-10">
28          <?= form_textarea(['name' => "message", 'id' => "message", 'class' => '
                form-control']) ?>
29        </div>
30      </div>
31      <div class="form-group">
32        <div class="col-md-offset-2 col-md-10">
33          <?= form_submit("send", "Envoyer", ['class'=>"btn btn-default"]); ?>
34        </div>
35      </div>
36      <?= form_close() ?>
37    </div>
38  </div>
```

Pour ce qui est du code HTML, il est assez simple et répétitif. Je me suis basé sur les exemples de la documentation de Bootstrap. Vous pouvez bien évidemment utiliser votre propre framework d'affichage (ou pas).

Il est à noter que la plupart des frameworks utilisent leurs propres fonctions. Si vous voulez les adapter à vos besoins, c'est possible, mais fastidieux. Vous allez passer des heures et des heures à trouver LE paramètre à changer pour avoir le résultat voulu (je parle d'expérience). Encore une fois, avec CodeIgniter, tout est très simple.

Mais revenons à notre vue. Je vous passe l'usage de la fonction `heading()`, cela a déjà été vu précédemment.

La première fonction sur laquelle nous allons nous pencher est `form_open()`. Celle-ci va créer la balise d'ouverture du formulaire `<form>`, mais pas uniquement. Le premier paramètre est la route donnant l'adresse où sera envoyé le formulaire. Cette route sera calculée comme avec la fonction `site_url()`. Ainsi, dans notre exemple, `contact` sera transformé en `http://localhost/contact.html`. `form_open()` possède une second paramètre optionnel permettant d'ajouter des attributs HTML sous forme d'un tableau « clé - valeur ».

Cette fonction a également un intérêt supplémentaire. Il existe un type d'attaque de sites web appelé « *Cross-Site Request Forgery* », abrégé CSRF (parfois prononcé *sea-surfing* en anglais) ou XSRF. Pour éviter cela, on ajoute un champ caché avec un jeton. Si le jeton n'est pas présent, ou est erroné, les données du formulaire ne seront pas traitées. `form_open()` permet de générer cette balise. Pour cela, il faut l'activer dans le fichier de configuration `application/config/config.php` :

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3 // (...)
4 $config['csrf_protection'] = TRUE;
5 $config['csrf_token_name'] = 'csrf_token_name';
6 $config['csrf_cookie_name'] = 'csrf_cookie_name';
7 $config['csrf_expire'] = 7200;
8 $config['csrf_regenerate'] = TRUE;
9 $config['csrf_exclude_uris'] = array();
```

```
// (...)
}
```

Évidemment, si on ouvre la balise du formulaire, il faut la fermer. Pour cela, nous utiliserons la fonction `form_close()`. Cette dernière ne prend aucun paramètre et va simplement afficher `</form>`. Vous pouvez aisément vous en passer, ou pas si vous voulez garder une certaine cohérence dans votre code.

Les autres fonctions ont des noms qui sont assez explicites. `form_label()` va générer la balise `<label>`, `form_input()` va générer la balise `<input>`, etc. Pour plus de détails n'hésitez pas à consulter l'aide en ligne.

2.5.3 Validation des données reçues

Un aspect qui est souvent négligé dans le développement en général, c'est la validation des données reçues. Et c'est comme cela qu'un site se fait facilement pirater. Entendons-nous bien : je ne vais pas vous faire un cours exhaustif sur la sécurité en ligne. Je ne suis d'ailleurs pas compétent pour cela. Mais je peux vous dire que je me souviens encore aujourd'hui de la première démonstration de la vulnérabilité d'un de mes sites (et il y a des années de cela). Alors s'il y a un point sur lequel notre attention doit se porter, c'est celui-ci.

Ainsi, CodeIgniter possède la librairie « Form Validation » qui s'occupe de, je vous le donne en mille 🙂, la validation des données reçues d'un formulaire. Il s'agit d'une librairie assez évoluée qui permet de réaliser la même chose de différentes manière. Nous allons en voir une seule. Ce n'est peut-être pas celle que vous préfèrerez, mais elle me séduit par sons côté « magique ».

Je vous invite à garder la page de documentation sur la librairie de validation ouvert en même temps que ce document. Ça peut toujours servir (d'ailleurs, je suis sûr que cela **va** vous servir 😇).

Pour valider notre formulaire, nous allons créer un fichier de configuration et associer un groupe de règles à la méthode de notre contrôleur (comme décrit ici). OK, je n'ai peut-être pas été très clair... Pour être simple, nous allons faire en sorte que les règles de validation s'appliquent automatiquement à notre formulaire de contact.

Ainsi, dans un premier temps, créons notre fichier de configuration `application/config/form_validation.php` où seront stockées toutes nos règles de validation. Ces règles détermineront que tous les champs sont requis et que, en plus, l'adresse e-mail doit être valide. Le contenu de notre fichier doit ressembler à ceci :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

$config = array(
  'site/contact' => array(
    array(
      'field' => 'name',
      'label' => 'Nom',
      'rules' => 'required'
    ),
    array(
      'field' => 'email',
      'label' => 'E-mail',
      'rules' => array('valid_email', 'required')
    ),
    array(
      'field' => 'title',
      'label' => 'Titre',
```

```
      'rules' => 'required'
    ),
    array(
      'field' => 'message',
      'label' => 'Message',
      'rules' => 'required'
    )
  )
);
```

Notez la clé `'site/contact'` de notre tableau de configuration. C'est elle qui va dire que les règles s'appliquent à la méthode « `contact` » du contrôleur « `site` ». Une règle peut définir plusieurs exigences, comme pour le champ `email`.

Maintenant, nous créons la vue qui sera affichée en cas de succès. Son contenu n'est pas spécialement important, il s'agit juste d'un message de réussite. Voici le code de la vue `application/views/site/contact_result.php` :

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
  </div>
  <div class="row alert alert-success" role="alert">
    Merci de nous avoir envoyé ce mail. Nous y répondrons dans les meilleurs dé
       lais.
  </div>
  <div class="row text-center">
    <?= anchor("index", "Fermer", ['class' => "btn btn-primary"]); ?>
  </div>
</div>
```

Pour terminer, il va nous falloir modifier notre contrôleur afin de lancer la validation et, suivant que les données d'entrée sont valides ou pas, afficher la page de succès ou à nouveau afficher la page de contact. La validation se fera en appelant la méthode `run()` de la librairie de validation des formulaires. La méthode de notre contrôleur ressemblera à ceci :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
    $this->load->helper("form");
    $this->load->library('form_validation');

    $data["title"] = "Contact";

    $this->load->view('common/header', $data);
    if ($this->form_validation->run()) {
      // TODO: envoyer le mail
      $this->load->view('site/contact_result', $data);
    } else {
      $this->load->view('site/contact', $data);
    }
    $this->load->view('common/footer', $data);
  }
}
```

Voilà, maintenant, tant que vous n'envoyez pas de données valides, vous restez sur le formulaire de contact. Sinon, vous avez une page indiquant que votre message a bien été envoyé. Et ce qui est magique, c'est qu'aucune règle de validation n'apparaît dans la méthode du contrôleur.

2.5.4 Affichage des erreurs

Si vous l'avez essayé, lorsque vous envoyez un formulaire non valide, vous revenez sur la même page, mais vous ne pouvez pas déterminer quel est le problème. Je vais donc maintenant vous montrer comment afficher les erreurs survenues. Aussi, les champs seront remplis avec les valeurs déjà introduites.

Pour réaliser ce que nous voulons faire, nous allons à nouveau faire appel au « Form Helper ». Pour l'affichage des erreurs, nous avons les fonctions `validation_errors()` (affiche toutes les erreurs en une fois) et `form_error()` (affiche l'erreur associée à un champ passé en paramètre). Pour remplir les champs avec les valeurs déjà envoyées, nous allons utiliser les fonctions `set_*()` (`set_value()`, `set_select()`...).

Attention, les fonctions précédemment citées doivent être utilisées en association avec la librairie « Form Validation ». Si ce n'est pas le cas, il ne se passera rien.

Ci-dessous je vous propose une manière d'utiliser ces fonctions. J'ai utilisé le champ `email` comme exemple. Tout d'abord, s'il y a un message d'erreur, j'ajoute la classe `has-error` au bloc contenant le champ du formulaire. Ensuite, j'ajoute la valeur du champ reçue par le serveur comme deuxième paramètre de la fonction `form_input()`. Cette dernière remplira le champ avec cette valeur. Et finalement, j'ajoute le message d'erreur, s'il existe, en dessous du champ. Notez qu'il faut à chaque fois donner le nom du champ (et si vous mélanger les noms de champ, vous allez avoir de jolies surprises 😇).

```
1     <div class="form-group">
2       <?= form_label("Votre␣e-mail :", "email", ['class' => "col-md-2␣
          control-label"]) ?>
3       <div class="col-md-10␣<?=␣empty(form_error('email'))?'':'has-error'␣?>">
4         <?= form_input(['name' => "email", 'id' => "email", 'type'=>'email', '
            class' => 'form-control'], set_value('email')) ?>
5         <span class="help-block"><?= form_error('email'); ?></span>
6       </div>
7     </div>
```

Bon, j'espère que vous avez bien compris ce qui a été fait, car, maintenant, vous allez répéter l'opération pour tous les autres champs. Vous devriez avoir un résultat ressemblant à l'image 2.6 à la page 23.

2.5.5 Envoi de l'e-mail

Bon, nous avons fait notre formulaire, envoyé nos données, et nous nous sommes assuré que les données étaient valides. Maintenant, il ne nous reste qu'à effectuer l'envoi de l'e-mail proprement dit. Cela peut être quelque chose de magique... ou pas. Généralement, si vous mettez votre site chez un hébergeur, ce dernier a configuré l'envoi de mail pour vous et la magie peut opérer. Nous allons donc envisager ce cas de figure pour commencer.

Donc, au risque de me faire passer pour un radoteur, l'envoi d'e-mail avec CodeIgniter est très simple. 🙂 Nous avons à notre disposition la librairie « Email » qui va nous fournir les outils nécessaires. Comme pour les autres librairies, il suffit de la charger, d'appeler les méthodes voulues pour créer notre émail et de l'envoyer. Voici comment nous allons faire :

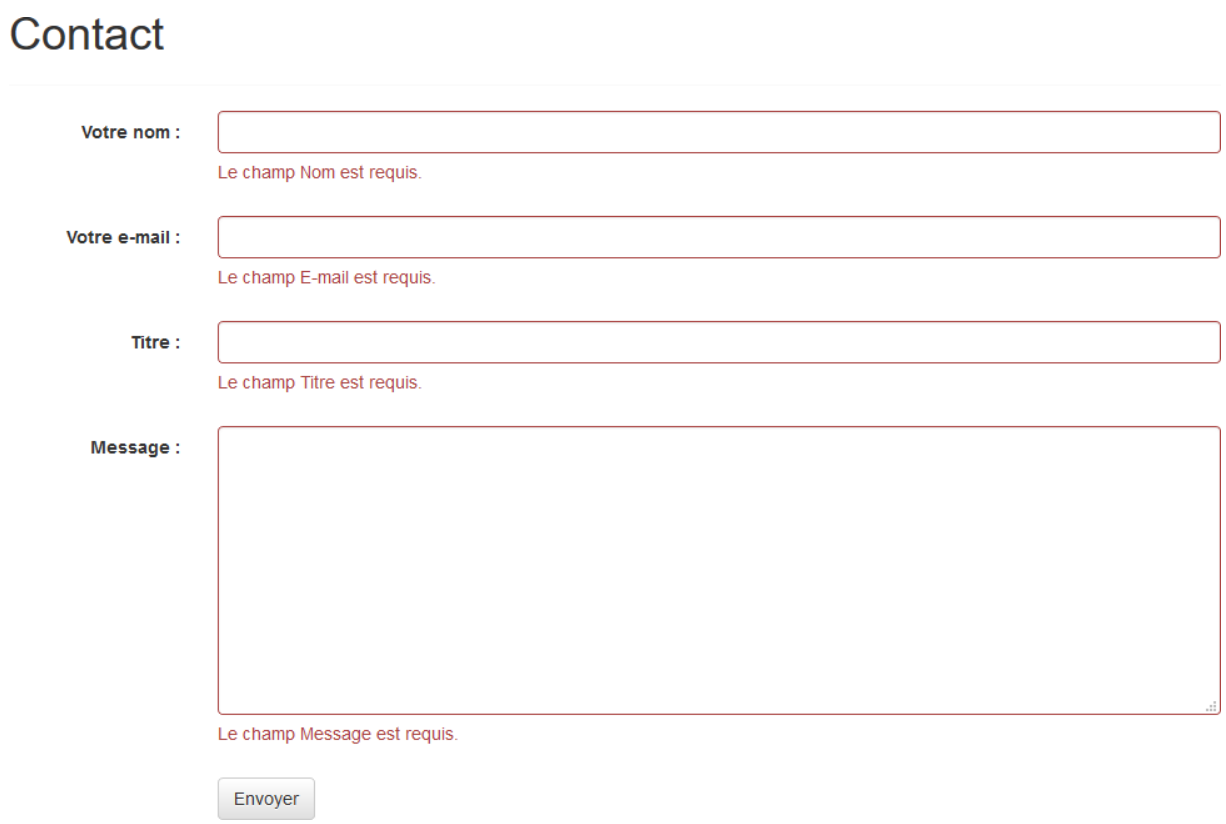

FIGURE 2.6 – Validation du formulaire de contact

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
    // (...)
    if ($this->form_validation->run()) {
      $this->load->library('email');
      $this->email->from($this->input->post('email'), $this->input->post('name')
          );
      $this->email->to('sebastien.adam.webdev@gmail.com');
      $this->email->subject($this->input->post('title'));
      $this->email->message($this->input->post('message'));
      $this->email->send();
      $this->load->view('site/contact_result', $data);
    } else {
      $this->load->view('site/contact', $data);
    }
    $this->load->view('common/footer', $data);
  }
}
```

Vous pouvez constater que nous avons les méthodes `from()` pour définir la source du message, `to()` pour définir le destinataire, `subject()` pour définir le sujet du message, et `message()` pour définir le corps de ce dernier. Une fois que tous ces paramètres auront été introduits, nous pouvons appeler la méthode `send()` pour effectivement envoyer l'e-mail. Si tout se passe correctement, le

message envoyé via le formulaire de votre site vous arrivera. Notez que j'ai ici directement donné mon adresse e-mail comme destinataire. En respectant les bonnes pratiques, vous aurez mis cette adresse dans un fichier de configuration. Ainsi si elle doit changer, vous n'aurez pas à parcourir tout votre code à la recherche des corrections à effectuer.

Il se peut que, pour un raison ou une autre, votre e-mail ne puisse pas être envoyé. Comment peut-on le savoir allez-vous me dire ? C'est, une nouvelle fois, très simple. La méthode `send()` possède une valeur de retour qui l'indique. Nous avons également à notre disposition une méthode `print_debugger()` qui va nous afficher ce qui s'est passé.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
      // (...)
      if($this->email->send()) {
        $data['result_class'] = "alert-success";
        $data['result_message'] = "Merci de nous avoir envoyé ce mail. Nous y ré
           pondrons dans les meilleurs délais.";
      } else {
        $data['result_class'] = "alert-danger";
        $data['result_message'] = "Votre message n'a pas pu être envoyé. Nous
           mettons tout en oeuvre pour résoudre le probléme.";
        // Ne faites jamais ceci dans le "vrai monde"
        $data['result_message'] .= "<pre>\n";
        $data['result_message'] .= $this->email->print_debugger();
        $data['result_message'] .= "</pre>\n";
        $this->email->clear();
      }
      $this->load->view('site/contact_result', $data);
      // (...)
  }
}
```

Dans l'exemple ci-dessus, nous avons testé la valeur de retour de la méthode `send()`. Suivant le cas, nous allons définir un message de succès ou d'échec. Notez que dans l'exemple, nous affichons les informations d'erreur. **Ne faites jamais cela sur un « vrai » site**. Il s'agit simplement ici d'illustrer l'usage de la méthode `print_debugger()`. Idéalement, ces informations devraient être fournies dans le cadre d'une gestion globale des erreurs, et cela sort du tutoriel. Notez l'usage de la méthode `clear()` en cas d'erreur. Par défaut, CodeIgniter fait une remise à zéro des paramètres de l'e-mail en cas de réussite, mais pas en cas d'échec. Il faut alors le faire manuellement, sinon, nous risquons d'envoyer des informations à des personnes qui n'en sont pas les destinataires.

Nous allons également modifier notre vue `application/views/site/contact_result.php` pour qu'elle affiche ces messages.

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
  </div>
  <div class="row alert <?= $result_class; ?>" role="alert">
    <?= $result_message; ?>
  </div>
  <div class="row text-center">
    <?= anchor("index", "Fermer", ['class' => "btn btn-default"]); ?>
  </div>
```

FIGURE 2.7 – Envoi réussi d'un e-mail

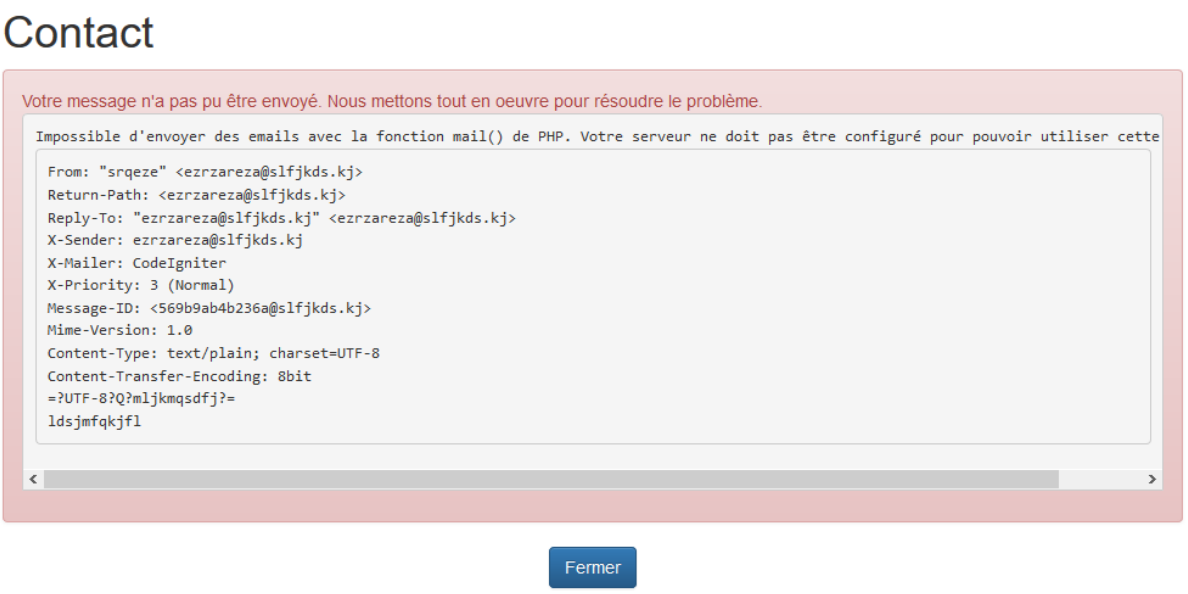

FIGURE 2.8 – Échec de l'envoi d'un e-mail

```
11 </div>
```

En cas de succès, nous devrions ainsi avoir un page qui ressemble à la figure 2.7 à la page 25 et en cas d'échec, une page qui ressemble à la figure 2.8 à la page 25.

Tout à l'heure, je vous avais parlé de magie. Et si elle n'opérait pas ? Pas de panique, CodeIgniter a prévu le coup. Il est possible de paramétrer l'envoi d'e-mail. Pour tous les paramètres disponibles, je vous invite à consulter l'aide en ligne.

CodeIgniter permet d'envoyer les e-mail via la fonction `mail()` de PHP, le protocole SMTP (comme avec un client de messagerie) ou `sendmail` (système de messagerie de linux). La méthode utilisable dépend de la configuration de votre serveur. Pour plus de détail sur ce qui est possible de faire, vous devez contacter votre fournisseur d'accès.

Je vais vous montrer comment configurer CodeIgniter pour envoyer des mail via SMTP. Pour travailler de manière propre, nous allons d'abord créer le fichier de configuration `application/config/email.php` ci-dessous. Pour connaître les paramètres à donner, vous devez contacter le fournisseur de service mail (si vous avez un client de messagerie, il s'agit des paramètres du compte).

```
1 <?php
2 defined('BASEPATH') OR exit('No direct script access allowed');
3
4 $config['protocol'] = "smtp";
5 $config['smtp_host'] = "**********";
6 $config['smtp_user'] = "**********";
7 $config['smtp_pass'] = "**********";
```

```
$config['smtp_port'] = 25;
$config['smtp_crypto'] = "tls";
```

Maintenant, dans notre contrôleur, nous allons charger ces paramètres. Notez le deuxième paramètre `TRUE` de la méthode `$this->config->load()`. Comme tous les paramètres de configuration se trouvent dans le même espace, nous pourrions écraser d'autres paramètres avec les nôtres. En mettant le second paramètre à `TRUE`, nous créons un seule entrée `email` et tous nos paramètres s'y trouverons.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function contact() {
      // (...)
      $this->load->library('email');
      $this->config->load('email', TRUE);
      $this->email->initialize($this->config->item('email'));
      // (...)
  }
}
```

2.6 Exercices

Normalement, si vous avez bien suivi ce cours, vous devriez avoir un site avec deux pages : la page d'accueil et une page de contact (tout ça pour ça ☺). Si ce n'est pas le cas, vous pouvez utiliser le code se trouvant dans le fichier « `site_base1.7z` » accompagnant ce document (également disponible sur mon site personnel).

2.6.1 Énoncés

Page de présentation

Dans ce petit exercice, vous allez créer une page de présentation « À propos ». Le menu principal (celui qui se trouve en haut de chaque page) devra avoir une entrée pointant sur cette page de présentation. Le but de cet exercice est de voir si vous avez compris le principe d'ajout d'une page.

Confirmation e-mail

Dans la page de contact, vous allez ajouter un champ qui demande à l'utilisateur de confirmer sont e-mail. Ce champ est requis et devra être égal au champ e-mail. Le but de cet exercice est de voir si vous pouvez exploiter les règles de validation des formulaires.

2.6.2 Corrections

Voici les solutions que je suggère pour les exercices proposés. Je vous invite ardemment à d'abord faire les exercices avant de lire ces solutions.

Page de présentation

Pour ajouter une page, il y a trois choses à faire : créer la méthode correspondante dans le contrôleur, créer la vue et ajouter l'entrée dans le menu principal.

Nous allons modifier notre contrôleur `application/controllers/Site.php` pour lui ajouter la méthode `apropos()` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function apropos() {
    $data["title"] = "À propose de moi...";

    $this->load->view('common/header', $data);
    $this->load->view('site/apropos', $data);
    $this->load->view('common/footer', $data);
  }
  // (...)
}
```

Créons maintenant notre vue `application/views/site/apropos.php` :

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
    <hr />
  </div>
  <div class="row">
    <p>bla, bla, bla...</p>
    <p>bla, bla, bla...</p>
    <p>bla, bla, bla...</p>
    <p>bla, bla, bla...</p>
  </div>
</div>
```

Et pour finir, modifions la vue `application/views/common/header.php` qui sert d'entête de nos pages :

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('apropos', "À propos"); ?></li>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
</div>
<!-- (...) -->
```

Confirmation e-mail

Le champ de validation de l'adresse e-mail de fait en deux étapes : ajouter le champs dans la vue et ajouter la règle de validation correspondante.

Modifions notre vues `application/views/site/contact.php` pour ajouter la demande de confirmation entre l'adresse e-mail et le titre :

```
<!-- (...) -->
<div class="form-group">
  <?= form_label("Confirmation␣e-mail :", "email", ['class' => "col-md-2␣
     control-label"]) ?>
  <div class="col-md-10␣<?=␣empty(form_error('emailconf'))␣?␣""␣:␣"has-error"␣?>
     ">
    <?= form_input(['name' => "emailconf", 'id' => "emailconf", 'type' => 'email
       ', 'class' => 'form-control'], set_value('emailconf')) ?>
    <span class="help-block"><?= form_error('emailconf'); ?></span>
  </div>
</div>
<!-- (...) -->
```

Pour notre règle de validation, nous modifierons le fichier de configuration `application/config/form_validation.php` :

```
<?php
defined('BASEPATH') OR exit('No␣direct␣script␣access␣allowed');
// (...)
$config['site/contact'][] = array(
    'field' => 'emailconf',
    'label' => 'Confirmation␣e-mail',
    'rules' => array('valid_email', 'required', 'matches[email]')
);
```

2.7 Conclusion

Voilà, vous êtes à ce stade l'heureux propriétaire d'un site web minimaliste : (au moins) une page d'accueil et un formulaire de contact. C'est vraiment le minimum pour un site, mais vous pourriez déjà le mettre en ligne. Oh, si vous avez fait l'exercice avec la confirmation de l'adresse e-mail pour la page de contact, vous pouvez l'enlever, ce n'est pas une pratique très courante sur le web.

Il est certain que pour obtenir ce résultat, nous avons utilisé un outils trop puissant, mais ne vous inquiétez pas, nous allons aborder dès le prochain chapitre des aspects beaucoup plus intéressants.

Chapitre 3

Authentification

3.1 Introduction

Dans ce chapitre, nous allons voir comment faire pour vous identifier sur votre site. Nous verrons comment adapter le contenu de vos pages suivant que vous soyez identifié ou non. Soyons clair dès le départ. Ce que je vais vous montrer ici n'est pas une sécurisation digne de « Fort Knox ». Il y a d'autres éléments à prendre en compte, comme le chiffrement...

Mais avant de rentrer dans le vif du sujet, je dois d'abord introduire quelques concepts comme les sessions et les bases de données. Bon, OK, vous êtes sensés maitriser ces domaines, mais on verra comment les gérer avec CodeIgniter. Et un petit rafraichissement ne fait pas de tord. 😃

3.2 Sessions

3.2.1 Sessions de PHP

Si vous ne connaissez pas les sessions de PHP, je vous invite à consulter l'aide en ligne à l'adresse. Pour faire court, les sessions permette à votre site de « retenir » des informations d'une page à l'autre, comme le fait d'être identifié ou pas, d'avoir mis des articles dans son panier ou pas, etc. Ces informations sont stockées sur le serveur web.

Pour démarrer une session, on utiliser l'instruction `session_start()`. PHP va alors créer un identifiant spécial qui va permettre de récupérer les données sauvegardées. Ne vous prenez pas trop la tête pour savoir comment cela fonctionne, PHP fait ça très bien sans vous 😃. Si vous voulez quand-même comprendre, ou si vous avez des problèmes, vous pouvez consulter l'aide en ligne (ce n'est pas indispensable, mais c'est chaudement recommandé 😇).

Les données à sauvegarder devront être enregistrées dans la variable spéciale `$_SESSION`. Il s'agit d'un tableau associatif, une structure de données très courante en PHP, qui est accessible depuis n'importe où dans votre programme. Attention, cette variable n'accepter pas tous les types de données. Les données scalaires (nombres, chaines de caractères...) sont très bien acceptées. Pour les autres, n'hésitez pas à faire des tests.

Voici un petit exemple d'utilisation des sessions en PHP :

```
1 <?php
2 session_start();
3 if (!isset($_SESSION['count'])) {
4   $_SESSION['count'] = 0;
5 } else {
6   $_SESSION['count']++;
```

```
}
echo "Valeur : ".$_SESSION['count'];
?>
```

La première vois que vous allez charger la page, la variable `$_SESSION['count']` sera initialisée à `0`, ensuite, chaque fois que vous chargerez la page, la valeur sera incrémentée de `1`. Cette incrémentation ne vaut que pour vous. Si quelqu'un d'autre affiche la page, ou si vous utilisez un autre navigateur, le décompte sera différent.

3.2.2 Sessions avec CodeIgniter

Pour utiliser les sessions avec CodeIgniter, nous pourrions très bien utiliser les mécanismes internes de PHP. Cependant, CodeIgniter ajoute certaines fonctionnalités aux sessions. Je ne les développerai pas toutes ici, mais pour plus de détails, vous pouvez toujours consulter l'aide en ligne (j'espère qu'il n'y a pas de royalties à payer pour utiliser cette phrase, sinon je risque d'être ruiné à ce stade 😁).

Donc, pour utiliser les sessions, nous allons devoir charger la librairie `Session`. Comme nous allons l'utiliser tout le temps, je vous conseille de la charger automatiquement. Modifions ainsi le fichier `application/config/autoload.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
// (...)
$autoload['libraries'] = array('session');
// (...)
```

Pour accéder aux données de session, nous pouvons utiliser indifféremment la variable classique `$_SESSION` ou l'objet `$this->session`. Et pour réaliser l'exemple de notre compteur de la section ci-avant, nous pourrions définir dans notre contrôleur :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function session_test() {
    $this->session->count ++;
    echo "Valeur : " . $this->session->count;
  }
}
```

Comme d'habitude, avec CodeIgniter, tout est simple. 😁

3.3 Base de données

Avec la base de données, nous allons pouvoir commencer les choses intéressantes. Comme preuve de l'importance des bases de données pour un site web, l'aide en ligne (encore une fois) de CodeIgniter sur le sujet est très conséquente par rapport au reste du *framework*. Elle y consacre même un chapitre complet. Alors, on va y aller doucement. Nous n'allons ici envisager que la lecture des données. Nous aborderons la modification des données dans un prochain chapitre.

3.3.1 Configuration de la base de données

CodeIgniter permet des configurations complexes de bases de données. Nous allons rester simple. Si vous voulez plus d'information sur le sujet, vous pouvez consulter l'aide en ligne.

La toute première chose à faire est de modifier le fichier `application/config/database.php` avec les paramètres suivants. Attention, les paramètres non cités ici ne doivent pas être supprimés.

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
// (...)
$db['default'] = array(
    // (...)
    'hostname' => 'localhost',  // nom d'hôte du serveur de base de données
    'username' => 'ocrci',      // nom d'utilisateur de la base de données
    'password' => 'ocrci',      // mot de passe de la base de données
    'database' => 'ocrci',      // nom de la base de données
    'dbdriver' => 'mysqli',     // pilote du serveur de la base de données
    // (...)
);
// (...)
```

Pour une configuration sur un serveur MySQL, les paramètres importants sont le nom du serveur de base de données, le nom d'utilisateur de la base de données, son mot de passe, le nom de la base de données et le pilote. Normalement le pilote est `'mysqli'` et le reste est donné par votre fournisseur d'accès. Si vous utilisez un autre type serveur de base de données que MySQL, il faudra vous référer à la documentation (allez ! je change un coup ), et utiliser le paramètre `'dsn'`.

Ensuite, pour pouvoir accéder à la base de données, il suffira de charger la librairie « `Database` ». Comme nous en aurons besoin très souvent, nous la chargerons automatiquement en modifiant le fichier `application/config/autoload.php`.

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
// (...)
$autoload['libraries'] = array('database', 'session');
// (...)
```

Nous accèderons à la base de données en utilisant le code `$this->db`.

3.3.2 Constructeur de requête

Nous pourrions écrire nos requêtes avec des commandes SQL et les exécuter. C'est très simple avec CodeIgniter :

```php
$query = $this->db->query(/* requête */);
```

Personnellement, je préfère utiliser le constructeur de requête (*Query Builder*). Il s'agit d'un objet qui permet de créer sa requête en faisant appel à certaines méthodes. Vu comme ça, ça pourrait paraitre plus compliqué que d'écrire soi-même ses requêtes en SQL et de les exécuter. Mais si celles-ci dépendent fortement de paramètres (comme par exemple pour effectuer des recherches complexes), le constructeur de requête s'avère très utile.

```php
$query = $this->db->select(/* colonnes */)
                  ->from(/* nom table */)
                  ->where(/* condition */)
                  ->get();
```

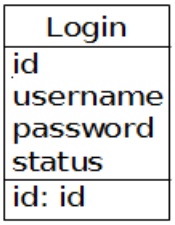

FIGURE 3.1 – Table des utilisateurs

Les méthodes du constructeur de requête se rapprochent très fort des commandes SQL. Il en propose aussi des tas d'autres. Si cela vous intéresse, n'hésitez pas à consulter l'aide en ligne (ça y est, je recommence 🙂).

3.3.3 Récupération des données

Une fois que notre requête de sélection est créée et exécutée, nous pouvons parcourir le résultat très simplement grâce à sa méthode `result()`. Supposons que nous voulions parcourir une table `client`. IL suffirait de travailler comme dans l'exemple ci-dessous.

```php
$query = $this->db->get('client');

foreach ($query->result() as $row) {
    /* faites ce que vous avez à faire ici */
}
```

3.4 Implémentation

Voyons comment utiliser les sessions et la base de données. Pour pouvoir identifier une personne, nous allons définir une table avec des noms d'utilisateurs et leur mot de passe et créer un modèle pour y accéder.

3.4.1 Table des utilisateurs

Ici, nous allons rester simple. Notre table, illustrée par ma figure 3.1 à la page 32, contiendra les noms d'utilisateur et les mots de passe des utilisateurs. Elle contiendra également un identifiant et un statut pour chaque utilisateur, mais nous ne l'utiliseront pas maintenant.

Voici le code SQL pour créer la table pour MySQL. Si vous utilisez un autre serveur de base de données, il faudra peut-être adapter un peu les requêtes. CodeIgniter possède des fonctionnalités pour créer des tables, mais je n'en parlerai pas ici, allons-y doucement.

```sql
CREATE TABLE IF NOT EXISTS `login` (
  `id` int(11) NOT NULL,
  `username` varchar(32) NOT NULL,
  `password` varchar(128) NOT NULL,
  `status` char(1) NOT NULL DEFAULT 'A'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `login` ADD PRIMARY KEY (`id`);
ALTER TABLE `login` ADD UNIQUE KEY `uk_login` (`username`);
ALTER TABLE `login` MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;
```

L'identifiant est de type entier et son incrémentation est automatique. Nous avons également veiller à ce que les noms d'utilisateur soient uniques.

Maintenant ajoutons un utilisateur. Comme nous n'avons implémenté aucune interface à ce sujet, nous devrons le faire via une requête SQL.

```
INSERT INTO `login` (`username`, `password`, `status`) VALUES ('admin', '
    $2y$10$UyXEYppMuVnYN3Vd8l/enu3UoLr9zPTOXuQGWiZ/h4GQejoCJvlH.', 'A');
```

Oups ! J'en vois qui sont un peu effrayé par le mot de passe. Ne vous inquiétez pas, ce n'est pas ce que vous allez devoir taper pour vous identifier. Un bonne pratique (pour ne pas dire une obligation) est de ne jamais sauvegarder en clair un mot de passe dans une base de données. Ainsi, si cette dernière se fait pirater, les pirates ne pourront pas faire grand-chose avec (OK, c'est encore un honteux raccourcis, disons que cela ralenti fortement le processus de piratage). Le mot de passe est ici « `password` » et nous verrons plus tard comment il a été encodé.

3.4.2 Modèle

Comme nous sommes dans une structure MVC, nous allons créer un modèle pour accéder aux données d'identification. Attention à la claque, je vous livre le code en vrac. Il est à sauvegarder dans le fichier `application/models/Auth_user.php`.

```
<?php
defined('BASEPATH') OR exit('No␣direct␣script␣access␣allowed');

class Auth_user extends CI_Model {

  protected $_username;
  protected $_id;

  public function __construct() {
    parent::__construct();
    $this->load_from_session();
  }

  public function __get($key) {
    $method_name = 'get_property_' . $key;
    if (method_exists($this, $method_name)) {
      return $this->$method_name();
    } else {
      return parent::__get($key);
    }
  }

  protected function clear_data() {
    $this->_id = NULL;
    $this->_username = NULL;
  }

  protected function clear_session() {
    $this->session->auth_user = NULL;
  }

  protected function get_property_id() {
    return $this->_id;
  }

```

```
36    protected function get_property_is_connected() {
37      return $this->_id !== NULL;
38    }
39
40    protected function get_property_username() {
41      return $this->_username;
42    }
43
44    protected function load_from_session() {
45      if ($this->session->auth_user) {
46        $this->_id = $this->session->auth_user['id'];
47        $this->_username = $this->session->auth_user['username'];
48      } else {
49        $this->clear_data();
50      }
51    }
52
53    protected function load_user($username) {
54      return $this->db
55              ->select('id, username, password')
56              ->from('login')
57              ->where('username', $username)
58              ->where('status', 'A')
59              ->get()
60              ->first_row();
61    }
62
63    public function login($username, $password) {
64      $user = $this->load_user($username);
65      if (($user !== NULL) && password_verify($password, $user->password)) {
66        $this->_id = $user->id;
67        $this->_username = $user->username;
68        $this->save_session();
69      } else {
70        $this->logout();
71      }
72    }
73
74    public function logout() {
75      $this->clear_data();
76      $this->clear_session();
77    }
78
79    protected function save_session() {
80      $this->session->auth_user = [
81          'id' => $this->_id,
82          'username' => $this->_username
83      ];
84    }
85  }
```

Vous avez tenu le choc (ou pas 🙂) ? OK, passons maintenant aux explications.

L'idée de base, est de permettre à un utilisateur de se connecter et de se déconnecter, et de garder cette information dans une session.

Tout d'abord, notre modèle va étendre la classe `CI_Model`. Cela va nous permettre d'utiliser les librairies, les paramètres de configurations... déjà chargés. Pour plus de détails sur les modèles, vous pouvez consulter l'aide en ligne.

Passons maintenant à la méthode `__construct()`. Je suppose que vous l'aurez deviné, il s'agit du constructeur. Nous allons dans un premier temps faire appel au constructeur de `CI_Model`. Ensuite, nous allons charger les informations depuis la session en appelant la méthode `load_from_session()`. Si des informations existent, elles seront chargées. Attention, si vous n'appelez pas explicitement le constructeur de la classe parente, il ne sera pas exécuté et vous vous exposez à de sérieux problèmes. Faites-moi confiance, je l'ai (involontairement 🙂) testé.

J'ai aussi défini une méthode `__get()`. Il s'agit d'une des méthodes magique de PHP, et j'aime ce qui est magique. Pour faire court, si vous essayez d'accéder à un attribut qui n'existe pas, c'est cette méthode qui sera appelée. Par exemple, si nous essayons d'accéder à l'attribut `username`, le modèle retournera le résultat de la méthode `get_property_username()` car c'est ce que nous avons défini dans la méthode `__get()`. Nous aurons ainsi une propriété en lecture seule, car il n'y a pas de méthode pour l'affecter. Vous pourriez dire qu'il suffirait d'accéder directement à la méthode `get_property_username()` pour avoir le nom d'utilisateur. Ouais ! vous avez raisons. Mais c'est moins « fun ». 😇 Ah ! N'oubliez pas d'appeler la méthode `__get()` du parent (`CI_Model`), car ce dernier utilise le même mécanisme et si vous ne le faites pas, vous aurez de sacrés surprises (je l'ai aussi testé involontairement 🙂).

Un autre usage de `__get()`, c'est la possibilité d'avoir des attributs « caclulés ». Voyons le cas de `is_connected`. Ni le modèle, ni la base de données ne possèdent de champ pouvant dire si l'utilisateur est connecté ou non (en fait, on pourrait en avoir un, mais cela ne servirait pas l'exemple...). Pour savoir si l'utilisateur est connecté ou pas, nous testerons si l'identifiant est nul et retournerons le résultat. Tout cela de manière totalement transparente.

Les sections suivantes expliqueront comme se connecter, se déconnecter et vérifier si on est connecté.

Pour que cela fonctionne, nous devons également charger automatiquement notre modèle grâce au fichier `application/config/autoload.php` :

```
1 <?php
2 // (...)
3 $autoload['model'] = array('auth_user');
4 // (...)
```

3.4.3 Authentification

Pour pouvoir s'identifier, nous devrons utiliser la méthode `login()` en lui passant le nom d'utilisateur et le mot de passe.

```
1 $this->auth_user->login($username, $password);
```

La méthode `login()` va appeler la méthode `load_user()` pour retrouver l'utilisateur dans la base de données sur base de son nom d'utilisateur. Ici, j'utilise le constructeur de requête (voir section 3.3.2 à la page 31). Si un utilisateur est trouvé, on vérifiera le mot de passe. CodeIgniter, dans ses recommandations de sécurité, invite à utiliser les fonctions propres à PHP en matière de sécurisation des mots de passes. Ainsi, pour générer le mot de passe de l'utilisateur que nous avons créé (voir section 3.4.1 à la page 32), j'ai utilisé la fonction `password_hash()`. Et pour vérifier si celui fournit par l'utilisateur est correct, j'utilise la fonction `password_verify()`. Notez que je ne sauvegarde pas le mot de passe dans la session, cela n'est d'aucune utilité et il vaut mieux éviter de propager ce genre d'information.

Mettons maintenant cela en pratique sur notre site. Il y a un peu de travail, mais ce n'est pas compliqué.

Tout d'abord, modifions la vue `application/views/common/header.php` qui sert d'entête de nos pages pour ajouter le lien vers la page d'identification :

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('apropos', "À propos"); ?></li>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li><?= anchor('connexion', "Connexion"); ?></li>
  </ul>
</div>
<!-- (...) -->
```

Vous pouvez afficher la page d'accueil pour vérifier que le lien apparait bien dans le menu. Et ça ne sert à rien de cliquer maintenant sur ce lien, il ne mène à rien...

Créons maintenant la vue `application/views/site/connexion.php`. Il s'agit d'un simple formulaire, similaire à la page de contact.

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
    <hr />
  </div>
  <div class="row">
    <?= form_open('connexion', ['class' => 'form-horizontal']); ?>
    <div class="form-group">
      <?= form_label("Nom d'utilisateur :", "username", ['class' => "col-md
          -2 control-label"]) ?>
      <div class="col-md-10 <?= empty(form_error('username')) ? "" : "has-error"
          ?>">
        <?= form_input(['name' => "username", 'id' => "username", 'class' => '
            form-control'], set_value('username')) ?>
        <span class="help-block"><?= form_error('username'); ?></span>
      </div>
    </div>
    <div class="form-group">
      <?= form_label("Mot de passe :", "password", ['class' => "col-md-2
          control-label"]) ?>
      <div class="col-md-10 <?= empty(form_error('password')) ? "" : "has-error"
          ?>">
        <?= form_password(['name' => "password", 'id' => "password", 'class' =>
            'form-control'], set_value('password')) ?>
        <span class="help-block"><?= form_error('password'); ?></span>
      </div>
    </div>
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <?= form_submit("send", "Envoyer", ['class' => "btn btn-default"]); ?>
      </div>
    </div>
    <?= form_close() ?>
  </div>
</div>
```

Passons à la validation du formulaire en modifiant le fichier `application/config/form_validation.php`. Nous allons simplement exiger que le nom d'utilisateur et le mot de passe soient fournis.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

$config = array(
    // (...)
    'site/connexion' => array(
        array(
            'field' => 'username',
            'label' => "Nom d'utilisateur",
            'rules' => 'required'
        ),
        array(
            'field' => 'password',
            'label' => 'Mot de passe',
            'rules' => 'required'
        )
    )
);
```

Pour finir, nous ajoutons la méthode `connexion()` à notre contrôleur `application/controllers/Site.php` pour afficher le formulaire d'authentification et s'identifier si les données du formulaire sont valides.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function connexion() {
    $this->load->helper("form");
    $this->load->library('form_validation');

    $data["title"] = "Identification";

    if ($this->form_validation->run()) {
      $username = $this->input->post('username');
      $password = $this->input->post('password');
      $this->auth_user->login($username, $password);
      redirect('index');
    } else {
      $this->load->view('common/header', $data);
      $this->load->view('site/connexion', $data);
      $this->load->view('common/footer', $data);
    }
  }
}
```

Si les données du formulaire sont correctes, nous allons tenter d'identifier l'utilisateur et nous rediriger vers la page d'accueil, sans vérifier le résultat de l'authentification. Nous règlerons ce problème plus tard.

3.4.4 Déconnexion

Pour pouvoir de déconnecter, nous devrons utiliser la méthode `logout()`.

```
$this->auth_user->logout();
```

Cette dernière va simplement effacer les données du modèle et de la session grâce aux méthodes `clear_data()` et `clear_session()`.

Pour mettre cela en pratique, nous allons simplement créer une entrée dans le menu principal et une méthode dans notre contrôleur.

La modification de la vue `application/views/common/header.php` sera faire comme suit :

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('apropos', "À propos"); ?></li>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li><?= anchor('connexion', "Connexion"); ?></li>
    <li><?= anchor('deconnexion', "Déconnexion"); ?></li>
  </ul>
</div>
<!-- (...) -->
```

Pour le contrôleur, nous ne feront qu'ajouter la méthode `deconnexion()` ci-dessous. Elle va simplement déconnecter l'utilisateur et renvoyer vers la page d'accueil.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  function deconnexion() {
    $this->auth_user->logout();
    redirect('index');
  }
}
```

3.4.5 Contrôle d'accès

Jusqu'à présent, notre système d'authentification fonctionne de manière bancale. Nous pouvons nous identifier et nous déconnecter, mais nous ne pouvons pas voir dans quel état nous sommes. C'est normal, nous allons régler cela grâce au contrôle d'accès.

Pour savoir si un utilisateur est authentifié, nous allons pouvoir utiliser la propriété `is_connected` de notre modèle.

Nous allons d'abord utilise ce contrôle pour la vue d'entête. En effet, nous avons aussi bien le lien de connexion et de déconnexion dans le menu. Nous allons ainsi afficher le lien de connexion si le visiteur n'est pas identifié et le lien de déconnexion si un utilisateur est identifié. Nous pouvons également en profiter pour ajouter un petit bonjour à la personne connectée. La vue `application/views/common/header.php` devient ainsi :

FIGURE 3.2 – Menu pour un visiteur non connectée

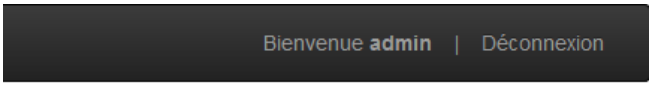

FIGURE 3.3 – Menu pour un utilisateur connecté

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('apropos', "À propos"); ?></li>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <?php if ($this->auth_user->is_connected): ?>
      <li><?= anchor('deconnexion', "Déconnexion"); ?></li>
    <?php else: ?>
      <li><?= anchor('connexion', "Connexion"); ?></li>
    <?php endif; ?>
  </ul>
  <?php if ($this->auth_user->is_connected): ?>
    <p class="navbar-text navbar-right">|</p>
    <p class="navbar-text navbar-right">Bienvenue <strong><?= $this->auth_user->
       username; ?></strong></p>
  <?php endif; ?>
</div>
<!-- (...) -->
```

Les figures 3.2 à la page 39 et 3.3 à la page 39 nous montrent les différences qui devraient apparaitre sur vos pages suivant que le visiteur s'est identifié ou non.

Maintenant que nous avons fait un peut de chirurgie esthétique, faisons un peu de médecine curative sur notre contrôleur. Avant, nous ne vérifions pas si l'identification avait réussi ou pas. Nous allons le faire maintenant. Et en cas d'échec, nous allons réafficher le formulaire de connexion avec un message d'erreur. Voici à quoi va ressembler le nouveau code de notre méthode `connexion()`.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Site extends CI_Controller {
  // (...)
  public function connexion() {
    $this->load->helper("form");
    $this->load->library('form_validation');

    $data["title"] = "Identification";

    if ($this->form_validation->run()) {
      $username = $this->input->post('username');
      $password = $this->input->post('password');
      $this->auth_user->login($username, $password);
      if ($this->auth_user->is_connected) {
        redirect('index');
      } else {
        $data['login_error'] = "Échec de l'authentification";
      }
```

```
21      }
22
23      $this->load->view('common/header', $data);
24      $this->load->view('site/connexion', $data);
25      $this->load->view('common/footer', $data);
26    }
27  }
```

Maintenant, si le formulaire est valide, nous tentons une authentification. Si elle réussi, nous serons redirigé vers la page d'accueil. Sinon, un message d'erreur est défini. Nous allons modifier la vue `application/views/site/connexion.php` pour afficher ce message d'erreur en cas d'échec d'identification.

```
1  <div class="container">
2  <!-- (...) -->
3  <div class="row">
4    <?= form_open('connexion', ['class' => 'form-horizontal']); ?>
5    <?php if (!empty($login_error)): ?>
6      <div class="form-group">
7        <div class="col-md-offset-2 col-md-10 has-error">
8          <span class="help-block"><?= $login_error; ?></span>
9        </div>
10     </div>
11   <!-- (...) -->
12   <?php endif; ?>
13   <?= form_close() ?>
14 </div>
15 <!-- (...) -->
```

3.5 Exercice

Normalement, si vous avez bien suivi ce cours, vous devriez avoir un site avec trois pages : la page d'accueil, une page « à propos » et une page de contact. Vous avez également un mécanisme qui vous permet de vous identifier. Si ce n'est pas le cas, vous pouvez utiliser le code se trouvant dans le fichier « `site_base2.7z` » accompagnant ce document (également disponible sur mon site personnel).

3.5.1 Énoncé

Voici un exercice qui va vous permettre de réviser quelques concepts et de mettre en application ce que nous venons de voir. Vous devez créer un contrôleur qui va gérer un panneau de contrôle. Ce dernier ne va pour l'instant rien contenir, et le contrôleur ne devra avoir qu'une page d'index. Par contre la page ne peut être accessible que par des personnes identifiées. L'entrée du menu vers ce panneau de contrôle ne sera également visible que pour les personnes identifiées.

3.5.2 Correction

Voici la solution que je suggère pour l'exercice proposé. Je vous invite ardemment à d'abord faire l'exercice avant de lire cette solution.

Pour réaliser cet exercice, nous allons devoir créer un nouveau contrôleur, une nouvelle vue et modifier une autre.

Nous allons ainsi créer le contrôleur `application/controllers/Panneau_de_controle.php` comme suit :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Panneau_de_controle extends CI_Controller {

  public function index() {
    if (!$this->auth_user->is_connected) {
      redirect('index');
    }
    $data["title"] = "Paneau de configuration";

    $this->load->view('common/header', $data);
    $this->load->view('dashboard/index', $data);
    $this->load->view('common/footer', $data);
  }
}
```

Le contrôleur n'a qu'une méthode : `index()`. Cette dernière va simplement rediriger les visiteurs sur la page d'accueil s'ils ne sont pas identifiés ou, sinon, afficher la page principale du panneau de contrôle.

Nous allons ensuite créer la vue `application/views/dashboard/index.php` comme ci-dessous. Elle ne fera qu'afficher le titre et un petit texte quelconque.

```
<div class="container">
  <div class="row">
    <?= heading($title); ?>
    <hr />
  </div>
  <div class="row">
    <p>Ceci est mon panneau de contrôle</p>
  </div>
</div>
```

Pour terminer, nous allons modifier la vue `application/views/common/header.php` pour ajouter l'entrée de menu pour notre panneau de contrôle.

```
<!-- (...) -->
<div class="collapse navbar-collapse" id="main_nav">
  <ul class="nav navbar-nav">
    <li><?= anchor('index', "Accueil"); ?></li>
    <li><?= anchor('apropos', "À propos"); ?></li>
    <?php if ($this->auth_user->is_connected): ?>
      <li>
        <?= anchor('panneau_de_controle/index', "Panneau de contrôle"); ?>
      </li>
    <?php endif; ?>
    <li><?= anchor('contact', "Contact"); ?></li>
  </ul>
  <!-- (...) -->
</div>
<!-- (...) -->
```

3.6 Conclusion

Voilà, vous avez un système d'authentification basic qui fonctionne très correctement. Je ne suis pas un expert de la sécurité, je ne doute pas non plus qu'une attaque de grande ampleur ne fasse tomber cette barrière, mais pour un site personnel, ce devrait être très largement suffisant.

Le système illustré ici est très simple et ne permet de différencier que les utilisateurs connectés des simples visiteurs. Toutes les personnes identifiées ont ainsi les mêmes droits. Nous pourrions utiliser les noms des utilisateurs pour les différencier et leur donner des droits différents, mais c'est une très mauvaise pratique. Il est préférable d'utiliser des groupes et de donner les permissions sur ces derniers. On appelle cela le contrôle d'accès basé sur les rôles ou « RBAC » (acronyme de « *Role-based access control* »). Je vous laisse vous renseigner sur le sujet par vous-même, les références sur internet sont nombreuses.

Chapitre 4

Blog

4.1 Introduction

Maintenant que nous avons mis en place un système d'authentification, nous allons pouvoir commencer les choses sérieuses. Nous allons mettre en place un blog. Bon, ne vous imaginez pas que vous allez créer le prochain gestionnaire de contenu à la mode (hé, je l'aurai fait avant vous si c'était si simple ! 😄), le but de ce chapitre est juste de vous montrer au travers d'un exemple concret comment interagir avec une base de données.

Au niveau des fonctionnalités, nous allons développer un blog basic, mais fonctionnel. Nous allons créer, publier, modifier, supprimer des articles. Avec cela, nous allons couvrir à peu près toutes les opérations élémentaires concernant les bases de données. Libre à vous, ensuite, de creuser le sujet et d'aller plus loin.

4.2 Base de données

La première chose à faire, c'est de définir nos tables dans la base de données. La figure 4.1 à la page 43 illustre celles que nous allons mettre en place. Il s'agit d'une construction minimaliste, mais largement suffisant pour notre tutoriel.

La table `Login` a déjà été créée au chapitre précédent (voir section 3.4.1 à la page 32).

La table `Article` va contenir nos articles. Chacun de ceux-ci auront un titre, un contenu, une date et un auteur. Nous avons également un champ « alias » qui va servir pour la création de l'URL et un champ « status » qui va permettre de dire si l'article est publié ou non. Cela vous permettra de rédiger des brouillons et de les publier seulement lorsqu'ils seront prêts. Nous avons également

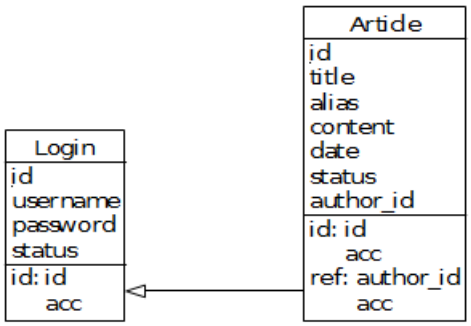

FIGURE 4.1 – Table des utilisateurs et des articles

un champ « id » pour identifier les articles de manière unique dans la table.

Voici le code SQL pour créer la table pour MySQL. Si vous utilisez un autre serveur de base de données, il faudra peut-être adapter un peu les requêtes. Attention, souvenez-vous que vous avez déjà créé la table `Login`. Si ce n'est pas le cas, reportez-vous d'abord à la section 3.4.1 à la page 32.

```
CREATE TABLE IF NOT EXISTS `article` (
  `id` int(11) NOT NULL,
  `title` varchar(64) NOT NULL,
  `alias` varchar(64) NOT NULL,
  `content` text NOT NULL,
  `date` datetime NOT NULL,
  `status` char(1) NOT NULL DEFAULT 'W',
  `author_id` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `article` ADD PRIMARY KEY (`id`);
ALTER TABLE `article` ADD KEY `idx_author_id` (`author_id`);

ALTER TABLE `article` MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;

ALTER TABLE `article` ADD CONSTRAINT `fk_article_login_id` FOREIGN KEY (`
    author_id`) REFERENCES `login` (`id`);
```

4.3 Liste des articles

La première chose que nous allons faire, c'est afficher la liste des articles déjà postés. OK, vous allez dire qu'il n'y a encore rien à afficher, mais il faut bien commencer par quelque chose. Et puis, un aspect qui est souvent négligé, c'est la gestion des données absentes, et nous allons nous y atteler ici.

Pour afficher la liste des articles, nous allons devoir créer un modèle, un contrôleur (qui servira à gérer tout le blog) et des vues.

4.3.1 Modèle

Généralement, on crée un modèle par entité. Personnellement, je travaille un peu différemment. J'aime bien avoir en plus un modèle par liste. Ainsi, nous allons d'abord créer notre modèle pour la liste des articles, et nous créerons un peu plus tard le modèle pour l'article en lui-même. Si vous voulez polémiquer sur la pertinence de cette manière de faire, on risque d'y passer du temps. Alors, si vous n'êtes pas du même avis, faites comme bon vous semble. 😃

Pour définir notre modèle, nous allons dans un premier temps définir ce que nous allons afficher. C'est simple, il s'agit du contenu de la table `Article`. Toutefois, nous devrons remplacer l'identifiant de l'auteur par son nom d'utilisateur et le code du statut pas un libellé compréhensible. De plus, nous n'afficherons qu'une partie du contenu de l'article. Cela n'aurait pas beaucoup de sens d'afficher l'article en entier lorsqu'on liste tous les articles.

Pour limiter le contenu de l'article, nous allons utiliser une fonction SQL, nous n'avons pas trop le choix (en fait, si, mais cela va beaucoup compliquer les choses, croyez moi sur parole). Si nous voulons limiter le contenu sur base d'un certain nombre de caractères, nous pouvons utiliser la fonction `SUBSTRING()`, mais il y a de fortes chances pour que le dernier mot soit coupé, et ce ne sera pas très joli. Nous pourrions également utiliser la fonction `SUBSTRING_INDEX()` en donnant

un espace comme délimiteur, mais alors, là, nous aurons une incertitude quand à la taille du texte affiché (suivant la taille des mots affichés). Choisissons la seconde solution, c'est la meilleure (bon, c'est vrai, ça se discute aussi, disons qu'il fallait en choisir une et il fallait justifier ce choix, alors...). Attention, les fonctions citées ici concernent MySQL. Si vous utiliser un autre serveur de base de données, veuillez vous référer à sa documentation.

Pour avoir le nom de l'auteur de l'article plutôt que son identifiant, il va falloir faire une jointure entre les tables `Login` et `Article`. Le constructeur de requête de CodeIgniter fait ça très bien, mais au niveau performance, il vaut mieux utiliser une vue. En effet, cette requête sera très souvent effectuée, même pour afficher les détails d'un article.

Nous allons donc créer une vue comme suit :

```
CREATE VIEW `article_username`
AS
SELECT `article`.`id`,
       `title`,
       `alias`,
       `content`,
       `date`,
       `article`.`status`,
       `username` AS `author`,
       `author_id`
FROM `article` INNER JOIN `login` ON `article`.`author_id` = `login`.`id`;
```

Et voici donc notre modèle `application/models/Articles.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Articles extends CI_Model {

  protected $_list;

  public function __construct() {
    parent::__construct();
    $this->_list = [];
  }

  public function __get($key) {
    $method_name = 'get_property_' . $key;
    if (method_exists($this, $method_name)) {
      return $this->$method_name();
    } else {
      return parent::__get($key);
    }
  }

  protected function get_property_has_items() {
    return count($this->_list) > 0;
  }

  protected function get_property_items() {
    return $this->_list;
  }

  protected function get_property_num_items() {
    return count($this->_list);
  }
```

```
33
34  public function load($show_hidden = FALSE) {
35     $this->db->select("id, title, alias, SUBSTRING_INDEX(content, ' ', 20) AS
           content, date, status, author")
36         ->from('article_username')
37         ->order_by('date', 'DESC');
38     if (!$show_hidden) {
39       $this->db->where('status', 'P');
40     }
41     $this->_list = $this->db->get()
42         ->result();
43   }
44 }
```

Vous devez commencer à me connaître, maintenant. Je vous livre d'abord le paquet et j'explique ensuite. 😃

J'ai utilisé la même méthode `__get()` que pour le modèle servant à l'authentification (il faudra que je pense à créer une sur-classe avec les méthodes courantes). Si vous avez raté les explications la concernant, je vous invite pour une séance de rattrapage à la section 3.4.2 à la page 33.

J'ai défini trois propriétés en lecture seule : `items` [1] qui retourne la liste des articles et `num_items` qui donne le nombre d'articles et `has_items` qui indique s'il y a des articles ou pas. J'ai également défini une méthode `load()` qui va charger les articles.

Vous constaterez que la méthode `load()` prend un paramètre `$show_hidden` qui est `FALSE` par défaut. Nous laissons ici la possibilité de n'afficher que les articles publiés ou également les brouillons et les articles supprimés. Nous pouvons mesurer ici l'intérêt du constructeur de requête.

La séquence « normale » d'utilisation de ce modèle est d'abord de le charger, puis d'appeler la méthode `load()`. Ensuite, les articles seront accessibles via la propriété `items`. Les autres éléments seront utiles dans les vues (croyez-moi, vous vous en rendrez vite compte 😁). Si vous comptez publier votre site, il faudra faire évoluer ce modèle, notamment pour ajouter des fonctions de pagination.

4.3.2 Vue

J'avais d'abord envisagé de faire le contrôleur avant la vue, mais je suis sûr que certain(e)s d'entre vous auraient été tenté(e)s d'afficher la page avant d'avoir fini. Alors, faisons la vue `application/views/blog/index.php` en premier :

```
1  <div class="container">
2    <div class="row">
3      <div class="col-md-12">
4        <?= heading($title); ?>
5        <hr />
6      </div>
7    </div>
8    <div class="row">
9      <div class="col-md-12">
10       <p class="text-right">Nombre d'articles: <?= $this->articles->num_items;
             ?></p>
11     </div>
12   </div>
13   <div class="row">
```

1. Idéalement, au lieu de la propriété `items`, nous utiliserions un itérateur, mais je ne voulais pas vous perdre ici.

```
14      <?php if ($this->articles->has_items) : ?>
15        <?php
16        foreach ($this->articles->items as $article) {
17          $this->load->view('blog/article_resume', $article);
18        }
19        ?>
20      <?php else: ?>
21        <div class="col-md-12">
22          <p class="alert␣alert-warning" role="alert">
23            Il n'y a encore aucun article.
24          </p>
25        </div>
26      <?php endif; ?>
27    </div>
28  </div>
```

Nous allons afficher le titre de la page et ensuite le nombre d'article grâce à la propriété `num_items` de notre modèle (je vous avais dit que ça servirait 😇). Ensuite, nous faisons un test pour savoir s'il y a des articles ou pas grâce à la propriété `has_items`. S'il n'y a pas d'article, un message d'avertissement sera affiché.

S'il y a des articles, nous exécuterons une boucle pour les afficher. Alors ici, j'ai inclus un petit concept intéressant à placer dans un entretien d'embauche : la **vue partielle**. Certaines personnes en font des pataquès autour de ce concept, mais en réalité, c'est très simple : il s'agit d'une vue appelée à partir d'une autre vue. Une même vue partielle peut être appelée à partir de différentes autres vues pour avoir un bout de code identique sur plusieurs pages. Cela évite la redondance. Alors si vous obtenez un job grâce à ça, n'oubliez pas ma commission. 😉

Voici la vue partielle `application/views/blog/article_resume.php` :

```
1   <div class="col-md-4">
2     <?= heading(htmlentities($title), 2); ?>
3     <p>
4       <small>
5         <?= nice_date($date, 'd/m/Y'); ?>
6         -
7         <?= $author ?>
8         <?php if ($this->auth_user->is_connected) : ?>
9           -
10          <?= $this->article_status->label[$status]; ?>
11        <?php endif; ?>
12      </small>
13    </p>
14    <p><?= nl2br(htmlentities($content)); ?>...</p>
15  </div>
```

Comme nous avons passé un article en paramètre de notre vue, nous accédons directement à ses attributs. Aussi, vous aurez remarqué l'utilisation de la fonction PHP `htmlentities()`. Si un utilisateur tape du code HTML, il sera affiché et non interprété. Une règle de base : **n'afficher jamais tel quel du texte introduit par une utilisateur**. Vous pourriez avoir de drôles de surprises...

Je vois que certains d'entre vous se posent des questions à propos de certains éléments de cette vue. C'est normal, elle contient des choses que vous n'avez pas encore vues. Pour ceux qui ne se posent pas de question, relisez bien le code, vous avez loupé des trucs. 😉

Tout d'abord, il y a la fonction `nice_date()`. Elle est fournie par le *helper* `Date` et permet de formater facilement une date.

Ensuite, il y a `article_status`. Kesako ? Souvenez-vous, nous avons défini un code pour le statut d'un article. Pour le rendre plus parlant, j'ai défini le modèle `application/models/Article_status.php`. Ce dernier remplacera le code avec un texte compréhensible par les humains.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Article_status {

  protected $_status;

  public function __construct() {
    $this->_status = [
        'W' => [
            'text' => 'Brouillon',
            'decoration' => 'warning'
        ],
        'P' => [
            'text' => 'Publié',
            'decoration' => 'primary'
        ],
        'D' => [
            'text' => 'Supprimé',
            'decoration' => 'danger'
        ]
    ];
  }

  public function __get($key) {
    $method_name = 'get_property_' . $key;
    if (method_exists($this, $method_name)) {
      return $this->$method_name();
    } else {
      return parent::__get($key);
    }
  }

  protected function get_property_label() {
    $result = [];
    foreach ($this->_status as $key => $value) {
      $result[$key] = '<span class="label label-' . $value['decoration'] . '">'
          . $value['text'] . '</span>';
    }
    return $result;
  }

  protected function get_property_text() {
    $result = [];
    foreach ($this->_status as $key => $value) {
      $result[$key] = $value['text'];
    }
    return $result;
  }

  protected function get_property_codes() {
    return array_keys($this->_status);
  }
```

```
53 }
```

Notre modèle contiendra une propriété `label` qui retournera un tableau des textes des statuts avec un formatage HTML, une propriété `text` qui retournera un tableau des textes des statuts et une propriété `codes` qui retourne la liste des codes possibles.

Vous constaterez que les données sont « *hard-codées*[2] » dans le constructeur. Sachez que c'est une pratique à proscrire. Idéalement, vous créerez une table dans votre base de données pour contenir la liste des statuts. J'avoue qu'ici, j'ai fait preuve d'un peu de paresse. Alors, « faites ce que je dis, mais pas ce que je fais ». 😇

4.3.3 Contrôleur

Voilà, maintenant tout devrait être prêt pour pouvoir afficher la liste des articles de notre blog. Il n'y a plus que le contrôleur `application/controllers/Blog.php` à ajouter :

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {

  function index() {
    $this->load->helper('date');
    $this->load->model('articles');
    $this->load->model('article_status');
    $this->articles->load($this->auth_user->is_connected);

    $data['title'] = "Blog";

    $this->load->view('common/header', $data);
    $this->load->view('blog/index', $data);
    $this->load->view('common/footer', $data);
  }
}
```

Dans la page d'index, nous allons charger le *helper* `Date` (pour la fonction `nice_date()`), et les modèles `Article` et `Article_status`. Nous chargeons ensuite les données de la base de données et affichons les vues. Suivant que l'utilisateur soit connecté ou pas, nous afficherons les articles cachés ou non. En allant sur l'adresse `http://localhost/blog/index.html`, vous devriez avoir une page ressemblant à la figure 4.2 à la page 50.

Ah ! oui, si vous avez bien regardé l'illustration 4.2 à la page 50, vous aurez constaté qu'il y avait une entrée dans le menu principal pour le blog. Vous devriez être capable de le faire vous-même maintenant 🙂. Vous avez du mal et avez besoin d'un indice ? Revoyez les exercices de la section 2.6.1 à la page 26.

4.3.4 Routage

Pour consulter la page d'accueil du blog, ile faut aller à l'adresse `http://localhost/blog/index.html`. Si nous utilisons l'adresse `http://localhost/blog/`, nous aurons une erreur. En effet, si nous nous reportons à la section 2.4.1 à la page 9, nous avons défini que toutes les URI ne comportant qu'un seul paramètre seront redirigées vers le contrôleur `Site`. Ainsi, `blog` sera redirigé vers `site/blog`.

2. codées « en dur », c'est à dire que les données se retrouvent dans le code

FIGURE 4.2 – Page d'accueil du blog sans articles

J'aimerais que l'adresse http://localhost/blog/ nous donne la liste des articles de notre blog. Comme toujours, avec CodeIgniter, c'est très simple. Nous allons juste modifier le fichier `application/config/routes.php` :

```
1  <?php
2  defined('BASEPATH') OR exit('No direct script access allowed');
3  // (...)
4  $route['default_controller'] = 'site';
5  $route['blog'] = 'blog/index';  // l'URI 'blog' sera redirigée vers 'blog/index'
6  $route['(:any)'] = 'site/$1';
```

Attention ! `$route['blog']` doit se trouver **avant** `$route['(:any)']`, sinon c'est cette dernière qui sera prise en compte et nous aurons toujours une erreur si nous voulons accéder à l'adresse http://localhost/blog/.

4.4 Création d'un article

Nous pouvons maintenant afficher la liste (vide) des articles. Tâchons de la remplir un peu...

4.4.1 Modèle

Nous allons créer le modèle `application/models/Article.php` qui va représenter un article comme suit :

```
1  <?php
2  defined('BASEPATH') OR exit('No direct script access allowed');
3
4  class Article extends CI_Model {
5
6    protected $_alias;
7    protected $_author;
8    protected $_author_id;
9    protected $_content;
10   protected $_date;
11   protected $_id;
```

```
12    protected $_status;
13    protected $_title;
14
15    public function __construct() {
16      parent::__construct();
17      $this->clear_data();
18    }
19
20    public function __get($key) {
21      $method_name = 'get_property_' . $key;
22      if (method_exists($this, $method_name)) {
23        return $this->$method_name();
24      } else {
25        return parent::__get($key);
26      }
27    }
28
29    public function __set($key, $value) {
30      $method_name = 'set_property_' . $key;
31      if (method_exists($this, $method_name)) {
32        $this->$method_name($value);
33      } else {
34        parent::__set($key, $value);
35      }
36    }
37
38    protected function clear_data() {
39      $this->_alias = NULL;
40      $this->_author = NULL;
41      $this->_author_id = NULL;
42      $this->_content = NULL;
43      $this->_date = NULL;
44      $this->_id = NULL;
45      $this->_status = NULL;
46      $this->_title = NULL;
47    }
48
49    protected function get_property_alias() {
50      return $this->_alias;
51    }
52
53    protected function get_property_author() {
54      return $this->_author;
55    }
56
57    protected function get_property_author_id() {
58      return $this->_author_id;
59    }
60
61    protected function get_property_content() {
62      return $this->_content;
63    }
64
65    protected function get_property_date() {
66      return $this->_date;
67    }
68
69    protected function get_property_id() {
```

```
70      return $this->_id;
71    }
72
73    protected function get_property_is_found() {
74      return $this->_id !== NULL;
75    }
76
77    protected function get_property_status() {
78      return $this->_status;
79    }
80
81    protected function get_property_title() {
82      return $this->_title;
83    }
84
85    public function load($id, $show_hidden = FALSE) {
86      $this->clear_data();
87      $this->db
88          ->from('article_username')
89          ->where('id', $id);
90      if (!$show_hidden) {
91        $this->db->where('status', 'P');
92      }
93      $data = $this->db
94          ->get()
95          ->first_row();
96      if ($data !== NULL) {
97        $this->_alias = $data->alias;
98        $this->_author = $data->author;
99        $this->_author_id = $data->author_id;
100       $this->_content = $data->content;
101       $this->_date = $data->date;
102       $this->_id = $data->id;
103       $this->_status = $data->status;
104       $this->_title = $data->title;
105     }
106   }
107
108   public function save() {
109     $data = [
110         'alias' => $this->_alias,
111         'author_id' => $this->_author_id,
112         'content' => $this->_content,
113         'status' => $this->_status,
114         'title' => $this->_title
115     ];
116     if ($this->is_found) {
117       $this->db->where('id', $this->_id)
118           ->update('article', $data);
119     } else {
120       $data['date'] = date('Y-m-d␣H:i:s');
121       $this->db->insert('article', $data);
122       $id = $this->db->insert_id();
123       $this->load($id, TRUE);
124     }
125   }
126
127   protected function set_property_author_id($author_id) {
```

```
128     $this->_author_id = $author_id;
129   }
130
131   protected function set_property_content($content) {
132     $this->_content = $content;
133   }
134
135   protected function set_property_status($status) {
136     $this->_status = $status;
137   }
138
139   protected function set_property_title($title) {
140     $alias = url_title($title, 'underscore', TRUE);
141     $this->_title = $title;
142     $this->_alias = $alias;
143   }
144 }
```

Je reconnais que le code est un peu long, mais il n'a rien de compliqué.

Comme pour les autres modèles, nous avons la méthode `__get()` et les méthodes `get_property_*()` pour accéder aux propriétés de l'objet, comme nous l'avons déjà fait auparavant.

J'ai également utilisé la méthode `__set()` pour pouvoir affecter certaines propriétés. Le principe de fonctionnement est ici identique qu'avec `__get()`. Remarquez la méthode `set_property_title($title)`. Elle va non seulement affecté le contenu de l'attribut `$_title`, mais également celui de `$_alias` qui est la transposition du titre en URL. La méthode `url_title()` est fournie par le *helper* `URL`.

La méthode `load()` permet de charger les données d'un article sur base de son identifiant. Notez qu'il y a un paramètre permettant d'autoriser ou pas le chargement d'un article non publié, comme avec la liste des articles. Les données récupérées seront ensuite affectées aux attributs du modèle. Les données seront récupérées dans la vue `article_username` définie à la section 4.3.1 à la page 44.

La méthode `save()` va sauvegarder les données du modèle dans la base de données. Cette méthode pourra aussi bien être utilisée pour faire une mise à jour qu'une nouvelle insertion. Attention, en cas de création d'un nouvel article, nous insérons la date depuis le code PHP. Ce n'est pas du tout une bonne pratique. Normalement, on met le temps courant comme valeur par défaut pour le champ `date` de notre table dans la base de données. Cette possibilité n'est disponible pour les serveurs MySQL qu'à partir de la version 5.6 pour les types `datetime` et j'utilise un version 5.5[3]. Il y a d'autres moyens pour contourner le problème, mais c'est un peu compliqué.

Vous constaterez que je recharge les données après une insertion. Il s'agit d'une manière d'assigner les données manquantes et m'assurer que le modèle est correctement sauvegardé. C'est un peu gourmand en ressource (ne le faites pas sur un système à forte charge), mais terriblement efficace.

4.4.2 Vue

Comme vue pour la création d'un article, nous aurons un formulaire classique comparable aux formulaires de contact et d'authentification. Voici donc la vue `application/views/blog/form.php` :

3. OK, c'est une veille version, mais debian est debian

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <?= heading($title); ?>
      <hr />
    </div>
  </div>
  <div class="row">
    <div class="col-md-12">
      <?= form_open('blog/nouvel_article', ['class' => 'form-horizontal']); ?>
      <div class="form-group">
        <?= form_label("Titre :", "title", ['class' => "col-md-2 control-
            label"]) ?>
        <div class="col-md-10 <?= empty(form_error('title')) ? "" : "has-error" 
            ?>">
          <?= form_input(['name' => "title", 'id' => "title", 'class' => 'form-
              control'], set_value('title')) ?>
          <span class="help-block"><?= form_error('title'); ?></span>
        </div>
      </div>
      <div class="form-group">
        <?= form_label("Texte :", "content", ['class' => "col-md-2 control-
            label"]) ?>
        <div class="col-md-10 <?= empty(form_error('content')) ? "" : "has-error
            " ?>">
          <?= form_textarea(['name' => "content", 'id' => "content", 'class' =>
              'form-control'], set_value('content')) ?>
          <span class="help-block"><?= form_error('content'); ?></span>
        </div>
      </div>
      <div class="form-group">
        <?= form_label("Statut :", "status", ['class' => "col-md-2 control-
            label"]) ?>
        <div class="col-md-10 <?= empty(form_error('status')) ? "" : "has-error" 
            ?>">
          <?= form_dropdown("status", $this->article_status->text, set_value('
              status'), ['id' => "content", 'class' => 'form-control']) ?>
          <span class="help-block"><?= form_error('status'); ?></span>
        </div>
      </div>
      <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
          <?= form_submit("send", "Envoyer", ['class' => "btn btn-default"]); ?>
        </div>
      </div>
      <?= form_close() ?>
    </div>
  </div>
</div>
```

Notez que pour le statut, nous utilisons un liste de choix. Pour la remplir, nous nous servons du modèle `Article_status` défini à la section 4.3.2 à la page 46.

Pour accéder à ce formulaire, nous devons ajouter un lien quelque part. Je vous propose de le faire sur la page affichant la liste des articles, à côté du nombre d'articles. Modifions donc la vue `application/views/blog/index.php` :

```
<!-- (...) -->
```

```
  <div class="row">
    <div class="col-md-10">
      <ul class="nav nav-pills nav-justified">
        <?php if ($this->auth_user->is_connected) : ?>
          <li role="presentation">
            <?= anchor('blog/nouvel_article', "Nouvel article"); ?>
          </li>
        <?php endif; ?>
      </ul>
    </div>
    <div class="col-md-2">
      <p class="text-right">
        Nombre d'articles: <?= $this->articles->num_items; ?>
      </p>
    </div>
  </div>
<!-- (...) -->
```

Le bouton de création d'article ne sera visible que pour les personnes connectées. C'est normal, nous avions défini qu'il fallait être connecté pour pouvoir créer un article. Et aussi, ce n'est pas encore la peine de cliquer dessus, il faut que le contrôleur soit modifié pour afficher le formulaire.

4.4.3 Contrôleur

Nous allons modifier le contrôleur `application/controllers/Blog.php` créé à la section 4.3.3 à la page 49 en ajoutant la méthode publique `nouvel_article()`

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {
// (...)
  public function nouvel_article() {
    if (!$this->auth_user->is_connected) {
      redirect('blog/index');
    }
    $this->load->helper('form');
    $this->load->library('form_validation');
    $this->load->model('article_status');
    $this->set_blog_post_validation();

    $data['title'] = "Nouvel article";

    if ($this->form_validation->run() == TRUE) {
      $this->load->model('article');
      $this->article->author_id = $this->auth_user->id;
      $this->article->content = $this->input->post('content');
      $this->article->status = $this->input->post('status');
      $this->article->title = $this->input->post('title');
      $this->article->save();
      if ($this->article->is_found) {
        redirect('blog/index');
      }
    }
    $this->load->view('common/header', $data);
    $this->load->view('blog/form', $data);
```

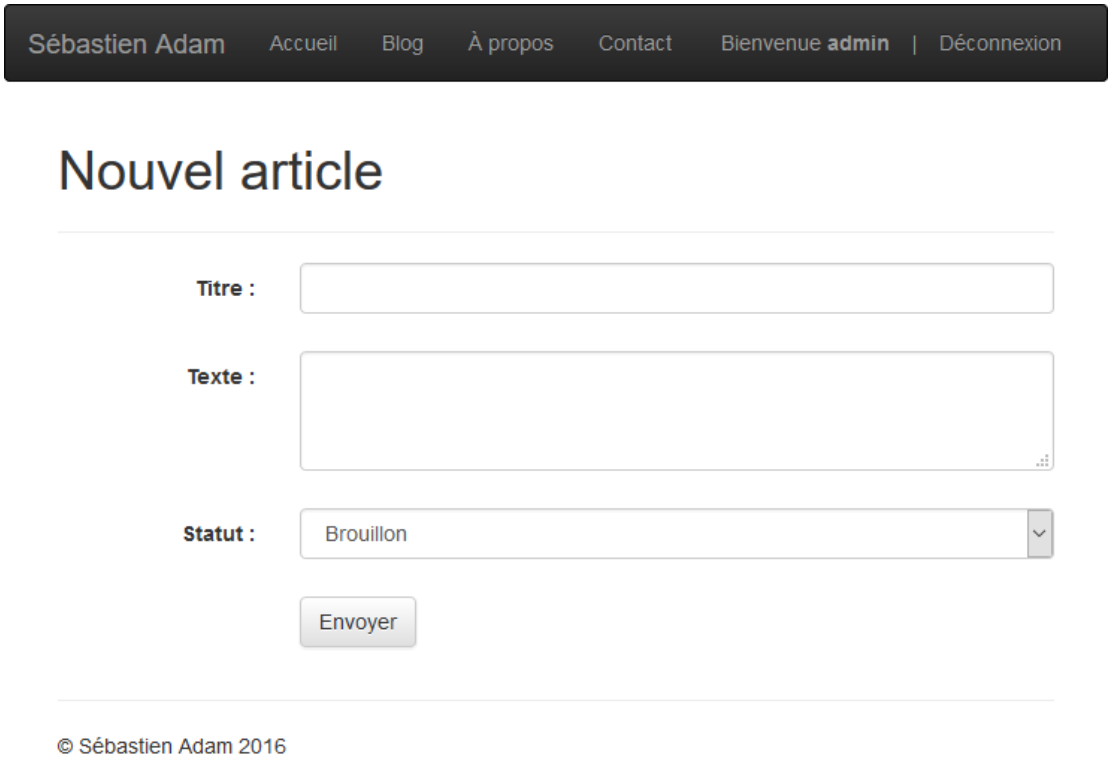

FIGURE 4.3 – Formulaire de création d'un article du blog

```
30      $this->load->view('common/footer', $data);
31    }
32
33    protected function set_blog_post_validation() {
34      $list = join(',', $this->article_status->codes);
35      $this->form_validation->set_rules('title', 'Titre', 'required|max_length[64]
            ');
36      $this->form_validation->set_rules('content', 'Contenu', 'required');
37      $this->form_validation->set_rules('status', 'Statut', 'required|in_list[' .
            $list . ']');
38    }
39  }
```

La première chose qui est faite dans la méthode `nouvel_article()`, c'est de vérifier que l'utilisateur est bien identifié. Si ce n'est pas le cas, nous allons afficher la page d'accueil de notre blog (la liste des articles).

Nous avons déjà utilisé le fichier `application/config/form_validation.php` pour valider des données d'un formulaire. Mais ici, nous allons faire un peu différemment. Personnellement, je ne conseille pas de mélanger différentes techniques dans un même projet, mais on va dire que c'est l'exercice qui le veut.

Donc, pour la validation du formulaire, j'ai créé une méthode `set_blog_post_validation()`, et cela pour deux raisons. Tout d'abord, je vais utiliser les mêmes paramètres de validation pour un nouvel article et pour la modification d'un article. Vous avez raison, on peut le faire avec le fichier de configuration, mais c'est un peu brouillon. Deuxièmement, je récupère la liste des codes possibles pour le statut à partir du modèle `Article_status`.

Si les données du formulaire sont valides, nous chargeons le modèle `Article`, remplissons les données et le sauvegardons. Si cela a fonctionné, nous affichons la liste des articles.

Maintenant, il ne vous reste plus qu'à aller à la page http://localhost/blog/nouvel_article.html pour afficher le formulaire permettant de créer votre premier article sur votre tout nouveau blog. Votre formulaire devrait ressembler à la figure 4.3 à la page 56.

4.5 Affichage d'un article

Maintenant que vous avez publié votre premier article, il s'agit de l'afficher. Pour ce chapitre, je ne vais pas respecter l'ordre des deux précédents (modèle, vue, contrôleur), car il y a quelques petites choses à expliquer. Alors, attendez bien d'avoir fini cette partie avant de vouloir afficher votre page, nous allons faire des trucs sympas.

4.5.1 Modèle

Nous avons déjà défini notre modèle à la section 4.4.1 à la page 50. Il n'y a rien à changer.

4.5.2 Contrôleur

Dans notre contrôleur `application/controllers/Blog.php`, nous allons définir une méthode `article()` prenant l'identifiant de l'article à afficher :

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {
// (...)
  public function article($id = NULL) {
    if (!is_numeric($id)) {
      redirect('blog/index');
    }
    $this->load->helper('date');
    $this->load->model('article');
    $this->load->model('article_status');
    $this->article->load($id, $this->auth_user->is_connected);

    if ($this->article->is_found) {
      $data['title'] = htmlentities($this->article->title);

      $this->load->view('common/header', $data);
      $this->load->view('blog/article', $data);
      $this->load->view('common/footer', $data);
    } else {
      redirect('blog/index');
    }
  }
}
```

Nous allons d'abord nous assurer que l'identifiant donné est bien un nombre (dans la base de données, les identifiants sont numériques). Ensuite, chargeons nos différentes ressources (*helper*, modèles). Et enfin, nous chargeons les données depuis la base de données. Si le visiteur est identifié, il aura accès aux articles non publiées, sinon, seuls les articles publiés seront visibles. Si l'article est trouvé, il sera affiché. Sinon, nous serons redirigé sur la page d'accueil du blog.

4.5.3 Routage

Normalement, pour accéder à notre article, nous devons utiliser l'adresse http://localhost/blog/article/<id>. Sachez que les moteurs de recherche utilisent les URL pour évaluer la pertinence des pages par rapport aux termes recherchés, et l'adresse que nous avons n'est pas très parlante.

Nous allons donc optimiser nos adresses pour que nos pages soient mieux référencées dans les moteurs de recherche (en anglais on parle de SEO, *Search Engine Optimization*). Je vous avais parlé du champ `alias` qui allait être utilisé pour générer nos URL. Voici le moment de le mettre en pratique. 😃

Donc, pour optimiser nos adresses, nous pourrions avoir une URL du style http://localhost/blog/article/<alias>. Mais voilà, dans notre base de données, nous pourrions avoir deux articles avec le même titre, et donc avec le même alias. Il nous faut donc garder l'identifiant dans l'adresse. Les possibilités de combinaison de l'identifiant et de l'alias sont nombreuses, il nous faut donc en choisir une : http://localhost/blog/<alias>_<id>.html.

Vous constaterez que j'ai retiré la partie « article » de l'URL. C'est tout à fait possible. Nous pourrions même retirer la partie « blog », mais ce serait plus aléatoire pour le routage, et moins cohérent pour les visiteurs. Voici comment nous allons configurer cela dans le fichier `application/config/routes.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
// (...)
$route['blog/(:any)_(:num)'] = 'blog/article/$2'; // $2 se réfère au contenu du
                                                  // deuxiéme jeu de parenthéses
// (...)
```

Ainsi, http://localhost/blog/<alias>_<id>.html seront redirigées vers http://localhost/blog/article/<id>. Notez que nous pourrions mettre n'importe quoi comme alias, il n'est pas pris en compte pour le routage. Il ne sert que pour les moteurs de recherche. Si vous utiliser l'identifiant d'un article avec l'alias d'un autre, c'est l'article dont l'identifiant a été donné qui sera affiché.

4.5.4 Vue

Maintenant que nous avons défini la manière d'accéder à notre article, nous allons définir la vue `application/views/blog/article.php` :

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <div class="row">
        <div class="col-md-12">
          <?= heading($title); ?>
          <p>
            <small>
              <?= nice_date($this->article->date, 'd/m/Y'); ?>
              -
              <?= $this->article->author ?>
              <?php if ($this->auth_user->is_connected) : ?>
                -
                <?= $this->article_status->label[$this->article->status]; ?>
              <?php endif; ?>
            </small>
```

```
          </p>
          <hr />
        </div>
      </div>
      <div class="row">
        <div class="col-md-12">
          <?= nl2br(htmlentities($this->article->content)); ?>
        </div>
      </div>
    </div>
    <div class="col-md-2">
      <ul class="nav nav-pills nav-stacked">
        <li><?= anchor('blog/index', "Liste articles") ?></li>
        <?php if ($this->auth_user->is_connected) : ?>
          <li><?= anchor('blog/nouvel_article', "Nouvel article") ?></li>
        <?php endif; ?>
      </ul>
    </div>
  </div>
</div>
```

Notre vue aura deux parties : l'article en lui-même et un menu. Pour l'article, nous allons afficher à peu près les mêmes informations que pour notre résumé, sauf qu'ici, nous aurons l'article en entier. Pour le menu, nous aurons un lien vers la page d'accueil de notre blog et, pour les personnes identifiées, un lien pour la création d'un nouvel article.

Nous allons également modifier la vue `application/views/blog/article_resume.php` de l'article afin d'ajouter un lien pour accéder à la page de l'article :

```
<?php
$article_url = 'blog/' . $alias . '_' . $id;
?>
<div class="col-md-4">
  <?= heading(anchor($article_url, htmlentities($title)), 2); ?>
  <p>
    <small>
      <?= nice_date($date, 'd/m/Y'); ?>
      -
      <?= $author ?>
      <?php if ($this->auth_user->is_connected) : ?>
        -
        <?= $this->article_status->label[$status]; ?>
      <?php endif; ?>
    </small>
  </p>
  <p><?= nl2br(htmlentities($content)); ?>... <?= anchor($article_url, "Lire la
     suite"); ?></p>
</div>
```

Nous avons ajouté du code PHP pour affecter l'URI de l'article dans une variable et utilisé cette variable pour générer un lien sur le titre et après le résumé de l'article. Maintenant, en affichant la page d'accueil du blog, vous pouvez cliquer sur le titre de l'article à visualiser ou sur son lien « Lire la suite » (si, si, vous pouvez le faire maintenant, ça doit marcher 😁).

4.6 Modification d'un article

Maintenant que nous avons publié et affiché un article, nous allons le modifier (je ne sais pas pour vous, mais moi, il faut toujours que je fasse l'une ou l'autre erreur lorsque j'écris).

En fait, le processus de modification d'un article est similaire à celui de sa création. Nous allons simplement modifier un peu la vue et le contrôleur pour pouvoir utiliser la même méthode pour la création et la modification.

4.6.1 Contrôleur

Nous allons simplement modifier la méthode `nouvel_article()` du contrôleur `application/controllers/Blog.php` comme suit :

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {
// (...)
  public function edition($id = NULL) { // modification nom méthode et
                                        // $id comme paramétre
    if (!$this->auth_user->is_connected) {
      redirect('blog/index');
    }
    $this->load->helper('form');
    $this->load->library('form_validation');
    $this->load->model('article_status');
    $this->load->model('article');      // on charge le modéle Article() ici

    if ($id !== NULL) {          // si identifiant donné, modification
      if (is_numeric($id)) {     // vérification validité de l'identifiant
        $this->article->load($id, TRUE);
        if (!$this->article->is_found) {
          redirect('blog/index');
        }
      } else {
        redirect('blog/index');
      }
      $data['title'] = "Modification article";
    } else {                     // si aucun identifiant donné, création
      $data['title'] = "Nouvel article";
      $this->article->author_id = $this->auth_user->id;
    }

    $this->set_blog_post_validation();

    if ($this->form_validation->run() == TRUE) {
      // le modéle Article() n'est plus chargé ici
      // l'auteur de l'article n'est plus défini ici
      $this->article->content = $this->input->post('content');
      $this->article->status = $this->input->post('status');
      $this->article->title = $this->input->post('title');
      $this->article->save();
      if ($this->article->is_found) {
        redirect('blog/' . $this->article->alias . '_' . $this->article->id);
      }
    }
```

```
44      $this->load->view('common/header', $data);
45      $this->load->view('blog/form', $data);
46      $this->load->view('common/footer', $data);
47    }
48  // (...)
49  }
```

Tout d'abord, la méthode `nouvel_article()` sera renommée en `edition()` (je vous laisse deviner pourquoi). Aussi, cette méthode acceptera maintenant un éventuel identifiant d'article en paramètre.

Le modèle `Article()` sera chargé dès le départ, et non plus lorsque les données du formulaire auront été validées. En effet, nous devrons pouvoir retrouver un article si un identifiant est donné.

Si un identifiant est donné, nous sommes dans le cas d'une modification. Nous allons alors valider cet identifiant. Il doit être numérique et correspondre à un article existant. Si ce n'est pas le cas, nous chargerons la page d'accueil du blog. Si l'identifiant est correct, nous chargerons l'article concerné.

Si aucun identifiant n'est donné, nous sommes dans le cas d'une création. J'ai décidé de définir l'identifiant de l'auteur ici. C'est une chose discutable. L'auteur est-il celui qui crée l'article, ou celui qui le modifie ? J'ai opté pour la première solution, « parce que c'était la meilleure[4] ».

Pour le reste, cela ne change pas trop de ce que nous faisions avant. Remarquez toutefois la définition du titre (via la variable `$data['title']`). Ce dernier sera différent suivant que nous allons créer ou modifier un article.

4.6.2 Vue

Nous allons apporter quelques petites modifications sur notre vue `application/views/blog/form.php`. Voici les changements :

```
1  <!-- (...) -->
2  <?= form_open(uri_string(), ['class' => 'form-horizontal']); ?>
3  <!-- (...) -->
4  <?= form_input(['name' => "title", 'id' => "title", 'class' => 'form-control'],
       set_value('title', $this->article->title)) ?>
5  <!-- (...) -->
6  <?= form_textarea(['name' => "content", 'id' => "content", 'class' => 'form-
       control'], set_value('content', $this->article->content)) ?>
7  <!-- (...) -->
8  <?= form_dropdown("status", $this->article_status->text, set_value('status',
       $this->article->status), ['id' => "content", 'class' => 'form-control']) ?>
9  <!-- (...) -->
```

La première chose à modifier est l'adresse où les données du formulaire sont envoyées. En effet, dans notre contrôleur, nous avons défini qu'il fallait donner un identifiant d'article pour être dans le cas d'une modification, et que sans identifiant, nous sommes dans le cas d'une création d'article. Nous utiliserons alors la fonction `uri_string()` comme premier paramètre de la fonction `form_open()`.

Nous devons également modifier le remplissage du formulaire. Nous allons donc donner les données du modèle comme deuxième paramètre de la fonction `set_value()`. Ainsi, le formulaire sera d'abord rempli avec les données postées. S'il n'y a pas de donnée postée, ce seront les données du modèle qui seront utilisées. Et s'il n'y a pas de données dans le modèle non plus, me direz-vous ? Et bien, le formulaire sera vide.

Il faudra également modifier le menu de la vue `application/views/blog/article.php` pour modifier ou ajouter un nouvel article :

4. lisez : « il fallait faire un choix et il fallait le justifier »

```
<ul class="nav nav-pills nav-stacked">
  <li><?= anchor('blog/index', "Liste articles") ?></li>
  <?php if ($this->auth_user->is_connected) : ?>
    <li>
      <?= anchor(['blog', 'edition', $this->article->id], "Modifier article") ?>
    </li>
    <li><?= anchor('blog/edition', "Nouvel article") ?></li>
  <?php endif; ?>
</ul>
```

Et pour finir, n'oublions pas de modifier la vue `application/views/blog/index.php` pour que le lien de création d'un nouvel article pointe sur notre nouvelle méthode :

```
  <div class="row">
    <div class="col-md-10">
      <ul class="nav nav-pills nav-justified">
        <?php if ($this->auth_user->is_connected) : ?>
          <li role="presentation"><?= anchor('blog/edition', "Nouvel article");
             ?></li>
        <?php endif; ?>
      </ul>
    </div>
    <div class="col-md-2">
      <p class="text-right">Nombre d'articles: <?= $this->articles->num_items;
         ?></p>
    </div>
  </div>
```

4.7 Suppression d'un article

Pour supprimer un article, il suffit de changer son statut. Mais nous allons créer un bouton de suppression pour simplifier la manœuvre... Aussi, je vais vous parler un peu d'AJAX (encore un truc à placer dans un entretien d'embauche, même si c'est un peu passé de mode). L'idée est ici de demander une confirmation avant de supprimer un article.

4.7.1 Modèle

Pour supprimer un article, nous allons simplement changer son statut en « `D` ». Pour ce faire, nous allons créer la méthode `delete()` dans notre modèle `application/models/Article.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Article extends CI_Model {
// (...)
  public function delete() {
    if ($this->is_found) {
      $this->_status = 'D';
      $this->save();
    }
  }
// (...)
}
```

Notez qu'il ne faut pas sauvegarder les données après avoir appeler la méthode `delete()`. Notez également que l'article n'est pas effectivement supprimé de la base de données. Il est simplement rendu inaccessible pour le public.

4.7.2 Vue

Nous allons devoir créer deux vues : la première pour la demande de confirmation de la suppression et l'autre pour afficher le résultat. En fait, nous n'allons créer qu'une seule vue et deux vues partielles. Nous appellerons l'une ou l'autre suivant le cas dans lequel nous seront. C'est au niveau du contrôleur que nous déterminerons dans quelle situation nous sommes.

Voici donc notre vue principale `application/views/blog/delete.php` :

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <?= heading($title); ?>
      <hr />
    </div>
  </div>
  <div class="row">
    <div class="col-md-12">
      <?php $this->load->view('blog/delete_' . $action); ?>
    </div>
  </div>
</div>
```

Notez que c'est la variable `$action` qui détermine la situation dans laquelle nous sommes. La valeur de cette variable sera déterminée dans notre contrôleur.

Voici la vue `application/views/blog/delete_confirm.php` qui sera utilisée pour demande la confirmation de la suppression d'un article :

```
<p class="alert␣alert-danger" role="alert">
  Êtes-vous sûr(e) de vouloir supprimer cet article?
</p>
<?= form_open(uri_string(), ['class' => 'form-horizontal']); ?>
<div class="form-group">
  <p style="text-align:␣center">
    <?= form_submit('confirm', "Supprimer", ['class' => "btn␣btn-default"]); ?>
       
    <?= anchor(['blog', $this->article->alias . '_' . $this->article->id], "
       Annuler", ['id' => 'cancel_delete', 'class' => 'btn btn-default']) ?>
  </p>
</div>
<?= form_close() ?>
```

Cette vue affiche un message invitant l'utilisateur à confirmer son choix. Pour ce faire, il postera un formulaire. Cette méthode permet d'empêcher la suppression d'un article via une requête directe (`GET`). Si l'utilisateur change d'avis, il aura un lien pour revenir à l'article.

Voici la vue `application/views/blog/delete_result.php` qui sera utilisée pour afficher le résultat de la suppression d'un article :

```
<p class="alert␣alert-success" role="alert">
  L'article a été supprimé.
</p>
<p style="text-align:␣center">
  <?= anchor('blog/index', "Fermer", ['class' => "btn␣btn-default"]); ?>
```

```
</p>
```

La vue affichera simplement un message indiquant que la suppression a bien eu lieu. Ici, je pars du principe que si cela ne s'est pas bien déroulé, le système « plantera » et la vue ne s'affichera pas. Je sais que ce n'est pas bien. Il faudrait gérer ce genre d'erreur et ne jamais faire cela dans la « vraie vie ». 

Il faudra également modifier le menu de la vue `application/views/blog/article.php` afin d'ajouter une entrée pour pouvoir supprimer un article :

```
<!-- (...) -->
<ul class="nav␣nav-pills␣nav-stacked">
  <li><?= anchor('blog/index', "Liste␣articles") ?></li>
  <?php if ($this->auth_user->is_connected) : ?>
    <li>
      <?= anchor(['blog', 'edition', $this->article->id], "Modifier") ?>
    </li>
    <li>
      <?= anchor(['blog', 'suppression', $this->article->id], "Supprimer") ?>
    </li>
    <li>
      <?= anchor('blog/edition', "Nouvel␣article") ?>
    </li>
  <?php endif; ?>
</ul>
<!-- (...) -->
```

4.7.3 Contrôleur

Nous allons ajouter la méthode `suppression()`, qui prend l'identifiant d'un article comme paramètre, à notre contrôleur `application/controllers/Blog.php` :

```
<?php
defined('BASEPATH') OR exit('No␣direct␣script␣access␣allowed');

class Blog extends CI_Controller {
// (...)
  public function suppression($id = NULL) {
    if (!$this->auth_user->is_connected) {
      redirect('blog/index');
    }
    if (!is_numeric($id)) {
      redirect('blog/index');
    }
    $this->load->model('article');
    $this->article->load($id, TRUE);
    if (!$this->article->is_found) {
      redirect('blog/index');
    }
    if ($this->input->post('confirm') === NULL) {
      $data['action'] = "confirm";
    } else {
      $this->article->delete();
      $data['action'] = "result";
    }
    $data['title'] = "Suppression␣article";
    $this->load->helper('form');
```

```
26      $this->load->view('common/header', $data);
27      $this->load->view('blog/delete', $data);
28      $this->load->view('common/footer', $data);
29    }
30  }
```

Dans un premier temps, nous vérifions si l'utilisateur est bien identifié. Ensuite, nous vérifions si l'identifiant est bien un nombre. Nous chargeons alors l'article correspondant et vérifions qu'il a bien été trouvé. Si le formulaire de confirmation n'a pas été posté, nous l'affichons. Sinon, nous supprimons l'article et affichons le résultat. Cela prend quelques lignes, mais le travail du contrôleur n'est pas très compliqué.

4.7.4 Confirmation AJAX

Je ne vais malheureusement pas vous faire un cours d'AJAX ici. Je vais simplement vous dire qu'il s'agit d'une technique permettant d'envoyer des requêtes au serveur sans recharger toute la page. Pour (beaucoup) plus de détails, je vous invite à consulter l'aide en ligne sur le réseau MDN. Pour réaliser notre requête, nous aurons besoin d'un script JavaScript. Je ne n'épancherai pas non plus sur cette partie, mais plutôt sur notre code PHP.

Pour nous simplifier les choses, nous allons créer une fenêtre modale grâce à Bootstrap et ajouter un identifiant à notre lien de suppression dans la vue de l'article. Modifions donc notre vue `application/views/blog/article.php` :

```
<div class="modal fade" id="deleteModal" tabindex="-1" role="dialog" aria-
    labelledby="myModalLabel">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal" aria-label="
            Close"><span aria-hidden="true">&times;</span></button>
        <h4 class="modal-title" id="myModalLabel">Suppression de l'article</h4>
      </div>
      <div class="modal-body" id="deleteModalContent">
      </div>
    </div>
  </div>
</div>
<!-- (...) -->
<ul class="nav nav-pills nav-stacked">
  <!-- (...) -->
    <li>
      <?= anchor(['blog', 'suppression', $this->article->id], "Supprimer", ['id'
          => 'menu_delete_article']) ?>
    </li>
  <!-- (...) -->
</ul>
<!-- (...) -->
```

Ensuite, créons un script JavaScript `js/article.js` :

```
$(function () {
  $('#deleteModal').on('hidden.bs.modal', function () {
    // Efface le contenu de la fenêtre modale à la fermeture
    $('#deleteModalContent').html('');
  });
  $('#menu_delete_article').click(function () {
```

```
    var delete_article_url = $(this).attr('href');
    $.ajax({
      url: delete_article_url
    }).done(function (data) {
      $('#deleteModalContent').html(data);
      $('#cancel_delete').click(function () {
        $('#deleteModal').modal('hide');
        return false;
      });
      $('#deleteModal').modal('show');
    });
    return false;
  });
});
```

Si vous ne comprenez pas bien ce qui a été fait ici, ne vous inquiétez pas. J'ai utilisé JQuery pour intercepter le clic sur le lien de suppression et de le remplacer par une requête AJAX. J'affiche ensuite le résultat de cette requête dans la fenêtre modale que nous venons d'ajouter à notre vue. Enfin, j'intercepte aussi le clic sur le lien d'annulation de la suppression pour fermer la fenêtre modale plutôt que de recharger la page.

Nous allons devoir charger notre script dans notre page, mais après avoir chargé JQuery. Comme ce dernier est chargé dans la vue `application/views/common/footer.php`, nous devons modifier cette dernière afin d'ajouter un script après ceux déjà référencés :

```
<!-- (...) -->
<script src="<?= base_url("js/jquery-2.1.4.min.js") ?>"></script>
<script src="<?= base_url("js/bootstrap.min.js") ?>"></script>
<?php
if (isset($script)) {
  echo $script;
}
?>
<!-- (...) -->
```

J'ai défini ici la possibilité de créer une variable `$script` dans le contrôleur afin d'y insérer un éventuel script. Nous allons tout de suite voir comment l'utiliser.

Voici donc le code qui nous intéresse vraiment dans le contrôleur `application/controllers/Blog.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {
  public function article($id = NULL) {
    // (...)
      $data['title'] = htmlentities($this->article->title);
      $data['script'] = '<script src="' . base_url('js/article.js') . '"></
          script>';
    // (...)
  }
// (...)
  public function suppression($id = NULL) {
    // (...)
    $this->load->helper('form');
    if ($this->input->is_ajax_request()) {
      // nous avons reçu une requête ajax
      $this->load->view('blog/delete_confirm');
```

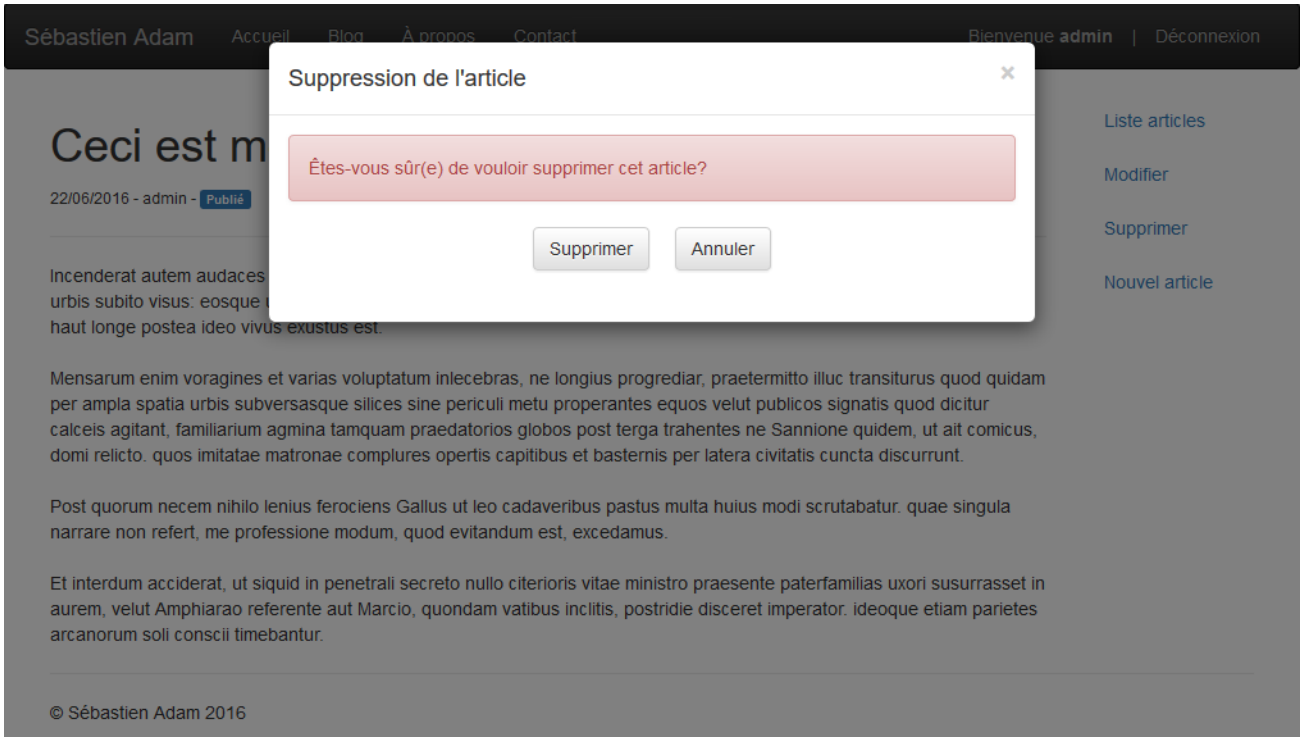

FIGURE 4.4 – Confirmation de la suppression d'article avec AJAX

```
18      } else {
19        // nous avons reçu une requête classique
20        if ($this->input->post('confirm') === NULL) {
21          $data['action'] = "confirm";
22        } else {
23          $this->article->delete();
24          $data['action'] = "result";
25        }
26        $data['title'] = "Suppression article";
27        $this->load->view('common/header', $data);
28        $this->load->view('blog/delete', $data);
29        $this->load->view('common/footer', $data);
30      }
31    }
32  }
```

Dans la méthode `article()`, j'ai créé la variable `$data['script']` (qui sera interprétée en tant que `$script` dans notre vue) afin d'insérer le script que nous avons créé précédemment dans la page de l'article.

Examinons maintenant la méthode `suppression()`, celle qui nous intéresse vraiment. CodeIgniter nous fournit la possibilité de détecter si une requête AJAX a été effectuée grâce à `$this->input->is_ajax_request()`. Si c'est le cas, nous ne retournons que la vue partielle `application/views/blog/delete_confirm.php` (dans notre exemple, la requête AJAX n'est effectuée que dans le cas de la confirmation). Notre script `js/article.js` va réceptionner cette vue et l'afficher dans la fenêtre modale. Vous devriez avoir un affichage qui ressemble à la figure 4.4 à la page 67.

En cliquant sur le bouton « Supprimer », l'article sera supprimé et la page de résultat s'affichera. Si vous cliquez sur « Annuler », la fenêtre modale sera simplement fermée.

4.8 Exercices

Normalement, si vous avez bien suivi ce cours, vous devriez avoir un site avec quelques pages statiques, un mécanisme qui vous permet de vous identifier, et un blog. Si ce n'est pas le cas, vous pouvez utiliser le code se trouvant dans le fichier « `site_bas3.7z` » accompagnant ce document (également disponible sur mon site personnel).

4.8.1 Énoncés

Bouton de publication

Dans la vue affichant un article, créez un bouton permettant la publication d'un article, à l'instar du bouton de suppression. Lorsqu'on clique sur le bouton de publication, le statut de l'article change (sans demander de confirmation) et la page de l'article se rafraichit.

Affichage sélectif des boutons de publication et de suppression

Faites en sorte que le bouton de suppression ne s'affiche pas si l'article est supprimé et que le bouton de publication ne s'affiche pas si l'article est publié.

4.8.2 Corrections

Voici les solutions que je suggère pour les exercices proposés. Je vous invite ardemment à d'abord faire les exercices avant de lire ces solutions.

Bouton de publication

Nous résoudrons cet exercice en deux temps : créer un lien dans le menu de la vue de l'article et créer une méthode dans le contrôleur du blog pour publier l'article.

Modifions donc la vue `application/views/blog/article.php` afin d'ajouter le lien de publication de l'article :

```
<!-- (...) -->
<ul class="nav nav-pills nav-stacked">
  <li><?= anchor('blog/index', "Liste articles") ?></li>
  <?php if ($this->auth_user->is_connected) : ?>
    <li><?= anchor(['blog', 'edition', $this->article->id], "Modifier") ?></li>
    <li><?= anchor(['blog', 'publication', $this->article->id], "Publier") ?></
       li>
    <li><?= anchor(['blog', 'suppression', $this->article->id], "Supprimer", ['
       id' => 'menu_delete_article']) ?></li>
    <li><?= anchor('blog/edition', "Nouvel article") ?></li>
  <?php endif; ?>
</ul>
<!-- (...) -->
```

Maintenant, ajoutons la méthode `publication()` à notre contrôleur `application/controllers/Blog.php` :

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Blog extends CI_Controller {
```

```
// (...)
  public function publication($id = NULL) {
    if (!$this->auth_user->is_connected) {
      redirect('blog/index');
    }
    if (!is_numeric($id)) {
      redirect('blog/index');
    }
    $this->load->model('article');
    $this->article->load($id, TRUE);
    if (!$this->article->is_found) {
      redirect('blog/index');
    }
    $this->article->status = 'P';
    $this->article->save();
    redirect('blog/' . $this->article->alias . '_' . $id);
  }
}
```

Une vérification est faite pour savoir si l'utilisateur est connecté. Si ce n'est pas le cas, nous sommes redirigé vers la page d'accueil du blog.

Nous vérifions ensuite que l'identifiant est valide, nous chargeons les données de l'article à partir de la base de données, modifions le statut de l'article, et sauvegardons le tout. Une fois que c'est fait, nous chargeons la page de l'article.

Affichage sélectif des boutons de publication et de suppression

Ici, nous n'allons modifier que la vue `application/views/blog/article.php`. Nous ferons un test pour chaque bouton concerné pour savoir dans quel état est l'article.

```
<!-- (...) -->
<ul class="nav␣nav-pills␣nav-stacked">
  <li><?= anchor('blog/index', "Liste␣articles") ?></li>
  <?php if ($this->auth_user->is_connected) : ?>
    <li><?= anchor(['blog', 'edition', $this->article->id], "Modifier") ?></li>
    <?php if ($this->article->status !== 'P'): ?>
      <li><?= anchor(['blog', 'publication', $this->article->id], "Publier") ?><
          /li>
    <?php endif; ?>
    <?php if ($this->article->status !== 'D'): ?>
      <li><?= anchor(['blog', 'suppression', $this->article->id], "Supprimer",
          ['id' => 'menu_delete_article']) ?></li>
    <?php endif; ?>
    <li><?= anchor('blog/edition', "Nouvel␣article") ?></li>
  <?php endif; ?>
</ul>
<!-- (...) -->
```

4.9 Conclusion

Après avoir réalisé ce tutoriel, vous devriez être l'heureux propriétaire d'un blog réalisé avec CodeIgniter. Bon, vous n'allez pas concurrencer d'autres produits comme WordPress, Joomla!, Drupal et consorts. Et le chemin est encore long pour finaliser ce projet de blog, mais si cela vous dit, vous avez tous les outils en main pour cela.

J'espère que vous aurez reçu un bon aperçu de la simplicité d'utilisation de CodeIgniter, et l'envie de continuer avec. Il est particulièrement bien adapté pour des projets de petite (ou même moyenne) taille qui doivent aller vite. Je ne vous le recommanderais peut-être pas pour de (très) gros projets (quoique...).

Vous pouvez télécharger les ressources concernant ce tutoriel sur mon site personnel, et notamment une version PDF ainsi que le code source du projet.

Maintenant, c'est à vous de jouer. 

Table des figures