Programmer avec Axiom

Introduction

Axiom est un Framework orienté objet écrit en PHP. Il à été créé pour des projets de petite et moyenne envergure et à ce titre il est léger et simple d'utilisation. L'idée de départ était de fabriquer un Framework propre et stable sans la complexité ni les couches d'abstractions présentes dans les grands Framework du moment.
Au cours de cet article nous allons voir comment déployer, configurer et utiliser le Framework pour un site de petite taille (à peine quelques vues) et nous allons voir comment Axiom peut nous aider à accélérer nos développements au quotidien. Je tenterai d'y exposer exhaustivement les caractéristiques et les possibilités offertes par Axiom tout en tendant à rester le plus simple et le plus clair possible. Vos commentaires sur cet article sont évidement les bienvenues.

Ce qu'est Axiom:

- Un ensemble cohérent de libraires PHP
- Une implémentation simple et abordable du MVC
- Un moyen de créer des applications web facilement et proprement
- Un projet open source

Ce qu'Axiom n'est pas:

- Un CMS
- Un "couteau-suisse" programmatique
- Une baguette magique qui fera le travail à votre place
- Un grille-pain

Pourquoi un Framework plutôt qu'un CMS ?

Je n'aime pas vraiment les CMS. Là ou le CMS exige souvent de tordre un besoin selon ses propres contraintes, un Framework se tord pour vous permettre de les exprimer. Un Framework apporte au développeur une plus grande liberté, une plus grande souplesse et surtout une plus grande flexibilité.

Important: Axiom vous permet de coder proprement et vous y incite mais ne peut pas vous y forcer. Si vous faites n'importe quoi avec le modèle MVC, que vous créez vos classes sans tenir compte des règles de bons usages, que vous invoquez des modèles à tord et à travers dans les vues ou que vous avez cassé le Framework en éditant des librairies pour un usage pas très net, je ne pourrai rien pour vous et vous devrez vous débrouiller avec vos problèmes. Une sécurité qui protège le programmeur de ses propres erreurs est non seulement un facteur de lourdeur au niveau de la structure du Framework mais à également un coût significatif au niveau des performances de l'application.

Note: Axiom est distribué tel quel sous les termes de la licence GNU Lesser GPL, je ne fournis aucune garantie quand à son fonctionnement ou à son utilisation pour un besoin particulier. Si vous trouvez un bug, une incohérence ou que vous avec tout simplement une remarque ou une bonne idée, je vous invite à rapporter les bugs sur le tacker du projet http://code.google.com/p/php-axiom/issues/list et à faire part de vos remarques sur le forum PHP.

Prérequis

Pour lire et comprendre cet article vous aurez besoin d'une connaissance de base de la programmation orientée objet en PHP 5.1 (et supérieur) et de la programmation web en général (XHTML, CSS, JavaScript etc.)
Le but de cet article n'étant pas d'expliquer ces concepts, je vous recommande ces articles si vous ne vous sentez pas à l'aise dans ces domaines:
- XXXXX
- YYYYY

Les sources d'Axiom ainsi que la documentation technique fournie avec le Framework sont en anglais. Des notions d'anglais informatique sont donc souhaitables
(rassurez-vous, il n'y a rien de bien compliqué).

Axiom nécessite PHP 5.1 ou supérieur, MySQL, Apache et le mod_rewrite pour fonctionner. Il est possible toutefois de le déployer sur d'autres infrastructures, reportez-vous à la section **> aller plus loin** pour plus de détails.

Installation

Axiom se présente sous forme d'une archive Zip que vous pourrez trouver sur la page du projet http://code.google.com/p/php-axiom/

Pour l'installer, il suffit de décompresser l'archive dans le répertoire de votre choix. Au cours de cet article, nous supposeront que Axiom est déployé à la racine de votre serveur web sous le répertoire axiom (C:\wamp\www\axiom sous Windows ou /var/www/axiom sous Linux.)

Vous pouvez également déployer le Framework où bon vous semble sur votre serveur web tant que vous paramétrez les fichiers .htaccess fournis afin que la directive RewriteBase pointe vers le bon répertoire. Par exemple si vous avez déployé sous /mon-site/test/axiom/ (depuis le répertoire racine de votre serveur web), assurez vous de mettre les 3 directives RewriteBase à /mon-site/test/axiom/. Si vous avez choisi de déployer directement à la racine de votre serveur web, enlevez tout simplement les directives RewriteBase.

Pour que Axiom génère correctement les liens, il a besoin de connaître cet emplacement, vous trouverez dans le fichier /application/config/bootstrap/locale.php un paramètre 'base_url' qu'il faut définir avec la même valeur que la directive RewriteBase (ou / si Axiom est déployé à la racine).

Note: Axiom est fourni avec 3 fichiers .htaccess pré configurés, ils se trouvent à ces emplacements:
- /.htaccess
- /application/.htaccess
- /application/webroot/.htaccess

Attention: N'éditez pas ces règles si vous ne savez pas ce que vous faites.

Repertoires

Le framework présente l'arborescence suivante:
- application
    - config
        - bootstrap

        - controller
        - locale
            - langs
        - model
        - module
        - ressources
            - databases
            - temp
        - view
        - webroot
            - js
            - css
            - img
            - upload
- librairies
    - feeds
    - helpers

Le point d'entrée de l'application se situe dans /application/webroot/index.php. Les directives .htaccess ont été prédéfinies afin de permettre d'y accéder directement (la réécriture d'URL au niveau d'Apache doit être activée). C'est également sous ce répertoire que se trouveront les fichiers publiques de votre application, à savoir les scripts JavaScript, les images et les feuilles de style.

Les noms de ces répertoires sont assez intuitifs pour ceux qui connaissent déjà le pattern MVC. Toute la logique de l'application s'exprime dans /application/ cette logique est découpée en 3 ensembles:

- Model : les données persistantes de notre application (/application/model)
  ce sont nos objets de données qui la plupart du temps sont liés à la base de données
- View : les vues d'affichage de note application (/application/view)
  ce sont nos pages HTML, XML, JSon etc. qui mettent en forme les données pour les présenter à l'utilisateur.
- Controllers : les classes responsable de la prise en charge des interaction entre l'utilisateur et l'application et responsables du contrôle des données en entrée (/application/controller)
  Ce sont notamment eux qui vont valider les formulaires, récupérer les objets du modèle, effectuer tous les contrôles nécessaires et mettre le tout en forme avant l'appel des vues.

Le répertoire /application/locale contiens tous les paramètres de la locale - pour l'instant uniquement les traductions définies par des fichiers .ini sous /application/locale/langs. Axiom est capable de trouver automatiquement la langue appropriée à l'utilisateur en fonction des traductions définies dans ce répertoire. voir la section **> Multilinguisme.**

Le répertoire /application/config contiens tous les scripts PHP nécessaires à la configuration de notre application. Voir plus bas la section > Configuration.

Le répertoire /application/module contiendra tous les modules génériques, à l'installation il est vide. Voir plus bas la section > Créer un module pour Axiom.

Démarrage Rapide

A ce stade, vous êtes déjà capable d'utiliser le Framework, alors pourquoi ne pas l'essayer tout de

suite ?

Ouvrez votre navigateur et rendez-vous à la racine du serveur sur lequel vous avez déployé Axiom - par exemple http://localhost/axiom si vous êtes en local. Vous devriez atterrir sur la page d'accueil du Framework qui vous propose de commencer à jouer avec.

Fonctionnement

Avant de parler du fonctionnement interne d'Axiom, un mot sur qu'est ce qu'un Framework:

> *En programmation informatique, un Framework est un kit de composants logiciels structurels, qui servent à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel.*
>
> *En programmation orientée objet un Framework est typiquement composé de classes mères qui seront dérivées et étendues par héritage en fonction des besoins spécifiques à chaque logiciel qui utilise le Framework." (Wikipedia).*

Le mot important ici est structurel: un Framework est une structure, un cadre de travail dans lequel nous allons introduire notre besoins.
Il est important de noter que Axiom suit le principe de l'inversion de contrôle (voir http://fr.wikipedia.org/wiki/Inversion_de_contr%C3%B4le) ce qui signifie que c'est lui qui prends en charge le déroulement de l'application.

Axiom suit le principe du MVC pour traiter les requêtes des utilisateurs et procède par étapes:

1. Toutes les requêtes arrivent par le même point d'entrée (/application/webroot/index.php)
2. Le routeur va chercher à déterminer la route appropriée pour cette requête (voir la section > **routage**)
3. Le contrôleur correspondant à cette route va être invoqué
4. Le contrôleur renvoie un jeu de résultats pour cette requête
5. La vue est invoquée et on lui injecte des données renvoyées par le contrôleur

Par exemple:

1. L'utilisateur demande de voir les derniers articles publiés
2. Le routeur va invoquer le contrôleur chargé de traiter les articles
3. Le contrôleur va invoquer à son tour des classes modèle pour trouver les informations nécessaires sur la base de données
4. Le contrôleur va renvoyer les articles correspondants
5. Ces articles seront passés à la vue qui se chargera de mettre en forme le HTML servant à les afficher.

Chacun son rôle. L'intérêt principal de cette architecture réside dans sa flexibilité: vous pouvez changer le comportement à quelque niveau que ce soit sans obligatoirement impacter l'ensemble de l'application.
Imaginez que vous décidiez de changer de moteur de bases de données et abandonner MySQL pour MongoDB, avec Axiom vous n'avez qu'a appliquer vos changements dans la couche de modèle et le reste de votre application restera inchangé. De la même façon, vous pouvez changer l'affichage du tout au tout sans toucher au comportement de l'application.

Configuration

Avant toute autre action, Axiom va charger en mémoire sa configuration définie dans /application/config/bootstrap.php
On appelle cette étape le bootstrap (ou amorçage), c'est elle qui va définir et charger les paramètres de notre application. Elle remplace le traditionnel fichier config.php qu'on avait l'habitude de mettre un peu partout.

Note: Les classes d'Axiom sont généralement configurées automatiquement donc l'appel de la méthode setConfig sans paramètres (soit le paramétrage par défaut) suffit dans la plupart des cas. Les exemples ci-dessous montrent des cas spécifiques d'utilisation ou vous voudriez un paramétrage particulier.

Important: Ne changez pas les paramètres de configuration si vous n'êtes pas sûrs de ce que vous faites. Votre application pourrait ne plus fonctionner correctement.

Le bootstrap se décline en plusieurs sous-aspects:
- les paramètres globaux et la configuration de PHP (/application/config/bootstrap/settings.php)
Cette section comporte tous les ini_set, les définitions de constantes et les ajouts de fonctions manquantes à pHP 5.1 nécéssaires au bon fonctionnement d'Axiom.

- L'autoloader pour le chargement automatique des classes (/application/config/bootstrap/autoload.php)
Cette section paramètre et lance l'autoloader, c'est à dire la classe responsable du chargement dynamique des autre classes du système.
Par défaut, l'autoloader charge les classes d'Axiom (les librairies, les classes modèles, les controlleurs) mais il est possible de spécifier manuellement des chemins supplémentaires pour charger davantages de classes (celles de PEAR par exemple).

Note: L'autoloader utilise l'include path pour le chargement des classes, si votre directive include_path de php.ini (ou telle que définie dans un .htaccess) contiens déjà les paths des classes que vous souhaitez utiliser, vous n'avez pas besoin de les spécifier explicitement à l'autoloader.

Note: il est possible d'ajouter des répertoires à l'autoloader même après qu'il ait démaré (Autoloader::start) en utilisant Autoloader::add, mais vous devrez démarer à nouveau l'autolaoder pour que les nouveaux paths soient pris en compte. A priori pour un usage classique, seuls la connexion à la base de données et les routes doivent être configurés.

Note: Les paths que vous spécifiez manuellement viennent s'ajouter au paths par défaut de l'autoloader (qui charge nativement les chemins de Axiom).

Exemple:
***
```
// Ajouter un chemin
Autoloader::add('/var/www/php/lib/');

// Ajouter plusieurs chemins
Autoloader::setConfig(array(
  'paths' => array('/var/www/php/lib/', '/var/www/php/common/lib/');
));
```

```
// Démarer (ou redémarer) l'autoloader
Autoloader::start();

// Arrêter l'autoloader
Autoloader::stop();
```
***

- Les sessions (/application/config/bootstrap/session.php)
Cette section paramètre l'objet qui caractérise votre session ($_SESSION) pour un usage immédiat ou ultérieur. Vous pouvez notamment définir un nom de session différent
ou un offset de $_SESSION à utiliser pour éviter les collisions avec d'autres application PHP hébergées sur le même serveur.

Exemple:
***
```
// Définir un index de session à utiliser
Session::setConfig(array('index' => 'foobar')); // toutes les données enregistrées sur la sessions seront dans $_SESSION['foobar']

// Définir un nom de session (voir session_name)
Session::name('foobar');

// Démarer la session
Session::start();

// Détruire la session
Session::destroy();
```
***

- Les langues et la locale (/application/config/bootstrap/locale.php)
Ces section définis les langues disponibles, la langue par défaut (auto est acceptée) et l'URL racine de l'application nécéssaire à la construction automatique des liens.

Exemple
***
```
// Configurer les langues et les dates
Lang::setConfig(array(
  'locale' => 'auto', // déterminé par l'user agent
  'locales' => array('en', 'fr'),
  'date_format' => 'd/m/y H:i:s', // voir date()
  'base_url' => '/mon-application/', // correspond au répertoire sous lequel vous avez déployé Axiom en partant de la racine du serveur web
));

// Définir la langue par défault et recharger les traductions
Lang::setLocale('fr', true);

// Précharger les langues
Lang::loadLanguage();
```
***

- La connection avec la base de données (/application/config/bootstrap/connection.php)

Cette section paramètre l'objet PDO responsable de la connexion avec la base de données (MySQL, PostgreSQL, Oracle etc.)

Exemple:
***
```
// Définir les paramètres de connection
Database::setConfig(array(
    'host' => 'localhost',
    'database' => 'axiom',
    'username' => 'root',
    'password' => 'mot-de-passe',
    'type' => 'mysql',
));

// Définir directement le DSN
Database::setConfig(array(
  'dsn' => 'mysql:dbname=axiom;host=localhost',
  'username' => 'root',
  'password' => 'mot-de-passe',
));

// Ouvrir la connexion
Database::open();
```
***

Note: l'application refusera de démarer si la connection avec la base échoue. Pour prévenir ce comportement il est possible de passer un paramètre booléen à
Database::open pour qu'il renvoie renvoie false au lieu d'arrêter l'application.

- les routes (/application/config/bootstrap/routes.php)
Cette section va vous permettre de personnaliser les routes de votre application et de paramétrer le routeur. Une route fait pointer une URL sur un couple controlleur /
action à éxécuter.
Au moins une route par défaut doit être définie dans cette section.

Voir la section > Routage pour plus d'informations.

Exemple:
***
```
// Définir une route par défaut
Router::addRoute('default', array('index'));

// Définir une route personnalisée
Router::addRoute('~^((?<lang>[[:alnum:]]{2})/)?blog/(?<date>[^/]+)/?$~', array('blog', 'articles'));
```
***

Note: les paramètres nommés dans les expressions régulières définies ainsi dans le routeur seront automatiquement considérés comme des paramètres GET que le controlleur
pourra lire avec l'objet Request (voir ci après). Dans l'exemple, la paramètre date pourra être lu dans un contrôleur par self::$_request->date.

- les modules (/application/config/bootstrap/modules.php)

Cette section permet de paramétrer le gestionnaire de modules.

Exemple:
***
```
// Définir le chemin des modules
ModuleManager::setConfig(array(
  'module_path' => '/var/www/my_modules/',
));

// Determiner si un module existe
ModuleManager::exists('blog');

// Charger un module
if (ModuleManager::exists('blog'))
  ModuleManager::load('blog');
```
***

- les vues (/application/config/bootstrap/views.php)
Cette section vous permet de définir les comportements des vues, leur types, et les layouts.

Voir la section > Vues

Exemple:
***
```
// Configurer le layout manager
ViewManager::setConfig(array(
   'header' => 'html', // définir le format de sortie par défaut
   'layout_file' => 'default', // défini le fichier de layout
   'layout_content_var' => 'content', // définir le nom de la variable à utiliser dans le layout pour le contenu
));

// Ajouter des variables globales aux vues
// Ces variables pourront être vues dans le layout et dans toutes les vues, elles sont en quelque sorte globales au vues mais ne sont pas visibles dans les contrôleurs
ViewManager::addLayoutVars(array(
   'lang' => Lang::getLocale(),
   'description' => 'Axiom est un framework léger comme l'air',
   'keywords' => array('axiom', 'framework'),
   'title' => 'Ma première application avec Axiom',
));
```
***

- Les flux RSS / Atom (/application/config/bootstrap/feed.php)
Cette section vousp permet de configurer le gestionnaire de flux XML RSS ou ATOM.

Voir la section > Flux Rss / Atom

Exemple:
***
```
// Définir le format des flux
Feed::setConfig(array(
```

```
    'type' => 'Rss', // ou 'Atom'
));

// Définir les méta-informations du flux
Feed::setMetaInf(array(
    'title' => 'Axiom Feed',
    'date' => date('r'),
    'author' => array(
            'name' => 'Benjamin DELESPIERRE',
            'mail' => 'benjamin.delespierre@gmail.com'),
    'lang' => Lang::getLocale(),
    'description' => 'Axiom Generic Feed',
));
***
```

Routage

Avant de se lancer dans l'écritue de nos contrôleurs et de nos vues, il est nécéssaire de comprendre comment Axiom détermine les actions à effectuer lors d'une requête.
On appelle cette démarche le routage car elle consiste à trouver un route à emprunter lors d'une requête par analogie avec le routage réseau.

Axiom utiise nativement les URL réécrites, vous n'avez donc aucunement besoin de les décrire à nouveau avec des directives RewriteRule au niveau d'Apache.

- Routage automatique

Par défaut, Axiom attends de l'utilisateur deux paramètres: la route et l'action. Ces paramètres sont passés par l'URL sous la forme
http://mon.site.com/axiom/ln/route/action (le paramètre de langue "ln" est optionnel, voir la section > Multilinguisme). La réécriture d'URL transforme cette requête en
http://mon.site.com/axiom/index.php?url=ln/route/action.

Ce paramètre "url" sera ensuite découpé par le routeur pour trouver automatiquement le contrôleur et la méthode correspondants.
- Si aucune route n'a pu être trouvé, le routeur va rediriger l'utilisateur sur une erreur 404 en invoquant le contrôleur d'erreur.
- Si aucune route n'est demandé, le routeur invoquera le contrôleur par défaut tel que défini dans sa configuration (voir la section > Configuration).
- Si aucune action n'est spécifiée, la méthode index sera éxécutée (d'ailleurs, tous les contrôleurs doivent impérativement implémenter cette méthode).

Par exemple: la route /blog/articles va invoquer BlogController::articles.

- Routage avancé

Il est possiblee de faire en sorte qu'Axiom "reconnaisse" des URL définies par des expressions régulières pour déterminer une route. Par exemple, on peut faire en sorte
que http://mon.site.com/axiom/articles/2011-09-23/ emprunte la route blog avec l'action articles avec un paramètre date supplémentaire.
Pour que le routeur "reconnaisse" ainsi les URL, il faut lui spécifier, grace à Router::addRoute, une

série d'expressions régulières qui serviront à les identifier et qui
seront attachées à un tableau qui contiens le nom du contrôleur à invoquer et l'action à effectuer. Le routeur utilise la syntaxe PCRE pour ces expressions, si vous
n'êtes pas familier des expressions régulières, je vous suggère de lire cet article en préambule: XXXXXXX.

Important: Cette configuration DOIT IMPERATIVEMENT se faire dans la section "routes" du bootstrap car une fois la configuration des routes chargées, elle ne peut plus
être ultérieurement modifée*. Cela signifie également qu'un module ne peut pas importer ses routes, elles doivent systématiquement être définies dans le bootstrap (voir
la section > Créer des modules pour Axiom).

Exemple:
***
```
// Une route qui lie /xx/blog/ à ArticlesController::index
Router::addRoute('~^((?<lang>[a-z]{2})/)?blog/?$~', array('articles'));

// Une route qui lie /xx/news/jj-mm-aaaa à ArticlesController::view
// ici le paramètre date sera automatiquement extrait et ajouté aux paramètres de la requête
Router::addRoute('~^((?<lang>[a-z]{2})/)?news/(?<date>[0-9]{2}\-[0-9]{2}\-[0-9]{3}/?$~',
array('articles', 'news');
```
***

On peut en quelque sorte parler d'URL rewriting interne en PHP. l'avantage de cette technique est qu'elle vous rends maître de la réécriture au niveau de PHP et non au
niveau d'Apache, ce qui vous permet en somme de la paramétrer comme bon vous semble.

Les Contrôleurs

Les contrôlleurs sont les points névralgiques des applications MVC, ce sont eux qui sont responsables de trouver les données et de les traiter avant l'affiche. C'est là
que va se situer la machinerie de notre application.

Les contrôleurs sont des classes dont toutes les méthodes sont statiques - pour des raisons de performances (en PHP les méthodes statiques sont invoquées 4x plus vite que
les méthodes d'instance). De plus, nous n'aurons jamais besoin d'une instance de contrôleur (et encore moins de plusieurs instances).

Dans Axiom, tous les contrôleurs doivent impérativement hériter de la classe abstraite BaseController:
***
```
/**
 * Base Controller Abstract Class
 *
 * @abstract
 * @author Delespierre
 * @version $Rev: 22988 $
 * @subpackage BaseController
 */
abstract class BaseController {
```

```
    /**
     * Request object
     * @var Request
     */
    protected static $_request;

    /**
     * Response object
     * @var Response
     */
    protected static $_response;

    /**
     * Init the controller
     * @param Request $request
     * @param Response $response
     * @return void
     */
    public static function _init (Request &$request, Response &$response) {
        self::setRequest($request);
        self::setResponse($response);
    }

    /**
     * Set the request object
     * @param Request $request
     * @return void
     */
    final public static function setRequest (Request &$request) {
        self::$_request = $request;
    }

    /**
     * Set the resoibse object
     * @param Response $response
     * @return void
     */
    final public static function setResponse (Response &$response) {
        self::$_response = $response;
    }

    /**
     * Index method
     * @abstract
     */
    abstract public static function index ();
}
***
```

Cette classe comporte deux méthodes importantes:
- _init qui est toujours lancée avant que l'action du contrôleur ne soit executée
- index que les classes filles de BaseController doivent implémenter

Le rôle principal de la méthode _init est, comme son nom l'indique, d'initialiser le contrôleur. Pour ce faire, on passe au contrôleur les objets Request et Response qui
nous serviront;
- A réccupréer les paramètres fournis par l'utilisateur ($_POST, $_GET etc.)
- A mettre en forme la réponse (vue à invoquer, format de la vue, messages d'erreurs ou d'avertissement etc.)
Basiquement ces deux instances sont des dictionnaires de données que vous pouvez manipuler en utilisant respectivement self::$_request ou self::$_response dans les
méthodes du contrôleur.

Exemple:
***
```
// Récupérer un paramètre
if ($id = self::$_request->id) // renverra null (sans erreur) si le paramètre n'est pas fourni
  $article = new Article($id);

// Définir une variable pour une vue
self::$_response->variable = $valeur;
```
***

Les contrôleurs pevent renvoyer des données de deux manières possibles:
- soit en définissant directement les champs de l'objet Response comme mentionné ci-dessus
- soit en renvoyant tout simplement un tableau de variables
En ce qui concerne la seconde alternative (sans doute la plus pratique dans la plupart des cas), il est préférable d'utiliser compact plutôt que de générer un tableau
associatif pour des raisons pratiques: compact ne renverra pas d'erreur si la variable qu'on essaie d'adresser n'existe pas (voir l'exemple ci dessous).

Exemple:
***
```
// Renvoyer les données avec un tableau
return array('variable' => $valeur, 'autre_variable' => $autre->valeur);

// Renvoyer les données avec compact
return compact('variable', 'autre_variable');
```
***

Note: les vues ne peuvent pas lire les données de l'objet Request et il est incorrect qu'elles utilisent $_REQUEST. Les vues doivent se contenter des variables renvoyées
par le contrôleur. Voir la section > Vues.

Comme mentionné plus haut, si aucune action n'est spécifiée dans l'URL ou dans la définition de la route, c'est l'action index qui sera automatiquement invoquée.
La méthode _init sera pratique si vous souhaitez par exemple initialisée une donnée qui sera utilisée par toutes les méthode du contrôleur et vous évite ainsi de devoir
répliquer ce traitement autant de fois que vous avez de méthodes. C'est typiquement le cas d'un contrôleur qui a besoin de manipuler des données de session, dans ce cas,
on initialisera l'objet de session dans la méthode _init.

Important: Si vous créez un contrôleur sans définir la méthode index, vous vous retrouverez devant une page blanche en tentant d'y accéder. N'oubliez pas de la définir.

Exemple de contrôleur:
***
```
class ArticleController extends BaseController {

  public static function index () {
    $articles = Article::getArticles();
    return compact('articles);
  }

  public static function news () {
    $date = self::$_request->date ? self::$_request->date : date('d/m/Y');
    $articles = Article::getArticlesFromDate($date);
    return compact('articles');
  }

  public static function view () {
    if ($id = self::$_request->id) {
      $article = new Article($id);
    }
    return compact('articles');
  }

  public static function comment () {
    if (!self::$_request->c_name || !self::$_request->c_body)
          self::$_response->addMessage('Le nom et le corp du commentaire sont mandataire');
    else {
      if ($id = self::$_request->id) {
        $article = new Article($id);
        $article->addComment(new Comment(self::$_request->c_name, self::$_request->c_body));
      }
    }
    return self::view();
  }
}
```
***

Note: les contrôleurs (ou les modèles qu'ils invoquent) sont susceptibles de lever des exceptions. Si cela se produit et que l'exception n'est pas catchée au niveau du
contrôleur, c'est le routeur qui à invoqué le contrôleur qui attrapera cet erreur et reroutera l'utilisateur sur une erreur 500.

Le modèle

Les classes de modèle sont destinées au dialogue avec le système de persistence de données (base de données, flux XML, fichiers etc.) Axiom vous fournit une classe
abstraite Model pour manipuler une entité du SGBD (ce qui représente une vaste majorité des usages), à savoir une ligne d'une table sur un SGBD. On peut parler ici d'ORM
léger car il vous permet de convertir des données relationnelles en objets PHP - on qualifiera cet ORM de léger car il ne s'agit pas d'une couche complexe faisant
intervenir une connaissance détaillée du modèle au niveau de l'application comme c'est le cas

notamment avec Doctrine par exemple.

La classe modèle à besoin de connaitre le champ utilisé pour la clé primaire de la table qu'il lit, par défaut, Model suppose que ce champ est nommé 'id' mais il est
possible de spécifier cette valeur en surchargant la propriété protégé Model::$_id_key.

```
***
/**
 * Model Base Class
 *
 * @abstract
 * @author Delespierre
 * @version $Rev: 22988 $
 * @subpackage Model
 */
abstract class Model {

    /**
     * Model key id
     * @var string
     */
    protected $_id_key = 'id';

    /**
     * Model's data
     * @var array
     */
    protected $_data = array();

    /**
     * Statements cache
     * @var array
     */
    protected $_statements = array();

    /**
     * Initialize a model statement (part of CRUD)
     * @abstract
     * @param string $statement
     * @return PDOStatement
     */
    abstract protected function _init ($statement);

    /**
     * Default constructor
     * @param mixed $id
     * @throws RuntimeException
     */
    public function __construct ($id = null) {
        if ($id !== null && $id !== false && !$this->find($id))
            throw new RuntimeException("Cannot instanciate model");
    }
```

```
/**
 * __sleep overloading
 * @return array
 */
public function __sleep () {
    return array('_id_key', '_data');
}

/**
 * Getter
 * @param string $key
 * @return mixed
 */
public function __get ($key) {
    return isset($this->_data[$key]) ? $this->_data[$key] : null;
}

/**
 * Setter
 * @param string $key
 * @param mixed $value
 * @return void
 */
public function __set ($key, $value) {
    $this->_data[$key] = $value;
}

/**
 * __isset overloading
 * @param string $key
 * @return boolean
 */
public function __isset ($key) {
    return isset($this->_data[$key]);
}

/**
 * Get internal data
 * @internal
 * @return array
 */
public function getData () {
    return $this->_data;
}

/**
 * Retrieve method
 * Will return false in case of error
 * @param mixed $id
 * @return Model
 */
```

```
public function find ($id) {
    if (!$this->_init("retrieve"))
        throw new RuntimeException("Cannot initialize " . __METHOD__);

    if ($this->_statements['retrieve']->execute(array(":{$this->_id_key}" => $id))) {
        if ($this->_statements['retrieve']->rowCount()) {
            $this->_data = $this->_statements['retrieve']->fetch(PDO::FETCH_ASSOC);
            return $this;
        }
    }
    return false;
}

/**
 * Create methode
 * Will return false in case of error
 * @param array $data
 * @throws RuntimeException
 * @return Model
 */
public function create ($data) {
    if (!$this->_init("create"))
        throw new RuntimeException("Cannot initialize " . __METHOD__);

    if ($this->_statements['create']->execute(array_keys_prefix($data, ':'))) {
        $id = Database::lastInsertId();
        return $this->find($id);
    }
    return false;
}

/**
 * Update method
 * @throws RuntimeException
 * @return boolean
 */
public function update ($data = array()) {
    if (!$this->_init("update"))
        throw new RuntimeException("Cannot initialize " . __METHOD__);

    if (!empty($this->_data)) {
        $inputs = array_merge($this->_data, array_intersect_key($data, $this->_data));
        return $this->_statements['update']->execute(array_keys_prefix($inputs, ':'));
    }
    return false;
}

/**
 * Delete method
 * @throws RuntimeException
 * @return boolean
 */
```

```
    public function delete () {
        if (!$this->_init("delete"))
            throw new RuntimeException("Cannot initialize " . __METHOD__);

        if (!empty($this->_data))
        return $this->_statements['delete']->execute(array(":{$this->_id_key}" => $this->_data[$this->_id_key]));
        return false;
    }
}
***
```

Note: La classe modèle utilise la classe Database (simple Singleton PDO) pour accéder à la base de données. Voir la section > Configuration.

Important: il est de la responsabilité du programmeur de connaitre son modèle de données et de créer un jeu d'objet adéquat. Contrairement à Symphony, Axion ne génèrera pas ces classes pour vous.

- le CRUD

Cette classe prends en charge les méthodes CRUD (Create, Retrieve, Update, Delete), qui sont exprimées respectivement par Model::create, Model::find, Model::update et Model::delete, nécéssaires à n'importe quel modèle en relation avec le SGBD.
On remarque que chaque action du modèle provoque un appel à la méthode _init chargée de construire la requête SQL à éxécuter (représentée par un objet PDOStatement). Bien évidement, Model seule ne peut pas connaitre ces requêtes, elle va donc demander aux classes filles de surcharger cette méthode protégée _init. Si l'init échoue, la classe Model lève une RuntimeException.

Il est également possible de surcharger ces émthodes CRUD pour leur affecter un comportement particulier. On rappelle qu'il est toujorus possible dans ce cas de se servir du mot clé parent pour appeller les méthodes de la mère.

Exemple avec une classe File:

```
***
/**
 * File Model
 *
 * @author Delespierre
 * @version $Rev: 22988 $
 * @subpackage File
 */
class File extends Model {

    protected function _init ($statement) {
        if (isset($this->_statements[$statement]))
            return $this->_statements[$statement];

        switch ($statement) {
            case 'create':
                $query = 'INSERT INTO `ax_files`
```

```
(`filename`,`path`,`mime_type`,`upload`,`ax_users_id`) '.
                    'VALUES (:filename,:path,:mime_type,:upload,:ax_users_id)';
                break;
            case 'retrieve':
                $query = 'SELECT * FROM `ax_files` WHERE `id`=:id';
                break;
            case 'update':
                $query = 'UPDATE `ax_files` SET
`filename`=:filename,`path`=:path,`mime_type`=:mime_type,`upload`=:upload,'.
                    '`ax_users_id`=:users_id WHERE `id`=:id';
                break;
            case 'delete':
                $query = 'DELETE FROM `ax_files` WHERE `id`=:id';
                break;
            default:
                throw new RuntimeException("$statement is unexepected for " . __METHOD__);
        }

        return $this->_statements[$statement] = Database::prepare($query);
    }

    public function delete () {
        if (!parent::delete())
            throw new RuntimeException("Cannot delete File database record");

        return unlink($this->path);
    }

    public static function getFiles ($search_params = array(), $order = array('mime_type', 'upload')) {
        $query = "SELECT * FROM `ax_files`";

        if (!empty($search_params)) {
            $pieces = array();
            foreach ($search_params as $key => $value)
                $pieces[] = "`$key`=:$key";
            $query .= " WHERE " . implode(' AND ', $pieces);
        }

        $query .= " ORDER BY `" . implode('`,`', $order) . "`";

        $stmt = Database::prepare($query);
        if ($stmt->execute(array_keys_prefix($search_params, ':'))) {
            $file = new self;
            $stmt->setFetchMode(PDO::FETCH_INTO, $file);
            return new PDOStatementIterator($stmt);
        }
        return false;
    }

    public function getSize () {
        if (empty($this->_data))
            throw new RuntimeException("Malformed instance");
```

```
        if (file_exists($this->path))
            return filesize($this->path);
        return false;
    }

    public function getUser () {
        if (empty($this->_data))
            throw new RuntimeException("Malformed instance");

        return new User($this->ax_users_id);
    }

    public function getType () {
        if (empty($this->_data))
            throw new RuntimeException("Malformed instance");

        $mime = $this->mime_type;
        return substr($mime, 0, strpos($mime, '/'));
    }

    public function getSubtype () {
        if (empty($this->_data))
            throw new RuntimeException("Malformed instance");

        $mime = $this->mime_type;
        return substr($mime, strpos($mime, '/') +1);
    }
}
***
```

On note par exemple que la méthode File::delete provoque à la fois la suppression du tuple en base de données ainsi que la suppression effective du fichier.
On remarquera qu'on à étendu les fonctionnalité de Model pour lui fournir des méthodes spécifiques à File comme File::getUser qui renvoie l'utilisateur propriétaire du
fichier. En définissant ainsi nos relations (au sens SGBD du terme) par des méthodes, il devient possible de "naviguer" dans le modèle de données naturellement.

Les donnnées du modèle sont accessibles directement, exactement comme s'il s'agissait de propriétés publiques, au travers des méthodes magiques __get et __set.

Exemple:
```
***
// Créer
$user = new User;
$user->create(array(
  'name' => 'Delespierre',
  'surname' => 'Benjamin',
  'password' => md5('test'),
));

// Lire
```

```
$user = new User($id_user);
echo "Bienvenue {$user->name} {$user->surname}";

// Ecrire
$user->password = md5('foobar');
$user->update();
// équivalent à
$user->update(array('password' => md5('foobar')));

// Supprimer
$user->delete();
***
```

Note: vous pouvez obtenir une représentation sous forme de tableau des données de l'objet en utilisant la méthode getData.

- Serialization

Il est également possible de déposer les objets modèles sur session pour un usage ultérieur, la méthode magique __sleep permet de les sérialiser sans encombre:

```
***
// Poser
$_SESSION['user'] = new User($id_user);

// Réccupérer
$user = & $_SESSION['user'];
***
```

- Traitement par lots

Il est régulièrement nécéssaire de réccupérer un ensemble d'objets, comme par exemple une liste d'articles. Plutôt que faire une boucle pour réccupérer successivement
toutes les instances d'articles présentes en base, Axiom vous fournis un Iterateur pour manipuler une collection d'objets modèle: PDOStatementIterator.

Grâce à cette classe nous allons pouvoir nous servir d'une instance vide unique de n'importe quelle classe modèle comme d'un curseur sur une collection.

Exemple avec la méthode Article::getArticles qui renvoie tous les articles présents:

```
***
class Article extends Model {

  // ...

  public static function getArticles () {
    $query = "SELECT * FROM `ax_articles`;

    $stmt = Database::prepare($query);
    if ($stmt->execute) {
      $article = new self; // instance vide d'Article
      $stmt->setFetchMode(PDO::FETCH_INTO, $article); // on va indiquer au statement qu'il faut
se servir de $article lors de l'itération
```

```
      return new PDOStatementIterator($stmt);
    }
    return false;
  }

  // ...
}

$articles = Article::getArticles();

echo "Il y a " . count($articles) . " articles";
```
***

Note: PDOStatementIterator implémente l'interface Countable, c'est à dire que vous pouvez compter le nombre d'éléments qu'elle contiens en utilisant un count comme vous le feriez avec un simple tableau.

Note: PDOStatementIterator implémente l'interface seekable ce qui vous permet de "déplacer" le curseur de l'élément en cours en invoquant PDOStatementIteator::seek (mais pas de revenir en arrière car PDOStatement vous en empêche).

Note: PDOStatementIterator comprends deux méthodes first et last pour vous permettre de réccupérer le premier ou le dernier élément de la liste directement.

Vu que PDOStatementIterator étends IteratorIterator, il devient possible de le décorer avec d'autres itérateurs comme FilterIterator ou LimitIterator par exemple, ce qui est impossible avec un PDOStatement qui n'est que traversable.

Attention: il n'est pas possible de sérialiser (donc de mettre sur session) un PDOStatementIterator.

Attention: n'appellez jamais un modèle depuis une vue ! Cela aurait pour conséquence de perdre la traçabilité des données dans le MVC et vous finiriez par vous y perdre à ne plus savoir à quel niveau les modèles sont créés. De plus, les vues sont incapables d'effectuer des traitement sur les modèles et de prendre des directives en conséquence (par exemple éxécuter une action différente de l'action en cours si un modèle n'a pas été trouvé). Conservez toujours une séparation claire.

Les Vues

La vue vient complêter le triplet Modèle Vue Contrôleur. Une vue n'est ni plus ni moins qu'une mise en forme des données pour l'affichage. Dans le modèle en couches, les vues caractérisent une partie de la couche présentation, son rôle comme son nom l'indique est de présenter les données à l'utilisateur quel que soit le format.
Axiom comprends que les besoins d'une application web se limitent rarement aux seules page HTML et vous permet de définir plusieurs formats pour vos données (voir plus bas). Il est donc possible, pour une un couple route / action donnée de présenter les résultats en HTML, en JSon en XML afin de permettre une meilleure communication avec des programmes tiers. Par exemple, les vues JSon vous seront très pratique quand vous ferez communiquer PHP et JavaScript avec Ajax.

Les vues sont des scripts PHP dont le nom doit impérativement suivre le format suivant:

***
<nom>.<format>.php
***
Où:
- <nom> correspond au nom de l'action (en minuscule) c'est à dire une méthode d'un contrôleur
- <form> correspond au type de vue (html,json,xml,text)

Reprennons l'exemple des articles et écrivons la vue associée à ArticlesController::index soit la vue qui va afficher les articles:
/axiom/application/views/articles/index.html.php
***
```
<h1>Articles</h1>
<?php foreach ($articles as $article): ?>
  <div class="article">
    <h2><?=$article->title?></h2>
    <span><?=$article->author?></span>
    <p><?=$article->body?></p>
  </div>
<?php endforeach ?>
<span>Il y a <?=count($articles)?> articles publiés</span>
```
***

Nous pouvons aussi bien créér une vue JSon:
/axiom/application/views/articles/index.json.php
***
TODO
***

Note: Axiom n'utilise pas de moteur de template car PHP EST un moteur de templates !

Important: Il est possible que les vues fassent des appels implicites au modèles. Par exemple l'appel de $article->getComments va implicitement instancier des classes de
commentaires pour l'article donné. C'est tout à fait légal. En revanche, il est fortement déconseillé de faire des appels explicites (par exemple new Articles($id)),
c'est le rôle du contrôleur d'instancier les modèles pour une raison simple: la sécurité. Lors que vous êtes dans une vue, vous vous situez en fin de queue pour le
traitement de la requête, ce qui signifie que vous ne pourrez plus changer le flux des traitements. Par exemple, vous êtes incapables de retourner dans une autre action
d'un contrôleur (vous en êtes déjà sorti à ce stade) ou encore vous ne pouvez plus rerouter vers un contrôleur d'erreur si le besoin s'en fait sentir.

Aller plus loin

En cours de rédaction...

Conclusion

En cours de rédaction...