Cours de PHP

Cours n°4 : Tests & Framework PHP

Tests

Multimedia
Technologies de l'Information

Tests

Pourquoi tester ?

- **Lors de la réalisation de test, la définition d'une stratégie de test est une étape indispensable pour « prouver » la qualité d'un projet**
- **Une stratégie de test se construit autour de différents types de tests**
  - Tests techniques
  - Tests unitaires
  - Tests fonctionnels
  - Use Case
  - Tests de performances
- **PHP offre des bibliothèques permettant la mise en œuvre de tests unitaires**
  - PHPUnit
  - SimpleTest
- **Les tests fonctionnels/Use case se font avec d'autres outils comme Sélénium.**

Tests

PHP Unit

- **Une bibliothèque simple à utiliser pour créer des cas de test unitaire**

```php
<?php
class Person
{
    private $name;

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

    public function getName()
    {
        return $this->name;
    }
}
```

```php
<?php
class PersonTest extends PHPUnit_Framework_TestCase
{
    // ...

    public function testgetName()
    {
        // Arrange
        $a = new Person("Simon");

        // Assert
        $this->assertEquals("Simon", $a->getName());
    }

    // ...
}
```

- **Et pour tester ?**

```
phpunit --bootstrap src/autoload.php tests/PersonTest
```

- Le fichier src/autoload.php contient une fonction de type spl_autoload_register qui permettra le chargement de votre classe dans les cas de tests
- Si seul un dossier est fourni (par exemple tests), PHPUnit executera tous les fichiers finissant par *Test.php

Gestion de bibliothèque

Multimedia
Technologies de l'Information

Composer

Présentation

- **Composer est un gestionnaire de dépendance sur les bibliothèque PHP**
  - Composer ne gère pas l'installation de packages ou la mise en œuvre de squelette d'application nativement
  - Composer installe les bibliothèques dont dépendent votre projet

Fonctionnement

- **Chaque projet utilisant Composer nécessite un fichier « composer.json »**
  - Ce fichier contient les bibliothèques dont dépendent le projet
  - Composer va aller télécharger ces bibliothèques et leurs dépendances

```
{
    "require": {
        "phpunit/phpunit": "4.0.*"
    }
}
```

  - La commande « composer install » va installer PHPUnit et toutes ses dépendances dans un dossier « vendor »
  - Pour utiliser vos bibliothèques dans votre projet PHP, ajoutez à votre index.php :

```
<?php
    require 'vendor/autoload.php';
```

  - Et? C'est tout!

Composer doit être utilisé lorsque vous travailler à plusieurs sur un projet et qu'il utilise différentes bibliothèques!

Multimedia
Technologies de l'Information

Pear / Pyrus

Présentation

- **Pyrus = Pear 2**
  - Pear était un dépôt central de toutes les bibliothèques OpenSource PHP
  - Pyrus a pour objectif d'apporter plus de souplesse que Pear (qui ne se limitait qu'à l'installation de package sans gestion de dépendance)

Fonctionnement

- **Pyrus ne peut s'utiliser qu'en ligne de commande**
  - Il est nécessaire de créer un script pour automatiser ses commandes Pyrus

```
pyrus.phar install package_name
```

La communauté (et donc le nombre de packages) de Pyrus est beaucoup moins active que celle de composer.

Framework PHP

Framework PHP

Définition d'un Framework

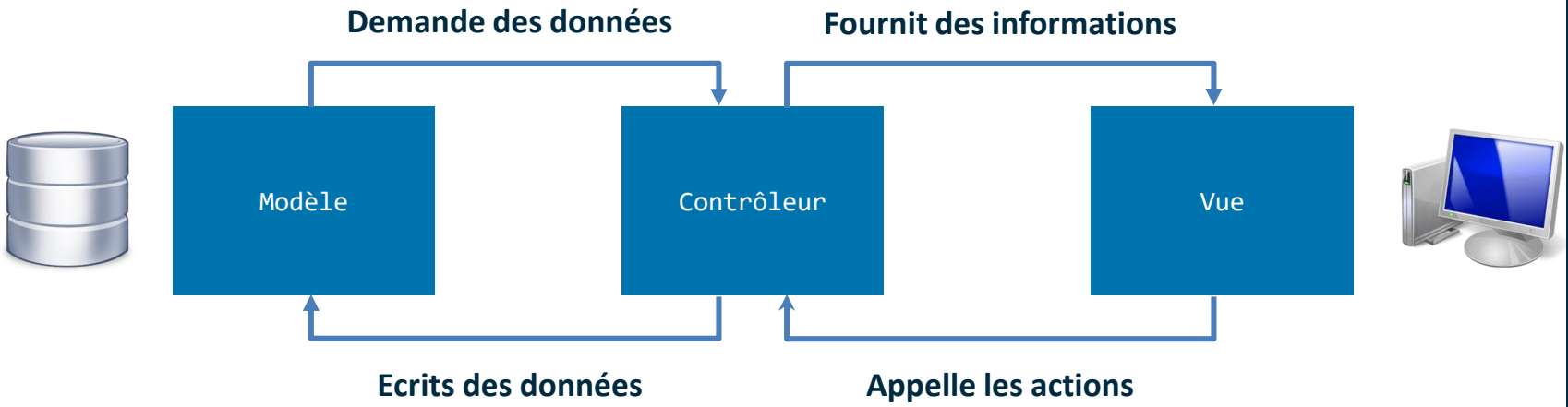
- **Ensemble de composants logiciels**
  - Faiblement spécialisés (permet de « tout » faire)
  - Organisés logiquement et forçant l'organisation via des patterns

Framework appliqué au PHP

- **La plupart des frameworks PHP suivent et obligent l'utilisation du modèle MVC**

Présentation du MVC

- **Définition :**
  - MVC pour Modèle-Vue-Contrôleur est un pattern de programmation dédiés à la réalisation d'application
  - Son principal objectif est de garantir la séparation entre les données (modèle), l'affichage (la vue) et les actions (le contrôleur)



Framework PHP

Présentation du MVC

- **Le modèle**
  - Le modèle correspond aux données
    - Elles sont généralement stockées dans une base de données
    - Le modèle est représenté sous forme de classe dans le code PHP
- **La vue**
  - La vue correspond à ce qui est affiché à l'utilisateur
    - La vue ne connait pas le modèle de données
    - La vue ne s'occupe que de l'affichage
    - Une vue est essentiellement constituée d'HTML
- **Le contrôleur**
  - Le contrôle contient la logique de votre application
    - Un contrôleur exécute des actions
    - Les actions utilise les données du modèle pour les transmettre à la vue

Multimedia
Technologies de l'Information

Framework PHP

Principaux Framework PHP

- **Zend Framework 2**
  - Création : 2006 (version 1)
- **Symfony 2**
  - Création : 2005 (version 1)
- **CakePHP 2**
  - Création : 2005 (version 1)
- **CodeIgniter 2**
  - Création : 2006 (version 1)
- **Yii**
  - Création : 2008
- **Laravel**
  - Création : 2011 (version 1)

TP Zend Framework 2

Multimedia
Technologies de l'Information

Zend Framework 2

Objectifs du TP

- Configuration d'un serveur apache pour utiliser ZendFramework
- Création d'un Blog à partir de ZendFramework
- Créer un WebService Rest à partir de ZendFramework

Zend Framework 2

Génération du Zend « Squelette »

- **Le package ZendSqueletton est disponible à cette adresse**
  - https://github.com/zendframework/ZendSkeletonApplication
  - Télécharger le projet en .ZIP
  - Dézipper l'application dans le dossier www (celui utilisé par votre Apache)
  - Renommer le dossier ZendApply
- **Ouvrir une console ou un terminal**
  - Exécuter la commande suivante (depuis le dossier ZendApply)

```
Shell$ /var/to/php composer.phar install
ou
Cmd$ c:\wamp\bin\php[…]\php.exe composer.phar install
```

  - Si ca ne fonctionne pas, il faut activer l'extension openssl

Zend Framework 2

Création et activation du vhost 1/2

- **Activer le mod_rewrite sur Apache**
  - Le mode rewrite permet de récrire les URL à la volée pour passer d'une URL du type www.toto.com/read.php?id=1 par www.toto.com/read/1
  - Ouvrir le fichier httpd.conf
  - Remplacer la ligne suivante

```
#LoadModule rewrite_module modules/mod_rewrite.so
LoadModule rewrite_module modules/mod_rewrite.so
```

  - Redémarrer apache
- **Editer de fichier host**
  - Le fichier host est un fichier qui permet d'associer à un hostname une adresse IP sans passer par le serveur DNS
  - Modifier le fichier host
    - Pour Windows (C:\Windows\System32\Driver\etc\hosts) avec un Bloc Note ouvert en administrateur
    - Pour Linux/Mac faite un sudo nano /etc/hosts

```
127.0.0.1       zf2app.mti
```

Zend Framework 2

Création et activation du vhost 2/2

- **Activer le vhost**
  - Un vhost pour virtualhost est un moyen de personnaliser la configuration d'apache pour un domaine/sous domaine
  - Ajouter le code suivant à la fin de votre fichier httpd.conf (en enlevant les commentaires)

```
<VirtualHost 127.0.0.1>
#Domaine associé VirtualHost
    ServerName zf2app.mti
#Chemin vers la racine du domaine
    DocumentRoot /cheminvers/ZendApply/public
#Ajout d'une variable d'environnement pour ZF
    SetEnv APPLICATION_ENV "development"
    <Directory /cheminvers/ZendApply/public>
#Règles applicable pour le dossier
        DirectoryIndex index.php
#Permet l'utilisation des .htaccess
        AllowOverride All
#Les règles de permission de type allow sont prioritaires
        Order allow,deny
#Permet l'accès à tous les utilisateurs
        Allow from all
    </Directory>
</VirtualHost>
```

  - Redémarrer Apache

Zend Framework 2

Accès à l'application

- **Si tout est OK, l'appel à l'URL suivante http://zf2app.mti devrait afficher le contenu suivant :**

Zend Framework 2

Création du module 1/5

- **Zend Framework 2 est basé sur un système de module**
  - Par défaut, le module application est présent. Il contient le bootstrap (la configuration général du framework, la gestion des erreurs, le routage, la page d'accueil)
  - Les modules sont gérés par le Module Manager
- **Pour créer un module, il faut ajouter l'arborescence suivante à ZendApply**

```
ZendApply/
    /module
        /Blog
            /config
            /src
                /Blog
                    /Controller
                    /Form
                    /Model
            /view
                /blog
                    /blog
```

Zend Framework 2

Création du module 2/5

- **Pour notifier Zend de la création du module**
  - Par défaut, le module application est présent. Il contient le bootstrap (la configuration générale du framework, la gestion des erreurs, le routage, la page d'accueil)
  - Pour que Zend détecte un module grâce au ModuleManager. Afin que celui-ci fonctionne un fichier Module.php doit être présent à la racine de notre module.

```php
<?php
namespace Blog;
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';
    }
}
```

  - Les fonctions **getAutoloaderConfig** et **getConfig** sont appelée automatiquement par ZF

Zend Framework 2

Création du module 3/5

- **Autoloading**
  - La fonction **getAutoloaderConfig** retourne les scripts permettant le chargement automatique des classes de notre module.

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

  - Pour assurer le fonctionnement de l'autoloader, il faut ajouter un fichier autoload_classmap.php

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

  - On laisse la main au StandardAutoloader pour l'instant

Zend Framework 2

Création du module 4/5

- **Gestion de la configuration**
  - La fonction **getConfig** retourne les scripts gérant la configuration du module

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

  - Pour assurer le fonctionnement de la méthode, il faut créer le fichier module.config.php dans le dossier ZendApply/Module/Blog/config

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

  - Ce fichier définit le fait que notre module contient un contrôleur et un gestionnaire de vue

Zend Framework 2

Création du module 5/5

- **Déclarer le module**
  - Pour que le ModuleManager connaisse l'existence du nouveau module, il faut modifier le fichier **ZendApply/config/application.config.php**
  - Le fichier contient déjà certains éléments, il faut arriver à un résultat de ce type

```php
<?php
    return array(
        'modules' => array(
            'Application',
            'Blog', // A ajouter
         ),
    [...]
        'module_listener_options' => array(
            'config_cache_enabled' => false, // A ajouter
            'cache_dir' => 'data/cache', // A ajouter
    [...]
),
?>
```

Zend Framework 2

Création du contrôleur 1/3

- **Choisir nos actions**
  - Un contrôleur est composé de plusieurs actions
  - Chaque action contient du code métier à exécuter pour un objectif cible
  - Dans le cadre de notre blog nous avons 4 actions que nous aimerions exécuter
    - Accueillir/Afficher nos articles (index)
    - Ajouter un nouvel article (add)
    - Editer un article (edit)
    - Supprimer un article (delete)
  - Afin de respecter la nomenclature de nommage de Zend Framework, notre contrôleur s'appelera **BlogController**

Zend Framework 2

Création du contrôleur 2/3

Definir les règles de routage

- La route est l'URL nécessaire pour atteindre une action, chaque élément de la route contient une information (en général, /contrôleur/action/identifiant)
- Afin de pouvoir utiliser le contrôleur, nos allons définir les règles de routage pour son utilisation.
  - Ces règles se définissent dans le **module.config.php** du Module Blog

```php
<?php
return array(
    'controllers' => array(
    [...]
    ),
    'router' => array(
       'routes' => array(
           'blog' => array(
               'type' => 'segment',
               'options' => array(
                   'route' => '/blog[/:action][/:id]', // règle de routage
                   'constraints' => array(
                       'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                       'id' => '[0-9]+',
                   ),
                   'defaults' => array(
                       'controller' => 'Blog\Controller\Blog',
                       'action' => 'index',
                   ),
               ),
           ),
       ),
    ),
    'view_manager' => array(
    [...]
```

  - Comme vu précédemment, la règle de routage est **/blog[/:action][/:id]**
  - Ces règles permettent de définir des URL du type suivant
    - zf2app.mti/blog (index)
    - zf2app.mti/blog/add (index)
    - zf2app.mti/blog/edit/X (edit)
    - Zf2app.mti/blog/delete/Y

Zend Framework 2

Création du contrôleur 3/3

- **Définition de la classe du contrôleur**
  - Le routage étant définit, le contrôleur peut-être écrit.
  - Créer le fichier ZendApply/module/Blog/src/Blog/Controller/BlogController.php

```php
<?php

namespace Blog\Controller;

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

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

    public function addAction()
    {
    }

    public function editAction()
    {
    }

    public function deleteAction()
    {
    }

}
```

  - Notre classe contient une méthode pour chaque action à effectuer

Zend Framework 2

Initialisation des vues

- **Création des fichiers nécessaires à la vue**
  - Une vue doit être créée pour chaque action de notre contrôleur
  - Les vues se situe dans le dossier suivant ZendApply/module/blog/view/blog/blog/*
  - Créer les fichiers suivants avec un `<?='Nom du fichier'?>` dans le code
    - index.phtml
    - add.phtml
    - edit.phtml
    - delete.phtml
  - Ces fichiers seront appelé par le ViewListener du module Application (le module principal de Zend Framework)
  - A partir de cette étape, vous devriez être capable d'appeler les URL suivantes sans erreurs :
    - www.zf2app.mti/blog
    - www.zf2app.mti/blog/add
    - www.zf2app.mti/blog/delete
    - www.zf2app.mti/blog/edit

Zend Framework 2

Création de la base de données

- **Afin de pouvoir utiliser notre modèle, nous allons avoir besoin d'une base de données avec des informations à stocker**
  - Accéder à PhpMyAdmin (http://127.0.0.1/phpmyadmin)
  - Créer une base nommée mti_db

```
CREATE TABLE IF NOT EXISTS `blog` ( `id` int(10)
unsigned NOT NULL AUTO_INCREMENT, `title` varchar(100)
NOT NULL, `text` text NOT NULL, `blogtime` timestamp
NOT NULL DEFAULT CURRENT_TIMESTAMP, PRIMARY KEY
(`id`));
```

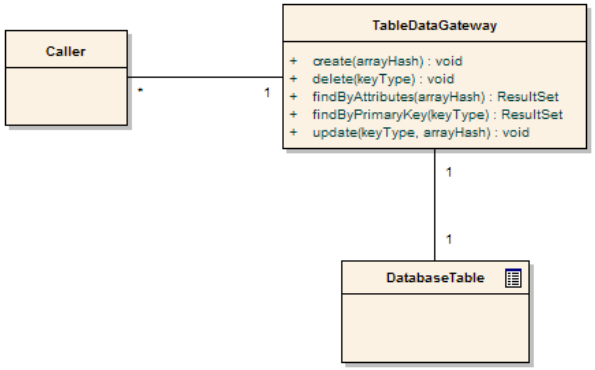
```
INSERT INTO `mti_db`.`blog` (
`id` ,
`title` ,
`text` ,
`blogtime`
)
VALUES (
NULL , 'Mon super Article', 'Voici un exemple de super
Article. Qu''en pensez-vous?<br/> -- <br/> Simon',
CURRENT_TIMESTAMP
);
```

  - Ajouter quelques entrées

Zend Framework 2

Création du modèle

- **Zend Framework ne fournit pas un composant dédié à la gestion du modèle mais plusieurs suivant vos besoins.**
  - Il est aussi possible d'utiliser un ORM comme doctrine
  - Pour ce TP, une table BlogTable va être créée et étendre la classe TableGateway
  - TableGateway est une mise en œuvre du pattern Table Data Gateway
    - Ce Design Pattern créé une interface entre l'appelant et la base de données pour des opérations de type CRUD



Zend Framework 2

Création du modèle

Création du modèle

- Afin de représenter notre objet, nous allons créer une classe PHP qui représentera une entrée dans la base de donnée.
- Le fichier à créer sera ZendApply/module/Blog/src/Blog/Model/Blog.php

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

    class Blog
    {
        public $id;
        public $title;
        public $text;

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

Zend Framework 2

Création du modèle

- **Création de la classe d'accès**
  - La classe est limitée pour l'instant à la récupération de l'ensemble des données
  - Le fichier à créer sera :
    - ZendApply/module/Blog/src/Blog/Model/BlogTable.php
  - Pour l'instant seule la récupération et la sauvegarde sont créés

```php
<?php
namespace Blog\Model;
use Zend\Db\TableGateway\TableGateway;

class BlogTable
{
    protected $tableGateway;

    public function __construct(TableGateway $tableGateway)
    {
        $this->tableGateway = $tableGateway;
}
    public function fetchAll()
    {
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }
    public function saveBlog(Blog $blog)
    {
        $data = array(
            'text' => $blog->text,
            'title'  => $blog->title,
        );

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

Zend Framework 2

Création du modèle

- **Configuration de l'application pour l'accès à la base de données**
  - Sans information de connexion, il ne sera pas possible de se connecter à la BDD
  - Ces informations doivent être initialisée dans **ZendApply/config/Autoload/global.php**

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

  - Le mot de passe doit être mis dans un fichier à part (qui n'est pas pris en compte si vous utilisez git) : **ZendApply/config/Autoload/local.php**

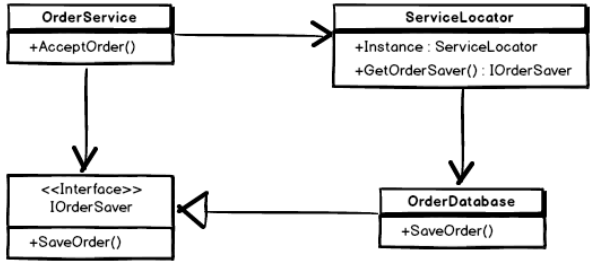
```php
<?php
return array(
    'db' => array(
        'username' => 'root', // Pas bien...
        'password' => '',  // Vraiment pas bien...
    ),
);
```

Zend Framework 2

Création du modèle

- **Utiliser le Service Manager pour injecter notre modèle dans le contrôleur**
  - Le Service Manager est une classe mettant en œuvre le design pattern Inversion of Control (fichier Module.php)

```php
<?php
    /* Ajouter les use qui vont bien…. ResultSet, TableGateway…*%
    class Module
    {
        // Code de getAutoloaderConfig et getConfig...
        public function getServiceConfig()
        {
            return array(
                'factories' => array(
                    'Blog\Model\BlogTable' =>  function($sm) {
                        $tableGateway = $sm->get('BlogTableGateway');
                        $table = new BlogTable($tableGateway);
                        return $table;
                    },
                    'BlogTableGateway' => function ($sm) {
                        $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
                        $resultSetPrototype = new ResultSet();
                        $resultSetPrototype->setArrayObjectPrototype(new Blog());
                        return new TableGateway('blog', $dbAdapter, null, $resultSetPrototype);
                    },
                ),
            );
        }
    }
```



  - Afin que notre Service Manager puisse injecter le contenu du TableGateway dans notre Contrôleur, il faut ajouter une ligne à **ZendApply/module/blog/src/Controller/BlogController.php**

```php
<?php
    [...]
    class BlogController extends AbstractActionController
    {
        protected $blogTable; // Ligne à ajouter
        [...]
    }
```

Zend Framework 2

Mise à jour du contrôleur

- **Définition d'une première action**
  - Notre contrôleur étant maintenant capable de récupérer des données, on peut créer la première action dans notre fichier **ZendApply/module/blog/src/Controller/BlogController.php**

```php
<?php
[...]
class BlogController extends AbstractActionController
{
    public function indexAction()
    {
        return new ViewModel(array(
            'blogs' => $this->getBlogTable()->fetchAll(),
        ));
    }

    public function getBlogTable()
    {
        if (!$this->blogTable) {
            $sm = $this->getServiceLocator();
            $this->blogTable = $sm->get('Blog\Model\BlogTable');
        }
        return $this->blogTable;
    }
    [...]
}
```

  - La classe ViewModel permet de transférer des données à la vue et de définir la vue qui sera appelée (par défault c'est la vue qui porte le nom de l'action, donc index.phtml dans ce cas)

Zend Framework 2

Mise à jour du la vue

- **Affichage des données**
  - Dans la vue, la variable **$this** représente le **ViewHelper** qui possède pleins de méthodes pour gérer l'affichage
  - Les données sont appelables via la variable **$blogs** (initialisée dans indexAction)
  - Le fichier **ZendApply/module/blog/view/blog/blog/index.phtml**

```
<?php
    $title = 'My blogs';
    $this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>
<p>
    <a href="<?php echo $this->url('blog', array('action'=>'add'));?>">Ajouter un nouvel
Article</a>
</p>

<table class="table">
<tr>
    <th>Title</th>
    <th>Text</th>
    <th> </th>
</tr>
<?php foreach ($blogs as $blog) : ?>
<tr>
    <td><?php echo $this->escapeHtml($blog->title);?></td>
    <td><?php echo $blog->text?></td>
    <td>
        <a href="<?php echo $this->url('blog',
            array('action'=>'edit', 'id' => $blog->id));?>">Editer</a>
        <a href="<?php echo $this->url('blog',
            array('action'=>'delete', 'id' => $blog->id));?>">Ajouter</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
```

  - L'url www.zf2app.mti/blog devrait afficher les articles du blog

Zend Framework 2

Formulaire et validation 1/4

Création du formulaire d'ajout

- Maintenant qu'on peut lister nos articles de blog, il serait intéressant de pouvoir en ajouter.
- Nous allons avoir besoin d'un formulaire pour demander le texte et le titre de notre article de blog. Pour cela, il faut créer le fichier suivant **ZendApply/module/blog/src/Blog/Form/BlogForm.php**
- La création de formulaire se fait grâce à Zend/Form

```php
<?php
namespace Blog\Form;

use Zend\Form\Form;

class BlogForm extends Form
{
    public function __construct($name = null)
    {
        parent::__construct('blog');
        $this->setAttribute('method', 'post'); // Définition d'un formulaire en post
        $this->add(array(     // Ajout d'un champ id de type hidden
            'name' => 'id',
            'type' => 'Hidden',
        ));
        $this->add(array(     // Ajout d'un champ title de type texte
            'name' => 'title',
            'type' => 'Text',
            'options' => array(
                'label' => 'Title',
            ),
        ));
        $this->add(array(      // Ajout d'un champ text de type hidden
            'name' => 'text',
            'type' => 'Textarea',
            'options' => array(
                'label' => 'Text',
            ),
        ));
        $this->add(array(      // Ajout du bouton de validation du formulaire
            'name' => 'submit',
            'type' => 'Submit',
            'attributes' => array(
                'value' => 'Go',
                'id' => 'submitbutton',
            ),
        ));
    }
}
```

Zend Framework 2

Formulaire et validation 2/4

- **Validation des données du formulaire**
  - Pour valider les données du formulaire, on va définir les contraintes directement dans le modèle
  - Pour cela, il faut éditer le fichier **ZendApply/module/Blog/src/Model/Blog.php**
    - Notre modèle va implémenter une interface : InputFilterAwareInterface
  - L'ajout de ces éléments va permettre de gérer de façon automatique la validation des données

```php
<?php
    namespace Blog\Model;
    use Zend\InputFilter\Factory as InputFactory;  // Ajouter ces uses
    use Zend\InputFilter\InputFilter;
    use Zend\InputFilter\InputFilterAwareInterface;
    use Zend\InputFilter\InputFilterInterface;

    class Blog implements InputFilterAwareInterface
    {
       protected $inputFilter;  // ajouter cette variable
       [...]
       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(   // ajoute une verification sur l'id
                   'name'     => 'id',
                   'required' => true,
                   'filters'  => array(
                       array('name' => 'Int'),
                   ),
               )));

               $inputFilter->add($factory->createInput(array(    // ajoute une verification sur le titre
                   'name'     => 'title',
                   'required' => true,
                   'filters'  => array(/* A compléter */),
                   'validators' => array(
                       array(
                           'name'    => 'StringLength',
                           'options' => array(
                               'encoding' => 'UTF-8',
                               'min'      => 1,
                               'max'      => 100,
                           ),
                       ),
                   ),
               )));
               $inputFilter->add(/* A completer */)
               $this->inputFilter = $inputFilter;
           }
           return $this->inputFilter;
       }
    }
```

Zend Framework 2

Formulaire et validation 3/4

- **Mise à jour de l'action**
  - L'avant dernière étape est la mise à jour de l'action afin de gérer l'ajout de notre article de Blog

```php
<?php
use Blog\Model\Blog;          // Ajouter ces uses
use Blog\Form\BlogForm;
[...]
class BlogController extends AbstractActionController
{
    public function addAction()
    {
        $form = new BlogForm(); //On génère le formulaire
        $form->get('submit')->setValue('Add'); /*L'envoie des données du formulaire redirigera
vers cette action */
        $request = $this->getRequest(); // On récupère les données du formulaire
        if ($request->isPost()) // Si le formulaire a été rempli
        {
            $blog = new Blog(); // On crée un nouvel article
            $form->setInputFilter($blog->getInputFilter()); // On définit la validation
            $form->setData($request->getPost()); // On récupère les données

            if ($form->isValid()) {
                $blog->exchangeArray($form->getData());
                $this->getBlogTable()->saveBlog($blog);

                // Redirige vers la liste des articles
                return $this->redirect()->toRoute('blog');
            }
        }
        return array('form' => $form);
    }
    [...]
}
```

Zend Framework 2

Formulaire et validation 4/4

- **Définition de la vue**
  - Maintenant que tout est prêt, il faut créer notre vue dans **ZendApply/module/blog/view/blog/blog/add.phtml**

```php
<?php

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

    echo $this->form()->openTag($form);
    echo $this->formCollection($form); // génère le formulaire automatiquement
    echo $this->form()->closeTag();
```

  - Il ne reste plus qu'à tester : www.zf2app.mti/add

Zend Framework 2

Les autres actions

- **La suite…**
  - Maintenant que vous savez comment remplir une action dans le controleur et récupérer les données dans la vue vous pouvez faire de même avec les autres actions
  - Ajouter la gestion de l'action edit et delete
    - Pensez à mettre à jour
      - Les actions
      - Les vues
      - Le modèle (pour mettre à jour BlogTable http://framework.zend.com/manual/2.0/en/modules/zend.db.table-gateway.html)