MINZ Is Not Zend

version 1.0

Documentation

en cours de rédaction

Marien Fressinaud

24 mars 2012

Table des matières

1 Fonctionnement général **3**
1.1 Principes fondamentaux . . . 3
1.2 Architecture . . . 3
1.3 Pré-requis et installation . . . 5

2 Développer avec MINZ **6**
2.1 Bien commencer : comprendre le index.php . . . 6
2.2 Tour d'horizon du répertoire application . . . 6
2.3 Écrire un Controller . . . 7
2.4 Écrire une View . . . 8
2.5 Écrire un Model . . . 8
2.6 Lier le tout . . . 10
2.7 Utiliser l'AppBootstrap . . . 11
2.8 Router les URLs . . . 12

3 Aller plus loin **14**
3.1 Utilisation d'un layout . . . 14
3.2 Les variables de session . . . 14
3.3 Historisez le parcours de vos utilisateurs . . . 14
3.4 Utilisation du système de Cache . . . 14
3.5 Internationalisez votre application . . . 14
3.6 Écrire sa propre librairie . . . 14
3.7 La View en détails . . . 14
3.8 Sécuriser son application MINZ . . . 14

1 Fonctionnement général

1.1 Principes fondamentaux

Pour bien commencer cette documentation, il est important de définir ce qu'est MINZ et de délimiter l'usage qu'il est possible d'en faire.

MINZ est un **framework PHP**, c'est-à-dire qu'il propose une architecture particulière pour l'écriture d'application PHP. On peut le voir comme un squelette, et l'application comme les muscles, cerveaux, peau, etc. Ce framework repose lui-même sur la **modélisation MVC** (pour Model View Controller). Le modèle MVC permet de séparer logiquement les données (un utilisateur avec un nom, un prénom, un mot de passe par exemple), leur représentation (la façon dont on va les afficher) et leur traitement. Cela permet d'avoir une application facile à maintenir.

MINZ est fortement inspiré de *Zend Framework*, qui est un autre framework PHP. Bien qu'inspiré, il s'en éloigne par bien des aspects, d'où son nom en clin d'oeil : *MINZ Is Not Zend*. Il se veut notamment bien plus léger et plus facile à mettre en place afin de faciliter le déploiement d'applications MINZ. Si vous connaissez déjà Zend Framework, il est certain que vous y trouverez de nombreuses similitudes, notamment au niveau de l'architecture.

J'ai dans l'idée de rendre MINZ "compatible" XMPP. C'est-à-dire que la création d'applications reposant sur XMPP sera grandement facilité. Cela pour démocratiser ce protocole très puissant, et surtout très utile.

Si vous souhaitez démarrer rapidement dans le domaine des frameworks, je ne peux que vous conseiller de commencer par MINZ ! :)

1.2 Architecture

MINZ repose sur trois répertoires principaux :

- `app` : C'est dans celui-ci que tout le code PHP/HTML de l'application sera écrit, vous n'aurez donc à vous préoccuper **presque** que de celui-ci.
- `lib` : Tout le code du framework en lui-même se trouve ici, en principe vous n'avez pas besoin d'y toucher. Nous verrons plus tard qu'il est possible d'écrire sa propre librairie, qui se retrouvera donc ici.
- `public` : C'est la partie accessible de l'extérieur, on y retrouve le fichier `index.php` principal qui lance l'application ainsi que les thèmes, les scripts javascript, les images, etc. Nous verrons plus loin qu'il est possible de mettre son contenu à la racine si jamais on ne peut pas faire pointer le nom de domaine sur ce répertoire.

Comme dit ci-dessus, le répertoire `app` est le plus important car c'est celui-ci qui contiendra les contrôleurs, modèles et vues ainsi que d'autres éléments que nous détaillerons plus loin.

Un schéma du fonctionnement est sans doute nécessaire pour bien comprendre comment tout cela fonctionne :

Ce qu'il faut voir ici, c'est l'enchaînement des actions (représentées par les flèches) et comprendre les couleurs. Les ronds représentent des classes PHP. On pourra noter des soucis d'enchaînement (je pense surtout au lancement de la classe Route). Cela devrait

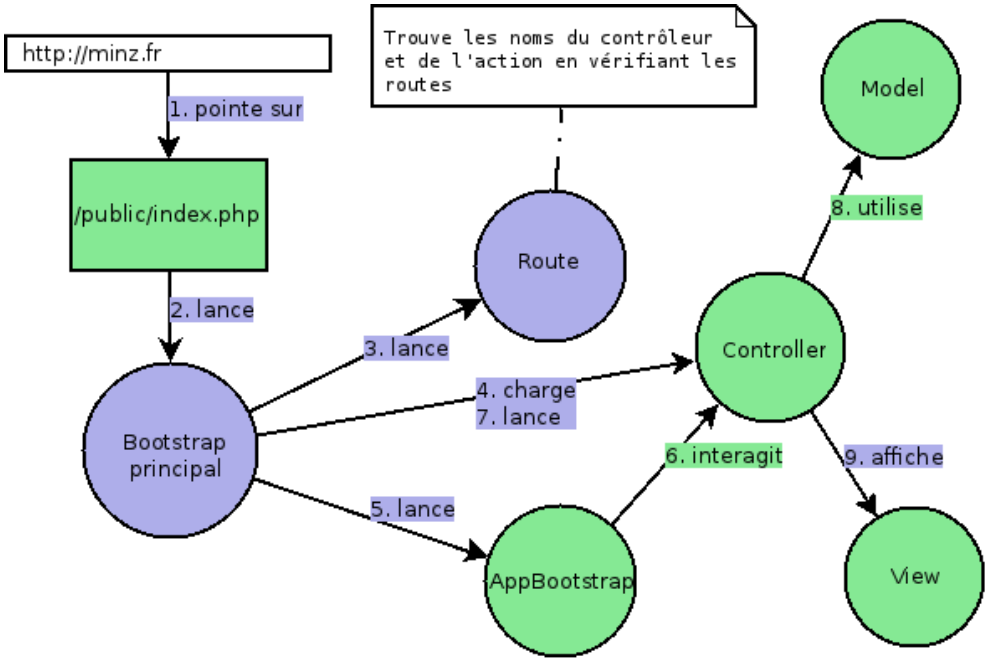

FIG. 1 – Schéma du fonctionnement de MINZ

être revu dans la version 2 de MINZ, grâce notamment à l'utilisation des *design patterns*.

Les flèches sont numérotées de 1 à 9, et c'est donc l'ordre dans lequel se déroule le fonctionnement du framework. On peut donc voir que lorsque l'url http://minz.fr est demandée, on va d'abord chercher le fichier `/public/index.php` qui se charge de lancer le `Bootstrap`, qui va lui-même demander à récupérer le nom du contrôleur et de l'action à la classe `Route`, etc.

Pour ce qui est des couleurs, tout ce qui est en bleu représente des classes et méthodes dont vous n'avez pas à vous soucier puisque c'est le framework qui s'en charge lui-même. Tout ce qui est de couleur verte, c'est à vous de le coder. Notez toutefois, si on prend l'exemple du contrôleur, que celui-ci hérite d'une classe parente, donc vous n'avez pas **tout** à écrire. Si vous avez suivi, en principe, le bleu se trouve dans le répertoire `lib`, et le vert, dans le répertoire `app` (à l'exception de `index.php` qui se trouve dans `public`).

Si tout cela vous paraît un peu obscur, c'est sans doute que mon schéma est mauvais... ou que l'architecture n'est pas terrible... ou les deux ! Il ne tient qu'à vous de me le faire savoir.

Par ailleurs, il est important de bien comprendre l'importance de l'action et du contrôleur qui lui est associé. Le framework nécessite de ces deux informations pour fonctionner. Pour cela, on lui indique via l'URL avec la variable `$_GET['c']` le nom du contrôleur à exécuter. La variable `$_GET['a']` sert à donner le nom de l'action. Concrètement, le contrôleur sert à indiquer ce que l'on veut manipuler (par exemple, un Utilisateur) et l'action, une opération à faire sur ce contrôleur (par exemple, ajouter un Utilisateur en base de données).

1.3 Pré-requis et installation

Comme il est indiqué dans le fichier `README` fourni avec le framework, MINZ ne requiert que de peu de choses. La première est **un serveur Apache** pour l'hébergement. Il a besoin aussi de **PHP 5**. Pour le moment, je sais qu'il fonctionne sur PHP 5.2.9 et supérieur, mais je pense que des versions antérieures devraient fonctionner aussi. MINZ n'utilise pas beaucoup de fonctions "exotiques". Enfin, activer l'**URL Rewriting** est recommandé pour pouvoir utiliser correctement le système de routes. Sachez toutefois qu'il n'est pas obligatoire (vous devrez utiliser alors des URLs plus traditionnelles). Tout ceci est expliqué plus loin.
Et c'est tout ce que je préconise pour pouvoir utiliser le framework, soit des choses que l'on trouve chez la plupart des hébergeurs.

Passons à l'installation. Là encore, c'est assez simple. Pour le moment, vous pouvez trouver le code complet sur la page GitHub dédiée :
https://github.com/marienfressinaud/MINZ

Une fois les sources téléchargées, il ne vous reste plus qu'à décompresser l'archive sur votre serveur. Assurez-vous que votre nom de domaine pointe sur le répertoire `public` et modifiez le nom de domaine dans le fichier de configuration (`/app/configuration/application.ini`). Rendez-vous sur votre site à partir de votre navigateur, et vous devriez voir apparaître l'application par défaut de MINZ ! Félicitations.

Remarque : il se peut que vous ne puissiez faire pointer votre nom de domaine au bon endroit. Deux solutions s'offrent à vous :

1. Coupez/collez le contenu de `public` dans le répertoire dans lequel pointe votre nom de domaine. Des modifications sont alors à porter aux constantes de chemins dans le contenu de `index.php`.
2. Dans le fichier de configuration, lors de la modification du nom de domaine, ajoutez le chemin complémentaire. Par exemple, vous aurez la ligne `domain = "http ://www.minz.test/public` si votre nom de domaine pointe sur le répertoire parent de `public`.

2 Développer avec MINZ

2.1 Bien commencer : comprendre le index.php

Le code de ce fichier est déjà écrit par défaut dans le framework fourni. Il est toutefois important de le comprendre car c'est par là que tout commence. Le code que l'on peut trouver est (à quelque chose près) le suivant :

```
<?php
// Constantes de chemins
define('PUBLIC_PATH', realpath(dirname(__FILE__)));
define('LIB_PATH', realpath(PUBLIC_PATH.'/../lib'));
define('APP_PATH', realpath(PUBLIC_PATH.'/../app'));
define('LOG_PATH', realpath(PUBLIC_PATH.'/../log'));
define('CACHE_PATH', realpath(PUBLIC_PATH.'/../cache'));

// Ajout du repertoire /lib dans l'include_path
set_include_path(get_include_path().PATH_SEPARATOR.LIB_PATH);

// Parametres du lancement de l'application
$params = array(
'config_file' => APP_PATH.'/configuration/application.ini'
);

require_once('Bootstrap.class.php');

$bootstrap = Bootstrap::getInstance($params);
$bootstrap->init(); // lancement de l'application
```

Le principe ici est d'abord de définir des constantes pour les différents chemins pour faciliter leur utilisation. Ensuite, nous ajoutons notre chemin de librairie à l'`include_path` afin de pouvoir inclure les fichiers de librairie en indiquant simplement leur nom plutôt que leur chemin complet.

Le chemin vers le fichier de configuration est défini dans un tableau de paramètres. On pourra imaginer dans le futur ajouter d'autres informations à ce tableau.

Enfin, on lance le `Bootstrap`. Tout cela est plutôt simple et ne devrait pas poser de problème mais je pense qu'il est important d'en parler, ne serait-ce que pour avoir connaissance des constantes de chemins qui sont très utiles.

2.2 Tour d'horizon du répertoire application

Le répertoire `app` regroupe à peu près tous les fichiers que vous aurez à écrire. Son contenu est principalement du PHP, mais contient aussi du HTML pour les vues (l'extension est en fait .phtml pour pouvoir utiliser un peu de PHP à l'intérieur quand même). On y retrouve

- le répertoire `configuration` qui contient le fichier de configuration mais aussi le fichier décrivant les routes (nous y reviendrons après)
- le répertoire `controllers` regroupe les contrôleurs de l'application

- le répertoire `models` regroupe les modèles de l'application (y compris ceux permettant d'accéder à la base de données, et autres systèmes de persistance)
- le répertoire `views` regroupe les vues de l'application. À l'intérieur, les vues sont regroupées par contrôleurs et nommées selon l'action qu'elle représente. Ainsi, la vue associée à l'action `login` du contrôleur `utilisateur` est representée sur le système de fichier par `/app/views/utilisateur/login.phtml`.
- le répertoire `layout` contient le squelette HTML du site.
- le fichier `AppBootstrap.php` est le Bootstrap de l'application lui permettant d'automatiser un certain nombre d'actions à son lancement.

2.3 Écrire un Controller

Tout d'abord, la norme pour les fichiers de contrôleurs veut que ceux-ci se trouvent dans le répertoire `/app/controllers` et que leur nom soit de la forme `nomController` (où `Controller` est commun à tout contrôleur). Attention à ce que la casse (majuscule/-minuscule) corresponde entre le nom de fichier, le paramètre `$_GET['c']` de l'URL et le nom de classe.

Le code minimal d'un contrôleur est très simple :

```
<?php
class indexController extends Controller {
  public function indexAction() {}
}
```

Il y a deux trois choses à noter ici.

Le nom de la classe est ici le nom que l'on passera par l'URL (à l'aide de la variable `$_GET['c']`).

De plus, notre contrôleur hérite de la classe `Controller`. Cela induit quelques petits éléments, notamment l'accès à la `View` à travers l'attribut `$this->view`. Nous verrons dans la partie suivante les actions que l'on peut lui appliquer. Ensuite, les méthodes `firstAction()` et `lastAction()` peuvent être redéfinies. Ce sont des actions qui vont se lancer avant et après chacune des actions que vous allez écrire par la suite pour ce contrôleur particulier.

Enfin la méthode `indexAction` correspond à l'action `index` (l'action par défaut pour tout contrôleur). Vous aurez donc à écrire d'autres actions. Celles-ci se lanceront en fonction du paramètre `$_GET['a']` passé dans l'URL. Vous pourrez manipuler la vue à l'intérieur de ces méthodes. Par exemple, si votre méthode correspond à l'accueil de votre site, vous pouvez ajouter un titre comme ceci :

```
public function indexAction() {
  $this->view->appendTitle('Accueil - ');
  // ou
  $this->view->prependTitle(' - Accueil');
}
```

La syntaxe sera expliquée plus loin mais est tout de même simple à comprendre.

Ainsi, dans ces méthodes, vous pouvez aussi manipuler les `Models` qui permettront d'accéder à la base de données et les transmettre à la vue pour que celle-ci les affiche.

2.4 Écrire une View

Il est important de comprendre que les Views s'utilisent de deux manières complémentaires. Tout d'abord à l'intérieur des Controllers, puis ensuite dans les fichiers de type .phtml qui décrivent le code HTML.

Dans un contrôleur, nous l'avons vu, la vue est accessible à l'aide de l'attribut `$this->view`. On peut lui ajouter des variables très facilement de cette manière : `$this->view->maChaine = 'ma super chaine';`
Ici il s'agit d'une chaîne, mais bien sûr il peut s'agir de tout type de variable, et le plus souvent il s'agira d'objets contenant les données à manipuler, issues de la base de données. De cette manière vous pouvez transmettre facilement des données à la vue.

À noter qu'il existe plusieurs méthodes applicables sur une `View` qui permettent notamment d'ajouter des fichiers CSS et des scripts javascript. Le détail est donné plus loin.

Ensuite, l'affichage des vues se fait à l'aide de code HTML. À noter qu'une vue correspond à une action donnée, correspondant elle-même à un contrôleur donné. Ainsi le nommage se fera de cette manière :
`/app/views/controller/action.phtml` où **controller** correspond au nom du contrôleur concerné, et **action** à l'action concernée.
Le contenu de ces fichiers sera du HTML mais vous pouvez tout de même y utiliser du PHP, notamment pour manipuler les données. À noter que lorsque vous travaillez dans ces fichiers vous êtes **dans** la vue, c'est-à-dire que les variables qui étaient accessibles tout à l'heure à travers `$this->view->variable` dans le contrôleur, seront accessibles directement : `$this->variable`. Ainsi un contenu possible, en reprenant l'exemple de la chaîne de tout à l'heure, sera :

```
<div id="header"><?php echo $this->title; ?></div>
<p><?php echo $this->maChaine; ?></p>
```

qui affichera deux blocs affichant d'abord le titre de l'application, puis le paragraphe "*ma super chaîne*".

2.5 Écrire un Model

Le modèle dit "de base" est le coeur de l'application. C'est lui qui représente les éléments "métier". Ainsi le modèle pourra représenter un utilisateur, un article de blog, une photo d'une galerie. On peut appliquer à chacun de ces modèles des actions : par exemple, l'utilisateur possédera une méthode `login()` qui lui permettra de se connecter.
Tout modèle doit étendre la classe `Model`.

```
class User extends Model {
  private $username;
  private $mail;

  public function __construct($username, $mail) {
    $this->username = $username;
```

```
    $this->mail = $mail;
  }

  public function login($username, $password) {
    // dans un cas reel, on ira verifier les infos en base de
        donnees
    if($username=='john' && $password=='doe') {
      // attribue $username a une variable de session
      Session::_param('username', $username);
    }
  }

  public function isLogged() {
    // permet de recuperer la variable de session 'username'
    $username = Session::param('username');

    return !empty($username);
  }
}
```

Le modèle de base est très utile, mais bien souvent insuffisant, surtout dans une application web qui stocke des données. En effet, on aura bien souvent besoin de sauvegarder une liste d'utilisateurs en base de données (par exemple). Ainsi, le modèle DAO va permettre d'accéder à ces informations. À ce jour, 3 modes d'accès à des supports de stockage sont disponibles dans MINZ : les fichiers textes, les tableaux PHP (stockés dans des fichiers texte) et les bases de données MySQL (utilisant PDO)

```
// La connexion a la BDD se fait automatiquement grace aux
   infos donnees dans le fichier de configuration
// a inclure avec include_once(APP_PATH.'/models/dao/UserDAO.
   sql.php'); dans User.php
class UserDAO extends Model_sql {
  public function searchByUsername($username) {
    $sql = 'SELECT * FROM user WHERE loginUser=?';

    $stm = $this->bd->prepare($sql);
    $stm->execute(array($username));
    $res = $stm->fetchAll(PDO::FETCH_CLASS);

    if(isset($res[0])) {
      return $res[0];%$
    } else {
      return false;
    }
  }
}
```

Notre problème ici est que nous renvoyons une classe "non connue" par l'application et tirée de la base de données. Une bonne pratique est de "mapper" cet objet "BDD" avec un modèle déjà écrit afin que le modèle DAO ne renvoie que des modèles "User", codé plus haut. Pour cela nous allons utiliser un Helper :

```
class HelperUser {
  // permet de convertir un Model DAO (tire de BDD ici) en
     Model
  // on peut aussi creer une methode convertissant des listes
      de Model DAO
  public static function daoToUser($dao) {
    return new User($dao->loginUser, $dao->mailUser);
  }
}
```

Ainsi, nous utiliserons cette méthode dans la classe UserDAO en remplaçant

```
return $res[0];
```

par

```
return HelperUser::daoToUser($res[0]);
```

En pensant bien à inclure le fichier de Helper dans UserDAO

```
include_once(APP_PATH.'/models/helper/HelperUser.php');
```

2.6 Lier le tout

Maintenant que nous avons vu chacun des modules important, il est tant d'articuler tout ça autour d'un contrôleur concret. Nous allons donc réutiliser les modèles écrits jusque-là. Cet exemple permet de rechercher dans une base de données un utilisateur, à partir d'un username passé via l'url.

Le contrôleur

```
include(APP_PATH.'/models/User.php');

class indexController extends Controller {
  public function indexAction() {
    // recupere le username dans l'url (variable $_GET['
       username'])
    // on met $username = 'john' par defaut
    $username = Helper::fetch_get('username', 'john');

    // recherche en BD de l'utilisateur
    $userDAO = new UserDAO();
    // si non trouve, retourne false
    $this->view->user = $userDAO->searchByUsername($username)
       ;

    // On change quelques infos sur la page
    $this->view->appendTitle("Recherche d'un utilisateur - ")
       ;
    // url permet de gerer facilement les urls internes a l'
       application MINZ
```

```
    // son utilisation sera decrite plus loin
    $this->view->prependStyle($this->view->url->display().'/
       theme/defaut.css');
  }
}
```

La vue

```
<!DOCTYPE html>
<html>
  <head>
    <?php echo $this->headTitle(); ?>
    <?php echo $this->headStyle(); ?>
  </head>
  <body>
    <h1><a href="<?php echo $this->url->display();?>"><?php
       echo $this->title; ?></a></h1>

    <?php if(!$this->user) { ?>
    <p class="error">L'utilisateur recherche n'existe pas.</p
       >
    <?php } else { ?>
    <p class="infos">L'utilisateur <?php echo $this->user->
       username(); ?> possede l'adresse mail <?php echo $this
       ->user->mail(); ?>.</p>%$
    <?php } ?>
  </body>
</html>
```

2.7 Utiliser l'AppBootstrap

Très vite, notre application va grossir, on va devoir ajouter des actions au contrôleur, des contrôleurs à l'application, etc. Et un problème va vite se faire sentir : qu'en est-il de la duplication de code ? Par exemple, la feuille de style que l'on doit ajouter dans chaque contrôleur ne pourrait-elle pas être ajoutée une bonne fois pour toute ?

C'est là qu'intervient l'`AppBootstrap`. Ce mécanisme permet d'automatiser certaines tâches répétitives. Pour se faire, rien de plus simple : il suffit de créer le fichier /app/AppBootstrap.php et d'y ajouter le code suivant (obligatoire)

```
class AppBootstrap {
  private $view;

  public function __construct() {
    $controller = Controller::getInstance();
    $this->view = $controller->getView();
  }

  // Fonction lancee par le Boostrap de la librairie et
     permet de charger les elements redondants dans chaque
     Controller.
```

```
  public function run() {}
}
```

Il ne vous reste plus alors qu'à ajouter ce qu'il vous faut dans la méthode `run()`. Notez que tout ce qu'il est possible de faire dans un contrôleur, est possible dans l'AppBootstrap. Il vaut mieux cependant s'en tenir au strict minimum. Par exemple :

```
public function run() {
  include(APP_PATH.'/models/User.php');
  $this->view->prependStyle($this->view->url->display().'/
     theme/defaut.css');
}
```

2.8 Router les URLs

Ok, nous avons une application qui tourne à peu près correctement. Malheureusement, comme nous l'avons vu, le contrôleur et l'action à exécuter passe via les paramètres renseignés dans l'url. Du coup celle-ci n'est pas très jolie :(Par exemple, l'url pour accéder à un article de blog pourrait être de la forme http://votre-domaine.fr/?c=blog&a=voir&id=42

Heureusement, l'url rewriting est là pour vous ! Ici, nous n'allons pas vraiment voir ce que nous propose Apache car ce n'est pas le but (et je n'ai jamais trouvé ça facile !). Aussi, la seule chose que vous avez à faire dans votre fichier `.htaccess` est d'ajouter ces lignes :

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

Cela permet tout simplement de rediriger toutes les requêtes tombant sur votre site sur le fichier index.php, laissant la main ainsi au framework. Assurez-vous maintenant que la variable `use_url_rewriting` dans le fichier de configuration soit bien à `true`.

Le principe de l'url rewriting dans Minz repose sur un tableau PHP associant 3 ou 4 éléments :

- la `route` qui correspond à l'url amenant à la page demandée
- le contrôleur associé à cette route
- l'action associée elle aussi à cette route
- et un tableau de paramètres supplémentaires pour pouvoir utiliser des urls dynamiques

Aussi, vu qu'un exemple est plus parlant que du blabla, voici le contenu d'un fichier /app/configuration/routes.php

```
return array(
  // blog
  array(
    'route'      => '/blog',
    'controller' => 'blog',
    'action'     => 'index'
  ),
  array(
```

```
    'route'        => '/blog/tag/*',
    'controller'   => 'blog',
    'action'       => 'search',
    'params'       => array('tag')
  ),
  array(
    'route'        => '/blog/*/*',
    'controller'   => 'blog',
    'action'       => 'voir',
    'params'       => array('id','title')
  ),
  array(
    'route'        => '/*',
    'controller'   => 'index',
    'action'       => 'index',
    'params'       => array('page')
    ),
);
```

Ici, nous renvoyons un tableau contenant l'ensemble des règles de réécriture. Chaque règle est définie elle-même dans un tableau. Afin de définir une url dynamique, nous utilisons l'astérisque (*). Attention, celle-ci doit obligatoirement se trouver **seule** derrière un slash (/). Afin de définir le nom de la variable `$_GET[]`, on l'indique dans le tableau associé à `params`.

Une petite remarque importante : comme j'ai voulu l'illustrer avec la dernière règle, celle-ci pourrait être confondue avec la première. En effet, "blog" pourrait être mappé avec l'astérisque et être envoyé en tant que valeur de la variable `$_GET['page']`. Pour pallier à ce problème, il faut savoir que Minz s'arrête à la première règle rencontrée correspondant à l'url actuelle. Dans mon exemple, il n'y a donc pas de problème, mais en inversant les deux règles, on ne pourrait plus accéder au contrôleur "Blog" et son action "index". Attention donc à l'ordre de vos règles et faire en sorte qu'elles ne s'écrasent pas les unes les autres.

On pourra imaginer un certain nombre d'améliorations possibles au niveau du routage des urls. Le routage dynamique est limité par la séparation obligatoire des variables entre des slashs (/), j'avais commencé à penser à les définir, à la place des astérisques, par des accolades et l'on aurait pu avoir quelque chose de la forme :

```
array(
  'route'        => '/blog/{id}-{title}.html',
  'controller'   => 'blog',
  'action'       => 'voir'
)
```

Mais ça demande pas mal de changements, et je n'ai jamais pris le temps de le faire. De plus, le problème de l'ordre des règles peux poser quelques problèmes.

3 Aller plus loin

3.1 Utilisation d'un layout

// structure du site
// layout global + partial

3.2 Les variables de session

// gérées par la classe Session
// history et language = cas particuliers

3.3 Historisez le parcours de vos utilisateurs

// classe History enregistre à chaque page le parcours de l'utilisateur
// possibilité de revenir aux pages précédentes grâce à la classe Url
// possibilité d'annuler cet enregistrement

3.4 Utilisation du système de Cache

// Met en cache les pages visitées
// Possibilité d'annuler la mise en cache
// Attention si vous êtes loggué, la mise en cache enregistrera des choses qui ne sont pas forcément accessibles à tout le monde
// Améliorations possibles

3.5 Internationalisez votre application

// Structure
// Utilisation simple de la classe Translate

3.6 Écrire sa propre librairie

// Fonctionnement

3.7 La View en détails

// Explication des différentes méthodes

3.8 Sécuriser son application MINZ

// Le fichier de configuration
// Les droits sur les fichiers

Licence