Zend Framework 2
Getting started

Traduction de Getting started http://framework.zend.com/learn/

Préambule

Ce tuto est issue de la volonté à traduire le guide Zend Framework 2 proposé à l'adresse suivante: http://framework.zend.com/learn/

Les titres des chapitres n'ont volontairement pas été traduits pour permettre de repérer les étapes si l'on suit en parallèle le guide original.

Pour l'installation du 'ZendSkeletonApplication', nous utilisons la deuxième méthode, celle utilisant le zip de 'github'.

Les chemins utilisés ici, pour l'installations du Zend Framework 2 et du Virtual host, peuvent différés selon le poste de travail utilisé. A vérifier !

Pour les suggestions, corrections et autres: manuelRebe@orange.fr

Sommaire

Getting started

Notre 'QuickStart' vous informera en un tour rapide sur quelques composants de Zend Framework et démontrera comment une ZF2 (Zend Fraùework 2) application peut être construite depuis le tout début.

Getting started with Zend Framework 2

Ce tutorrial est défini de manière à donner une introduction sur l'utilisation de Zend Framework 2 en créant une simple application pilotant une base de données et utilisant le modèle 'Model-View-Controller'. A la fin vous aurai construit une application ZF2 et vous pourrez alors analyser le code pour connaître bien plus sur le fonctionnement de l'application ZF2.

Some assumptions

Ce tutorrial requière que vous utilisiez PHP 5.3.3 avec le server web Apache et MySQL, accessible via l'extension PDO.
Votre installation Apache doit avoir l'extension 'mod_rewrite' installée et configurée.

Vous devez aussi vous assurer que Apache est configuré pour supporter les fichiers '.htaccess'. Ceci est habituellement fait en changeant la définition 'AllowOverride None' en 'AllowOverride FileInfo' dans le fichier 'httpd.conf'. Lisez votre documentation pour obtenir plus de détails à ce sujet.

Vous ne pourrez que naviguer sur la page 'home' de ce tutoriel si vous ne configurez pas correctement l'extension 'mod_rewrite' pour un usage correct du fichier '.htaccess'.

The tutorial application

L'application que nous allons développer est un simple système d'inventaire pour afficher quelques-uns de nos albums de musique. La page principale listera notre collection d'albums et nous permettra d'ajouter, de modifier, et de supprimmer des albums.

Nous avons besoins de 4 pages pour notre site web:

- la page qui liste et affiche les albums. Cette page affichera des liens pour ajouter, modifier et supprimer les albums de la liste ;
- la page d'ajout des albums. Cette page affiche un simple formulaire d'ajout d'albums ;
- La page d'édition permettant de modifier les albums ;
- la page de suppression des albums. Cette page demandera une confirmation de suppression ;

Nous aurons besoin aussi de stocker les données dans une base de données. Une simple table sera nécessaire avec les champs suivants:

- nom:id, type:integer, null:no, primary key auto-increment ;
- nom:artist, type:varchar(100), null:no ;
- nom:title, type:varchar(100), null:no ;

Getting started: A skeleton application

Dans l'ordre, pour construire notre application, nous commencerons avec le 'ZendSkeletonApplication' disponible sur 'github' (https://github.com/zendframework/ZendSkeletonApplication).

Pour installer le 'ZendSkeletonApplication', cliquer sur le bouton 'zip' depuis cette l'adresse 'github' https://github.com/zendframework/ZendSkeletonApplication. Ceci téléchargera un fichier de nom 'ZendSkeletonApplication-master.zip'. Dézipper ce fichier dans le dossier où vous développez vos sites habituellement (wamp/www), et renommez-le 'zf2-tutorial'.

'ZendSkeletonApplication' est défini pour utiliser 'Composer' pour résoudre les dépendances. Dans ce cas, la dépendance est 'Zend Framework 2' lui-même.

Pour installer 'Zend Framework 2' pour notre application nous saisissons 'simplement' en ligne de commande depuis le dossier 'zf2-tutorial':

```
php composer.phar self-update
php composer.phar install
```

L'installation peut prendre quelque moment. Une fois le processus terminé vous devriez voir ceci:

```
Installing dependencies from lock file
- Installing zendframework/zendframework (dev-master)
Cloning 18c8e223f070deb07c17543ed938b54542aa0ed8

Generating autoload files
```

Nous pouvonc maintenant définir un 'hôte virtuel'.

Notes:

Pour que les lignes de commandes fonctionnent, il est nécessaire de définir une variable d'environnement. Aussi, il faut saisir ces lignes de commandes en ayant pris soin de s'être placé, en ligne de commande aussi, dans le dossier 'zf2-tutorial'.

Procéder ainsi:

Depuis le 'panneau de configuration' double-cliquer sur 'système' puis cliquer sur 'paramètres système avancés'. De là cliquer sur 'variables d'environnement ...' . Un tableau vous sera alors présenté. Dans ce tableau double-cliquer sur 'path' et en fin de ligne 'valeur de la variable' coller ce chemin: ;C:\wamp\bin\php\php5.4.3 (avec le point virgule !)
Cliquer sur ok. Un redémarrage de l'ordinateur est conseillé. (Ce chemin peut être différent selon votre poste de travail, veillez à vérifer).

Une fois redémarré, pour se placer dans le dossier 'zf2-tutorial', en ligne de commande saisissez ceci :

```
cd\
cd wamp/www/zf2-tutorial
php composer.phar self-update
php composer.phar install
```

Virtual host

Vous avez maintenant besoin de créer un hôte virtuel Apache pour l'application et éditer votre fichier 'hosts' tel que l'adresse 'http://zf2-tutorial.localhost' servira pour accéder au fichier 'index' du dossier zf2-tutorial/public.

Définir le 'hôte virtuel' est habituellement fait dans le fichier 'httpd.conf' ou le fichier 'extra/httpd-vhosts.conf'. Si vous utilisez 'extra/httpd-vhosts.conf', assurez-vous que ce fichier est inclus dans le fichier 'httpd.conf' (décommenter la ligne 'include conf/extra/httpd-vhosts.conf ').

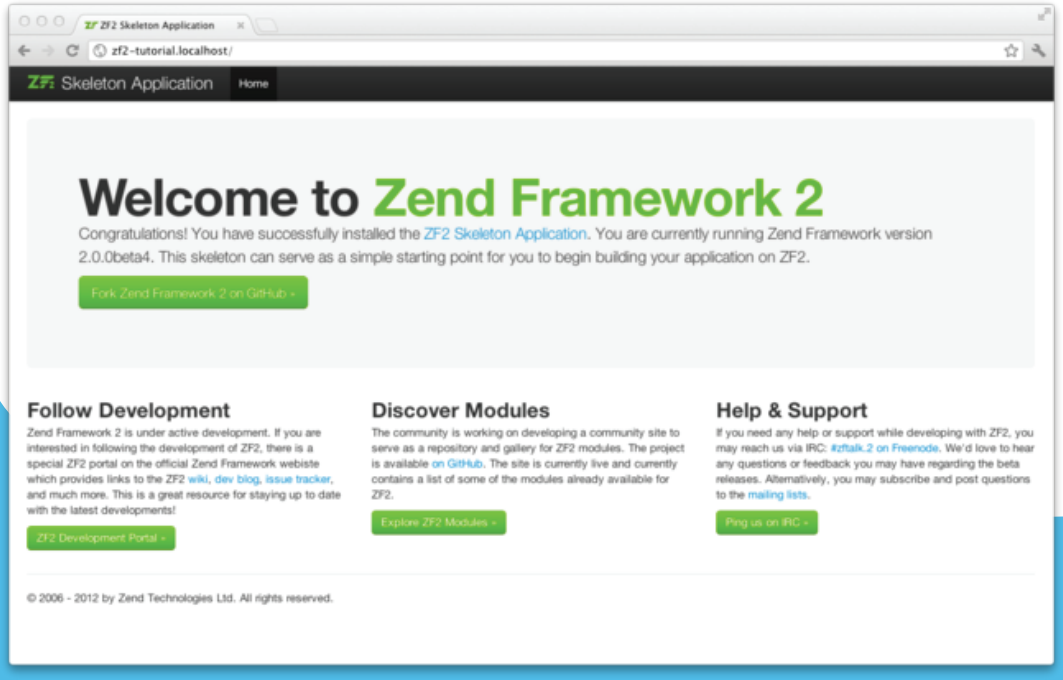
Dans le fichier 'conf/extra/httpd-vhosts.conf' assurez-vous que 'NameVirtualHost' est défini et configuré sur *:80. Puis saisir dans le fichier 'conf/extra/httpd-vhosts.conf' les lignes suivantes:

```
<VirtualHost *:80>
        ServerName zf2-tutorial.localhost
        DocumentRoot /path/to/zf2-tutorial/public
        SetEnv APPLICATION_ENV "development"
        <Directory /path/to/zf2-tutorial/public>
            DirectoryIndex index.php
            AllowOverride All
            Order allow,deny
            Allow from all
        </Directory>
</VirtualHost>
```

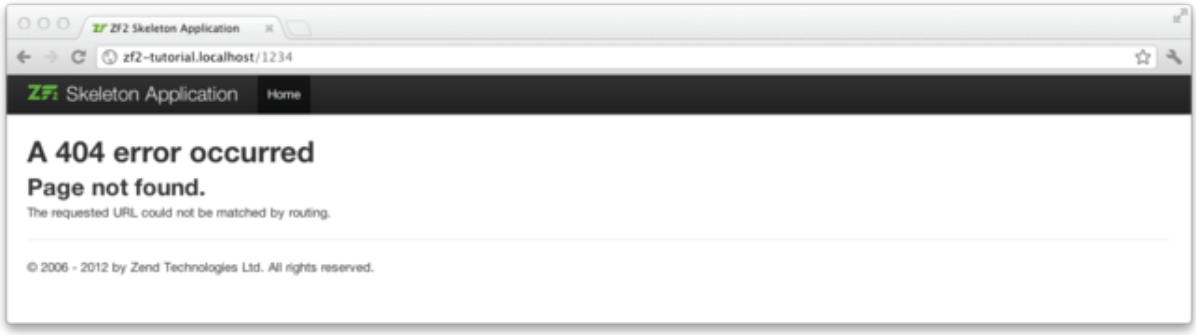
Vérifier que dans le fichier 'c:\windows\system32\drivers\etc\hosts' le nom 'zf2-tutorial.localhost' est mappé sur 127.0.0.1 comme ainsi:

```
127.0.0.1       zf2-tutorial.localhost localhost
```

Le site peut alors être appelé à l'adresse suivante: http://zf2-tutorial.localhost



Pour tester le fichier '.htaccess', tapez l'adresse 'http://zf2-tutorial.localhost/1234' et vous devriez voir ceci:



Si vous voyez une page 404 standard Apache, alors vous avez besoin de redéfinir le fichier '.htaccess' avant de continuer. Si vous utilisez IIS avec le Url Rewrite Module, ajoutez les lignes suivantes dans le fichier '.htaccess':

```
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^.*$ index.php [NC,L]
```

Vous avez maintenant une application squelette fonctionnelle et nous pouvons commencer à ajouter les spécificités pour notre application.

Error reporting

Optionnellement vous pouvez utiliser le 'APPLICATION_ENV' défini dans votre 'hôte virtuel' pour laisser PHP sortir tous les erreurs sur le navigateur. Ceci peut être utlisé durant le développement de votre application.

Editez le fichier 'index.php' du dossier 'zf2-tutorial/public/' et changez ce qui suit:

```
<?php

/**
 * Display all errors when APPLICATION_ENV is development.
 */
if ($_SERVER['APPLICATION_ENV'] == 'development') {
    error_reporting(E_ALL);
    ini_set("display_errors", 1);
}

/**
 * This makes our life easier when dealing with paths. Everything is relative
 * to the application root now.
 */
chdir(dirname(__DIR__));

// Setup autoloading
require 'init_autoloader.php';

// Run the application!
Zend\Mvc\Application::init(require 'config/application.config.php')->run();
```

Modules

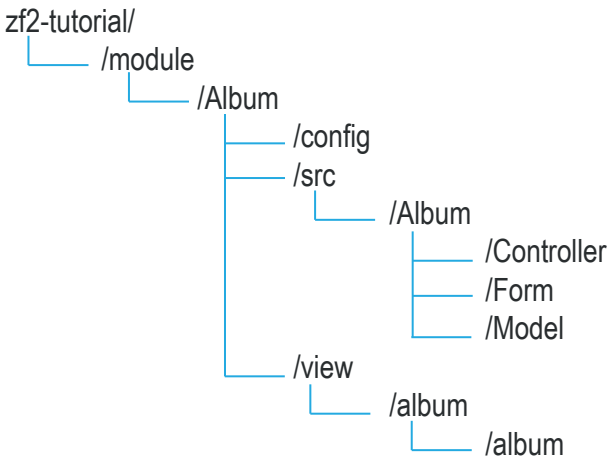
Zend Framework 2 utilise un système de modules, vous organisez votre code d'application spécifique dans chaque module. Le module Application fourni par le skeleton est utilisé pour fournir l'amorçage, le renvoie d'erreurs et la procédure de configuration à l'ensemble de l'application. Il est courant d'utiliser les 'controllers' de niveau de l'application pour définir la page d'accueil d'une application, mais nous n'allons pas utiliser la page d'accueil fournie par défaut dans ce tutorial comme nous le voulons pour notre liste d'albums, dans nos modules personnels, chaque page d'accueil sera intégrée dans le module lui correspondant.

Nous allons mettre tout nos codes à l'intérieur du module 'Album' qui contiendra nos 'controllers', 'models', 'forms' et 'view', ceci en même temps que la configuration. Nous allons aussi appliquer au module 'Application' les modifications nécessaires.

Commençons à définir l'arborescence de dossiers nécessaires.

Setting up the Album module

Commencer par créer un dossier nommé 'Album' dans le dossier 'module' de l'application principale. Dans ce dossier, définir l'arborescence de dossiers suivante:



Comme vous pouvez le voir, le module 'Album' est divisé en différents dossiers qui contiendrons les différents types de fichiers que nous aurons besoin. Les fichiers PHP qui contiennent les classes dans l'espace de nom 'Album' sont dans le dossier src/Album de manière que vous pouvez avoir de multiples espaces de noms dans notre module nous étant nécessaire. L'arborescence montre aussi un sous-dossier appelé 'album' pour nos scriptes de 'view' (vue) de notre module.

Pour le chargement et la configuration, Zend Framework 2 a un 'ModuleManager'. Celui-ci cherchera le fichier 'Module.php' à la racine du module (module/Album) et examinera ce fichier pour y trouver une classe appelée 'Album\Module'. Les classes dans un module donné aurons pour espace de nom le nom du module, qui est dans le répertoire du module.

Créons donc le fichier 'Module.php' dans le module 'Album' (zf2-tutorial/module/Album) et y saisir le code suivant:

```
<?php
namespace Album;

class Module
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

Le 'ModuleManager' appellera 'getAutoloaderConfig()' et 'getConfig()' atuomatiquement pour nous.

Autoloading files

Notre méthode 'getAutoloaderConfig()' retourne un tableau qui est compatible avec 'AutoloaderFactory' de ZF2. Nous la configurons de manière que nous ajoutons à la classe 'ClassmapAutoloader ' un chemin vers un fichier et ajoutons aussi l'espace de nom du module à la classe 'StandardAutoloader'. Ceci est conforme à PSR-0 et tel que les classes correspondent aux fichiers respectant les règles de PSR-0.

Comme nous sommes dans le cadre d'un développement, nous n'aurons pas besoin de charger les fichiers via la 'classmap', si bien que nous fournissons un tableau vide pour l'autochargement de la 'classmap'. Créer un fichier appelé 'autoload_classmap.php' à l'intérieur du dossier 'zf2-tutorial/module/Album':

```
<?php
return array();
```

Comme ceci est un tableau vide, à chaque fois que l'autoloader cherchera une classe dans l'espace de nom 'Album', il se redirigera vers 'StandardAutoloader ' pour nous.

Configuration

Ayant enregistré l'autoloader, regardons rapidement la méthode 'getConfig()' dans 'Album\Module'. Cette simple méthode charge le fichier 'config/module.config.php'.

Créer un fichier nommé 'module.config.php' au sein du dossier 'zf2-tutorial/module/Album/config':

```
<?php
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),
    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

Les informations de configurations sont passées aux composants relevants du service 'ServiceManager'. Nous avons besoins d'initialiser deux sections: 'controllers' et 'view_manager'. La section des 'controllers' fournie une liste de tous les 'controllers' fournis par le module. Nous aurons besoin d'un 'controller', 'AlbumController', que nous référencerons comme 'Album\Controller\Album'. Le 'controller' clé doit être unique à travers tous les modules, si bien que nous le préfixons avec le nom de notre module.

A l'intérieur de la section 'view_manager ', nous ajoutons notre dossier 'view' à la configuration de 'Template-PathStack '. Ceci permettra de trouver le scripte de vue pour notre module 'Album' que nous avons stocké dans le répertoire 'view/ '.

Informing the application about our new module

Maintenant nous avons besoin de dire au 'ModuleManager' que le nouveau module 'Album' existe. Ceci est fait dans le fichier 'config/application.config.php' de l'application qui est fourni par le skeleton de l'application. Modifier ce fichier de façon à ce qu'il soit renseigné du module 'Album' comme ci-dessous:

(Les changements à faire sont indiqués par les commentaires.)

```
<?php
return array(
    'modules' => array(
        'Application',
        'Album',                // <-- Add this line
    ),
    'module_listener_options' => array(
        'config_glob_paths'    => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

Comme vous pouvez le voir, nous avons ajouté notre module 'Album' dans la liste des modules juste après le module 'Application' déjà présent.

Nous avons maintenant défini le module de façon à ce qu'il soit prêt pour être renseigné de nos codes personnels.

Routing and controllers

Nous construirons un système d'inventaire très simple pour afficher notre collection d'albums. La page d'accueil listera notre collection et nous permettra d'ajouter, de modifier et de supprimer des albums. Par conséquent les pages suivantes sont recquises:

Pages	Descriptions
Home	Cette page affichera la liste des albums et fournira des liens pour éditer et supprimer les albums. Aussi, un lien pour permettre d'ajouter un nouvel album sera fourni.
Add new album	Cette page fournira un formulaire pour ajouter un nouvel album.
Edit album	Cette page fournira un formulaire pour éditer un album.
Delete album	Cette page demandera confirmation que nous voulons bien supprimer un album et supprimera alors cet album.

Avant de définir nos fichiers, il est important de savoir comment le framework examine les pages pour être organisées. Chaque page de l'application est reconnue comme une action et les actions sont groupées dans les controllers à l'intérieur du modules. En conséquence, vous souhaiterez en général grouper les actions connexes dans un controller; par exemple, un controller d'informations pouvant avoir des actions, d'archive et de view.

Comme nous avons 4 pages qui s'appliquent toutes aux albums, nous les grouperons dans un simple controller 'AlbumController' à l'intérieur du module 'Album' comme 4 actions. Les 4 actions seront:

Pages	Descriptions	Actions
Home	AlbumController	index
Add new album	AlbumController	add
Edit album	AlbumController	edit
Delete album	AlbumController	delete

La carte d'un URL d'une action particulière est faite en utilisant les routes qui sont définies dans le module du fichier 'module.config.php'. Nous ajouterons une route pour nos actions album. Ceci est le fichier config du module mis à jour avec le nouveau code:

```
<?php
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),

    // The following section is new and should be added to your file
    'router' => array(
        'routes' => array(
            'album' => array(
                'type'    => 'segment',
                'options' => array(
                    'route'    => '/album[/][:action][/:id]',
                    'constraints' => array(
                        'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'id'     => '[0-9]+',
                    ),
                    'defaults' => array(
                        'controller' => 'Album\Controller\Album',
                        'action'     => 'index',
                    ),
                ),
            ),
        ),
    ),

    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

Le nom de la route est ‘album’ et a un type ‘segment’. La route ‘segment’ nous permet de spécifier les espaces réservés dans le morceau URL (route) qui sera considé en un paramètre nommé dans la route visible. Dans cette condition, la route est ‘/album[/:action][/:id]’ que verra tout URL qui commencera avec ‘/album’. Le prochain segment sera un nom d’action optionnel, et alors finalement le prochain segment sera considéré en un ‘id’ optionnel. Les crochets indiquent qu’un segment est optionnel. Les sections contraintes nous permettent d’assurer que les caractères dans un segment sont examiner, aussi nous avons limité les actions commençant avec une lettre, alors seulement les caractères séquencés sont numériques, undescore ou traits d’unions. Nous limitons aussi le ‘id’ à un nombre.

Cette route nous permet d’avoir les URL suivants:

URL	Page	Actions
/album	Accueil (liste des albums)	index
/album/add	Ajoute un nouvel album	add
/album/edit/2	Edite avec l‘id 2	edit
/album/delete/4	Supprime avec l’id 4	delete

Create the controllers

Nous sommes maintenant prêt pour définir notre controller. Dans Zend Framework 2, le controller est une classe qui est généralement appelé {Controller name}Controller. Noté que {Controller name} doit commencer par une lettre en majuscule. Cette classe réside dans un fichier appelé {Controller name}Controller.php à l'intérieur du répertoire du 'Controller' appartenant au module. Dans notre cas c'est 'module/Album/src/Album/Controller'. Chaque action est une méthode public dans la classe du controller qui est nommée {action name}Action. Dans ce cas, {action name} doit commencer avec une lettre minuscule.

Créons notre classe controller 'AlbumController.php' dans 'zf2-tutorials/module/Album/src/Album/Controller':

```
<?php
namespace Album\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class AlbumController extends AbstractActionController
{
    public function indexAction()
    {
    }

    public function addAction()
    {
    }

    public function editAction()
    {
    }

    public function deleteAction()
    {
    }
}
```

Nous avons maintenant défini les 4 actions que nous voulons utiliser. Ils ne fonctionnement pas encore jusqu'à que nous définissions les views (vues). Les URL pour chaque action sont:

URL	Méthode appelée
http://zf2-tutorial.localhost/album	Album\Controller\AlbumController::indexAction
http://zf2-tutorial.localhost/album/add	Album\Controller\AlbumController::addAction
http://zf2-tutorial.localhost/album/edit	Album\Controller\AlbumController::editAction
http://zf2-tutorial.localhost/album/delete	Album\Controller\AlbumController::deleteAction

Maintenant nous avons un 'router' fonctionnel et les actions sont définies pour chaque page de notre application.

Il est temps de construire la vue (view) et la couche modèle.

Initialise the view scripts

Pour intégrer la vue (view) dans notre application entière nous avons besoin de créer quelques fichiers scriptes de vue. Ces fichiers seront éxécutés par 'DefaultViewStrategy' et passerons toutes les variables ou modèles de vue qui sont retournés depuis le méthode d'action du controller. Ces scriptes de vue sont stockés dans le répertoire de nos vues du module dont le répertoire est nommé après le controller. Créér ces 4 fichiers vides:

- module/Album/view/album/album/index.phtml
- module/Album/view/album/album/add.phtml
- module/Album/view/album/album/edit.phtml
- module/Album/view/album/album/delete.phtml

Nous pouvons commencer à remplir quelque-chose, en commençant par notre base de données et modèles.

Database and models

The database

Maintenant que vous avez le module ‘Album’ défini avec les méthodes d‘action du controller et les scripts de vues (view), il est temps de regarder la section ‘model’ de notre application. Rappelez-vous que le model est une partie qui communique avec le corps logique de l‘application (’les règles métiers’) et, dans notre cas, communique avec la base de données. Nous utiliserons la classe ‘Zend\Db\TableGateway\TableGateway ’ de Zend Framework qui est utilisée pour trouver, insérer, modifier et supprimer les lignes dans la base de données.

Nous allons utiliser MySQL, via le pilote PHP’s PDO, aussi créons une base de données appelée ‘zf2tutorial’, et démarrons cette étape pour créer la table album avec quelques données à l’intérieur.

```
CREATE TABLE album (
  id int(11) NOT NULL auto_increment,
  artist varchar(100) NOT NULL,
  title varchar(100) NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO album (artist, title)
    VALUES ('The Military Wives', 'In My Dreams');
INSERT INTO album (artist, title)
    VALUES ('Adele', '21');
INSERT INTO album (artist, title)
    VALUES ('Bruce Springsteen', 'Wrecking Ball (Deluxe)');
INSERT INTO album (artist, title)
    VALUES ('Lana Del Rey', 'Born To Die');
INSERT INTO album (artist, title)
    VALUES ('Gotye', 'Making Mirrors');
```

Nous avons maintenant quelques données dans la base de données et pouvons écrire un très model simple pour celle-ci .

The model files

Zend Framework ne fourni pas un composant 'Zend\Model ' parce que le model est votre affaire et c'est à vous de décider comment vous voulez qu'il travail. Il y a beaucoup de composant que vous pouvez utiliser qui dépend de vos besoins. Une approche est d'avoir des classes 'model' réprésentant chaque entité dans votre application et alors vous identifiez les objets qui charge et sauvegarde les entités dans la base de données. Une autre façon est d'utiliser une technologie Object-relationnel, comme Doctrine ou Propel.

Pour ce tutorial, nous allons créer un model très simple en créant une classe AlbumTable qui utilise la classe Zend\Db\TableGateway\TableGateway dans laquel chaque objet album est un objet Album (connu comme une entité). Ceci est une implémentation du modèle de conception 'Table Data Gateway' à autoriser pour interfacer avec les données de la table de la base de données. Nous sommes conscient que le modèle 'Table Data Gateway' peut être limiter dans de larges sytèmes. On peut aussi être tenté d'ajouter un code d'accès à la base de données dans les méthodes du controller d'action comme vu par Zend\Db\TableGateway\AbstractTableGateway. Ne jamais faire cela !

Commençons par créer un fichier appelé 'Album.php' situé dans module/Album/src/Album/Model:

```
<?php
namespace Album\Model;

class Album
{
    public $id;
    public $artist;
    public $title;

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id'])) ? $data['id'] : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title'])) ? $data['title'] : null;
    }
}
```

Notre objet entité 'Album' est une simple classe PHP. Afin de travailler avec la classe TableGateway de Zend\Db, nous avons besoin d'implémenter la méthode exchangeArray(). Cette simple méthode copie les données depuis le passage dans un tableau des propriétés de votre entité.
Nous ajouterons plus tard un filtre d'entrées pour utiliser notre formulaire.

Ensuite, nous créons notre fichier 'AlbumTable.php' situé dans module/Album/src/Album/Model et ressemblant à ce qui suit:

```php
<?php
namespace Album\Model;

use Zend\Db\TableGateway\TableGateway;

class AlbumTable
{
    protected $tableGateway;

    public function __construct(TableGateway $tableGateway)
    {
        $this->tableGateway = $tableGateway;
    }

    public function fetchAll()
    {
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }

    public function getAlbum($id)
    {
        $id  = (int) $id;
        $rowset = $this->tableGateway->select(array('id' => $id));
        $row = $rowset->current();
        if (!$row) {
            throw new \Exception("Could not find row $id");
        }
        return $row;
    }

    public function saveAlbum(Album $album)
    {
        $data = array(
            'artist' => $album->artist,
            'title'  => $album->title,
        );

        $id = (int)$album->id;
        if ($id == 0) {
            $this->tableGateway->insert($data);
        } else {
            if ($this->getAlbum($id)) {
                $this->tableGateway->update($data, array('id' => $id));
            } else {
                throw new \Exception('Form id does not exist');
            }
        }
    }

    public function deleteAlbum($id)
    {
        $this->tableGateway->delete(array('id' => $id));
    }
}
```

Il y a beaucoup de fait ici. Premièrement, nous définissons la propriétés protégées '$tableGateway' à l'instance 'TableGateway' passé au constructeur. Nous utiliserons ceci pour améliorer les opérations sur la table de base de données de nos albums.

Nous créons alors quelques méthodes 'helper' que notre application utilisera pour communiquer avec la table. 'fetchAll()' récupère toutes les lignes 'albums' depuis la base de données comme un 'ResultSet', 'getAlbum()' récupère une simple ligne comme un objet 'Album', 'saveAlbum()' créé une nouvelle ligne dans la base de données ou modifie une ligne qui existe déjà, et 'deleteAlbum()' supprime la ligne complète. Le code pour chacune de ces méthodes est, je l'espère, explicite.

Using ServiceManager to configure the table gateway and inject into the AlbumTable

Afin de toujours utiliser les quelques instances de notre AlbumTable, nous utiliserons le ServiceManager pour définir comment en créer une. Ceci est plus facile à faire dans la classe 'Module' où nous créons une méthode appelée 'getServiceConfig()' qui est automatiquement appelée par le 'ModuleManager' et appliquée au ServiceManager. Nous serons alors capable de la recupérer dans notre controler quand nous en aurons besoin.

Pour configurer le ServiceManager, nous pouvons aussi fournir le nom de la classe à être instanciée ou une 'fabrique' qui instancie l'objet quand le ServiceManager en à besoin. Nous commençons par implémenter getServiceConfig() pour fournir une 'fabrique' qui créée un 'AlbumTable'. Ajouter cette méthode en bas du fichier 'Module.php' situé dans module/Album.

```
<?php
namespace Album;

// Add these import statements:
use Album\Model\Album;
use Album\Model\AlbumTable;
use Zend\Db\ResultSet\ResultSet;
use Zend\Db\TableGateway\TableGateway;

class Module
{
    // getAutoloaderConfig() and getConfig() methods here

    // Add this method:
    public function getServiceConfig()
    {
        return array(
            'factories' => array(
                'Album\Model\AlbumTable' =>  function($sm) {
                    $tableGateway = $sm->get('AlbumTableGateway');
                    $table = new AlbumTable($tableGateway);
                    return $table;
                },
                'AlbumTableGateway' => function ($sm) {
                    $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
                    $resultSetPrototype = new ResultSet();
                    $resultSetPrototype->setArrayObjectPrototype(new Album());
                    return new TableGateway('album', $dbAdapter, null, $resultSetPrototype);
                },
            ),
        );
    }
}
```

Cette méthode retourne un tableau de 'fabriques' qui sont toutes fusionnées par le ModuleManager avant de passer au ServiceManager. La 'fabrique' pour 'Album\Model\AlbumTable' utilise le ServiceManager pour créer un AlbumTableGateway à passer à AlbumTable. Nous disons aussi au ServiceManager qu'un AlbumTableGateway est créé en obtenant un Zend\Db\Adapter\Adapter (aussi depuis le ServiceManager) et l'utilisant pour créer un objet TableGateway. Le TableGateway est sollicité pour utiliser un objet Album lorsqu'il créé une nouvelle ligne de résultat. La classe TableGateway utilise le concept de prototype pour la création de définition de résultats et d'entités. Cela signifie qu'au lieu d'instancier quand nécessaire, le système clone un objet déjà instancié. Voir PHP Constructor Best Practices et le Prototype Pattern pour plus de détails.

Finalement, nous aurons besoin de configurer le ServiceManager pour qu'il connaisse comment obtenir un Zend\Db\Adapter\Adapter. Cela est fait en utilisant une 'fabrique' appelée' Zend\Db\Adapter\AdapterServiceFactory que nous pouvons configurer dans le système de configuration. Le ModuleManager de Zend Framework 2 fusionne toutes les configurations depuis chaque fichier module.config.php de module et lors de la fusion dans les fichiers situés dans config/autoload. Nous ajouterons notre configuration de base de données au fichier 'global.php' qui vous imposera à votre système de controle. Vous pouvez utiliser 'local.php' (en dehors de VCS) pour stocker les identifications pour votre base de données si vous le voulez. Modifions 'config/autoload/global.php' (dans le chemin Zend Skelete, non pas à l'intérieur du module 'Album') avec le code suivant:

```
<?php
return array(
    'db' => array(
        'driver'         => 'Pdo',
        'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter'
                    => 'Zend\Db\Adapter\AdapterServiceFactory',
        ),
    ),
);
```

Il faudra mettre vos identifiants de base de données dans 'config/autoload/local.php' tel qu'ils ne sont pas dans le dépôt git (comme 'local.php' est ignoré).

```
<?php
return array(
    'db' => array(
        'username' => 'YOUR USERNAME HERE',
        'password' => 'YOUR PASSWORD HERE',
    ),
);
```

Back to the controller

Maintenant que le 'ServiceManager' peut créer une instance de 'Album Table' pour nous, nous pouvons ajouter une méthode au controller pour le récupérer. Ajouter 'getAlbumTable' à la classe 'AlbumController ':

```
// module/Album/src/Album/Controller/AlbumController.php:
    public function getAlbumTable()
    {
        if (!$this->albumTable) {
            $sm = $this->getServiceLocator();
            $this->albumTable = $sm->get('Album\Model\AlbumTable');
        }
        return $this->albumTable;
    }
```

En haut de la classe vous devrait aussi ajouter: `protected $albumTable;`

Nous pouvons maintenant appeler 'getAlbumTable()' depuis notre controller chaque fois que nous avons besoin d'intéragir avec notre modèle.

Si le sevice est configuré correctement dans le 'Module.php', alors nous devrions obtenir une instance de 'Album\Model\AlbumTable ' lors de l'appel de 'getAlbumTable()'.

Listing albums

Dans l'ordre de la liste des albums, nous avons besoin de récupérer les albums depuis le modèle et de les passer à la view. Pour faire ceci, nous remplissons 'indexAction()' dans 'AlbumController'. Modifions la méthode 'indexAction()' de 'AlbumController' pour cela ressemble à cela:

```
// module/Album/src/Album/Controller/AlbumController.php:
// ...
    public function indexAction()
    {
        return new ViewModel(array(
            'albums' => $this->getAlbumTable()->fetchAll(),
        ));
    }
// ...
```

Avec Zend Framework 2, afin de définir des variables dans la vue, nous retournons depuis l'action contenant les données que nous avons besoin une instance de 'ViewModel' où le premier paramètre du constructeur est un tableau. Ces données sont alors automatiquement passées au scripte de vue (view). L'objet 'ViewModel' nous permet aussi de changer le script de vue qui est utilisé, mais par défaut c'est {controller name}/{action name}. Nous pouvons maintenant remplir le scripte de vue 'index.phtml'.

```
<?php
// module/Album/view/album/album/index.phtml:

$title = 'My albums';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>
<p>
    <a href="<?php echo $this->url('album', array('action'=>'add'));?>">Add new album</a>
</p>

<table class="table">
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th> </th>
</tr>
<?php foreach ($albums as $album) : ?>
<tr>
    <td><?php echo $this->escapeHtml($album->title);?></td>
    <td><?php echo $this->escapeHtml($album->artist);?></td>
    <td>
        <a href="<?php echo $this->url('album',
            array('action'=>'edit', 'id' => $album->id));?>">Edit</a>
        <a href="<?php echo $this->url('album',
            array('action'=>'delete', 'id' => $album->id));?>">Delete</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
```

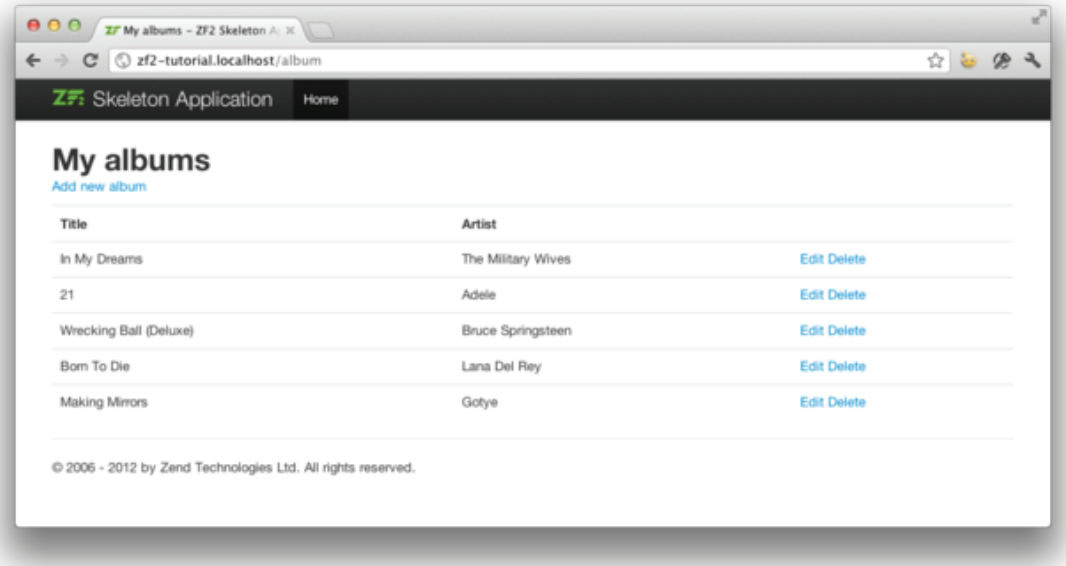
La première chose que nous faisont est de définir le titre de la page (utilisé dans le layout) et de définir le titre dans la section <head> utilisant la méthode helper de vue 'headTitle()' qui affichera le titre dans la barre de titre du browser. Nous créons alors un lien pour ajouter un nouvel album.

La méthode de vue helper 'url()' est fournie par Zend Framework 2 et est utilisée pour créer les liens dont nous avons besoin. Le premier paramètre de 'url()' est le nom de route que nous souhaitons utiliser pour la construction de l'URL, et le second paramètre est un tableau de toutes les variables à adapter dans l'espace de noms réservés que l'on utilise. Dans cette condition nous utilisons notre route 'album' qui est définie pour accepter 2 espaces de noms réservés des variables: 'action' et 'id'.

Nous parcourons le '$albums' que nous assignons depuis le controller d'action. Le sytème de vue Zend Framework 2 assure automatiquement que ces variables sont extraites dans le champ du scripte de vue, tel que nous n'avons pas à nous soucier d'eux puisqu'elles sont préfixées avec $this-> comme nous utilisions avec Zend Framework 1, toutefois vous pouvez le faire aussi si vous le souhaitez.

Nous créons alors une table pour afficher chaque titre et artiste d'album, et fournissons des liens pour permettre d'éditer et supprimer l'enregistrement. Une boucle 'foreach:' standard est utilisée à chaque parcours de la liste des albums, et nous utilisons un formulaire alternatif utilisant une colonne et un 'endforeach;' car il est plus facile à balayer que d'essayer de faire correspondre les accolades. Encore une fois, la méthode de vue helper 'url()' est utilisée pour créer les liens d'édition et de suppression.

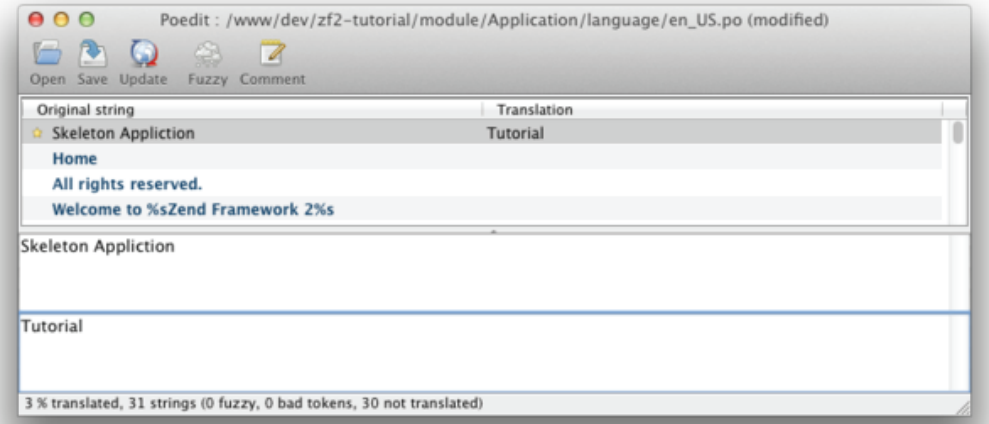
Si vous ouvrez l'adresse http://zf2-tutorial.localhost/album vous devriez voir ceci:



Styling ans Translations

Nous avons choisi le style SkeletonApplication, qui est très bien, mais nous voulons changer le titre et enlever le message de copyright.

Le ZendSkeletonApplication est défini pour utiliser la fonctionnalité de traduction 'Zend\I18n' pour tous les textes. Ceci utilise les fichiers de type '.po' situés dans Application/language, et vous avez besoin d'un éditeur 'poedit' pour changer le texte. Démarrez 'poedit' et ouvrez application/language/en_US.po. Cliquez sur 'Skeleton Application' dans la liste de 'Original string' et alors saisissez dans 'Tutorial' comme traduction.

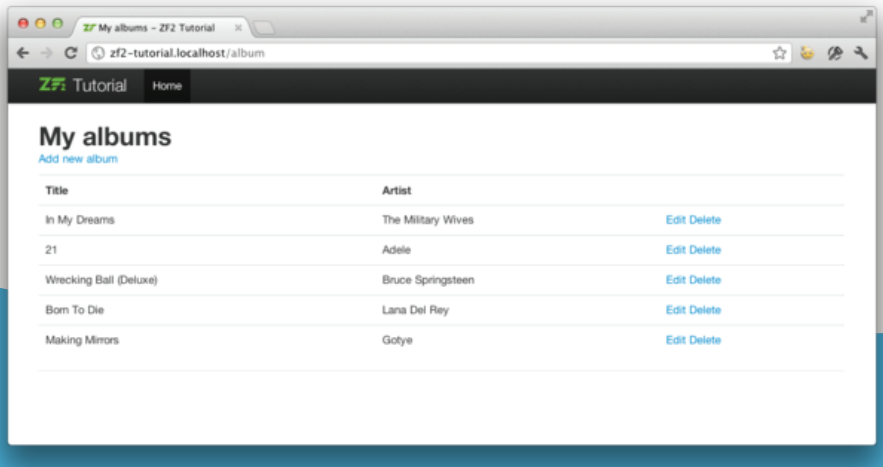


Cliquez sur 'Save' dans la barre d'outils et 'poedit' créera un fichier 'en_US.mo' pour nous. Si le fichier de type '.mo' n'est pas généré, cliquer sur 'Preferences -> Editor -> Behavior' et regardez si la case 'Automatically compile .mo file on save' est cochée.

Pour supprimer le message de copyright, nous avons besoin d'éditer le script de vue ' layout.phtml' du module 'Application':

```
// module/Application/view/layout/layout.phtml:
// Remove this line:
<p>&copy; 2005 - 2012 by Zend Technologies Ltd. <?php echo
$this->translate('All
rights reserved.') ?></p>
```

La page ressemble maintenant à ceci:



Source: Zend Framework 2
http://framework.zend.com/manual/2.2/en/user-guide/overview.html

Forms and actions

Adding news albums

Nous pouvons maintenant réaliser le fonctionnement pour ajouter de nouveaux albums. Il y a deux sections pour cette partie:

- Afficher un formulaire pour que l'utilisateur fournisse les détails ;
- Procéder la soumission du formulaire et stocker dans la base de données.

Nous utilisons 'Zend\Form' pour faire cela. Le composant 'Zend\Form' commande le formulaire et la validation du formulaire, nous ajoutons Zend\InputFilter à notre entité 'Album'. Nous commençons par créer une nouvelle classe 'Album\Form\AlbumForm' qui étend 'Zend\Form\Form' pour définir notre formulaire. Créer un fichier appelé 'AlbumForm.php' situé dans module/Album/src/Album/Form:

```
<?php
namespace Album\Form;

use Zend\Form\Form;

class AlbumForm extends Form
{
    public function __construct($name = null)
    {
        // we want to ignore the name passed
        parent::__construct('album');
        $this->setAttribute('method', 'post');
        $this->add(array(
            'name' => 'id',
            'type' => 'Hidden',
        ));
        $this->add(array(
            'name' => 'title',
            'type' => 'Text',
            'options' => array(
                'label' => 'Title',
            ),
        ));
        $this->add(array(
            'name' => 'artist',
            'type' => 'Text',
            'options' => array(
                'label' => 'Artist',
            ),
        ));
        $this->add(array(
            'name' => 'submit',
            'type' => 'Submit',
            'attributes' => array(
                'value' => 'Go',
                'id' => 'submitbutton',
            ),
        ));
    }
}
```

Dans le constructeur de 'AlbumForm' nous faisons plusieurs choses. Premièrement nous définissons le nom du formulaire avec le nom du constructeur parent. Nous définissons alors la méthode d'envoie du formulaire qui est dans notre cas 'post'. Finalement nous créons 4 éléments de formulaire: id, title, artist et le bouton de soumission. Pour chaque item nous définissons différentes variables et options, incluant le label pour être affiché.

Nous avons besoin aussi de définir la validation de ce formulaire. Dans Zend Framework 2 cela est fait en utilisant un 'input filter', qui peut être soit indépendant ou définie dans toutes classes qui implémentent l'interface 'InputFilterAwareInterface ', tel une entité 'model'. Dans notre cas, nous allons ajouter un 'input filter' à la classe 'Album', qui réside dans le fichier 'Album.php' situé dans module/Album/src/Album/Model:

```
<?php
namespace Album\Model;

// Add these import statements
use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;

class Album implements InputFilterAwareInterface
{
    public $id;
    public $artist;
    public $title;
    protected $inputFilter;                    // <-- Add this variable

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id']))     ? $data['id']     : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title']))  ? $data['title']  : null;
    }

    // Add content to these methods:
    public function setInputFilter(InputFilterInterface $inputFilter)
    {
        throw new \Exception("Not used");
    }

    public function getInputFilter()
    {
        if (!$this->inputFilter) {
            $inputFilter = new InputFilter();
            $factory     = new InputFactory();

            $inputFilter->add($factory->createInput(array(
                'name'     => 'id',
                'required' => true,
                'filters'  => array(
                    array('name' => 'Int'),
                ),
            )));
```

```
            $inputFilter->add($factory->createInput(array(
                'name'     => 'artist',
                'required' => true,
                'filters'  => array(
                    array('name' => 'StripTags'),
                    array('name' => 'StringTrim'),
                ),
                'validators' => array(
                    array(
                        'name'    => 'StringLength',
                        'options' => array(
                            'encoding' => 'UTF-8',
                            'min'      => 1,
                            'max'      => 100,
                        ),
                    ),
                ),
            )));

            $inputFilter->add($factory->createInput(array(
                'name'     => 'title',
                'required' => true,
                'filters'  => array(
                    array('name' => 'StripTags'),
                    array('name' => 'StringTrim'),
                ),
                'validators' => array(
                    array(
                        'name'    => 'StringLength',
                        'options' => array(
                            'encoding' => 'UTF-8',
                            'min'      => 1,
                            'max'      => 100,
                        ),
                    ),
                ),
            )));

            $this->inputFilter = $inputFilter;
        }

        return $this->inputFilter;
    }
}
```

Le InputFilterAwareInterface définie deux méthodes: setInputFilter() et getInputFilter(). Nous avons seulement besoin d'implémenter getInputFilter() pour simplement lancer une exception dans setInputFilter().

Dans getInputFilter(), nous instancions un InputFilter et alors nous ajoutons les 'input' que nous avons besoin. Nous en ajoutons un pour chaque propriété que nous souhaitons filtrer où valider. Pour le champ 'id' nous ajoutons un filtre 'Int' puisque nous avons besoin d'entiers.Pour les éléments textes, nous ajoutons deux filtres, StripTags et StringTrim, pour supprimer le code HTML indésirable et accessoirement les espaces blancs. Nous les définissons aussi pour qu'ils soient obligatoires et ajoutons un validateur StringLength pour s'assurer que l'utilisateur ne saisisse pas trop de caractères que nous voulons stocker dans la base de données.

Nous avons besoin maintenant d'obtenir le formulaire pour afficher et procéder à la soumission. Ceci est fait avec la méthode 'addAction()' du controller 'AlbumController':

```
// module/Album/src/Album/Controller/AlbumController.php:

//...
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Album\Model\Album;          // <-- Add this import
use Album\Form\AlbumForm;       // <-- Add this import
//...

    // Add content to this method:
    public function addAction()
    {
        $form = new AlbumForm();
        $form->get('submit')->setValue('Add');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $album = new Album();
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());

            if ($form->isValid()) {
                $album->exchangeArray($form->getData());
                $this->getAlbumTable()->saveAlbum($album);

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }
        return array('form' => $form);
    }
//...
```

Après avoir ajouté ‘AlbumForm’ à la liste d’utilisation, nous implémentons la méthode ‘addAction()’. Regardons le code de la méthode ‘addAction()’ un peu plus en détail:

```
$form = new AlbumForm();
$form->get('submit')->setValue('Add');
```

Nous instancions ‘AlbumForm’ et nous définissons le label du bouton à ‘Add’. Nous faisons ceci ici puisque nous réutiliserons le formulaire lors de l’édition d’un album et utiliserons différents label.

```
$request = $this->getRequest();
if ($request->isPost()) {
    $album = new Album();
    $form->setInputFilter($album->getInputFilter());
    $form->setData($request->getPost());
    if ($form->isValid()) {
```

Si la méthode isPost() de l’objet ‘Request’ est vraie, alors le formulaire a été soumis et nous définissons un ‘input filter’ du formulaire depuis une instance album. Nous définissons alors les données postées au formulaire et vérifions qu’il est valide en utilisant la méthode membre ‘isValid()’ du formulaire.

```
$album->exchangeArray($form->getData());
$this->getAlbumTable()->saveAlbum($album);
```

Si le formulaire est valide, alors nous saisissons les données depuis le formulaire et stockons par l’utilisation de la méthode ‘saveAlbum’ du ‘model’.

```
/ Redirect to list of albums
return $this->redirect()->toRoute('album');
```

Après que nous avons sauvegardé la ligne du nouvel album, nous sommes redirigés vers la liste des albums en utilisant le controller ‘Redirect’ du plugin.

```
return array('form' => $form);
```

Finalement, nous retournons les variables que nous voulons assigner à la vue. Dans notre cas, seulement l’objet formulaire. Notez que Zend Framework 2 vous permet aussi de retourner simplement un tableau contenant les variables devant être assignées à la vue et il créer pour vous un ‘ViewModel’ en arrière plan.

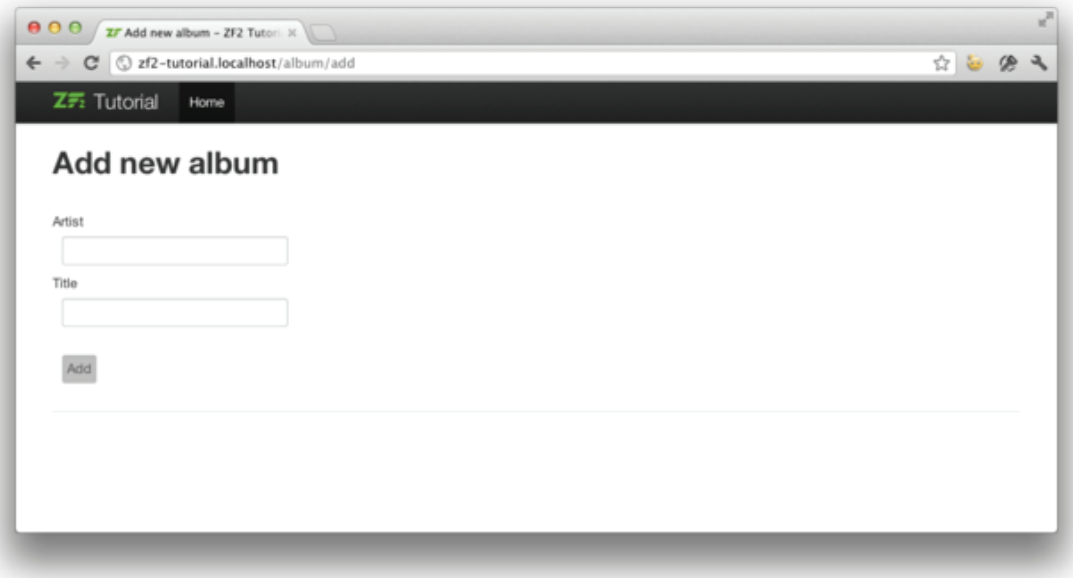
Nous avons besoin maintenant de mettre le formulaire dans le scripte de vue ‘add.phtml’:

```
<?php
// module/Album/view/album/album/add.phtml:

$title = 'Add new album';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>
<?php
$form = $this->form;
$form->setAttribute('action', $this->url('album', array('action' => 'add')));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formSubmit($form->get('submit'));
echo $this->form()->closeTag();
```

De nouveau, nous affichons un titre avant et alors nous mettons le formulaire. Zend Framework fournis quelques méthodes ‘helpers’ pour créer facilement ceci. La méthode de vue form() ‘helper’ a une méthode openTag() et closeTag() que nous utilisons pour ouvrir et fermer le formulaire. Alors pour chaque élément avec un label, nous utilisons formRow(), mais pour deux éléments qui sont idépendants, nous utilisons formHidden() et formSubmit().



Alternativement, le processus de rendu du formulaire peut être simplifié en utilisant la vue 'helper' 'formCollection' fournie. Par exemple, dans le script de vue précédent, remplacez toutes les déclarations 'echo' de formulaire par:

```
echo $this->formCollection($form);
```

Note: vous devrez toujours appeler les méthodes d'ouverture et de fermeture du formulaire. Vous remplacez les autres 'echo' ci-dessus avec l'appel à formCollection.

Ceci parcoura la structure du formulaire, appelant les labels appropriés, les éléments et messages d'aide d'erreurs pour chaque élément, mais vous devrez encore développer le FormCollection ($ form) avec les balises de formulaire qui s'ouvrent et se ferment. Cela permet de réduire la complexité de votre script de vue dans des situations où le rendu HTML par défaut de la forme est acceptable.

Vous devriez être maintenant capable d'utliser le lien 'Add new album' de la page d'application pour ajouter un nouveau enregistrement d'album.

Editer un album est presque identique à l'ajout d'album, tant le code est similaire. Cette fois nous utilisons editAction() dans AlbumController:

```
// module/Album/src/Album/Controller/AlbumController.php:
//...

    // Add content to this method:
    public function editAction()
    {
        $id = (int) $this->params()->fromRoute('id', 0);
        if (!$id) {
            return $this->redirect()->toRoute('album', array(
                'action' => 'add'
            ));
        }

        // Get the Album with the specified id.  An exception is thrown
        // if it cannot be found, in which case go to the index page.
        try {
            $album = $this->getAlbumTable()->getAlbum($id);
        }
        catch (\Exception $ex) {
            return $this->redirect()->toRoute('album', array(
                'action' => 'index'
            ));
        }

        $form  = new AlbumForm();
        $form->bind($album);
        $form->get('submit')->setAttribute('value', 'Edit');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());

            if ($form->isValid()) {
                $this->getAlbumTable()->saveAlbum($album);

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }

        return array(
            'id' => $id,
            'form' => $form,
        );
    }
//...
```

Ce code nous est familier. Regardons les différences avec l'ajout d'albums. Premièrement, nous voyons que le 'id' qui est dans le chemin identifie et est utilisé pour charger l'album à être édité:

```
$id = (int) $this->params()->fromRoute('id', 0);
if (!$id) {
    return $this->redirect()->toRoute('album', array(
        'action' => 'add'
    ));
}

// Get the album with the specified id.  An exception is thrown
// if it cannot be found, in which case go to the index page.
try {
    $album = $this->getAlbumTable()->getAlbum($id);
}
catch (\Exception $ex) {
    return $this->redirect()->toRoute('album', array(
        'action' => 'index'
    ));
}
```

Params est un plugin controller qui fournis une façon convenable de récupérer les paramètres depuis le chemin. Nous l'utilisons pour récupérer le 'id' depuis le chemin que nous créons dans le fichier 'module.config.php' du module. Si le 'id' est nul, alors nous redirigeons vers l'action d'ajout, sinon, nous continuons pour obtenir l'album depuis la base de données.

Nous avons validé le fonctionnement nous assurant que Album avec le 'id' spécifié peut être trouvé. Si ce n'est pas le cas, alors une exception est transmise. Nous réceptionnons cette exception et redirigeons l'utilisateur vers la page 'index'.

```
$form = new AlbumForm();
$form->bind($album);
$form->get('submit')->setAttribute('value', 'Edit');
```

La méthode bind() du formulaire attache le 'model' au formulaire. Ceci est utilisé de deux manières:

- Lors de l'affichage du formulaire, les valeurs initiales pour chaque élément sont extraites depuis le model ;
- Après succès de la validation dans isValid(), les données depuis le formulaire sont mises dans le model.

Ces opérations sont faites en utilisant un objet hydrateur. Il y a un certain nombre d'hydrateurs, mais celui par défaut est Zend\Stdlib\Hydrator\ArraySerializable qui s'utilise pour trouver deux méthodes dans le model: getArrayCopy() et exchangeArray(). Nous avons une écriture exchangeArray() toute prête dans notre Album, aussi nous avons besoin d'écrire getArrayCopy():

```
// module/Album/src/Album/Model/Album.php:
// ...
    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id']))     ? $data['id']     : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title']))  ? $data['title']  : null;
    }

    // Add the following method:
    public function getArrayCopy()
    {
        return get_object_vars($this);
    }
// ...
```

A la suite de l'utilisation de bind() avec son hydrateur, nous avons pas besoin de remplir $album de données de formulaire puisque celà à déjà été fait, seulement nous devons appeler la méthode 'saveAlbum()' du 'mapper' pour stocker les changements dans la base de données.

La vue squelette, edit.phtml, est très similaire à la vue d'ajout d'album:

```
<?php
// module/Album/view/album/album/edit.phtml:

$title = 'Edit album';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url(
    'album',
    array(
        'action' => 'edit',
        'id'     => $this->id,
    )
));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formSubmit($form->get('submit'));
echo $this->form()->closeTag();
```

Les seuls changements sont l'utilisation de 'title' de Edit Album et la définition d'action du formulaire à l'action 'edit'.

Vous êtes capable maintenant d'éditer des albums.

Deleting a album

Pour compléter notre application, nous avons besoin d'ajouter un système de suppression. Nous avons un lien 'Delete' après chaque album sur notre page de liste et l'approche serait de faire une suppression lors du clic. Ceci est faux. Rappelons-nous de notre HTTP, nous rappelons que vous ne devriez pas faire une action irresversible utilisant GET et devrait plutôt utiliser POST.

Nous devons voir un formulaire de confirmation quand l'utilisateur clique sur 'delete' et lorsqu'il clique sur 'yes', nous ferons la suppression. Comme le formulaire est simple, nous allons coder directement dans notre vue (Zend\Form est, après tout, optionnel).

Commençons avec l'action code dans AlbumController::deleteAction():

```
// module/Album/src/Album/Controller/AlbumController.php:
//...
    // Add content to the following method:
    public function deleteAction()
    {
        $id = (int) $this->params()->fromRoute('id', 0);
        if (!$id) {
            return $this->redirect()->toRoute('album');
        }

        $request = $this->getRequest();
        if ($request->isPost()) {
            $del = $request->getPost('del', 'No');

            if ($del == 'Yes') {
                $id = (int) $request->getPost('id');
                $this->getAlbumTable()->deleteAlbum($id);
            }

            // Redirect to list of albums
            return $this->redirect()->toRoute('album');
        }

        return array(
            'id'    => $id,
            'album' => $this->getAlbumTable()->getAlbum($id)
        );
    }
//...
```

Comme précédemment, nous obtenons l'id de l'itinéraire, et vérifions l'isPost () de l'objet de requête afin de déterminer s'il faut afficher la page de confirmation ou de supprimer l'album. Nous utilisons l'objet de la table pour supprimer la ligne en utilisant la méthode supprimerAlbum () puis retournons à la liste des albums. Si la demande n'est pas un POST, alors nous récupérons l'enregistrement de base de données correct et nous attribuons à la vue, avec l'id.

Le script de vue est un simple formulaire:

```
<?php
// module/Album/view/album/album/delete.phtml:

$title = 'Delete album';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>

<p>Are you sure that you want to delete
    '<?php echo $this->escapeHtml($album->title); ?>' by
    '<?php echo $this->escapeHtml($album->artist); ?>'?
</p>
<?php
$url = $this->url('album', array(
    'action' => 'delete',
    'id'     => $this->id,
));
?>
<form action="<?php echo $url; ?>" method="post">
<div>
    <input type="hidden" name="id" value="<?php echo (int) $album->id; ?>" />
    <input type="submit" name="del" value="Yes" />
    <input type="submit" name="del" value="No" />
</div>
</form>
```

Dans ce script, nous affichons un message de confirmation à l'utilisateur et un formulaire avec des boutons 'Yes' et 'No' . Dans cette action, nous vérifions spécifiquement la valeur "Oui" lorsque vous effectuez la suppression

Ensuring that the home page displays the list of albums

Le point final. A ctuellement, la page ‘home’, http://zf2-tutorial.localhost/ n’affiche pas la liste d’albums.

Ceci est dû au chemin mis dans le fichier module.config.php du module Application. Pour changer cela, ouvrir module/Application/config/module.config.php et trouver le chemin ‘home:’

```
'home' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/',
        'defaults' => array(
            'controller' => 'Application\Controller\Index',
            'action'     => 'index',
        ),
    ),
),
```

Changer le controller depuis Application\Controller\Index à Album\Controller\Album:

```
'home' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/',
        'defaults' => array(
            'controller' => 'Album\Controller\Album', // <-- change here
            'action'     => 'index',
        ),
    ),
),
```

Ceci fait vous avez maintenant une application fonctionnelle.

Conclusion

Ceci conclue notre briève construction, mais fonctionnelle, MVC application utilisant Zend Framework 2.

Dans ce tutorial nous avons brièvement vue un nombre de différentes parties du framework.

La partie la plus importante de la construction d'applications avec Zend Framework 2 sont les modules, les blocs de construction de toute application MVC Zend Framework.

Pour faciliter le travail avec des dépendances à l'intérieur de nos applications, nous utilisons le gestionnaire de services.

Pour être en mesure de cartographier une demande aux contrôleurs et leurs actions, nous utilisons des routes.

La persistance des données, dans la plupart des cas, consiste à utiliser Zend \ Db pour communiquer avec l'une des bases de données. Les données d'entrée sont filtrées et validées avec des filtres d'entrée et avec Zend \ Form ils fournissent un pont solide entre le modèle du domaine et la couche de vue.

Zend\View est responsable de l'affichage de la pile MVC, avec une grande quantité de méthodes de vue.