

Au cœur de Zend Framework 2

Préface

Je me souviens encore de mon premier site internet. J'avais dix ans et, à l'époque, FrontPage était mon meilleur allié. Peu importe qu'il produise du code non valide, qu'il soit complètement statique ou que son code soit vulnérable à la moindre faille. Finalement, je m'amusais bien et c'était le principal.

Évidemment, le Web est devenu au fil des années une industrie à part entière, de plus en plus influente et aujourd'hui, les développeurs du monde entier le savent bien, il faut être de plus en plus productif, écrire du code toujours plus évolutif, toujours plus performant, toujours plus sécurisé et... toujours dans le même laps de temps. À cet égard, l'arrivée des frameworks dans l'univers PHP a révolutionné notre manière de penser et concevoir le Web. Et, surtout, il a contribué à redonner une image crédible à ce langage, bien trop souvent terni par les mémoires du vénérable PHP 4.

À sa sortie en juin 2007, la première version stable du Zend Framework a largement contribué à cet essor. Contrairement à d'autres frameworks, il n'imposait rien, mais fournissait des briques réutilisables. Ce fut là sa grande force et l'une des raisons de son succès. Depuis, de l'eau a coulé. Le petit framework des débuts a grossi, s'est enrichi de nombreuses fonctionnalités et, douze versions majeures plus tard, il a atteint sa maturité mais, pour continuer à évoluer, il était nécessaire de le repenser de zéro.

C'est dorénavant chose faite, et cette réécriture complète du framework a, une fois de plus, été une véritable démonstration de force de la communauté open-source, tant les chiffres donnent le tournis : plus de 200 contributeurs, plus de 1000 forks, près de 14 000 tests unitaires écrits et près de 1800 pull requests. Le tout en un peu plus d'un an. C'était la première fois que je participais à une aventure open-source, en contribuant notamment largement au code du composant des formulaires, et je dois dire que ce fut une expérience très formatrice. Il n'y a pas de petites contributions, et que cela soit pour corriger un bogue, ajouter de la documentation ou créer une nouvelle fonctionnalité, c'est grâce à l'aide de tous que notre outil de travail est ce qu'il est aujourd'hui.

À ce propos, le livre que vous tenez entre les mains constitue la toute première ressource francophone consacrée à ce nouveau framework, et Vincent a réalisé un incroyable travail pour vous permettre non seulement de vous lancer dans l'ap-

prentissage de Zend Framework 2, mais également pour y comprendre les rouages internes (événements, injection de dépendances...), indispensables pour tirer parti au maximum de ce framework à la fois simple et complexe. Que vous soyez un actuel développeur de ZF 1 ou que vous ayez déjà une expérience avec ZF 2, chacun y trouvera son compte !

Michaël Gallego

Contributeur au Zend Framework 2

Administrateur du forum dédié à la communauté française du Zend Framework

Table des matières

Avant-propos	11
À qui s'adresse ce livre ?	11
Structure de l'ouvrage	12
Remerciements	13
CHAPITRE 1	
Le Zend Framework.....	15
Le Zend Framework 1	15
Le Zend Framework 2	15
PHP 5.3.3.....	15
Design pattern.....	16
Les axes d'amélioration.....	17
Le squelette de l'application	17
CHAPITRE 2	
Les autoloaders	19
L'autoloader standard	19
Le ClassMapAutoloader.....	24
La classe ModuleAutoloader.....	28
L'interface SplAutoloader	28
CHAPITRE 3	
Les évènements	31
Présentation des évènements.....	31
Le gestionnaire partagé	33
L'évènement wildcard	39
Le gestionnaire d'évènements global	40
Le type ListenerAggregate.....	40
Créer ses évènements personnalisés	43
L'objet de réponse.....	46
Stopper la propagation d'un événement.....	47
Gérer l'ordre des notifications.....	49
Les évènements du framework	50
CHAPITRE 4	
L'injection de dépendances.....	53
Configuration manuelle du composant Di	54

Configuration automatique du composant Di.....	56
Configuration semi-automatique du Di	59
Le gestionnaire d'instances	61

CHAPITRE 5

Le gestionnaire de services..... 65

La configuration du gestionnaire.....	65
Mise en œuvre	67
Le ServiceManager complémentaire du Di.....	77
La fabrique du Zend\ServiceManager\Di.....	77
La fabrique abstraite du Zend\ServiceManager\Di.....	79
Implémentation dans le framework.....	80

CHAPITRE 6

Les modules 83

Configuration et écouteurs	83
Chargement des modules	88
Initialisation de l'autoloader du module.....	96
Initialisation du module.....	97
Notification lors de l'initialisation de l'application.....	98
Configuration du gestionnaire de services.....	99
Nous remarquons que la clé « service_manager » est utilisée pour récupérer l'instance de l'objet étendant de ServiceManager qui sera alors configuré.	102
Initialisation des instances partagées	102
Mise à jour de la configuration des modules.....	102
Traitement postchargement	104
Récapitulatif du processus	108
Processus de chargement	108
Squelette de notre classe Module.....	109

CHAPITRE 7

Les configurations..... 111

Accès à la configuration	111
Configuration par environnement.....	112
Configuration par fichiers PHP.....	113
Configuration par héritage de fichier.....	114
Configuration par fichier INI	116
Configuration recommandée	117

CHAPITRE 8

Le router 119

Le router et la brique MVC	119
Les routes Http\Literal	124
Les routes Http\Regex	126
Les routes Http\Scheme	127
Les routes Http\Method	127
Les routes Http\Hostname	128
Les routes Http\Segment	129
Les routes Http\Wildcard	130
Les routes Http\Part	132
L'écouteur pour les modules.....	133
CHAPITRE 9	
Les contrôleurs	137
Le contrôleur <code>AbstractActionController</code>	137
Méthodes d'action par défaut	143
Les interfaces du contrôleur de base	144
Le contrôleur <code>AbstractRestfulController</code>	148
CHAPITRE 10	
Les aides d'action	151
L'aide d'action <code>Forward</code>	151
L'aide d'action <code>Redirect</code>	153
L'aide d'action <code>Url</code>	155
L'aide d'action <code>PostRedirectGet</code>	156
Création d'une aide d'action	157
CHAPITRE 11	
Le gestionnaire de plugins.....	159
La classe de base	159
Les gestionnaires du framework.....	161
CHAPITRE 12	
Les vues	165
Le gestionnaire de vues	165
Préparation des vues	166
Préparation des vues d'erreurs.....	168
Rendu de la vue.....	173
Construction des vues	173
Rendu des vues	177
Récapitulatif du rendu.....	191
Types de vue.....	193

Manipulation des vues	195
-----------------------------	-----

CHAPITRE 13

Les aides de vue 199

Le fil d'Ariane	200
Le sitemap	205
L'aide Layout et ViewModel	207
L'aide Partial	208
Le rendu d'élément HTML	210
Le rendu de JSON	211
L'aide Url	212
Création d'une aide de vue	213

CHAPITRE 14

Le cœur du framework 215

Initialisation des autoloaders	215
Le bootstrap	217
Le run	221

CHAPITRE 15

Les composants 229

Zend\Db	229
L'adaptateur de base de données	229
L'abstraction SQL	234
L'interface TableGateway	235
L'interface RowGateway	238
Zend\Cache	241
Les adaptateurs	241
Les plugins	242
Analyse des plugins	243
Le composant Zend\Cache\Pattern	246
Zend\Console	248
Utilisation basique de la console	248
Les classes d'interactions	250
L'abstraction de l'environnement	253
Le router pour la console	254
La vue pour la console	257
Zend\Crypt	259
Hachage à sens unique	259
Hachage symétrique	260
Zend\Form	261

Le formulaire, les fielsets et les éléments	261
Les fabriques et hydrateurs	262
Le composant Zend\InputFilter	267
Les aides de vue	270
 Zend\Mail.....	272
 Zend\Session.....	275
 Zend\Stdlib.....	278
La classe AbstractOptions.....	278
La classe ArrayUtils.....	280
La classe CallbackHandler.....	281
La classe ErrorHandler.....	284
La classe Glob	285
Les hydrateurs	285

CHAPITRE 16

 Les exceptions.....	291
--------------------------------	------------

CHAPITRE 17

 Cas d'utilisation	293
Concept et fonctionnalités	293
Présentation du projet.....	293
Les besoins du projet.....	294
Organisation et architecture	294
Personnalisation du chargement de modules.....	296
La restriction de chargement	296
L'implémentation du chargement à la demande.....	296
La configuration.....	299
Les contrôleurs de l'application.....	301
Les tâches automatisées	305
Implémentation.....	305
Utilisation du composant ZendService.....	307
Les modules du framework.....	308

CHAPITRE 18

 Contribuer au framework	309
Le planning	309
La gestion des versions	311
Le bug tracker	312
Corriger un bogue ou proposer une amélioration	313
Récapitulatif des étapes.....	314

Avant-propos

À qui s'adresse ce livre ?

L'ouvrage est technique et s'adresse à des développeurs informatiques ayant déjà une expérience de la programmation en PHP, ainsi que des connaissances de base du Zend Framework, que ce soit dans sa première ou deuxième version.

Nous essaierons de faire, tout au long de l'ouvrage, un parallèle entre la première et cette seconde version du framework afin de bien comprendre pourquoi et dans quel but les composants ont été réécrits, et ce que peuvent nous apporter les nouvelles fonctionnalités du framework. Bien que de nombreuses portions de code soient incluses et expliquées dans l'ouvrage, il est conseillé de lire les différents chapitres avec le code source du framework sous la main pour mieux comprendre les explications données. Afin d'éviter l'ajout de code non pertinent, certaines portions inutiles à la compréhension sont coupées et représentées par des crochets « [...] ». Cela peut être le cas pour la gestion de certaines erreurs ou traitements basiques par exemple.

Lors de la première partie du livre et l'explication du cœur du framework, nous prendrons comme exemple le squelette d'application proposé sur le compte github du zend framework : <https://github.com/zendframework/ZendSkeletonApplication>. Nous ferons peu de modifications au niveau du code du squelette d'application, mis à part pour des exemples précis. Les premiers chapitres sont consacrés au code du Zend Framework, ce qui nous permettra d'en maîtriser les concepts afin de pouvoir les mettre en œuvre lors de la deuxième partie avec le cas d'utilisation.

J'ai avant tout souhaité écrire ce livre pour témoigner l'intérêt que j'ai pour le Zend Framework et le PHP en général, j'espère que celui-ci répondra à vos attentes et vous permettra de mieux appréhender le fonctionnement du framework. La découverte et la compréhension en détail du framework m'ont aussi permis de pouvoir contribuer au code du Zend Framework 2 à travers l'implémentation de nouvelles fonctionnalités et la correction d'erreurs. J'espère donner aussi l'envie à d'autres développeurs de venir contribuer à l'amélioration de cet excellent framework. Vous trouverez au tout dernier chapitre les étapes de la contribution au Zend Framework.

Structure de l'ouvrage

L'ouvrage se divise en deux grandes parties. La première est consacrée à l'étude du cœur du Zend Framework, ce qui va nous permettre de comprendre les grands changements effectués lors de cette deuxième version. La deuxième partie présente un cas d'utilisation qui nous permet la mise en œuvre d'un exemple concret utilisant un grand nombre de composants.

- Le chapitre 1 est une introduction au Zend Framework 2, avec un petit historique sur la première version du framework. Ce chapitre permet de se faire rapidement une idée sur les améliorations apportées au Zend Framework 2, ainsi que les conditions requises afin de l'exploiter au mieux.
- Le chapitre 2 présente les autoloaders existants dans le framework, avec des exemples d'utilisation et l'explication de leur comportement détaillé. La maîtrise du chargement de nos classes permet de sensiblement améliorer les performances de nos applications Web, souvent peu optimisées sur cet aspect.
- Le chapitre 3 étudie le gestionnaire d'événements et l'implémentation de la programmation évènementielle au sein du Zend Framework. Cette partie est l'une des grandes nouveautés du framework, l'utilisation des événements permet l'amélioration de la flexibilité du framework.
- Le chapitre 4 présente le composant d'injection de dépendance qui permet aux développeurs de trouver ici les bonnes pratiques de programmation en découpant chaque objet et supprimant les dépendances dans le code.
- Le cinquième chapitre aborde le gestionnaire de services, nouveau composant du framework, responsable de la fabrication des objets du framework. Le gestionnaire peut être utilisé en complément du composant d'injection de dépendances.
- Le chapitre 6 présente une des autres nouveautés du framework : le fonctionnement des modules. Ceux-ci sont comparables à des briques individuelles prêtes à être intégrées de projet en projet.
- Le chapitre 7 fait le point sur les configurations, notamment sur leur fonctionnement général ainsi que sur l'utilisation de celles-ci pour chacun de nos modules.
- Le huitième chapitre nous présente le fonctionnement du router, l'optimisation de celui-ci et les différents types de routes existantes.
- Les contrôleurs seront abordés au chapitre 9 qui détaille le processus de distribution au sein du contrôleur, ainsi que la possibilité de jouer n'importe quelle action d'un contrôleur en le rendant distribuable.
- Le chapitre 10 nous propose de comprendre comment ont été refactorisées les aides d'action du framework. Plus performantes et plus simples à comprendre, les aides d'actions sont maintenant plus efficaces.
- Les gestionnaires de plugins sont présentés au chapitre 11, ce qui nous permet de voir le fonctionnement du chargement et de la gestion des instances de nos plugins.
- Les vues, leurs types et le fonctionnement de leurs rendus sont expliqués au

chapitre 12. Un gros travail de réécriture a été effectué par l'équipe du Zend Framework sur le mécanisme des vues.

- Les aides de vue et leurs optimisations sont présentés au chapitre 13.
- Le cœur du patron d'architecture MVC de Zend est étudié au chapitre 14 avec l'explication de nos points d'entrées et le processus de lancement de l'application.
- Plusieurs composants importants (bases de données, formulaires, etc.) sont détaillés au chapitre 15 afin d'en maîtriser leur utilisation.
- Le chapitre 16 passe en revue le nouveau système de gestion des types d'exceptions au sein du framework.
- Le dix-septième chapitre présente l'étude d'un cas d'utilisation afin de nous permettre la mise en application de tout ce que l'on a pu voir au sein des chapitres précédents.
- Le dernier chapitre explique en détail les étapes afin de pouvoir contribuer au Zend Framework : correction de bogue, création de fonctionnalité, suivi du calendrier de développement, etc.

Remerciements

Je tiens à remercier tous ceux qui m'ont aidé à rédiger cet ouvrage, que ce soit par leur relecture ou leur soutien :

- Clément Caubet pour sa magnifique couverture de livre ;
- Michel Di Bella pour le template et la mise en page du livre ;
- Yoann Gobert pour le design et l'intégration du site au-coeur-de-zend-framework-2.fr ;
- Christophe Mailfert et Frédéric Blanc pour la relecture technique du livre ;
- Claude Leloup et Fabien Le Bris pour leur relecture orthographique ;
- Michael Gallego pour sa préface ;
- au site developpez.com et ainsi qu'à ses membres pour leurs relectures et commentaires ;
- Marie qui m'a soutenu et écouté pendant les six mois d'écriture ;
- l'équipe du Zend Framework pour leur très bon travail, ce qui fait que ce livre existe ;
- vous qui venez de l'acheter et qui lisez jusqu'à la dernière ligne des remerciements.

1

Le Zend Framework

Le Zend Framework 1

La première révision du framework était la version 0.1, sortie en 2006 avant qu'une première version stable voie le jour en juillet 2007 après plus d'un an et demi de développement. Cette première version n'intégrait pas toutes les fonctionnalités que l'on connaît maintenant. De nombreux composants majeurs, comme le Zend_Layout, sont sortis avec la version 1.5 et d'autres comme le Zend_Application, avec la version 1.8. La version 1.5 a marqué un tournant pour le Zend Framework, c'est une des versions qui ont apporté le plus de fonctionnalités. C'est d'ailleurs sur cette version que repose la certification Zend Framework 1.

Seulement, au fur et à mesure du temps, le Zend Framework, même s'il est compatible avec PHP 5.3, ne tire pas tous les avantages des dernières versions de PHP. La version 1.11 du framework est considérée comme très stable et c'était donc l'occasion pour commencer de développer une nouvelle version qui casserait complètement la compatibilité avec la version précédente. C'est aussi l'occasion pour l'équipe de développement de mettre en place de nombreux patterns et bonnes pratiques pour les développeurs qui manquent actuellement au framework.

Le Zend Framework 2

PHP 5.3.3

Le Zend Framework 2 est pensé et écrit pour PHP 5.3, branche d'évolution majeure sortie en 2009. PHP 5.3 a intégré un grand nombre d'évolutions comme les espaces de nom, le garbage collector, les améliorations de la SPL (Standard PHP Library, bibliothèque standard de PHP qui fournit classes et interfaces pour résoudre des problèmes récurrents), etc. Le Zend Framework 2 utilise abondamment ces nouveautés, en particulier les espaces de noms qui sont omniprésents dans le framework où chaque classe possède son propre espace de nom. Le framework utilise aussi beaucoup les nouveautés de la SPL comme le GlobIterator ou la SplPriorityQueue qui permettent une meilleure compatibilité. Typiquement, ces classes et méthodes de la SPL sont régulièrement utilisées à travers la bibliothèque

standard du Zend Framework 2 pour homogénéiser le code au sein du framework. Il y a donc une première cassure avec le Zend Framework 1 où les espaces de noms ne pouvaient pas être utilisés pour garder une compatibilité avec les versions inférieures à PHP 5.3.

Au niveau de la branche 5.3, le cœur du langage, dont le Zend Engine, a été profondément remanié ce qui entraîne des gains de performance ainsi que des réductions de consommation mémoire assez importantes. Il n'est pas nécessaire de s'étendre plus sur les nouveautés de PHP 5.3, nous sommes aujourd'hui en version 5.4, la branche 5.3 ayant déjà trois ans, la documentation PHP sera plus précise à ce sujet pour ceux qui souhaitent découvrir les évolutions de cette version. Si la version 2 du framework est restée en développement pendant plus de deux ans, il semblerait que l'équipe de développement souhaite accélérer légèrement les sorties de versions majeures pour toujours bénéficier au maximum des dernières branches de PHP ainsi que les bonnes pratiques et design pattern. Pour venir discuter avec l'équipe, contributeurs ou autres développeurs Zend Framework 2, n'hésitez pas à venir sur le channel IRC dédié #zftalk.2.

Design pattern

Le patron de conception principal est le patron d'architecture MVC (modèle - vue - contrôleur), qui sépare le code d'une application entre le modèle (traitement des données et interaction avec la base de données), la vue (gestion des fichiers de vue et du rendu) et le contrôleur qui est le chef d'orchestre de l'application et va interagir avec les deux premiers composants pour injecter les données à la vue.

Dans cette nouvelle version du framework, la programmation par contrat est encore plus présente. Chaque classe abstraite et chaque composant indépendant implémentent une interface, ce qui va permettre de faciliter l'injection de dépendance. En effet, un objet injecté dans un autre objet devra respecter un contrat (implémenter une interface) afin de pouvoir s'assurer que celui-ci va pouvoir répondre aux attentes de l'objet qui le reçoit. Cela permet d'étendre les possibilités et de supprimer les dépendances de son code.

Comme nous le verrons, l'injection de dépendances est très présente dans le Zend Framework 2 et autorise la suppression des dépendances codées en dur, ce qui permet aux développeurs d'améliorer la qualité de leur code.

Une des grandes nouveautés du Zend Framework 2 est la programmation évènementielle. Le principe est d'offrir un système de communication au travers de tous les composants du framework sans que celui casse le principe du pattern MVC. La programmation évènementielle a été introduite pour faciliter les échanges entre composants et supprimer les mauvaises pratiques que pouvaient mettre en place les développeurs pour pallier ce manque de transversalité. De plus, les évènements permettent la suppression des hooks, trop présents dans la première version du framework. La programmation évènementielle est une manière de programmer largement répandue dans d'autres langages comme le Java ou l'ActionScript, Zend Framework y intègre les bonnes pratiques. Un chapitre est consacré aux évènements et aux gestionnaires d'événements.

Les axes d'amélioration

Les premiers axes d'amélioration du Zend Framework 2 concernent l'amélioration des performances qui devenaient de moins en moins bonnes au fur et à mesure des versions, ainsi que l'amélioration de la courbe d'apprentissage du framework.

L'utilisation des méthodes magiques est en cause sur les baisses de performance, dans le Zend Framework 1, la magie est utilisée à tout va. Le deuxième point d'amélioration est le système de chargement de classes qui reste trop gourmand, nous verrons que le Zend Framework 2 utilise un système de chargement beaucoup plus performant. Le framework veut aussi supprimer l'utilisation excessive de la directive « `include_path` » qui rend le chargement des classes beaucoup trop lent. Un nouveau système de résolution de nom de classe par tableau PHP décrivant le chemin pour chacune des classes utilisées en clé du tableau a été introduit. Ce chargement basé sur le couple clé/valeur devient alors beaucoup plus performant, même s'il reste la possibilité d'utiliser un chargement basé sur les préfixes, espaces de noms ou sur la directive « `include_path` ».

Au-delà des composants trop gourmands en termes de performances ou de temps d'apprentissage, la deuxième version du framework veut améliorer l'environnement du développeur en mettant l'accent sur la documentation et les exemples mis en œuvre. Par exemple, des classes comme le `Zend_Form` et ses décorateurs sont considérées comme trop complexes d'utilisation.

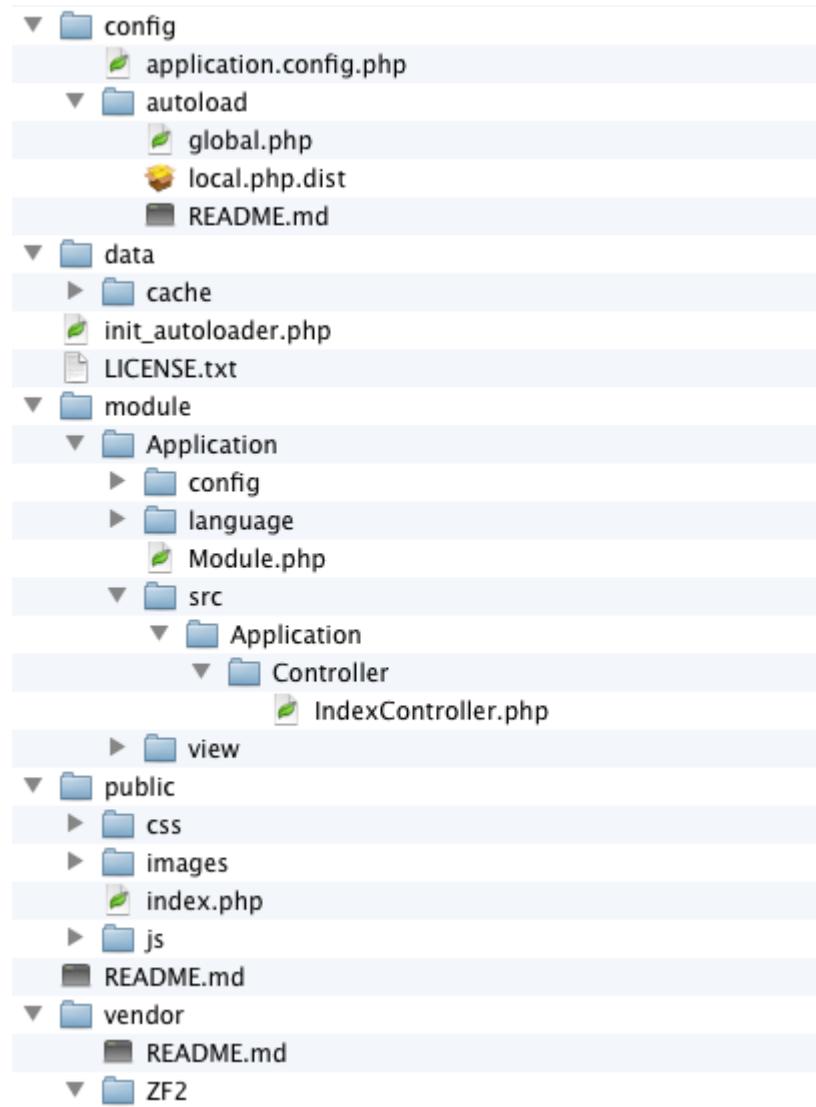
Afin de mettre en place toutes ces bonnes pratiques, les développeurs ont dû réécrire certains composants entièrement et en créer d'autres. Si le développement a paru long pour ses utilisateurs, la priorité de l'équipe était d'avoir un produit fini qui leur convienne parfaitement.

Nous allons maintenant voir comment mettre en place rapidement un squelette de projet sous Zend Framework 2, ce qui nous servira ensuite de base pour notre étude approfondie du framework.

Le squelette de l'application

Lors des chapitres suivants, tous nos exemples de configurations et de modules seront directement tirés du `ZendSkeletonApplication`, disponible sur compte github de ZendFramework : <https://github.com/zendframework/ZendSkeletonApplication>. Ce projet propose un squelette d'application fonctionnel afin de comprendre les principales étapes de la mise en place d'un projet sous Zend Framework 2.

Voici la hiérarchie des fichiers proposés :



Au fil des chapitres, lorsqu'une section sera traitée, un aperçu des fichiers concernés sera joint aux explications et chaque partie du cœur de Zend Framework passée en revue sera affichée dans les explications de l'ouvrage afin d'en faciliter la compréhension.

2

Les autoloaders

Dans la liste des axes d'amélioration du Zend Framework 2 figurent les autoloaders. Trop souvent basées sur la directive « `include_path` », les classes de chargements du Zend Framework 1 sont trop lentes. Si l'utilisation des préfixes de classe permet de rendre le chargement un peu plus performant, cette étape reste tout de même coûteuse en performances. De nouveaux autoloaders ont fait leur apparition pour rendre le processus encore plus rapide, faisons un tour d'horizon pour les présenter.

L'autoloader standard

L'autoloader standard, de type `Zend\Loader\StandardAutoloader`, permet de charger une classe suivant un préfixe de nom ou un espace de nom. Il est aussi possible de configurer l'autoloader standard afin d'utiliser la directive « `include_path` », ce qui serait préjudiciable en termes de performances.

Prenons un exemple d'utilisation avec le fichier d'initialisation des autoloaders qui l'utilise par défaut avec l'instruction :

```
init_autoloader.php
|| Zend\Loader\AutoloaderFactory::factory(array(
||   'Zend\Loader\StandardAutoloader' => array(
||     'autoregister_zf' => true
||   )
|| ));
```

L'autoloader standard est utilisé explicitement, mais sans argument, la fabrique instancie un objet de type `StandardAutoloader` par défaut comme nous le voyons depuis la fabrique :

```
Zend/Loader/AutoloaderFactory.php
|| const STANDARD_AUTOLOADER = 'Zend\Loader\StandardAutoloader';
```

Zend/Loader/AutoloaderFactory.php

```

public static function factory($options = null)
{
    if (null === $options) {
        if (!isset(static::$loaders[static::STANDARD_AUTOLOADER])) {
            $autoloader = static::getStandardAutoloader();
            $autoloader->register();
            static::$loaders[static::STANDARD_AUTOLOADER] =
$autoloader;
        }
        return;
    }
    [...]
}

```

La méthode « `getStandardAutoloader()` » crée une instance de cet objet :

Zend/Loader/AutoloaderFactory.php

```

protected static function getStandardAutoloader()
{
    if (null !== static::$standardAutoloader) {
        return static::$standardAutoloader;
    }
    $stdAutoloader = substr(strrchr(static::STANDARD_AUTOLOADER, '\\'), 1);
    if (!class_exists(static::STANDARD_AUTOLOADER)) {
        require_once __DIR__ . "/$stdAutoloader.php";
    }
    $loader = new StandardAutoloader();
    static::$standardAutoloader = $loader;
    return static::$standardAutoloader;
}

```

La méthode « `register()` » enregistre ensuite la fonction « `autoload()` » de l'objet courant, comme méthode de chargement, auprès de la SPL :

Zend/Loader/StandardAutoloader.php

```

public fonction register()
{
    spl_autoload_register(array($this, 'autoload'));
}

```

Le constructeur de l'autoloader standard enregistre par défaut l'espace de nom « Zend » si cela lui est demandé, comme dans l'initialisation des autoloaders, depuis le tableau d'options afin qu'il soit prêt à l'emploi pour le chargement de classes de la bibliothèque du framework :

Zend/Loader/StandardAutoloader.php

```

const AUTOREGISTER_ZF = 'autoregister_zf';

```

Zend/Loader/StandardAutoloader.php

```

public fonction setOptions($options)
{
[...]
foreach ($options as $type => $pairs) {
    switch ($type) {
        case self::AUTOREGISTER_ZF:
            if ($pairs) {
                $this->registerNamespace('Zend', dirname(__DIR__));
            }
            break;
    [...]
}
return $this;
}

```

Ceci explique la possibilité d'instancier ensuite un objet de l'espace de nom « Zend » sans difficulté. Plusieurs méthodes de cette classe permettent l'enregistrement de préfixes et espaces de nom :

Zend/Loader/StandardAutoloader.php

```

public function registerNamespace($namespace, $directory)
{
$namespace = rtrim($namespace, self::NS_SEPARATOR). self::NS_SEPARATOR;
$this->namespaces[$namespace] = $this->normalizeDirectory($directory);
return $this;
}

```

Zend/Loader/StandardAutoloader.php

```

public function registerPrefix($prefix, $directory)
{
$prefix = rtrim($prefix, self::PREFIX_SEPARATOR). self::PREFIX_SEPARATOR;
$this->prefixes[$prefix] = $this->normalizeDirectory($directory);
return $this;
}

```

Prenons d'autres exemples et ajoutons un dossier à la racine, nommé « autoload-dir », qui contient deux dossiers « Namesp/ » et « Prefix/ ». Le dossier « Namesp » possède un premier fichier Test.php :

Test.php :

```

namespace Namesp;
class Test{}

```

Le dossier « Prefix/ » contient un autre fichier Test.php :

Test.php

```

class Prefix_Test{}

```

Afin de charger ces deux classes, il nous suffit de récupérer l'autoloader standard et d'y ajouter les espaces de noms et préfixes dont on a besoin :

Utilisation des préfixes

```
|| $autoloader = Zend\Loader\AutoloaderFactory::getRegisteredAutoloader(
|| Zend\Loader\AutoloaderFactory::STANDARD_AUTOLOADER);
|| $autoloader->registerPrefix('Prefix_', __DIR__ . '/autoload-dir/
|| Prefix');

|| new Prefix\Test();
```

Utilisation des espaces de nom

```
|| $autoloader = Zend\Loader\AutoloaderFactory::getRegisteredAutoloader(
|| Zend\Loader\AutoloaderFactory::STANDARD_AUTOLOADER);
|| $autoloader->registerNamespace('Namesp', __DIR__ . '/autoload-dir/
|| Namesp');

|| new Namesp\Test();
```

Comme nous le remarquons, l'utilisation de la classe StandardAutoloader est très simple. Comme on l'a dit précédemment, il est aussi possible d'utiliser la directive « `include_path` » depuis la méthode « `setFallbackAutoloader()` » :

Utilisation de la directive « `include_path` »

```
|| set_include_path(implode(PATH_SEPARATOR, array(
||     realpath(__DIR__ . '/autoload-dir'), get_include_path()
|| )));
|| $autoloader = Zend\Loader\AutoloaderFactory::getRegisteredAutoloader(
|| Zend\Loader\AutoloaderFactory::STANDARD_AUTOLOADER);
|| $autoloader->setFallbackAutoloader(true);

|| new Namesp\Test();
```

Si le chargement de classe est activé avec la directive « `include_path` » et que des espaces de noms ou préfixes sont enregistrés, le framework leur donnera la priorité lors du chargement :

Zend/Loader/StandardAutoloader.php

```
|| const NS_SEPARATOR      = '\\\\';
```

Zend/Loader/StandardAutoloader.php

```
|| public function autoload($class)
|| {
||     $isFallback = $this->isFallbackAutoloader();
||     if (false !== strpos($class, self::NS_SEPARATOR)) {
||         if ($this->loadClass($class, self::LOAD_NS)) {
||             return $class;
||         } elseif ($isFallback) {
||             return $this->loadClass($class, self::ACT_AS_FALLBACK);
||         }
||     }
||     return false;
|| }
```

```

    if (false !== strpos($class, self::PREFIX_SEPARATOR)) {
        if ($this->loadClass($class, self::LOAD_PREFIX)) {
            return $class;
        } elseif ($isFallback) {
            return $this->loadClass($class, self::ACT_AS_FALLBACK);
        }
        return false;
    }
    if ($isFallback) {
        return $this->loadClass($class, self::ACT_AS_FALLBACK);
    }
    return false;
}

```

L'autoloader vérifie alors si le chargement concerne un nom de classe avec espace de nom (simple vérification de la présence d'un double-slash) et utilise la méthode « `loadClass()` » afin de la charger. C'est seulement si le chargement n'a pas abouti que l'autoloader appelle cette même méthode, qui utilisera alors la directive « `include_path` », comme nous l'avons spécifié :

Zend/Loader/StandardAutoloader.php

```

protected fonction loadClass($class, $type)
{
[...]
if ($type === self::ACT_AS_FALLBACK) {
    $filename      = $this->transformClassNameToFilename($class, '');
    $resolvedName = stream_resolve_include_path($filename);
    if ($resolvedName !== false) {
        return include $resolvedName;
    }
    return false;
}
[...]
}

```

Comme son nom l'indique, la méthode de chargement « `stream_resolve_include_path()` » est basée sur la directive « `include_path` ». Même si l'utilisation de celle-ci est à éviter, nous pouvons nous en servir une fois nos préfixes et autres espaces de noms enregistrés, ceux-ci étant prioritaires lors du chargement.

Notons qu'il est aussi possible de fournir un tableau d'options directement dans le constructeur de notre objet de chargement de classe ou depuis sa méthode « `setOptions()` » :

Utilisation d'un tableau d'options

```

$autoloader = Zend\Loader\AutoloaderFactory::getRegisteredAutoloader(
Zend\Loader\AutoloaderFactory::STANDARD_AUTOLOADER);
$autoloader->setOptions(array(
    'namespaces'=>array('Namesp'=> './autoload-dir/Namesp'),
    'prefixes'=>array('Prefix_'=> './autoload-dir/Prefix'),
)
);
new Namesp\Test();
new Prefix_Test();

```

Afin de gagner en performances, le framework propose dans cette version, un nouvel autoloader plus rapide et encore plus simple d'utilisation, le ClassMapAutoloader.

AVEC LE ZEND FRAMEWORK 1

Dans la version 1 du framework, la classe de chargement Zend_Loader_Autoloader permet de charger des espaces de noms (préfixes en réalité), la classe StandardAutoloader reprend un peu ce principe. La classe Zend_Loader_Autoloader peut aussi enregistrer d'autres autoloaders pour un même préfixe et ceux-ci seront utilisés en priorité. Il faut aussi noter qu'il n'y a pas de notion d'espace de nom PHP dans le Zend Framework 1, l'autoloader se base donc sur ce que fournit le développeur. Dans cette nouvelle version, espace de nom PHP et préfixes sont bien différenciés, ce qui rend les performances meilleures car le framework peut maintenant différencier les deux.

Le ClassMapAutoloader

Associant nom de classe et chemin, le Zend\Loader\ClassMapAutoloader est certainement l'autoloader le plus performant. Dès lors que le tableau de chemins correspondant aux noms de classes est configuré, le ClassMapAutoloader est fonctionnel.

Prenons un exemple avec la même structure de dossier que pour l'autoloader standard de la section précédente :

Utilisation de la classe ClassMapAutoloader

```
chdir(dirname(__DIR__));
require_once (getenv('ZF2_PATH') ?: 'vendor/ZendFramework/library') .
'Zend/Loader/AutoloaderFactory.php';
Zend\Loader\AutoloaderFactory::factory();

$maps = include 'config/application.autoload_classmap.php';
Zend\Loader\AutoloaderFactory::factory(array('Zend\Loader\
ClassMapAutoloader' => array($maps)));

new Namesp\Test();
new Prefix_Test();
```

Le fichier de configuration est un simple tableau de type clé/valeur :

config/application.autoload_classmap.php

```
<?php
return array(
    'Namesp\Test' => __DIR__ . '/../autoload-dir/Namesp/Test.php',
    'Prefix_Test' => __DIR__ . '/../autoload-dir/Prefix/Test.php',
);
```

Nous indiquons à la méthode « factory() » de la fabrique de classe de chargement

que nous souhaitons enregistrer une instance de la classe ClassMapAutoloader auprès de la SPL. La méthode s'attend à recevoir en paramètre un nom de classe avec sa configuration associée :

Zend/Loader/AutoloaderFactory.php

```
public static fonction factory($options = null)
{
    if (null === $options) {
        [...]
    }
    if (!is_array($options) && !($options instanceof \Traversable)) {
        [...]
    }
    foreach ($options as $class => $options) {
        if (!isset(static::$loaders[$class])) {
            $autoloader = static::getStandardAutoloader();
            [...]
            if ($class === static::STANDARD_AUTOLOADER) {
                $autoloader->setOptions($options);
            } else {
                $autoloader = new $class($options);
            }
            $autoloader->register();
            static::$loaders[$class] = $autoloader;
        } else {
            static::$loaders[$class]->setOptions($options);
        }
    }
}
```

La méthode de fabrication des autoloaders contrôle l'existence de l'instance demandée avant de la créer avec les options passées en paramètre. L'objet créé s'enregistre ensuite automatiquement auprès de la SPL, ce qui nous évite de l'enregistrer explicitement depuis sa méthode « register() ». Nous remarquons aussi que si l'instance de notre objet est existante, la fabrique se contente alors de mettre à jour les options de cette instance. La méthode d'altération des options « setOptions() » ne remplace pas mais ajoute les nouvelles options à la configuration existante, comme nous le remarquons dans la classe ClassMapAutoloader :

Zend/Loader/ClassMapAutoloader.php

```
public fonction setOptions($options)
{
    $this->registerAutoloadMaps($options
    return $this;
}
```

Zend/Loader/ClassMapAutoloader.php

```
public fonction registerAutoloadMaps($locations)
{
    [...]
    foreach ($locations as $location) {
        $this->registerAutoloadMap($location);
    }
    return $this;
}
```

Zend/Loader/ClassMapAutoloader.php

```

public fonction registerAutoloadMap($map)
{
    if (is_string($map)) {
        $location = $map;
        if ($this === ($map = $this->loadMapFromFile($location))) {
            return $this;
        }
    }
    [...]
    $this->map = array_merge($this->map, $map);
    if (isset($location)) {
        $this->mapsLoaded[] = $location;
    }
    return $this;
}

```

La mise à jour des options se fait ensuite avec la fusion des anciennes configurations avec les nouvelles, avant de stocker le résultat. Attention, pour deux clés identiques, la fusion utilise en priorité la clé du nouveau tableau d'options, car la dernière valeur rencontrée écrase l'ancienne. Pour plus de précision, je vous conseille d'aller voir directement la documentation PHP de la fonction « `array_merge()` ».

Dans notre exemple, nous avons passé en paramètre de la fabrique un tableau d'options, mais il est également possible d'indiquer au constructeur du `ClassMapAutoloader` un chemin de fichier de configuration. L'autoloader s'occupe alors de récupérer automatiquement les options en utilisant l'instruction de langage « `include` », par exemple :

Utilisation d'un fichier de correspondance

```

Zend\Loader\AutoloaderFactory::factory(array(
    'Zend\Loader\ClassMapAutoloader' => array('config/application.
    autoload_classmap.php')
));

```

Zend/Loader/ClassMapAutoloader.php

```

public fonction registerAutoloadMap($map)
{
    if (is_string($map)) {
        $location = $map;
        if ($this === ($map = $this->loadMapFromFile($location))) {
            return $this;
        }
    }
    [...]
}

```

La méthode « `loadMapFromFile()` » permet le chargement de configuration depuis un fichier. Notons que l'objet de type `ClassMapAutoloader` enregistre tous les chemins de fichiers fournis dans la variable « `$mapsLoaded` », ce qui permet de s'assurer de ne pas faire le traitement deux fois. En lui passant un tableau directement en paramètre comme dans le premier exemple, c'est au développeur de s'assurer de ne pas faire deux fois le traitement car aucun contrôle ne sera effectué du côté de l'autoloader :

Zend/Loader/ClassMapAutoloader.php

```
protected fonction loadMapFromFile($location)
{
    if (!file_exists($location)) {
        [...]
    }
    if (!$path = static::realPharPath($location)) {
        $path = realpath($location);
    }
    if (in_array($path, $this->mapsLoaded)) {
        return $this;
    }
    $map = include $path;
    return $map;
}
```

Avec quelques traitements en plus, les deux manières de faire sont identiques. Nous savons à présent paramétrier les classes de chargement, analysons maintenant l'enregistrement auprès de la SPL ainsi que la fonction appelée lors du chargement :

Zend/Loader/ClassMapAutoloader.php

```
public function register()
{
    spl_autoload_register(array($this, 'autoload'), true, true);
}
```

Le troisième paramètre passé à la méthode de la SPL indique que l'on souhaite placer cette classe de chargement en haut de la pile des autoloaders de notre application, par défaut les enregistrements se placent à la fin. Étant donné les très bonnes performances de l'objet ClassMapAutoloader, il est dans l'intérêt de l'application de l'utiliser en premier.

Comme il est indiqué lors de l'enregistrement auprès de la SPL, c'est la méthode « `autoload()` » qui s'occupe du chargement de la classe :

Zend/Loader/ClassMapAutoloader.php

```
public fonction autoload($class)
{
    if (isset($this->map[$class])) {
        require_once $this->map[$class];
    }
}
```

Nous comprenons alors tout de suite l'intérêt d'utiliser la classe ClassMapLoader, le traitement est simple et performant. Il suffit que la classe demandée soit bien configurée depuis les options de l'autoloader afin de pouvoir l'inclure directement dans notre code sans plus de recherche.

Maintenant que nous avons saisi l'intérêt de la classe de chargement ClassMapAutoloader, nous pourrions vouloir l'utiliser pour la bibliothèque du Zend Framework plutôt que de passer par l'enregistrement d'un espace de nom auprès de notre autoloader standard. Les développeurs du framework mettent à disposition un fichier « `classemap_generator.php` » qui permet la génération automatique d'une configuration à partir d'un répertoire cible. Il suffit de consulter les options

du script afin d'en comprendre le fonctionnement :

Aide sur la génération de fichier de mapping

```
|| php classmap_generator.php --help
```

Une fois le fichier de configuration généré, toutes les classes de l'espace de nom « Zend » seront automatiquement chargées depuis cet autoloader, ce qui permet un gain de performances.

La classe **ModuleAutoloader**

Comme son nom l'indique et comme nous le verrons dans le chapitre consacré aux modules, le Zend\Loader\ModuleAutoloader s'adapte uniquement à nos modules, ce que nous prouvent les premières lignes du chargement de classe :

Zend/Loader/ModuleAutoloader.php

```
|| public fonction autoload($class)
  {
    if (substr($class, -7) !== '\Module') {
      return false;
    }
    [...]
  }
```

L'autoloader ne travaille que si nous recherchons une classe nommée « Module », point d'entrée des modules de l'application. L'utiliser comme autoloader pour le reste de notre application n'aurait alors aucune utilité.

La classe ModuleAutoloader sera étudiée en détail dans la section dédiée aux modules.

L'interface **SplAutoloader**

Les autoloaders ClassMapAutoloader et StandardAutoloader devraient suffire pour nos applications, leur fonctionnement étant assez complet. Cependant, il est possible de créer sa propre classe de chargement en implémentant l'interface Zend\Loader\SplAutoloader qui définit quatre méthodes :

Zend/Loader/SplAutoloader.php

```
|| interface SplAutoloader
  {
    public function __construct($options = null);

    public function setOptions($options);

    public function autoload($class);

    public function register();
  }
```

Le constructeur doit pouvoir accepter les options de l'autoloader en paramètre afin de permettre à la méthode « `factory()` », de la fabrique `AutoloaderFactory`, de créer directement l'instance en lui passant les options en paramètre comme nous l'avons vu précédemment. La méthode « `setOptions()` » sera utilisée pour la mise à jour de la configuration de l'autoloader par la fabrique lorsque l'instance de la classe demandée existe déjà. Les deux autres méthodes, « `register()` » et « `autoload()` », sont également utilisées par la méthode de fabrication « `factory()` » de l'`AutoloaderFactory`.

3

Les évènements

La programmation évènementielle est au cœur du Zend Framework 2. Les évènements permettent aux différents composants de communiquer, d'interagir et de partager des informations entre eux. Ce type de programmation n'est pas antipattern et permet d'instaurer de bonnes pratiques de codage. Cela va aussi permettre de supprimer les hooks que l'on connaît du Zend Framework 1 tout en augmentant la souplesse de notre application. Nous verrons que la gestion des priorités sur les écouteurs de nos évènements est une bonne solution de remplacement aux pré et postévenements (« `preDispatch` » et « `postDispatch` » par exemple) que l'on connaît.

Présentation des évènements

Un nouveau composant a fait son apparition dans Zend Framework 2, le gestionnaire d'évènements `Zend\EventManager\EventManager`. Définissons d'abord les différents termes :

- un « `event` » ou évènement est une action ;
- un « `listener` » ou écouteur est un écouteur d'événements ;
- un « `event manager` » ou gestionnaire d'évènements est un objet qui contient un ensemble d'écouteurs et qui va lancer des évènements.

Toute méthode « écoutant » un évènement est notifiée lorsque celui-ci est lancé. Le gestionnaire d'évènements de type `EventManager` fait la passerelle entre les écouteurs et les évènements qui se produisent. Il est possible d'attacher un évènement à un objet à travers la méthode « `attach()` » et de lancer un évènement depuis la méthode « `trigger()` ».

Afin de comprendre l'intérêt et le fonctionnement des évènements, prenons un exemple d'utilisation du gestionnaire d'évènements.

Exemple d'utilisation d'évènement

```
class Developpeur
{
protected $events;
protected $pauseHandler;

public function travail()
{
    $this->pauseHandler = $this->getEventManager()->attach('pause',
function($e) {
    printf('Evenement en cours "%s" sur objet "%s", avec les
paramètres %s',
        $e->getName(),
        get_class($e->getTarget()),
        json_encode($e->getParams())
    );
});
}
public function termine()
{
    if(!$this->pauseHandler){
        return;
    }
    $this->getEventManager()->detach($this->pauseHandler);
    $this->pauseHandler = null;
}

public function setEventManager(EventManagerInterface $events)
{
    $this->events = $events;
}

public function getEventManager()
{
    if (!$this->events){
        $this->setEventManager(new EventManager(
            array(__CLASS__, get_called_class())
        ));
    }
    return $this->events;
}

public function pause($params)
{
    $this->getEventManager()->trigger(__FUNCTION__, $this,
$params);
}

$developpeur = new Developpeur();
$developpeur->travail();
$developpeur->pause(array('cafe', 'court'));
$developpeur->termine();
$developpeur->pause(array('cafe', 'long'));
```

Seule la phrase « Evenement en cours "pause" » sur l'objet « Developpeur », avec les paramètres « ["cafe", "court"] » s'affiche. En effet, l'objet « \$developpeur » s'est désabonné de l'évènement « pause » lorsque la méthode « termine() » s'est effectuée.

Chaque gestionnaire d'évènements est identifié par une chaîne de caractères que l'on passe à son constructeur (ici le nom de la classe en cours avec `__CLASS__`), ce qui lui permet ensuite d'effectuer une liaison avec le gestionnaire partagé.

Le gestionnaire partagé

Un deuxième type de gestionnaire d'évènements est disponible, le `Zend\EventManager\SharedEventManager`. Assez simple d'utilisation, il possède un tableau d'objets de type `EventManager` en interne et permet aussi d'attacher des écouteurs à un gestionnaire identifié depuis la méthode « `attach()` » :

Création de gestionnaire partagé

```
$events = new SharedEventManager();
$events->attach('Developpeur', 'pause', function($e) {
    printf('Evenement en cours "%s" sur objet "%s", avec les paramètres
%s',
        $e->getName(),
        get_class($e->getTarget()),
        json_encode($e->getParams())
    );
});
```

Ce code instancie en réalité un gestionnaire d'évènements avec comme identifiant la chaîne de caractères passée en premier paramètre, ici « `Developpeur` ». Analysons le traitement effectué dans la méthode « `attach()` » de la classe `SharedEventManager` :

Zend/EventManager/SharedEventManager.php

```
public function attach($id, $event, $callback, $priority = 1)
{
    $ids = (array) $id;
    foreach ($ids as $id) {
        if (!array_key_exists($id, $this->identifiers)) {
            $this->identifiers[$id] = new EventManager();
        }
        $this->identifiers[$id]->attach($event, $callback, $priority);
    }
}
```

L'identifiant « `Developpeur` » sert de clé dans le tableau associatif « `$identifiers` », où est alors instancié un gestionnaire d'évènements auquel on ajoute l'écouteur pour l'évènement passé en deuxième paramètre.

Tentons maintenant de comprendre le lien entre un gestionnaire d'évènements que l'on a créé plus haut et l'objet `SharedEventManager`. Examinons de plus près la méthode « `trigger()` » de la classe `EventManager`.

Zend/EventManager/EventManager.php

```
public function trigger($event, $target = null, $argv = array(),
    $callback = null)
{
    [...]
    return $this->triggerListeners($event, $e, $callback);
}
```

Voici la méthode « triggerListeners() » :

Zend/EventManager/EventManager.php

```
protected function triggerListeners($event, EventInterface $e,
    $callback = null)
{
    $responses = new ResponseCollection;
    $listeners = $this->getListeners($event);

    $sharedListeners      = $this->getSharedListeners($event);
    $sharedWildcardListeners = $this->getSharedListeners('*');
    $wildcardListeners     = $this->getListeners('*');

    if (count($sharedListeners) || count($sharedWildcardListeners) ||
        count($wildcardListeners)) {
        $listeners = clone $listeners;
    }

    $this->insertListeners($listeners, $sharedListeners);
    $this->insertListeners($listeners, $sharedWildcardListeners);
    $this->insertListeners($listeners, $wildcardListeners);

    if ($listeners->isEmpty()) {
        return $responses;
    }

    foreach ($listeners as $listener) {
        $responses->push(call_user_func($listener->getCallback(), $e));

        if ($e->propagationIsStopped()) {
            $responses->setStopped(true);
            break;
        }

        if ($callback && call_user_func($callback, $responses->last())) {
            $responses->setStopped(true);
            break;
        }
    }
    return $responses;
}
```

Cette méthode récupère la liste des écouteurs du gestionnaire partagé depuis la méthode « getSharedListeners() » :

Zend/EventManager/EventManager.php

```
protected function triggerListeners($event, EventInterface $e,
$callback = null)
{
[...]
$sharedListeners      = $this->getSharedListeners($event);
[...]
}
```

Cette méthode récupère tous les écouteurs des gestionnaires d'évènements du gestionnaire partagé, dont l'identifiant correspond à un des identifiants du gestionnaire courant. Le tableau de chaînes de caractères passé au constructeur du gestionnaire sert d'identifiant, nous comprenons maintenant son utilité. Par exemple, la clé passée au constructeur du gestionnaire suivant :

Les identifiants du gestionnaire

```
new EventManager(
    array(__CLASS__, get_called_class())
)
```

Ici « __CLASS__ » ou « get_called_class() », doit être de la même valeur que le premier argument de la méthode « attach() » de la classe SharedEventManager vue précédemment :

Attache d'écouteur au gestionnaire partagé

```
$events->attach('Developpeur', 'pause', function($e) {
    [...]
});
```

Dans cet exemple, le lien entre le gestionnaire d'évènements instancié précédemment (qui possède comme clé la constante PHP __CLASS__, qui a pour valeur le nom de la classe, ici « Developpeur ») et l'ajout d'écouteurs sur l'instance de l'objet de type SharedEventManager (dont on demande la création d'un gestionnaire pour identifiant « Developpeur ») peut être fait. Cette manière de fonctionner peut s'avérer assez utile pour ajouter facilement des évènements sans avoir à disposition l'instance du gestionnaire d'évènements concerné, si le partage est effectué.

Cependant, ce comportement peut s'avérer parfois déstabilisant et il est possible de le désactiver. Si l'on examine la méthode qui s'occupe de récupérer la liste des écouteurs du gestionnaire partagé, nous remarquons qu'elle va tout d'abord récupérer la liste des gestionnaires d'évènements du SharedEventManager avant de faire correspondre avec les identifiants.

Zend/EventManager/EventManager.php

```
protected function getSharedListeners($event)
{
if (!$sharedManager = $this->getSharedManager()) {
    return array();
}
$identifiers      = $this->getIdentifiers();
$sharedListeners = array();
```

```

    foreach ($identifiers as $id) {
        if (!$listeners = $sharedManager->getListeners($id, $event)) {
            continue;
        }

        if (!is_array($listeners) && !($listeners instanceof
Traversable)) {
            continue;
        }

        foreach ($listeners as $listener) {
            if (!$listener instanceof CallbackHandler) {
                continue;
            }
            $sharedListeners[] = $listener;
        }
    }
    return $sharedListeners;
}

```

La première ligne tente d'obtenir une connexion avec notre gestionnaire partagé. Cette connexion est matérialisée par l'existence d'une instance de type SharedEventManager :

Zend/EventManager/EventManager.php

```

protected function getSharedListeners($event)
{
    if (!$sharedManager = $this->getSharedManager()) {
        return array();
    }
    [...]
}

```

Zend/EventManager/EventManager.php

```

public function getSharedManager()
{
    if (false === $this->sharedManager
        || $this->sharedManager instanceof SharedEventManagerInterface
    ) {
        return $this->sharedManager;
    }
    if (!StaticEventManager::hasInstance()) {
        return false;
    }
    $this->sharedManager = StaticEventManager::getInstance();
    return $this->sharedManager;
}

```

La relation entre le gestionnaire d'évènements courant et l'objet SharedEventManager se fait automatiquement. Pour désactiver cette fonctionnalité et bloquer la passerelle entre nos gestionnaires d'évènements et notre gestionnaire statique, il suffit d'appeler la méthode « `unsetSharedManager()` » qui permet cette désactivation en passant la valeur « `false` » à l'attribut « `$sharedManager` », valeur qui est synonyme de désactivation comme nous venons de le voir :

Désactivation de la liaison avec le gestionnaire partagé

```
|| $this->events->unsetSharedManager();
```

Il est alors possible de réactiver cette liaison en rejouant le code de la méthode «`getSharedManager()`» :

Activation de la liaison avec le gestionnaire partagé

```
|| $this->events->setSharedManager(StaticEventManager::getInstance());
```

Le gestionnaire d'évènements récupère ensuite la liste des écouteurs présents dans les objets de type `EventManager` du gestionnaire partagé pour l'évènement courant :

`Zend/EventManager/EventManager.php`

```
protected function getSharedListeners($event)
{
    [...]
    $identifiers      = $this->getIdentifiers();
    $sharedListeners = array();

    foreach ($identifiers as $id) {
        if (!$listeners = $sharedManager->getListeners($id, $event)) {
            continue;
        }
        if (!is_array($listeners) && !($listeners instanceof
Traversable)) {
            continue;
        }
        foreach ($listeners as $listener) {
            if (!$listener instanceof CallbackHandler) {
                continue;
            }
            $sharedListeners[] = $listener;
        }
    }
    return $sharedListeners;
}
```

Afin de bien maîtriser la programmation évènementielle du Zend Framework, il est important de bien assimiler que le gestionnaire partagé agit comme un simple conteneur de gestionnaire d'évènements. Ce n'est en aucun cas un gestionnaire d'évènements, il a pour seule responsabilité de stocker des objets de type `EventManager` qui peuvent avoir les mêmes identifiants que d'autres gestionnaires que l'on a créés. Il peut ensuite faire la liaison avec d'autres gestionnaires d'évènements qui possèdent le même identifiant que l'un de ses objets stockés. Cette liaison peut être supprimée et il est important de bien comprendre qu'il est différent d'attacher un évènement sur le gestionnaire partagé plutôt que sur le gestionnaire d'évènements lui-même.

Nous avons remarqué que le gestionnaire partagé est par défaut une instance de la classe `StaticEventManager` :

Zend/EventManager/EventManager.php

```
public function getSharedManager()
{
    [...]
    $this->sharedManager = StaticEventManager::getInstance();
    return $this->sharedManager;
}
```

La classe StaticEventManager est en fait une représentation statique de la classe SharedEventManager :

Zend/EventManager/StaticEventManager.php

```
public static function getInstance()
{
    if (null === static::$instance) {
        static::$instance = new static();
    }
    return static::$instance;
}
```

La classe StaticEventManager hérite de la classe SharedEventManager et se comporte comme un singleton afin d'offrir une instance unique d'objet de type SharedEventManager. Ce comportement permet aux gestionnaires d'évènements de partager la même instance de gestionnaire partagé afin de pouvoir ajouter des évènements depuis n'importe quelle instance de gestionnaire d'évènements.

Examinons l'implémentation du gestionnaire d'évènements partagé du framework avec la fabrique Zend\ Mvc\ Service\ EventManagerFactory qui est responsable de la fabrication du gestionnaire d'évènements de l'application. Pour plus d'informations sur le gestionnaire de services, qui permet de gérer les fabriques d'objets, reportez-vous au chapitre qui lui est consacré. Voici la fabrique EventManagerFactory du gestionnaire d'évènements :

Zend/Mvc/Service/EventManagerFactory.php

```
class EventManagerFactory implements FactoryInterface
{
    public function createService(ServiceLocatorInterface
        $serviceLocator)
    {
        $em = new EventManager();
        $em->setSharedManager($serviceLocator-
>get('SharedEventManager'));
        return $em;
    }
}
```

La classe utilise la fabrique partagée SharedEventManager afin de toujours utiliser la même instance de gestionnaire d'évènements partagé. En effet, la fabrique « SharedEventManager » est une fabrique dont l'instance sera partagée, car la création systématique d'une nouvelle instance du gestionnaire partagé ferait perdre tout son sens à la notion de partage.

Nous pouvons donc constater que l'instance du gestionnaire partagé sera identique

avec une instantiation sans la fabrique du gestionnaire d'évènements si l'on utilise toujours la fabrique SharedEventManager.

L'évènement wildcard

Nous avons parcouru le code de la classe EventManger et nous avons remarqué la présence d'un évènement sous forme de « joker » :

Analyse de la récupération des évènements « wildcard »

```
protected function triggerListeners(EventInterface $event, $callback = null)
{
    $responses = new ResponseCollection;
    $listeners = $this->getListeners($event);

    $sharedListeners      = $this->getSharedListeners($event);
    $sharedWildcardListeners = $this->getSharedListeners('*');
    $wildcardListeners     = $this->getListeners('*');
    [...]
}
```

Pour chaque évènement, le gestionnaire d'évènements récupère les écouteurs attachés sur l'évènement « * », ce qui leur permet d'être notifié dès qu'un évènement est lancé. L'écouteur notifié pourra alors faire appel à la méthode « getName() » de l'évènement afin de savoir quel est l'évènement qui a été lancé. Prenons un exemple dans la classe Module de notre module Application :

Application/Module.php

```
class Module implements AutoloaderProvider
{
    public function init(ModuleManager $moduleManager)
    {
        $events = $moduleManager->events()->getSharedManager();
        $events->attach('Zend\Mvc\Application', '*', array($this, 'onApplicationEvent'), 100);
        [...]
    }
    public function onApplicationEvent(Event $e)
    {
        echo $e->getName(); // affiche successivement route dispatch
        render finish
    }
    [...]
}
```

Les évènements wildcard doivent être utilisés avec précaution, car nous ne connaissons pas à l'avance tous les évènements qui peuvent être lancés sur une instance de gestionnaire d'évènements, il est important de s'assurer de pouvoir agir lors des différents évènements possibles.

Le gestionnaire d'évènements global

Le gestionnaire d'évènements global est très simple d'utilisation. Toutes les méthodes sont statiques, il n'y a pas de notion d'identifiants comme pour le gestionnaire partagé, ni d'interaction avec d'autres gestionnaires d'évènements. La classe GlobalEventManager est donc accessible de n'importe où et fonctionne de manière indépendante :

Zend/EventManager/GlobalEventManager.php

```
|| GlobalEventManager::attach('random-event', function($e)
|| {
||     echo $e->getName() . ' en cours, paramètre reçu : ' .
||     $e->getParam('myparam');
|| }
|| );
|| GlobalEventManager::trigger('random-event', null,
||     array('myparam'=>'value1'));
```

Affichage en sortie

```
|| random-event en cours, paramètre reçu : value1
```

Si le gestionnaire global est plus simple de compréhension, il est conseillé de modérer son utilisation et de ne pas s'en servir comme « fourre-tout » afin d'éviter les collisions de noms d'évènements. Utiliser les identifiants des gestionnaires d'évènements permet une sécurité supplémentaire, ainsi qu'un code mieux organisé. Cette manière de regrouper les évènements va permettre de donner du sens à leur gestion et permettre à n'importe quel développeur travaillant sur l'application de comprendre plus vite l'utilité et le domaine d'interaction de chacun des gestionnaires et de leurs évènements.

Le type ListenerAggregate

Nous pouvons parfois avoir besoin d'une classe capable de gérer une multitude d'évènements avec les écouteurs associés. Afin de pouvoir confier cette responsabilité à une classe indépendante, l'interface Zend\EventManager\ListenerAggregate est disponible, voici les méthodes que l'on devra alors implémenter au sein de notre classe responsable du traitement des évènements :

Zend/EventManager/ListenerAggregate.php

```
|| interface ListenerAggregate
|| {
||     public function attach(EventManagerInterface $events);
||     public function detach(EventManagerInterface $events);
|| }
```

Voici un squelette de classe pouvant gérer différents types d'évènements en un seul objet :

Exemple de conteneur d'écouteur

```

class TravailleurEvents implements ListenerAggregate
{
protected $handlers = array();

public function attach(EventManagerInterface $events)
{
    $this->handlers[] = $events->attach('pause-dejeuner',
array($this, 'absent'));
    $this->handlers[] = $events->attach('reunion', array($this,
'occupe'));
}

public function detach(EventManagerInterface $events)
{
    foreach ($this->handlers as $key => $handler)
    {
        $events->detach($handler);
        unset($this->handlers[$key]);
    }
    $this->handlers = array();
}

public function absent(Event $e)
{
    $horaire = $e->getParam('retour');
    echo «Je suis actuellement absent du bureau.»;
    if($horaire)
    {
        echo «Je serai de retour vers » . $horaire;
    }
}
public function occupe(Event $e)
{
    $remplacant = $e->getParam('collegue',null);
    echo «Je suis actuellement indisponible.»;
    if($remplacant)
    {
        echo «Pour toutes questions, merci de vous adresser à »
. $remplacant;
    }
}
}

```

Ce conteneur capable de gérer les écouteurs peut être attaché comme ceci :

Utilisation d'un conteneur d'écouteur

```

$events = new EventManager('Travailleur');
$gestionHoraire = new TravailleurEvents();
$events->attachAggregate($gestionHoraire);

$midiEvent = new Event();
$midiEvent->setName('pause-dejeuner');
$midiEvent->setParam('retour', '14h');

$brainstormingEvent = new Event();
$brainstormingEvent->setName('reunion');
$brainstormingEvent->setParam('collegue', 'Fred');
$events->trigger($midiEvent);
$events->trigger($brainstormingEvent);

```

Affichage en sortie

```
|| Je suis actuellement absent du bureau. Je serai de retour vers 14 h
|| Je suis actuellement indisponible. Pour toutes questions, merci de
|| vous adresser à Fred
```

Les exemples ici sont très simples, mais permettent d'avoir une première approche sur l'intérêt de ce genre de pratique assez répandue dans le framework.

Découvrons maintenant la méthode « `attachAggregate()` » de la classe `EventManager` :

Zend/EventManager/EventManager.php

```
|| public function attachAggregate(ListenerAggregate $aggregate,
|| $priority = 1)
|| {
|| return $aggregate->attach($this, $priority);
|| }
```

Comme on peut s'y attendre, celle-ci passe en paramètre l'instance du gestionnaire courant à l'agrégateur afin qu'il puisse attacher l'ensemble des évènements gérés par la classe `TravailleurEvents`. Nous pourrions donc remplacer directement l'instruction suivante :

Attache des écouteurs du conteneur

```
|| $events->attachAggregate($gestionHoraire);
```

Par le code suivant :

Attache des écouteurs du conteneur

```
|| $gestionHoraire->attach($events);
```

Pour une question de compréhension et de lisibilité, nous conserverons la première instruction. Notons qu'il est aussi possible d'utiliser la méthode « `attach()` » de la classe `EventManager` pour ajouter l'agrégateur, car celle-ci a prévu de gérer ce cas en vérifiant le type d'argument reçu :

Zend/EventManager/EventManager.php

```
|| public function attach($event, $callback = null, $priority = 1)
|| {
|| if ($event instanceof ListenerAggregate) {
||     return $this->attachAggregate($event, $callback);
|| }
|| [...]
|| }
```

La méthode « `attach()` » de la classe `TravailleurEvents` s'occupe, elle, d'attacher les évènements « `pause-dejeuner` » et « `reunion` ». Notons qu'il est aussi possible de passer la priorité de l'écouteur en paramètre. Nous verrons dans la section consacrée à l'ordre de la gestion des notifications, l'intérêt de celle-ci.

Il est important de stocker les objets retournés depuis la méthode « attach() » du gestionnaire d'évènements, car ces objets représentent un écouteur d'évènements :

Zend/EventManager/EventManager.php

```
|| public function attach($event, $callback = null, $priority = 1)
{
[...]
$listener = new CallbackHandler($callback, array('event' => $event,
'priority' => $priority));
$this->events[$event]->insert($listener, $priority);
return $listener;
}
```

L'objet Zend\Stdlib\CallbackHandler est le type de l'objet qui représente l'écouteur et qui est retourné par cette méthode, il est important de garder une trace de cet objet afin de pouvoir utiliser la méthode « detach() » qui permet de supprimer l'écoute d'un évènement depuis son identifiant :

Zend/EventManager/EventManager.php

```
|| public function detach($listener);
```

Le paramètre « \$listener » peut être un objet de type ListenerAggregate (comme on l'a vu pour la méthode « attach() », cela fonctionne de la même manière pour la méthode « detach() »), ou un objet de type CallbackHandler.

Nous pourrions vouloir améliorer l'exemple précédent avec l'utilisation d'évènements personnalisés plutôt que de passer en paramètre les données que l'on souhaite transmettre à l'écouteur. Cela permettrait une encapsulation et un découplage plus complet et un descriptif des évènements plus intéressant.

Créer ses évènements personnalisés

Lors du lancement d'évènement, si la méthode « trigger() » reçoit un évènement sous forme de chaîne de caractères, elle s'occupe de l'envelopper sous la forme d'objet :

Zend/EventManager/EventManager.php

```
|| public function trigger($event, $target = null, $argv = array(),
$callback = null)
{
if ($event instanceof EventInterface) {
    $e      = $event;
    $event  = $e->getName();
    $callback = $target;
} elseif ($target instanceof EventInterface) {
    $e = $target;
    $e->setName($event);
    $callback = $argv;
} elseif ($argv instanceof EventInterface) {
    $e = $argv;
    $e->setName($event);
    $e->setTarget($target);
```

```
    } else {
        $e = new $this->eventClass();
        $e->setName($event);
        $e->setTarget($target);
        $e->setParams($argv);
    }
    [...]
}
```

L'objet construit sera du type de l'attribut « \$eventClass » :

Zend/EventManager/EventManager.php

```
protected $eventClass = 'Zend\EventManager\Event';
```

La classe Zend\EventManager\Event représente le type des évènements utilisé par défaut, il est possible de le changer avec la méthode :

Zend/EventManager/EventManager.php

```
public function setEventClass($class);
```

Le paramètre « \$class » est une chaîne de caractères qui représente le nom de la classe enveloppant les nouveaux évènements créés au sein du gestionnaire d'évènements. Cela permet d'initialiser l'évènement avec la classe de notre choix implémentant l'interface descriptive des évènements Zend\EventManager\EventInterface, car comme nous le voyons, la méthode « triggerListeners() » qui notifie les écouteurs s'attend à un objet de ce type :

Zend/EventManager/EventManager.php

```
protected function triggerListeners($event, EventInterface $e,
    $callback = null);
```

La première des choses à faire lors de la création d'évènements personnalisés est donc de créer une classe implémentant cette interface :

Zend/EventManager/EventInterface.php

```
interface EventInterface
{
    public function getName();
    public function getTarget();
    public function getParams();
    public function getParam($name, $default = null);
    public function setName($name);
    public function setTarget($target);
    public function setParams($params);

    public function setParam($name, $value);
    public function stopPropagation($flag = true);
    public function propagationIsStopped();
}
```

Voici comment construire et utiliser un évènement personnalisé en lui fournissant des options :

Construction d'un évènement personnalisé

```
|| $event = new MyEvent();
|| $event->setMyCustomParam($value);
```

La classe MyEvent hérite de la classe de base Event et définit une méthode « setMyCustomParam() ». Il ne reste alors plus qu'à l'injecter dans la méthode « trigger() » :

Lancement de l'évènement personnalisé

```
|| $events->trigger('mon-evenement', $this, $event);
```

Nous pouvons définir l'attribut « \$target » à l'évènement :

Modification de la cible de l'évènement

```
|| $event->setTarget($this);
```

Ou encore le nom de l'évènement :

Modification du nom de l'évènement

```
|| $event->setName('mon-evenement');
```

Ce qui nous donne une encapsulation complète :

Encapsulation de l'évènement

```
|| $event = new MyEvent();
|| $event->setMyResult($value);
|| $event->setTarget($this);
|| $event->setName('mon-evenement');
|| $events->trigger($event);
```

Il est toujours possible de passer un callback pour le court-circuitage en deuxième argument lorsque l'on utilise ce système d'encapsulation :

Lancement de l'évènement personnalisé

```
|| $events->trigger($event, $callback);
```

Nous verrons la notion de court-circuit et de propagation dans les deux prochains chapitres.

Maintenant que nous savons manipuler les objets de type EventInterface, nous allons nous intéresser aux objets de réponse des écouteurs, ainsi qu'au processus de propagation.

L'objet de réponse

Chaque écouteur d'évènements retourne une réponse qui est stockée par le gestionnaire d'évènements, et il peut être parfois utile d'avoir accès à cette liste de réponses. Nous avons pu remarquer que la méthode « `triggerListeners()` », du gestionnaire d'évènements, s'occupe de gérer les notifications et la récupération des réponses afin de les retourner en fin de traitement :

Zend/EventManager/EventManager.php

```
protected function triggerListeners(EventInterface $event, $callback = null)
{
    $responses = new ResponseCollection;
    $listeners = $this->getListeners($event);
    [...]
    foreach ($listeners as $listener) {
        $responses->push(call_user_func($listener->getCallback(), $event));
        if ($event->propagationIsStopped()) {
            $responses->setStopped(true);
            break;
        }
        if ($callback && call_user_func($callback, $responses->last())) {
            $responses->setStopped(true);
            break;
        }
    }
    return $responses;
}
```

L'objet « `$responses` » de type `Zend\EventManager\ResponseCollection` est responsable du stockage de la liste des réponses des écouteurs notifiés. L'objet `ResponseCollection` hérite de la classe `SplStack`, pile basée sur une liste doublement chaînée. La liste des méthodes de la classe `ResponseCollection` est assez courte :

Zend/EventManager/ResponseCollection.php

```
class ResponseCollection extends SplStack
{
    public function stopped();

    public function setStopped($flag);

    public function first();

    public function last();

    public function contains($value);
}
```

La méthode « `stopped()` » permet de connaître l'état de la propagation, si elle a été stoppée ou non. Le premier objet de réponse peut être récupéré par la méthode « `first()` » qui représente l'objet tout en bas de la pile, le dernier objet de réponse est extrait depuis la méthode « `last()` ». Enfin, la méthode « `contains()` » permet de savoir si une valeur donnée est contenue dans la liste des réponses des écouteurs.

Nous allons maintenant expliquer la notion de court-circuit, ainsi que la manière

d'agir sur la propagation des évènements, ce qui va nous permettre de contrôler intégralement le workflow de l'évènement.

Stopper la propagation d'un évènement

Lors du lancement d'un évènement, il est possible de stopper la propagation des notifications depuis un écouteur. La première manière de court-circuiter la propagation d'un évènement est de l'indiquer explicitement au sein du callback de l'écouteur :

Arrêt de la propagation d'un évènement

```
|| $events->attach('pause', function($e) {
||   $e->stopPropagation();
||   return new MonObjetDeReponse();
||});
```

Expliquons maintenant en détail le comportement de la méthode « stopPropagation() » :

Zend/EventManager/Event.php

```
|| public function stopPropagation($flag = true)
|| {
||   $this->stopPropagation = (bool) $flag;
|| }
|| public function propagationIsStopped()
|| {
||   return $this->stopPropagation;
|| }
```

Ces deux méthodes permettent le maintien et la consultation de l'attribut « stopPropagation » qui est utilisé dans le gestionnaire d'évènements au sein de la méthode « triggerListeners() ». En effet, cette méthode s'occupe de récupérer les écouteurs de l'évènement afin de les notifier :

Zend/EventManager/EventManager.php

```
|| protected function triggerListeners(EventInterface $event, $callback = null)
|| {
||   $responses = new ResponseCollection;
||   $listeners = $this->getListeners($event);
||   [...]
||   foreach ($listeners as $listener) {
||     $responses->push(call_user_func($listener->getCallback(), $event));
||     if ($event->propagationIsStopped()) {
||       $responses->setStopped(true);
||       break;
||     }
||     [...]
||   }
||   [...]
|| }
```

Le gestionnaire notifie les évènements un à un et ajoute leur retour au tableau de réponses. Si l'attribut « `$stopPropagation` », que l'on vérifie à l'aide de la méthode « `propagationIsStopped()` », est passé à la valeur « `false` » au sein d'un écouteur, alors la pile de réponses est notifiée et la propagation de l'évènement est stoppée.

La deuxième manière d'arrêter la propagation d'un évènement est de passer une fonction de callback à la méthode « `trigger()` », qui prend en paramètre la réponse retournée par le dernier écouteur notifié et retourne un booléen. Si le retour de cette fonction de callback est vrai, alors la propagation s'arrête.

Reprenons la méthode « `triggerListener()` » du gestionnaire d'évènements pour mieux comprendre comment est géré le court-circuitage :

Zend/EventManager/EventManager.php

```
protected function triggerListeners(EventInterface $event, $callback = null)
{
    [...]
    foreach ($listeners as $listener) {
        $responses->push(call_user_func($listener->getCallback(), $event));
        if ($event->propagationIsStopped()) {
            $responses->setStopped(true);
            break;
        }

        if ($callback && call_user_func($callback, $responses->last())) {
            $responses->setStopped(true);
            break;
        }
    }
    [...]
}
```

L'objet de réponse est passé en paramètre à la fonction de callback si elle existe, et si le retour de la fonction est vrai, alors la propagation est stoppée. Voici un exemple trivial :

Exemple d'utilisation de court-circuitage

```
$events->attach('pause', function($event) {
    return new A();
});
$events->attach('pause', function($event) {
    return new B();
});
$events->attach('pause', function($event) {
    return new C();
});
$events->trigger('pause', $this, $params, function($response)
{
    return $response instanceof B; // arrête la propagation dès qu'un écouteur
    retourne un objet de type B
});
```

La propagation va alors s'arrêter dès la deuxième notification.

Attention, comme indiqué plus haut, il suffit que le booléen de retour soit vrai, un

court-circuit comme ci-dessous arrête immédiatement la propagation :

Court-circuitage depuis une valeur « vraie »

```
|| $events->trigger('pause', $this, $params, function($r)
|| {
||     return 'false';
||});
```

En effet, la condition d'arrêt ne vérifie pas la valeur « true » mais si celle-ci est « vrai ». Pour rappel, en PHP tout ce qui n'est pas faux est considéré comme vrai. Pour plus de détails, je vous suggère d'aller consulter la documentation de PHP pour voir que ce qui peut être considéré comme faux lors d'une conversion en booléen, comme : le booléen FALSE, l'entier 0, la chaîne vide, un tableau avec éléments, le type NULL, etc.

Nous avons donc vu à quel point la notification des évènements est très maniable et totalement contrôlée par le développeur. Il est aussi nécessaire de comprendre comment ordonner les notifications afin de les maîtriser pleinement.

Gérer l'ordre des notifications

Lors du lancement d'un évènement, il est important de comprendre l'ordre utilisé pour la notification des écouteurs. Le gestionnaire d'évènements se base sur un objet de queue à priorité, la classe `SplPriorityQueue` qui permet de gérer facilement les priorités. Les méthodes « `attach()` » du gestionnaire d'évènements et du gestionnaire partagé `SharedEventManager` possèdent un argument facultatif qui permet de définir la priorité. Voici le prototype des deux méthodes :

Zend/EventManager/EventManager.php

```
|| public function attach($event, $callback = null, $priority = 1);
```

Zend/EventManager/SharedEventManager.php

```
|| public function attach($id, $event, $callback, $priority = 1);
```

Par défaut la priorité est égale à 1. Il est donc possible de définir une priorité plus haute, supérieure ou égale à 1, ce qui permet à l'écouteur d'être notifié le premier, ou une priorité plus basse qui notifiera l'écouteur en fin de liste. Notons aussi qu'il est possible de connaître la priorité maximum de la liste en récupérant l'écouteur avec la plus haute priorité :

Depuis la classe EventManager

```
|| $handler = $events->getListeners('pause')->top();
```

Depuis la classe SharedEventManager

```
|| $handler = $events->getListeners('Developpeur', 'pause')->top();
```

Ces instructions retournent un objet de type `Zend\Stdlib\CallbackHandler` dont

il suffit de récupérer la métadonnée associée à la priorité :

Récupération de la priorité

```
|| $priority_max = $handler->getMetadatum('priority');
```

Nous comprenons maintenant que ces instructions vont être utilisées lors du lancement d'un évènement afin de classer les écouteurs par priorité :

Zend/EventManager/EventManager.php

```
protected function insertListeners($masterListeners, $listeners)
{
    if (!count($listeners)) {
        return;
    }

    foreach ($listeners as $listener) {
        $priority = $listener->getMetadatum('priority');
        if (null === $priority) {
            $priority = 1;
        } elseif (is_array($priority)) {
            $priority = array_shift($priority);
        }
        $masterListeners->insert($listener, $priority);
    }
}
```

L'objet « \$masterListeners » de type PriorityQueue s'occupe d'ajouter les écouteurs par priorité afin de pouvoir les parcourir dans l'ordre choisi par le développeur. Chacun sera ensuite notifié à son tour.

Les évènements du framework

Afin de maîtriser la gestion des évènements qui interviennent dans le framework et des interactions que l'on va pouvoir effectuer dans notre application, nous devons établir la liste des principaux évènements utilisés par le cœur du Zend Framework 2 :

Liste des évènements du gestionnaire de modules :

- Zend\ModuleManager\ModuleManager : « loadModules » lors du début de linitialisation des modules

Pour chacun des modules :

- Zend\ModuleManager\ModuleManager : « loadModule.resolve » lors du chargement du module, résolution de la classe Module du module en cours,
- Zend\ModuleManager\ModuleManager : « loadModule » lorsque le module est chargé,
- Zend\ModuleManager\ModuleManager : « loadModules.post » lorsque les modules sont initialisés.

Liste des évènements de l'application :

- Zend\ Mvc\ Application : « bootstrap » en fin de traitement, lorsque la méthode bootstrap a initialisé les ressources ;
- Zend\ Mvc\ Application : « route » lors de la demande de routage ;
- Zend\ Mvc\ Application : « dispatch » lors de la demande du dispatch de l'action en cours. Cet évènement est écouté par le contrôleur Zend\ Mvc\ Controller\ ActionController qui lance un évènement :
 - Zend\ Mvc\ Controller\ ActionController : « dispatch » lors de la distribution par le contrôleur abstrait ;
- Zend\ Mvc\ Application : « render » lors de la demande du rendu de la vue ;
- Zend\ View\ View : « renderer » lors de la demande du gestionnaire de rendus à utiliser ;
- Zend\ View\ View : « response » lorsque le processus de rendu est terminé, la réponse est prête ;
- Zend\ Mvc\ Application : « finish » lors de la fin du processus MVC, la réponse de l'application est prête et retournée.

Un évènement lié aux erreurs du framework est disponible :

- Zend\ Mvc\ Application : « dispatch.error » lorsqu'une erreur est survenue lors du dispatch de l'action (contrôleur invalide, aucune route correspondante, etc.).

Les évènements présents lors du rendu de la vue ne sont pas listés ici, vous les retrouverez en détail dans le chapitre consacré aux vues.

AVEC LE ZEND FRAMEWORK 1

La première version du framework n'utilise pas les évènements mais des hooks qui permettent de notifier les plugins ou contrôleurs en fonction de la présence ou non d'une méthode de nom donnée. Pour rappel, voici la liste des hooks disponibles sur le Zend Framework 1 :

- « routeStartup() » et « routeShutdown() » pour notifier le début et la fin du routage.
- « dispatchLoopStartup() » et « dispatchLoopShutdown() » qui notifie après le routage (et donc avant la distribution) et après la distribution.
- « preDispatch() » et « postDispatch() » qui permettent aux plugins d'être notifiés du début et de la fin de la distribution sur le contrôleur
- « preDispatch() » et « postDispatch() » qui permettent aux contrôleurs d'être notifiés avant et après l'action courante.

Nous avons maintenant toutes les cartes en main pour mieux comprendre les évènements dans le Zend Framework 2. Cela nous permet de comprendre l'intérêt

de la programmation évènementielle comparée aux hooks du Zend Framework 1 et l'étendue de tout ce qu'il est possible de faire. Nous verrons dans le chapitre consacré à la classe Zend\ Mvc\ Application, point d'entrée de notre application, la gestion détaillée des évènements liés au cœur du framework.

4

L'injection de dépendances

Zend Framework 2 propose dans cette nouvelle version un composant capable de gérer l'injection de dépendances. L'injection de dépendances permet de créer une application où chacun des composants et chacune des classes ne seront plus dépendants fortement les uns des autres. Il n'y a plus de dépendance codée en dur.

Prenons par exemple une classe A qui consomme un objet B, nécessaire à son fonctionnement :

Classe avec des dépendances dans le code

```
class A
{
protected $b;
public function __construct()
{
    $this->b = new B();
}
public function uneaction()
{
    $this->b->quelquechose();
}
```

La classe A est fortement couplée avec la classe B. La dépendance avec B est écrite en dur dans le code, ce qui diminue la flexibilité de notre application.

Reprenons l'exemple avec l'ajout d'une interface, qui permet de s'assurer que l'objet reçu en paramètre sera capable de remplir ses responsabilités en passant un contrat avec cette interface :

Classe sans les dépendances dans le code

```
Class A
{
protected $b;
public function __construct(capabledefaireQuelquechose $b)
{
    $this->b = $b;
}
```

```

public function uneaction()
{
    $this->b->quelquechose();
}

interface capabledefaireQuelquechose
{
    public function quelquechose();
}

class B implements capabledefaireQuelquechose
{
protected $b;
public function quelquechose()
{
    // traitements
}
}

```

L'objet A est alors totalement découplé de l'objet B. Il est maintenant possible de créer un nouvel objet C, qui implémente cette nouvelle interface et que l'on peut donner comme paramètre à l'objet A. Il est aussi nécessaire d'ajouter les méthodes d'altération de A, « getter » et « setter », afin de pouvoir récupérer ou modifier l'objet reçu dans le constructeur.

Examinons l'implémentation de l'injection de dépendances et du composant qui lui est dédié dans le Zend Framework 2.

Configuration manuelle du composant Di

Le composant du framework qui va nous servir à gérer nos injections de dépendances est le Zend\Di\Di. Ce composant doit être configuré afin de lui indiquer quelles vont être les dépendances de chacun des objets. Pour le configurer manuellement, il est possible de lui fournir une liste de définitions sous forme de tableau qui lui permet d'enregistrer une liste de classes avec leurs dépendances. Pour les objets avec une configuration manquante, le composant se chargera lui-même de l'introspection, il est donc préférable d'utiliser une configuration statique afin de gagner en performances.

Voici un exemple de tableau de définitions que l'on pourrait utiliser pour décrire le router de l'application ou encore la stratégie de rendu par défaut, objets dont nous verrons l'utilité ultérieurement :

Configuration du composant d'injection de dépendance

```

$diConfig = new DiConfiguration(array('definition' => array('class' =>
array(
'Zend\Mvc\Router\RouteStack' => array(
'instantiator' => array(
'Zend\Mvc\Router\Http\TreeRouteStack',
'factory'
),
),
),
)
);

```

```

'Zend\Mvc\Router\Http\TreeRouteStack' => array(
    'instantiator' => array(
        'Zend\Mvc\Router\Http\TreeRouteStack',
        'factory'
    ),
),
[...]
'Zend\Mvc\View\DefaultRenderingStrategy' => array(
    'setLayoutTemplate' => array(
        'layoutTemplate' => array(
            'required' => false,
            'type'      => false,
        ),
    ),
),
))));
```

Chaque entrée du sous-tableau « class » du tableau « definition » représente le nom de la classe avec pour valeur un tableau indiquant la procédure afin de l'instancier et les paramètres qui lui sont nécessaires. Nous remarquons que pour obtenir une instance d'objet correspondant à l'identifiant « Zend\Mvc\Router\RouteStack » il est nécessaire d'utiliser la méthode « factory() » de la classe Zend\Mvc\Router\Http\TreeRouteStack.

L'identifiant de la configuration représente un type d'objet, qui n'est pas nécessairement le type de la classe qui sera instanciée. L'identifiant « Zend\Mvc\Router\RouteStack » est en réalité une interface au sein du framework, elle ne peut donc pas réellement être instanciée. Lorsque nous souhaiterons obtenir, depuis notre composant d'injection de dépendances, une instance de la classe Zend\Mvc\Router\RouteStack, le gestionnaire s'occupera de lire la définition de cet objet et pourra alors instancier la classe concernée depuis les paramètres de la définition. Ceci nous permet d'attacher une classe de notre choix aux différentes interfaces, comme ici pour l'interface RouteStack où nous lui affectons la correspondance avec un objet de type TreeRouteStack.

Une fois les définitions créées, nous pouvons indiquer les paramètres que l'on souhaite utiliser avec les définitions lors de l'instanciation. Prenons un exemple de configuration avec l'instanciation d'un adaptateur de base de données et le passage de nos paramètres :

Configuration du Di

```

$di = new Zend\Di\Di;
$diConfig = new Zend\Di\Configuration(
    array('instance' => array(
        'Zend\Db\Adapter\Adapter' => array(
            'parameters' => array(
                'driver' => array(
                    'driver' => 'Pdo',
                    'dsn'    => 'mysql:dbname=db;host=host',
                    'username' => 'username',
                    'password' => 'password'
                ),
            ),
        ),
    )));
$diConfig->configure($di);
$db = $di->get('Zend\Db\Adapter\Adapter');
```

Nous savons maintenant configurer et utiliser le composant d'injection de dépendances. Voyons comment nous pouvons automatiser ce genre de configuration afin d'éviter d'écrire manuellement toutes les configurations dont on pourrait avoir besoin.

Configuration automatique du composant Di

Nous venons de voir comment configurer manuellement le composant d'injection de dépendances. Seulement, nous pouvons nous demander comment celui-ci gère la définition d'une classe dont nous ne lui avons pas fourni de description. Dans ce cas, le framework va alors charger les définitions des classes à la demande, ce qui permet de n'utiliser des ressources que lorsque cela est nécessaire.

Prenons un exemple en lui demandant de récupérer une instance dont il ne connaît pas la définition. La demande d'objet se fait depuis la méthode « `get()` » de la classe Di. Nous ne rentrerons pas dans les détails du gestionnaire d'instances qui sera étudié à la prochaine section.

Demandons par exemple à notre composant l'instance de la classe `Zend\Barcode\Barcode`, définition dont il n'a pas connaissance :

Instanciation de la classe Barcode

```
|| $barcode = $di->get('Zend\Barcode\Barcode');
```

Le composant `Zend\Di\Di` ne connaît pas encore ce composant et va donc tenter de l'instancier afin de nous retourner l'objet demandé si celui-ci n'existe pas déjà dans son gestionnaire d'instances :

Zend/Di/Di.php

```
|| Zend/Di/Di.php
|| public function get($name, array $params = array())
|| {
|| // vérification au sein du gestionnaire
|| [...]
|| $instance = $this->newInstance($name, $params);
|| array_pop($this->instanceContext);
|| return $instance;
|| }
```

Lorsque le Di tente d'instancier la classe souhaitée, il vérifie d'abord si elle existe dans sa liste de définitions au sein de la méthode « `newInstance()` » qui est responsable de la création de nouvelles instances :

Zend/Di/Di.php

```
|| public function newInstance($name, array $params = array(), $isShared
|| = true)
|| {
|| $definitions      = $this->definitions;
|| [...]
|| if (!$definitions->hasClass($class)) {
|| [...] // erreur
```

```

    }
    $instantiator = $definitions->getInstantiator($class);
    [...]
}

```

Nous voyons qu'il est obligatoire que la définition soit renseignée pour ne pas lever d'erreur. Cependant, nous allons voir que la vérification de définition va donner lieu à la création automatique de celle-ci si elle n'existe pas.

L'attribut « \$definitions », objet de type Zend\Di\DefinitionList, qui contient la liste des définitions, est basé sur une liste doublement chaînée, la SplDoublyLinkedList :

Zend/Di/DefinitionList.php

```

class DefinitionList extends SplDoublyLinkedList implements Definition\Definition
{
    public function __construct($definitions)
    {
        if (!is_array($definitions)) {
            $definitions = array($definitions);
        }
        foreach ($definitions as $definition) {
            $this->push($definition);
        }
    }

    public function addDefinition(Definition\Definition $definition,
        $addToBackOfList = true)
    {
        if ($addToBackOfList) {
            $this->push($definition);
        } else {
            $this->unshift($definition);
        }
    }
    [...]
}

```

La liste contient toutes les instances des objets de description de classe qui permettent de décrire le processus d'instanciation d'une classe. Ces objets doivent passer un contrat avec l'interface Zend\Di\Definition\Definition. Parmi les classes qui l'implémentent, nous retrouvons la classe Zend\Di\Definition\ClassDefinition qui encapsule la définition d'une classe. Un autre objet plus intéressant de type Zend\Di\Definition\RuntimeDefinition est capable de récupérer les définitions de classes à la volée. Comme nous pouvons nous en douter, c'est cet objet qui est utilisé par défaut dans le composant d'injection de dépendances lors de l'absence de définition :

Zend/Di/Di.php

```

public function __construct(DefinitionList $definitions = null,
    InstanceManager $instanceManager = null, Configuration $config = null)
{
    $this->definitions = ($definitions) ?: new DefinitionList(new Definition\RuntimeDefinition());
}

```

```

    $this->instanceManager = ($instanceManager) ?: new InstanceManager();
    if ($config) {
        $this->configure($config);
    }
}

```

Par défaut, le composant Di ajoute un objet RuntimeDefinition à la liste des définitions de classe, ce qui lui permet de l'utiliser en cas de définition manquante. Seulement, cet objet sera le dernier des objets de définitions notifiés, car comme nous l'avons vu dans le constructeur de la classe DefinitionList, les objets reçus en paramètre sont, par défaut, ajoutés avec la méthode « `push()` », ce qui les place à la fin de la liste. Si l'on souhaite utiliser un conteneur de définitions en priorité, il est nécessaire de l'ajouter en utilisant la valeur du deuxième paramètre que l'on passera avec la valeur « `true` » :

Zend/Di/DefinitionList.php

```

public function addDefinition(Definition\Definition $definition,
    $addToBackOfList = true)
{
    if ($addToBackOfList) {
        $this->push($definition);
    } else {
        $this->unshift($definition);
    }
}

```

En effet, il est normal que l'objet RuntimeDefinition soit le dernier utilisé afin de trouver une définition car il trouvera toujours la définition d'une classe existante. En effet, le fait de le faire à la volée est moins performant qu'un objet qui contiendrait déjà cette définition, il est donc nécessaire de le placer à la fin si l'on ne souhaite pas systématiquement obtenir les définitions à l'exécution. Analysons comment l'objet RuntimeDefinition crée nos définitions à la volée.

Lorsque la classe RuntimeDefinition est interrogée sur une définition de classe, celle-ci se contente de vérifier l'existence de la classe dans la liste de ses définitions déjà en mémoire, et si elle ne la possède pas, vérifie son existence :

Zend/Di/Definition/RuntimeDefinition.php

```

public function hasClass($class)
{
    if ($this->explicitLookups === true) {
        return (array_key_exists($class, $this->classes));
    }
    return class_exists($class) || interface_exists($class);
}

```

Si les autoloaders sont capables de localiser la classe ou l'interface demandée, notre objet sera alors capable de la charger afin de l'introspecter.

Ensuite lorsque le composant Di reçoit la confirmation de l'existence de la définition, il souhaite connaître la méthode d'instanciation de la classe demandée afin de la créer, et interroge donc la méthode « `getInstantiator()` » de notre conteneur de définition :

Zend/Di/Definition/RuntimeDefinition.php

```

  public function getInstantiator($class)
  {
    if (!array_key_exists($class, $this->classes)) {
      $this->processClass($class);
    }
    return $this->classes[$class]['instantiator'];
  }

```

Le composant vérifie s'il possède la définition au sein de son tableau de définitions en mémoire, et comme ce ne sera pas le cas dans notre exemple, il va ensuite devoir récupérer la définition de la classe depuis une introspection :

Zend/Di/Definition/RuntimeDefinition.php

```

  protected function processClass($class)
  {
    $strategy = $this->introspectionStrategy;
    $rClass = new Reflection\ClassReflection($class);
    $className = $rClass->getName();
    [...]
  }

```

Il n'est pas nécessaire d'analyser le reste de la classe, le code est assez long et nous pouvons voir que l'introspection de la classe est basée sur la classe Zend\Code\Reflection\ClassReflection et Zend\Di\Definition\IntrospectionStrategy qui vont servir à construire la définition.

La configuration automatique, ou à la volée, des définitions est pratique car celle-ci ne se réalise que sur demande, mais est évidemment beaucoup moins performante qu'une configuration manuelle. Afin d'éviter de devoir écrire des centaines de lignes de configuration, nous allons voir comment charger semi-automatiquement les définitions dont on a besoin, en les compilant dans un fichier de configuration.

Configuration semi-automatique du Di

Zend Framework 2 propose plusieurs classes permettant d'englober la gestion des définitions. Nous venons de voir la classe RuntimeDefinition et nous avons aussi parlé de ClassDefinition qui enveloppe une seule et unique définition de classe. Examinons maintenant une classe qui va nous permettre de gérer les définitions en compilant celles d'un dossier complet. Le composant responsable de ce traitement est le Zend\Di\Definition\CompilerDefinition.

Voici tout de suite un exemple d'utilisation de cette classe :

Compilation de définitions

```

$compiler = new \Zend\Di\Definition\CompilerDefinition();
$compiler->addDirectoryScanner(
  new \Zend\Code\Scanner\DirectoryScanner('/path/to/library/Zend/
  Acl')
);

```

```

$compiler->addCodeScannerFile(
    new \Zend\Code\Scanner\FileScanner('/path/to/library/Zend/
Barcode/Barcode.php')
);
$compiler->compile();
$di->definitions()->addDefinition($compiler, false);

```

La classe CompilerDefinition permet d'ajouter un fichier ou un répertoire entier à scanner. Nous pouvons ajouter ce conteneur de définitions à notre liste de gestionnaires, en n'oubliant pas d'indiquer le deuxième paramètre de la méthode « addDefinition() » à la valeur « false » afin de pouvoir ajouter ce gestionnaire en tout début de liste.

La compilation de définitions représente un coût en termes de performances, il est donc conseillé d'envelopper la liste des définitions dans un objet de type ArrayDefinition, ce qui va nous permettre d'exporter nos définitions dans un fichier de cache :

Compilation et cache des définitions

```

if(!file_exists(__DIR__ . '/di-compilier-definitions.php'))
{
    $compiler = new \Zend\Di\Definition\CompilerDefinition();
    $compiler->addDirectoryScanner(
        new \Zend\Code\Scanner\DirectoryScanner('library/Zend/Acl')
    );
    $compiler->addCodeScannerFile(
        new \Zend\Code\Scanner\FileScanner('library/Zend/Barcode/
Barcode.php')
    );
    $compiler->compile();
    file_put_contents(__DIR__ . '/di-compilier-definitions.php', '<?php
    return ' . var_export($compiler->toArrayDefinition()->toArray(), true
    . ';');
}
else
{
    $compiler = new \Zend\Di\Definition\ArrayDefinition( include __DIR__ .
    '/di-compilier-definitions.php' );
}

```

Il est également possible de créer des définitions depuis des objets de type Zend\Di\Definition\Builder\PhpClass que l'on ajoute au conteneur de définitions Zend\Di\Definition\BuilderDefinition :

Utilisation de la classe PhpClass

```

$builder = new \Zend\Di\Definition\BuilderDefinition();
$phpClass = new \Zend\Di\Definition\Builder\PhpClass();
$phpClass->setName('Zend\EventManager\EventManager');
$phpClass->setInstantiator('__construct');
$phpClass->createInjectionMethod('setStaticConnectio
ns')->addParameter('connections', 'Zend\EventManager\
StaticEventCollection', false);
$phpClass->createInjectionMethod('setIdentifiers')->addParameter('iden
tifiers', 'array', false);

```

```

$builder->addClass($phpClass);
$this->getLocator()->definitions()->addDefinition($builder, false);
$events = $this->getLocator()->newInstance('Zend\EventManager\
EventManager', array(
'connections'=>\Zend\EventManager\StaticEventManager::getInstance(),
'identifiers'=>'MonIdentifiant')
);

```

Cette notation permet de décrire facilement la définition de classes PHP. Nous avons pris l'exemple du gestionnaire d'évènements dont on a créé la définition pour l'ajouter à l'objet de type `BuilderDefinition`.

Nous avons parcouru les différents conteneurs de définitions, intéressons-nous maintenant au processus d'instanciation et de partage d'instance du composant d'injection de dépendances.

Le gestionnaire d'instances

Le composant Di fonctionne avec d'autres composants afin de pouvoir déléguer certaines de ses tâches. Il dispose d'une classe permettant de conserver la liste des instances qu'il a créées, afin de lui permettre de les utiliser ou de les partager. La classe utilisée est `Zend\Di\InstanceManager` qui représente le gestionnaire d'instances du composant Di.

Examinons la liste des méthodes du composant Di qui vont nous permettre de récupérer les instances demandées :

Méthodes d'instanciation depuis le Di

```

class Di implements DependencyInjection
{
[...]
public function get($name, array $params = array());
public function newInstance($name, array $params = array(), $isShared
= true);
[...]
}

```

Deux méthodes sont disponibles afin de pouvoir récupérer les instances d'objets. La première méthode « `get()` » permet de récupérer l'instance sans connaître son existence dans le gestionnaire d'instances. Cette méthode vérifie si celle-ci existe dans le gestionnaire et nous la retourne si c'est le cas. Si l'instance n'est pas disponible, le composant Di va alors créer un nouvel objet avant de le stocker automatiquement dans le gestionnaire :

Zend/Di/Di.php

```

public function get($name, array $params = array())
{
[...]
$im = $this->instanceManager;
if ($params) {
    $fastHash = $im->hasSharedInstanceWithParameters($name,
$params, true);
}

```

```

        if ($fastHash) {
            array_pop($this->instanceContext);
            return $im->getSharedInstanceWithParameters(null,
array(), $fastHash);
        }
    } else {
        if ($im->hasSharedInstance($name, $params)) {
            array_pop($this->instanceContext);
            return $im->getSharedInstance($name, $params);
        }
    }
$instance = $this->newInstance($name, $params);
[...]
return $instance;
}

```

Les méthodes « hasSharedInstanceWithParameters() » et « hasSharedInstance() » permettent de vérifier l'existence d'une instance partagée au sein du gestionnaire. Si celle-ci est disponible, les méthodes « getSharedInstanceWithParameters() » et « getSharedInstance() » retournent alors l'objet existant, sinon une nouvelle instance est créée.

La méthode « newInstance() » permet de créer une nouvelle instance de la classe demandée. L'appel direct à la méthode « newInstance() » depuis nos contrôleur sera donc plus couteux en termes de ressources, car celle-ci instancie un nouvel objet sans se soucier de l'existence de celui-ci, mais aura l'avantage de pouvoir s'en servir sans avoir à connaître son contexte d'utilisation dans le framework. Bien évidemment, si le contexte nous importe peu et que l'altération de l'objet par d'autres méthodes du framework n'a pas d'importance, il est préférable d'utiliser la méthode « get() » afin de bénéficier du gain de performances et du contexte de cet objet.

Nous ne détaillerons pas tout le code de la méthode « newInstance() » que nous avons déjà légèrement observé et qui est plutôt long, mais les principales étapes sont les suivantes :

Zend/Di/Di.php

```

public function newInstance($name, array $params = array(), $isShared
= true)
{
[...]
if ($instanceManager->hasAlias($name)) {
    $class = $instanceManager->getClassFromAlias($name);
    $alias = $name;
}
[...]
if (!$definitions->hasClass($class)) {
    [...]
}

$instantiator      = $definitions->getInstantiator($class);
$injectionMethods = $definitions->getMethods($class);
[...]
if ($instantiator === '__construct') {
    $instance = $this->createInstanceViaConstructor($class,
$params, $alias);
}

```

```
        if (array_key_exists('__construct', $injectionMethods)) {
            unset($injectionMethods['__construct']);
        }
    }
    [...]
    if ($isShared) {
        [...]
        else {
            $this->instanceManager->addSharedInstance($instance,
$name);
        }
    }
    [...]
    return $instance;
}
```

Le composant vérifie que le nom demandé n'est pas un alias, auquel cas il récupère le nom réel de la classe, puis vérifie l'existence de la définition. Il récupère ensuite les informations dont il a besoin, depuis les définitions disponibles afin d'instancier la classe comme indiqué dans la configuration. Si le paramètre « \$isShared » de la méthode a pour valeur « true », le gestionnaire s'occupe alors de partager l'instance au sein du manager. Lors de l'appel à la méthode « get() », comme ce paramètre n'est pas disponible, celle-ci appellera toujours la méthode « newInstance() » sans indiquer de valeur de partage. Le paramètre étant positionné à la valeur « true » par défaut, l'instance sera donc automatiquement partagée si celle-ci n'existe pas déjà dans le gestionnaire. Les méthodes d'altération vont ensuite permettre l'injection des paramètres fournis avant de retourner l'instance de l'objet.

5

Le gestionnaire de services

Si le composant d'injection de dépendances présente de nombreux avantages, comme une plus grande flexibilité dans la gestion de nos différents services et de nos dépendances, celui-ci a les inconvénients d'être difficile à appréhender et de dégrader les performances. En effet, si les définitions des classes utilisées ne sont pas connues, le composant Di s'occupe alors de récupérer la définition des objets à l'exécution. Seulement, le fait de précompiler les définitions des composants et bibliothèques ne sera peut-être pas suffisant pour retrouver de meilleures performances, et cela ajouterait une couche de complexité supplémentaire à notre application.

Afin de répondre à cette problématique, un nouveau composant laisse la possibilité aux développeurs de créer leurs propres objets de fabrique sans la gestion automatique des définitions et des dépendances, le Zend\ServiceManager. Ce composant, bien qu'autonome, possède un sous-composant Zend\ServiceManager\Di capable de tirer les avantages de l'injection de dépendances avec la simplicité de ce gestionnaire de services.

Ce nouveau composant permet une approche plus simple et plus performante aux développeurs qui font face à des problématiques de création et de partage de services. Le ServiceManager s'initialise avec des objets de configuration que l'on va examiner afin de comprendre toutes les possibilités de ce gestionnaire de services.

La configuration du gestionnaire

Le gestionnaire Zend\ServiceManager\ServiceManager s'initialise grâce à un objet de type Zend\ServiceManager\ConfigInterface :

Zend/ServiceManager/ServiceManager.php

```
public function __construct(ConfigurationInterface $configuration = null)
{
    if ($configuration) {
        $configuration->configureServiceManager($this);
    }
}
```

Une classe de base implémentant l'interface de configuration Zend\ServiceManager

ger\ConfigInterface existe dans le framework :

Zend/ServiceManager/Config.php

```
class Config implements ConfigInterface
{
    protected $config = array();
    public function __construct($config = array())
    {
        $this->config = $config;
    }

    public function getFactories()
    {
        return (isset($this->config['factories'])) ? $this->config['factories'] : array();
    }

    public function getAbstractFactories()
    {
        return (isset($this->config['abstract_factories'])) ? $this->config['abstract_factories'] : array();
    }
    [...]

    public function configureServiceManager(ServiceManager $serviceManager)
    {
        $allowOverride = $this->getAllowOverride();
        isset($allowOverride) ? $serviceManager->setAllowOverride($allowOverride) : null;
        foreach ($this->getFactories() as $name => $factory) {
            $serviceManager->setFactory($name, $factory);
        }
        foreach ($this->getAbstractFactories() as $factory) {
            $serviceManager->addAbstractFactory($factory);
        }
        [...]
    }
}
```

La classe de configuration encapsule la configuration du gestionnaire de services afin de pouvoir l'initialiser depuis ses méthodes d'altération du même nom. Voici les différents attributs de l'objet de configuration :

- « *invokables* » : les classes renseignées comme étant « *invokables* » seront instanciées directement sans la nécessité d'utiliser une fabrique. Ce type conviendra aux objets sans dépendance particulière ;
- « *factories* » : la configuration liée à la clé « *factories* » permet de définir la liste des classes de fabrique. Une classe de fabrique est associée à un type d'objet et retourne une instance de celui-ci ;
- « *abstract_factories* » : les fabriques abstraites ressemblent aux classes de fabrique à la différence que celles-ci ne sont pas attachées à un objet en particulier, elles sont appelées lorsqu'aucune fabrique n'existe pour l'objet demandé ;
- « *services* » : les valeurs renseignées dans le champ « *services* » peuvent être de tout type (entier, chaîne de caractères, tableau, objets, etc.) et sont associées à la

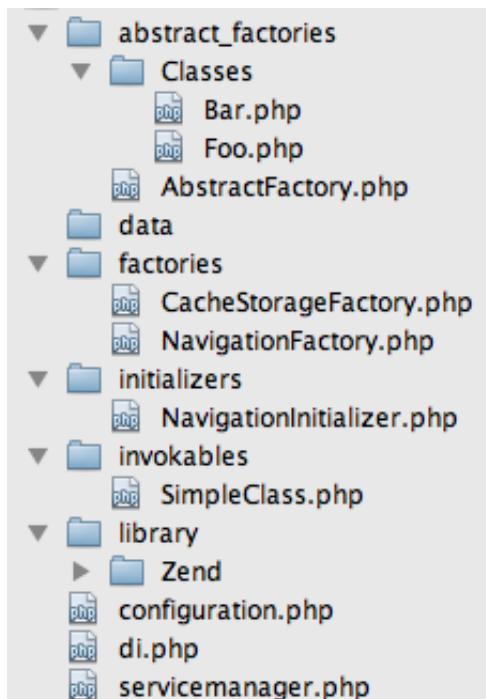
clé donnée ;

- « aliases » : les alias permettent de définir plusieurs clés pour un même objet. Il est aussi possible de faire des alias d'alias autant que l'on souhaite ;
- « shared » : les services que l'on définit comme partagés permettent de ne pas instancier un nouvel objet à chaque appel. L'instance créée est enregistrée afin d'être partagée à chaque appel ;
- « initializers » : la configuration liée à cette clé permet d'initialiser les objets une fois instanciés ;

Maintenant que nous connaissons les configurations possibles pour le gestionnaire de services, voyons comment les mettre en œuvre.

Mise en œuvre

Afin de comprendre les différentes possibilités du ServiceManager, nous partirons de l'architecture d'exemple ci-dessous pour l'illustration de nos propos :



Le script d'exemple utilisé est le fichier « `servicemanager.php` », le fichier « `configuration.php` » est un fichier de configuration et le fichier « `di.php` » est le fichier pour l'initialisation du composant Zend\Di.

Commençons par instancier la classe ServiceManager :

Exemple de configuration

```

$serviceManager = new ServiceManager(new Configuration(array(
    'invokables' => array(
        'simple' => 'Invokables\SimpleClass',
    ),
    'factories' => array(
        'cache' => 'Factories\CacheStorageFactory',
        'navigation' => 'Factories\NavigationFactory',
    ),
    'abstract_factories' => array(
        'generic' => 'AbstractFactories\AbstractFactory',
    ),
    'aliases' => array(
        'simpleclass' => 'simple',
    ),
));
$serviceManager->setService('Configuration', include 'configuration.
php');

```

La première ligne de notre configuration indique que nous avons accès à la classe SimpleClass de l'espace de noms « invokables » depuis l'identifiant « simple ». Il est alors possible d'obtenir une instance de cette classe depuis l'instruction suivante :

Utilisation des classes invokables

```

$simpleClass = $serviceManager->get('simple');
echo get_class($simpleClass);

```

Affichage de la sortie

```

Invokables\SimpleClass

```

La récupération d'instances existantes ou la création de celles-ci passe par la méthode « `get()` », nom de méthode que l'on utilisait déjà avec notre composant d'injection de dépendances.

Afin de comprendre comment est retourné l'objet désiré, analysons la création d'instances depuis le `ServiceManager` avec la méthode « `get()` » :

Zend/ServiceManager/ServiceManager.php

```

public function get($name, $usePeeringServiceManagers = true)
{
    $cName = $this->canonicalizeName($name);
    $rName = $name;

    if ($this->hasAlias($cName)) {
        do {
            $cName = $this->aliases[$cName];
        } while ($this->hasAlias($cName));
        if (!$this->has(array($cName, $rName))) {
            [...]
        }
    }

    if (isset($this->instances[$cName])) {
        return $this->instances[$cName];
    }
}

```

```

$instance = null;
$retrieveFromPeeringManagerFirst = $this->retrieveFromPeeringManagerFirst();
if ($usePeeringServiceManagers && $retrieveFromPeeringManagerFirst) {
    $instance = $this->retrieveFromPeeringManager($name);
}
if (!$instance) {
    if ($this->canCreate(array($cName, $rName))) {
        $instance = $this->create(array($cName, $rName));
    } elseif ($usePeeringServiceManagers &&
    !$retrieveFromPeeringManagerFirst) {
        $instance = $this->retrieveFromPeeringManager($name);
    }
}
if (!$instance && !is_array($instance)) {
    [...]
}
if ($this->shareByDefault() && (!isset($this->shared[$cName]) || $this->shared[$cName] === true))
{
    $this->instances[$cName] = $instance;
}
return $instance;
}

```

La première instruction formate le nom demandé avant de vérifier si celui-ci n'est pas un alias. Le fait de vérifier l'existence d'alias en boucle permet d'ajouter des alias d'alias autant que l'on souhaite.

Le tableau « \$instances » est ensuite contrôlé afin de vérifier l'existence de l'instance de l'objet demandé.

Pour rappel, les objets instanciés partagés et les valeurs passées lors de notre configuration depuis le paramètre « services » (entier, chaîne de caractères, tableau, objets, etc.) sont enregistrés dans cet attribut :

Zend/ServiceManager/Configuration.php

```

public function configureServiceManager(ServiceManager $serviceManager)
{
    [...]
    foreach ($this->getServices() as $name => $service) {
        $serviceManager->setService($name, $service);
    }
    [...]
}

```

Zend/ServiceManager/ServiceManager.php

```

public function setService($name, $service, $shared = true)
{
    [...]
    $this->instances[$name] = $service;
    $this->shared[$name] = (bool) $shared;
    return $this;
}

```

Si aucun objet ne correspond à la requête dans le tableau d'instances, la procédure de création va être effectuée. Pour comprendre la procédure de création il faut garder à l'esprit que chaque gestionnaire de services peut fonctionner avec d'autres instances de ce même objet en tant que gestionnaire parent ou gestionnaire enfant. En effet, il est possible de relier plusieurs gestionnaires de services entre eux afin de tirer parti des fabriques de chacun d'entre eux, tout en permettant de séparer et redéfinir chaque fabrique au sein des différents gestionnaires. Les méthodes « `createScopedServiceManager()` » et « `addPeeringServiceManager()` » permettent de relier un gestionnaire de services à l'objet courant :

Zend/ServiceManager/ServiceManager.php

```
public function createScopedServiceManager($peering = self::SCOPE_PARENT)
{
    $scopedServiceManager = new ServiceManager();
    if ($peering == self::SCOPE_PARENT) {
        $scopedServiceManager->peeringServiceManagers[] = $this;
    }
    if ($peering == self::SCOPE_CHILD) {
        $this->peeringServiceManagers[] = $scopedServiceManager;
    }
    return $scopedServiceManager;
}
```

Cette méthode permet d'attacher un nouveau gestionnaire à un autre. Par défaut, l'objet courant est attaché comme gestionnaire partagé au nouveau. Si nous indiquons que la liaison doit se faire comme un objet enfant, alors le nouveau gestionnaire sera inscrit dans la liste des gestionnaires partagés de l'objet courant. La méthode « `addPeeringServiceManager()` » réalise la même chose mais avec un gestionnaire existant.

Lors de la création de l'objet demandé, nous avons le choix d'utiliser ou non en priorité les gestionnaires partagés pour l'instanciation de l'objet. Ce choix peut être réalisé depuis la méthode « `setRetrieveFromPeeringManagerFirst()` » qui permet de définir un booléen afin d'indiquer si l'on doit utiliser ou non les gestionnaires partagés en premier. La méthode « `retrieveFromPeeringManagerFirst()` » permet de consulter l'état de ce booléen. La procédure d'instanciation se charge alors de vérifier cet état avant de procéder à la création depuis le gestionnaire choisi :

Zend/ServiceManager/ServiceManager.php

```
public function get($name, $usePeeringServiceManagers = true)
{
    [...]
    $instance = null;
    $retrieveFromPeeringManagerFirst = $this->retrieveFromPeeringManagerFirst();
    if ($usePeeringServiceManagers && $retrieveFromPeeringManagerFirst) {
        $instance = $this->retrieveFromPeeringManager($name);
    }
    if (!$instance) {
        if ($this->canCreate(array($cName, $rName))) {
            $instance = $this->create(array($cName, $rName));
        } elseif ($usePeeringServiceManagers &&
        !$retrieveFromPeeringManagerFirst) {
            $instance = $this->retrieveFromPeeringManager($name);
        }
    }
}
```

```
    }
}
[...]
}
```

La méthode « `retrieveFromPeeringManager()` » permet de récupérer l’instance depuis un gestionnaire partagé. Cette méthode se contente d’appeler la méthode « `get()` » de ses gestionnaires :

Zend/ServiceManager/ServiceManager.php

```
protected function retrieveFromPeeringManager($name)
{
    foreach ($this->peeringServiceManagers as $peeringServiceManager) {
        if ($peeringServiceManager->has($name)) {
            return $peeringServiceManager->get($name);
        }
    }
    return null;
}
```

Le gestionnaire partagé peut lui aussi avoir des gestionnaires partagés et ainsi de suite, ce qui génère des appels récursifs et peut complexifier la compréhension de la création d’un objet. Il est donc conseillé de ne pas abuser des partages de gestionnaires et d’en maîtriser le processus.

Lors de la création de l’objet, la première vérification de la méthode « `create()` » porte sur la présence d’une fabrique avant de vérifier le tableau de classes instantiables marquées comme « `invokables` » :

Zend/ServiceManager/ServiceManager.php

```
public function create($name)
{
    $instance = false;
    if (is_array($name)) {
        list($cName, $rName) = $name;
    } else {
        $rName = $name;
        $cName = $this->canonicalizeName($rName);
    }

    if (isset($this->factories[$cName])) {
        $instance = $this->createFromFactory($cName, $rName);
    }
    if (!$instance && isset($this->invokableClasses[$cName])) {
        $instance = $this->createFromInvokable($cName, $rName);
    }
    [...]
}
```

La méthode « `createFromFactory()` » crée l’objet depuis la méthode « `createService()` » de la fabrique si celle-ci implémente l’interface `FactoryInterface` :

Zend/ServiceManager/ServiceManager.php

```

protected function createFromFactory($canonicalName, $requestedName)
{
[...]
if ($factory instanceof FactoryInterface) {
    $instance = $this->createServiceViaCallback(array($factory,
'createService'), $canonicalName, $requestedName);
} elseif (is_callable($factory)) {
    $instance = $this->createServiceViaCallback($factory,
$canonicalName, $requestedName);
} else {
    [...]
}
return $instance;
}

```

L'interface FactoryInterface ne définit que la méthode de construction :

Zend/ServiceManager/FactoryInterface.php

```

interface FactoryInterface
{
public function createService(ServiceLocatorInterface
$serviceLocator);
}

```

Notons que la création depuis une fabrique transite par la méthode « createServiceViaCallback() » qui permet de se prémunir contre les références circulaires. En effet, les fabriques reçoivent l'instance du gestionnaire de services, ce qui peut entraîner des boucles si l'on demande des objets qui dépendent chacun les uns des autres.

Afin d'illustrer nos propos, voici un exemple de fabrique avec la fabrication d'un simple cache :

Utilisation des fabriques

```

$cache = $serviceManager->get('cache');
echo get_class($cache);

```

Affichage de la sortie

```

Zend\Cache\Storage\Adapter\Filesystem

```

La configuration indique que la fabrique correspondant à l'objet d'identifiant « cache » est la classe Factories\CacheStorageFactory dont voici le code :

Factories\CacheStorageFactory.php

```

class CacheStorageFactory implements FactoryInterface
{
public function createService(ServiceLocatorInterface
$serviceLocator)
{
    $configuration = $serviceLocator->get('Configuration');
}

```

```
    return StorageFactory::factory($configuration['cache']);  
}  
}
```

La fabrique de cache récupère aussi la configuration que l'on avait ajoutée depuis l'instruction de notre exemple :

Ajout de la configuration au gestionnaire de services

```
$serviceManager->setService('Configuration', include 'configuration.  
php');
```

Ce fichier contient la configuration globale de notre exemple. La fabrique l'utilise alors pour créer l'instance de l'objet demandé.

Lors de la création, si aucune fabrique n'existe pour l'objet demandé, une vérification est faite sur les classes de type « invokable » :

Zend/ServiceManager/ServiceManager.php

```
public function create($name)  
{  
    [...]  
    if (isset($this->factories[$cName])) {  
        $instance = $this->createFromFactory($cName, $rName);  
    }  
    if (!$instance && isset($this->invokableClasses[$cName])) {  
        $instance = $this->createFromInvokable($cName, $rName);  
    }  
    [...]  
}
```

La méthode « createFromInvokable() » gère les classes instanciables directement et se contente d'instancier l'objet demandé depuis l'opérateur « new » :

Zend/ServiceManager/ServiceManager.php

```
protected function createFromInvokable($canonicalName,  
$requestedName)  
{  
    $invokable = $this->invokableClasses[$canonicalName];  
    if (!class_exists($invokable)) {  
        [...]  
    }  
    $instance = new $invokable;  
    return $instance;  
}
```

Encore une fois, si aucun objet de fabrique ne correspond à l'objet à instancier, les fabriques abstraites vont être parcourues afin de savoir si l'une d'entre elles peut traiter l'instanciation de l'objet demandé. Les classes abstraites implémentent l'interface Zend\ServiceManager\AbstractFactoryInterface qui définit deux méthodes :

Zend/ServiceManager/AbstractFactoryInterface.php

```
interface AbstractFactoryInterface
{
    public function canCreateServiceWithName(ServiceLocatorInterface
        $serviceLocator, $name, $requestedName);

    public function createServiceWithName(ServiceLocatorInterface
        $serviceLocator, $name, $requestedName);
}
```

La méthode « `canCreateServiceWithName()` » permet de s'assurer que la fabrique peut instancier l'objet demandé. En effet, cette fabrique est généraliste et n'est pas rattachée à un type particulier, il faut donc s'assurer qu'elle peut effectuer le travail demandé. La méthode « `createServiceWithName()` » crée l'instance de l'objet et reçoit l'instance du gestionnaire de services ainsi que le nom de la classe de l'objet souhaité.

Lors de la création d'objets, chaque fabrique abstraite est donc parcourue jusqu'à ce qu'une d'entre elles retourne une instance d'objet :

Zend/ServiceManager/ServiceManager.php

```
public function create($name)
{
    [...]
    if (!$instance && $this->canCreateFromAbstractFactory($cName,
        $rName)) {
        $instance = $this->createFromAbstractFactory($cName, $rName);
    }
    [...]
}
```

Zend/ServiceManager/ServiceManager.php

```
protected function createFromAbstractFactory($canonicalName,
    $requestedName)
{
    foreach ($this->abstractFactories as $index => $abstractFactory) {
        [...]
        try {
            $this->pendingAbstractFactoryRequests[get_
                class($abstractFactory)] = $requestedName;
            $instance = $this->createServiceViaCallback(
                array($abstractFactory, 'createServiceWithName'),
                $canonicalName,
                $requestedName
            );
            unset($this->pendingAbstractFactoryRequests[get_
                class($abstractFactory)]);
        } catch (\Exception $e) {
            [...]
        }
        [...]
    }
    return $instance;
}
```

Prenons un exemple d'utilisation avec la hiérarchie présentée en début de chapitre :

Utilisation des fabriques abstraites

```
|| $bar = $serviceManager->get('bar');  
|| echo get_class($bar);
```

Affichage de la sortie

```
|| AbstractFactories\Classes\Bar
```

Voici le code de la fabrique abstraite AbstractFactories\AbstractFactory de notre exemple :

AbstractFactories/AbstractFactory.php

```
|| class AbstractFactory implements AbstractFactoryInterface  
|| {  
||     public function canCreateServiceWithName(ServiceLocatorInterface  
||         $serviceLocator, $name, $requestedName)  
||     {  
||         return in_array(strtolower($name), array('foo', 'bar'));  
||     }  
||  
||     public function createServiceWithName(ServiceLocatorInterface  
||         $serviceLocator, $name, $requestedName)  
||     {  
||         $className = 'AbstractFactories\\Classes\\' . ucfirst($name);  
||         return new $className;  
||     }  
|| }
```

La fabrique abstraite est capable d'instancier les objets correspondant aux identifiants « foo » et « bar ». L'exemple ici est très simple, la méthode « createServiceWithName() » se contente d'instancier l'objet sans plus d'instructions.

Notons que la méthode reçoit deux paramètres différents sur le nom de l'instance demandée : « \$name » et « \$requestedName ». Le second paramètre est le nom utilisé lors de l'appel et qui correspond à l'alias si celui-ci existe, alors que le paramètre « \$name » correspond à la résolution de l'alias. Si aucun alias n'existe, alors les deux paramètres sont identiques. Prenons par exemple la création du type « bar-alias » que nous avons défini dans notre configuration d'exemple :

Utilisation d'un alias

```
|| $bar = $serviceManager->get('bar-alias');
```

Dans cet exemple, la variable « \$name » vaut « bar » et la variable « \$requestedName » vaut « bar-alias ».

Une fois les types de création parcourus (fabrique d'objets, classe « invokable », fabrique abstraite), l'objet instancié peut alors être initialisé si l'on a ajouté nos classes d'initialisation au sein du gestionnaire de services :

Zend/ServiceManager/ServiceManager.php

```

public function create($name)
{
[...]
foreach ($this->initializers as $initializer) {
    if ($initializer instanceof InitializerInterface) {
        $initializer->initialize($instance, $this);
    } elseif (is_object($initializer) && is_callable($initializer)) {
        $initializer($instance, $this);
    } else {
        call_user_func($initializer, $instance, $this);
    }
}
return $instance;
}

```

Une interface est disponible pour les classes d'initialisation :

Zend/ServiceManager/InitializerInterface.php

```

interface InitializerInterface
{
public function initialize($instance, ServiceLocatorInterface
$serviceLocator);
}

```

Voici un exemple de classe d'initialisation pour l'objet de navigation, de type Zend\Navigation\Navigation :

Utilisation des classes d'initialisation

```

$navigationInitializer = new Initializers\NavigationInitializer(array
(array('uri' => 'zend.com'), array('uri' => 'zend.fr')));
$serviceManager->addInitializer($navigationInitializer);

```

Initializers/NavigationInitializer.php

```

class NavigationInitializer implements InitializerInterface
{
protected $pages = array();

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

public function initialize($instance, ServiceLocatorInterface
$serviceLocator)
{
    if(!$instance instanceof Navigation) {
        return;
    }
    if($this->pages && count($instance->getPages()) == 0) {
        $instance->addPages($this->pages);
    }
}
}

```

L'initialisation et l'ajout de pages dans le composant de navigation se font depuis

la classe d'initialisation.

La création d'objets depuis le gestionnaire de services offre une plus grande souplesse aux développeurs pour la mise en place de fabriques d'objets. Ce composant permet aussi de gagner en performances, comparé à la création d'objets depuis le composant Di. Ce dernier peut aussi fonctionner en complément du gestionnaire de services.

Le ServiceManager complémentaire du Di

Le composant ServiceManager n'a pas été introduit afin de remplacer le composant d'injection de dépendances, mais afin de lui être complémentaire en offrant d'autres possibilités de fabriques d'objets. Les deux composants peuvent fonctionner ensemble, une passerelle existe avec le sous-composant Zend\ServiceManager\Di.

La fabrique du Zend\ServiceManager\Di

Analysons l'implémentation d'une fabrique basée sur le composant Di. Prenons un exemple de création depuis le composant d'injection de dépendances et un autre avec une fabrique utilisant le ServiceManager couplé au Di :

Utilisation du Di

```
|| $di = new Di;  
|| $config = new DiConfiguration(include 'di.php');  
|| $config->configure($di);  
  
|| $navigation = $di->get('navigation');  
|| echo get_class($navigation) . "\n";  
|| echo count($navigation->getPages());
```

Affichage de la sortie

```
|| Zend\Navigation\Navigation  
|| 1
```

Utilisation du Di avec le ServiceManager

```
|| $diFactory = new DiServiceFactory($di, 'navigation', array(),  
|| DiServiceFactory::USE_SL_AFTER_DI)  
|| $serviceManager->setFactory('di-navigation-after', $diFactory);  
  
|| $navigation = $serviceManager->get('di-navigation-after')  
|| echo get_class($navigation) . "\n";  
  
|| echo count($navigation->getPages());
```

Affichage de la sortie

```
|| Zend\Navigation\Navigation  
|| 1
```

Voici la configuration du composant d'injection de dépendances :

```
di.php
|| return array(
||   'instance' => array(
||     'aliases' => array(
||       'navigation' => 'Zend\Navigation\Navigation',
||       'json' => 'Zend\Json\Json',
||     ),
||     'Zend\Navigation\Navigation' => array(
||       'parameters' => array(
||         'pages' => array(
||           array('uri' => 'zend.com'),
||         ),
||       ),
||     ),
||   ),
|| );
```

La classe `DiServiceFactory` prend une instance du composant `Di` en paramètre, le nom de l'objet que la fabrique prend en charge, et la position du `ServiceManager` par rapport au `Di` lors de la création d'instances. Le gestionnaire de services peut être utilisé avant, paramètre « `DiServiceFactory::USE_SL_BEFORE_DI` », ou après, paramètre « `DiServiceFactory::USE_SL_AFTER_DI` », le composant d'injection de dépendances.

Dans les deux exemples, c'est le composant `Di` qui prend en charge la création de l'objet de navigation. En effet, le gestionnaire de services est explicitement utilisé avec le paramètre « `DiServiceFactory::USE_SL_AFTER_DI` », ce qui le placera alors après le `Di` lors de la création d'objets. Dans les deux cas, l'initialisation est la même et une seule page existe au sein du composant de navigation.

Gardons le premier exemple et modifions le deuxième en demandant l'utilisation du `ServiceManager` avant le composant `Di` afin que ce dernier soit utilisé si seulement aucune fabrique ne correspond à l'objet demandé au sein du gestionnaire de services :

Utilisation du gestionnaire avant le Di

```
$diFactory = new DiServiceFactory($di, 'navigation', array(),
DiServiceFactory::USE_SL_BEFORE_DI);

$serviceManager->setFactory('di-navigation-before', $diFactory)
$navigation = $serviceManager->get('di-navigation-before');
echo get_class($navigation) . "\n";

echo count($navigation->getPages());
```

Affichage de la sortie

```
|| Zend\Navigation\Navigation
|| 2
```

Le composant de navigation est cette fois instancié et initialisé par le `ServiceManager` qui lui ajoute deux pages.

La classe `DiServiceFactory` est simple à analyser, elle se contente de vérifier la position du gestionnaire de services afin de savoir lequel utiliser en premier :

Zend/ServiceManager/Di/DiServiceFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $this->serviceLocator = $serviceLocator;
    return $this->get($this->name, $this->parameters, true);
}
```

Zend/ServiceManager/Di/DiServiceFactory.php

```
public function get($name, array $params = array())
{
    if ($this->useServiceLocator == self::USE_SL_BEFORE_DI && $this-
>serviceLocator->has($name)) {
        return $this->serviceLocator->get($name);
    }
    try {
        $service = parent::get($name, $params)
        return $service
    } catch (DiClassNotFoundException $e) {
        if ($this->useServiceLocator == self::USE_SL_AFTER_DI && $this-
>serviceLocator->has($name)) {
            return $this->serviceLocator->get($name);
        } else {
            [...] // exception
        }
    }
}
```

Il est aussi possible de désactiver l'utilisation du ServiceManager en utilisant le paramètre « DiServiceFactory ::USE_SL_NONE » lors de sa construction.

La fabrique abstraite du Zend\ServiceManager\Di

Une fabrique abstraite basée sur le composant d'injection de dépendances est disponible, il s'agit de la classe Zend\ServiceManager\Di\DiAbstractServiceFactory. Cette classe plus générale permet l'ajout du composant Di sur l'ensemble des chargements du ServiceManager. Voici comment ajouter une fabrique abstraite au ServiceManager qui utilisera le composant d'injection de dépendances :

Utilisation de la fabrique abstraite basée sur le Di

```
$di = new Di;
$config = new DiConfiguration(include 'di.php');

$config->configure($di);
$diAbstractFactory = new DiAbstractServiceFactory($di);

$serviceManager->addAbstractFactory($diAbstractFactory);
```

Comme pour la fabrique, si nous ne précisons pas la position du gestionnaire de services au sein de la classe abstraite, celui-ci ne sera pas utilisé. En effet, l'intérêt d'utiliser le gestionnaire de services en premier est limité dans ce cas, le but étant d'ajouter le composant d'injection de dépendances en dernière couche comme fabrique abstraite, afin de l'utiliser uniquement si aucune de nos fabriques ne

convient pour l'objet demandé. Il devient alors possible de récupérer n'importe quel composant géré par le Di, comme le composant de date dont nous avons ajouté l'alias précédemment :

Récupération du composant JSON

```
|| $json = $serviceManager->get('json');
|| echo $json->encode(array('foo' => 'bar'));
```

Affichage de la sortie

```
|| {"foo":"bar"}
```

Il est aussi possible de le récupérer directement depuis le nom de sa classe, comme pour l'utilisation classique du Di :

Récupération du composant JSON

```
|| $json = $serviceManager->get('Zend\Json\Json');
|| echo $json->encode(array('foo' => 'bar'));
```

Affichage de la sortie

```
|| {"foo":"bar"}
```

Flexible et performant, le gestionnaire de services a aussi été intégré dans le processus MVC du framework où il occupe une grande place.

Implémentation dans le framework

Le gestionnaire de services est utilisé pour construire les principaux éléments du framework comme le gestionnaire de modules, de vues ou la classe Application, chef d'orchestre du processus MVC.

Le gestionnaire de services utilisé par le processus MVC est construit dans la méthode d'initialisation de la classe Application, point d'entrée de notre application Web :

Zend/Mvc/Application.php

```
|| public static function init($configuration = array())
|| {
||     $smConfig = isset($configuration['service_manager']) ?
||     $configuration['service_manager'] : array();
|
||     $serviceManager = new ServiceManager(new Service\ServiceManagerConfig(
||         $smConfig));
||     [...]
|| }
```

Comme nous le remarquons, l'objet Zend\Mvc\Service\ServiceManagerConfig définit les fabriques de base du framework.

Zend/Mvc/Service/ServiceManagerConfig.php

```
class ServiceManagerConfig implements ConfigInterface
{
    protected $invokables = array(
        'SharedEventManager' => 'Zend\EventManager\SharedEventManager',
    );
    protected $factories = array(
        'EventManager' => 'Zend\Mvc\Service\EventManagerFactory',
        'ModuleManager' => 'Zend\Mvc\Service\ModuleManagerFactory',
    );
    protected $abstractFactories = array();
    protected $aliases = array(
        'Zend\EventManager\EventManagerInterface' => 'EventManager',
    );
    protected $shared = array(
        'EventManager' => false,
    );
    [...]
}
```

Nous remarquons que la fabrique du gestionnaire d'évènements ne partage pas son instance afin de pouvoir en créer une nouvelle à chaque demande.

Les autres fabriques définissant les objets utilisés par le processus MVC sont définies par la classe `Zend\Mvc\Service\ServiceListenerFactory`, écouteur permettant d'enrichir la configuration du gestionnaire de services qui définit les fabriques principales du framework :

Zend/Mvc/Service/ServiceListenerFactory.php

```
class ServiceListenerFactory implements FactoryInterface
{
    [...]
    protected $defaultServiceConfig = array(
        'invokables' => array(
            'DispatchListener' => 'Zend\Mvc\DispatchListener',
            'RouteListener' => 'Zend\Mvc\RouteListener',
        ),
        'factories' => array(
            'Application' => 'Zend\Mvc\Service\ApplicationFactory',
            'Config' => 'Zend\Mvc\Service\ConfigFactory',
            [...]
        ),
        'aliases' => array(
            'Configuration' => 'Config',
            'Console' => 'ConsoleAdapter',
            'ControllerPluginBroker' => 'ControllerPluginManager',
        ),
    );
    [...]
}
```

L'attribut « factories » contient la liste des fabriques qui permettront l'initialisation des composants MVC du framework. Ces fabriques seront détaillées dans les chapitres concernés par chacune d'entre elles.

6

Les modules

Dans le Zend Framework 2, la notion de module a évolué. Un module est un composant indépendant qui possède au minimum son propre espace de noms ainsi qu'une classe nommée `Module` dans un fichier `Module.php`.

Chaque module possède ses fichiers de configuration, ainsi que ses contrôleurs et vues et chacun peut maintenant être indépendant de l'environnement afin d'être réutilisé facilement dans une autre application. Sa configuration étant propre à son fonctionnement, son installation peut donc se faire simplement en ajoutant son nom et son chemin à la configuration du gestionnaire de modules, ainsi qu'en adaptant la configuration du module à sa propre application.

Configuration et écouteurs

Comme nous l'avons vu dans le chapitre précédent, c'est le gestionnaire de services qui est chargé de fabriquer le gestionnaire de modules. Nous remarquons depuis la classe `Zend\ Mvc\ Application`, point d'entrée de notre application, que la fabrique est enregistrée sous la clé « `ModuleManager` » :

`Zend/ Mvc/ Application.php`

```
public static function init($configuration = array())
{
    [...]
    $serviceManager->get('ModuleManager')->loadModules();
    return $serviceManager->get('Application')->bootstrap();
}
```

Analysons le comportement de la fabrique `ModuleManagerFactory` et le rôle qu'elle occupe au sein du framework. Nous aborderons les principales étapes avant de les détailler dans les sections suivantes. Voici le code correspondant à la création du service :

`Zend/ Mvc/ Service/ ModuleManagerFactory.php`

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    if (!$serviceLocator->has('ServiceListener')) {
```

```

    $serviceLocator->setFactory('ServiceListener', 'Zend\Mvc\Service\ServiceListenerFactory');
}
$configuration = $serviceLocator->get('ApplicationConfig');
$listenerOptions = new ListenerOptions($configuration['module_listener_options']);
$defaultListeners = new DefaultListenerAggregate($listenerOptions);
$serviceListener = $serviceLocator->get('ServiceListener');
[...]
$events = $serviceLocator->get('EventManager');
$events->attach($defaultListeners);
$events->attach($serviceListener);

$moduleEvent = new ModuleEvent;
$moduleEvent->setParam('ServiceManager', $serviceLocator);
$moduleManager = new ModuleManager($configuration['modules'], $events);
$moduleManager->setEvent($moduleEvent);
return $moduleManager;
}

```

La première instruction définit la fabrique de l'écouteur du gestionnaire de services, si celle-ci ne l'est pas déjà, qui permet d'initialiser les fabriques du framework par défaut qui sont utilisées tout au long du framework :

Zend/Mvc/Service/ModuleManagerFactory.php

```

public function createService(ServiceLocatorInterface
$serviceLocator)
{
if (!$serviceLocator->has('ServiceListener')) {
    $serviceLocator->setFactory('ServiceListener', 'Zend\Mvc\Service\ServiceListenerFactory');
}
[...]
}

```

Nous verrons plus loin la liste des fabriques de l'écouteur du gestionnaire de services. La configuration globale de l'application est ensuite récupérée :

Zend/Mvc/Service/ModuleManagerFactory.php

```

public function createService(ServiceLocatorInterface
$serviceLocator)
{
[...]
$configuration = $serviceLocator->get('ApplicationConfiguration');
[...]
}

```

Nous prendrons, comme exemple, le fichier de configuration suivant :

application.config.php

```

<?php
return array(
'modules' => array(
    'Application',
),
)

```

```
    'module_listener_options' => array(
        'config_cache_enabled' => false,
        'cache_dir' => 'data/cache',
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

L'instruction suivante concerne l'initialisation de l'objet responsable des options du gestionnaire de modules avec la création d'un objet de type Zend\ModuleManager\Listener\ListenerOptions :

Zend/Mvc/Service/ModuleManagerFactory.php

```
    public function createService(ServiceLocatorInterface
    $serviceLocator)
    {
    [...]
    $listenerOptions = new ListenerOptions($configuration['module_
    listener_options']);
    [...]
}
```

L'objet ListenerOptions enregistre les options que l'on utilisera pour la gestion des modules, il possède peu d'attributs :

Zend/ModuleManager/Listener/ListenerOptions.php

```
class ListenerOptions extends Options
{
    protected $modulePaths = array();

    protected $configGlobPaths = array();

    protected $configStaticPaths = array();

    protected $extraConfig = array();

    protected $configCacheEnabled = false;

    protected $configCacheKey;

    protected $cacheDir;
    [...]
}
```

L'attribut « \$modulePaths » est un tableau qui contient la liste des chemins où se situent nos modules. Le premier chemin « ./module » contient les modules propres à notre application et « ./vendor » la liste des modules externes ajoutés à l'application. Les trois attributs suivants : « \$configGlobPaths », « \$configStaticPaths », « \$extraConfig » permettent de stocker les chemins de fichiers de configuration globale à l'application. L'attribut suivant « \$configCacheEnabled » permet d'activer ou non le cache interne de configuration des modules. Ce cache, sous forme de fichier, sera stocké dans le dossier dont le chemin est représenté par l'attribut « \$cacheDir ». Le nom du cache aura pour nom « module-config-cache.ma-cle.php » où « ma-cle » sera remplacé par la variable « \$configCacheKey ». Dans notre

exemple, il n'y a pas de valeur renseignée dans le tableau de configuration concernant le nom de la clé du fichier de cache, mais nous pouvons la personnaliser en ajoutant la valeur comme ceci :

Définition d'une clé de cache

```
<?php
return array(
[...]
'module_listener_options' => array(
    'config_cache_enabled' => true,
    'config_cache_key' => 'ma-cle',
    'cache_dir' => 'data/cache',
    'module_paths' => array(
        './module',
        './vendor',
    ),
),
[...]
);
```

Notons que la classe `ListenerOptions` hérite de `Zend\Stdlib\AbstractOptions` qui fournit des méthodes de base afin de peupler un objet depuis un tableau d'options. La classe `AbstractOptions` transforme le nom des clés du tableau d'options en nom de méthode préfixé par « `set` » ce qui permet de peupler automatiquement l'objet qui étend cette classe. Pour plus de détails sur cette classe, reportez-vous au chapitre traitant la bibliothèque standard `Zend\Stdlib`.

Retournons à la fabrique du gestionnaire de modules où l'instruction suivante instancie un objet de type `ListenerAggregate`, expliqué dans un chapitre précédent :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
$serviceLocator)
{
[...]
$defaultListeners = new DefaultListenerAggregate($listenerOptions);
[...]
}
```

La classe `DefaultListenerAggregate` contient l'ensemble des écouteurs qui vont intervenir lors du chargement de nos différents modules, de la classe de chargement de module en passant par l'objet responsable des initialisations de modules. Les écouteurs de cette classe seront détaillés dans la section suivante.

L'instruction suivante initialise un conteneur d'écouteurs spécialisés dans la récupération de configuration liée au gestionnaire de services :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
$serviceLocator) {
[...]
$serviceListener = $serviceLocator->get('ServiceListener');
[...]
}
```

La fabrique de l'écouteur du gestionnaire de services initialise son objet avec sa liste de fabriques par défaut :

Zend/Mvc/Service/ServiceListenerFactory.php

```
class ServiceListenerFactory implements FactoryInterface
{
[...]
protected $defaultServiceConfig = array(
    'invokables' => array(
        'DispatchListener' => 'Zend\Mvc\DispatchListener',
        'Routelistener'     => 'Zend\Mvc\RouteListener',
    ),
    'factories' => array(
        'Application' => 'Zend\Mvc\Service\ApplicationFactory',
        'Config' => 'Zend\Mvc\Service\ConfigFactory'
        [...]
    ),
    [...]
);
[...]
}
```

La configuration par défaut représente la majorité des fabriques disponibles dans le framework. Une fois cette configuration ajoutée à l'écouteur chargé d'initialiser le gestionnaire de services, la fabrique du gestionnaire de modules lui ajoute des configurations sur les clés et méthodes permettant de surcharger la configuration du gestionnaire de services de l'application :

```
public function createService(ServiceLocatorInterface
$serviceLocator)
{
[...]
$serviceListener->addServiceManager(
    $serviceLocator,
    'service_manager',
    'Zend\ModuleManager\Feature\ServiceProviderInterface',
    'getServiceConfig'
);
$serviceListener->addServiceManager(
    'ControllerLoader',
    'controllers',
    'Zend\ModuleManager\Feature\ControllerProviderInterface',
    'getControllerConfig'
);
$serviceListener->addServiceManager(
    'ControllerPluginManager',
    'controller_plugins',
    'Zend\ModuleManager\Feature\ControllerPluginProviderInterface',
    'getControllerPluginConfig'
);
$serviceListener->addServiceManager(
    'ViewHelperManager',
    'view_helpers',
    'Zend\ModuleManager\Feature\ViewHelperProviderInterface',
    'getViewHelperConfig'
);
[...]
}
```

Nous verrons plus loin l'intérêt et le principe de cette configuration et comment la personnaliser. Nous verrons que ces quelques lignes nous permettront de comprendre comment configurer le gestionnaire de services rapidement depuis nos modules.

Les écouteurs liés à la configuration du gestionnaire de services et les écouteurs par défaut sont ensuite attachés au gestionnaire d'évènements :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    $events      = $serviceLocator->get('EventManager');
    $events->attach($defaultListeners);
    $events->attach($serviceListener);
    [...]
}
```

Enfin, évènement et gestionnaire sont initialisés :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    $moduleEvent  = new ModuleEvent;
    $moduleEvent->setParam('ServiceManager', $serviceLocator);
    $moduleManager = new ModuleManager($configuration['modules'],
        $events);
    $moduleManager->setEvent($moduleEvent);
    return $moduleManager;
}
```

Nous voici maintenant prêts pour le chargement des modules. La configuration liée aux modules est enregistrée et les écouteurs sont initialisés, il ne reste plus qu'à orchestrer le tout avec le gestionnaire de modules.

Chargement des modules

Lors de la création du gestionnaire de modules ModuleManager, la liste des modules à utiliser dans notre application lui a été passée en paramètre :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    $moduleManager = new ModuleManager($configuration['modules'],
        $events);
    [...]
}
```

Dans notre exemple, le fichier de configuration possède le module suivant :

application.config.php

```
||| <?php
||| return array(
|||     'modules' => array(
|||         'Application',
|||     ),
|||     [...]
||| );
```

Le gestionnaire de modules est prêt pour le chargement des modules, il ne nécessite pas plus d'initialisations ou de traitements.

Le chargement de modules s'effectue à l'aide de la méthode « `loadModules()` » du gestionnaire :

Zend/Mvc/Application.php

```
||| public static function init($configuration = array())
||| {
|||     [...]
|||     $serviceManager->get('ModuleManager')->loadModules();
|||     return $serviceManager->get('Application')->bootstrap();
||| }
```

Le gestionnaire d'évènements lance un premier évènement « `loadModules` », représenté par la constante « `ModuleEvent::EVENT_LOAD_MODULES` », afin de notifier le début du chargement des modules :

Zend/ModuleManager/ModuleManager.php

```
||| public function loadModules()
||| {
|||     [...]
|||     $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_MODULES,
|||     $this, $this->getEvent());
|||     [...]
||| }
```

Le premier écouteur notifié, attaché depuis le conteneur par défaut vu précédemment, est la classe `ModuleAutoloader` qui s'enregistre comme autoloader lors de l'initialisation des modules afin de pouvoir être utilisée lors du chargement de chacun d'entre eux :

Zend/ModuleManager/Listener/DefaultListenerAggregate.php

```
||| public function attach(EventManagerInterface $events)
||| {
|||     [...]
|||     $moduleAutoloader = new ModuleAutoloader($options->getModulePaths());
|||     $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES,
|||     array($moduleAutoloader, 'register'), 9000);
|||     [...]
||| }
```

C'est en effet cette classe qui va permettre le chargement des classes « Module.php » de chacun des modules :

Zend/Loader/ModuleAutoloader.php

```
public function register()
{
    spl_autoload_register(array($this, 'autoload'));
}
```

Sa méthode « `autoload()` » sera utilisée, si besoin, afin de localiser les classes de modules. Nous verrons le fonctionnement en détail de cette méthode lorsque nous renconterons l'étape de chargement de module.

L'écouteur qui est ensuite notifié est un objet de type `ConfigListener` depuis sa méthode « `onloadModulesPre()` », attaché depuis le conteneur d'écouteurs par défaut :

Zend/ModuleManager/Listener/DefaultListenerAggregate.php

```
public function attach(EventManagerInterface $events)
{
    [...]
    $configListener = $this->getConfigListener();
    [...]
    $this->listeners[] = $events->attach($configListener);
    [...]
}
```

Zend/ModuleManager/Listener/ConfigListener.php

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES,
array($this, 'onloadModulesPre'), 1000);
    if ($this->skipConfig) {
        return $this;
    }

    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULE,
array($this, 'onLoadModule'));
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES,
array($this, 'onLoadModulesPost'), -1000);

    return $this;
}
```

L'objet spécialisé dans la gestion de la configuration s'injecte au sein de l'évènement, ce qui permet aux autres écouteurs d'avoir accès à l'objet de configuration des modules :

Zend/ModuleManager/Listener/ConfigListener.php

```
public function onloadModulesPre(ModuleEvent $e)
{
    $e->setConfigListener($this);
    return $this;
}
```

Lorsque les traitements précédant le chargement des modules sont terminés, l'écouteur attaché par défaut au sein du gestionnaire de modules sur le même évènement est ensuite notifié, ce qui lui permet de lancer le chargement des modules :

Zend/ModuleManager/ModuleManager.php

```
protected function attachDefaultListeners()
{
    $events = $this->getEventManager();
    $events->attach(ModuleEvent::EVENT_LOAD_MODULES, array($this,
        'onLoadModules'));
}
```

Zend/ModuleManager/ModuleManager.php

```
public function onLoadModules()
{
    if (true === $this->modulesAreLoaded) {
        return $this;
    }
    foreach ($this->getModules() as $moduleName) {
        $this->loadModule($moduleName);
    }
    $this->modulesAreLoaded = true;
}
```

C'est la méthode « `loadModule()` » qui s'occupe du chargement, nous détaillerons cette méthode plus loin :

Zend/ModuleManager/ModuleManager.php

```
public function loadModule($moduleName)
{
    if (isset($this->loadedModules[$moduleName])) {
        return $this->loadedModules[$moduleName];
    }

    $event = ($this->loadFinished === false) ? clone $this->getEvent() :
    $this->getEvent();
    $event->setModuleName($moduleName);
    $this->loadFinished = false;
    $result = $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_
    MODULE_RESOLVE, $this, $event, function ($r) {
        return (is_object($r));
    });

    [...]
    $event->setModule($module);
    $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_MODULE,
    $this, $event);
    $this->loadedModules[$moduleName] = $module;

    $this->loadFinished = true;
    return $module;
}
```

Avant de continuer et de voir le chargement des modules et les évènements qui lui sont consacrés, notons que deux autres écouteurs sont attachés sur l'évènement « `loadModules` » avec des priorités très faibles, ils seront donc notifiés après le chargement des modules, et nous verrons en détail leur fonctionnement dans la

section concernant le postchargement :

- la méthode « onLoadModulesPost() » de l'objet ConfigListener est attachée avec une priorité de -1000 et chargée de la fusion des configurations avec la mise en cache ;
- la méthode « onLoadModulesPost() » de l'objet LocatorRegistrationListener qui permet d'enregistrer en tant que service le gestionnaire de modules et ses différents modules.

Revenons à la méthode « loadModule() » où la méthode contrôle, dans un premier temps, si le module n'a pas déjà été chargé. Cela, afin de ne pas faire ce traitement plusieurs fois, et lance immédiatement l'évènement « loadModule.resolve », représenté par la constante « ModuleEvent::EVENT_LOAD_MODULE_RESOLVE », afin de notifier les écouteurs capables de résoudre le chargement du module. Le conteneur d'écouteurs possède un objet spécialisé dans la résolution de module :

Zend/ModuleManager/Listener/DefaultListenerAggregate.php

```
public function attach(EventManagerInterface $events)
{
    [...]
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULE_
    RESOLVE, new ModuleResolverListener);
    [...]
}
```

L'objet Zend\ModuleManager\Listener\ModuleResolverListener est le composant responsable du chargement des classes de modules, il ne possède qu'une seule méthode :

Zend/ModuleManager/Listener/ModuleResolverListener.php

```
public function __invoke($e)
{
    $moduleName = $e->getModuleName();
    $class = $moduleName . '\Module';

    if (!class_exists($class)) {
        return false;
    }
    $module = new $class;
    return $module;
}
```

La méthode « __invoke() » a été introduite en PHP 5.3, elle est appelée lorsque l'on tente d'appeler un objet comme une fonction, ce qui est le cas ici en affectant l'instance de l'objet de type ModuleResolverListener comme fonction de callback. La fonction « __invoke » recherche alors une classe qui possède le nom « Module » associé à l'espace de noms du module, cela explique pourquoi nous devons nommer nos classes d'initialisation de module « Module ».

La fonction « class_exists() » va ensuite s'appuyer sur la classe de chargement ModuleAutoloader, spécialisée dans la localisation de classes de module, afin de tester

son existence. La méthode « autoload() » de l'autoloader ModuleAutoloader est alors appelée :

Zend/Loader/ModuleAutoloader.php

```
public function autoload($class)
{
    if (substr($class, -7) !== '\Module') {
        return false;
    }
    $moduleName = substr($class, 0, -7);
    if (isset($this->explicitPaths[$moduleName])) {
        $classLoaded = $this->loadModuleFromDir($this-
>explicitPaths[$moduleName], $class);
        [...]
    }

    $moduleClassPath    = str_replace('\\', DIRECTORY_SEPARATOR,
$moduleName);
    [...]

    foreach ($this->paths as $path) {
        $path = $path . $moduleClassPath;
        $classLoaded = $this->loadModuleFromDir($path, $class);
        if ($classLoaded) {
            return $classLoaded;
        }
        if ($pharSuffixPattern) {
            [...]
        }
    }
    return false;
}
```

L'autoloader recherche la classe « MonModule/Module.php » dans les chemins de l'attribut « \$explicitPaths », qui contient les chemins que l'on a associés à nos modules dans notre fichier de configuration. Pour rappel, le fichier de configuration contient les chemins suivants :

application.config.php

```
return array(
    [...]
    'module_paths' => array(
        './module',
        './vendor',
    ),
);
```

Cependant, comme nous n'avons pas associé le nom du module à chacun des chemins, le chargement ne va pas pouvoir se faire depuis un chemin statique, il va devoir être effectué depuis un répertoire. Analysons l'enregistrement des chemins de nos modules :

Zend/Loader/ModuleAutoloader.php

```

public function registerPath($path, $moduleName = false)
{
    if (!is_string($path)) {
        [...]
    }
    if ($moduleName) {
        $this->explicitPaths[$moduleName] =
            static::normalizePath($path);
    } else {
        $this->paths[] = static::normalizePath($path);
    }
    return $this;
}

```

Nous remarquons que la classe ne fait la liaison entre nom de module et chemin, que si nous indiquons en clé le nom du module. Nous pouvons donc modifier notre fichier de configuration afin de bénéficier d'un chargement plus rapide :

application.config.php

```

<?php
return array(
    [...]
    'module_paths' => array(
        'Application' => './module/Application',
    ),
    [...]
);

```

L'autoloader n'aura alors aucun traitement supplémentaire à effectuer pour localiser les différents modules. Dans le cas contraire, chacun des dossiers indiqués dans la configuration est parcouru afin de tenter le chargement du module depuis le nom de ce répertoire, suivi du nom de notre module :

Zend/Loader/ModuleAutoloader.php

```

public function autoload($class)
{
    [...]
    foreach ($this->paths as $path) {
        $path = $path . $moduleClassPath;
        $classLoaded = $this->loadModuleFromDir($path, $class);
    }
}

```

Ensuite, lorsque la localisation du répertoire correspondant au module est faite, soit par l'intermédiaire d'un chemin statique soit depuis le parcours de la liste de dossiers, l'autoloader fait appel à la méthode « `loadModuleFromDirectory()` » afin de procéder à son chargement :

Zend/Loader/ModuleAutoloader.php

```

protected function loadModuleFromDir($dirPath, $class)
{
    $file = new SplFileInfo($dirPath . '/Module.php');
}

```

```
    if ($file->isReadable() && $file->isFile()) {
        require_once $file->getRealPath();
        if (class_exists($class)) {
            $this->moduleClassMap[$class] = $file->getRealPath();
            return $class;
        }
    }
    return false;
}
```

La méthode vérifie que la classe « `MonModule\Module` » est bien présente dans le fichier nommé « `Module.php` » du dossier. Nous comprenons aussi pourquoi nous devons nommer la classe du module « `Module.php` ».

Une fois la localisation de la classe réussie, l'objet `ModuleResolverListener` peut alors instancier le module :

Zend/ModuleManager/Listener/ModuleResolverListener.php

```
public function __invoke($e)
{
[...]
$module = new $class;
return $module;
}
```

Nous venons d'expliquer le processus de chargement lié à l'évènement « `loadModule.resolve` » où l'écouteur `ModuleResolverListener` retourne une instance du module. Étant le seul écouteur de résolution de nom de module, le retour de l'écouteur satisfait alors la condition de court-circuitage qui s'attend au retour d'un objet :

Zend/ModuleManager/ModuleManager.php

```
public function loadModule($moduleName)
{
[...]
$result = $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_
MODULE_RESOLVE, $this, $event, function ($r) {
    return (is_object($r));
});
$module = $result->last();
[...]
}
```

Maintenant que la classe du module est effectivement instanciée, revenons à la méthode de chargement de module du gestionnaire de modules :

Zend/ModuleManager/ModuleManager.php

```
public function loadModule($moduleName)
{
[...]
$result = $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_
MODULE_RESOLVE,$this, $event, function ($r) {
    return (is_object($r));
});
}
```

```
    $module = $result->last();
    if (!is_object($module)) {
        [...]
    }
    $event->setModule($module);
    $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_MODULE,
    $this, $event);
    $this->loadedModules[$moduleName] = $module;
    $this->loadFinished = true;
    return $module;
}
```

L'évènement « `loadModule` », représenté par la constante « `ModuleEvent::EVENT_LOAD_MODULE` », correspondant au chargement du module, est lancé avant d'enregistrer le module au sein du tableau d'instances de modules chargées. À la suite de cet évènement, les écouteurs suivants vont être notifiés :

- la classe `Zend\ModuleManager\Listener\AutoloaderListener`, qui comme la classe `ModuleResolverListener` sert de callback, est notifiée depuis sa méthode « `__invoke` ». Cet écouteur permet l'initialisation de la configuration des autoloaders du module depuis la méthode « `getAutoloaderConfig()` » du fichier `Module.php` et sera le premier écouteur notifié ;
- un objet de type `Zend\ModuleManager\Listener\InitTrigger` qui lui aussi est passé comme fonction de callback. L'écouteur permet l'initialisation du module depuis la méthode « `init()` » et est attaché à l'évènement sans priorité ;
- un objet de type `Zend\ModuleManager\Listener\OnBootstrapListener` qui lui aussi est passé comme fonction de callback. L'écouteur permet la notification du module lors du processus de bootstrap si celui-ci implémente l'interface `BootstrapListenerInterface` ou la méthode « `onBootstrap()` » ;
- la classe `Zend\ModuleManager\Listener\ServiceListener` et sa méthode « `onLoadModule` » permettent l'initialisation et l'enrichissement de la configuration des gestionnaires de services de l'application. Elle est attachée sans priorité ;
- un objet de type `Zend\ModuleManager\Listener\LocatorRegistrationListener` permet le partage d'instances des modules et est attaché à l'évènement sans priorité ;
- l'objet de type `Zend\ModuleManager\Listener\ConfigListener` et sa méthode « `loadModule` » permettent la mise à jour de la configuration des modules depuis leur méthode « `getConfig()` ». Elle est attachée sans priorité.

Afin de comprendre toutes les étapes d'initialisation du module, analysons chacun des écouteurs.

Initialisation de l'autoloader du module

Le premier écouteur notifié permet d'ajouter à l'application, les autoloaders propres au chargement des classes liées au module courant. Afin d'enrichir la configuration du chargement de classe, l'écouteur de type `AutoloaderListener` est notifié depuis l'appel de la méthode « `getAutoloaderConfig()` » du module, si celui implémente

l'interface AutoloaderProvider qui définit cette seule méthode :

Zend/ModuleManager/Feature/AutoloaderProviderInterface.php

```
interface AutoloaderProviderInterface
{
    public function getAutoloaderConfig();
}
```

Zend/ModuleManager/Listener/AutoloaderListener.php

```
class AutoloaderListener extends AbstractListener
{
    public function __invoke(ModuleEvent $e)
    {
        $module = $e->getModule();
        if (!($module instanceof AutoloaderProviderInterface
              && !method_exists($module, 'getAutoloaderConfig')))
        {
            $autoloaderConfig = $module->getAutoloaderConfig();
            AutoloaderFactory::factory($autoloaderConfig);
        }
    }
}
```

Dans notre exemple, voici le contenu de cette méthode au sein de notre module Application :

Application/Module.php

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

La méthode « getAutoloaderConfig() » du module Application retourne un premier autoloader basé sur un fichier « autoload_classmap.php », ainsi qu'un autoloader basé sur un espace de noms propre au module. Pour des raisons de performances, il est bien sûr préférable d'utiliser au maximum le ClassMapAutoloader.

Initialisation du module

L'écouteur suivant est responsable de l'initialisation du module. L'objet de type InitTrigger, qui sert de fonction de callback sur l'évènement « loadModule », initialise le module en appelant la méthode « init() » du module si elle existe :

Zend/ModuleManager/Listener/InitTrigger.php

```

public function __invoke(ModuleEvent $e)
{
    $module = $e->getModule();
    if (!$module instanceof InitProviderInterface
        && !method_exists($module, 'init'))
    ) {
        return;
    }
    $module->init($e->getTarget());
}

```

Notification lors de l'initialisation de l'application

L'écouteur de type OnBootstrapListener permet de faciliter la vie du développeur en ajoutant un écouteur sur l'évènement de bootstrap afin de notifier les plugins de l'initialisation de l'application. La classe OnBootstrapListener attache un écouteur sur la méthode « `onBootstrap()` » du module si celui-ci implémente cette méthode ou l'interface BootstrapListenerInterface qui la définit :

Zend/ModuleManager/Feature/BootstrapListenerInterface.php

```

interface BootstrapListenerInterface
{
    public function onBootstrap(EventInterface $e);
}

```

La classe OnBootstrapListener utilise la possibilité du partage de gestionnaires d'évènements afin d'attacher l'écouteur :

Zend/ModuleManager/Listener/OnBootstrapListener.php

```

class OnBootstrapListener extends AbstractListener
{
    public function __invoke(ModuleEvent $e)
    {
        $module = $e->getModule();
        if (!$module instanceof BootstrapListenerInterface
            && !method_exists($module, 'onBootstrap'))
        ) {
            return;
        }
        $moduleManager = $e->getTarget();
        $events        = $moduleManager->getEventManager();
        $sharedEvents  = $events->getSharedManager();
        $sharedEvents->attach('Zend\Mvc\Application', 'bootstrap',
array($module, 'onBootstrap'));
    }
}

```

Notre module possède alors un écouteur fonctionnel si celui-ci implémente l'interface BootstrapListenerInterface.

Configuration du gestionnaire de services

L'écouteur suivant a pour responsabilité d'enrichir la configuration du gestionnaire de services. La méthode « `onLoadModule()` » de la classe `ServiceListener` enrichit la configuration du gestionnaire de services de l'application à l'aide des méthodes et clés de configuration que l'on a enregistrées précédemment :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    $serviceListener->addServiceManager(
        $serviceLocator,
        'service_manager',
        'Zend\ModuleManager\Feature\ServiceProviderInterface',
        'getServiceConfig'
    );
    $serviceListener->addServiceManager(
        'ControllerLoader',
        'controllers',
        'Zend\ModuleManager\Feature\ControllerProviderInterface',
        'getControllerConfig'
    );
    $serviceListener->addServiceManager(
        'ControllerPluginManager',
        'controller_plugins',
        'Zend\ModuleManager\Feature\ControllerPluginProviderInterface',
        'getControllerPluginConfig'
    );
    $serviceListener->addServiceManager(
        'ViewHelperManager',
        'view_helpers',
        'Zend\ModuleManager\Feature\ViewHelperProviderInterface',
        'getViewHelperConfig'
    );
    [...]
}
```

L'écouteur va parcourir l'ensemble des clés et vérifier la liste des méthodes enregistrées afin de pouvoir récupérer la configuration propre à chacune de ses entrées. Chaque entrée représente un type de configuration spécifique :

- la clé « `service_manager` » permet de définir ses propres fabriques dans la configuration. La méthode « `getServiceConfig()` » où l'interface `Zend\ModuleManager\Feature\ServiceProviderInterface` devra être implémentée si l'on souhaite définir les fabriques depuis son module ;
- la clé « `controllers` » permet de définir ses fabriques de contrôleurs dans la configuration. La méthode « `getControllerConfig()` » où l'interface `Zend\ModuleManager\Feature\ControllerProviderInterface` devra être implémentée si l'on souhaite définir les fabriques depuis son module ;
- la clé « `controller_plugins` » permet de définir ses propres fabriques d'aides d'actions dans la configuration. La méthode « `getControllerPluginConfig()` » où l'interface `Zend\ModuleManager\Feature\ControllerPluginProviderInterface` devra être implémentée si l'on souhaite définir les fabriques depuis son module ;

- la clé « view_helpers » permet de définir ses propres fabriques d'aides de vue dans la configuration. La méthode « getViewHelperConfig() » où l'interface Zend\ModuleManager\Feature\ViewHelperProviderInterface devra être implémentée si l'on souhaite définir les fabriques depuis son module.

Nous comprenons donc maintenant pourquoi ces clés sont utilisées dans les configurations pour définir nos propres fabriques et plugins. Voici l'implémentation de la méthode écouteur « onLoadModule() » qui se charge de récupérer les configurations correspondantes :

Zend/ModuleManager/Listener/ServiceListener.php

```
public function onLoadModule(ModuleEvent $e)
{
    $module = $e->getModule();
    foreach ($this->serviceManagers as $key => $sm) {
        if (!$module instanceof $sm['module_class_interface']
            && !method_exists($module, $sm['module_class_method']))
        ) {
            continue;
        }
        $config = $module->{$sm['module_class_method']}();
        [...]
        $fullname = $e->getModuleName() . ':' . $sm['module_class_
method'] . '()';
        $this->serviceManagers[$key]['configuration'][$fullname] =
        $config;
    }
}
```

Chaque entrée enregistrée est parcourue afin de tester l'existence de la méthode ou de l'implémentation de l'interface correspondante pour s'assurer de pouvoir récupérer la configuration correspondante. Celle-ci est ensuite enregistrée pour un traitement ultérieur où elle sera fusionnée.

Chaque module a la possibilité de surcharger la configuration du gestionnaire de services de l'application en passant des contrats avec les interfaces décrites plus haut. Cette possibilité laisse plus de liberté aux développeurs. Cependant, la possibilité d'extension ne s'arrête pas là, il est également possible de définir ses propres triplets de clés/interfaces/méthodes depuis la configuration de l'application. Re-prenons le code de la fabrique de la classe :

Zend/Mvc/Service/ServiceListenerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $configuration = $serviceLocator->get('ApplicationConfig');
    [...]
    if (isset($configuration['service_listener_options'])) {
        [...]
        foreach ($configuration['service_listener_options'] as $key =>
        $newServiceManager) {
            if (!isset($newServiceManager['service_manager'])) {
                [...]
            }
        }
    }
}
```

```
        if (!isset($newServiceManager['config_key'])) {
            [...]
        }
        if (!isset($newServiceManager['interface'])) {
            [...]
        }
        if (!isset($newServiceManager['method'])) {
            [...]
        }
        $serviceListener->addServiceManager(
            $newServiceManager['service_manager'],
            $newServiceManager['config_key'],
            $newServiceManager['interface'],
            $newServiceManager['method']
        );
    }
}
return $serviceListener;
}
```

La clé « `service_listener_options` » du tableau de configuration de l'application est parcourue et chaque tableau de cette clé est vérifié avant d'être enregistré dans l'écouteur. Une fois celui-ci enregistré, l'écouteur adoptera le même comportement pour nos clés personnalisées que pour celles utilisées par le framework.

Dans la configuration à fournir, la clé « `service_manager` » représente l'instance du gestionnaire de services qui devra automatiquement être configuré par les fabriques que l'on définit. Si nous analysons les valeurs par défaut du framework, nous comprenons qu'il est nécessaire de passer comme paramètre, soit une instance d'un gestionnaire, soit la clé sous laquelle il est enregistré par le gestionnaire de services :

Zend/Mvc/Service/ModuleManagerFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    $serviceListener->addServiceManager(
        $serviceLocator,
        [...]
    );
    $serviceListener->addServiceManager(
        'ControllerLoader',
        [...]
    );
    $serviceListener->addServiceManager(
        'ControllerPluginManager',
        [...]
    );
    $serviceListener->addServiceManager(
        'ViewHelperManager',
        [...]
    );
    [...]
}
```

Chaque clé représente un objet qui sera récupéré par le gestionnaire avant d'être

configuré. Cet objet doit étendre la classe ServiceManager. Voici les instructions effectuées par le framework lors des instructions de postchargement que nous verrons plus loin :

Zend/ModuleManager/Listener/ServiceListener.php

```
public function onLoadModulesPost(ModuleEvent $e)
{
    [...]
    foreach ($this->serviceManagers as $key => $sm) {
        [...]
        if (!$sm['service_manager'] instanceof ServiceManager) {
            $instance = $this->defaultServiceManager-
>get($sm['service_manager']);
            [...]
        }
        $serviceConfig = new ServiceConfig($smConfig);
        $serviceConfig->configureServiceManager($sm['service_manager']);
    }
}
```

Nous remarquons que la clé « service_manager » est utilisée pour récupérer l’instance de l’objet étendant de ServiceManager qui sera alors configuré.

Initialisation des instances partagées

L’écouteur suivant, de type LocatorRegistrationListener, qui implémente l’interface LocatorRegistered est chargé d’enregistrer les modules comme instance partagée :

Zend/ModuleManager/Listener/LocatorRegistrationListener.php

```
public function loadModule(ModuleEvent $e)
{
    if (!$e->getModule() instanceof LocatorRegistered) {
        return;
    }
    $this->modules[] = $e->getModule();
}
```

Ceci lui permettra par la suite d’ajouter au gestionnaire d’instances nos différents objets de modules afin de permettre un accès aux modules depuis l’application.

Mise à jour de la configuration des modules

Le dernier écouteur à intervenir est l’objet ConfigListener qui se charge de vérifier l’implémentation de la méthode « getConfig() » ou l’interface ConfigProviderInterface par le module, afin de récupérer la configuration propre au module et ainsi la fusionner avec les autres :

Zend/ModuleManager/Listener/ConfigListener.php

```

public function onLoadModule(ModuleEvent $e)
{
    $module = $e->getModule();
    if (!$module instanceof ConfigProviderInterface
        && !is_callable(array($module, 'getConfig')))
    ) {
        return $this;
    }
    $config = $module->getConfig();
    $this->addConfig($e->getModuleName(), $config);
    return $this;
}

```

Notons que cet écouteur n'est attaché que si le cache interne de configuration des modules n'est pas activé, car dans le cas contraire la configuration est récupérée depuis le cache existant :

Zend/ModuleManager/Listener/ConfigListener.php

```

public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES,
array($this, 'onLoadModulesPre'), 1000);

    if ($this->skipConfig) {
        return $this;
    }
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULE,
array($this, 'onLoadModule'));

    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES,
array($this, 'onLoadModulesPost'), -1000);
    return $this;
}

```

Comme vu précédemment, il est possible d'activer ce cache depuis la configuration de l'application. Il est assez intéressant d'activer le cache, car celui-ci se met à jour une fois les configurations fusionnées, ce qui lui évite ensuite de devoir le faire à nouveau lors d'une prochaine requête par le client. La méthode « `updateCache()` » a pour responsabilité de mettre le fichier de cache à jour :

Zend/ModuleManager/Listener/ConfigListener.php

```

protected function updateCache()
{
    if (($this->getOptions()->getConfigCacheEnabled()
        && (false === $this->skipConfig))
    ) {
        $configFile = $this->getOptions()->getConfigCacheFile();
        $this->writeArrayToFile($configFile, $this-
>getMergedConfig(false));
    }
    return $this;
}

```

La méthode utilisée pour la mise en cache sous forme de fichier est la méthode

« writeArrayToFile() » :

Zend/ModuleManager/Listener/AbstractListener.php

```
protected function writeArrayToFile($filePath, $array)
{
    $content = "<?php\nreturn " . var_export($array, 1) . ';';
    file_put_contents($filePath, $content);
    return $this;
}
```

La méthode utilisée pour la mise en cache sous forme de fichier est la méthode « writeArrayToFile() » :

Zend/ModuleManager/Listener/ConfigListener.php

```
protected function writeArrayToFile($filePath, $array)
{
    $content = "<?php\nreturn " . var_export($array, 1) . ';';
    file_put_contents($filePath, $content);
    return $this;
}
```

La méthode se contente d'écrire dans le fichier de cache une instruction PHP qui retourne un tableau associatif. Le fichier n'a ensuite plus qu'à être inclus dans le code PHP depuis l'instruction « include » afin d'avoir un accès aux données du tableau. Nous retrouvons ici le même fonctionnement que pour les fichiers de configuration que l'on utilise pour l'application ou celui des modules.

Traitement postchargement

Les modules de l'application sont maintenant chargés et il reste, comme vu plus haut, deux écouteurs à notifier sur l'évènement « loadModules ».

- L'écouteur de type ConfigListener qui met à jour la configuration avec l'ajout de fichiers de configuration supplémentaires. Ces chemins sont indiqués dans la configuration de l'application :

application.config.php

```
return array(
    [...]
    'module_listener_options' => array(
        'config_glob_paths'      => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        [...]
    );
)
```

Ces fichiers sont enregistrés dans le constructeur de l'objet :

Zend/ModuleManager/Listener/ConfigListener.php

```
public function __construct(ListenerOptions $options = null)
{
[...]
} else {
    $this->addConfigGlobPaths($this->getOptions()->getConfigGlobPaths());
    $this->addConfigStaticPaths($this->getOptions()->getConfigStaticPaths());
}
```

Chaque chemin, statique ou non, est alors enregistré dans la variable « \$paths ». La méthode utilisée par la classe ConfigListener afin d'ajouter des fichiers de configuration globale est la méthode « addConfigGlobPath() », basée sur la fonction « glob() » de PHP. Cette fonction recherche tous les chemins vérifiant le pattern donné en paramètre. Cependant, si des chemins statiques suffisent à notre application, il est possible d'utiliser les méthodes de chargement basées sur des chemins statiques :

Zend/ModuleManager/Listener/ConfigListener.php

```
public function addConfigStaticPaths($staticPaths);
public function addConfigStaticPath($staticPath);
```

Si un ou plusieurs fichiers sont à charger pour l'environnement en cours, ces méthodes seront plus adaptées afin de profiter pleinement des performances que l'on peut tirer du Zend Framework :

Utilisation de chemins statiques

```
return array(
[...]
'module_listener_options' => array(
    'config_static_paths' => array(
        'config/autoload/global.config.php',
    ),
[...]
);
```

Ces méthodes permettent de venir surcharger la configuration de l'application, initialisée depuis celles de nos modules. Comme nous venons de le voir, l'ajout de fichiers de configuration se fait après le chargement des modules, lors de la notification de la fin de chargement. Dans l'application, les fichiers de configuration globale et locale que l'on utilise se situent dans le répertoire « /config/autoload ».

Revenons au chargement des modules et à l'écouteur notifié de la fin du chargement des modules qui est la méthode « loadModules() » de l'objet ConfigListener :

Zend/ModuleManager/Listener/ConfigListener.php

```

public function onLoadModulesPost(ModuleEvent $e)
{
    foreach ($this->paths as $path) {
        $this->addConfigByPath($path['path'], $path['type']);
    }
    $this->mergedConfig = $this->getOptions()->getExtraConfig() ?: array();
    foreach ($this->configs as $key => $config) {
        $this->mergedConfig = ArrayUtils::merge($this->mergedConfig,
        $config);
    }

    if ($this->getOptions()->getConfigCacheEnabled()) {
        $this->updateCache();
    }
    return $this;
}

```

Les fichiers de configuration sont listés afin d'enregistrer leur configuration. Une fois toutes les configurations récupérées, le tableau de configuration finale est initialisé par le tableau de configuration « extra_config » que l'on peut définir dans la configuration de l'application :

application.config.php

```

return array(
    [...]
    'module_listener_options' => array(
        [...]
        'extra_config' => array(
            'view_manager' => array(
                'display_not_found_reason' => false,
                'display_exceptions'       => false,
            ),
            [...]
        );
    )
);

```

La configuration « extra_config » étant enregistrée en première, cela lui permet de définir une configuration par défaut qui pourra alors être surchargée par les modules si besoin. Chaque configuration des modules et configuration globale est ensuite fusionnée avec la configuration actuelle avant d'être cachée si le cache interne est activé. Il est important de comprendre l'ordre de la fusion des configurations si l'on ne souhaite pas voir des valeurs écrasées par d'autres fichiers de configuration. Voici le récapitulatif de l'ordre : configuration enregistrée sous la clé « extra_config », configuration des modules dans leur ordre de chargement et enfin les configurations enregistrées sous les clés « config_glob_paths » ou « config_static_paths ».

- Le deuxième écouteur est la classe LocatorRegistrationListener, qui attache deux écouteurs sur l'évènement « bootstrap » :

Zend/ModuleManager/Listener/LocatorRegistrationListener.php

```

public function loadModulesPost(Event $e)
{
    $moduleManager = $e->getTarget();
    $events        = $moduleManager->getEventManager()->getSharedManager();

    $events->attach('Zend\Mvc\Application', 'bootstrap', function ($e)
use ($moduleManager) {
    $moduleClassName = get_class($moduleManager);
    $application    = $e->getApplication();
    $services        = $application->getServiceManager();
    if (!$services->has($moduleClassName)) {
        $services->setService($moduleClassName, $moduleManager);
    }
}, 1000);
if (0 === count($this->modules)) {
    return;
}

$events->attach('Zend\Mvc\Application', 'bootstrap', array($this,
'onBootstrap'), 1000);
}

```

Zend/ModuleManager/Listener/LocatorRegistrationListener.php

```

public function onBootstrap(Event $e)
{
    $application = $e->getApplication();
    $services    = $application->getServiceManager();

    foreach ($this->modules as $module) {
        $moduleClassName = get_class($module);
        if (!$services->has($moduleClassName)) {
            $services->setService($moduleClassName, $module);
        }
    }
}

```

Les deux écouteurs de la méthode « bootstrap » permettent l'enregistrement du gestionnaire de modules ainsi que l'enregistrement des différents modules qui souhaitent être partagés, modules que l'objet avait précédemment enregistrés.

Une fois l'initialisation de nos modules réussie, la méthode de chargement informe la fin du traitement avec l'évènement « loadModules.post », représenté par la constante `ModuleEvent::EVENT_LOAD_MODULES_POST`, qui ne comporte qu'un seul écouteur avec l'objet `ServiceListener` :

Zend/ModuleManager/ModuleManager.php

```

public function loadModules()
{
    [...]
    $this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_MODULES_
POST, $this, $this->getEvent());
    return $this;
}

```

Zend/ModuleManager/Listener/ServiceListener.php

```
public function attach(EventManagerInterface $events)
{
    [...]
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_MODULES_POST, array($this, 'onLoadModulesPost'));
    return $this;
}
```

Zend/ModuleManager/Listener/ServiceListener.php

```
public function onLoadModulesPost(ModuleEvent $e)
{
    $configListener = $e->getConfigListener();
    $config         = $configListener->getMergedConfig(false);

    foreach ($this->serviceManagers as $key => $sm) {
        [...]
    }
}
```

La liste des clés et méthodes pour l'enrichissement du gestionnaire de services est parcourue et filtrée afin de mettre à jour la configuration. Nous ne détaillerons pas le contenu de la mise à jour qui est assez complexe à décrypter avec l'accès au tableau de tableaux. Retenons que c'est cet écouteur qui a la responsabilité de filtrer la configuration que nous ajoutons depuis les clés ou méthodes spéciales destinées à l'enrichissement du gestionnaire de services. Vous retrouverez en détail cette liste dans la section de ce chapitre consacrée à l'écouteur lié à la configuration du gestionnaire de services.

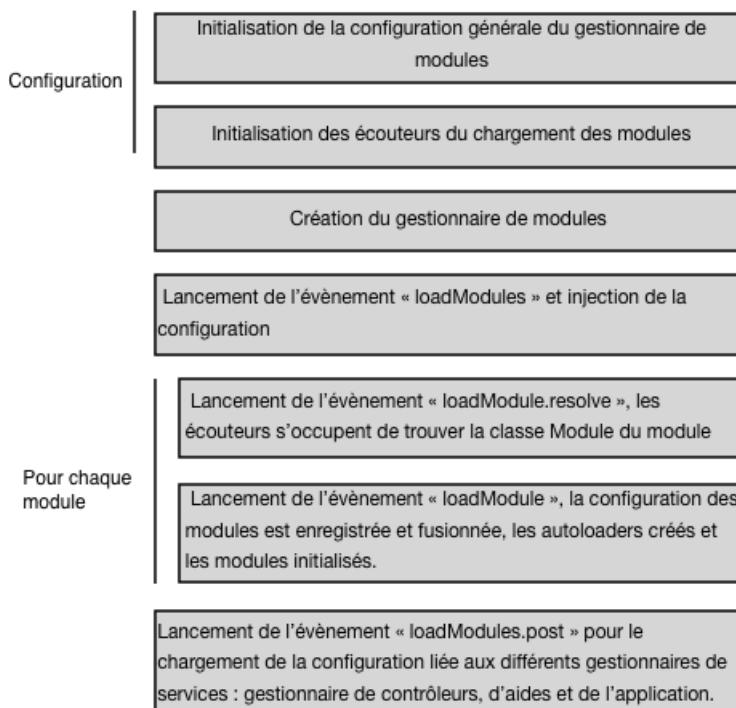
Nous avons parcouru le chemin effectué par le framework lors du chargement des modules. Nous avons pu voir que chacun des écouteurs est responsable d'une tâche précise, ce qui permet de découpler et de rendre plus flexible le chargement des modules.

Les configurations et le chargement des modules sont assez simples à prendre en main et la gestion du cache interne permet d'augmenter sensiblement les performances en permettant de ne pas exécuter à chaque fois la compilation des configurations.

Récapitulatif du processus

Processus de chargement

Voici un schéma récapitulatif des différentes étapes et événements du chargement de nos modules.



Squelette de notre classe Module

Voici un squelette de module type qui comprend l'ajout de fichier de configuration pour l'autoloader, les contrôleurs, le gestionnaire de services, les aides de vue et d'action, l'initialisation du module ainsi que le partage de l'instance du module au sein du gestionnaire d'instances :

Squelette de module

```

class Module
    implements InitProviderInterface, AutoloaderProvider,
    LocatorRegisteredInterface, ControllerPluginProviderInterface,
    ControllerProviderInterface, ViewHelperProviderInterface,
    ServiceProviderInterface, BootstrapListenerInterface,
    ConfigProviderInterface
{
    public function init(ModuleManagerInterface $manager)
    {
        // initialisation du module
    }
    public function onBootstrap(EventInterface $e)
    {
        // initialisation au bootstrap
    }
    public function getAutoloaderConfig()
    {
        return array(
            // définitions de nos autoloaders ici
    }
}

```

```
        );
    }
    public function getControllerConfig()
    {
        return array(
            // définitions des contrôleurs ici
        );
    }
    public function getControllerPluginConfig()
    {
        return array(
            // on définit nos aides d'actions ici
        );
    }
    public function getViewHelperConfig()
    {
        return array(
            // on définit nos aides de vue ici
        );
    }
    public function getServiceConfig()
    {
        return array(
            // on définit nos services ici
        );
    }
    public function getConfig()
    {
        // on inclut le fichier de configuration du module
    }
}
```

Dans cet exemple, toutes les interfaces sont implémentées afin de montrer toutes les possibilités, mais chacun implémentera les interfaces dont il a besoin.

Il est conseillé de faire implémenter la classe de module par les interfaces concernées même si, comme nous l'avons vu plus haut, il n'est pas obligatoire de le faire. L'interface permet de définir un contrat et d'indiquer explicitement ce que le module peut fournir comme données. Cela permet de s'assurer que les contrats sont respectés et que l'application reçoit les données qu'elle demande.

7

Les configurations

Le système de configuration du Zend Framework a bien évolué. Habitué aux fichiers INI ou XML lors de la première version, il est maintenant possible d'utiliser une configuration peut-être un peu moins lisible au premier abord, mais beaucoup plus performante, le fichier PHP.

AVEC LE ZEND FRAMEWORK 1

Lorsque l'on utilise un fichier de configuration INI ou XML pour la configuration générale de l'application, la classe `Zend_Application` se charge de le convertir en tableau pour le traiter. Au final, Zend Framework 2 propose une méthode pour supprimer l'étape de la conversion du fichier INI ou XML qui pouvait s'avérer lourde.

Afin de maîtriser les configurations de l'application, nous allons analyser comment la récupérer depuis un module ou contrôleur avant de voir les possibilités de séparation suivant l'environnement de travail du développeur, ainsi que les formats qui s'offrent à nous.

Accès à la configuration

Nous avons vu, dans le chapitre concernant les modules, que la configuration est créée à partir de la fusion de celles des modules, surchargée de fichiers de configurations plus globales.

Nous pourrions avoir besoin de récupérer l'objet de configuration depuis notre application, ce qui peut être fait facilement depuis nos classes de modules :

Récupération de la configuration

```
class Module implements BootstrapListenerInterface
{
    public function onBootstrap(EventInterface $e)
    {
        $config = $e->getApplication()->getConfig();
    }
}
```

Afin de récupérer la configuration depuis le module, nous faisons appel à la méthode « `getApplication()` » de l'évènement de type `MvcEvent` qui représente l'objet `Application`, instance qui dispose d'une méthode de récupération de configuration :

Zend/Mvc/Application.php

```
public function getConfig()
{
    return $this->serviceManager->get('Config');
}
```

La méthode « `getConfig()` » utilise le gestionnaire de services afin de récupérer la configuration de l'application, nous pourrions donc changer notre code par les instructions suivantes :

Récupération de la configuration avec le gestionnaire de services

```
class Module implements BootstrapListenerInterface
{
    public function onBootstrap(EventInterface $e)
    {
        $config = $e->getApplication()->getServiceManager()->get('Config');
    }
}
```

La configuration de l'application peut aussi être récupérée depuis un de nos contrôleurs suivant le même principe :

Récupération de la configuration depuis un contrôleur

```
class IndexController extends ActionController
{
    public function indexAction()
    {
        $config = $this->getServiceLocator()->get('Config');
    }
}
```

Nous savons maintenant récupérer la configuration depuis nos modules et contrôleurs. Cependant, lorsque notre module ajoute la sienne, il peut être nécessaire de la définir suivant l'environnement de travail du développeur. En effet, les accès aux bases de données, fichiers temporaires ou autres paramètres qui diffèrent en environnement de développement ou de production. Nous allons maintenant passer en revue les différentes manières de configurer son application en fonction de l'environnement de travail.

Configuration par environnement

Comme on l'a vu, il est préférable d'utiliser des fichiers de configurations de type PHP où les tableaux permettent un traitement immédiat. Cependant, comme nous avons besoin de configurations différentes suivant notre environnement (« déve-

lagement», «production», «préproduction», etc.), nous analyserons quatre manières de gérer facilement ses différentes configurations.

Afin de continuer cette section, nous partons du principe que l'environnement en cours est défini depuis notre fichier htaccess ou de vhosts avec la valeur «development» pour la variable « APPLICATION_ENV ».

Configuration par fichiers PHP

Rappelons, dans un premier temps, les principales étapes du chargement de la configuration. Chaque module est initialisé, et chaque configuration du module, donnée par la méthode « getConfig() » de la classe Module, est fusionnée à la configuration précédente avec la fonction PHP « array_replace_recursive() », qui ajoute récursivement toutes les entrées des tableaux au tableau existant. Si une clé est déjà présente pour un tableau ou sous-tableau existant, celle-ci est alors remplacée par la nouvelle valeur.

Ensuite, lorsque le Zend Framework a fusionné les configurations des modules, le résultat est surchargé par une configuration globale que l'on ajoute lors de l'initialisation du gestionnaire de modules à travers les méthodes « addConfigGlobPath() » ou « addConfigStaticPath() » du gestionnaire de configurations :

application.config.php

```
<?php
return array(
    [...]
    'module_listener_options' => array(
        'config_glob_paths'      => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        'config_static_paths'    => array(
            'config/autoload/config.override.php',
        ),
    [...]
);
```

Nous pouvons donc agir lors du chargement de la configuration des modules ou lors de la configuration globale. Pour cela, nous allons créer un fichier pour chaque environnement en indiquant les paramètres de chacun. Dans notre exemple, pour le chargement des modules nous créons deux fichiers dans le dossier de configuration du module :

- config/development.module.config.php ;
- config/production.module.config.php.

Il suffit ensuite de changer la méthode « getConfig() » de notre classe Module :

Application/Module.php

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

Nous pouvons ensuite faire de même pour le fichier de configuration globale :

- config/autoload/development.application.config.php ;
- config/autoload/production.application.config.php.

Et nous modifions alors la surcharge de configuration globale :

application.config.php

```
<?php
return array(
    [...]
    'module_listener_options' => array(
        'config_static_paths' => array(
            'config/autoload/' . APPLICATION_ENV .
            '.application.config.php',
        ),
        [...]
    );
);
```

Ceci nous permet de surcharger la configuration de l'application depuis un fichier spécifique lié à l'environnement en cours. C'est une première manière de surcharger les configurations en fonction de notre environnement. Cependant, il est parfois préférable de garder un héritage dans les configurations, comme ce que l'on avait dans la première version du framework, afin de garder une certaine cohérence et ne pas se retrouver avec des configurations dupliquées ou les effets de bord peuvent exister.

Configuration par héritage de fichier

Afin de conserver un système d'héritage de configuration, nous créons un fichier de base qui définira toutes les configurations dont on a besoin, et chaque fichier spécifique à un environnement peut définir uniquement ses propres valeurs. Dans notre exemple, lors du chargement de la configuration du module Application, nous avons deux fichiers dans notre dossier :

- config/module.config.php ;
- config/production.module.config.php.

Le fichier de configuration « module.config.php » définit toutes les configurations de base, et le fichier « production.module.config » redéfinit uniquement les valeurs propres à cet environnement. Prenons un exemple basique avec le gestionnaire de vues et l'affichage des erreurs :

config/module.config.php

```
||| <?php
||| return array(
||| [...]
|||     'view_manager' => array(
|||         'display_not_found_reason' => true,
|||         'display_exceptions'       => true,
|||         [...]
|||     );
||| );
```

config/production.module.config.php

```
||| <?php
||| return array(
||| [...]
|||     'view_manager' => array(
|||         'display_not_found_reason' => false,
|||         'display_exceptions'       => false,
|||         [...]
|||     );
||| );
```

Dans cet exemple, nous redéfinissons les clés « display_not_found_reason » et « display_exceptions » du gestionnaire de vues. Si l'exemple est trivial, dans la réalité de nombreuses clés seront redéfinies dans les différents environnements, comme l'accès à la base de données, la gestion des caches, etc.

Voici les instructions pour la gestion des différents environnements dans la classe de Module :

Application/Module.php

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

Nous utilisons ici la même méthode que lors de la fusion des configurations des modules, la fonction PHP « array_replace_recursive() » qui permet de remplacer récursivement dans les sous-tableaux les valeurs redéfinies dans les autres tableaux. Nous avons maintenant l'affichage des erreurs suivant l'environnement de travail en cours.

Nous savons maintenant gérer des configurations d'environnements différents tout en conservant un héritage entre les fichiers. Une autre solution s'offre à nous et à ceux qui préfèrent la gestion des configurations depuis un fichier INI, ce qui peut s'avérer parfois un peu plus lisible.

Configuration par fichier INI

Il est aussi possible de gérer ses configurations depuis des fichiers INI afin de rendre la configuration un peu plus lisible. Voici un exemple de ce que peut donner un fichier de configuration de ce format, l'exemple prend le fichier de configuration du module Application :

config/config.module.ini

```
[router]
routes.home.type = "Zend\Mvc\Router\Http\Literal"
routes.home.options.route = "/"
routes.home.options.defaults.controller = "index"
routes.home.options.defaults.action = "index"
[controller]
classes.index = "Application\Controller\IndexController"
[view_manager]
display_not_found_reason = true
display_exceptions = true
[...]
```

Ce fichier INI a été créé avec la classe Zend\Config\Writer\Ini :

Transformation d'une configuration en fichier INI

```
$config = new \Zend\Config\Writer\Ini();
$config->toFile(
    __DIR__ . '/config/module.config.ini',
    include __DIR__ . '/config/module.config.php'
);
```

Cela nous permet d'avoir un aperçu de la syntaxe à utiliser pour le fichier INI. Voici comment injecter notre fichier de configuration dans notre module :

Application/Module.php

```
public function getConfig()
{
    $config = new \Zend\Config\Reader\Ini();
    return $config->fromFile(__DIR__ . '/config/module.config.ini');
}
```

Il est aussi possible de fusionner les fichiers INI avec les fichiers de configuration en tableau PHP :

Application/Module.php

```
public function getConfig()
{
    $config = new \Zend\Config\Reader\Ini();
    return array_replace_recursive(
        $config->fromFile(__DIR__ . '/config/module.config.ini'),
        include __DIR__ . '/config/module.config.php'
    );
}
```

Il existe plusieurs manières de gérer ses configurations, chacun choisira celle qui lui conviendra le mieux même si pour des raisons de performances il est préférable de conserver les tableaux PHP. Notez qu'il est aussi possible d'ajouter des fichiers XML ou YAML si besoin, une classe de lecture et d'écriture existent pour chacun de ces types de fichier.

Configuration recommandée

Nous avons vu comment utiliser plusieurs configurations par environnement comme nous avions l'habitude lors de l'utilisation du Zend Framework 1. Cependant, cette manière de fonctionner n'est pas recommandée par l'équipe de développement du framework. En effet, avoir plusieurs fichiers de configuration versionnés peut poser des problèmes de sécurité avec les informations sensibles qu'ils peuvent contenir, comme les mots de passe de la base de données ou d'API.

L'équipe de développement préconise de s'appuyer sur des fichiers locaux qui seraient déployés sur les environnements respectifs sans être versionnés. Chacun ne contiendrait uniquement que ce qui lui permettrait de fonctionner sur cet environnement précis et se trouverait donc dans le dossier « config/autoload » de l'application qui permet d'être chargé automatiquement grâce aux instructions de la configuration :

application.config.php

```
<?php
return array(
    'module_listener_options' => array(
        'config_glob_paths'      => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        [...]
    ),
);
```

La convention établie par le framework veut que les fichiers considérés comme locaux soient ignorés du système de gestion de versions, ce qui permet alors de ne plus partager d'informations sensibles. Lorsque ceci aura été fait, vous n'aurez plus qu'à modifier votre processus de déploiement pour venir ajouter le fichier local à la configuration. Cette manière de fonctionner permet aussi de s'affranchir de la manipulation de variables d'environnement. Il est nécessaire de garder à l'esprit que chaque valeur concernant un environnement spécifique doit se trouver dans le dossier local qui lui est consacré. Cela permet également d'obtenir une meilleure séparation de la configuration.

8

Le router

Le router est le composant du Zend Framework qui nous permet de trouver, à partir de l'URL de la requête de notre client, le couple contrôleur/action qui sera utilisé comme action à effectuer. La configuration du router permet l'utilisation de nombreuses routes comme les routes statiques, avec expressions régulières, basées sur des noms de domaines, etc.

Le router et la brique MVC

Le composant responsable du routage a subi quelques améliorations depuis la première version du framework, et il reste toujours important de bien le maîtriser afin d'optimiser les performances de ses applications. En effet, beaucoup d'applications basées sur le Zend Framework ont tendance à systématiquement utiliser des routes qui font appel à un moteur de correspondance complexe au lieu de simples routes statiques, plus performantes, lorsque cela est possible.

Le router de l'application est initialisé dans le processus de bootstrap de l'application où il est enregistré au sein de l'évènement utilisé par la classe Application :

Zend/Mvc/Application.php

```
public function bootstrap()
{
    [...]
    $this->event = $event = new MvcEvent();
    $event->setTarget($this);
    $event->setApplication($this)
        ->setRequest($this->getRequest())
        ->setResponse($this->getResponse())
        ->setRouter($serviceManager->get('Router'));
    [...]
}
```

Le router est instancié depuis le gestionnaire de services qui utilise la fabrique RouterFactory :

Zend/Mvc/Service/RouterFactory.php

```

class RouterFactory implements FactoryInterface
{
    $config = $serviceLocator->get('Configuration');
    if(
        $rName === 'ConsoleRouter' ||
        ($cName === 'router' && Console::isConsole())
    ){
        [...]
    } else {
        $routerConfig = isset($config['router']) ? $config['router'] :
        array();
        $router = HttpRouter::factory($routerConfig);
    }
    return $router;
}

```

La fabrique vérifie tout d'abord si le client est en ligne de commande afin d'utiliser le router dédié à la console. Pour plus d'informations sur la gestion de la console et son intégration dans le framework, reportez-vous au chapitre consacré aux composants et la section qui lui est dédiée.

Nous remarquons que la clé utilisée dans la configuration pour la définition des routes est « `router` ». Si nous souhaitons définir nos routes, il sera nécessaire d'utiliser un tableau de la forme suivante :

module.config.php

```

<?php
return array(
    'router' => array(
        'routes' => array(
            'home' => array(
                'type' => 'Zend\Mvc\Router\Http\Literal',
                'options' => array(
                    'route' => '/',
                    'defaults' => array(
                        'controller' => 'index',
                        'action' => 'index',
                    )
                )
            ),
        ),
    ),
    [...]
);

```

La classe Application ne conserve pas l'instance du `router` comme attribut, celui-ci étant disponible auprès du gestionnaire de services, il est alors possible de récupérer cette instance depuis les instructions suivantes :

IndexController.php

```

class IndexController extends ActionController
{
    public function indexAction()
    {
        $router = $this->getServiceLocator()->get('Router');
    }
}

```

La configuration permet de définir le type de route que l'on souhaite utiliser pour chacune de nos routes. Notons qu'il est possible d'utiliser le nom complet de la route ou l'alias correspondant :

Routes avec nom court

```

<?php
return array(
    'router' => array(
        'routes' => array(
            'home' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => '/',
                    'defaults' => array(
                        'controller' => 'index',
                        'action' => 'index',
                    )
                ),
            ),
        ),
    ),
    [...]
);

```

Les noms courts sont disponibles grâce au gestionnaire de plugins utilisé par le router :

Zend/Mvc/Router/Http/TreeRouteStack.php

```

protected function init()
{
    $routes = $this->routePluginManager;
    foreach(array(
        'hostname' => __NAMESPACE__ . '\Hostname',
        'literal' => __NAMESPACE__ . '\Literal',
        'part' => __NAMESPACE__ . '\Part',
        'regex' => __NAMESPACE__ . '\Regex',
        'scheme' => __NAMESPACE__ . '\Scheme',
        'segment' => __NAMESPACE__ . '\Segment',
        'wildcard' => __NAMESPACE__ . '\Wildcard',
        'query' => __NAMESPACE__ . '\Query',
        'method' => __NAMESPACE__ . '\Method',
    ) as $name => $class
    ) {
        $routes->setInvokableClass($name, $class);
    };
}

```

Pour nous aider à comprendre l'intérêt des différents types de routes, analysons le comportement du router lors de l'évènement « route », lancé afin de connaître l'action à effectuer. Nous verrons que c'est la classe écouteur RouteListener qui s'occupe de gérer le routage de l'application :

Zend/Mvc/RouteListener.php

```
public function onRoute($e)
{
    $target      = $e->getTarget();
    $request     = $e->getRequest();
    $router      = $e->getRouter();
    $routeMatch = $router->match($request);

    if (!($routeMatch instanceof Router\RouteMatch)) {
        $e->setError($target::ERROR_ROUTER_NO_MATCH);
        $results = $target->getEventManager()->trigger(MvcEvent::EVENT_DISPATCH_ERROR, $e);
        if (count($results)) {
            $return = $results->last();
        } else {
            $return = $e->getParams();
        }
        return $return;
    }

    $e->setRouteMatch($routeMatch);
    return $routeMatch;
}
```

Le router, objet de type Zend\ Mvc\ Router\ Http\ TreeRouteStack récupère l'objet de requête qu'il passe ensuite à sa méthode « match() ». L'objet qui lui est retourné correspond au type Zend\ Mvc\ Router\ Http\ RouteMatch qui étend Zend\ Mvc\ Router\ RouteMatch contenant deux paramètres :

Zend/Mvc/Router/RouteMatch.php

```
class RouteMatch
{
    protected $params = array();

    protected $matchedRouteName;
    [...]
}
```

Ces paramètres permettent à la classe de conserver le nom de la route correspondant à la requête, ainsi que les paramètres de la route si celle-ci en utilise.

Revenons à la méthode qui vérifie la correspondance de l'objet TreeRouteStack, afin de mieux comprendre les étapes de la vérification des routes :

Zend/Mvc/Router/Http/TreeRouteStack.php

```
public function match(Request $request)
{
    [...]
    if ($baseUrlLength !== null) {
        $pathLength = strlen($uri->getPath()) - $baseUrlLength;
```

```

        foreach ($this->routes as $name => $route) {
            if (($match = $route->match($request, $baseUrlLength))
                instanceof RouteMatch && $match->getLength() === $pathLength) {
                $match->setMatchedRouteName($name);

                foreach ($this->defaultParams as $name => $value) {
                    if ($match->getParam($name) === null) {
                        $match->setParam($name, $value);
                    }
                }
                return $match;
            }
        }
    } else {
        return parent::match($request);
    }
    return null;
}

```

Toutes les routes sont parcourues, et le moteur de chacune d'elles est appelé afin de vérifier la correspondance. On peut d'ores et déjà noter l'importance d'avoir ses routes les plus statiques (donc le plus rapide à vérifier la correspondance), les routes les plus utilisées ou celles avec les besoins de performances les plus importants (routes statiques pour des actions marketings, routes les plus crawlées ou apportant le plus de trafic par exemple) en premier afin de pouvoir effectuer un minimum d'opérations de correspondance.

Le conteneur de routes utilisé est de type LIFO, la dernière route ajoutée sera la première vérifiée. Cependant, il est possible de casser ce fonctionnement avec l'ajout de priorité. Le conteneur sera d'abord trié sur les priorités et reprend son fonctionnement de type LIFO sur les routes de priorités équivalentes.

Analysons ce fonctionnement du router afin d'en comprendre la gestion. Le composant TreeRouteStack n'ayant pas de constructeur, celui de sa classe parent SimpleRouteStack sera donc celui qui sera appelé lors de l'instanciation :

Zend/Mvc/Router/SimpleRouteStack.php

```

public function __construct()
{
    $this->routes      = new PriorityList();
    $this->routeBroker = new RouteBroker();
    $this->init();
}

```

L'objet Zend\ Mvc\ Router\ PriorityList représente le conteneur de routes, il utilise deux attributs pour le classement des routes :

Zend/Mvc/Router/PriorityList.php

```

public function insert($name, Route $route, $priority)
{
    $this->sorted = false;
    $this->count++;
    $this->routes[$name] = array(
        'route'    => $route,
        'priority' => (int) $priority,
    );
}

```

```

    'serial' => $this->serial++,
);
}

```

Le conteneur trie les routes suivant leurs priorités, puis en fonction de l'ordre d'entrée dans la pile pour deux priorités équivalentes :

Zend/Mvc/Router/PriorityList.php

```

protected function compare(array $route1, array $route2)
{
    if ($route1['priority'] === $route2['priority']) {
        return ($route1['serial'] > $route2['serial']) ? -1 : 1;
    }
    return ($route1['priority'] > $route2['priority']) ? -1 : 1;
}

```

Il est donc possible d'ajouter une priorité haute sur notre page d'accueil pour la mettre en haut de la pile :

module.config.php

```

<?php
return array(
    'router' => array(
        'routes' => array(
            'home' => array(
                'type' => 'Zend\Mvc\Router\Http\Literal',
                'options' => array(
                    'route' => '/',
                    'defaults' => array(
                        'controller' => 'index',
                        'action' => 'index',
                    ),
                ),
                'priority' => 10,
            ),
            [...]
        ),
    );
);

```

Ceci permet à la route correspondant à la page d'accueil d'être la première de la liste à être analysée. Examinons ensuite les différents types de routes que l'on peut utiliser dans la configuration.

Les routes `Http\Literal`

Cette route est la plus simple à comprendre et à mettre en place. La vérification porte sur une comparaison brute de la route, de chaîne à chaîne. Ce type de route est donc très rapide à vérifier car elle demande simplement une vérification d'égalité :

Zend/Mvc/Router/Http/Literal.php

```
public function match(Request $request, $pathOffset = null)
{
[...]
if ($pathOffset !== null) {
    if ($pathOffset >= 0 && strlen($path) >= $pathOffset) {
        if (strpos($path, $this->route, $pathOffset) ===
$pathOffset) {
            return new RouteMatch($this->defaults,
strlen($this->route));
        }
    }
    return null;
}
if ($path === $this->route) {
    return new RouteMatch($this->defaults, strlen($this->route));
}
return null;
}
```

Nous remarquons que la méthode accepte un paramètre « `$pathOffset` » qui permet de faire la correspondance sur une partie de l'URL et non l'URL entière. Toutes les routes peuvent utiliser cette possibilité, le router `TreeRouteStack` utilise ce paramètre pour gérer ses arbres d'URL. Arbres qui seront détaillés dans la section de la route `Http\Part`. Le paramètre « `$pathOffset` » n'est utilisé que si l'objet de requête est un objet de requête HTTP. En effet, la possibilité de faire correspondre juste une partie de l'URL lors de la gestion d'arbre d'URL et de sous-URL empêche la comparaison brute d'égalité.

Une fois la correspondance sur l'égalité effectuée, le résultat est enveloppé dans un objet `RouteMatch`, qui permet la récupération des paramètres et nom de route correspondante, avant d'être retourné :

Zend/Mvc/Router/Http/Literal.php

```
public function match(Request $request, $pathOffset = null)
{
[...]
if (strpos($path, $this->route, $pathOffset) === $pathOffset) {
    return new RouteMatch($this->defaults, strlen($this->route));
}
[...]
}
```

AVEC LE ZEND FRAMEWORK 1

La route `Zend\Mvc\Router\Http\Literal` correspond aux routes `Zend_Controller_Router_Route_Static`.

Les routes Http\Regex

Les routes basées sur la vérification de chaînes avec expressions régulières utilisent le moteur d'expressions régulières de PHP avec la fonction « `preg_match()` » :

Zend/Mvc/Router/Http/Regex.php

```
public function match(Request $request, $pathOffset = null)
{
[...]
if ($pathOffset !== null) {
    $result = preg_match('(\G . ' . $this->regex . ')', $path,
$matches, null, $pathOffset);
} else {
    $result = preg_match('(^' . $this->regex . '$)', $path,
$matches);
}

if (!$result) {
    return null;
}

$matchedLength = strlen($matches[0]);
foreach ($matches as $key => $value) {
    if (is_numeric($key) || is_int($key)) {
        unset($matches[$key]);
    } else {
        $matches[$key] = urldecode($matches[$key]);
    }
}

return new RouteMatch(array_merge($this->defaults, $matches),
$matchedLength);
}
```

Comme nous le voyons, la méthode vérifie la chaîne de caractères avec la fonction « `preg_match()` » et récupère les paramètres qui correspondent au pattern afin de les enregistrer dans les paramètres de l'objet `RouteMatch`. L'utilisation d'expressions régulières offre de nombreuses possibilités pour la définition de ses routes dont voici un exemple :

module.config.php

```
<?php
return array(
    'router' =>
        'routes' => array(
            [...]
            'edit' => array(
                'type' => 'Regex',
                'options' => array(
                    'regex' => '/edit/(\d+)',
                    'spec' => '/edit/%user_id%',
                    'defaults' => array(
                        'action' => 'edit',
                    ),
                ),
            ),
        ),
    ),
);
```

```
    [...]  
    ),  
    );
```

Les routes Http\Scheme

La route Zend\Mvc\Router\Http\Scheme permet de faire la correspondance avec le nom du protocole de l'URL en cours :

Zend/Mvc/Router/Http/Scheme.php

```
public function match(Request $request)  
{  
    [...]  
    $scheme = $uri->getScheme();  
    if ($scheme !== $this->scheme) {  
        return null;  
    }  
    return new RouteMatch($this->defaults);  
}
```

Nous voyons que le protocole est récupéré depuis l'objet de requête afin d'être vérifié avec celui reçu en paramètre.

Les routes Http\Method

La route Zend\Mvc\Router\Http\Method permet de vérifier la correspondance avec la méthode utilisée dans la requête du client :

Zend/Mvc/Router/Http/Method.php

```
public function match(Request $request)  
{  
    if (!method_exists($request, 'getMethod')) {  
        return null;  
    }  
  
    $requestVerb = strtoupper($request->getMethod());  
    $matchVerbs = explode(',', strtoupper($this->verb));  
    $matchVerbs = array_map('trim', $matchVerbs);  
  
    if (in_array($requestVerb, $matchVerbs)) {  
        return new RouteMatch($this->defaults);  
    }  
  
    return null;  
}
```

La route se base sur un attribut « verb » qui contient la liste des méthodes acceptées séparées par une virgule. La méthode « explode() » de PHP transforme une chaîne de caractères en tableau depuis un délimiteur qui est ici une virgule. Il est possible de passer cette liste de méthodes en paramètres depuis la clé « verb » comme on le voit dans la fabrique de la classe :

Zend/Mvc/Router/Http/Method.php

```

public static function factory($options = array())
{
[...]
if (!isset($options['verb'])) {
    throw new Exception\InvalidArgumentException('Missing "verb" in
options array');
}
[...]
}

```

Il est donc très simple de créer une route depuis la fabrique :

Création de route Method

```

$route = Method::factory(array(
    'verb' => 'get, post, put',
    'defaults' => array(
        'controller' => 'mon-controller',
        'action' => 'mon-action',
    ),
));

```

Notons que la casse et les espaces dans la chaîne de caractères n'ont aucune importance car celle est passée en majuscules et les espaces supprimés depuis la méthode « `trim()` » de PHP :

Zend/Mvc/Router/Http/Method.php

```

public function match(Request $request)
{
[...]
$requestVerb = strtoupper($request->getMethod());
[...]
$matchVerbs = array_map('trim', $matchVerbs);
[...]
}

```

Les routes Http\Hostname

Les routes de type `Zend\Mvc\Router\Http\Hostname` permettent de vérifier la correspondance avec un nom de domaine. Cette route utilise le moteur d'expressions régulières pour chaque partie du domaine qui est séparée suivant le caractère du point, chaque partie est ensuite vérifiée avec « `preg_match()` » ce qui rend ce type de route moins performant que les autres, la vérification étant plus longue à faire :

Zend/Mvc/Router/Http/Hostname.php

```

public function match(Request $request)
{
[...]
$hostname = explode('.', $uri->getHost());
$params = array();

```

```

    if (count($hostname) !== count($this->route)) {
        return null;
    }
    foreach ($this->route as $index => $routePart) {
        if (preg_match('^(:(?<name>.+)$)', $routePart, $matches)) {
            if (isset($this->constraints[$matches['name']])) &&
                !preg_match('^(^' . $this->constraints[$matches['name']] . '$)', $hostname[$index])) {
                return null;
            }
            $params[$matches['name']] = $hostname[$index];
        } elseif ($hostname[$index] !== $routePart) {
            return null;
        }
    }
    return new RouteMatch(array_merge($this->defaults, $params));
}

```

Cette méthode peut être utile pour récupérer la correspondance de sous-domaine, car comme nous le voyons, la route utilise le moteur d'expressions régulières et conserve les correspondances trouvées.

Prenons un exemple avec la recherche de correspondance d'une URL avec un sous-domaine :

Vérification de sous-domaine

```

$route = \Zend\Mvc\Router\Http\Hostname::factory(array(
    'route' => ':subdomain.domaine.tld',
    'constraints' => array(
        'subdomain' => '\w{2,}-(\d)'
    ),
    'defaults' => array(
        'type' => 'montype',
    ),
));
$match = $route->match($this->getRequest());

```

L'objet « \$match » conserve le paramètre correspondant au sous-domaine, accessible depuis l'instruction :

Récupération du paramètre lié au sous-domaine

```

$match->getParam('subdomain');

```

Ce type de route peut aussi être utilisé pour filtrer l'accès à un sous-domaine de notre application Web.

Les routes Http\Segment

La route Zend\Mvc\Router\Http\Segment vérifie la correspondance de la même manière que la route Http\Regex. La seule différence est que la classe Segment construit l'expression régulière qu'il doit vérifier à partir des segments passés en paramètres :

Zend/Mvc/Router/Http/Segment.php

```

public function match(Request $request, $pathOffset = null)
{
[...]
if ($pathOffset !== null) {
    $result = preg_match('(\G . ' . $this->regex . ')', $path,
$matches, null, $pathOffset);
} else {
    $result = preg_match('(^' . $this->regex . '$)', $path,
$matches);
}

if (!$result) {
    return null;
}

$matchedLength = strlen($matches[0]);
foreach ($matches as $key => $value) {
    if (is_numeric($key) || is_int($key)) {
        unset($matches[$key]);
    } else {
        $matches[$key] = urldecode($matches[$key]);
    }
}

return new RouteMatch(array_merge($this->defaults, $matches),
$matchedLength);
}

```

La route de type Segment est simplement une manière plus lisible de décrire ses routes, il est donc préférable d'utiliser la route Http\Regex afin d'optimiser les performances.

Les routes Http\Wildcard

La route de type Wildcard s'utilise dans les routes en tant que routes filles en utilisant le paramètre « child_routes » lors de la configuration des routes. Ce type de route permet de récupérer les paramètres passés dans l'URL sous n'importe quelle forme que ce soit, y compris avec l'utilisation de routes statiques :

```

module.config.php
'router' => array(
    'routes' => array(
        'default' => array(
            'type'    => 'literal',
            'options' => array(
                'route'    => '/wildcard',
                'defaults' => array(
                    'controller' => 'index',
                    'action'      => 'index',
                ),
            ),
            'may_terminate' => true,
            'child_routes' => array(
                'wildcard' => array(

```

Il est alors possible d'utiliser l'URL « `monsite.com/wildcard/param1-value` » afin de récupérer les paramètres depuis le contrôleur courant par exemple :

IndexController.php

```
public function indexAction()
{
    $event = $this->getEvent();
    $routeMatch = $event->getRouteMatch();
    $value = $routeMatch->getParam('param1'); // 'value'

    return new ViewModel(array('key'=>$value));
}
```

La classe Wildcard gère la séparation de l'URL suivant les délimiteurs indiqués dans la configuration et enregistre chacun des paramètres suivant des couples clé/valeur :

Zend/Mvc/Router/Http/Wildcard.php

```
public function match(Request $request, $pathOffset = null)
{
[...]
$matches = array();
$params = explode($this->paramDelimiter, $path);
if (count($params) > 1 && ($params[0] !== '' || end($params) === '')) {
    return null;
}
if ($this->keyValueDelimiter === $this->paramDelimiter) {
    $count = count($params);
    for ($i = 1; $i < $count; $i += 2) {
        if (isset($params[$i + 1])) {
            $matches[urldecode($params[$i])] =
urldecode($params[$i + 1]);
        }
    }
} else {
    array_shift($params);
    foreach ($params as $param) {
        $param = explode($this->keyValueDelimiter, $param, 2);
        if (isset($param[1])) {
            $matches[urldecode($param[0])] =
urldecode($param[1]);
        }
    }
}
```

```

    return new RouteMatch(array_merge($this->defaults, $matches),
        strlen($path));
}

```

Chaque paramètre est donc ensuite accessible depuis l'objet RouteMatch retourné par le router, car nous le voyons, chaque paramètre est traité afin d'être stocké dans l'objet RouteMatch.

Les routes Http\Part

La route Zend\Mvc\Router\Http\Part permet de créer un arbre ou une arborescence de routes possibles à partir d'une URL. Prenons un exemple en modifiant le router de l'application pour utiliser un objet de type Http\Part. Pour l'exemple, le code est placé dans un écouteur de l'évènement de routage « route » au sein du module Application :

Création d'une arborescence de routes

```

$part = \Zend\Mvc\Router\Http\Part::factory(array(
    'route' => array(
        'type' => 'literal',
        'options' => array(
            'route' => '/',
            'defaults' => array(
                'controller' => 'index',
                'action' => 'index',
            ),
        ),
    ),
    'may_terminate' => true,
    'route_broker' => $e->getRouter()->routeBroker(),
    'child_routes' => array(
        'blog' => array(
            'type' => 'literal',
            'options' => array(
                'route' => 'blog',
                'defaults' => array(
                    'controller' => 'blog',
                    'action' => 'index',
                ),
            ),
        ),
        'may_terminate' => true,
        'child_routes' => array(
            'actu' => array(
                'type' => 'Zend\Mvc\Router\Http\Literal',
                'options' => array(
                    'route' => '/actu',
                    'defaults' => array(
                        'action' => 'actu',
                    ),
                ),
            ),
            'may_terminate' => true,
        ),
    ),
),
));
$e->setRouter($part);

```

On voit que l'on a créé ici une arborescence telle que l'URL :

- « / » nous envoie sur le contrôleur IndexController et l'action « index » ;
- « /blog » nous envoie sur le contrôleur BlogController et l'action « index » ;
- « /blog/actu » nous envoie sur le contrôleur BlogController et l'action « actu ».

Il est possible de gérer des arbres d'URL complexes. Cependant, si cette utilisation offre de nombreuses possibilités, ce n'est certainement pas la plus performante.

L'écouteur pour les modules

La classe ModuleRouteListener est un écouteur spécialisé dans la construction de routes par défaut pour un module donné. En effet, regardons les routes du module Application de notre squelette pour comprendre le cheminement :

```
module.config.php
  return array(
    'router' => array(
      [...]
      'application' => array(
        'type'      => 'Literal',
        'options'   => array(
          'route'    => '/application',
          'defaults' => array(
            '__NAMESPACE__' => 'Application\
Controller',
            'controller'    => 'Index',
            'action'        => 'index',
          ),
        ),
        'may_terminate' => true,
        'child_routes' => array(
          'default' => array(
            'type'      => 'Segment',
            'options'   => array(
              'route'    => '/',
              ':controller[/:action]]',
              'constraints' => array(
                'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                'action'      => '[a-zA-Z][a-zA-Z0-9_-]*',
              ),
              'defaults'   => array(),
            ),
          ),
        ),
      ),
    ),
  );

```

La première route définie, « /application », est celle correspondant au point d'entrée du module Application. Cette route, qui possède des routes filles, laisse la pos-

sibilité d'utiliser le couple contrôleur/action à la suite du nom du module. Ce système permet de déterminer la route qui correspond au module en laissant ensuite les noms des contrôleurs et actions par défaut.

Analysons le code de la classe `ModuleRouteListener` afin de comprendre son fonctionnement :

Zend/Mvc/ModuleRouteListener.php

```
public function onRoute(MvcEvent $e)
{
    $matches = $e->getRouteMatch();
    if (!$matches instanceof Router\RouteMatch) {
        return;
    }

    $module = $matches->getParam(self::MODULE_NAMESPACE, false);
    if (!$module) {
        return;
    }

    $controller = $matches->getParam('controller', false);
    if (!$controller) {
        return;
    }

    if (0 === strpos($controller, $module)) {
        return;
    }

    $matches->setParam(self::ORIGINAL_CONTROLLER, $controller);
    $controller = $module . '\\' . str_replace(' ', '', ucwords(str_replace('-', ' ', $controller)));
    $matches->setParam('controller', $controller);
}
```

L'écouteur vérifie qu'une route a été trouvée avec la validité de la propriété `RouteMatch` avant de vérifier s'il existe une clé correspondant à la constante `ModuleRouteListener::MODULE_NAMESPACE` qui a pour valeur « `__NAMESPACE__` ». Cette clé permet de spécifier le namespace du module afin de construire automatiquement le nom du contrôleur. Nous comprenons donc son intérêt dans la configuration. Une fois cette opération effectuée, l'écouteur vérifie la présence du contrôleur et la validité du nom du contrôleur. Le nom du contrôleur inscrit dans la configuration est sauvegardé, et le nom du contrôleur est construit à partir du module :

Zend/Mvc/ModuleRouteListener.php

```
public function onRoute(MvcEvent $e)
{
    [...]
    $controller = $module . '\\' . str_replace(' ', '', ucwords(str_replace('-', ' ', $controller)));
    $matches->setParam('controller', $controller);
}
```

Cette approche n'est peut-être pas forcément intuitive pour les développeurs. Il

est cependant assez facile de modifier ce comportement afin d'obtenir une route basée sur le triplet module/contrôleur/action.

La première chose à faire est d'ajouter la route qui nous permettra d'avoir ce comportement par défaut :

module.config.php

```

return array(
    'router' => array(
        'routes' => array(
            [...]
            'module-controller-action' => array(
                'type'      => 'Segment',
                'options'   => array(
                    'route'    => '/[:module[:controller[:action]]]',
                    'constraints' => array(
                        'module' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'action'    => '[a-zA-Z][a-zA-Z0-9_-]*',
                    ),
                ),
                [...]
            )
        );
    );
);

```

Une fois la route par défaut définie, il suffit de reprendre la même dynamique que l'écouteur précédent en supprimant la vérification du nom du contrôleur et en ajoutant le formatage nécessaire.

ZendX/Mvc/DefaultModuleRouteListener.php

```

public function onRoute(MvcEvent $e)
{
    $matches = $e->getRouteMatch();
    if (!$matches instanceof Router\RouteMatch) {
        return;
    }

    $module = $matches->getParam(self::MODULE_NAMESPACE, false);
    if (!$module) {
        return;
    }

    $controller = $matches->getParam('controller', false);
    if (!$controller) {
        return;
    }

    $controller = ucfirst($module) . '\\' . 'Controller\\' . str_
        replace(' ', '', ucwords(str_replace('-', ' ', $controller))) .
        'Controller';
    $matches->setParam('controller', $controller);
}

```

Une fois ces opérations effectuées, il suffit d'ajouter l'écouteur dans notre module :

Module.php

```
public function onBootstrap($e)
{
    $eventManager = $e->getApplication()->getEventManager();

    $defaultModuleRouteListener = new DefaultModuleRouteListener();
    $defaultModuleRouteListener->attach($eventManager);

    $moduleRouteListener = new ModuleRouteListener();
    $moduleRouteListener->attach($eventManager);
}
```

Comme nous le voyons, il est très simple de surcharger le comportement par défaut du router afin de créer le fonctionnement que l'on attend.

9

Les contrôleurs

Le contrôleur est l'élément qui permet de mettre en œuvre les traitements liés à la demande du client. Il utilise l'environnement de l'application (modèle, vue, configuration, évènements, etc.) afin de pouvoir injecter ses données dans la vue qui sera ensuite retournée au client.

Afin de simplifier la vie des développeurs et de leur permettre d'avoir des contrôleurs d'actions prêts à l'emploi, le framework implémente plusieurs contrôleurs abstraits qui fournissent de nombreuses méthodes permettant de gérer ces principales problématiques, ainsi qu'un fonctionnement par défaut.

Le contrôleur **AbstractActionController**

Le contrôleur d'actions abstrait fourni par le Zend Framework est un objet de type Zend\ Mvc\ Controller\ AbstractActionController dont voici le squelette :

Zend/ Mvc/ Controller/ AbstractActionController.php

```
abstract class AbstractActionController extends AbstractController
{
    public function indexAction() {}

    public function notFoundAction() {}

    public function onDispatch(MvcEvent $e);
}
```

Ce contrôleur permet de fournir un comportement et des fonctionnalités par défaut afin de fournir un contrôleur d'actions prêt à l'emploi aux développeurs. Le contrôleur d'actions abstrait hérite du contrôleur de base AbstractController qui permet la gestion des objets nécessaires à son bon fonctionnement :

Zend/ Mvc/ Controller/ AbstractController.php

```
abstract class AbstractController implements
Dispatchable,
EventManagerAwareInterface,
InjectApplicationEventInterface,
ServiceLocatorAwareInterface
{
```

```

abstract public function onDispatch(MvcEvent $e);

public function dispatch(Request $request, Response $response =
null);
public function getRequest();
public function getResponse();
public function setEventManager(EventManagerInterface $events);
public function getEventManager();
public function setEvent(Event $e);
public function getEvent();
public function setServiceLocator(ServiceLocatorInterface
$serviceManager);
public function getServiceLocator();
public function getPluginManager();
public function setPluginManager(PluginManager $plugins);
public function plugin($name, array $options = null);
protected function attachDefaultListeners();
public static function getMethodFromAction($action);
}

```

Nous comprenons que le contrôleur de base fournit un ensemble de méthodes qui permet la gestion des objets nécessaires au fonctionnement minimal d'un contrôleur avec par exemple la gestion du gestionnaire d'évènements, gestionnaire de services, gestionnaire de plugins, etc. Le contrôleur d'actions abstrait permet, lui, de définir un comportement par défaut pour les contrôleurs d'actions classiques. Nous ne reviendrons pas sur ce contrôleur abstrait de base qui ne possède aucune méthode spécifique, si ce n'est la méthode « `setEventManager()` » qui définit les identifiants de notre contrôleur :

Zend/Mvc/Controller/AbstractController.php

```

public function setEventManager(EventManagerInterface $events)
{
    $events->setIdentifiers(array(
        'Zend\Stdlib\DispatchableInterface',
        __CLASS__,
        get_called_class(),
        $this->eventIdentifier,
        substr(get_called_class(), 0, strpos(get_called_class(), '\\'))
    ));

    $this->events = $events;
    $this->attachDefaultListeners();

    return $this;
}

```

La liste de ces identifiants va nous permettre de comprendre comment il est possible d'agir sur les contrôleurs ou ensembles de contrôleurs d'un module. En effet, chaque contrôleur possède comme identifiant :

- « `Zend\Stdlib\DispatchableInterface` » qui est utilisé lors de la construction des vues sur tout contrôleur qui peut être distribué ;
- « `__CLASS__` » qui représente le nom complet de la classe du contrôleur abstrait ;
- « `get_called_class()` » qui représente le nom de la classe de l'objet instancié ;

- « `$this->eventIdentifier` » qui représente un identifiant que l'on peut personnaliser en le surchargeant au sein de son contrôleur ;
- « `substr(get_called_class(), 0, strpos(get_called_class(), '\\\\'))` » qui représente la première partie de l'espace de noms du contrôleur, soit l'équivalent de son module correspondant.

Tous ces identifiants nous permettent d'agir aussi bien sur le contrôleur lui-même que sur son module correspondant ou encore sur un périmètre que l'on aurait défini depuis la surcharge de l'attribut « `$eventIdentifier` ».

Revenons à notre contrôleur d'actions et à son fonctionnement. Nous verrons dans le chapitre consacré au cœur du framework que le point d'entrée du contrôleur d'actions est la méthode « `dispatch()` », appelée par l'objet `DispatchListener` responsable de la distribution :

Zend/Mvc/DispatchListener.php

```
public function onDispatch(MvcEvent $e)
{
    [...]
    try {
        $return = $controller->dispatch($request, $response);
    } catch (\Exception $ex) {
        [...]
    }
}
```

Voici le descriptif de la méthode « `dispatch()` », point d'entrée de la classe `AbstractController` :

Zend/Mvc/Controller/AbstractController.php

```
public function dispatch(Request $request, Response $response = null)
{
    $this->request = $request;
    if (!$response) {
        $response = new HttpResponse();
    }

    $this->response = $response;
    $e = $this->getEvent();
    $e->setRequest($request)
        ->setResponse($response)
        ->setTarget($this);

    $result = $this->getEventManager()->trigger(MvcEvent::EVENT_DISPATCH,
        $e, function($test) {
            return ($test instanceof Response);
        });
    if ($result->stopped()) {
        return $result->last();
    }
    return $e->getResult();
}
```

Les premières instructions initialisent les objets de requête et de réponse du contrôleur qui peuvent ensuite être partagés au sein des évènements du contrôleur.

AVEC LE ZEND FRAMEWORK 1

Les attributs de requête et de réponse sont nommés « `$_request` » et « `$_response` ». Si les noms ont changé et l'underscore retiré, les méthodes « `getRequest()` » et `getResponse()` sont toujours disponibles pour l'accès aux objets respectifs. La méthode « `dispatch()` » est gérée par la classe `Zend_Controller_Action`. Le manque d'évènement oblige le framework à créer des hooks afin de pouvoir agir sur le contrôleur, ainsi que sur l'objet de requête ou de réponse. Ces deux objets sont maintenant partagés au sein des évènements du contrôleur, tout écouteur peut donc intervenir dessus.

Le framework injecte ensuite les deux objets de requête et de réponse dans l'objet d'évènement avant de lancer l'évènement « `dispatch` » qui permet la distribution de l'action à effectuer :

`Zend/Mvc/Controller/AbstractController.php`

```
public function dispatch($action)
{
    [...]
    $result = $this->getEventManager()->trigger(MvcEvent::EVENT_DISPATCH,
    $e,
    function($test) {
        return ($test instanceof Response);
    });
    [...]
}
```

Le contrôleur attache un écouteur sur l'évènement « `dispatch` » avant de lancer cet évènement dont il est aussi un écouteur depuis la méthode « `onDispatch()` » lors de l'ajout des écouteurs par défaut :

`Zend/Mvc/Controller/AbstractController.php`

```
protected function attachDefaultListeners()
{
    $events = $this->getEventManager();
    $events->attach(MvcEvent::EVENT_DISPATCH, array($this,
    'onDispatch'));
}
```

Nous pouvons nous demander l'intérêt d'une telle procédure et le fait que la méthode « `onDispatch()` » n'est pas appelée directement par le contrôleur. Cette manière de fonctionner permet à d'autres écouteurs de venir s'attacher avec une priorité différente et d'être aussi notifiés de l'évènement. Nous pouvons créer ici un système identique à ce que l'on connaît avec les hooks « `preDispatch` » et « `postDispatch` » du Zend Framework 1 en attachant des écouteurs avant et après la notification du contrôleur lui-même. Cependant, attention au fait que le court-circuitage peut exister si un élément de type `Zend\Stdlib\ResponseInterface` est renvoyé depuis les écouteurs de priorités les plus fortes :

Zend/Mvc/Controller/AbstractController.php

```
public function dispatch($action)
{
[...]
$result = $this->getEventManager()->trigger(MvcEvent::EVENT_DISPATCH,
$e,
function($test) {
    return ($test instanceof Response);
});
[...]
}
```

Le premier écouteur à renvoyer un objet de type ResponseInterface arrête la propagation de l'évènement. Si c'est le cas, la méthode « `dispatch()` » va logiquement retourner le dernier objet de réponse des écouteurs, sinon elle renverra le résultat de l'évènement depuis la méthode `« getResult()` :

Zend/Mvc/Controller/AbstractController.php

```
public function dispatch(Request $request, Response $response = null)
{
[...]
$result = $this->getEventManager()->trigger(MvcEvent::EVENT_DISPATCH,
$e,
function($test) {
    return ($test instanceof Response);
});

if ($result->stopped()) {
    return $result->last();
}
return $e->getResult();
}
```

Notons que la méthode ne retourne pas systématiquement le dernier objet de réponse reçu. Cela s'explique par le fait que rien ne lui indique que le dernier objet retourné sera bien un objet de type ResponseInterface. Il est donc important de bien peupler l'évènement de notre résultat.

Analysons maintenant l'écouteur par défaut de `AbstractActionController` :

Zend/Mvc/Controller/AbstractActionController.php

```
public function onDispatch(MvcEvent $e)
{
$routeMatch = $e->getRouteMatch();
if (!$routeMatch) {
    [...]
}

$action = $routeMatch->getParam('action', 'not-found');
$method = static::getMethodFromAction($action);
if (!method_exists($this, $method)) {
    $method = 'notFoundAction';
}

$actionResponse = $this->$method();
```

```

    $e->setResult($actionResponse);
    return $actionResponse;
}
```

La méthode récupère la route vérifiée par le router depuis l'évènement en cours et la méthode « `getRouteMatch()` », qui stocke l'objet de type `RouteMatch`. Cet objet permet de récupérer les informations liées à la route en cours (nom, paramètres de la route, contrôleur et action à utiliser).

Si aucune route n'a été trouvée lors de la correspondance entre notre configuration et la demande du client, le contrôleur utilise la route d'erreur par défaut « `not-found` ». Il formate ensuite le nom de l'action à distribuer depuis les paramètres de la route courante :

Zend/Mvc/Controller/AbstractActionController.php

```

public function onDispatch(MvcEvent $e)
{
[...]
$action = $routeMatch->getParam('action', 'not-found');
$method = static::getMethodFromAction($action);
if (!method_exists($this, $method)) {
    $method = 'NotFoundAction';
}
$actionResponse = $this->$method();
[...]
}
```

Zend/Mvc/Controller/AbstractActionController.php

```

public static function getMethodFromAction($action)
{
$action = str_replace(array('.', '-', '_'), ' ', $action);
$action = ucwords($action);
$action = str_replace(' ', '', $action);
$action = lcfirst($action);
$action .= 'Action';
return $action;
}
```

La méthode « `getMethodFromAction()` » nous permet de comprendre pourquoi les actions des contrôleurs doivent avoir comme suffixe le mot « `Action` ». Le contrôleur peut maintenant appeler la méthode d'action :

Zend/Mvc/Controller/AbstractActionController.php

```

public function onDispatch(MvcEvent $e)
{
[...]
$actionResponse = $this->$method();
[...]
}
```

Le résultat de cette méthode sera stocké dans le résultat de l'évènement courant et la réponse de l'action est tout de même retournée :

Zend/Mvc/Controller/AbstractActionController.php

```
public function onDispatch(MvcEvent $e)
{
    [...]
    $actionResponse = $this->$method();
    $e->setResult($actionResponse);
    return $actionResponse;
}
```

Ces instructions rejoignent le point évoqué tout à l'heure avec l'importance de peupler l'attribut de résultat de l'évènement, afin de s'assurer de la bonne transmission du résultat à la méthode « `dispatch()` » qui retourne ensuite ce résultat :

Zend/Mvc/Controller/AbstractController.php

```
public function dispatch(Request $request, Response $response = null)
{
    [...]
    return $e->getResult();
}
```

Cependant, nous pouvons nous demander pourquoi la méthode retourne tout de même la réponse depuis la méthode « `onDispatch()` ». Comme nous l'avons vu lors de la gestion du court-circuitage de notre évènement « `dispatch` », dès lors qu'un écouteur retourne un objet de type `ResponseInterface`, la propagation s'arrête. Et nous ne connaissons pas ici l'objet qui est retourné par la méthode de contrôleur, cela pourrait aussi être un objet de ce type.

Comme nous le verrons dans le chapitre consacré au cœur du framework, la classe `Application` cherche à construire un objet de réponse afin de pouvoir envoyer son contenu au client. Nous comprenons donc l'intérêt de court-circuiter le traitement si l'objet de réponse est déjà construit.

Méthodes d'action par défaut

Dans le contrôleur d'actions abstrait `AbstractActionController`, deux méthodes d'action existent par défaut :

Zend/Mvc/Controller/AbstractActionController.php

```
public function indexAction()
{
    return new ViewModel(array(
        'content' => 'Placeholder page'
));
}
```

Zend/Mvc/Controller/AbstractActionController.php

```
public function notFoundAction()
{
    $response = $this->response;
    $event = $this->getEvent();
    $routeMatch = $event->getRouteMatch();
```

```

    $response->setStatusCode(404);
    $routeMatch->setParam('action', 'not-found');
    return new ViewModel(array(
        'content' => 'Page not found'
    )));
}

```

La méthode « `indexAction()` » permet d'obtenir un rendu de la vue sur l'action « `index` » sans pour autant définir une méthode de ce nom dans notre contrôleur. Cette méthode par défaut permet de ne pas s'encombrer de code inutile si nous possérons une vue statique correspondant à l'action « `index` ».

La deuxième méthode, « `notFoundAction()` », est liée aux erreurs de route. Si aucune route ne vérifie la demande du client, cette action est appelée par défaut, comme nous le voyons depuis la méthode « `onDispatch()` » :

Zend/Mvc/Controller/AbstractActionController.php

```

public function onDispatch(MvcEvent $e)
{
    [...]
    $action = $routeMatch->getParam('action', 'not-found');
    $method = static::getMethodFromAction($action);

    if (!method_exists($this, $method)) {
        $method = 'notFoundAction';
    }

    $actionResponse = $this->$method();
    $e->setResult($actionResponse);
    return $actionResponse;
}

```

Nous remarquons que la même route est utilisée si le contrôleur ne trouve pas de méthode correspondant à l'action en cours au sein de son instance.

AVEC LE ZEND FRAMEWORK 1

Dans la première version, un contrôleur d'erreur est attribué à la gestion des erreurs, ce qui force une nouvelle distribution. Ici, seul le nom de la méthode est modifié ce qui permet un affichage de la page d'erreur plus rapide. Le plugin `Zend_Controller_Plugin_ErrorHandler` réinitialise les contrôleurs, actions et modules avant de remettre le flag de distribution dans son état initial. Nous avons maintenant une action d'erreur gérée directement par le contrôleur.

Les interfaces du contrôleur de base

Le contrôleur de base abstrait `AbstractController` implémente quatre interfaces : `Dispatchable`, `EventManagerAwareInterface`, `InjectApplicationEventInterface` et `ServiceLocatorAwareInterface`.

- La première interface `Dispatchable`, ne définit qu'une seule méthode :

Zend/Stdlib/Dispatchable.php

```
interface Dispatchable
{
    public function dispatch(Request $request, Response $response =
        null);
}
```

Le contrat passé avec cette interface permet de s'assurer que le contrôleur est « dispatchable ». Cela signifie qu'il va pouvoir être distribué depuis la classe Application, mais aussi que nous pouvons effectuer nous-mêmes ce traitement depuis les contrôleurs pour en récupérer la réponse.

- La deuxième interface EventManagerAwareInterface définit seulement une méthode :

Zend/EventManager/EventManagerAwareInterface.php

```
interface EventManagerAwareInterface
{
    public function setEventManager(EventManagerInterface $eventManager);
}
```

L'interface EventManagerAwareInterface permet de s'assurer de la possibilité d'ajouter l'instance du gestionnaire de services de l'application au sein de l'objet créé. Comme pour le gestionnaire d'évènements, l'instance du gestionnaire de services est automatiquement enregistrée lors de la création d'un objet depuis le gestionnaire de contrôleurs :

Zend/Mvc/Controller/ControllerManager.php

```
public function __construct(ConfigInterface $configuration = null)
{
    [...]
    $this->addInitializer(array($this, 'injectControllerDependencies'),
        false);
}
```

Zend/Mvc/Controller/ControllerManager.php

```
public function injectControllerDependencies($controller,
    ServiceLocatorInterface $serviceLocator)
{
    [...]
    if ($controller instanceof EventManagerAwareInterface) {
        $controller->setEventManager($parentLocator-
            >get('EventManager'));
    }
    [...]
}
```

Cette méthode permet de s'assurer de la possibilité d'ajouter des écouteurs au gestionnaire d'évènements de la classe en récupérant son instance depuis la méthode « getEventManager() ».

- L'interface InjectApplicationEventInterface définit deux méthodes :

Zend/Mvc/InjectApplicationEventInterface.php

```
interface InjectApplicationEventInterface
{
    public function setEvent(Event $event);
    public function getEvent();
}
```

Le contrat passé avec cette interface permet de s'assurer que l'objet peut utiliser des instances d'évènements. Il va aussi servir lors de la création des contrôleurs d'actions afin de leur injecter l'évènement de type MvcEvent. Cet évènement est propre au processus de distribution de l'application, comme on peut le voir dans la liste de ses attributs :

Zend/Mvc/MvcEvent.php

```
class MvcEvent extends Event
{
    protected $application;
    protected $request;
    protected $response;
    protected $result;
    protected $router;
    protected $routeMatch;
    protected $viewModel;
    [...]
}
```

La liste des méthodes d'altération correspondant à chacun de ces attributs n'a pas besoin d'être listée. Nous comprenons l'intérêt de cet évènement qui partage l'objet de requête, de réponse, de route et de vue, ce qui permet à n'importe quel plugin ou contrôleur d'agir avec le contexte MVC de l'application.

L'interface permet quant à elle l'injection de cet objet lors de la distribution :

Zend/Mvc/DispatchListener.php

```
public function onDispatch(MvcEvent $e)
{
    [...]
    if ($controller instanceof InjectApplicationEventInterface) {
        $controller->setEvent($e);
    }
    [...]
}
```

- L'interface ServiceLocatorAwareInterface définit deux méthodes :

Zend/ServiceManager/ServiceLocatorAwareInterface.php

```
interface ServiceLocatorAwareInterface
{
    public function setServiceLocator(ServiceLocatorInterface
        $serviceLocator);
    public function getServiceLocator();
}
```

Ce contrat permet de s'assurer de la possibilité d'ajouter un objet de type ServiceLocatorInterface à l'instance utilisée. Par exemple, le contrôleur AbstractActionController reçoit une instance de type ServiceManager depuis le gestionnaire de contrôleurs qui permet l'instanciation de contrôleur :

Zend/Mvc/Controller/ControllerManager.php

```
public function __construct(ConfigInterface $configuration = null)
{
[...]
$this->addInitializer(array($this, 'injectControllerDependencies'),
false);
}
```

Zend/Mvc/Controller/ControllerManager.php

```
public function injectControllerDependencies($controller,
ServiceLocatorInterface $serviceLocator)
{
[...]
if ($controller instanceof ServiceLocatorAwareInterface) {
    $controller->setServiceLocator($parentLocator->get('Zend\
ServiceManager\ServiceLocatorInterface'));
}
[...]
}
```

Cette interface est très utilisée dans le framework afin de s'assurer de pouvoir transmettre le gestionnaire de services au maximum de composants qui pourraient en avoir besoin. En effet, le gestionnaire de services est un des piliers du framework et il est important que les composants entrant en jeu lors de la partie principale de la couche MVC puissent en bénéficier. Ce comportement se retrouve dans le gestionnaire de plugins ou encore dans certaines aides de vue.

Toutes ces interfaces permettent, dans le processus MVC, l'injection des éléments dont a besoin le contrôleur afin de pouvoir fonctionner au mieux. Tout contrôleur héritant du contrôleur abstrait AbstractActionController a donc un accès à toutes ces informations.

Nous venons de comprendre les différentes injections faites à notre contrôleur, nous allons maintenant analyser le fonctionnement du gestionnaire de contrôleurs que nous avons pu décrire en partie précédemment. Le gestionnaire de contrôleurs hérite du gestionnaire de plugins de base et possède donc ses méthodes d'initialisation ainsi que sa méthode de validation :

Zend/Mvc/Controller/ControllerManager.php

```
public function __construct(ConfigInterface $configuration = null)
{
[...]
$this->addInitializer(array($this, 'injectControllerDependencies'),
false);
}
```

Zend/Mvc/Controller/ControllerManager.php

```

public function injectControllerDependencies($controller,
ServiceLocatorInterface $serviceLocator)
{
if (!$controller instanceof DispatchableInterface) {
    return;
}
$parentLocator = $serviceLocator->getServiceLocator();
if ($controller instanceof ServiceLocatorAwareInterface) {
    $controller->setServiceLocator($parentLocator->get('Zend\
ServiceManager\ServiceLocatorInterface'));
}
if ($controller instanceof EventManagerAwareInterface) {
    $controller->setEventManager($parentLocator-
>get('EventManager'));
}

if (method_exists($controller, 'setPluginManager')) {
    $controller->setPluginManager($parentLocator->get('ControllerPl
uginBroker'));
}
}
}

```

Le gestionnaire ajoute la méthode d'initialisation « injectControllerDependencies() » afin d'initialiser les contrôleurs en lui injectant les objets nécessaires à son fonctionnement. Le gestionnaire de services, le gestionnaire d'événements ainsi que le gestionnaire de plugins sont donc injectés au sein du contrôleur afin qu'il soit opérationnel. Pour plus d'informations sur le fonctionnement du gestionnaire de plugins de base, reportez-vous à la section qui lui est consacrée.

Le contrôleur AbstractRestfulController

Un deuxième contrôleur d'actions abstrait prêt à l'emploi est disponible dans le framework, il s'agit du `AbstractRestfulController`, qui étend aussi la classe `AbstractController`.

La classe `AbstractRestfulController` définit également la méthode « `dispatch()` » qui lui servira de point d'entrée. Une de leurs différences repose sur la méthode « `onDispatch()` », écouteur principal de la distribution du contrôleur :

Zend/Mvc/Controller/AbstractRestfulController.php

```

public function onDispatch(MvcEvent $e)
{
$routeMatch = $e->getRouteMatch();
if (!$routeMatch) {
    [...]
}

$request = $e->getRequest();
$action = $routeMatch->getParam('action', false);
if ($action) {
    $method = static::getMethodFromAction($action);
    if (!method_exists($this, $method)) {
        $method = 'notFoundAction';
    }
}
}

```

```
        }
        $return = $this->$method();
    } else {
        switch (strtolower($request->getMethod())) {
            case 'get':
                if (null !== $id = $routeMatch->getParam('id')) {
                    $action = 'get';
                    $return = $this->get($id);
                    break;
                }
                if (null !== $id = $request->getQuery()->get('id')) {
                    $action = 'get';
                    $return = $this->get($id);
                    break;
                }
                $action = 'getList';
                $return = $this->getList();
                break;
            case 'post':
                $action = 'create';
                $return = $this->processPostData($request);
                break;
            case 'put':
                $action = 'update';
                $return = $this->processPutData($request,
$routeMatch);
                break;
            case 'delete':
                if (null === $id = $routeMatch->getParam('id')) {
                    if (!($id = $request->getQuery()->get('id', false))) {
                        [...]
                    }
                }
                $action = 'delete';
                $return = $this->delete($id);
                break;
            default:
                throw new Exception\DomainException('Invalid HTTP
method!');
        }
        $routeMatch->setParam('action', $action);
    }
    $e->setResult($return);
    return $return;
}
```

Comme nous le remarquons, cette méthode est conçue spécialement pour fonctionner avec l'architecture de web service REST. Son fonctionnement est plutôt simple, le contrôleur implémente des méthodes prêtes à l'emploi afin de pouvoir répondre aux opérations basiques. Une requête de type GET qui contient un paramètre « id » est automatiquement redirigée vers la méthode « get() », sinon la méthode « getList() » est appelée. La méthode POST appellera toujours la méthode « processPostData() », PUT la méthode « processPutData() » et la méthode DELETE provoque l'appel de la méthode « delete() ». Toutes ces fonctions sont implémentées dans cette même classe :

Zend/Mvc/Controller/AbstractRestfulController.php

```
abstract class AbstractRestfulController extends AbstractController
{
    [...]
    abstract public function getList();
    abstract public function get($id);
    abstract public function create($data);
    abstract public function update($id, $data);
    abstract public function delete($id);
    [...]
    public function processPostData(Request $request)
    {
        return $this->create($request->getPost()->toArray());
    }
    public function processPutData(Request $request, $routeMatch)
    {
        [...]
        $content = $request->getContent();
        parse_str($content, $parsedParams);
        return $this->update($id, $parsedParams);
    }
}
```

Cette classe permet de bénéficier d'un contrôle minimum lors de la création d'un web service basé sur l'architecture REST. Il ne reste alors plus qu'à implémenter les méthodes citées au-dessus.

Une méthode « notFoundAction » est aussi implémentée afin de répondre aux demandes d'action qui n'existent pas. Comme nous le remarquons, les méthodes que l'on a présentées sont appelées uniquement si aucune action spécifique n'est demandée :

Zend/Mvc/Controller/AbstractRestfulController.php

```
public function onDispatch(MvcEvent $e)
{
    $routeMatch = $e->getRouteMatch();
    [...]
    $request = $e->getRequest();
    $action = $routeMatch->getParam('action', false);
    if ($action) {
        $method = static::getMethodFromAction($action);
        if (!method_exists($this, $method)) {
            $method = 'notFoundAction';
        }
        $return = $this->$method();
    }
    [...]
}
```

Les actions demandées depuis le paramètre « action » seront donc prioritaires. La classe fournit des méthodes de confort qui permettent à l'utilisateur de ne pas s'encombrer du comportement par défaut pour une action inexistante.

Avec le contrôleur abstrait `AbstractRestfulController` qui présente un squelette de classe dédiée à ce type de traitement, l'implémentation de web service REST n'en sera que facilitée.

10

Les aides d'action

Le Zend Framework 2 a profondément modifié certaines aides d'action peu performantes ou trop complexes pour la tâche qui leur est confiée. Une aide d'action particulièrement intéressante a été ajoutée au framework, il s'agit de l'aide Forward. Cette aide d'action vient à remplacer la méthode « `_forward()` » du Zend Framework 1 qui est particulièrement couteuse en ressources car elle nécessite de refaire le processus de distribution de l'application. Comme nous allons le voir, cette aide d'action tire ses performances de la possibilité de distribuer les actions de chaque contrôleur en dehors du processus MVC.

D'autres aides d'action ont été totalement réécrites afin de simplifier leur interface en réduisant leur nombre de méthodes, comme l'aide Redirect ou Url. La magie liée aux appels des aides d'action a été supprimée, ce qui permet de gagner en clarté et en performances.

AVEC LE ZEND FRAMEWORK 1

La méthode de redirection appelée par « `$this->_helper->redirector("mon-action","mon-controller","mon-module");` » fait appel à une méthode magique du helper d'action. En effet, aucune méthode du nom de « `redirector()` » n'existe dans le contrôleur d'action, c'est donc la méthode « `__call()` » qui appelle elle-même la méthode `direct()` de l'aide d'action `Redirector`.

L'aide d'action Forward

L'aide d'action Forward permet la distribution d'un nouveau couple contrôleur/action afin d'en récupérer le résultat. Voyons un exemple de cette aide d'action dans notre contrôleur d'index :

IndexController.php

```
class IndexController extends ActionController
{
    public function indexAction()
    {
        return new ViewModel();
    }
}
```

```

public function forwardAction()
{
    return $this->plugin('forward')->dispatch('Application\
Controller\IndexController',array('action'=>'index'));
}
}

```

L'action « forwardAction() » retourne maintenant le résultat de l'action « index ». Analysons le comportement de l'aide d'action Forward :

Zend/Mvc/Controller/Plugin/Forward.php

```

public function dispatch($name, array $params = null)
{
    $event    = clone($this->getEvent());
    $locator  = $this->getLocator();
    $scoped   = false;

    if ($locator->has('ControllerLoader')) {
        $locator = $locator->get('ControllerLoader');
        $scoped  = true;
    }
    $controller = $locator->get($name);
    if (!$controller instanceof Dispatchable) {
        [...]
    }
    if (!$scoped) {
        if ($controller instanceof InjectApplicationEventInterface) {
            $controller->setEvent($event);
        }
        if ($controller instanceof ServiceLocatorAwareInterface) {
            $controller->setServiceLocator($locator);
        }
    }
    if ($params) {
        $event->setRouteMatch(new RouteMatch($params));
    }
    [...]

    $return = $controller->dispatch($event->getRequest(), $event-
>getResponse());
    [...]
    return $return;
}

```

L'aide d'action récupère l'environnement du contrôleur courant, évènements et gestionnaire de services, afin de pouvoir les injecter dans les paramètres du nouveau contrôleur à distribuer. Elle utilise la classe de chargement de contrôleur « ControllerLoader » représenté par la fabrique ControllerLoaderFactory que l'on a présentée précédemment. Le gestionnaire de services, l'évènement courant et l'objet de correspondance des routes sont injectés à la nouvelle instance de contrôleur avant sa distribution.

AVEC LE ZEND FRAMEWORK 1

Dans la première version du framework, aucune aide d'action ne permet de distribuer un autre couple contrôleur/action sans perdre le contexte de l'action en cours. La méthode « `_forward()` » du contrôleur de base permet de relancer la boucle de distribution en modifiant le contrôleur, le module et l'action courante. Cette méthode possède l'inconvénient de refaire tous les traitements de la boucle de distribution, mais permet de simuler une autre requête sans devoir forcer une redirection HTTP. Une aide de vue permet par contre de retourner la vue d'un triplet module/contrôleur/action, il s'agit de l'aide « `action()` ». Mais comme on l'a vu, celle-ci est très gourmande et est à éviter. L'aide de vue `Action` fait un clone de la requête, de la réponse et du dispatcher, initialise la réponse et fait la distribution. Rendre n'importe quel contrôleur distribuable permet une plus grande liberté et une plus grande flexibilité aux développeurs.

L'aide d'action Redirect

L'aide d'action `Redirect` a été totalement réécrite afin de simplifier les méthodes proposées au développeur. Deux méthodes sont maintenant disponibles :

Zend/Mvc/Controller/Plugin/Redirect.php

```
public function toRoute($route, array $params = array(), array
$options = array())
{
    $controller = $this->getController();
    [...]
    $response = $this->getResponse();
    $urlPlugin = $controller->plugin('url');
    if (is_scalar($options)) {
        $url = $urlPlugin->fromRoute($route, $params, $options);
    } else {
        $url = $urlPlugin->fromRoute($route, $params, $options,
$reuseMatchedParams);
    }

    $response->headers()->addHeaderLine('Location', $url);
    $response->setStatusCode(302);
    return $response;
}
```

Zend/Mvc/Controller/Plugin/Redirect.php

```
public function toUrl($url)
{
    $response = $this->getResponse();
    $response->headers()->addHeaderLine('Location', $url);
    $response->setStatusCode(302);
    return $response;
}
```

Ce sont deux méthodes simples à comprendre et à implémenter. La première per-

met de rediriger le client suivant une route, ce qui permet de construire automatiquement l'URL concernée, et la deuxième redirige le client suivant une URL définie en paramètre. Voici un exemple trivial de la méthode « `toRoute()` », basée sur les noms de route :

IndexController.php

```
|| public function redirectAction()
|| {
||     return $this->plugin('redirect')->toRoute('home');
|| }
```

Comme nous venons de le voir, l'aide de redirection retourne un objet de type `ResponseInterface`, ce qui permet de court-circuiter l'évènement « `dispatch` » du contrôleur `ActionController` et faciliter le travail de la classe `Application` qui reçoit alors un objet prêt à être envoyé au client.

La deuxième méthode de l'aide d'action est toute aussi simple d'utilisation :

IndexController.php

```
|| public function forwardAction()
|| {
||     return $this->plugin('redirect')->toUrl('http://www.zend.com/fr');
|| }
```

La méthode « `toUrl()` » se contente de rediriger le client vers l'URL indiquée en paramètre, en effectuant une vérification sur sa valeur afin de déterminer si celle-ci est une URL externe valide, dans le cas contraire l'URL sera transformée en une URL relative. Comme pour la méthode précédente, la méthode « `toUrl()` » retourne un objet de réponse dont les headers ont été modifiés afin de permettre une redirection. Aucune redirection n'est effectuée directement depuis l'aide d'action.

Nous pouvons noter que les deux méthodes créent des redirections avec un code HTTP 302 qui représente une redirection temporaire. Si l'on souhaite mettre en place des redirections permanentes, nous avons la possibilité de changer le code HTTP à la volée :

IndexController.php

```
|| public function forwardAction()
|| {
||     return $this->plugin('redirect')->toUrl('http://www.zend.com/fr')-
||           >setStatusCode(301);
|| }
```

L'aide d'action `Redirect` est maintenant plus simple à prendre en main. Il n'existe plus de redirections effectuées sur les triplets module/contrôleur/action qui peuvent générer quelques effets de bord sur les URL, mais uniquement par les identifiants de route ou saisis directement dans le code.

AVEC LE ZEND FRAMEWORK 1

Pour rappel, dans le Zend Framework 1, le routage effectué sur le module/contrôleur/action se base sur les routes par défaut. Le router n'utilise donc pas les routes réécrites si elles existent, ce qui peut donner des URL réécrites d'un côté de l'application et des URL statiques, basées sur la règle module/contrôleur/action, de l'autre. Cela peut s'avérer pénalisant pour les clients de l'application ou encore pour l'application elle-même et son référencement, deux URL pointant sur un contenu identique est pénalisant.

L'aide d'action Url

L'aide d'action Url permet de générer une route depuis un contrôleur en se basant sur le nom de la route. En effet, une seule méthode est disponible :

Zend/Mvc/Controller/Plugin/Url.php

```
public function fromRoute($route, array $params = array(), array
$options = array())
{
    $controller = $this->getController();
    [...]
    $event = $controller->getEvent();
    $router = null;
    [...]
    $options['name'] = $route;
    return $router->assemble($params, $options);
}
```

La méthode « fromRoute() » récupère le router de l'application en utilisant les propriétés de l'évènement du contrôleur courant, et peut ainsi composer la route demandée afin de la retourner.

Voici un exemple d'utilisation simple :

IndexController.php

```
public function indexAction()
{
    $home = $this->plugin('url')->fromRoute('home');
    $homeCanonical = $this->plugin('url')->fromRoute('home', array(),
array('force_canonical' => true));
}
```

Les routes retournées vont être successivement « / » et « <http://monserveur.local:80/> ».

AVEC LE ZEND FRAMEWORK 1

Une autre méthode est disponible dans l'aide d'action Url, il s'agit de la méthode « simple() ». Cette méthode permet de créer une route en fonction du triplet module/contrôleur/action basé sur les routes par défaut. Comme pour l'ancienne aide d'action Redirector, les mêmes problèmes de routes multiples peuvent se poser car le router n'utilise pas les routes réécrites.

L'aide d'action PostRedirectGet

L'aide d'action PostRedirectGet permet de mettre en œuvre le pattern PRG (Post/Redirect/Get) qui résout les problèmes de soumissions multiples de formulaire. L'aide d'action vérifie si des données ont été postées depuis la méthode POST afin de les enregistrer en session avant d'effectuer une redirection :

Zend/Mvc/Controller/Plugin/PostRedirectGet.php

```
public function __invoke($redirect, $redirectToUrl = false)
{
    $controller = $this->getController();
    $request = $controller->getRequest();
    [...]
    $container = new Container('prg_post1');
    if ($request->isPost()) {
        $container->setExpirationHops(1, 'post');
        $container->post = $request->getPost()->toArray();
        [...]
    }
    [...]
}
```

Le contenu des données en POST est enregistré en session sous la clé « prg_post1 » afin d'être récupéré depuis la page où l'on sera redirigé :

Zend/Mvc/Controller/Plugin/PostRedirectGet.php

```
public function __invoke($redirect, $redirectToUrl = false)
{
    [...]
    if ($request->isPost()) {
        [...]
    } else {
        if ($container->post !== null) {
            $post = $container->post;
            unset($container->post);
            return $post;
        }
        return false;
    }
}
```

L'aide d'action retourne les données enregistrées lors de la soumission des données. Une fois celles-ci récupérées, elles se détruisent alors automatiquement. Voici

un exemple simple d'utilisation depuis un contrôleur d'action :

IndexController.php

```
class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        $this->request->setMethod('POST');
        $this->request->setPost(new Parameters(array(
            'post_key' => 'value'
        )));
        return $this->prg('/forward', true);
    }

    public function forwardAction()
    {
        $datas = $this->prg(null);
    }
}
```

La méthode « index » utilise l'aide d'action PostRedirectGet afin d'effectuer sa redirection sur l'action « forward ». Une fois redirigées, les données passées en POST peuvent alors être récupérées pour effectuer le traitement souhaité. L'aide d'action PostRedirectGet peut aussi être récupérée depuis l'identifiant « postredirectget », comme nous le voyons depuis le gestionnaire de plugins :

Zend/Mvc/Controller/PluginManager.php

```
class PluginManager extends AbstractPluginManager
{
    protected $invokableClasses = array(
        [...]
        'postredirectget' => 'Zend\Mvc\Controller\Plugin\PostRedirectGet',
        [...]
    );
    protected $aliases = array(
        'prg'           => 'postredirectget',
    );
    [...]
}
```

Comme nous pouvons le voir, « prg » n'est qu'un alias de « postredirectget », les deux identifiants peuvent être utilisés.

Création d'une aide d'action

Lorsque vous développez votre application, vous pouvez avoir besoin de créer une aide d'action personnalisée afin de factoriser le code redondant de votre application. La première étape est d'étendre la classe de base Zend\Mvc\Controller\Plugin\AbstractPlugin qui donne un accès au contrôleur courant :

Zend/Mvc/Controller/Plugin/AbstractPlugin.php

```
abstract class AbstractPlugin
{
    protected $controller;
    public function setController(Dispatchable $controller)
    {
        $this->controller = $controller;
    }

    public function getController()
    {
        return $this->controller;
    }
}
```

Une fois l'aide d'action écrite, il ne reste plus qu'à l'enregistrer auprès du gestionnaire de plugins. Cette opération peut se faire depuis la méthode « `getControllerPluginConfiguration()` » de la classe de module :

Module.php

```
public function getControllerPluginConfig()
{
    return array(
        'invokables' => array(
            'myplugin' => 'MyLib\Mvc\Controller\Plugin\MyPlugin',
        ),
    );
}
```

Il est également possible d'inscrire l'aide de vue depuis le fichier de configuration du module sous le nom de clé « `controller_plugins` » :

module.config.php

```
<?php
return array(
[...]
'controller_plugins' => array(
    'invokables' => array(
        'myplugin' => 'MyLib\Mvc\Controller\Plugin\MyPlugin',
    ),
),);
```

Pour plus d'informations sur la gestion interne de la configuration, reportez-vous au chapitre consacré aux modules. L'exemple ci-dessus permet l'ajout d'une autre définition du plugin « `myplugin` » que l'on peut alors récupérer depuis notre contrôleur avec l'instruction :

IndexController.php

```
public function indexAction()
{
    $myPlugin = $this->plugin('myplugin');
[...]
}
```

Le gestionnaire de plugins

Le terme « plugin » est utilisé dans le Zend Framework 2 au sens large et englobe de nombreux objets. Il peut être utilisé pour désigner les aides d'action ou de vue, les filtres, les validateurs ou encore les adaptateurs des différents composants. Afin de simplifier la gestion des plugins, une classe de base existe et implémente une méthode de chargement afin de permettre aux gestionnaires d'en tirer profit, facilitant le chargement et la personnalisation des noms de plugins.

La classe de base

Le gestionnaire de plugins gère la création et l'enregistrement des instances de plugins, mais est aussi capable de gérer la résolution des noms de plugins. Chaque gestionnaire du framework étend le gestionnaire de plugins de base Zend\ServiceManager\AbstractPluginManager qui est une implémentation du gestionnaire de services. En effet, il n'existe plus de composant spécifique au chargement de plugins, le gestionnaire de services est utilisé comme élément de base pour le chargement de classes car il dispose de toutes les méthodes pour répondre aux principaux besoins de chargement :

Zend\ServiceManager\AbstractPluginManager.php

```
|| abstract class AbstractPluginManager extends ServiceManager
||   implements ServiceLocatorAwareInterface
||   {}
```

Notons aussi que le gestionnaire de base implémente l'interface ServiceLocatorAware, ce qui va lui permettre de recevoir une instance du gestionnaire de services. Nous aurons donc accès à notre gestionnaire de services principal au sein de tous les gestionnaires du framework : gestionnaire des aides de vue, des aides d'action, etc. Ce comportement pourra s'avérer pratique afin d'utiliser les fabriques que l'on a définies dans notre application au sein de nos différents gestionnaires.

Le composant ServiceManager est spécialisé dans le chargement, l'instanciation et l'initialisation de classe, il serait dommage de ne pas utiliser sa puissance afin de charger les plugins du framework. Cela permet d'autant plus une meilleure cohérence dans le framework.

Voyons comment est initialisé et configuré le gestionnaire de base :

Zend\ServiceManager\AbstractPluginManager.php

```
public function __construct(ConfigurationInterface $configuration = null)
{
    parent::__construct($configuration);

    $self = $this;
    $this->addInitializer(function ($instance) use ($self) {
        if ($instance instanceof ServiceLocatorAwareInterface) {
            $instance->setServiceLocator($self);
        }
        if ($instance instanceof ServiceManagerAwareInterface) {
            $instance->setServiceManager($self);
        }
    });
}
```

Le gestionnaire accepte un objet de configuration dans son constructeur, ce qui va permettre de personnaliser le gestionnaire qui sera instancié. Une fois le gestionnaire configuré, celui-ci ajoute une méthode d'initialisation qui permettra de fournir aux objets créés l'instance du gestionnaire de services. Pour rappel, l'interface ServiceManagerAwareInterface définit une méthode permettant à l'objet qui l'implémente de s'assurer que celui-ci peut recevoir une instance du gestionnaire de services :

Zend\ServiceManager\ServiceManagerAwareInterface.php

```
interface ServiceManagerAwareInterface
{
    public function setServiceManager(ServiceManager $serviceManager);
}
```

C'est la méthode « `get()` » qui réalise le chargement de l'instance du plugin. Voici le code de la méthode de chargement :

Zend\ServiceManager\AbstractPluginManager.php

```
public function get($name, $options = array(),
    $usePeeringServiceManagers = true)
{
    if (!$this->has($name) && $this->autoAddInvokableClass && class_
        exists($name)) {
        $this->setInvokableClass($name, $name);
    }
    $this->creationOptions = $options;
    $instance = parent::__get($name, $usePeeringServiceManagers);
    $this->creationOptions = null;
    $this->validatePlugin($instance);

    return $instance;
}
```

Le gestionnaire vérifie en premier lieu s'il est capable de charger l'objet demandé depuis la méthode « `has()` » du gestionnaire de services. Si celui-ci n'est pas en mesure de charger l'objet demandé mais que la classe existe bien, le gestionnaire

ajoute alors la classe comme étant « invokable » si l'ajout automatique est activé depuis l'attribut « autoAddInvokableClass », ce qui va lui permettre tout de même le chargement. Le fait de pouvoir charger une classe existante depuis son nom complet est une des particularités du gestionnaire de plugins.

Les options reçues en paramètres sont enregistrées afin de pouvoir être utilisées dans la fabrique de classes dites « invokables ». Le gestionnaire de services utilise la méthode « createFromInvokable() » afin d'instancier les classes de ce type. Cette fabrique est redéfinie dans le gestionnaire de plugins afin de pouvoir utiliser les options fournies en paramètre :

Zend\ServiceManager\AbstractPluginManager.php

```
protected function createFromInvokable($canonicalName,
$requestedName)
{
$invokable = $this->invokableClasses[$canonicalName];
if (!class_exists($invokable)) {
    [...]
}
if (null === $this->creationOptions
    || (is_array($this->creationOptions) && empty($this->creationOptions))
    ) {
    $instance = new $invokable();
} else {
    $instance = new $invokable($this->creationOptions);
}

return $instance;
}
```

Nous retrouvons l'attribut « creationOptions » lors l'instanciation de la classe de plugin, et qui est ensuite remis à la valeur « null ». Une fois le plugin instancié, la méthode « validatePlugin() » définie par le gestionnaire qui étend la classe AbstractPluginManager permet de valider le type du plugin instancié.

Maintenant que nous avons vu l'étape de l'instanciation, il est nécessaire de comprendre comment se passe la résolution des noms de plugins. Chaque plugin dispose d'un nom court qui facilite l'écriture.

Nous avons vu que le gestionnaire de base étend la classe ServiceManager qui lui permet de tirer pleinement profit de ses capacités. Chaque gestionnaire qui hérite de la classe AbstractPluginManager aura donc la possibilité de définir ses classes de type « invokable » et autres fabriques personnalisées, ce qui lui permettra de lui affecter comme identifiant de fabriques, le nom court du plugin.

Afin de mettre en œuvre ce que nous venons d'expliquer, analysons alors les différents gestionnaires disponibles dans le framework.

Les gestionnaires du framework

Les gestionnaires de plugins du framework définissent chacun leur fabrique pour la résolution de nom de plugins. Chacun des gestionnaires peut alors ajouter une

couche de traitement supplémentaire lors du chargement depuis les méthodes d'initialisation fournies par le gestionnaire de services afin de pouvoir peupler le plugin d'objets propres à son fonctionnement. Prenons un premier exemple avec le composant Zend\ Mvc\ Controller\ PluginManager, gestionnaire des aides d'action :

Zend/Mvc/Controller/PluginManager.php

```
class PluginManager extends AbstractPluginManager
{
    protected $invokableClasses = array(
        'flashmessenger' => 'Zend\ Mvc\ Controller\ Plugin\ FlashMessenger',
        'forward'         => 'Zend\ Mvc\ Controller\ Plugin\ Forward',
        [...]
    );
    protected $controller;

    public function __construct(ConfigurationInterface $configuration =
        null)
    {
        parent::__construct($configuration);
        $this->addInitializer(array($this, 'injectController'));
    }

    public function setController(DispatchableInterface $controller)
    {...}
    public function getController() {...}

    public function injectController($plugin)
    {
        [...]
        if (!method_exists($plugin, 'setController')) {
            return;
        }
        $controller = $this->getController();
        if (!($controller instanceof DispatchableInterface)) {
            return;
        }
        $plugin->setController($controller);
    }

    public function validatePlugin($plugin)
    {
        if ($plugin instanceof Plugin\PluginInterface) {
            return;
        }
        throw new Exception\ InvalidPluginException(...);
    }
}
```

Le gestionnaire de plugins ajoute une méthode d'initialisation lors de son instantiation. Cette méthode permet l'injection du contrôleur courant à l'aide d'action utilisée afin qu'elle puisse l'utiliser lors de la tâche qu'elle effectue :

Zend/Mvc/Controller/PluginManager.php

```
public function __construct(ConfigurationInterface $configuration =
null)
{
[...]
$this->addInitializer(array($this, 'injectController'));
}
```

Notons aussi que la méthode de validation des plugins vérifie que celui-ci implémente bien l'interface Zend\Mvc\Controller\Plugin\PluginInterface qui définit deux méthodes :

Zend\Mvc\Controller\Plugin\PluginInterface.php

```
interface PluginInterface
{
public function setController(Dispatchable $controller);
public function getController();
}
```

Pour répondre à ce besoin, votre aide d'action devra comporter ces deux méthodes ou étendre l'aide d'action abstraite AbstractPlugin qui implémente déjà ces deux méthodes.

Le gestionnaire d'aides de vue du framework fonctionne de la même manière :

Zend\View\Helper\PluginManager.php

```
class HelperPluginManager extends AbstractPluginManager
{
protected $invokableClasses = array(
[...]
'headlink'           => 'Zend\View\Helper\HeadLink',
'headmeta'           => 'Zend\View\Helper\HeadMeta',
'headscript'         => 'Zend\View\Helper\HeadScript',
[...]
);
protected $renderer;

public function __construct(ConfigurationInterface $configuration =
null)
{
    parent::__construct($configuration);
    $this->addInitializer(array($this, 'injectRenderer'));
}

public function setRenderer(Renderer\RendererInterface $renderer)
{[...]}
public function getRenderer() {[...]}
public function injectRenderer($helper)
{
    $renderer = $this->getRenderer();
    if (null === $renderer) {
        return;
    }
    $helper->setView($renderer);
}
```

```

public function validatePlugin($plugin)
{
    if ($plugin instanceof Helper\HelperInterface) {
        return;
    }
    throw new Exception\InvalidHelperException([...]);
}
}

```

Le gestionnaire d'aides de vue définit ses propres classes « invokables » et injecte l'objet de rendu à chacun des plugins. Le type du plugin est valide si celui-ci implémente les deux méthodes de l'interface Zend\View\Helper\HelperInterface :

Zend/View/Helper/HelperInterface.php

```

interface HelperInterface
{
    public function setView(Renderer $view);
    public function getView();
}

```

Tous les gestionnaires de plugins fonctionnent de la même manière. Les gestionnaires étendent le gestionnaire de base, qui étend la classe ServiceManager afin de tirer profit de ses possibilités de fabrication, et redéfinissent chacun leur classe de validation et ajoutent leurs méthodes d'initialisation.

D'autres composants utilisent également la classe de base AbstractPluginManager afin de charger leurs plugins. Le composant de cache, par exemple, utilise le gestionnaire de base afin de charger les plugins qui interviendront lors de la création de cache :

Zend/Cache/Storage/PluginManager.php

```

class PluginManager extends AbstractPluginManager
{
    protected $invokableClasses = array(
        'clearexpiredbyfactor' => 'Zend\Cache\Storage\Plugin\ClearExpiredByFactor',
        [...]
    );
    protected $shareByDefault = false;

    public function validatePlugin($plugin)
    {
        if ($plugin instanceof Plugin\PluginInterface) {
            return;
        }
        throw new Exception\RuntimeException([...]);
    }
}

```

Le composant de pagination, de cryptage ou encore de gestion de logs utilisent le gestionnaire de base pour gérer le chargement de leurs plugins. Cette manière de fonctionner est largement répandue dans le framework et nos composants personnalisés peuvent également reprendre ce fonctionnement pour la gestion des chargements de plugins.

12

Les vues

Le composant de vue a été refondu entièrement afin d'améliorer sa flexibilité et sa répartition des tâches. Capable de gérer les imbrications de vues entre elles ou encore le format des rendus, les composants responsables des actions liées à la vue sont nombreux et permettant maintenant aux développeurs une grande liberté sur le rendu de leur application.

Les responsabilités de rendu (conteneur de variables, injection de valeurs, localisation de template, rendu de vue, injection de rendu, gestion des erreurs) ont chacune été affectées à un composant dédié à sa tâche. Il est maintenant beaucoup moins complexe de manier les vues ou d'intervenir sur leur rendu tant l'organisation est séparée.

AVEC LE ZEND FRAMEWORK 1

Le `Zend_View` a trop de responsabilités, ce qui le rend peu flexible. Le `Zend_Layout` est indépendant du `Zend_View` alors qu'il se comporte comme une vue mais avec un niveau de hiérarchisation supérieur. Le `ViewRenderer` a aussi été supprimé afin de séparer les responsabilités lors du processus de rendu.

Le gestionnaire de vues

Afin de comprendre rapidement le rôle de chacun des composants liés à la vue, il est important de se faire une rapide idée sur le cycle de la gestion des vues.

Les vues sont construites depuis le retour des méthodes d'actions de nos contrôleurs. Le retour de l'action est récupéré depuis des écouteurs spécialisés qui transforment ce retour en un objet de vue. Une fois la vue construite, le template lui est injecté avant qu'elle ne soit elle-même injectée au sein de la vue principale, le layout. Une fois l'évènement demandant le rendu lancé, un objet spécialisé prend le relais afin de rendre la vue pour être ensuite injecté à l'objet de réponse un peu plus tard. Un dernier évènement permet de notifier un objet spécialisé dans le renvoi d'objet de réponse au client.

Chacune de ces grandes étapes va être approfondie dans ce chapitre au fil des sections. Ce chapitre est assez dense et il est conseillé d'avoir le code du framework

sous la main afin de mieux comprendre les explications et les relations entre objets.

Avant d'examiner en détail le comportement des composants intervenant lors du rendu de la vue, il est important de comprendre les évènements et écouteurs qui entrent en jeu. La liste des écouteurs qui s'attachent aux actions liées à la vue est disponible dans la classe du gestionnaire de vues. Nous détaillerons chacun des composants, mais le nom de chacun peut vous donner un rapide aperçu de son rôle précis :

Zend/Mvc/View/Http/ViewManager.php

```
public function onBootstrap($event)
{
    [...]
    $routeNotFoundStrategy = $this->getRouteNotFoundStrategy();
    $exceptionStrategy = $this->getExceptionStrategy();
    $mvcRenderingStrategy = $this->getMvcRenderingStrategy();
    $createViewModelListener = new CreateViewModelListener();
    $injectTemplateListener = new InjectTemplateListener();
    $injectViewModelListener = new InjectViewModelListener();
    $sendResponseListener = new SendResponseListener();
    [...]
    $sharedEvents->attach('Zend\Stdlib\DispatchableInterface',
    MvcEvent::EVENT_DISPATCH, array($createViewModelListener,
    'createViewModelFromArray'), -80);

    $sharedEvents->attach('Zend\Stdlib\DispatchableInterface',
    MvcEvent::EVENT_DISPATCH, array($routeNotFoundStrategy,
    'prepareNotFoundViewModel'), -90);

    $sharedEvents->attach('Zend\Stdlib\DispatchableInterface',
    MvcEvent::EVENT_DISPATCH, array($createViewModelListener,
    'createViewModelFromNull'), -80);

    $sharedEvents->attach('Zend\Stdlib\DispatchableInterface',
    MvcEvent::EVENT_DISPATCH, array($injectTemplateListener,
    'injectTemplate'), -90);

    $sharedEvents->attach('Zend\Stdlib\DispatchableInterface',
    MvcEvent::EVENT_DISPATCH, array($injectViewModelListener,
    'injectViewModel'), -100);
}
```

Notons pour l'instant que le gestionnaire de vues gère la liste entière des écouteurs de vues. Cela nous permettra par la suite, et au cours des développements de nos projets, de savoir où regarder lorsque nous recherchons à intervenir sur le rendu de vue ou tout simplement comprendre un détail qui nous aurait échappé.

Examinons en détail le travail de chacun des écouteurs pour mieux comprendre leur rôle dans le rendu de la vue et les grandes étapes de celle-ci. Les premiers écouteurs à intervenir sont ceux qui s'occupent de préparer l'objet de vue.

Préparation des vues

La préparation des vues est la première étape du cycle de rendu. Cette étape permet de transformer le retour de l'action courante en un objet de vue qui pourra

facilement être travaillé par le framework.

Lors du lancement de l'évènement « dispatch » par l'application, le premier objet notifié est l'objet DispatchListener qui s'occupe de gérer la distribution de l'action à effectuer. En effet, il est nécessaire que l'action soit distribuée si l'on souhaite intervenir sur le résultat de celle-ci. Les autres écouteurs de l'évènement sont ensuite notifiés, dont ceux destinés à la préparation des vues.

Les écouteurs responsables de la préparation des vues agissent sur l'évènement « dispatch » avec une faible priorité, ce qui les place à la fin des notifications. Voici leurs actions :

Zend/Mvc/View/Http/CreateViewModelListener.php

```
public function createViewModelFromArray(MvcEvent $e)
{
    $result = $e->getResult();
    if (!ArrayUtils::hasStringKeys($result, true)) {
        return;
    }
    $model = new ViewModel($result);
    $e->setResult($model);
}
```

Zend/Mvc/View/Http/CreateViewModelListener.php

```
public function createViewModelFromNull(MvcEvent $e)
{
    $result = $e->getResult();
    if (null !== $result) {
        return;
    }
    $model = new ViewModel;
    $e->setResult($model);
}
```

Les écouteurs agissent sur le retour de l'action courante si celle-ci n'est pas au format attendu. Ces deux méthodes interviennent sur l'attribut « result » de l'objet MvcEvent, car comme nous l'avons vu dans la section dédiée aux contrôleurs, le retour de l'action en cours peuple cet attribut :

Zend/Mvc/Controller/AbstractActionController.php

```
public function onDispatch(MvcEvent $e)
{
    [...]
    $actionResponse = $this->$method();
    $e->setResult($actionResponse);
    return $actionResponse;
}
```

Si l'objet retourné par l'action est un tableau, alors la méthode « createViewModelFromArray() » se charge de le convertir en objet de type ViewModel. Cet écouteur offre la possibilité de retourner un tableau lors de la méthode d'action :

Retour de tableau depuis une vue

```
|| public function indexAction()
|| {
||     return array('key'=>'value') ;
|| }
```

La deuxième méthode, « `createViewModelFromNull()` », se charge de convertir les retours de valeurs « `null` » en une instance de `ViewModel` sans paramètre. Cela nous permet de ne pas faire de retour inutile au sein de notre action :

Retour null depuis une action

```
|| public function indexAction()
|| {
|| }
```

Ces deux écouteurs offrent plus de flexibilité avec le type de retour de l'action courante. Analysons maintenant la préparation des vues d'erreurs.

Préparation des vues d'erreurs

Les méthodes qui gèrent la préparation des vues d'erreurs écoutent l'évènement « `dispatch.error` ». Il y a trois méthodes distinctes qui agissent sur les vues et l'évènement d'erreur lors de la distribution :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
|| public function detectNotFoundError(MvcEvent $e)
|| {
||     $error = $e->getError();
||     if (empty($error)) {
||         return;
||     }
||     switch ($error) {
||         case Application::ERROR_CONTROLLER_NOT_FOUND:
||         case Application::ERROR_CONTROLLER_INVALID:
||         case Application::ERROR_ROUTER_NO_MATCH:
||             $this->reason = $error;
||             $response = $e->getResponse();
||             if (!$response) {
||                 $response = new HttpResponse();
||                 $e->setResponse($response);
||             }
||             $response->setStatusCode(404);
||             break;
||         default:
||             return;
||     }
|| }
```

Cette méthode permet de modifier le code HTTP de la réponse pour les erreurs de routes et de contrôleur inexistant ou invalide. La cause de l'erreur est enregistrée dans l'attribut « `$reason` » afin de pouvoir s'en resserrer ultérieurement pour une description de l'erreur en cours. Une fois ces modifications faites, la méthode de

préparation de la vue d'erreur peut donc intervenir :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
public function prepareNotFoundViewModel(MvcEvent $e)
{
    $vars = $e->getResult();
    if ($vars instanceof Response) {
        return;
    }
    $response = $e->getResponse();
    if ($response->getStatusCode() != 404) {
        return;
    }

    $model = new ViewModel\ViewModel();
    $model->setVariable('message', 'Page not found.');
    $model->setTemplate($this->getNotFoundTemplate());
    $this->injectNotFoundReason($model, $e);
    $this->injectException($model, $e);
    $this->injectController($model, $e);
    $e->setResult($model);
}
```

Cette méthode ne fonctionne que si le code HTTP a bien été modifié en code 404. Dans le cas d'erreur sur la route ou de contrôleur inexistant ou invalide, l'écouteur précédent s'est occupé de modifier le code HTTP. Une vue est donc créée afin de l'injecter au résultat de l'évènement courant, actuellement de valeur « null » car normalement peuplé par le retour de notre action courante lors d'une distribution sans erreur. Enfin, trois méthodes injectent les données nécessaires à la description de l'erreur :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
public function prepareNotFoundViewModel(MvcEvent $e)
{
    [...]
    $this->injectNotFoundReason($model, $e);
    $this->injectException($model, $e);
    $this->injectController($model, $e);
    [...]
}
```

La première méthode injecte l'origine de l'erreur :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
protected function injectNotFoundReason($model)
{
    if (!$this->displayNotFoundReason()) {
        return;
    }
    if ($this->reason) {
        $model->setVariable('reason', $this->reason);
        return;
    }
    $model->setVariable('reason', Application::ERROR_CONTROLLER_CANNOT_DISPATCH);
}
```

L'attribut « \$reason », descriptif de l'erreur en cours, peuplé dans une méthode précédente est maintenant utilisé. Les deux autres méthodes d'injection complètent le descriptif :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
protected function injectException($model, $e)
{
    if (!$this->displayExceptions()) {
        return;
    }
    $exception = $e->getParam('exception', false);
    if (!$exception instanceof \Exception) {
        return;
    }
    $model->setVariable('exception', $exception);
}
```

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
protected function injectController($model, $e)
{
    if (!$this->displayExceptions() && !$this->displayNotFoundReason()) {
        return;
    }
    [...]
    $controllerClass = $e->getControllerClass();
    $model->setVariable('controller', $controller);
    $model->setVariable('controller_class', $controllerClass);
}
```

Les erreurs sont affichées dans notre exemple, le fichier de configuration permet l'affichage de l'erreur et de son exception :

module.config.php

```
<?php
return array(
    [...]
    'view_manager' => array(
        'display_not_found_reason' => true,
        'display_exceptions'       => true,
        [...]
        ),
        ),
    );
);
```

Les paramètres définis pour le gestionnaire de vues sont injectés lors de la construction des objets correspondants. La construction des écouteurs responsables de la gestion des erreurs se fait dans l'écouteur du bootstrap du gestionnaire de vue :

Zend/Mvc/View/Http/ViewManager.php

```
public function onBootstrap($event) {
    [...]
    $routeNotFoundStrategy    = $this->getRouteNotFoundStrategy();
    $exceptionStrategy        = $this->getExceptionStrategy();
    [...]
}
```

Voici la construction de l'écouteur responsable la gestion des exceptions :

Zend/Mvc/View/Http/ViewManager.php

```
public function getExceptionStrategy()
{
    [...]
    $this->exceptionStrategy = new ExceptionStrategy();
    $displayExceptions = false;
    $exceptionTemplate = 'error';
    if (isset($this->config['display_exceptions'])) {
        $displayExceptions = $this->config['display_exceptions'];
    }
    if (isset($this->config['exception_template'])) {
        $exceptionTemplate = $this->config['exception_template'];
    }
    $this->exceptionStrategy->setDisplayExceptions($displayExceptions);
    $this->exceptionStrategy->setExceptionTemplate($exceptionTemplate);
    [...]
}
```

Nous retrouvons ici les clés « display_exceptions » et « exception_template » que l'on a définies dans la configuration ainsi que les valeurs par défaut qui leur sont affectées : « false » et « error ». L'écouteur concernant la gestion d'erreurs utilise aussi la configuration :

Zend/Mvc/View/Http/ViewManager.php

```
public function getRouteNotFoundStrategy()
{
    [...]
    $this->routeNotFoundStrategy = new RouteNotFoundStrategy();
    $displayExceptions = false;
    $displayNotFoundReason = false;
    $notFoundTemplate = '404';

    if (isset($this->config['display_exceptions'])) {
        $displayExceptions = $this->config['display_exceptions'];
    }
    if (isset($this->config['display_not_found_reason'])) {
        $displayNotFoundReason = $this->config['display_not_found_reason'];
    }
    if (isset($this->config['not_found_template'])) {
        $notFoundTemplate = $this->config['not_found_template'];
    }

    $this->routeNotFoundStrategy->setDisplayExceptions($displayExceptions);
    $this->routeNotFoundStrategy->setDisplayNotFoundReason($displayNotFoundReason);
    $this->routeNotFoundStrategy->setNotFoundTemplate($notFoundTemplate);
    [...]
}
```

Nous retrouvons cette fois les clés « display_exceptions », « not_found_template » et « display_not_found_reason » avec les valeurs par défaut « false », « false » et « 404 ».

Pour les autres erreurs de l'application, qui ne se produisent pas lors d'un problème de route ou de contrôleur, la méthode « `prepareExceptionViewModel()` » de la classe `Zend\Mvc\View\Http\ExceptionStrategy` gère le résultat de l'évènement `MvcEvent`, et modifie aussi le code HTTP de l'objet de réponse par le code 500. Cette méthode est peu différente des autres méthodes :

Zend/Mvc/View/Http/ExceptionStrategy.php

```
public function prepareExceptionViewModel(MvcEvent $e)
{
    $error = $e->getError();
    if (empty($error)) {
        return;
    }
    $result = $e->getResult();
    if ($result instanceof Response) {
        return;
    }
    switch ($error) {
        [...]
        case Application::ERROR_EXCEPTION:
        default:
            $model      = new ViewModel\ViewModel(array(
                'message'          => 'An error occurred during
execution; please try again later.',
                'exception'        => $e->getParam('exception'),
                'display_exceptions' => $this-
>displayExceptions(),
            ));
            $model->setTemplate($this->getExceptionTemplate());
            $e->setResult($model);
            $response = $e->getResponse();
            if (!$response) {
                $response = new HttpResponse();
                $e->setResponse($response);
            }
            $response->setStatusCode(500);
            break;
    }
}
```

Une méthode utilisée lors de la préparation du rendu d'erreur est attachée à l'évènement « `dispatch` », il s'agit de la méthode « `prepareNotFoundViewModel()` » que nous avons vue précédemment. Cet écouteur ne travaille que lors de la détection d'un code HTTP 404 :

Zend/Mvc/View/Http/RouteNotFoundStrategy.php

```
public function prepareNotFoundViewModel(MvcEvent $e)
{
    $vars = $e->getResult();
    if ($vars instanceof Response) {
        return;
    }
    $response = $e->getResponse();
    if ($response-&gtgetStatusCode() != 404) {
        return;
    }
    [...]
}
```

La modification du code HTTP au sein de notre action aurait alors pour effet de générer une page d'erreur :

Erreur depuis le changement de code HTTP

```
public function indexAction()
{
    $this->response->setStatusCode(-404);
    return new ViewModel(array('key'=>'value'));
}
```

L'analyse des écouteurs, avec la responsabilité du contrôle et de la préparation des vues est terminée. Nous pouvons maintenant nous intéresser au cœur du système du rendu de vue.

Rendu de la vue

Le processus d'affichage de la vue peut être décomposé en deux étapes. La première étape concerne la construction et la préparation des vues que nous venons de voir, suivi par le rendu effectif de la vue où de nombreux composants interviennent. Examinons le cheminement précis de chacune de ces étapes.

Construction des vues

Comme présenté dans la section précédente, la construction de la vue se fait lors de la notification des écouteurs sur l'évènement « `dispatch` ». Une fois l'objet de vue créé, la méthode « `injectTemplate()` » de la classe `Zend\ Mvc\ View\ InjectTemplateListener`, notifiée aussi par l'évènement « `dispatch` », se charge de modifier le nom du fichier de vue qui doit être rendu :

Zend/Mvc/View/Http/InjectTemplateListener.php

```
public function injectTemplate(MvcEvent $e)
{
    $model = $e->getResult();
    [...]
    $template = $model->getTemplate();
    [...]
    $routeMatch = $e->getRouteMatch();
    $controller = $e->getTarget();
    [...]
    $module    = $this->deriveModuleNamespace($controller);
    $controller = $this->deriveControllerClass($controller);
    $template   = $this->inflectName($module);
    if (!empty($template)) {
        $template .= '/';
    }
    $template .= $this->inflectName($controller);
    $action    = $routeMatch->getParam('action');
    if (null !== $action) {
        $template .= '/' . $this->inflectName($action);
    }
    $model->setTemplate($template);
}
```

Nous remarquons que, par défaut, le nom du template est composé du nom du module, concaténé avec le nom du contrôleur et de l'action :

Zend/Mvc/View/Http/InjectTemplateListener.php

```
public function injectTemplate(MvcEvent $e)
{
[...]
$module      = $this->deriveModuleNamespace($controller);
[...]
$template   = $this->inflectName($module);
[...]
$template .= $this->inflectName($controller);
[...]
if (null !== $action) {
    $template .= '/' . $this->inflectName($action);
}
$model->setTemplate($template);
}
```

Nous remarquons également que l'injection de fichier de vue est effectuée si celui-ci n'existe pas déjà :

Zend/Mvc/View/Http/InjectTemplateListener.php

```
public function injectTemplate(MvcEvent $e)
{
[...]
$template = $model->getTemplate();
if (!empty($template)) {
    return;
}
[...]
}
```

En effet, nous comprenons alors qu'il est possible de changer le template à la volée lors de la construction de notre objet de vue sans que celui-ci soit systématiquement modifié ultérieurement :

IndexController.php

```
public function indexAction()
{
$viewModel = new ViewModel();
$viewModel->setTemplate('index/promotion');
return $viewModel;
}
```

Cet exemple permet d'afficher la vue « index/promotion » lors de l'appel à la page d'index. Si le template n'a pas été modifié au cours de l'action, celui-ci prend le nom du contrôleur courant suivi du nom de son action.

La méthode « injectViewModel() » de la classe `InjectViewModelListener` injecte ensuite la vue courante dans la vue parente existante, le layout. Cette partie est intéressante du fait que ce système de fonctionnement est une nouveauté du Zend Framework 2 :

Zend/Mvc/View/Http/InjectViewModelListener.php

```
public function injectViewModel(MvcEvent $e)
{
    $result = $e->getResult();
    if (!$result instanceof ViewModel) {
        return;
    }
    $model = $e->getViewModel();
    if ($result->terminate()) {
        $e->setViewModel($result);
        return;
    }
    $model->addChild($result);
}
```

La variable « \$result » contient la vue courante retournée par notre action courante, ou construite depuis les méthodes de préparation que nous avons vues, et l'objet « \$model » représente la vue parente.

L'initialisation du layout est effectuée par le gestionnaire de vues :

Zend/Mvc/View/Http/ViewManager.php

```
public function getViewModel()
{
    if ($this->viewModel) {
        return $this->viewModel;
    }
    $this->viewModel = $model = $this->event->getViewModel();
    $model->setTemplate($this->getLayoutTemplate());
    return $this->viewModel;
}
```

L'appel à la méthode « getViewModel() » de l'évènement MvcEvent construit une instance de l'objet View\ViewModel si celle-ci a la valeur « null » :

Zend/Mvc/MvcEvent.php

```
public function getViewModel()
{
    if (null === $this->viewModel) {
        $this->setViewModel(new ViewModel\ViewModel());
    }
    return $this->viewModel;
}
```

Le gestionnaire de vues injecte alors le chemin du layout à cette nouvelle vue :

Zend/Mvc/View/Http/ViewManager.php

```
public function getViewModel()
{
    [...]
    $model->setTemplate($this->getLayoutTemplate());
    [...]
}
```

Zend/Mvc/View/Http/ViewManager.php

```

public function getLayoutTemplate()
{
    $layout = 'layout/layout';
    if (isset($this->config['layout'])) {
        $layout = $this->config['layout'];
    }
    return $layout;
}

```

Le fichier de layout défini dans la configuration est alors injecté comme nom de template.

La vue courante est ensuite attachée au layout comme vue fille :

Zend/Mvc/View/Http/InjectViewModelListener.php

```

public function injectViewModel(MvcEvent $e)
{
    [...]
    $model->addChild($result);
}

```

Le système de rendu des différentes vues repose maintenant sur une imbrication des vues entre elles, et non sur des éléments de vues séparés comme dans la première version du framework. Le layout et la vue courante sont maintenant initialisés et imbriqués, le layout englobe la vue courante en tant que vue fille, tout est enfin prêt pour le rendu des vues.

Analysons maintenant la manière de désactiver le layout dans cette nouvelle gestion des vues. Nous venons de voir que la vue courante est injectée au layout depuis ces instructions :

Zend/Mvc/View/Http/InjectViewModelListener.php

```

public function injectViewModel(MvcEvent $e)
{
    $result = $e->getResult(); // représente la vue courante
    if (!$result instanceof ViewModel) {
        return;
    }
    $model = $e->getViewModel();
    if ($result->terminate()) {
        $e->setViewModel($result); // la vue remplace le layout
        return;
    }
    $model->addChild($result); // la vue est injectée au layout
}

```

Nous remarquons que cette injection se réalise uniquement si l'attribut « `$terminate` » de l'objet `ViewModel` est positionné sur la valeur « `false` », dans le cas contraire la vue courante remplace la vue principale, et la supprime. Il est donc possible de faire appel à la méthode « `setTerminate()` » depuis l'objet de vue courante afin de supprimer le layout :

Suppression du layout

```
public function noLayoutAction()
{
    $model = new ViewModel(array(
        'key' => 'value'
    ));
    $model->setTerminal(true);
    return $model;
}
```

Rendu des vues

Le rendu est opéré lors de l'évènement « render », lancé par l'objet Application à la suite de la distribution de l'action courante :

Zend/Mvc/Application.php

```
public function run()
{
    [...]
    $result = $events->trigger(MvcEvent::EVENT_DISPATCH, $event,
    $shortCircuit);
    [...]
    return $this->completeRequest($event);
}
```

Zend/Mvc/Application.php

```
protected function completeRequest(MvcEvent $event)
{
    $events = $this->getEventManager();
    $event->setTarget($this);
    $events->trigger(MvcEvent::EVENT_RENDER, $event);
    $events->trigger(MvcEvent::EVENT_FINISH, $event);
    return $event->getResponse();
}
```

Pour rappel, l'écouteur de cet évènement est la méthode « render() » de l'objet Zend\Mvc\View\Http\DefaultRenderingStrategy, attaché depuis le gestionnaire de vues :

Zend/Mvc/View/Http/ViewManager.php

```
public function onBootstrap($event)
{
    [...]
    $mvcRenderingStrategy = $this->getMvcRenderingStrategy();
    [...]
    $events->attach($mvcRenderingStrategy);
    [...]
}
```

Zend/Mvc/View/Http/ViewManager.php

```

public function getMvcRenderingStrategy()
{
[...]
$this->mvcRenderingStrategy = new DefaultRenderingStrategy($this->getView());
[...]
return $this->mvcRenderingStrategy ;
}

```

Zend/Mvc/View/Http/DefaultRenderingStrategy.php

```

public function attach(EventManagerInterface $events)
{
$this->listeners[] = $events->attach(MvcEvent::EVENT_RENDER,
array($this, 'render'), -10000);
}

```

L'écouteur délègue le rendu de la vue à l'objet Zend\View\View qui porte cette responsabilité :

Zend/Mvc/View/Http/DefaultRenderingStrategy.php

```

public function render(MvcEvent $e)
{
[...]
$view = $this->view;
$view->setRequest($request);
$view->setResponse($response);
$view->render($viewModel);
return $response;
}

```

Après avoir injecté les objets de réponse et de requête au sein de l'objet View, la méthode « render() » de cet objet est appelée afin de rendre l'objet de vue, contenant la vue courante imbriquée dans le layout.

La méthode « render() » de la classe View organise alors le travail de rendu :

Zend/View/View.php

```

public function render(Model $model)
{
$event = $this->getEvent();
$event->setModel($model);
$events = $this->getEventManager();
$results = $events->trigger(ViewEvent::EVENT_RENDERER, $event,
function($result) {
    return ($result instanceof Renderer);
});
$renderer = $results->last();
[...]
if ($model->hasChildren()
    && (!$renderer instanceof Renderer\TreeRendererInterface
        || !$renderer->canRenderTrees()))
) {
    $this->renderChildren($model);
}
$event->setModel($model);

```

```
    $event->setRenderer($renderer);

    $rendered = $renderer->render($model);
    $options = $model->getOptions();
    if (array_key_exists('has_parent', $options) && $options['has_parent']) {
        return $rendered;
    }

    $event->setResult($rendered);
    $events->trigger(ViewEvent::EVENT_RESPONSE, $event) ;
}
```

L'objet de type `ViewEvent`, qui est l'instance de l'évènement dans le composant de vue, est initialisé avec la requête et la réponse de notre application :

Zend/View/View.php

```
public function render(Model $model)
{
    $event = $this->getEvent() ;
    [...]
}
```

Zend/View/View.php

```
protected function getEvent()
{
    $event = new ViewEvent();
    $event->setTarget($this);
    if (null !== ($request = $this->getRequest())) {
        $event->setRequest($request);
    }
    if (null !== ($response = $this->getResponse())) {
        $event->setResponse($response);
    }
    return $event;
}
```

La méthode de rendu récupère ensuite l'objet capable de restituer la vue depuis la stratégie que l'on a définie, grâce au lancement de l'évènement «`renderer`» :

Zend/View/View.php

```
public function render(Model $model)
{
    [...]
    $events = $this->getEventManager();
    $results = $events->trigger(ViewEvent::EVENT_RENDERER, $event,
        function($result) {
            return ($result instanceof Renderer);
        });
    [...]
}
```

N'oublions pas que dans le Zend Framework 2, chacun des composants est dédié à une tâche. La classe `Zend\View\View` agit comme un point d'entrée capable de gérer les composants de rendu. Il pilote l'objet responsable du rendu et partage le résultat avec l'évènement courant.

Afin de récupérer l'objet qui va restituer la vue, la méthode lance l'évènement « `renderer` », écouté par les classes responsables des objets de rendu de vue. Les stratégies de rendu se trouvent dans l'espace de nom `Zend\View\Strategy`. L'écouteur notifié est l'objet `PhpRendererStrategy`, stratégie de rendu par défaut. Nous verrons plus tard qu'il est possible de la modifier.

La classe `PhpRendererStrategy` retourne l'objet de rendu qui lui est associé, de type `PhpRenderer` :

Zend/View/Strategy/PhpRendererStrategy.php

```
public function selectRenderer(ViewEvent $e)
{
    return $this->renderer;
}
```

Rappelons-nous que c'est le gestionnaire de vues qui attache les évènements de la classe `PhpRendererStrategy` lors de l'initialisation de la vue :

Zend/Mvc/View/Http/ViewManager.php

```
public function getView()
{
    [...]
    $this->view->getEventManager()->attach($this->getRendererStrategy());
    [...]
}
```

Zend/Mvc/View/Http/ViewManager.php

```
public function getRendererStrategy()
{
    [...]
    $this->rendererStrategy = new PhpRendererStrategy(
        $this->getRenderer()
    );
    [...]
}
```

La méthode orchestrant le rendu vérifie ensuite si la vue possède des vues filles, afin de les rendre suivant l'ordre d'imbrication pour les injecter dans le contenu des vues parentes :

Zend/View/View.php

```
public function render(Model $model)
{
    [...]
    if ($model->hasChildren()
        && (!$renderer instanceof Renderer\TreeRendererInterface
        || !$renderer->canRenderTrees()))
    ) {
        $this->renderChildren($model);
    }
    [...]
}
```

La méthode contrôle l'existence de vues filles et vérifie si la vue principale est capable de gérer le rendu de son arborescence elle-même. L'objet `PhpRenderer` ne peut pas gérer lui-même le rendu de son arbre de vues :

Zend/View/Renderer/PhpRenderer.php

```
private $__renderTrees = false;  
  
public function canRenderTrees()  
{  
    return $this->__renderTrees;  
}
```

La classe `JsonRenderer` par exemple, est capable de traiter elle-même le rendu de son arbre de vues au format JSON :

Zend/View/Renderer/JsonRenderer.php

```
public function canRenderTrees()  
{  
    return true;  
}
```

Chacune des vues filles est ensuite rendue individuellement en appelant cette même méthode récursivement en passant par la méthode « `renderChildren()` » :

Zend/View/View.php

```
protected function renderChildren(Model $model)  
{  
    foreach ($model as $child) {  
        [...]  
        $child->setOption('has_parent', true);  
        $result = $this->render($child);  
        $child->setOption('has_parent', null);  
        $capture = $child->captureTo();  
        if (!empty($capture)) {  
            $model->setVariable($capture, $result);  
        }  
    }  
}
```

Cette méthode injecte les vues enfants dans la variable de la vue parente correspondant à la valeur du retour de la méthode « `captureTo()` » de la vue enfant :

Zend/View/Model/ViewModel.php

```
class ViewModel implements ModelInterface  
{  
    protected $captureTo = 'content';  
    [...]  
}
```

Le contenu est injecté par défaut dans la variable « `content` », ce qui explique l'instruction de rendu dans la vue de layout :

Fichier de layout

```
|| <?php echo $this->content; ?>
```

L'objet de vue et de rendu sont ensuite enregistrés dans l'évènement de la vue pour laisser la possibilité à des écouteurs intervenant plus tard d'y avoir accès :

Zend/View/View.php

```
|| public function render(Model $model)
{
[...]
$event->setModel($model);
$event->setRenderer($renderer);
[...]
}
```

Notons que l'objet de vue est à nouveau enregistré au cas où il aurait changé entretemps.

L'objet de type PhpRenderer peut alors restituer la vue depuis sa méthode « `render()` » :

Zend/View/View.php

```
|| public function render(Model $model)
{
[...]
$rendered = $renderer->render($model);
[...]
}
```

Zend/View/Renderer/PhpRenderer.php

```
|| public function render($nameOrModel, $values = null)
{
if ($nameOrModel instanceof Model) {
    $model      = $nameOrModel;
    $nameOrModel = $model->getTemplate();
[...]
$options = $model->getOptions();
foreach ($options as $setting => $value) {
    $method = 'set' . $setting;
    if (method_exists($this, $method)) {
        $this->$method($value);
    }
    unset($method, $setting, $value);
}
unset($options);
$helper = $this->plugin('view_model');
$helper->setCurrent($model);
$values = $model->getVariables();
unset($model);
}

$this->addTemplate($nameOrModel);
unset($nameOrModel);
$this->__varsCache[] = $this->vars();
```

```

if (null !== $values) {
    $this->setVars($values);
}
unset($values);

$__vars = $this->vars()->getArrayCopy();
if (array_key_exists('this', $__vars)) {
    unset($__vars['this']);
}
extract($__vars);
unset($__vars);

while ($this->__template = array_pop($this->__templates)) {
    $this->__file = $this->resolver($this->__template);
    [...]
    ob_start();
    include $this->__file;
    $this->__content = ob_get_clean();
}
$this->setVars(array_pop($this->__varsCache));
return $this->getFilterChain()->filter($this->__content);
}

```

Dans notre exemple, le paramètre reçu par la méthode « `render()` » est un objet de type `ViewModel`. Le nom du fichier de vue est alors récupéré et la liste des options du modèle de vue parcourue, afin de pouvoir modifier les attributs de la classe :

Zend/View/Renderer/PhpRenderer.php

```

public function render($nameOrModel, $values = null)
{
    [...]
    $options = $model->getOptions();
    foreach ($options as $setting => $value) {
        $method = 'set' . $setting;
        if (method_exists($this, $method)) {
            $this->$method($value);
        }
        unset($method, $setting, $value);
    }
    unset($options);
    [...]
}

```

Les options de rendu peuvent être passées en argument à la vue lors de sa création, dans une de nos actions par exemple :

Exemple d'ajout d'options

```

public function indexAction()
{
    $filterChain = new \Zend\Filter\FilterChain();
    $filterChain->attachByName('StripTags');
    $filterChain->attachByName('StringToUpper');

    return new ViewModel(array('key'=>'value'), array('filterchain'=>$filterChain));
}

```

Dans cet exemple, la vue sera filtrée après le rendu avec la chaîne de filtres, et

supprimera donc tous les tags avant de passer tous les caractères en majuscules. Attention, comme les vues filles sont rendues avant la vue principale ce filtre sera aussi appliqué à la vue parente si celle-ci ne redéfinit pas ses propres options. Il est donc nécessaire de passer une nouvelle instance de la classe FilterChain au layout si l'on ne souhaite pas la suppression des tags sur la vue principale :

Exemple d'ajout d'options

```
public function indexAction()
{
    $event = $this->getEvent();
    $model = $event->getViewModel();
    $model->setOptions(array('filterchain'=>new \Zend\Filter\FilterChain()));

    $filterChain = new \Zend\Filter\FilterChain();
    $filterChain->attachByName('StripTags');
    $filterChain->attachByName('StringToUpper');

    return new ViewModel(array('key'=>'value'), array('filterchain'=>$filterChain));
}
```

La méthode de rendu se charge de peupler le plugin « view_model » par l'instance de vue courante et le nom du template de vue est ensuite enregistré dans un tableau interne qui sera défilé afin de rendre chacun de ses templates :

Zend/View/Renderer/PhpRenderer.php

```
public function render(Model $model)
{
    [...]
    $this->addTemplate($nameOrModel);
    [...]
}
```

Les variables de vue définies dans nos actions sont extraites comme variables locales :

Zend/View/Renderer/PhpRenderer.php

```
public function render($nameOrModel, $values = null)
{
    [...]
    $this->__varsCache[] = $this->vars();
    if (null !== $values) {
        $this->setVars($values);
    }
    unset($values);
    $__vars = $this->vars()->getArrayCopy();

    if (array_key_exists('this', $__vars)) {
        unset($__vars['this']);
    }
    extract($__vars);
    unset($__vars);
    [...]
}
```

L'extraction de ce tableau au sein de notre méthode permet d'accéder aux variables de vue comme des variables locales :

Accès depuis la vue

```
|| <?php echo $key; ?>
```

Ou comme un attribut grâce à la magie :

Zend/View/Renderer/PhpRenderer.php

```
|| public function __get($name)
  {
  $vars = $this->vars();
  return $vars[$name];
}
```

Accès depuis la vue

```
|| <?php echo $this->key; ?>
```

Ou comme un attribut grâce à la magie :

Zend/View/Renderer/PhpRenderer.php

```
|| public function __get($name)
  {
  $vars = $this->vars();
  return $vars[$name];
}
```

Accès depuis la vue

```
|| <?php echo $this->key; ?>
```

Les variables existantes de l'instance courante sont sauvegardées dans un tableau « `$_varsCache` », ce qui permettra de les restaurer par la suite.

La classe `PhpRenderer` s'apprête maintenant à rendre la vue. Pour cela, il doit localiser le template de vue afin d'en récupérer le contenu :

Zend/View/Renderer/PhpRenderer.php

```
|| public function render($nameOrModel, $values = null)
  {
  [...]
  while ($this->__template = array_pop($this->__templates)) {
    $this->__file = $this->resolver($this->__template);
    if (!$this->__file) {
      [...]
    }
    ob_start();
    include $this->__file;
    $this->__content = ob_get_clean();
  }
  [...]
}
```

Dans notre exemple, le tableau de templates contient seulement celui qui est associé à l'action courante dont il faut résoudre le chemin correspondant :

Zend/View/Renderer/PhpRenderer.php

```
public function render($nameOrModel, $values = null)
{
    [...]
    while ($this->__template = array_pop($this->__templates)) {
        $this->__file = $this->resolver($this->__template);
    }
}
```

La tâche de localisation de template est déléguée à la classe Zend\View\Resolver\AggregateResolver, spécialisée dans la résolution des noms de templates. Cette classe est initialisée dans le gestionnaire de vues et comporte deux objets de résolution de noms :

Zend/Mvc/View/Http/ViewManager.php

```
public function getResolver()
{
    if (null === $this->resolver) {
        $this->resolver = $this->services->get('ViewResolver');
    }
    return $this->resolver;
}
```

Zend/Mvc/Service/ViewResolverFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $resolver = new ViewResolver\AggregateResolver();
    $resolver->attach($serviceLocator->get('ViewTemplateMapResolver'));
    $resolver->attach($serviceLocator->get('ViewTemplatePathStack'));
    return $resolver;
}
```

La classe AggregateResolver liste les objets capables de résoudre les noms de template, et les parcourt jusqu'à trouver l'objet qui peut localiser le fichier souhaité.

Dans notre exemple, cette classe de résolution contient deux objets de type ResolverInterface : l'objet de type TemplateMapResolver qui associe un nom de fichier à un chemin et l'objet de type TemplatePathStack qui base sa recherche à partir d'un chemin de répertoire de vues.

Notons que chacun peut être configuré avec les clés « template_path_stack » et « template_map » depuis la configuration comme on le voit depuis leur fabrique :

Zend/Mvc/Service/ViewTemplatePathStackFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    if (is_array($config) && isset($config['view_manager'])) {
```

```

    $config = $config['view_manager'];
    if (is_array($config) && isset($config['template_path_stack'])) {
        $stack = $config['template_path_stack'];
    }
}
$templatePathStack = new ViewResolver\TemplatePathStack();
$templatePathStack->addPaths($stack);
return $templatePathStack;
}

```

Zend/Mvc/Service/ViewTemplateMapResolverFactory.php

```

public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $config = $serviceLocator->get('Config');
    $map = array();
    if (is_array($config) && isset($config['view_manager'])) {
        $config = $config['view_manager'];
        if (is_array($config) && isset($config['template_map'])) {
            $map = $config['template_map'];
        }
    }
    return new ViewResolver\TemplateMapResolver($map);
}

```

Analysons le fonctionnement du premier composant enregistré, la classe TemplateMapResolver et sa méthode de résolution de nom :

Zend/View/Resolver/TemplateMapResolver.php

```

public function resolve($name, Renderer $renderer = null)
{
    return $this->get($name);
}

```

Zend/View/Resolver/TemplateMapResolver.php

```

public function get($name)
{
    if (!$this->has($name)) {
        return false;
    }
    return $this->map[$name];
}

```

Ce composant de localisation de template se base sur un tableau qui associe un nom de fichier de vue à un chemin. Il suffit donc de vérifier l'existence de ce nom de template afin de pouvoir retourner le chemin du fichier correspondant.

Voici le tableau de correspondance construit dans notre configuration :

module.config.php

```

<?php
return array(
    'view_manager' => array(
        [...]
)

```

```

        'template_map' => array(
            'application/layout/layout' => __DIR__ . '/../
view/layout/layout.phtml',
            ),
            [...]
        ),
    );
);

```

Le layout est la seule vue enregistrée, le composant ne trouvera donc pas la vue correspondant à l'action.

Ce composant de résolution de nom, basé sur un tableau associatif, est l'objet de localisation le plus performant, ce qui explique l'importance qu'il soit le premier de la liste des classes de résolution.

Si cet objet ne sait pas résoudre le nom du template reçu, l'objet suivant sera notifié de la recherche. Examinons alors le comportement de la classe Zend\View\Resolver\TemplatePathStack :

Zend\View\Resolver\TemplatePathStack.php

```

public function resolve($name, Renderer $renderer = null)
{
    [...]
    $defaultSuffix = $this->getDefaultSuffix();
    if (pathinfo($name, PATHINFO_EXTENSION) != $defaultSuffix) {
        $name .= '.' . $defaultSuffix;
    }
    foreach ($this->paths as $path) {
        $file = new SplFileInfo($path . $name);
        if ($file->isReadable()) {
            if ((($filePath = $file->getRealPath()) === false &&
substr($path, 0, 7) === 'phar://')) {
                $filePath = $path . $name;
                if (!file_exists($filePath)) {
                    break;
                }
            }
            if ($this->useStreamWrapper()) {
                $filePath = 'zend.view://'. $filePath;
            }
            return $filePath;
        }
    }
    $this->lastLookupFailure = static::FAILURE_NOT_FOUND;
    return false;
}

```

Le fichier est cherché parmi la liste des chemins enregistrés depuis notre configuration :

module.config.php

```

<?php
return array(
    'view_manager' => array(
        [...]
        'template_path_stack' => array(

```

```

    'application' => __DIR__ . '/../view',
),
[...]
);

```

Nous avons vu précédemment que l'injection du template ajoute un nom de template sous la forme « module/contrôleur/action » au modèle de vue courant. Cependant, attention de ne pas croire que le fait d'indiquer « application » en clé du tableau de configuration « template_path_stack » nous permet de créer un automatisme au sein du framework. Le tableau « template_path_stack » se contente d'enregistrer les valeurs des chemins comme nous avons pu le voir plus haut. La valeur de la clé n'a aucune importance, si ce n'est pour ne pas interférer avec d'autres lors de la fusion des configurations entre modules. En effet, la méthode PHP « array_replace_recursive() » se contente de fusionner les tableaux en écrasant les clés identiques. Lors de la recherche d'un fichier de vue, la classe TemplatePathStack va parcourir l'ensemble des chemins sans tenir compte des clés indiquées.

Cependant, si vous souhaitez créer un automatisme, il vous suffit de faire correspondre le nom de vos « module/contrôleur/action » à la même hiérarchie de dossiers et de fichiers dans votre répertoire de vue. Si certains fichiers de vue ne respectent pas cette règle, indiquez-les manuellement dans l'objet de résolution TemplateMapResolver.

Lorsque le chemin est trouvé, celui est retourné à l'objet PhpRenderer par l'intermédiaire de l'aggrégateur de classes de localisation.

Afin de pouvoir restituer la vue, l'objet de rendu inclut la vue dans son code qu'il capture depuis les contrôleurs de buffers de sortie. Le fichier, exécuté par la fonction « include », est capturé et retourné directement dans la variable « \$__content ».

Zend/View/Renderer/PhpRenderer.php

```

public function render($nameOrModel, $values = null)
{
[...]
while ($this->__template = array_pop($this->__templates)) {
    $this->__file = $this->resolver($this->__template);
    [...]
    ob_start();
    include $this->__file;
    $this->__content = ob_get_clean();
}
[...]
}

```

Les variables initiales sont ensuite rétablies à l'original pour le prochain rendu et la vue est filtrée par l'objet de type FilterChain avant d'être rendue :

Zend/View/Renderer/PhpRenderer.php

```

public function render($nameOrModel, $values = null)
{
[...]
$this->setVars(array_pop($this->__varsCache));
return $this->getFilterChain()->filter($this->__content);
}

```

La vue est alors retournée à l'objet de vue View :

Zend/View/View.php

```
public function render(Model $model)
{
    [...]
    $rendered = $renderer->render($model);
    $options = $model->getOptions();
    if (array_key_exists('has_parent', $options) && $options['has_parent']) {
        return $rendered;
    }
    $event->setResult($rendered);
    $events->trigger(ViewEvent::EVENT_RESPONSE, $event);
}
```

Lorsque la vue est rendue, la méthode vérifie le type grâce à l'attribut « has_parent » des options de la vue traitée afin de s'assurer qu'elle correspond à la vue principale. Si ce n'est pas le cas, le traitement est interrompu car cette méthode est appelée récursivement pour chacune des vues filles. Le rendu est ensuite injecté dans le résultat de l'évènement de la vue.

La méthode « render() » lance enfin l'évènement « response » afin de notifier de la fin de son traitement et le besoin d'injection du rendu à l'objet de réponse. Cet évènement est écouté par la classe PhpRendererStrategy, attaché depuis le gestionnaire de vues :

Zend/View/Strategy/PhpRendererStrategy.php

```
public function attach(EventManagerInterface $events, $priority = 1)
{
    $this->listeners[] = $events->attach(ViewEvent::EVENT_RENDERER,
    array($this, 'selectRenderer'), $priority);

    $this->listeners[] = $events->attach(ViewEvent::EVENT_RESPONSE,
    array($this, 'injectResponse'), $priority);
}
```

La méthode « injectResponse() » de la classe PhpRendererStrategy est alors notifiée :

Zend/View/Strategy/PhpRendererStrategy.php

```
public function injectResponse(ViewEvent $e)
{
    $renderer = $e->getRenderer();
    if ($renderer !== $this->renderer) {
        return;
    }
    $result = $e->getResult();
    $response = $e->getResponse();

    if (empty($result)) {
        $placeholders = $renderer->plugin('placeholder');
        $registry = $placeholders->getRegistry();
        foreach ($this->contentPlaceholders as $placeholder) {
```

```
    if ($registry->containerExists($placeholder)) {
        $result = (string) $registry-
>getContainer($placeholder);
        break;
    }
}
$response->setContent($result);
}
```

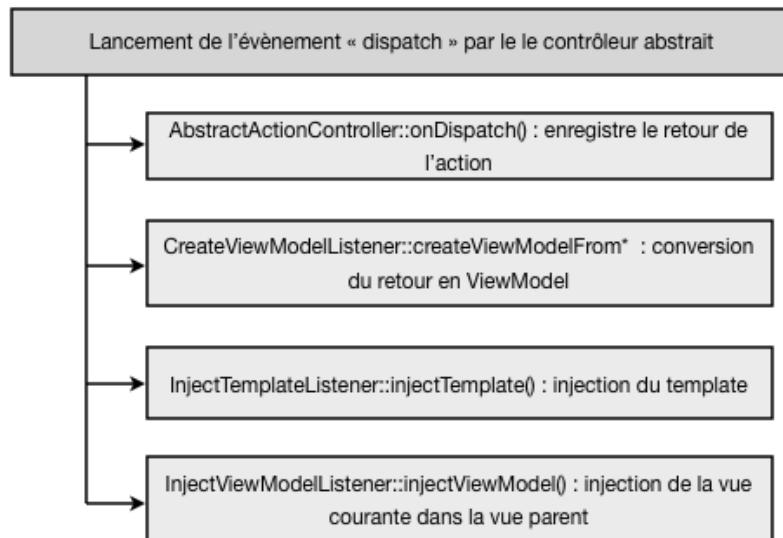
La méthode s'assure que l'objet de rendu utilisé correspond à sa stratégie de restitution des vues afin de pouvoir traiter l'évènement. C'est une précaution à prendre si l'on utilise une stratégie de rendu particulière car la classe `PhpRendererStrategy` étant ajoutée par défaut, elle sera systématiquement notifiée de l'évènement. Le résultat de l'évènement peut ensuite être injecté dans l'objet de réponse.

Une vérification est tout de même effectuée avant l'injection du résultat afin de vérifier que celui-ci n'est pas une chaîne vide. Dans le cas contraire, les principaux conteneurs de données sont utilisés afin d'extraire leur contenu.

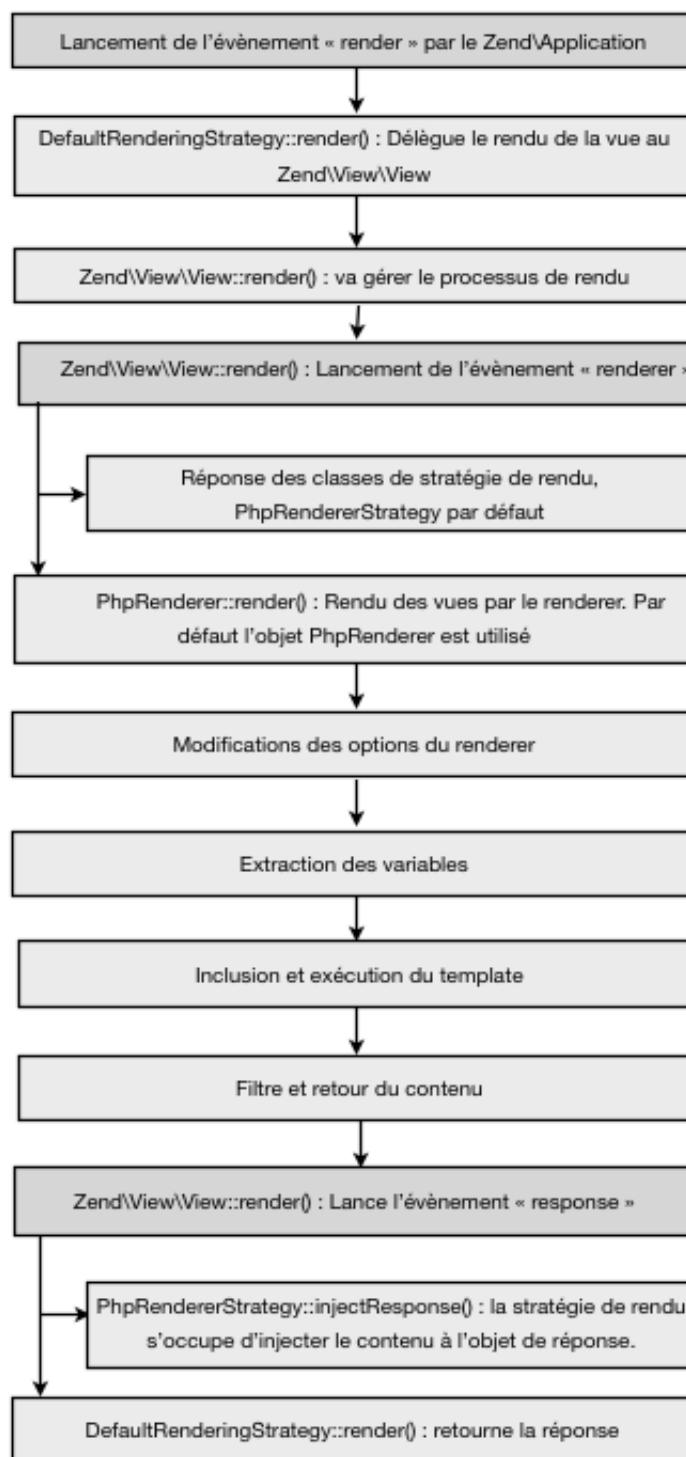
Récapitulatif du rendu

Nous venons de décrire la chaîne de traitement pour le rendu des vues, ce qui nous permet de bien assimiler le nouveau système de gestion de vues du Zend Framework 2. Les explications et le code passé en revue étant assez dense, et les sauts de méthode en méthode un peu déroutants, voici un schéma récapitulatif du processus de rendu :

Préparation des vues :



Rendu des vues :



Une fois la vue rendue, un dernier écouteur entre en jeu afin d'envoyer la réponse, qui contient la vue, au client. Cet écouteur, qui agit sur l'évènement « finish » est le Zend\ Mvc\ View\ SendResponseListener que l'on a présenté rapidement au début de chapitre :

Zend/Mvc/View/SendResponseListener.php

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(MvcEvent::EVENT_FINISH,
    array($this, 'sendResponse'), -10000);
}
```

Zend/Mvc/View/SendResponseListener.php

```
public function sendResponse(MvcEvent $e)
{
    $response = $e->getResponse();
    if (!$response instanceof Response) {
        return false;
    }
    if (is_callable(array($response, 'send'))) {
        return $response->send();
    }
}
```

Cet écouteur possède l'avantage de retourner automatiquement la vue au client. L'écouteur s'attache avec une priorité faible sur l'évènement, il suffit de le court-circuiter sur une priorité plus haute afin de ne pas envoyer automatiquement la réponse au client.

Types de vue

Comme nous venons de le voir, la stratégie de rendu par défaut est gérée par l'objet de type PhpRendererStrategy qui utilise la classe PhpRenderer afin de restituer la vue. Cependant, il existe d'autres stratégies de rendu, par exemple celle qui permet de retourner une vue au format JSON. Voici un exemple afin d'illustrer l'utilisation de cette stratégie de rendu :

Rendu de JSON

```
public function jsonAction()
{
    $locator = $this->getServiceLocator();
    $jsonRendererStrategy = $locator->get('ViewJsonStrategy');
    $view = $locator->get('View');
    $view->getEventManager()->attachAggregate($jsonRendererStrategy,
    1000);

    return $model = new JsonModel(array(
        'key' => 'value'
    ));
}
```

La stratégie de rendu correspondant au format JSON et l'objet de vue sont récu-

pérés afin de pouvoir attacher les écouteurs de notre nouvelle stratégie avec une priorité plus importante que celle par défaut. Cela nous permet de court-circuiter la classe `PhpRendererStrategy`, stratégie de rendu par défaut.

Nous devrons maintenant retourner un objet de type `JsonModel` dans nos actions si l'on souhaite utiliser la stratégie de rendu JSON. Analysons l'écouteur de sélection de l'objet de rendu de la classe `JsonStrategy` :

Zend/View/Strategy/JsonStrategy.php

```
public function selectRenderer(ViewEvent $e)
{
    $model = $e->getModel();
    if ($model instanceof Model\JsonModel) {
        return $this->renderer;
    }
    [...]
    $accept = $headers->get('Accept');

    if ((($match = $accept->match('application/json, application/
    javascript')) == false) {
        return;
    }
    if ($match->getTypeString() == 'application/json') {
        return $this->renderer;
    }
    if ($match->getTypeString() == 'application/javascript') {
        if (false != ($callback = $request->getQuery()->get('callback'))) {
            $this->renderer->setJsonpCallback($callback);
        }
        return $this->renderer;
    }
}
```

L'objet de rendu de JSON n'est utilisé que si le modèle en cours est de type `JsonModel`. Notons que la classe de stratégie de rendu vérifie les headers afin de valider l'objet de rendu si l'objet de vue n'est pas de type `ViewModel`. Une autre possibilité nous permet donc de rendre une vue au format JSON tout en conservant un objet de type `ViewModel` :

Rendu de JSON avec un objet de type `ViewModel`

```
public function jsonAction()
{
    $locator = $this->getServiceLocator();
    $jsonRendererStrategy = $locator->get('ViewJsonStrategy');
    $view = $locator->get('View');
    $view->getEventManager()->attachAggregate($jsonRendererStrategy,
    1000);

    $this->getRequest()->getHeaders()->get('accept')-
    >addMediaType('application/json') ;

    return new \Zend\View\Model\ViewModel(array(
        'key' => 'value',
        'state'=> 'ok'
    ));
}
```

En modifiant les headers de la réponse, nous obtenons un résultat JSON, mais au contenu légèrement différent :

Rendu de la vue de type ViewModel

```
|| {"content":{"key":"value","state":"ok"}}
```

Voici le rendu avec l'utilisation de l'objet JsonModel :

Rendu de la vue de type JsonModel

```
|| {"key":"value","state":"ok"}
```

Cette différence vient du fait que la méthode de rendu du JsonRenderer ne restitue pas une vue de type JsonModel de la même manière que les autres :

Zend/View/Renderer/JsonRenderer.php

```
public function render($nameOrModel, $values = null)
{
    if ($nameOrModel instanceof Model) {
        if ($nameOrModel instanceof Model\JsonModel) {
            $values = $nameOrModel->serialize();
        } else {
            $values = $this->recurseModel($nameOrModel);
            $values = Json::encode($values);
        }
        return $values;
    }
    [...]
}
```

Cependant, il y a peu d'intérêt à changer les headers depuis une action afin d'en récupérer du JSON. Cette fonctionnalité a été créée afin de pouvoir récupérer un format JSON depuis une API ou un script utilisant une commande telle que :

Forcer l'utilisation d'une vue JSON

```
|| curl -H «Accept: application/json» http://www.monsite.com/monurl
```

Nous savons maintenant parfaitement utiliser les différentes vues et stratégies de rendu.

Manipulation des vues

Lors du développement d'applications Web, il est souvent nécessaire de désactiver la vue ou le layout, ou bien de changer le template de vue au cours de l'action. Maintenant que nous comprenons la logique interne, voici comment les manipuler.

- Désactiver le layout

Comme nous l'avons vu précédemment, il est possible de désactiver l'injection du modèle de vue dans celui du layout depuis l'attribut « terminate » de la vue cou-

rante. Il est donc possible de désactiver le layout depuis les instructions :

Désactivation du layout

```
public function monAction()
{
    $viewModel = new ViewModel();
    $viewModel->setTerminal(true);

    return $viewModel;
}
```

- Désactiver la vue

Nous avons vu que la vue peut être construite depuis un tableau ou le retour « null », afin de fournir un objet de type Zend\View\Model à l'écouteur qui s'occupe de l'injection de template. Nous pouvons donc désactiver la vue courante depuis les instructions :

Désactiver la vue

```
public function monAction()
{
    return false;
}
```

- Modifier le template du layout

Le gestionnaire d'évènements du contrôleur abstrait AbstractActionController définit comme identifiant l'espace de nom du module courant :

Zend/Mvc/Controller/AbstractActionController.php

```
public function setEventManager(EventManagerInterface $events)
{
    $events->setIdentifiers(array(
        'Zend\Stdlib\DispatchableInterface',
        __CLASS__,
        get_class($this),
        substr(get_class($this), 0, strpos(get_class($this), '\\'))
    ));
    $this->events = $events;
    $this->attachDefaultListeners();
    return $this;
}
```

Nous pouvons donc attacher des évènements sur le nom de notre module afin de changer le template du layout pour les contrôleurs de ce module :

Modification du template pour un module

```
class Module implements AutoloaderProvider
{
    public function init(ModuleManager $moduleManager)
    {
        $sharedEvents = $moduleManager->getEventManager()->getSharedManager();
```

```
    $sharedEvents->attach('MonModule', MvcEvent::EVENT_DISPATCH,
function($e) {
    $controller = $e->getTarget();
    $controller->plugin('layout')->setTemplate('layout/
application');
    }, 100);
}
[...]
}
```

- Modifier le template de vue

Modification du template de vue

```
public function monAction()
{
$viewModel = new ViewModel();
$viewModel->setTemplate('index/other');
return $viewModel;
}
```

La modification du template se fait facilement depuis la méthode « `setTemplate()` » qui lui est consacrée.

- Désactiver la vue et le layout

L'objet Application distribue la requête courante afin de pouvoir construire un objet de réponse avec le résultat de celle-ci. Cependant, si un objet de réponse est reçu par la classe Application, celui-ci n'aura pas à le construire et pourra alors le retourner directement. La vue courante et le layout sont donc désactivables depuis le retour de l'objet de réponse courante vide ou depuis un nouvel objet de réponse vide :

Désactiver la vue et le layout depuis un objet de réponse vide

```
public function monAction()
{
return new \Zend\Http\PhpEnvironment\Response();
}
```

Il est également possible de retourner l'objet de réponse courant qui est vide lors de la distribution :

Désactiver la vue et le layout

```
public function monAction()
{
return $this->response;
}
```


13

Les aides de vue

Les aides de vue offrent des fonctionnalités supplémentaires lors du rendu de l'affichage des données, et permettent de factoriser du code que l'on écrit de manière répétitive. Nous passerons en revue quelques aides de vue présentes dans le framework, la plupart d'entre elles existent déjà dans la version précédente du framework.

Analysons tout de suite les appels aux aides de vue depuis un template afin de comprendre le chemin effectué par chacun des appels. Voici comment utiliser une aide de vue avec l'aide Partial :

Utilisation des aides de vue

```
|| echo $this->partial(...);
```

L'appel pour des aides de vue liées à un sous-ensemble, comme la navigation, se fait comme ceci :

Utilisation des aides de vue

```
|| echo $this->navigation()->breadcrumbs()->render();
```

Rappelons-nous que lorsque notre objet de rendu, de type PhpRenderer par défaut, souhaite obtenir le rendu de la vue courante, celui-ci inclut le fichier de vue directement au sein de sa méthode :

Zend/View/Renderer/PhpRenderer.php

```
|| public function render($nameOrModel, $values = null)
  {
  [...]
  ob_start();
  include $this->_file;
  $this->__content = ob_get_clean();
  [...]
  }
```

Lorsque l'aide de vue est appelée depuis le fichier de vue, celle-ci est donc appelée au sein de l'objet PhpRenderer, celui-ci venant d'inclure le code de la vue dans sa méthode de rendu. Seulement, notre gestionnaire de rendus ne dispose pas de

méthodes correspondant aux noms des aides de vue dans sa classe, celles-ci vont donc être prises en charge par la méthode magique « `__call()` » :

Zend/View/Renderer/PhpRenderer.php

```
|| public function __call($method, $argv)
|| {
||     $helper = $this->plugin($method);
||     if (is_callable($helper)) {
||         return call_user_func_array($helper, $argv);
||     }
||     return $helper;
|| }
```

La méthode magique récupère le plugin demandé grâce à la méthode « `plugin()` » et appelle ensuite l'objet d'aide de vue comme une fonction de callback, ce qui déclenche alors l'appel de la méthode « `__invoke()` » de l'objet. Il est donc possible de récupérer et utiliser une aide de vue en supprimant l'étape de la magie lors de la récupération du plugin :

Suppression de la magie lors de l'appel

```
|| <?php
|| $plugin = $this->plugin('partial');
|| echo $plugin('template.phtml');
|| ?>
```

Le premier plugin que l'on va étudier est généralement peu utilisé et assez complexe à prendre en main lors de la première approche, il s'agit du fil d'Ariane.

Le fil d'Ariane

L'aide de vue du fil d'Ariane repose sur les notions de conteneur, en particulier sur le composant Zend\Navigation\Navigation. Ce composant hérite de l'objet Zend\Navigation\AbstractContainer :

Zend/Navigation/Navigation.php

```
|| class Navigation extends AbstractContainer
|| {
||     public function __construct($pages = null)
||     {
||         [...]
||         if ($pages) {
||             $this->addPages($pages);
||         }
||     }
|| }
```

L'objet Zend\Navigation\AbstractContainer représente un conteneur d'objet, qui lui-même peut contenir des sous-objets de type Zend\Navigation\Page\AbstractPage qui étendent la classe AbstractContainer, ce qui explique la possibilité d'avoir un niveau d'encapsulation infini. Nous obtenons donc des conteneurs imbriqués les uns dans les autres afin de former une hiérarchie.

Pour créer cette hiérarchie, une fabrique existe et permet de construire le conteneur suivant les paramètres inscrits en configuration. Cette fabrique est la classe DefaultNavigationFactory qui étend AbstractNavigationFactory :

Zend/Navigation/Service/AbstractNavigationFactory.php

```
public function createService(ServiceLocatorInterface $serviceLocator)
{
    $pages = $this->getPages($serviceLocator);
    return new Navigation($pages);
}
```

La fabrique crée les pages avant de retourner le conteneur. Voici la création des pages :

Zend/Navigation/Service/AbstractNavigationFactory.php

```
protected function getPages(ServiceLocatorInterface $serviceLocator)
{
    if (null === $this->pages) {
        $configuration = $serviceLocator->get('Configuration');
        if (!isset($configuration['navigation'])) {
            [...]
        }
        if (!isset($configuration['navigation'][$this->getName()])) {
            [...]
        }
        $application = $serviceLocator->get('Application');
        $routeMatch = $application->getMvcEvent()->getRouteMatch();
        $router     = $application->getMvcEvent()->getRouter();
        $pages      = $this->getPagesFromConfig($configuration['navigation'][$this->getName()]);
        $this->pages = $this->injectComponents($pages, $routeMatch,
        $router);
    }
    return $this->pages;
}
```

La fabrique récupère la configuration de l'application ainsi que l'objet contenant les paramètres de la route et le router pour les injecter aux différentes pages qui les utilisent pour construire les routes. Une fois les pages construites et les injections faites, le composant peut alors être construit et renvoyé. Nous remarquons également que les pages sont recherchées sous la clé « navigation » et la sous-clé correspondant au nom de la fabrique, ici « default » :

Zend/Navigation/Service/DefaultNavigationFactory.php

```
protected function getName()
{
    return 'default';
}
```

Voici la configuration que nous pouvons créer :

module.config.php

```

<?php
return array(
    [...]
    'navigation' => array(
        'default' => array(
            'home' => array('type' => 'mvc', 'route' =>
'home','active'=>true, 'label' => 'Home'),
            [...]
            'contact' => array('type' => 'mvc', 'route' =>
'contact','active'=>false, 'label' => 'Contact'),
        ),
    ),
);

```

N'oublions pas d'ajouter la fabrique DefaultNavigationFactory au gestionnaire de services car celle-ci n'est pas inscrite par défaut :

module.config.php

```

<?php
return array(
    'service_manager' => array(
        'factories' => array(
            'DefaultNavigation' => 'Zend\Navigation\Service\
DefaultNavigationFactory',
            [...]
        ),
    ),
);

```

Nos pages sont créées, et il est possible d'en ajouter d'autres dynamiquement :

module.config.php

```

public function categoriesAction()
{
    $navigation = $this->getServiceLocator()->get('DefaultNavigation');

    $navigation->current()->addPage(
        array('type'=>'mvc','active'=>true,'route'=>'category','label'=>
'Catégories')
    );
}

```

Notons qu'il existe un autre type de page, le Zend\Navigation\Page\Uri qui permet de spécifier directement une URL lors de la création de la page :

Ajout d'une page de type « URI »

```

public function categoriesAction()
{
    $navigation = $this->getServiceLocator()->get('DefaultNavigation');
    $navigation->current()->addPage(array(
        'type'=>'uri','active'=>true,'uri'=>'/
mapage','label'=>'Catégories')
)
}

```

```
    } );
```

La page est ajoutée avec l'intitulé « Catégories ». Une fois les pages ajoutées, l'aide de vue utilisera la page qui est marquée comme « active » pour déterminer quelles sont les pages à afficher. Il n'est cependant pas nécessaire de forcer le paramètre « active » pour bénéficier de ce comportement, les pages de type Mvc fournissent une méthode capable de déterminer automatiquement si celles-ci sont actives ou non depuis l'objet contenant les paramètres de route :

Zend/Navigation/Page/Mvc.php

```
public function isActive($recursive = false)
{
    if (!$this->active) {
        [...]
        if ($this->routeMatch instanceof RouteMatch) {
            [...]
            if (null !== $this->getRoute()
                && $this->routeMatch->getMatchedRouteName() ===
            $this->getRoute()
                && (count(array_intersect_assoc($reqParams,
            $myParams)) == count($myParams))
            ) {
                $this->active = true;
                return true;
            }
        }
    }
    return parent::isActive($recursive);
}
```

Nous pouvons maintenant afficher le fil d'Ariane dans nos vues depuis ces instructions :

Affichage du fil d'Ariane

```
<?php
echo $this->navigation()->breadcrumbs('DefaultNavigation')-
>setMinDepth(0)->setLinkLast(true)->render();
?>
```

Afin de pouvoir utiliser l'aide Breadcrumbs, il est nécessaire de lui fournir un nom ou objet de conteneur. Si le paramètre est une chaîne de caractères, alors le gestionnaire de services tente de récupérer le service qui lui correspond :

Zend/View/Helper/Navigation/AbstractHelper.php

```
public function setContainer($container = null)
{
    $this->parseContainer($container);
    $this->container = $container;

    return $this;
}
```

Zend/View/Helper/Navigation/AbstractHelper.php

```

protected function parseContainer(&$container = null)
{
[...]
if (is_string($container)) {
    [...]
    $container = $s1->get($container);
}
[...]
}

```

Le passage du paramètre à l'aide de vue implique la modification du conteneur depuis la méthode « `setContainer()` » qui gère le type de paramètre reçu. Dans notre cas, nous nous rappelons que le service « `DefaultNavigation` » correspond à la fabrique `DefaultNavigationFactory` qui renvoie un objet de navigation.

L'aide de vue `Navigation` est chargée de récupérer l'aide de type `Navigation\Breadcrumbs`. Ensuite, la première méthode, « `setMinDepth()` », s'assure que l'arborescence affichée possède un niveau de profondeur supérieur à zéro, c'est-à-dire s'il y a au moins une page dans notre conteneur principal. La méthode « `setLinkLast()` » permet d'indiquer si nous souhaitons afficher un lien sur le dernier élément du fil d'Ariane. Notre vue rendra alors :

Rendu du fil d'Ariane

```
|| Accueil > Catégories
```

Si nous ne souhaitons pas de lien sur le dernier élément nous devrons passer le paramètre de la méthode « `setLinkLast()` » à la valeur « `false` » :

Suppression du lien sur le dernier élément

```

<?php
echo $this->navigation()->breadcrumbs('DefaultNavigation')-
>setMinDepth(0)->setLinkLast(false)->render();
?>

```

Le rendu sera alors :

Affichage du fil d'Ariane

```
|| Accueil > Catégories
```

La méthode « `setMinDepth()` » définit une condition d'affichage en imposant un niveau de profondeur minimum, ce qui donne la possibilité de gérer la pertinence de l'affichage du fil d'Ariane en fonction de la profondeur de la navigation. Dans l'exemple précédent, nous avions deux éléments dans le fil d'Ariane, modifions maintenant la valeur du niveau de profondeur minimum requis :

Modification du niveau de profondeur minimum

```

<?php echo $this->navigation()->breadcrumbs('DefaultNavigation')-
>setMinDepth(2)->setLinkLast(false)->render();
?>

```

Le fil d'Ariane ne s'affiche alors qu'à partir d'une hiérarchie de trois éléments minimum, l'aide de vue retournera donc une chaîne vide.

Analysons maintenant l'utilité de passer par l'aide de vue Navigation pour récupérer celle correspondant au Breadcrumbs. L'aide de vue Navigation dispose de son propre gestionnaire de plugins afin de charger les sous-aides de vue qui lui correspondent :

Zend/View/Helper/Navigation/PluginManager.php

```
class PluginManager extends HelperPluginManager
{
protected $invokableClasses = array(
    'breadcrumbs' => 'Zend\View\Helper\Navigation\Breadcrumbs',
    'links'         => 'Zend\View\Helper\Navigation\Links',
    'menu'          => 'Zend\View\Helper\Navigation\Menu',
    'sitemap'       => 'Zend\View\Helper\Navigation\Sitemap',
);
}
```

En appelant la méthode « `breadcrumbs()` » sur l'objet `Navigation`, notre appel est redirigé sur la méthode magique « `__call` » de l'aide de navigation :

Zend/View/Helper/Navigation.php

```
public function __call($method, array $arguments = array())
{
$helper = $this->findHelper($method, false);
if ($helper) {
    [...]
    return call_user_func_array($helper, $arguments);
}
[...]
}
```

Zend/View/Helper/Navigation.php

```
public function findHelper($proxy, $strict = true)
{
$plugins = $this->getPluginManager();
if (!$plugins->has($proxy)) {
    [...]
}
$helper = $plugins->get($proxy);
[...]
return $helper;
}
```

Une autre aide de vue dépendant de la navigation peut être intéressante à utiliser en complémentarité du fil d'Ariane, il s'agit de l'aide liée à la génération du sitemap.

Le sitemap

Comme nous l'avons vu précédemment, l'aide de vue Sitemap fait partie des aides de vue dépendant de celle de la navigation, et utilise par défaut notre conteneur `Navigation` s'il est présent dans le registre. Reprenons l'exemple précédent, et ajou-

tons une action sitemap dans notre contrôleur :

Utilisation de l'aide de vue sitemap

```
public function sitemapAction()
{
    $this->getResponse()->headers()->addHeaderLine('Content-Type:
application/xml');

    $viewModel = new ViewModel();
    $viewModel->setTerminal(true);
    return $viewModel;
}
```

Les headers de l'application sont modifiés afin d'indiquer le type du contenu de retour et le layout désactivé. Voici le code de notre vue :

Utilisation de l'aide de vue du sitemap

```
<?php
echo $this->navigation()->sitemap('DefaultNavigation')->render();
?>
```

Le rendu sera donc le suivant :

Rendu du sitemap

```
<urlset>
    <url>
        <loc>http://monsite.com/</loc>
    </url>
    [...]
    <url>
        <loc>http://monsite.com/contact</loc>
    </url>
</urlset>
```

Le sitemap retourne alors l'ensemble des pages que nous avons inscrites dans la configuration au chapitre précédent.

Plusieurs options s'offrent alors à nous. Soit nous créons toutes les pages de notre application depuis la configuration ou soit nous ajoutons toutes les pages uniquement lors de l'action sitemap afin de réaliser cette opération uniquement sur demande. Si nous choisissons la première option, celle-ci ne va créer aucun effet de bord sur le fil d'Ariane, car cette aide de vue se base sur la valeur de l'attribut « \$active » des différentes pages. Pour des raisons de performances, nous pouvons préférer l'initialisation de nos pages dans la méthode « sitemap() » afin de ne pas charger un grand nombre de pages non utilisées :

Initialisation du conteneur de navigation

```
public function sitemapAction()
{
    $this->getResponse()->headers()->addHeaderLine('Content-Type:
application/xml');

    $navigation = $this->getServiceLocator()->get('DefaultNavigation');
```

```

    $home = $navigation->current();
    $home->addPage(array('type'=>'mvc', 'route'=>'blog'));
    $home->addPage(array('type'=>'mvc', 'route'=>'contact'));

    $viewModel = new ViewModel();
    $viewModel->setTerminal(true);

    return $viewModel;
}

```

Les attributs « \$active » et « \$label » des pages n'ont pas d'intérêt ici car ils ne seront pas utilisés.

Pour les sites volumineux, la création du sitemap à la volée depuis l'aide de vue ne conviendra certainement pas et il reste préférable de les générer depuis des tâches automatisées ou de privilégier l'utilisation d'un cache.

L'aide Layout et ViewModel

Certaines aides de vue permettent aussi d'avoir la main sur les instances de vue créées dans notre application, comme le layout ou le modèle courant. L'aide de type Layout donne l'accès à la vue correspondant au layout, ce qui nous permet la récupération ou la modification des variables du layout depuis la vue courante. La vue courante est rendue par l'objet de type Renderer avant le layout. Il est donc possible d'accéder à la vue parente depuis la vue liée à l'action courante :

Utilisation du layout depuis la vue courante

```

<?php
$this->layout()->setVariable('test', 'passage depuis la vue');
?>

```

La variable « test » devient alors une nouvelle variable du layout. Notons que nous pourrions avoir le même résultat en passant par l'aide de vue ViewModel qui permet d'avoir accès aux instances de vues, mère et fille :

Utilisation du layout depuis la vue courante

```

<?php
$this->viewModel()->getRoot()->setVariable('test', 'passage depuis la
vue');
?>

```

Si nous analysons le code de l'aide de vue Layout, nous comprenons que notre premier exemple exécute les mêmes instructions que lors du deuxième exemple :

Zend/View/Helper/Layout.php

```

class Layout extends AbstractHelper
{
[...]
public function __invoke($template = null)
{
    if (null === $template) {
        return $this->getRoot();
}

```

```

        }
        return $this->setTemplate($template);
    }

    protected function getRoot()
    {
        $helper = $this->getViewModelHelper();
        if (!$helper->hasRoot()) {
            [...]
        }
        return $helper->getRoot();
    }
    [...]
}

```

La méthode « `__invoke` » fait appel à la méthode « `getRoot()` » qui utilise l'aide de vue `ViewModel` afin d'avoir accès à la vue principale.

Une méthode permet aussi de modifier le fichier de la vue utilisée par le layout depuis la vue courante :

Modification du template de layout

```

<?php
$this->layout()->setTemplate('layout/layout_2.phtml');
?>

```

Ce comportement est possible, car comme nous l'avons vu, la vue courante est rendue avant le template correspondant au layout.

L'aide Partial

L'aide de vue `Partial` permet de rendre d'autres fichiers de vue, depuis la vue courante. L'aide `Partial` rend un fichier dont le nom est passé en paramètre, la vue rendue n'aura accès qu'aux variables fournies en tableau du deuxième paramètre :

Test de l'aide `Partial`

```

<?php
echo $this->partial('index/bloc.phtml',array('param' => 'value'));
?>

```

Il est également possible de passer un objet de type `ArrayObject` ou définissant une méthode « `toArray()` » à l'objet en second paramètre. En effet, l'aide de vue accepte en paramètre un tableau ou un objet en récupérant ses variables soit depuis une méthode « `toArray()` », soit par introspection en utilisant la fonction « `get_object_vars()` » :

Zend/View/Helper/Partial.php

```

public function __invoke($name = null, $model = null)
{
    [...]
    if (!empty($model)) {
        if (is_array($model)) {

```

```
    $view->vars()->assign($model);
} elseif (is_object($model)) {
    if (null !== ($objectKey = $this->getObjectContext())) {
        $view->vars()->offsetSet($objectKey, $model);
    } elseif (method_exists($model, 'toArray')) {
        $view->vars()->assign($model->toArray());
    } else {
        $view->vars()->assign(get_object_vars($model));
    }
}
return $view->render($name);
}
```

Lorsque l'on analyse le fonctionnement de l'aide de vue Partial, nous pouvons remarquer que le gestionnaire de rendus de vue courant est cloné. En effet, c'est bien l'objet PhpRenderer qui est cloné, et non l'objet ViewModel. L'objet ViewModel ne représente maintenant qu'un conteneur, c'est l'objet de rendu qui gère les variables et aides de vue. Voici les instructions de clonage :

Zend/View/Helper/Partial.php

```
public function __invoke($name = null, $model = null)
{
[...]
$view = $this->cloneView();
[...]
}
```

Zend/View/Helper/Partial.php

```
public function cloneView()
{
    $view = clone $this->view ;
    $view->setVars(array());
    return $view;
}
```

L'objet de rendu est ensuite vidé de ses variables, ce qui permet de ne pas interférer avec les variables de la vue courante. La vue peut ensuite être rendue.

Cette manière de rendre une vue décorrélée du contexte en cours peut s'avérer pratique, car le développeur a la maîtrise des variables passées en paramètre. Cependant, ce processus fera légèrement baisser les performances, car l'objet de rendu doit être cloné, initialisé et repeuplé.

Une deuxième possibilité de rendu est disponible en utilisant directement la fonction « render() » de l'objet de rendu. Il est alors nécessaire de faire attention à la collision des noms de variables, celles définies dans la vue courante sont accessibles depuis le nouveau fichier de vue.

Le rendu avec l'aide de vue Partial :

Rendu depuis l'aide Partial

```
||| <?php
||| echo $this->partial('index/bloc.phtml',array(
|||   'param' => 'value', 'param2' => 'value2'
||| ));
||| ?>
```

Ce même rendu depuis la méthode « render() » :

Rendu depuis la méthode « render() »

```
||| <?php
||| $vars = $this->vars();
||| $vars['param'] = 'value';
||| $vars['param2'] = 'value2';
||| echo $this->render('index/bloc.phtml');
||| ?>
```

Le rendu avec l'utilisation de la magie est parfois plus pratique :

Rendu depuis la méthode « render() »

```
||| <?php
||| $this->param = 'value';
||| $this->param2 = 'value2';
||| echo $this->render('index/bloc.phtml');
||| ?>
```

Le rendu peut aussi se faire en utilisant les méthodes de l'objet de rendu :

Rendu depuis la méthode « render() »

```
||| <?php
||| $this->declarevars(array('param'=>'value'),array('param2'=>'val
||| ue2'));
||| echo $this->render('index/bloc.phtml');
||| ?>
```

L'utilisation que l'on fera de l'aide de vue Partial ou de l'utilisation de la méthode « render() » dépend du contexte de l'application. Il est cependant préférable d'utiliser la méthode « render() » lorsque le contexte nous le permet, celle-ci sera plus performante que l'aide de vue Partial.

Le rendu d'élément HTML

Dans la liste des éléments HTML générés, les aides de vue proposent le rendu d'éléments comme les objets Flash ou QuickTime. Voici un exemple de rendu d'objet Flash :

Rendu d'un élément de type Flash

```
<?php
echo $this->htmlFlash('monflash.swf', array('width' => 800, 'height'
=> 400), array('transparent' => 'wmode'));
?>
```

Et voici un exemple de rendu d'objet QuickTime :

Rendu d'un élément de type QuickTime

```
<?php
echo $this->htmlQuicktime('monfilm.mov');
?>
```

Le rendu d'élément HTML est simple à mettre en œuvre, mais il est peut-être parfois plus pratique que ces éléments soient intégrés directement par les équipes de développeurs frontend, c'est pourquoi ces aides sont peu utilisées.

Le rendu de JSON

Le framework fournit une aide de vue afin de générer du format JSON. Cette aide de vue peut être utilisée facilement pour écrire des blocs de code JSON dans nos vues HTML ou afin de rendre une vue entière dans ce format. L'aide de vue JSON permet de modifier automatiquement les headers de notre objet de réponse :

Zend/View/Helper/Json.php

```
public function __invoke($data, array $jsonOptions = array())
{
    $data = JsonFormatter::encode($data, null, $jsonOptions);
    if ($this->response instanceof Response) {
        $headers = $this->response->headers();
        $headers->addHeaderLine('Content-Type', 'application/json');
    }
    return $data;
}
```

L'initialisation de l'aide de vue JSON se fait simplement avec l'objet de réponse de l'application au sein de la classe Module et l'initialisation de la vue :

Module.php

```
public function initializeView($e)
{
    $app        = $e->getTarget();
    $locator    = $app->getServiceManager();
    $renderer   = $locator->get('ViewManager')->getRenderer();
    $renderer->plugin('json')->setResponse($app->getResponse());
}
```

Une fois l'objet de réponse passé à l'aide de vue, celle-ci peut alors modifier les headers de la réponse afin d'indiquer que le rendu de notre application sera au format JSON. Voici le code de l'action dans notre exemple :

Rendu de JSON

```

||| public function jsonrenderAction()
||| {
|||     $view = new ViewModel(array('content'=>array('value1','value2')));
|||     $view->setTerminal(true);
|||     return $view;
||| }
```

Notre vue contient simplement :

Rendu de JSON

```

||| <?php
||| echo $this->json($this->content);
||| ?>
```

Grâce à cette aide de vue, nous connaissons maintenant une troisième manière de rendre une vue au format JSON.

L'aide Url

L'aide de vue Url du framework ne possède qu'une seule méthode qui se base sur les identifiants de route afin de retourner l'URL correspondante. L'aide de vue est donc dépendante de l'objet de router et de l'objet contenant les paramètres de routes pour assembler les routes demandées. Cet objet est injecté dans la fabrique du gestionnaire de vues depuis une fabrique spécialisée :

Zend/Mvc/Service/ViewHelperManagerFactory.php

```

||| public function createService(ServiceLocatorInterface
||| $serviceLocator)
||| {
|||     [...]
|||     $plugins->setFactory('url', function($sm) use($serviceLocator) {
|||         $helper = new ViewHelper\Url;
|||         $helper->setRouter($serviceLocator->get('Router'));
|||         $match = $serviceLocator->get('application')
|||             ->getMvcEvent()
|||             ->getRouteMatch();
|||         if ($match instanceof RouteMatch) {
|||             $helper->setRouteMatch($match);
|||         }
|||         return $helper;
|||     });
|||     [...]
||| }
```

L'aide de vue est alors prête à fonctionner, nous pouvons l'utiliser :

Création d'une route

```

||| <?php echo $this->url('home');?>
```

Si le nom de notre route n'est pas nul, le router se contente de faire appel à la

méthode « assemble() » afin de créer l'URL :

Zend/View/Helper/Url.php

```
|| public function __invoke($name = null, array $params = array(),
|| $options = array(), $reuseMatchedParams = false)
|| {
|| [...]
|| $options['name'] = $name;
|| return $this->router->assemble($params, $options);
|| }
```

Si la route passée en paramètre a pour valeur « null », l'aide de vue utilise l'objet RouteMatch retourné par le router afin d'utiliser la route courante :

Zend/View/Helper/Url.php

```
|| public function __invoke($name = null, array $params = array(),
|| $options = array(), $reuseMatchedParams = false)
|| {
|| if ($name === null) {
|| [...]
||     $name = $this->routeMatch->getMatchedRouteName();
|| [...]
|| }
|| [...]
|| $options['name'] = $name;
|| return $this->router->assemble($params, $options);
|| }
```

Nous pouvons maintenant écrire dans notre vue :

Création de l'URL courante

```
|| <?php echo $this->url();?>
```

L'URL de la route courante est alors générée. Il est conseillé de toujours utiliser cette aide de vue pour gérer ses URL dans les différentes vues afin de toujours avoir la main sur les URL des routes depuis les fichiers de configuration.

Création d'une aide de vue

Lorsque vous développez votre application, vous pouvez avoir besoin de créer une aide de vue afin de factoriser le code des vues. La première étape est de faire étendre votre classe de la classe de base Zend\View\Helper\AbstractHelper qui vous donnera accès à l'objet de rendu :

Zend/View/Helper/AbstractHelper.php

```
|| abstract class AbstractHelper implements HelperInterface
|| {
||     protected $view = null;
|| 
||     public function setView(Renderer $view)
||     {
```

```

        $this->view = $view;
        return $this;
    }

    public function getView()
    {
        return $this->view;
    }
}

```

Une fois l'aide de vue écrite, il ne reste plus qu'à l'enregistrer auprès du gestionnaire de plugins. Cette opération peut se faire depuis la méthode « `getViewHelperConfiguration()` » de la classe de module :

Module.php

```

class Module implements ViewHelperProviderInterface
{
[...]
public function getViewHelperConfig()
{
    return array(
        'invokables' => array(
            'title' => 'Zend\View\Helper\HeadTitle',
        ),
    );
}
}

```

Il est également possible d'inscrire l'aide de vue depuis le fichier de configuration du module sous le nom de clé « `view_helpers` » :

module.config.php

```

return array(
    [...]
    'view_helpers' => array(
        'invokables' => array(
            'title' => 'Zend\View\Helper\HeadTitle'
        ),
    ),
)
;

```

Pour plus d'informations sur la gestion interne de la configuration et les clés de configuration disponibles pour l'enrichissement du gestionnaire de services, reportez-vous au chapitre consacré aux modules. Les exemples ci-dessus permettent d'ajouter un alias au plugin `HeadTitle` que l'on peut alors récupérer depuis la vue :

Vue de layout

```

<?php echo $this->title('Titre de mon application') ?>

```

14

Le cœur du framework

La plupart des composants de Zend Framework 2 ont été passés en revue, il est maintenant temps de nous plonger au cœur du framework afin d'analyser le processus de bootstrap et de lancement de l'application.

Il est indispensable aux développeurs de bien comprendre le fonctionnement interne du framework afin d'en tirer pleinement profit, d'optimiser les performances de leur application et de pouvoir répondre à toutes les problématiques auxquelles ils pourraient être confrontés.

Comprendre les points d'entrées et fonctionnement internes du framework nous permettra de développer plus facilement la couche métier de notre application Web. Par exemple, l'extension du Zend_Application dans la version 1 du framework pouvait parfois s'avérer très judicieuse afin de permettre quelques optimisations propres à notre application.

De plus, connaître les rouages du fonctionnement interne vous donnera les clés pour déboguer plus rapidement notre application et comprendre le cheminement effectué jusqu'au problème rencontré.

Initialisation des autoloaders

Afin de pouvoir travailler avec le framework et nos propres composants, il est nécessaire de configurer la gestion du chargement de classes. La première instruction de notre fichier index.php est la configuration des autoloaders :

```
index.php
|| chdir(dirname(__DIR__));
|| include 'init_autoloader.php';
```

La première instruction permet de toujours utiliser les chemins relatifs et de ne plus utiliser de chemin absolu afin de simplifier la gestion du chargement de classes. Cela permet de faciliter la vie du développeur, l'application entière peut alors être déplacée comme on le souhaite.

Analysons l'initialisation du chargement automatique de classe :

```
init_autoloader.php
if (file_exists('vendor/autoload.php')) {
    $loader = include 'vendor/autoload.php';
}
if (($zf2Path = getenv('ZF2_PATH') ?: (is_dir('vendor/ZF2/library') ?
    'vendor/ZF2/library' : false)) !== false) {

    if (isset($loader)) {
        $loader->add('Zend', $zf2Path . '/Zend');
    } else {
        include $zf2Path . '/Zend/Loader/AutoloaderFactory.php';
        Zend\Loader\AutoloaderFactory::factory(array(
            'Zend\Loader\StandardAutoloader' => array(
                'autoregister_zf' => true
            )
        ));
    }
}
```

Les premières instructions vérifient l'existence d'un autoloader afin de pouvoir l'utiliser. Si celui-ci existe, le namespace du framework est directement ajouté à la classe de chargement, sinon la fabrique AutoloaderFactory utilisera l'autoloader standard afin de charger les classes du framework. L'utilisation du paramètre « autoregister_zf » permet d'enregistrer automatiquement la bibliothèque, comme nous le voyons dans le code de l'autoloader :

Zend/Loader/StandardAutoloader.php

```
class StandardAutoloader implements SplAutoloader
{
    [...]
    const AUTOREGISTER_ZF = 'autoregister_zf';

    public function setOptions($options)
    {
        [...]
        foreach ($options as $type => $pairs) {
            switch ($type) {
                case self::AUTOREGISTER_ZF:
                    if ($pairs) {
                        $this->registerNamespace('Zend',
dirname(__DIR__));
                    }
                    break;
                [...]
            }
        }
    }
}
```

Une fois le chargement de classe configuré, le framework est fonctionnel et linitialisation de lapplication peut commencer. Notez que le fichier « init_autoloader.php » est une proposition faite par les développeurs afin de bénéficier dun autoloader rapide, mais libre à vous de le modifier partiellement ou complètement suivant les besoins de votre application.

Le bootstrap

Le premier élément à intervenir lors du processus de lancement de l'application est la méthode de bootstrap qui initialise les ressources nécessaires au processus MVC du framework afin de pouvoir laisser la classe Zend\ Mvc\ Application travailler, véritable chef d'orchestre du framework.

L'application est d'abord initialisée par la méthode statique « init() » qui s'occupe de gérer le chargement des modules, de construire le gestionnaire de services ainsi que de créer l'instance de l'application afin de lancer le processus de bootstrap :

index.php

```
|| Zend\ Mvc\ Application::init(include 'config/application.config.php')-
|| >run();
```

Zend/Mvc/Application.php

```
|| public static function init($configuration = array())
|| {
||     $smConfig = isset($configuration['service_manager']) ?
||     $configuration['service_manager'] : array();
|
||     $serviceManager = new ServiceManager(new Service\ServiceManagerConfigu-
||     ration($smConfig));
||     $serviceManager->setService('ApplicationConfiguration',
||     $configuration);
||     $serviceManager->get('ModuleManager')->loadModules();
|
||     return $serviceManager->get('Application')->bootstrap();
|| }
```

Nous remarquons que le gestionnaire de services est initialisé avec la clé « service_manager » de la configuration passée en paramètre. Il est donc possible d'initialiser le gestionnaire de services depuis son fichier de configuration :

application.config.php

```
|| return array(
||     [...]
||     'service_manager' => array(
||         'invokables' => array(
||             'ma-fabrique' => 'My\Factory',
||         ),
||     ),
|| );
```

La configuration est ensuite enregistrée comme service sous la clé « ApplicationConfiguration » avant que le gestionnaire de modules ne charge les modules et enregistre leur configuration. Pour plus de détails sur le chargement des modules, reportez-vous au chapitre consacré aux modules. Nous pourrons donc accéder à notre configuration depuis le gestionnaire de services et la clé « ApplicationConfiguration ».

Afin d'initialiser l'application, le gestionnaire de services utilise la fabrique sous l'identifiant « Application » disponible dans les fabriques du framework :

Zend/Mvc/Service/ServiceListenerFactory.php

```
class ServiceListenerFactory implements FactoryInterface
{
protected $defaultServiceConfig = array(
    [...],
    'factories' => array(
        'Application' => 'Zend\Mvc\Service\ApplicationFactory',
        'Config' => 'Zend\Mvc\Service\ConfigFactory',
        'ControllerLoader' => 'Zend\Mvc\Service\ControllerLoaderFactory',
        [...]
    ),
    [...]
);
[...]
}
```

La fabrique dédiée à la construction de la classe Application est la classe Zend\\Mvc\\Service\\ApplicationFactory qui se contente de passer la configuration de l'application à son constructeur :

Zend/Mvc/Service/ApplicationFactory.php

```
class ApplicationFactory implements FactoryInterface
{
public function createService(ServiceLocatorInterface
$serviceLocator)
{
    return new Application($serviceLocator->get('Configuration'),
$serviceLocator);
}
}
```

La classe Application enregistre la configuration ainsi que l'instance du gestionnaire de services :

Zend/Mvc/Application.php

```
public function __construct($configuration, ServiceManager  
$serviceManager)  
{  
    $this->configuration = $configuration;  
    $this->serviceManager = $serviceManager;  
  
    $this->setEventManager($serviceManager->get('EventManager'));  
  
    $this->request = $serviceManager->get('Request');  
    $this->response = $serviceManager->get('Response');  
}
```

Le constructeur initialise aussi l'objet de réponse et de requête qui lui seront nécessaires, et récupère les instances du gestionnaire de modules et d'événements.

Une fois notre classe initialisée, la méthode de bootstrap est ensuite appelée :

Zend/Mvc/Application.php

```
public static function init($configuration = array())
{
    [...]
    return $serviceManager->get('Application')->bootstrap();
}
```

La méthode « bootstrap() » initialise alors la classe de gestion de la distribution des actions, le router ainsi que le gestionnaire de vues :

Zend/Mvc/Application.php

```
public function bootstrap()
{
    $serviceManager = $this->serviceManager;
    $events        = $this->getEventManager();

    $events->attach($serviceManager->get('RouteListener'));
    $events->attach($serviceManager->get('DispatchListener'));
    $events->attach($serviceManager->get('ViewManager'));

    $this->event = $event = new MvcEvent();
    $event->setTarget($this);
    $event->setApplication($this)
        ->setRequest($this->getRequest())
        ->setResponse($this->getResponse())
        ->setRouter($serviceManager->get('Router'));

    $events->trigger(MvcEvent::EVENT_BOOTSTRAP, $event);
    return $this;
}
```

L'identifiant « RouteListener » représente l'agrégateur d'écouteur Zend\Mvc\RouteListener qui gère le routage de l'application :

Zend/Mvc/RouteListener.php

```
class RouteListener implements ListenerAggregateInterface
{
    [...]
    public function attach(EventManagerInterface $events)
    {
        $this->listeners[] = $events->attach(MvcEvent::EVENT_ROUTE,
        array($this, 'onRoute'));
    }
    [...]
}
```

Le routage est effectué lorsque l'application déclenche l'évènement « route ».

Comme pour le router, l'agrégateur Zend\Mvc\DispatchListener s'occupe de gérer la distribution de l'action de l'application :

Zend/Mvc/DispatchListener.php

```
class DispatchListener implements ListenerAggregateInterface
{
    [...]
    public function attach(EventManagerInterface $events)
    {
        $this->listeners[] = $events->attach(MvcEvent::EVENT_DISPATCH,
array($this, 'onDispatch'));
    }
    [...]
}
```

Les classes `RouteListener` et `DispatchListener` sont marquées comme « invocables » dans le gestionnaire de services, ce qui explique qu'aucune fabrique n'existe pour ces deux classes. En effet, les agrégateurs ne nécessitent pas d'initialisation particulière.

Le dernier élément construit par la méthode de bootstrap est le gestionnaire de vues « `ViewManager` » qui initialise tous les éléments nécessaires à la gestion des vues :

Zend/Mvc/View/Http/ViewManager.php

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(MvcEvent::EVENT_BOOTSTRAP,
array($this, 'onBootstrap'), 10000);
}
```

Le gestionnaire de vues attache tous les écouteurs nécessaires à la préparation, construction et rendu des vues sur l'évènement « `bootstrap` » qui est lancé à la fin de la méthode « `bootstrap()` » de la classe `Application`. Pour plus de détails sur le gestionnaire de vues, rendez-vous au chapitre consacré aux vues.

Le bootstrap construit alors l'objet d'évènement utilisé lors des différentes notifications avec les éléments dont il a besoin, et lance enfin l'évènement « `bootstrap` » :

Zend/Mvc/Application.php

```
public function bootstrap()
{
    [...]
    $this->event = $event = new MvcEvent();
    $event->setTarget($this);
    $event->setApplication($this)
        ->setRequest($this->getRequest())
        ->setResponse($this->getResponse())
        ->setRouter($serviceManager->get('Router'));

    $events->trigger(MvcEvent::EVENT_BOOTSTRAP, $event);
    return $this;
}
```

Une fois tous les éléments de l'application initialisés, le processus principal peut alors démarrer.

AVEC LE ZEND FRAMEWORK 1

Une classe de bootstrap spécialisée dans l'initialisation est utilisée afin de charger les ressources indiquées par l'utilisateur. Contrairement à la première version, Zend Framework 2 n'initialisera que les ressources qui seront nécessaires pour l'action en cours et qui sont demandées explicitement par l'utilisateur.

Le run

Le point d'entrée et de lancement de l'application est la méthode « `run()` » de la classe `Application` :

`index.php`

```
|| Zend\Mvc\Application::init(include 'config/application.config.php')->run();
```

Cette méthode récupère d'abord le gestionnaire d'évènements ainsi que l'objet d'évènement de la classe `Application` :

`Zend/Mvc/Application.php`

```
|| public function run()
  {
  $events = $this->getEventManager();
  $event  = $this->getMvcEvent();
  [...]
  }
```

L'objet d'évènement est créé dans la méthode « `bootstrap()` », alors que le gestionnaire d'évènements est récupéré depuis le gestionnaire de services, ce qui permet de récupérer l'instance partagée au sein de l'application. Lors de la récupération du gestionnaire d'évènements, ses identifiants lui sont ajoutés :

`Zend/Mvc/Application.php`

```
|| public function setEventManager(EventManagerInterface $eventManager)
  {
  $eventManager->setIdentifiers(array(
      __CLASS__,
      get_called_class(),
  ));
  $this->events = $eventManager;
  return $this;
  }
```

L'identifiant « `Zend\Mvc\Application` » est ajouté, ce qui permet alors d'attacher statiquement des écouteurs sur le gestionnaire d'évènements partagé avec cet identifiant pour écouter la classe `Application`.

L'évènement « route » est ensuite immédiatement lancé par la méthode « run() » de la classe Application, l'écouteur « onRoute() » de l'objet RouteListener, attaché dans le bootstrap est alors être notifiée :

Zend/Mvc/Application.php

```
public function run()
{
    $events = $this->getEventManager();
    $event = $this->getMvcEvent();
    $shortCircuit = function ($r) use ($event) {
        if ($r instanceof ResponseInterface) {
            return true;
        }
        if ($event->getError()) {
            return true;
        }
        return false;
    };
    $result = $events->trigger(MvcEvent::EVENT_ROUTE, $event,
    $shortCircuit);
    [...]
}
```

AVEC LE ZEND FRAMEWORK 1

Dans la version 1 du framework, la méthode « run() » du Zend_Application appelle la méthode « dispatch() » du Zend_Controller_Front. Cette classe crée les objets de requêtes et de réponses, puis notifie les plugins avec la méthode « routeStartup() » avant d'effectuer le routage. Chaque tâche et chaque rôle sont maintenant séparés ce qui donne plus de sens à chacun des composants.

Suite à l'évènement « route », l'application est capable de savoir quel est le couple contrôleur/action à utiliser lors de la distribution. L'étape de vérification des routes est détaillée dans le chapitre consacré aux routes.

La classe Application notifie ensuite ses écouteurs de l'évènement « dispatch » afin de lancer la distribution de l'action de la requête :

Zend/Mvc/Application.php

```
public function run()
{
    [...]
    $result = $events->trigger(MvcEvent::EVENT_DISPATCH, $event,
    $shortCircuit);
    [...]
}
```

Cet évènement notifie la méthode « onDispatch() » de l'objet DispatchListener, agrégateur spécialisé dans la distribution.

AVEC LE ZEND FRAMEWORK 1

Il n'existe pas d'évènements « `dispatch.pre` » et « `dispatch.post` » dans le Zend Framework 2, ceci est lié au fait que la méthode « `attach()` » permet de gérer les priorités de la `SplPriorityQueue`, il suffit donc de jouer avec les priorités pour obtenir le même effet que les méthodes « `preDispatch()` » et « `postDispatch()` » de la première version.

La méthode « `onDispatch()` » récupère les objets dont elle a besoin pour travailler :

Zend/Mvc/DispatchListener.php

```
public function onDispatch(MvcEvent $e)
{
    $routeMatch      = $e->getRouteMatch();
    $controllerName = $routeMatch->getParam('controller', 'not-found');
    $application    = $e->getApplication();
    $events         = $application->getEventManager();
    $controllerLoader = $application->getServiceManager()->get('ControllerLoader');
    [...]
}
```

L'objet « `$routeMatch` », créé lors du routage, enveloppe les noms du contrôleur et de l'action à distribuer. Le nom du contrôleur est récupéré ainsi que le gestionnaire de contrôleurs `ControllerManager` qui étend le gestionnaire de plugins de base `AbstractPluginManager`. Pour plus de détails sur le gestionnaire de plugins, reportez-vous au chapitre qui lui est consacré. Le gestionnaire de contrôleurs est construit depuis la fabrique « `ControllerLoader` » dont voici la classe `Zend\Mvc\Service\ControllerLoaderFactory` correspondante :

Zend/Mvc/Service/ControllerLoaderFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $controllerLoader = new ControllerManager();
    $controllerLoader->setServiceLocator($serviceLocator);
    $controllerLoader->addPeeringServiceManager($serviceLocator);

    $config = $serviceLocator->get('Config');
    if (isset($config['di']) && isset($config['di']['allowed_controllers']) && $serviceLocator->has('Di')) {
        $diAbstractFactory = new DiStrictAbstractServiceFactory(
            $serviceLocator->get('Di'),
            DiStrictAbstractServiceFactory::USE_SL_BEFORE_DI
        );
        $diAbstractFactory->setAllowedServiceNames($config['di']['allowed_controllers']);
        $controllerLoader->addAbstractFactory($diAbstractFactory);
    }
    return $controllerLoader;
}
```

Les premières instructions créent un gestionnaire de contrôleurs avant d'y injecter le gestionnaire de services et de le définir dans le périmètre comme gestionnaire parent de l'objet créé héritant de la classe de base ServiceManager :

Zend/Mvc/Service/ControllerLoaderFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    $controllerLoader = new ControllerManager();
    $controllerLoader->setServiceLocator($serviceLocator);
    $controllerLoader->addPeeringServiceManager($serviceLocator);
    [...]
}
```

La méthode « addPeeringServiceManager() » enregistre le gestionnaire de services courant comme gestionnaire enfant du gestionnaire de contrôleurs. Cette manière de faire permet d'exploiter les possibilités du gestionnaire de services depuis un objet indépendant avec ses propres fabriques et identifiants de fabriques, tout en laissant la possibilité d'utiliser le gestionnaire fils ou parent en cas de nécessité. Pour plus d'informations sur la gestion des gestionnaires de services imbriqués, reportez-vous au chapitre consacré au gestionnaire de services.

Une fois le nouveau gestionnaire instancié, le composant d'injection de dépendance est ajouté comme couche supplémentaire si celle-ci existe dans le gestionnaire courant :

Zend/Mvc/Service/ControllerLoaderFactory.php

```
public function createService(ServiceLocatorInterface
    $serviceLocator)
{
    [...]
    if (isset($config['di']) && isset($config['di']['allowed_controllers'])
        && $serviceLocator->has('Di')) {
        $diAbstractFactory = new DiStrictAbstractServiceFactory(
            $serviceLocator->get('Di'),
            DiStrictAbstractServiceFactory::USE_SL_BEFORE_DI
        );
        $diAbstractFactory->setAllowedServiceNames($config['di']
            ['allowed_controllers']);
        $controllerLoader->addAbstractFactory($diAbstractFactory);
    }
    [...]
}
```

Le composant Di pour le ServiceManager est expliqué en détail dans le chapitre sur le gestionnaire de services.

Une fois le gestionnaire de services dédié aux contrôleurs instancié, la distribution reprend en chargeant le contrôleur de l'action en cours :

Zend/Mvc/DispatchListener.php

```

public function onDispatch(MvcEvent $e)
{
[...]
try {
    $controller = $controllerLoader->get($controllerName);
} catch (ServiceNotFoundException $exception) {
    [...]
}
$request = $e->getRequest();
$response = $application->getResponse();

if ($controller instanceof InjectApplicationEventInterface) {
    $controller->setEvent($e);
}
try {
    $return = $controller->dispatch($request, $response);
} catch (\Exception $ex) {
    [...]
}
return $this->complete($return, $e);
}

```

Le contrôleur est instancié depuis la méthode « `get()` » du gestionnaire de services, et une vérification est faite sur le bon chargement du contrôleur. Si celui-ci n'a pas pu être chargé, alors les instructions de la gestion des erreurs sont exécutées. À noter que le type d'erreur est choisi en fonction de l'exception lancée afin de pouvoir fournir à l'utilisateur un descriptif d'erreur plus explicite.

AVEC LE ZEND FRAMEWORK 1

La distribution du `Zend_Controller_Front` gère les erreurs depuis les exceptions, et se contente de renvoyer aussi l'exception si cela est précisé, sinon il renseigne l'objet de réponse de cette exception.

La méthode s'assure ensuite que le contrôleur soit distribuable et injecte l'évènement courant au contrôleur avant de distribuer la requête :

Zend/Mvc/DispatchListener.php

```

public function dispatch(MvcEvent $e)
{
[...]
if (!$controller instanceof DispatchableInterface) {
    [...]
}
$request = $e->getRequest();
$response = $application->getResponse();
if ($controller instanceof InjectApplicationEventInterface) {
    $controller->setEvent($e);
}
try {
    $return = $controller->dispatch($request, $response);
}
[...]
}

```

Une fois l'action distribuée et le retour de celle-ci obtenu, la méthode injecte ce contenu dans le résultat de l'objet d'évènement :

Zend/Mvc/DispatchListener.php

```
public function dispatch(MvcEvent $e)
{
[...]
return $this->complete($return, $e);
}
```

Zend/Mvc/DispatchListener.php

```
protected function complete($return, MvcEvent $event)
{
if (!is_object($return)) {
    if (ArrayUtils::hasStringKeys($return)) {
        $return = new ArrayObject($return, ArrayObject::ARRAY_
AS_PROPS);
    }
}
$event->setResult($return);
return $return;
}
```

La réponse de l'écouteur est ensuite retournée au gestionnaire d'évènements.

Une fois la distribution effectuée, la méthode « `run()` » récupère la dernière réponse reçue par le gestionnaire et l'injecte à l'objet de réponse :

Zend/Mvc/Application.php

```
public function run()
{
[...]
$result = $events->trigger(MvcEvent::EVENT_DISPATCH, $event,
$shortCircuit);
$response = $result->last();
if ($response instanceof ResponseInterface) {
    $event->setTarget($this);
    $events->trigger(MvcEvent::EVENT_FINISH, $event);
    return $response;
}

$response = $this->getResponse();
$event->setResponse($response);
return $this->completeRequest($event);
}
```

Le processus se termine avec la méthode « `completeRequest()` » qui lance successivement l'évènement de rendu de vue, suivi de la notification de fin de traitement :

Zend/Mvc/Application.php

```
public function run()
{
[...]
$response = $this->getResponse();
```

```
    $event->setResponse($response);
    return $this->completeRequest($event);
}
```

Zend/Mvc/Application.php

```
protected function completeRequest(MvcEvent $event)
{
    $events = $this->getEventManager();
    $event->setTarget($this);
    $events->trigger(MvcEvent::EVENT_RENDER, $event);
    $events->trigger(MvcEvent::EVENT_FINISH, $event);
    return $event->getResponse();
}
```

L'évènement « finish », représenté par la constante MvcEvent::EVENT_FINISH, ne possède qu'un seul écouteur avec l'objet Zend\Mvc\View\SendResponseListener et sa méthode « sendResponse() » qui permet d'afficher la réponse pour le client :

Zend/Mvc/View/SendResponseListener.php

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(MvcEvent::EVENT_FINISH,
    array($this, 'sendResponse'), -10000);
}
```

Zend/Mvc/View/SendResponseListener.php

```
public function sendResponse(MvcEvent $e)
{
    $response = $e->getResponse();
    if (!$response instanceof Response) {
        return false;
    }

    if(is_callable(array($response, 'send'))){
        return $response->send();
    }
}
```

Zend/Http/PhpEnvironment/Response.php

```
public function sendContent()
{
    if ($this->contentSent()) {
        return $this;
    }
    echo $this->getContent();
    $this->contentSent = true;
    return $this;
}
```

Zend/Http/PhpEnvironment/Response.php

```
public function send()
{
    $this->sendHeaders()
        ->sendContent();
    return $this;
}
```

AVEC LE ZEND FRAMEWORK 1

La méthode « `dispatch()` » du contrôleur frontal rend automatiquement la vue si on ne le lui spécifie pas le contraire depuis la méthode « `returnResponse()` » qui permet de modifier la valeur de l'attribut de rendu. Le processus de bootstrap et de lancement de l'application ont été modifiés afin de mieux séparer et organiser toutes les tâches, tout en laissant un peu plus de liberté aux développeurs avec la gestion des évènements.

15

Les composants

Le Zend Framework fournit tout un ensemble de composants que l'on peut utiliser indépendamment de l'ensemble du framework ou depuis notre application. Certains composants de la première version sont jugés peu performants ou mal écrits, et le besoin de compatibilité ascendante fait qu'aucun changement majeur n'a été effectué. Cette deuxième version du framework permet de faire évoluer ces composants qui ne répondent plus aux besoins en termes de fonctionnalités ou de performances.

Les développeurs ont aussi profité de cette deuxième version pour effectuer des changements au niveau des différents dépôts fournis par l'équipe. Si la première version du Zend Framework englobait tous les composants, ce n'est plus le cas aujourd'hui. En effet, les composants agissant sur des services externes ne sont plus embarqués avec le code du framework. Ceci a été fait pour la simple et bonne raison que ce sont des produits avec des contraintes liées aux API des services qui leur sont rattachés. Une mise à jour de l'API de Twitter ou Amazon obligeait le framework à publier une nouvelle version de son code, bien que seul ce service soit mis à jour. C'est donc pour gérer plus facilement ce genre de contrainte que chaque service dépendant d'un service externe a été dissocié du framework. Vous retrouverez donc ces services sur le compte Github du zend framework qui seront mis à jour en fonction des besoins et toujours maintenus.

Zend\Db

L'adaptateur de base de données

L'adaptateur de base de données permet de piloter la connexion à la base de données depuis n'importe quel SGBD supporté (Mysqli, Pgsql, Sqlserver ou encore l'utilisation de Pdo). La construction de l'adaptateur de base de données se fait simplement en passant en paramètre le driver souhaité :

Construction de l'adaptateur à la base de données

```
|| $adapater = new \Zend\Db\Adapter\Adapter(array('driver'=>'mysqli'));
```

L'objet Zend\Db\Adapter\Adapter accepte un tableau de paramètres ou une ins-

tance de l'objet de driver, de type Zend\Db\Adapter\Driver\DriverInterface, en entrée :

Zend/Db/Adapter/Adapter.php

```
public function __construct($driver, Platform\PlatformInterface
    $platform = null, ResultSet\ResultSet $queryResultPrototype = null)
{
    if (is_array($driver)) {
        $driver = $this->createDriverFromParameters($driver);
    } elseif (!($driver instanceof Driver\DriverInterface)) {
        throw new Exception\InvalidArgumentException([...]);
    }
    [...]
}
```

Si nous passons un tableau en paramètre, comme dans l'exemple, le constructeur se charge d'instancier l'objet :

Zend/Db/Adapter/Adapter.php

```
protected function createDriverFromParameters(array $parameters)
{
    [...]
    $driverName = strtolower($parameters['driver']);
    switch ($driverName) {
        case 'mysqli':
            $driver = new Driver\Mysqli\Mysqli($parameters, null,
null, $options);
            break;
        case 'sqlsrv':
            $driver = new Driver\Sqlsrv\Sqlsrv($parameters);
            break;
        case 'pgsql':
            $driver = new Driver\Pgsql\Pgsql($parameters);
            break;
        case 'pdo':
        default:
            if ($driverName == 'pdo' || strpos($driverName,
'pdo') === 0) {
                $driver = new Driver\Pdo\Pdo($parameters);
            }
    }
    [...]
    return $driver;
}
```

Nous remarquons que l'objet de driver Mysqli est instancié avec les paramètres passés au constructeur de l'adaptateur. Le constructeur de la classe Mysqli instancie un objet de type Zend\Db\Adapter\Driver\Mysqli\Connection responsable de la gestion de la connexion à la base de données :

Zend/Db/Adapter/Driver/Mysqli/Mysqli.php

```
public function __construct($connection, Statement
    $statementPrototype = null, Result $resultPrototype = null, array
    $options = array())
{
```

```

    if (!$connection instanceof Connection) {
        $connection = new Connection($connection);
    }
    [...]
}

```

Zend/Db/Adapter/Driver/Mysqli/Connection.php

```

public function __construct($connectionInfo = null)
{
    if (is_array($connectionInfo)) {
        $this->setConnectionParameters($connectionInfo);
    } elseif ($connectionInfo instanceof \mysqli) {
        $this->setResource($connectionInfo);
    }
    [...]
}

```

Les paramètres de connexion à la base de données sont conservés afin d'établir la connexion lorsque celle-ci sera réellement nécessaire :

Zend/Db/Adapter/Driver/Mysqli/Connection.php

```

public function connect()
{
    [...]
    $p = $this->connectionParameters;
    $findParameterValue = function(array $names) use ($p) {
        foreach ($names as $name) {
            if (isset($p[$name])) {
                return $p[$name];
            }
        }
        return null;
    };

    $hostname = $findParameterValue(array('hostname', 'host'));
    $username = $findParameterValue(array('username', 'user'));
    $password = $findParameterValue(array('password', 'passwd', 'pw'));
    $database = $findParameterValue(array('database', 'dbname', 'db',
        'schema'));
    $port      = (isset($p['port'])) ? (int) $p['port'] : null;
    $socket    = (isset($p['socket'])) ? $p['socket'] : null;
    $this->resource = new \Mysqli($hostname, $username, $password,
        $database, $port, $socket);
    [...]
}

```

Nous remarquons que nous pouvons utiliser plusieurs clés pour spécifier les valeurs de l'utilisateur, le mot de passe ou encore la base de données :

Utilisation des noms de paramètres

```

public function connect()
{
    [...]
    $password = $findParameterValue(array('password', 'passwd', 'pw'));
    [...]
}

```

Le fait de pouvoir utiliser « password », « passwd » ou « pw » offre un peu plus de souplesse à l'utilisateur sans devoir se souvenir de la clé exacte pour configurer sa connexion.

Nous pouvons alors passer la configuration de connexion dans le constructeur de l'adaptateur :

Construction d'un adaptateur avec sa configuration

```
|| $adapater = new \Zend\Db\Adapter\Adapter(array(
||     'driver'=>'mysqli',
||     'hostname'=>'localhost',
||     'username'=>'root',
||     'password'=>'root',
||     'database'=>'zf2'
|| ));
```

Une fois le driver instancié, le constructeur se charge de vérifier l'environnement en cours afin de contrôler la bonne configuration des extensions PHP :

Zend/Db/Adapter/Adapter.php

```
|| public function __construct($driver, Platform\PlatformInterface
|| $platform = null, ResultSet\ResultSet $queryResultPrototype = null)
|| {
|| [...]
|| $driver->checkEnvironment();
|| $this->driver = $driver;
|| [...]
|| }
```

Zend/Db/Adapter/Driver/Mysqli/Mysqli.php

```
|| public function checkEnvironment()
|| {
||     if (!extension_loaded('mysqli')) {
||         throw new \Exception('The Mysqli extension is required for this
|| adapter but the extension is not loaded');
||     }
|| }
```

Lorsque les extensions ont été vérifiées, le driver est enregistré. L'adaptateur construit ensuite son objet de gestion de plateforme qui lui permet de bénéficier d'une API propre à celle-ci, qui sera utilisée ensuite lors de la construction des requêtes :

Zend/Db/Adapter/Adapter.php

```
|| public function __construct($driver, Platform\PlatformInterface
|| $platform = null, ResultSet\ResultSet $queryResultPrototype = null)
|| {
|| [...]
||     if ($platform == null) {
||         $platform = $this->createPlatformFromDriver($driver);
||     }
||     $this->platform = $platform;
```

```

    $this->queryResultSetPrototype = ($queryResultSetPrototype) ?: new
    ResultSet\ResultSet();
}

```

L'adaptateur est maintenant prêt à l'emploi et peut interroger directement la base de données depuis sa méthode « `query()` » en lui fournissant une chaîne de caractères correspondant à une requête SQL :

Requête en base de données

```

$results = $adapater->query('SELECT * FROM matable',\Zend\Db\Adapter\
Adapter::QUERY_MODE_EXECUTE);

```

L'objet retourné par la méthode « `query()` » est de type `Zend\Db\ResultSet\ResultSet`, conteneur de résultats par défaut si celui-ci n'est pas redéfini dans les arguments du constructeur :

Zend/Db/Adapter/Adapter.php

```

public function __construct($driver, Platform\PlatformInterface
$platform = null, ResultSet\ResultSet $queryResultSetPrototype = null)
{
[...]
$this->queryResultSetPrototype = ($queryResultSetPrototype) ?: new
ResultSet\ResultSet();
}

```

Zend/Db/Adapter/Adapter.php

```

public function query($sql, $parametersOrQueryMode = self::QUERY_
MODE_PREPARE)
{
[...]
if ($mode == self::QUERY_MODE_PREPARE) {
[...]
} else {
    $result = $this->driver->getConnection()->execute($sql);
}

if ($result instanceof Driver\ResultInterface && $result-
>isQueryResult()) {
    $resultSet = clone $this->queryResultSetPrototype;
    $resultSet->initialize($result);
    return $resultSet;
}
return $result;
}

```

Le deuxième paramètre de la fonction permet de spécifier le mode de préparation de la requête avant l'envoi en base de données.

L'adaptateur offre aussi la possibilité d'utiliser les spécificités de la plateforme utilisée depuis l'objet de type `PlatformInterface` créé lors de sa construction :

Utilisation de l'API liée à la plateforme

```

$sql = 'SELECT * FROM ' . $adapater->getPlatform()->quoteIdentifier('matable');

```

Requête SQL

```
|| SELECT * FROM `matable`
```

La préparation de la requête se réalise grâce au driver enregistré ainsi qu'à l'objet correspondant à la plateforme :

Préparation de requête

```
|| $sql = 'SELECT * FROM ' . $adapater->getPlatform()->quoteIdentifier('matable') . ' WHERE id = ' . $adapater->getDriver()->formatParameterName('id');

|| $statement = $adapater->query($sql);
|| $parameters = array('id' => 1);
|| $results = $statement->execute($parameters);
```

L'objet retourné par la méthode « execute() » est de type Zend\Db\Adapter\Driver\Mysqli\Result, capable d'être parcouru par itérations.

L'abstraction SQL

Les objets de types Zend\Db\Sql* représentent des couches d'abstraction qui permettent de générer des chaînes SQL. Voici un exemple simple de ce qu'il est possible d'écrire depuis ces objets :

Utilisation de l'abstraction

```
|| $select = new \Zend\Db\Sql\Select();
|| $select->from('matable')
|| ->where->equalTo('matable.id', '1');
|| echo $select->getSqlString();
```

Affichage de la sortie

```
|| SELECT "matable".* FROM "matable" WHERE "matable"."id" = '1'
```

Ils permettent aussi de créer des « statements » depuis l'adaptateur :

Utilisation de l'abstraction

```
|| $select = new \Zend\Db\Sql\Select();
|| $select->from('matable')
|| ->where->equalTo('matable.id', '1');

|| $statement = $adapater->createStatement();

|| $select->prepareStatement($adapater, $statement);
|| $results = $statement->execute();
|| $row = $results->current();
```

L'objet Zend\Db\Sql\Where, représentant les conditions sur les requêtes est désormais riche en fonctionnalités. Cette classe hérite de Zend\Db\Sql\Predicate\Predicate représentant les prédictats :

Zend/Db/Sql/Predicate/Predicate.php

```

class Predicate extends PredicateSet
{
[...]
public function equalTo($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE);

public function lessThan($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE);
[...]
public function literal($literal, $parameter);

public function isNotNull($identifier);

public function in($identifier, $valueSet = null);

public function between($identifier, $minValue, $maxValue);
}

```

Il est désormais plus facile de composer des clauses complexes avec la refonte des objets d'abstraction SQL.

L'interface TableGateway

L'interface Zend\Db\TableGateway\TableGatewayInterface permet aux classes qui l'implémentent de pouvoir représenter une table en base de données, voici ses prototypes :

Zend/Db/TableGateway/TableGatewayInterface.php

```

interface TableGatewayInterface
{
public function getTableName();

public function select($where = null);

public function insert($set);

public function update($set, $where = null);

public function delete($where);
}

```

La classe Zend\Db\TableGateway\TableGateway implémente cette interface et fournit des méthodes prêtes à l'emploi afin d'effectuer toutes les opérations usuelles sur une table en base de données :

Utilisation de la classe TableGateway

```

$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);

$rowset = $maTable->select(array('id' => 1));
$row = $rowset->current();

```

Il est possible d'utiliser l'objet de table avec les objets Zend\Db\Sql\Select :

Utilisation de la classe TableGateway

```
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);
$select = new \Zend\Db\Sql\Select();
$select->from('matable')
->where->equalTo('matable.id', '1');

$rowset = $maTable->selectWith($select);
$row = $rowset->current();
```

Il est également possible d'utiliser les closures dans la méthode « `select()` » de la classe :

Zend/Db/TableGateway/AbstractTableGateway.php

```
public function select($where = null)
{
    if (!$this->isInitialized) {
        $this->initialize();
    }
    $select = $this->sql->select();
    if ($where instanceof \Closure) {
        $where($select);
    } elseif ($where !== null) {
        $select->where($where);
    }
    return $this->selectWith($select);
}
```

La méthode « `initialize()` » permet de construire les objets nécessaires à la confection des requêtes SQL. En effet, ceux-ci se réalisent à la demande afin de ne pas construire des objets qui ne seraient pas utilisés.

Si le paramètre d'entrée est de type `\Closure`, la fonction anonyme sera alors utilisée de la même manière que l'objet de type `Zend\Db\Sql\Select` :

Utilisation des closures

```
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);
$rowset = $maTable->select(function (\Zend\Db\Sql\Select $select) {
    $select->where->equalTo('matable.id', '1');
});
$row = $rowset->current();
```

Le corps de la fonction permet de comprendre qu'il est aussi possible de passer des clauses directement à la méthode, avec une chaîne de caractères par exemple :

Ajout de clause sur la requête depuis une chaîne

```
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);
$rowset = $maTable->select('id = 1');
$row = $rowset->current();
```

Le tableau peut être également utilisé :

Ajout de clause sur la requête depuis un tableau

```
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);
$rowset = $maTable->select(array('id'=>'1'));
$row = $rowset->current();
```

Une notation spéciale pour la préparation de requête est disponible :

Ajout de clause sur la requête depuis la notation de préparation

```
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
$adapater);

$rowset = $maTable->select(array('id > ?'=>1));
$row = $rowset->current();
```

Ces différentes possibilités sont consultables dans la méthode « where() » de la classe Select, utilisée par défaut dans l'objet TableGateway lors de l'appel à la méthode « select() » :

Zend/Db/Sql/Select.php

```
public function where($predicate, $combination = Predicate\
PredicateSet::OP_AND)
{
    if ($predicate instanceof Where) {
        $this->where = $predicate;
    } elseif ($predicate instanceof \Closure) {
        $predicate($this->where);
    } else {
        if (is_string($predicate)) {
            $predicate = new Predicate\Expression($predicate);
            $this->where->addPredicate($predicate, $combination);
        } elseif (is_array($predicate)) {
            foreach ($predicate as $pkey => $pvalue) {
                [...]
            }
        }
    }
    return $this;
}
```

Une autre classe hérite de TableGateway afin de permettre la gestion automatique de deux adaptateurs pour la distribution en lecture et écriture sur un couple Master/Slave. Voici une description de cette classe nommée MasterSlaveTableGateway :

Zend/Db/TableGateway/Feature/MasterSlaveTableGateway.php

```
class MasterSlaveTableGateway extends TableGateway
{
    public function __construct(Adapter $slaveAdapter)
    {
        $this->slaveAdapter = $slaveAdapter;
    }
}
```

```

public function postInitialize()
{
    $this->masterAdapter = $this->tableGateway->adapter;
}

public function preSelect()
{
    $this->tableGateway->adapter = $this->slaveAdapter;
}

public function postSelect()
{
    $this->tableGateway->adapter = $this->masterAdapter;
}
}

```

Son utilisation est assez simple, il suffit de passer les deux adaptateurs correspondant au serveur maître ainsi qu'à celui du serveur esclave à la classe afin qu'elle puisse distribuer les requêtes de lecture au slave et les écritures au master. L'adaptateur correspondant au serveur esclave est utilisé uniquement pour les requêtes de type « select » grâce aux méthodes « preSelect() » et « postSelect() » qui sont notifiées avant l'exécution de ces requêtes :

Zend/Db/TableGateway/AbstractTableGateway.php

```

protected function executeSelect(Select $select)
{
[...]
$this->featureSet->apply('preSelect', array($select));
[...]
$this->featureSet->apply('postSelect', array($statement, $result,
$resultSet));
return $resultSet;
}

```

La classe MasterSlaveTableGateway peut donc changer l'adaptateur pour les requêtes de type « select » qu'elle peut distribuer au serveur esclave.

L'interface RowGateway

Le framework offre aussi la possibilité de représenter les lignes de résultats en base de données par des objets. L'interface Zend\Db\RowGateway\RowGatewayInterface qui définit le contrat à respecter pour ces objets possède deux méthodes :

Zend/Db/RowGateway/RowGatewayInterface.php

```

interface RowGatewayInterface
{
    public function save();
    public function delete();
}

```

L'implémentation de cette interface permet d'agir sur les lignes de résultats de requête en base de données. Par défaut, les lignes de résultats retournées par la méthode « select() » de l'objet TableGateway sont encapsulées dans une instance

de Zend\Db\ResultSet\ResultSet :

Zend/Db/TableGateway/TableGateway.php

```
public function __construct($table, Adapter $adapter, $features =
    null, ResultSetInterface $resultSetPrototype = null, Sql $sql = null)
{
    [...]
    $this->resultSetPrototype = ($resultSetPrototype) ?: new ResultSet;;
    [...]
}
```

La classe ResultSet implémente par défaut un objet de type ArrayObject, avec la propriété « ARRAY_AS_PROPS » qui permet d'accéder aux clés du tableau comme un objet, afin d'envelopper chacune des lignes :

Zend/Db/ResultSet/ResultSet.php

```
public function __construct($returnType = self::TYPE_ARRAYOBJECT,
    $arrayObjectPrototype = null)
{
    $this->returnType = (in_array($returnType, array(self::TYPE_ARRAY,
        self::TYPE_ARRAYOBJECT))) ? $returnType : self::TYPE_ARRAYOBJECT;

    if ($this->returnType === self::TYPE_ARRAYOBJECT) {
        $this->setArrayObjectPrototype(($arrayObjectPrototype) ?: new
        ArrayObject(array(), ArrayObject::ARRAY_AS_PROPS));
    }
}
```

Avec l'implémentation par défaut, nous ne pouvons donc pas intervenir sur la base de données depuis les lignes de résultats. La classe RowGateway définit un conteneur de résultats qui offre la possibilité d'interagir avec la base de données. Voici un exemple de son utilisation :

Utilisation de la classe RowGateway

```
$featureset = new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id');
$maTable = new \Zend\Db\TableGateway\TableGateway('matable',
    $adapter, $featureset)$rowset = $maTable->select(array('id' => 1));
$row = $rowset->current();
$row['myfield'] = 'myupdate';
$affected = $row->save();

$rowset = $maTable->select(array('id' => 1));
$row = $rowset->current();
echo $row['myfield'];
```

Sortie de l'écran

```
myupdate
```

La deuxième instruction injecte un objet de type RowGateway dans le conteneur de résultats afin qu'il soit utilisé pour envelopper les différentes lignes de résultats issues de la base de données à travers la classe RowGatewayFeature qui injecte l'objet désiré :

Zend\Db\TableGateway\Feature\RowGatewayFeature.php

```

class RowGatewayFeature extends AbstractFeature
{
    public function postInitialize()
    {
        [...]
        if (isset($args[0])) {
            if (is_string($args[0])) {
                $primaryKey = $args[0];
                $rowGatewayPrototype = new RowGateway($primaryKey,
                $this->tableGateway->table, $this->tableGateway->adapter, $this-
                >tableGateway->sql);
                $resultSetPrototype->setArrayObjectPrototype($rowG
                atewayPrototype);
            } elseif ($args[0] instanceof RowGatewayInterface) {
                $rowGatewayPrototype = $args[0];
                $resultSetPrototype->setArrayObjectPrototype($rowG
                atewayPrototype);
            }
        }
        [...]
    }
}

```

Une fois l'objet crée avec la clé primaire passée en premier argument, l'objet RowGateway s'inscrit au sein du conteneur de résultats pour envelopper les lignes de base de données que l'on va récupérer. Il serait donc également possible de modifier soi-même le conteneur de lignes pour lui fournir un objet de type RowGateway, mais les objets du composant Zend/Db/TableGateway/Feature permettent de venir ajouter les fonctionnalités demandées au moment où elles seront nécessaires.

L'instruction suivante effectue une requête en base de données, puis le premier élément de la liste de résultats est récupéré dans la variable « \$row ». La modification des données de la ligne de résultats s'effectue comme celle d'un tableau. Cette manière de faire est autorisée par l'interface Zend\Db\ResultSet\RowObjectInterface, dont implémente RowGateway qui hérite de ArrayAccess, interface permettant d'accéder aux propriétés des objets de la même façon que pour l'accès aux données des tableaux.

La méthode « save() » permet ensuite la mise à jour de la ligne de résultats avec la base de données :

Zend/Db/RowGateway/AbstractRowGateway.php

```

public function save()
{
    [...]
    $this->initialize();
    if ($this->rowExistsInDatabase()) {
        [...]
        $statement = $this->sql->prepareStatementForSqlObject($this-
        >sql->update()->set($data)->where($where));
        $result = $statement->execute();
        $rowsAffected = $result->getAffectedRows();
        [...]
    } else {
        $insert = $this->sql->insert();
        $insert->values($this->data);
    }
}

```

```

        [...]
    }
    $statement = $this->sql->prepareStatementForSqlObject($this->sql-
    >select()->where($where));
    $result = $statement->execute();
    $rowData = $result->current();
    unset($statement, $result);
    $this->populate($rowData, true);
    return $rowsAffected;
}

```

La sauvegarde consiste à faire une mise à jour si la ligne existe en base de données et un insert si ce n'est pas le cas. Une fois cette opération effectuée, un select est effectué pour récupérer les données enregistrées. L'intérêt de faire un rafraîchissement des données est de récupérer les données réellement en base, car celles-ci peuvent changer si un trigger est utilisé. Il n'est donc pas nécessaire de devoir récupérer manuellement la nouvelle ligne de résultats comme nous l'avions fait dans notre exemple.

Zend\Cache

Le composant de cache du framework a été entièrement revu afin de pouvoir mieux séparer les différentes responsabilités liées à la création du cache de l'application.

Les sous-composants qui gèrent le stockage des caches se trouvent dans l'espace de nom Zend\Cache\Storage\Adaptateur où l'on retrouve les principaux gestionnaires de stockage connus : APC, memcached, système de fichiers, etc.

Les adaptateurs

La configuration des adaptateurs se réalise à l'aide de classes d'options dédiées à chacun d'entre eux. Prenons un exemple avec la gestion du cache par système de fichiers :

Utilisation du cache par fichiers

```

$cache = new \Zend\Cache\Storage\Adapter\Filesystem();
$cacheOptions = new \Zend\Cache\Storage\Adapter\FilesystemOptions(
    array('cache_dir'=>__DIR__ . '/data/')
);
$cache->setOptions($cacheOptions);
if(!$cache->hasItem('ma-cle')) {
    $cache->addItem('ma-cle', 'ma valeur');
}

```

Les classes d'options portent le nom de l'adaptateur concerné concaténé avec le suffixe « Options » : ApcOptions, MemcachedOptions, etc., et chacune hérite de la classe Zend\Stdlib\Options qui offre la possibilité de recevoir un tableau d'options afin de peupler l'objet. Les clés du tableau d'options sont transformées en nom de méthode d'altération, la propriété « cache_dir » est, par exemple, transformée en nom de méthode « setCacheDir() » où la valeur « __DIR__ . '/data/' » lui

est fournie. Pour plus de détails, le chapitre 6.1 sur la configuration des modules analyse le fonctionnement de la classe `Zend\Stdlib\Options`.

Lorsque l'adaptateur est instancié et configuré, il est maintenant utilisable. Notre exemple ajoute la valeur « ma valeur » au fichier créé dans le cache. Celui-ci contient la valeur brute « ma valeur » que l'on a définie.

Afin de nous éviter de devoir créer les différents objets adaptateurs et de les peupler manuellement, une fabrique de caches existe avec la classe `Zend\Cache\StorageFactory` qui permet de créer directement les adaptateurs et plugins depuis la fabrique :

Utilisation de la fabrique de caches

```
$cache = \Zend\Cache\StorageFactory::factory(array
    'adapter' => array(
        'name' => 'filesystem',
        'options' => array('cache_dir'=>__DIR__ . '/data/')
    ),
    'plugins' => array(
        'exception_handler' => array('throw_exceptions' =>
    false),
    ),
);
$cache->addItem('ma-cle','ma valeur');
```

D'autres méthodes de récupération, de suppression ou de mise à jour de caches sont disponibles depuis l'adaptateur et il est également possible d'agir sur le contenu de notre fichier de cache grâce à l'utilisation des différents plugins.

Les plugins

Les plugins offrent des fonctionnalités supplémentaires à l'adaptateur de cache, et permettent aussi d'agir sur son contenu. Ils vont permettre, par exemple, la sérialisation ou encore la désactivation du lancement des exceptions générées par les adaptateurs de cache. Prenons l'exemple de deux plugins.

- Le plugin de gestion d'exception

Reprenons le premier exemple en désactivant le lancement des exceptions, ce qui nous permet de ne pas contrôler l'existence du fichier de cache :

Gestion automatique des exceptions

```
$cache = new \Zend\Cache\Storage\Adapter\Filesystem();
$cacheOptions = new \Zend\Cache\Storage\Adapter\FilesystemOptions(
    array('cache_dir'=>__DIR__ . '/data/')
);
$cache->setOptions($cacheOptions);
$cachePlugin = new \Zend\Cache\Storage\Plugin\ExceptionHandler();
$cachePluginOptions = new \Zend\Cache\Storage\Plugin\PluginOptions(
    array('throw_exceptions'=>false)
);
$cachePlugin->setOptions($cachePluginOptions);
$cache->addPlugin($cachePlugin);
$cache->addItem('ma-cle','ma valeur');
```

Aucune exception n'est lancée ici, mais la valeur de notre cache n'aura pas été mise à jour. Si nous souhaitons nous assurer de la bonne suppression du cache pour la mise à jour des données sans nous soucier des exceptions, il est nécessaire d'ajouter l'instruction liée à la suppression :

Suppression avec la gestion des exceptions

```
|| $cache->removeItem('ma-cle');
|| $cache->addItem('ma-cle','ma valeur');
```

- Le plugin de sérialisation

Le plugin de sérialisation permet de sérialiser automatiquement la valeur du cache en définissant un objet de sérialisation de type Zend\Serializer\Adapter :

Utilisation du plugin de sérialisation

```
|| $cache = new \Zend\Cache\Storage\Adapter\Filesystem();
|| $cacheOptions = new \Zend\Cache\Storage\Adapter\FilesystemOptions(
||   array('cache_dir'=>__DIR__ . '/data/')
|| );
|| $cache->setOptions($cacheOptions);
|| $cachePlugin = new \Zend\Cache\Storage\Plugin\Serializer();
|| $cachePluginOptions = new \Zend\Cache\Storage\Plugin\PluginOptions(
||   array('serializer'=>new \Zend\Serializer\Adapter\PhpSerialize())
|| );
|| $cachePlugin->setOptions($cachePluginOptions);
|| $cache->addPlugin($cachePlugin);
|| $cache->addItem('ma-cle',' ma valeur');
```

Le fichier de cache contient maintenant la chaîne de caractères "s:10:" ma valeur";". De nombreux objets de sérialisation sont disponibles dans le framework : Json, PhpCode, PhpSerialize, etc.

Analyse des plugins

Les plugins profitent pleinement des possibilités de gestion d'évènements du framework. Chacun écoute les évènements de l'adaptateur depuis la classe AbstractAdapter qui les notifient avant et après chacune des actions effectuées. Ce fonctionnement permet aux plugins d'effectuer leurs modifications sur l'objet de stockage ou les valeurs de nos caches.

Prenons comme exemple l'ajout d'un couple clé/valeur :

Zend/Cache/Storage/Adapter/Filesystem.php

```
|| public function addItem($key, $value, array $options = array())
|| {
||   $options = $this->getOptions();
||   if ($options->getWritable() && $options->getClearStatCache()) {
||     clearstatcache();
||   }
||   return parent::addItem($key, $value);
|| }
```

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```

public function addItem($key, $value, array $options = array())
{
    if (!$this->getOptions()->getWritable()) {
        return false;
    }
    $this->normalizeKey($key);
    $args = new ArrayObject(array(
        'key'    => & $key,
        'value'  => & $value,
    ));
    try {
        $eventRs = $this->triggerPre(__FUNCTION__, $args);
        if ($eventRs->stopped()) {
            return $eventRs->last();
        }
        $result = $this->internalAddItem($args['key'], $args['value']);
        return $this->triggerPost(__FUNCTION__, $args, $result);
    } catch (\Exception $e) {
        $result = false;
        return $this->triggerException(__FUNCTION__, $args, $result,
$e);
    }
}

```

La méthode « addItem() » de la classe AbstractAdapter gère l'ajout dans le cache. Une fois les contrôles et les normalisations effectués, la classe mère de notre adaptateur crée un tableau de paramètres contenant les références de notre clé et valeur :

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```

public function addItem($key, $value, array $options = array())
{
    [...]
    $args = new ArrayObject(array(
        'key'    => & $key,
        'value'  => & $value,
    ));
    [...]
}

```

Les variables étant enregistrées dans notre tableau par leur référence, nous aurons donc accès aux valeurs modifiées par les plugins depuis ces références. Une fois cette opération effectuée, un premier évènement de prétraitement est lancé afin de notifier les plugins qui souhaitent agir sur les contenus des variables avant enregistrement :

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```

public function addItem($key, $value, array $options = array())
{
    [...]
    try {
        $eventRs = $this->triggerPre(__FUNCTION__, $args);
        if ($eventRs->stopped()) {
            return $eventRs->last();
        }
        $result = $this->internalAddItem($args['key'], $args['value']);
        return $this->triggerPost(__FUNCTION__, $args, $result);
    }
}

```

```
    } catch (\Exception $e) {
        $result = false;
        return $this->triggerException(__FUNCTION__, $args, $result,
$e);
    }
}
```

Les deux méthodes qui permettent de lancer les évènements de prétraitement et de posttraitement sont « `triggerPre()` » et « `triggerPost()` » :

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```
protected function triggerPre($eventName, ArrayObject $args)
{
    return $this->getEventManager()->trigger(new Event($eventName .
    '.pre', $this, $args));
}
```

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```
protected function triggerPost($eventName, ArrayObject $args,
&$result)
{
$postEvent = new PostEvent($eventName . '.post', $this, $args,
$result);
$eventRs  = $this->getEventManager()->trigger($postEvent);
if ($eventRs->stopped()) {
    return $eventRs->last();
}
return $postEvent->getResult();
}
```

Ces deux méthodes sont responsables du lancement de l'évènement correspondant au nom de la fonction suivi du suffixe « .pre » ou « .post ».

Examinons quels évènements écoutent les plugins et comment ceux-ci répondent aux notifications.

Les plugins s'enregistrent d'abord dès leur passage à l'adaptateur

Zend/Cache/Storage/Adapter/AbstractAdapter.php

```
public function addPlugin(Plugin $plugin)
{
    $registry = $this->getPluginRegistry();
    if ($registry->contains($plugin)) {
        [...]
    }
    $plugin->attach($this->getEventManager());
    $registry->attach($plugin);
    return $this;
}
```

Le registre interne stocke ensuite l'instance du plugin afin de s'assurer de son unicité au sein de l'adaptateur. Ses écouteurs sont ensuite attachés aux différents évènements. Prenons un exemple avec la classe `Serializer` :

Zend/Cache/Storage/Plugin/Serializer.php

```

public function attach(EventManagerInterface $events)
{
    $index = spl_object_hash($events);
    if (isset($this->handles[$index])) {
        throw new Exception\LogicException('Plugin already attached');
    }
    $handles = array();
    $this->handles[$index] = & $handles;
    [...]
    $handles[] = $events->attach('setItem.pre', array($this,
        'onWriteItemPre'));
    $handles[] = $events->attach('setItems.pre', array($this,
        'onWriteItemsPre'));
    $handles[] = $events->attach('addItem.pre', array($this,
        'onWriteItemPre'));
    $handles[] = $events->attach('addItems.pre', array($this,
        'onWriteItemsPre'));
    [...]
    return $this;
}

```

Le plugin Serializer s'attache sur l'évènement « addItem.pre », ce qui lui permet d'agir sur la valeur du cache avant son écriture :

Zend/Cache/Storage/Plugin/Serializer.php

```

public function onWriteItemsPre(Event $event)
{
    $serializer = $this->getOptions()->getSerializer();
    $params     = $event->getParams();
    foreach ($params['keyValuePairs'] as &$value) {
        $value = $serializer->serialize($value);
    }
}

```

L'écouteur récupère l'objet de sérialisation afin de pouvoir agir sur la valeur du cache. Il n'est pas nécessaire de retourner la valeur du cache modifiée, celle-ci étant passée par référence, l'adaptateur aura accès au contenu de la variable maintenant sérialisée.

Il devient alors très simple de créer son propre plugin de cache. Nos plugins personnalisés devront simplement étendre la classe de base Zend\Cache\Storage\Plugin\AbstractPlugin afin de leur permettre de recevoir des options et de s'attacher sur les évènements dont ils ont besoin pour leur travail.

Le composant Zend\Cache\Pattern

Le composant Zend\Cache\Pattern permet de résoudre des problématiques de performances précises. Ce composant permet de cacher les retours de fonctions de « callback » ou d'objet. Cependant, ce composant tient compte du contexte et du résultat du retour des méthodes des objets cachés. En effet, pour un même objet et une même méthode, si cette dernière retourne un résultat différent, un objet de cache sera créé à chaque fois.

Analysons un exemple avec un cache d'objet avec un stockage sous forme de mémoire :

Utilisation du cache d'objet

```
$config = new \Zend\Config\Config(array('cle'=>'valeur'));
$proxyConfig = new \Zend\Cache\Pattern\ObjectCache();
$proxyConfigOptions = new \Zend\Cache\Pattern\PatternOptions(
    array(
        'object' => $config,
        'storage' => new \Zend\Cache\Storage\Adapter\Memory()
    )
);
$proxyConfig->setOptions($proxyConfigOptions);
$value = $proxyConfig->offsetGet('cle');
```

Au premier passage de ces instructions, un fichier de cache propre au retour de la fonction est généré. Les prochains appels à cette méthode de ce même objet pourront alors utiliser ce fichier de cache. Ce cache peut s'avérer pratique lors de traitements longs, récurrents et ayant des résultats identiques. Cependant, il est nécessaire de s'assurer que l'objet appelé fonctionne d'une manière identique à chaque appel au risque de devoir générer des caches à chaque passage. Afin de maîtriser la mise en cache des objets, il est nécessaire d'analyser la création et la gestion du cache du composant.

Lors de l'appel à l'objet Config depuis le composant ObjectCache, celui-ci génère une clé basée sur le contexte de l'objet afin que chaque mise en cache soit reliée à un contexte précis en limitant les effets de bord comme on l'a vu lors de l'exemple précédent. En effet, il est nécessaire de vérifier la composition interne de l'objet, car une méthode ne retournera peut-être pas le même résultat si un de ses attributs est modifié.

Lors de la création de la clé, gérée par la classe parente Zend\Cache\Pattern\CallbackCache qui génère le nom du cache, le composant sérialise l'objet utilisé qu'il pilote :

Zend/Cache/Pattern/CallbackCache.php

```
protected function generateCallbackKey($callback, array $args)
{
    [...]
    $callbackKey = strtolower($callbackKey);
    [...]
    if (isset($object)) {
        ErrorHandler::start();
        try {
            $serializedObject = serialize($object);
        }
        [...]
        $callbackKey .= $serializedObject;
    }
    return md5($callbackKey) . $this->generateArgumentsKey($args);
}
```

La sérialisation de l'objet tient compte des noms et valeurs des attributs. L'objet piloté doit alors être identique d'un appel à l'autre afin d'utiliser le cache généré.

La mise en cache des objets et de leurs valeurs de retour peut s'avérer très pra-

tique lors de longs traitements relativement identiques. La génération de cache et les vérifications que cela engendre nécessitent de ne pas en abuser et de pouvoir justifier l'utilisation d'un tel composant. Une mauvaise utilisation de ce cache peut s'avérer plus pénalisante que performante.

Zend\Console

Le composant Zend\Console offre une API qui permet facilement d'interagir avec le mode CLI de PHP : écriture de ligne, nettoyage de la console de sortie, récupération d'argument ou d'entrée utilisateur, etc.

Utilisation basique de la console

Afin de comprendre le fonctionnement de la console, voici un exemple simple d'affichage de ligne dans le terminal :

Affichage d'une ligne sur le terminal

```
|| $console = Console\Console::getInstance();
|| $console->writeLine('Hello world');
```

La classe Console est un singleton et possède une méthode statique permettant de récupérer l'instance de l'objet de console de sortie. Voici le code de cette méthode :

Zend\Console\Console.php

```
public static function getInstance($forceAdapter = null,
$forceCharset = null)
{
if (static::$instance instanceof Adapter\AdapterInterface) {
    return static::$instance;
}
if ($forceAdapter !== null) {
    if (substr($forceAdapter, 0, 1) == '\\') {
        [...]
    } else {
        $className = __NAMESPACE__.'\\Adapter\\'.$forceAdapter;
    }
    [...]
} else {
    $className = static::detectBestAdapter();
}
static::$instance = new $className();
if ($forceCharset !== null) {
    [...]
    static::$instance->setCharset(new $className());
}
return static::$instance;
}
```

Nous remarquons que la méthode « getInstance() » pilote en réalité un objet du namespace Zend\Console\Adapter qui permet de faire la correspondance avec l'environnement du développeur. La méthode prend en premier paramètre le nom

de l'adaptateur, nous aurions pu alors écrire le nom de l'adaptateur pour forcer l'initialisation (l'environnement de travail courant est Linux) :

Utilisation de l'environnement courant

```
|| $console = Console\Console::getInstance('posix');
|| $console->writeLine('Hello world');
```

Nous aurions pu passer directement par l'adaptateur pour avoir le même résultat :

Utilisation de l'adaptateur

```
|| $console = new Console\Adapter\Posix();
|| $console->writeLine('Hello world');
```

Il est évidemment conseillé de passer par la fabrique de la classe Console afin de ne conserver qu'une seule instance de l'adaptateur. Si aucun argument n'est passé en paramètre de la fabrique, la méthode statique « detectBestAdapter() » permet de connaître l'adaptateur à utiliser.

Une fois la console initialisée, nous pouvons utiliser son API, voici quelques possibilités :

Écriture d'une ligne en couleur

```
|| $console = Console\Console::getInstance();
|| $console->writeLine('Hello world', Console\ColorInterface::GREEN);
```

Nettoyage du terminal

```
|| $console = Console\Console::getInstance();
|| $console->clear();
```

Il est également possible de récupérer la saisie de l'utilisateur :

Récupération de saisie

```
|| $console = Console\Console::getInstance();
|| $line = $console->readLine();
|| echo "Vous avez tape : " . $line . "\n";
```

La classe Console permet aussi de connaître le type de console utilisée :

Type de console

```
|| echo (integer)Console\Console::isConsole();
|| echo (integer)Console\Console::isWindows();
```

Écran de sortie

```
|| 10
```

Les classes d'interactions

Le composant Console possède un sous-composant Zend\Console\Prompt qui permet d'interagir avec l'utilisateur en demandant la saisie d'un texte, entier ou encore de choisir parmi une liste de réponses. Voici une mise en œuvre simple :

Saisie d'un texte

```
$prompt = new Console\Prompt\Line('Entrez votre texte :');
$line = $prompt->show();
echo "Vous avez tape : " . $line . "\n";
```

Si l'on observe le fonctionnement interne de la classe Console\Prompt\Line, nous remarquons que celui-ci se base sur la méthode « `readLine()` » de la classe Console que l'on a présentée plus haut :

Zend\Console\Prompt\Line.php

```
public function show()
{
    do {
        $this->getConsole()->write($this->promptText);
        $line = $this->getConsole()->readLine($this->maxLength);
    } while (!$this->allowEmpty && !$line);
    return $this->lastResponse = $line;
}
```

La console affiche la ligne correspondant à la question que l'on a passée en paramètre et retourne le résultat de la méthode « `readLine()` » en s'assurant que la ligne n'est pas vide.

Il est également possible de récupérer la saisie d'un nombre :

Saisie d'un nombre

```
$prompt = new Console\Prompt\Number('Entrez un nombre :');
$number = $prompt->show();
echo "Vous avez tape : " . $number . "\n";
```

Notons que si la saisie ne correspond pas à un nombre, un message d'erreur est automatiquement affiché :

Zend\Console\Prompt\Number.php

```
public function show()
{
    do {
        $valid = true;
        $number = parent::show();
        if ($number === "" && !$this->allowEmpty) {
            $valid = false;
        } elseif ($number === "") {
            $number = null;
        } elseif (!is_numeric($number)){
            $this->getConsole()->writeLine("$number is not a
number\n");
            $valid = false;
        }
    } while (!$valid);
    return $number;
}
```

```

        }
        [...]
    } while (!$valid);
    if ($number !== null) {
        $number = $this->allowFloat ? (double)$number : (int)$number;
    }
    return $this->lastResponse = $number;
}

```

Comme nous le voyons, la classe s'occupe de gérer les différents types d'erreur comme le type de la saisie mais encore le minimum et maximum que l'on peut passer au constructeur de la classe :

Utilisation du minimum et maximum

```

$prompt = new Console\Prompt\Number('Entrez un nombre :', false,
false, 30, 40);
$number = $prompt->show();
echo "Vous avez tape : " . $number . "\n";

```

Dans l'exemple ci-dessus, l'utilisateur doit alors taper un nombre plus petit ou égal à 40 et plus grand ou égal à 30.

Le composant Zend\Console\Prompt permet aussi la saisie d'une réponse parmi une liste de propositions :

Liste de propositions

```

$prompt = new Console\Prompt\Select('Choisissez une reponse : ',
array(
    '1' => '1er choix',
    '2' => '2e choix',
));
$option = $prompt->show();
echo "Vous avez choisi : " . $option . "\n";

```

Notons qu'il est important de choisir un chiffre ou lettre pour la clé du tableau d'options, car la vérification porte sur les caractères tapés par l'utilisateur et présents en clé du tableau d'options :

Zend\Console\Prompt>Select.php

```

public function show()
{
    $console = $this->getConsole();
    $console->writeLine($this->promptText);
    foreach ($this->options as $k => $v) {
        $console->writeLine(' ' . $k . ') ' . $v);
    }
    $mask = implode("", array_keys($this->options));
    if ($this->allowEmpty) {
        $mask .= "\r\n";
    }
    $this->setAllowedChars($mask);
    $oldPrompt = $this->promptText;
    $this->promptText = 'Pick one option: ';
    $response = parent::show();
    $this->promptText = $oldPrompt;
}

```

```

    || return $response;
    ||
}
```

La méthode « show() » affiche d'abord la liste des options disponibles et marque comme caractères autorisés la concaténation de tous les caractères des clés du tableau :

Zend\Console\Prompt\Select.php

```

public function show()
{
    [...]
    $mask = implode("",array_keys($this->options));
    if ($this->allowEmpty) {
        $mask .= "\r\n";
    }
    $this->setAllowedChars($mask);
    [...]
    $response = parent::show();
    [...]
}
```

La classe Select étend la classe Char qui vérifie si un des caractères présents dans la liste des autorisés existe :

Zend\Console\Prompt\Char.php

```

public function show()
{
    [...]
    if (stristr($this->allowedChars,$char)) {
        if ($this->echo) {
            echo trim($char)."\n";
        } else {
            echo "\n";
        }
        break;
    }
} while (true);
return $this->lastResponse = $char;
}
```

Une autre possibilité est offerte par le framework avec la confirmation d'action et la classe Zend\Console\Prompt\Confirm :

Utilisation de confirmation

```

$prompt = new Console\Prompt\Confirm('Souhaitez-vous continuer ?
(y/n)');
$continue = $prompt->show();
if(!$continue){
    $console->writeLine("Good bye!");
    exit();
}
```

Si la question est personnalisable, le choix de confirmation « y/n » l'est aussi. Le constructeur attend en deuxième et troisième paramètres les deux caractères de confirmation :

Zend\Console\Prompt\Confirm.php

```

public function __construct(
    $promptText = 'Are you sure?',
    $yesChar = 'y',
    $noChar = 'n'
) {
    if ($promptText !== null) {
        $this->setPromptText($promptText);
    }
    if ($yesChar !== null) {
        $this->setYesChar($yesChar);
    }
    if ($noChar !== null) {
        $this->setNoChar($noChar);
    }
}

```

Deux méthodes d'altération sont aussi possibles pour ces caractères. La méthode « `show()` » vérifie ensuite l'égalité avec le caractère correspondant à la confirmation, « `y` » par défaut :

Zend\Console\Prompt\Confirm.php

```

public function show()
{
    $response = parent::show() === $this->yesChar;
    return $this->lastResponse = $response;
}

```

L'abstraction de l'environnement

Lorsque l'on utilise le CLI de PHP, il peut être nécessaire de récupérer les arguments ou variables d'environnement. La classe `Zend\Console\Request` s'occupe de gérer cela pour vous :

Utilisation de Zend\Console\Request

```

$request = new Console\Request();
$firstParameter = $request->getParams()->get(0);
$scriptName = $request->getScriptName();
$environment = $request->env();

```

Notons que le premier paramètre du tableau n'est pas le nom du script, qui est placé dans une variable à part, mais bien les arguments que l'on passe au script. Examinons la construction de cet objet :

Zend\Console\Request.php

```

public function __construct(array $args = null, array $env = null)
{
    if ($args === null) {
        if (!isset($_SERVER['argv'])) {
            [...]
        }
        $args = $_SERVER['argv'];
    }
}

```

```

    }
    if ($env === null) {
        $env = $_ENV;
    }
    if (count($args) > 0) {
        $this->setScriptName(array_shift($args));
    }
    $this->params()->fromArray($args);
    $this->setContent($args);
    $this->env()->fromArray($env);
}

```

Nous remarquons que le nom du script est séparé de la liste de ses arguments, et que la récupération des variables est basée sur les variables PHP `$_ENV` et `$_SERVER`.

Pour finir la présentation de ce composant, nous remarquerons que la classe `Getopt` est toujours présente dans cette version du framework :

Utilisation de Getopt

```

$ getopt = new Console\Getopt(array(
    'nom|n=s'  => 'argument en chaîne de caractère',
));
$ getopt->parse();
$ nom = $ getopt->getOption('nom');
echo "Votre nom est " . $ nom . "\n";

```

Lancement du script php

```
|| console.php --nom=blanchon
```

Le composant Zend\Console rend le code beaucoup plus facile lorsque l'on utilise le CLI de PHP. Tout ce que nous devions coder se trouve parfaitement intégré dans le framework pour une meilleure utilisation de la ligne de commande.

Le router pour la console

Si le composant de console peut parfaitement gérer les interactions avec le terminal pour l'écriture de scripts, il peut également gérer votre application depuis le terminal. En effet, un objet de requête et un router sont implémentés pour gérer les interactions avec la console. Il est possible de définir des routes composées d'arguments pour la ligne de commande afin de piloter l'application. Ce nouveau fonctionnement particulièrement souple va permettre aux développeurs d'avoir une API disponible pour gérer ses actions ou tâches automatisées depuis la ligne de commande. Plus besoin de développement spécifique, votre application est prête à fonctionner depuis votre terminal.

Examinons le code de la fabrique du router afin de comprendre comment est gérée en interne la console :

Zend/Mvc/Service/RouterFactory.php

```
public function createService(ServiceLocatorInterface $serviceLocator, $cName = null, $rName = null)
{
    $config = $serviceLocator->get('Configuration');
    if(
        $rName === 'ConsoleRouter' ||
        ($cName === 'router' && Console:::isConsole())
    ){
        if(isset($config['console']) && isset($config['console']['router'])) {
            $routerConfig = $config['console']['router'];
        } else {
            $routerConfig = array();
        }
        $router = ConsoleRouter::factory($routerConfig);
    } else {
        [...]
    }
    return $router;
}
```

Lors de la création du router, la fabrique vérifie si nous utilisons la console afin de pouvoir fabriquer le router qui lui correspond, de type Zend\Mvc\Router\Console\SimpleRouteStack. Nous remarquons que les routes correspondant à la console doivent être enregistrées sous la clé « ['console']['routes'] ». Voici un exemple de configuration utilisant les routes pour la console :

module.config.php

```
return array(
    'console' => array(
        'router' => array(
            'routes' => array(
                'catch-all' => array(
                    'type' => 'catchall',
                    'options' => array(
                        'route' => '',
                        'defaults' => array(
                            'controller' => 'Application\
Controller\Index',
                            'action'      => 'error',
                        ),
                    ),
                ),
                'cron-application' => array(
                    'type' => 'simple',
                    'options' => array(
                        'route' => '--run-cron-
application=value',
                        'defaults' => array(
                            'controller' => 'Application\
Controller\Index',
                            'action'      => 'cron',
                        ),
                    ),
                ),
            ),
        ),
    ),
);
```

La configuration des routes emploie les arguments à utiliser lors du lancement du script. Examinons le router pour la console afin de voir comment se construisent les routes :

Zend/Mvc/Router/Console/SimpleRouteStack.php

```
protected function init()
{
    $routes = $this->routePluginManager;
    foreach(array(
        'catchall' => __NAMESPACE__ . '\Catchall',
        'simple' => __NAMESPACE__ . '\Simple',
    ) as $name => $class) {
        $routes->setInvokableClass($name, $class);
    }
}
```

L'initialisation du router nous apprend qu'il existe deux types de routes disponibles :

- le type Simple qui permet de définir des arguments en ligne de commande qui serviront de route ;
- le type Catchall qui se vérifie à chaque fois. Si vous définissez une route de type Catchall, elle doit être la dernière dans la pile afin de laisser les traitements spécifiques en premier et celle-là s'exécuter si aucun n'est trouvé.

L'ajout des routes depuis un tableau fait en réalité appel au contrôleur de base Zend\Mvc\Router\SimpleRouteStack que nous connaissons. Le seul point qui diffère est que les routes sont, par défaut, de type Simple si aucun type n'est défini :

Zend/Mvc/Router/Console/SimpleRouteStack.php

```
protected function routeFromArray($specs)
{
    [...]
    if(!isset($specs['type'])) $specs['type'] = 'simple';
    $route = parent::routeFromArray($specs);
    [...]
}
```

Une fois les routes ajoutées, la correspondance se fera en fonction de la comparaison des arguments : arguments obligatoires présents et nombre d'arguments. Il est donc possible de définir une liste d'arguments obligatoires avec d'autres qui ne le sont pas, par exemple :

Utilisation d'arguments facultatifs

```
return array(
    'console' => array(
        'router' => array(
            'routes' => array(
                'cron-application' => array(
                    'type' => 'simple',
                    'options' => array(
                        'route' => '--run-cron-
application=value [--other=value]',

```

```

    }
    if ($env === null) {
        $env = $_ENV;
    }
    if (count($args) > 0) {
        $this->setScriptName(array_shift($args));
    }
    $this->params()->fromArray($args);
    $this->setContent($args);
    $this->env()->fromArray($env);
}

```

Une fois nos routes configurées, il y a quelques modifications à apporter à nos modules et actions. Tout d'abord, nous pouvons gérer le cas où les arguments en ligne de commande ne correspondent à aucune route depuis l'interface Zend\ModuleManager\Feature\ConsoleUsageProviderInterface qui permet de fournir une méthode décrivant l'utilisation de la console depuis le module correspondant :

Module.php

```

class Module implements ConsoleUsageProviderInterface
{
    [...]
    public function getConsoleUsage(AdapterInterface $console)
    {
        return 'Use --run-cron-application=value.' . "\n";
    }
    [...]
}

```

Le descriptif fourni par la méthode « `getConsoleUsage()` » sera affiché dans le terminal si aucune route ne correspond :

Utilisation de mauvais arguments

```

php index.php --unknow-arg
Zend Framework 2.0.0rc2 application.
Usage:
Use --run-cron-application=value.

```

Attention, pour notre exemple, la route de type Catchall a été supprimée. Si celle-ci est conservée, son couple contrôleur/action sera exécuté.

La vue pour la console

Maintenant que nous savons gérer l'aide en ligne de commande, voyons les modifications à apporter à nos actions pour interagir avec la console :

IndexController.php

```

class IndexController extends AbstractActionController
{
    public function cronAction()
    {

```

```
        $param = $this->getRequest()->getParam('cron'); // récupération  
du paramètre  
return new ConsoleModel(array(ConsoleModel::RESULT => 'ok'));  
    }  
}
```

Tout d'abord, il est possible de récupérer les valeurs de nos arguments depuis la méthode « `getParam()` » de l'objet de requête qui est de type `Zend\Console\Request`. Ensuite, la console utilise une vue de type `Zend\View\Model\Console\Model`. Cette vue est assez particulière car elle n'utilise qu'une variable correspondant à la clé « `result` » :

Zend/View/Model/ConsoleModel.php

```
class ConsoleModel extends ViewModel
{
    const RESULT = 'result';
    [...]
    public function setResult($text)
    {
        $this->setVariable(self::RESULT, $text);
        return $this;
    }

    public function getResult()
    {
        return $this->getVariable(self::RESULT);
    }
}
```

La vue n'utilisera que cette variable pour rendre la vue depuis la méthode « `getRe-
sult()` », comme nous le voyons depuis sa stratégie de rendu :

Zend/Mvc/View/Console/DefaultRenderingStrategy.php

```
public function render(MvcEvent $e)
{
[...]
$responseText .= $result->getResult();
$response->setContent(
    $response->getContent() . $responseText
);
[...]
}
```

Comme il peut être un peu fastidieux de toujours créer une instance de l'objet `ConsoleModel` pour construire sa vue, la console bénéficie aussi de ses écouteurs préparateurs de vues. Pour plus d'informations sur la création et préparation des vues, reportez-vous au chapitre consacré aux vues. En effet, ces écouteurs permettent de construire une vue depuis une chaîne de caractères :

Zend/Mvc/View/Console/CreateViewModelListener.php

```
public function createViewModelFromString(MvcEvent $e)
{
    $result = $e->getResult();
    if (!is_string($result)) {
        return;
```

```
    }
    $model = new ConsoleModel;
    $model->setVariable(ConsoleModel::RESULT, $result);
    $e->setResult($model);
}
```

Si l'action retourne une chaîne de caractères, l'écouteur construit et peuple automatiquement la vue avec la bonne variable. Nous pouvons donc réécrire notre action sous la forme simple :

IndexController.php

```
class IndexController extends AbstractActionController
{
    public function cronAction()
    {
        $param = $this->getRequest()->getParam('cron');
        return 'ok';
    }
}
```

La console affichera cette instruction en sortie.

Nous venons de voir l'utilisation de la console au sein de l'application et toutes les possibilités que celle-ci offre en redéfinissant ses propres écouteurs et types de vues, ainsi que ses propres objets de requête ou encore de router. La maîtrise de la console et de ce qui l'entoure permettra aux développeurs d'optimiser la gestion de leurs scripts et tâches automatisées en ligne de commande.

Zend\Crypt

Le composant de cryptage a été réécrit afin de proposer une API plus simple et intuitive. Il est désormais très simple de produire des informations cryptées depuis les algorithmes de cryptage symétrique ou à sens unique.

Hachage à sens unique

Le sous-composant Zend\Crypt\Password permet d'effectuer simplement un cryptage à sens unique indéchiffrable. Le cryptage à sens unique est réalisé depuis la méthode « crypt() » qui permet d'utiliser un salage, identifiant « salt », qui est pris en compte lors du chiffrement afin d'améliorer la sécurité liée au cryptage. En effet, trop d'applications Web se basent uniquement sur la méthode php « md5() », qui est moins sécurisée dans le sens où il est possible de retrouver un mot de passe avec la comparaison de la chaîne cryptée avec d'autres chaînes brouillées depuis la même commande. L'ajout d'un identifiant « salt » permet d'ajouter une couche de sécurité supplémentaire, il ne suffit plus de comparer des chaînes cryptées afin de décoder un mot de passe, il est nécessaire de connaître cet identifiant.

Voici un exemple simple :

Cryptage à sens unique

```

|| $bcrypt = new Zend\Crypt\Password\Bcrypt();
|| $bcrypt->setSalt('mylongsaltforsecurity');
|| $passwordCrypt = $bcrypt->create('mypassword');
|| echo $passwordCrypt;

```

Sortie à l'écran

```
|| $2a$14$bX1sb25nc2FsdGZvcnN1YuRPAXktiNvSUqE4ijz4CgG0yN9J/.hle
```

Par mesure de sécurité, la longueur du « salt » doit être d'au moins 16 octets.

La vérification d'un mot de passe valide se fait depuis la méthode « verify() » :

Vérification du cryptage

```

|| $valid = $bcrypt->verify('mypassword', $passwordCrypt);
|| echo intval($valid);

```

Sortie à l'écran

```
|| 1
```

Le composant Crypt permet aussi le chiffrage à double sens depuis la classe Zend\Crypt\Symmetric\Mcrypt basé sur les fonctions « mcrypt_*() » de PHP.

Hachage symétrique

Le sous-composant Zend\Crypt\Symmetric permet le chiffrage à double sens. Basée sur les méthodes « mcrypt_encrypt() » et « mcrypt_decrypt() », la classe Mcrypt nécessite une clé ainsi qu'un identifiant de salage pour fonctionner.

Cryptage symétrique

```

|| $key = Zend\Crypt\Key\Derivation\Pbkdf2::calc('md5', 'my-key',
|| 'mylongsaltforsecurity', 1, 32);
|| $padding = new Zend\Crypt\Symmetric\Padding\Pkcs7();
|| $mcrypt = new Zend\Crypt\Symmetric\Mcrypt();
|| $mcrypt->setAlgorithm(MCRYPT_DES);
|| $mcrypt->setSalt('mylongsalt');
|| $mcrypt->setKey($key);
|| $mcrypt->setPadding($padding);
|| $datasCrypt = $mcrypt->encrypt('long-datas-to-crypt');
|| $dataDecrypted = $mcrypt->decrypt($datasCrypt);

```

Le même traitement peut être effectué depuis un tableau de configuration :

Cryptage symétrique depuis une configuration en tableau

```

|| $mcrypt = new Zend\Crypt\Symmetric\Mcrypt(array(
||   'algorithm' => MCRYPT_DES,
||   'salt' => 'mylongsalt',
||   'padding' => 'pkcs7',
||   'key' => Zend\Crypt\Key\Derivation\Pbkdf2::calc('md5', 'my-key',
||   'mylongsaltforsecurity', 1, 32),
|| ));

```

```
|| $datasCrypted = $mcrypt->encrypt('long-datas-to-crypt');
|| $dataDecrypted = $mcrypt->decrypt($datasCrypted);
```

Le composant Zend\Crypt facilite grandement l'usage des fonctions natives de cryptage pour les opérations les plus communes au sein du Zend Framework 2.

Zend\Form

Les formulaires de la première version ont souvent fait débat au sein des développeurs : difficiles à appréhender et à mettre en œuvre, avec l'utilisation des décorateurs par exemple. L'équipe de développement du Zend Framework a souhaité revoir complètement l'organisation et la structure du composant.

Le formulaire, les fielsets et les éléments

Le Zend Framework 2 modifie complètement l'architecture du composant de formulaire. L'élément de base et au centre de tout le composant est le Zend\Form\Element. L'objet Element possède des attributs et des messages, le nom de l'élément sera lui-même un attribut avec pour clé « name ». Un élément est décrit par l'interface Zend\Form\ElementInterface qui lui est associée :

Zend/Form/ElementInterface.php

```
interface ElementInterface
{
    public function setName($name);
    public function getName();
    public function setOptions($options);
    public function getOptions();
    public function getOption($option);
    public function setAttribute($key, $value);
    public function getAttribute($key);
    public function hasAttribute($key);
    public function setAttributes($arrayOrTraversable);
    public function getAttributes();
    public function setValue($value);
    public function getValue();
    public function setLabel($label);
    public function getLabel();
    public function setMessages($messages);
    public function getMessages();
}
```

La classe Zend\Form\Fieldset hérite de cette classe de base pour la représentation d'un ensemble d'éléments. Afin de gérer la liste des éléments, les objets Fieldset disposent de méthodes représentées décrites par l'interface Zend\Form\FieldsetInterface :

Zend/Form/FieldsetInterface.php

```
interface FieldsetInterface extends Countable, IteratorAggregate,
ElementInterface
{
    public function add($elementOrFieldset, array $flags = array());
```

```

public function has($elementOrFieldset);
public function get($elementOrFieldset);
public function remove($elementOrFieldset);
public function setPriority($elementOrFieldset, $priority);
public function getElements();
public function getFieldsets();
public function populateValues($data);
public function setObject($object);
public function getObject();
public function allowObjectBinding($object);
public function setHydrator(HydratorInterface $hydrator);
public function getHydrator();
public function bindValues(array $values = array());
public function allowValueBinding();
}

```

Nous remarquons qu'une méthode « `getFieldsets()` » est disponible. En effet, les objets de type `Fieldset` peuvent contenir d'autres objets du même type. Ce système de conteneur permet l'ajout d'éléments de même nom dans des objets de type `Fieldset` différents, cette gestion s'approche du concept d'espaces de noms que l'on connaît. Chaque nom d'élément ajouté à un conteneur de type `Fieldset` est englobé dans l'espace de nom du conteneur.

Un formulaire doit pouvoir contenir plusieurs éléments et avoir des attributs tels que l'URL d'action ou la méthode de passage des éléments à l'action, il hérite donc de la classe `Fieldset`. La classe de formulaire `Zend\Form\Form` hérite de la classe `Fieldset`. Récapitulons le rôle de chacune de ces classes :

- `Zend\Form\Element` : représentation de base, définit des attributs et des messages ;
- `Zend\Form\Fieldset` : hérite de `Zend\Form\Element`, ce conteneur représente un ensemble d'objets de type `Element` et/ou de type `Fieldset` ;
- `Zend\Form\Form` : hérite de `Zend\Form\BaseForm` et gère la fabrique de formulaires, `fieldsets` et éléments. La classe s'occupe de la gestion du composant `Zend\InputFilter` du formulaire, qui contient filtres et validateurs.

Afin de pouvoir créer et peupler facilement les formulaires, des composants d'hydratations et une fabrique sont disponibles.

Les fabriques et hydrateurs

La fabrique utilisée par défaut dans le composant `Zend\Form` est de type `Zend\Form\Factory`. La fabrique permet la création d'objets de formulaires, `fieldsets` et éléments depuis un tableau de description passé en paramètre :

Création d'un élément depuis la fabrique

```

$fabriche = new Zend\Form\Factory();
$element = $fabriche->create(array(
    'type' => 'Zend\Form\Element',
    'name' => 'nom',
    'attributes' => array(
        'type' => 'text',
    )
});

```

```

    'label' => 'Nom :',
));

```

Il est également possible d'utiliser la fabrique d'éléments sans devoir indiquer le type d'objet que l'on souhaite créer :

Création d'un élément depuis la fabrique

```

$fabrique = new Zend\Form\Factory();
$element = $fabrique->createElement(array(
    'name' => 'nom',
    'attributes' => array(
        'type' => 'text',
        'label' => 'Nom :',
),
));

```

La fabrique est utilisée lors de la création d'éléments depuis la configuration en tableau au sein d'un objet de formulaire :

Formulaire utilisant la fabrique

```

class SimpleForm extends Form
{
public function __construct()
{
    parent::__construct();
    $this->setAttribute('method', 'post');
    $this->add(array(
        'name' => 'nom',
        'attributes' => array(
            'type' => 'text',
            'label' => 'Nom :',
),
));
}
}

```

AVEC LE ZEND FRAMEWORK 1

La méthode « `init()` » qui est appelée après la construction du formulaire permet d'initialiser les champs de formulaire. Cette méthode n'existe plus et la construction se fait directement depuis le constructeur.

Voici la méthode « `add()` » du composant `Zend\Form` qui permet d'ajouter les éléments :

Zend/Form/Form.php

```

public function add($elementOrFieldset, array $flags = array())
{
if (is_array($elementOrFieldset)
    || ($elementOrFieldset instanceof Traversable &&
    !$elementOrFieldset instanceof ElementInterface)
) {
    $factory = $this->getFormFactory();
}
}

```

```

        $elementOrFieldset = $factory->create($elementOrFieldset);
    }

    return parent::add($elementOrFieldset, $flags);
if ($elementOrFieldset instanceof Fieldset && $elementOrFieldset-
>useAsBaseFieldset()) {
    $this->baseFieldset = $elementOrFieldset;
}
return $this;
}

```

La fabrique intervient afin de créer l'objet demandé si le paramètre correspondant à l'élément est un tableau. Notons que nous n'avons pas passé le type d'élément à créer en paramètre, la fabrique utilise le type Zend\Form\Element par défaut :

Zend/Form/Factory.php

```

public function create($spec)
{
$spec = $this->validateSpecification($spec, __METHOD__);
$type = isset($spec['type']) ? $spec['type'] : 'Zend\Form\Element';
[...]
if (self::isSubclassOf($type, 'Zend\Form\ElementInterface')) {
    return $this->createElement($spec);
}
[...]
}

```

Il est aussi possible de créer un élément directement sans utiliser la fabrique :

Création d'un élément

```

$element = new Zend\Form\Element\Text();
$element->setName('nom');
$element->setAttribute('label', 'Nom ');

```

AVEC LE ZEND FRAMEWORK 1

Les classes d'éléments typées n'existent plus. La création d'une classe personnalisée héritant de Zend\Form\Element permet de retrouver ce comportement.

Une fois le formulaire construit, avec ou sans la fabrique, il doit être peuplé afin de pouvoir valider les informations injectées. Deux méthodes peuvent être utilisées pour l'injection de données :

- depuis un tableau et la méthode « setData() » ;
- depuis un objet auquel on affecte un objet de type Zend\Stdlib\Hydrator\HydratorInterface afin de permettre l'extraction et l'injection de données.

La méthode « setData() » est simple à utiliser, il suffit de lui passer un tableau en paramètre et les données validées seront retournées sous la forme d'un tableau de valeurs :

Utilisation de la méthode « setData() »

```
$form = new SimpleForm();
$filter = new SimpleFormFilter();
$form->setInputFilter($filter);
$form->setData(
    array(
        'nom' => 'Vincent',
        'sujet' => 'information',
        'message' => 'Ceci est un <strong>message</strong> depuis Zend\
Form',
    )
);
$valid = $form->isValid();
$datas = $form->getData();
```

Le formulaire SimpleForm est un formulaire composé de trois éléments de type « text ». Afin de valider les valeurs passées en paramètres, le formulaire a besoin d'un élément de type Zend\InputFilter\InputFilterInterface contenant filtres et validateurs à utiliser, nous étudierons ce composant à la prochaine section.

Les données à valider sont ensuite fournies à la méthode « setData() » et récupérées depuis la méthode « getData() » une fois ces dernières validées et filtrées.

Le composant de formulaire laisse la possibilité de lui fournir un objet enveloppant les données à valider, ce qui aura l'avantage de pouvoir récupérer les données filtrées et validées depuis ce même objet. Cependant, afin d'indiquer au formulaire comment extraire et peupler cet objet, il est impératif de lui fournir un objet d'hydratation afin qu'il sache quelle stratégie adopter pour le traitement de l'objet.

Trois hydrateurs d'objet sont disponibles dans le namespace Zend\Stdlib\Hydrator :

- **ArraySerializable** : l'extraction des données se fait depuis la méthode « getArrayCopy() » et l'hydratation depuis la méthode « exchangeArray() » ou « populate() ». L'utilisation d'un objet héritant de \ArrayObject permet la prise en charge automatique de ces deux premières méthodes ;
- **ObjectProperty** : basé sur les propriétés publiques de l'objet, il suffit de définir une propriété publique pour chaque élément que l'on souhaite extraire ou hydrater ;
- **ClassMethods** : basé sur les méthodes d'altération, cet hydrateur permet d'interagir facilement avec les objets que l'on a créés.

Pour plus de renseignements sur les hydrateurs, reportez-vous à la section qui leur est consacrée dans le chapitre sur la bibliothèque standard du framework.

Prenons un exemple avec l'hydrateur ClassMethods, hydrateur que j'ai eu l'occasion d'implémenter dans le framework. Le formulaire utilisé est le formulaire SimpleForm qui contient trois éléments (« nom », « prenom », « email ») et les filtres et validateurs sont stockés dans la classe SimpleFormFilter qui gère uniquement la suppression des espaces avant et après les valeurs, ainsi que la suppression des tags :

Gestion des filtres et validateurs

```

class SimpleFormFilter extends InputFilter
{
    public function __construct()
    {
        $this->add(array(
            'name'      => 'nom',
            'required'  => true,
            'filters'   => array(
                array(
                    'name'      => 'Zend\Filter\StripTags',
                ),
                array(
                    'name'      => 'Zend\Filter\StringTrim',
                ),
            ),
        ));
        $this->add(array(
            'name'      => 'prenom',
            'required'  => true,
            'filters'   => array(
                array(
                    'name'      => 'Zend\Filter\StripTags',
                ),
                array(
                    'name'      => 'Zend\Filter\StringTrim',
                ),
            ),
        ));
        $this->add(array(
            'name'      => 'email',
            'required'  => true,
            'filters'   => array(
                array(
                    'name'      => 'Zend\Filter\StripTags',
                ),
            ),
            'validators' => array(
                array(
                    'name'      => 'Zend\Validator\EmailAddress',
                ),
            ),
        ));
    }
}

```

Notre objet de données devra donc contenir les méthodes correspondant aux champs du formulaire :

Objet qui sera utilisé par l'hydrateur

```

class SimpleObject
{
    protected $nom;
    protected $prenom;
    protected $email;

    public function __construct($nom, $prenom, $email)
    {
        $this->nom = $nom;
        $this->prenom = $prenom;
        $this->email = $email;
    }
}

```

```
    }

    public function getNom()
    {
        return $this->nom;
    }
    [...]
}
```

Voici un exemple d'utilisation de ce filtre et du peuplement depuis un objet et un hydrateur :

Utilisation d'un hydrateur

```
$form = new SimpleForm();
$filter = new SimpleFormFilter();
$form->setInputFilter($filter);
$form->setHydrator(new Zend\Stdlib\Hydrator\ClassMethods());
$object = new SimpleObject('<strong>blanchon</strong>', 'vincent',
    'blanchon.vincent@gmail.com');
$form->bind($object);
$valid = $form->isValid();
$data = $form->getData();
$nom = $data->getNom();
echo $nom;
```

Sortie à l'écran

```
|| blanchon
```

Il est également possible de créer ses propres hydrateurs en implémentant l'interface Zend\Stdlib\Hydrator\HydratorInterface qui définit deux méthodes :

Zend/Stdlib/Hydrator/HydratorInterface.php

```
interface HydratorInterface
{
    public function extract($object);
    public function hydrate(array $data, $object);
}
```

La méthode « extract() » extrait les attributs avec leurs valeurs alors que la méthode « hydrate() » injecte les données à l'objet.

Les objets contenant les filtres et validateurs étendent la classe InputFilter qui fournit une API permettant la validation et le filtrage des données.

Le composant Zend\InputFilter

Basé sur des chaînes de filtres et de validateurs, le composant Zend\InputFilter\ InputFilter permet le filtrage et la validation de données. Voici un exemple de sa mise en œuvre :

Mise en œuvre du composant Zend\InputFilter\InputFilter

```
$inputFilter = new Zend\InputFilter\InputFilter();
$input = new Zend\InputFilter\Input('data');

$validators = new Zend\Validator\ValidatorChain();
$validators->addByName('emailaddress', array());
$input->setValidatorChain($validators);

$filters = new Zend\Filter\FilterChain();
$filters->attachByName('striptags');
$filters->attachByName('stringtrim');

$input->setFilterChain($filters);
$inputFilter->add($input);
$inputFilter->setData(array('data' => ' <strong>blanchon<strong>.
vincent@gmail.com'));

$valid = $inputFilter->isValid();
```

Chaque donnée à traiter doit être contenue dans un objet de type Input qui définit filtres et validateurs. Les filtres agissent sur la donnée avant de la valider :

Zend/InputFilter/Input.php

```
class Input implements InputInterface
{
    [...]
    public function getValue()
    {
        $filter = $this->getFilterChain();
        return $filter->filter($this->value);
    }

    public function isValid($context = null)
    {
        $this->injectNotEmptyValidator();
        $validator = $this->getValidatorChain();
        $value     = $this->getValue();
        return $validator->isValid($value, $context);
    }
    [...]
}
```

Le composant dispose aussi d'une fabrique qui permet la création de filtres et validateurs depuis un tableau de paramètres :

Utilisation de la fabrique du Zend\InputFilter\InputFilter

```
$inputFilter = new Zend\InputFilter\InputFilter();
$inputFilter->add(array(
    'name'      => 'data',
    'required'  => true,
    'validators' => array(
        array(
            'name'    => 'emailaddress',
        ),
    ),
    'filters'   => array(
        array(
```

```

        'name'      => 'striptags',
    ),
    array(
        'name'      => 'stringtrim',
    ),
),
));
$inputFilter->setData(array('data' => ' <strong>blanchon<strong>.
vincent@gmail.com'));
$valid = $inputFilter->isValid();

```

La fabrique utilisée par défaut est la classe Zend\InputFilter\Factory qui permet la création d'objets de type Input depuis sa méthode « createInput() », utilisée par la classe InputFilter :

Zend/InputFilter/InputFilter.php

```

public function add($input, $name = null)
{
if (is_array($input)
    || ($input instanceof Traversable && !$input instanceof
InputFilterInterface)
) {
    $factory = $this->getFactory();
    $input = $factory->createInput($input);
}
return parent::add($input, $name);
}

```

Dans l'exemple précédent, l'objet de type InputFilter utilise la fabrication d'objets Input depuis la description en tableau. Il est également possible de ne pas séparer les filtres et validateurs des éléments en implémentant l'interface Zend\InputFilter\InputProviderInterface dans la classe de l'élément, ce qui l'oblige à définir la méthode « getInputSpecification() », descriptif des filtres et validateurs à utiliser. Reprenons l'exemple du formulaire SimpleForm en ajoutant des classes pour chacun des éléments. Voici la classe associée à l'e-mail :

Création de filtres et validateurs au sein du formulaire

```

class Email extends Element implements InputProviderInterface
{
public function getInputSpecification()
{
    return array(
        'name' => $this->getName(),
        'attributes' => array(
            'type'  => 'text',
            'label' => 'Email :',
        ),
        'required' => true,
        'filters' => array(
            array('name' => 'Zend\Filter\StringTrim'),
            array('name' => 'Zend\Filter\StripTags'),
        ),
        'validators' => array(
            array('name' => 'Zend\Validator\EmailAddress'),
        ),
    );
}
}

```

L'implémentation du formulaire se fait légèrement différemment :

Implémentation du formulaire avec filtres et validateurs

```
$form = new SimpleForm();
$form->setInputFilter(new Zend\InputFilter\InputFilter);
$form->add(new Nom('nom'));
$form->add(new Prenom('prenom'));
$form->add(new Email('email'));
$form->setHydrator(new Zend\Stdlib\Hydrator\ClassMethods());
$object = new SimpleObject('blanchon', 'vincent', '
<strong>blanchon<strong>.vincent@gmail.com');
$form->bind($object);
$valid = $form->isValid();
$data = $form->getData();
$email = $data->getEmail();
```

Le composant de formulaire offre plusieurs manières simples de mettre en place des filtres et validateurs pour les éléments, ce qui permet à l'utilisateur de séparer les différentes couches afin de les réutiliser entre les différents formulaires de l'application.

Les aides de vue

Une fois le formulaire construit, il est nécessaire de le rendre dans la vue. Il existe dans le namespace Zend\Form\View\Helper des aides de vue capables de rendre un élément de type Zend\Form\ElementInterface. Prenons l'exemple de l'aide FormRow qui permet le rendu des éléments d'un formulaire de n'importe quel type. Cette aide de vue se base sur les autres aides qu'elle utilise en fonction du type d'élément et du besoin :

Rendu de formulaire

```
<?php
$form = $this->form;
$form->prepare();
$form->setAttribute('action', $this->url('my/url'));
$form->setAttribute('method', 'post');
echo $this->form()->openTag($form);

?>
<?php
echo $this->formRow($form->get('nom'));
echo $this->formRow($form->get('prenom'));
echo $this->formRow($form->get('email'));
?>
<?php echo $this->form()->closeTag() ?>
```

Analysons le fonctionnement de l'aide de vue capable de rendre les éléments :

Zend/Form/View/Helper/FormRow.php

```
public function render(ElementInterface $element)
{
[...]
$labelHelper      = $this->getLabelHelper();
$elementHelper    = $this->getElementHelper();
$elementErrorsHelper = $this->getElementErrorsHelper();
```

```

[...]
$label           = $element->getLabel();
[...]
$elementErrors  = $elementErrorsHelper->render($element);
[...]
$elementString  = $elementHelper->render($element);
if (!empty($label)) {
    [...]
    if ($this->renderErrors) {
        $markup .= $elementErrors;
    }
}
} else {
    if ($this->renderErrors) {
        $markup = $elementString . $elementErrors;
    } else {
        $markup = $elementString;
    }
}
return $markup;
}

```

L'aide de vue récupère les différentes aides de vue dont elle a besoin pour rendre tous les éléments (aide pour le label, l'élément et les erreurs) et construit ensuite le code HTML de l'élément avant de le renvoyer. Nous pourrions également rendre manuellement chacun des éléments grâce à chacune des aides de vue. Voici à quoi ressemblerait alors notre code :

Rendu depuis les aides de vue du formulaire

```

<?php
$form = $this->form;
$form->prepare();
$form->setAttribute('action', $this->url('my/url'));
$form->setAttribute('method', 'post');
echo $this->form()->openTag($form);
?>
<?php
echo $this->formLabel($form->get('nom'));
echo $this->formElement($form->get('nom'));
echo $this->formElementErrors($form->get('nom'));
?>
<?php
echo $this->formLabel($form->get('prenom'));
echo $this->formElement($form->get('prenom'));
echo $this->formElementErrors($form->get('prenom'));
?>
<?php
echo $this->formLabel($form->get('email'));
echo $this->formElement($form->get('email'));
echo $this->formElementErrors($form->get('email'));
?>
<?php echo $this->form()->closeTag() ?>

```

L'aide de vue FormElement permet de rendre n'importe quel élément car cette aide utilise celle qui est implémentée par l'élément correspond, comme nous le voyons dans la méthode de rendu :

Zend/Form/View/Helper/FormElement.php

```

public function render(ElementInterface $element)
{
    [...]
    $type = $element->getAttribute('type');
    if ('checkbox' == $type) {
        $helper = $renderer->plugin('form_checkbox');
        return $helper($element);
    }
    if ('color' == $type) {
        $helper = $renderer->plugin('form_color');
        return $helper($element);
    }
    if ('date' == $type) {
        $helper = $renderer->plugin('form_date');
        return $helper($element);
    }
    [...]
    $helper = $renderer->plugin('form_input');
    return $helper($element);
}

```

Le type de l'élément est récupéré pour obtenir l'aide de vue qui lui correspond. Nous aurions donc également pu remplacer l'aide de vue FormElement de notre code HTML par l'aide de vue correspondant au type du champ.

Zend\Mail

Le composant pour la gestion des e-mails a été réécrit en profondeur. Chaque tâche est maintenant clairement séparée et de nouvelles fonctionnalités ont été ajoutées.

La classe Zend\Mail\Message permet au développeur d'encapsuler dans un objet tout ce qui va correspondre au message à envoyer : sujet de l'e-mail, contenu de l'e-mail, adresse du destinataire, adresse de réponse, etc. Voici un exemple de création de message :

Création de message

```

$message = new Mail\Message();
$message->addTo('contact@au-coeur-de-zend-framework-2.fr');
$message->addReplyTo('blanchon.vincent@gmail.com', 'BLANCHON
Vincent');
$message->setFrom('blanchon.vincent@gmail.com', 'BLANCHON Vincent');
$message->setSubject('Question sur le livre');
$message->setBody('Comment puis-je acheter le livre ?');

```

Cependant, comme les tâches sont découpées au maximum, il est possible d'utiliser une meilleure encapsulation pour la gestion des adresses e-mail :

Création de message avec encapsulation des adresses

```

|| $message = new Mail\Message();
|| $message->addTo(
||     new Mail\Address('contact@au-coeur-de-zend-framework-2.fr')
|| );
|| $message->addReplyTo(
||     new Mail\Address('blanchon.vincent@gmail.com', 'BLANCHON
|| Vincent')
|| );
|| $message->setFrom(
||     new Mail\Address('blanchon.vincent@gmail.com', 'BLANCHON
|| Vincent')
|| );
|| $message->setSubject('Question sur le livre');
|| $message->setBody('Comment puis-je acheter le livre ?');

```

De même pour le corps du message de l'e-mail, il est possible de l'encapsuler depuis les classes Mime afin de gérer son type :

Encapsulation du type du corps du message

```

|| $message = new Mail\Message();
|| $message->addTo(
||     new Mail\Address('contact@au-coeur-de-zend-framework-2.fr')
|| );
|| $message->addReplyTo(
||     new Mail\Address('blanchon.vincent@gmail.com', 'BLANCHON
|| Vincent')
|| );
|| $message->setFrom(
||     new Mail\Address('blanchon.vincent@gmail.com', 'BLANCHON
|| Vincent')
|| );
|| $message->setSubject('Question sur le livre');
|| $body = new Mime\Message();
|| $text = new Mime\Part('Comment puis-je acheter le livre ?');
|| $text->type = 'text/plain';
|| $body->addPart($text);
|| $message->setBody($body);

```

Enfin, il est également possible de vérifier si le message est bien valide :

Vérification de la validité du message

```
|| $message->isValid();
```

Lors de la vérification du message, l'objet se contente de vérifier la liste des adresses d'envoi :

Zend/Mail/Message.php

```

|| public function isValid()
|| {
||     $from = $this->getFrom();
||     if (!$from instanceof AddressList) {
||         return false;
||     }

```

```

    || return (bool) count($from);
    ||
}
```

Une fois le message configuré et validé, nous pouvons utiliser un objet de transport afin d'envoyer le message :

Envoi du message

```

$smtp = new Mail\Transport\Smtp();
$connection = $smtp->plugin('login', array(
    'host' => 'smtp.gmail.com',
    'username' => 'username',
    'password' => 'password',
));
$connection->connect()->auth();
$smtp->setConnection($connection);
$smtp->send($message);
```

Notons que la classe Zend\Mail\Transport\Smtp possède son propre gestionnaire de plugins qui lui permet un accès facilité aux différents types disponibles :

Zend/Mail/Protocol/SmtpPluginManager.php

```

class SmtpPluginManager extends AbstractPluginManager
{
protected $invokableClasses = array(
    'crammd5' => 'Zend\Mail\Protocol\Smtp\Auth\Crammd5',
    'login'    => 'Zend\Mail\Protocol\Smtp\Auth>Login',
    'plain'    => 'Zend\Mail\Protocol\Smtp\Auth\Plain',
    'smtp'     => 'Zend\Mail\Protocol\Smtp',
);
[...]
}
```

Le composant de gestion de mails permet aussi la gestion des e-mails depuis de nombreux formats de stockage existants : gestion de la sauvegarde de courriel depuis la structure de répertoire « maildir », gestion du format de stockage ouvert « mbox » ou encore la gestion des e-mails depuis les protocoles « pop3 » ou « imap ».

Voici un exemple de la gestion du protocole « pop3 » :

Gestion du protocole « pop3 »

```

$mail = new Mail\Storage\Pop3(array(
    'host' => 'pop.gmail.com',
    'user' => 'username',
    'password' => 'password',
    'ssl' => true,
));
$ids = $mail->getUniqueId();
```

De même, il est possible de gérer facilement ses e-mails depuis le format ouvert « mbox » :

Gestion des e-mails avec le format « mbox »

```

$mail = new Mail\Storage\Mbox(
    array('filename' => 'path/to/my/mbox/INBOX')
);
$message = $mail->getMessage(3);
$content = $message->getContent();
echo "Voici le contenu de l'e-mail : " . $content;

```

Le composant Mail a enrichi la liste de ses fonctionnalités et en fait un composant incontournable du framework. Sa structure revue en profondeur en fait un composant que l'on peut étendre facilement pour répondre aux différentes problématiques que l'on peut rencontrer avec la messagerie et gestion des e-mails.

Zend\Session

Le composant de gestion des sessions a été totalement réécrit dans cette version du framework. La gestion par espace de nom est conservée afin de pouvoir séparer les différents conteneurs de session. Voici maintenant comment implémenter une session avec un espace de nom :

Création d'une session avec son espace de nom

```

$session = new \Zend\Session\Container('monnamespace');
$session->offsetSet('attribut', 'valeur');

```

Examinons en détail le constructeur du conteneur.

Zend/Session/Container.php

```

public function __construct($name = 'Default', Manager $manager =
null)
{
    if (!preg_match('/^([a-z][a-z0-9_\\\\]+)$/i', $name)) {
        [...]
    }
    $this->name = $name;
    $this->setManager($manager);
    parent::__construct(array(), ArrayObject::ARRAY_AS_PROPS);
    $this->getManager()->start();
}

```

La classe Container hérite de la classe ArrayObject et appelle le constructeur parent avec le paramètre « ARRAY_AS_PROPS », ce qui nous permet d'accéder aux clés de l'objet comme des attributs :

Utilisation d'un conteneur de session

```

$session = new \Zend\Session\Container('monnamespace');
$session->attribut = 'valeur';

```

Le constructeur initialise un gestionnaire de sessions qui permet le pilotage de la session dans sa globalité : destruction, identifiant de session, régénération de session, etc. Afin de construire cet espace de nom dans la session courante, le constructeur demande avant au gestionnaire de sessions de démarrer son méca-

nisme si celui-ci ne l'a pas déjà été. Nous expliquerons en détail le gestionnaire de sessions après la notion de conteneur.

Dans un conteneur de session, c'est uniquement lors de la modification d'une propriété que celle-ci est créée :

Zend/Session/Container.php

```
|| public function offsetSet($key, $value)
|| {
||     $this->expireKeys($key);
||     $storage = $this->verifyNamespace();
||     $name    = $this->getName();
||     $storage[$name][$key] = $value;
|| }
```

Zend/Session/Container.php

```
|| protected function verifyNamespace($createContainer = true)
|| {
||     $storage = $this->getStorage();
||     $name    = $this->getName();
||     if (!isset($storage[$name])) {
||         if (!$createContainer) {
||             return;
||         }
||         $storage[$name] = $this->createContainer();
||     }
||     if (!is_array($storage[$name]) && !$storage[$name] instanceof
||     ArrayObject) {
||         [...]
||     }
||     return $storage;
|| }
```

Lorsqu'une propriété est modifiée, si le conteneur courant n'existe pas dans le gestionnaire de stockages, alors celui-ci est créé. La création à la demande permet d'optimiser les ressources en allouant de l'espace uniquement en cas de besoin. L'espace de stockage par défaut est un objet de type Zend\Session\Storage\SessionStorage qui permet de piloter directement la variable « `$_SESSION` » de PHP.

Le test d'existence de l'espace de nom et la création de son conteneur associé sont effectués depuis la méthode « `verifyNamespace()` ». Le couple attribut/valeur est ensuite ajouté à notre objet de stockage.

Le gestionnaire de sessions, qui permet de piloter l'objet de session, est accessible depuis n'importe quel conteneur de session depuis l'instruction :

Accès au gestionnaire de sessions

```
|| $session = new \Zend\Session\Container('namespace');
|| $manager = $session->getManager();
```

Le gestionnaire est aussi accessible depuis la récupération du gestionnaire par défaut :

accès au gestionnaire de sessions

```
|| $manager = \Zend\Session\Container::getDefaultManager();
```

En effet, la méthode « `getManager()` » et « `getDefaultManager()` » retourne le même objet, car si l'on observe le constructeur du conteneur, nous remarquons que le gestionnaire est construit à cet endroit à partir du gestionnaire par défaut :

Zend/Session/Container.php

```
public function __construct($name = 'Default', Manager $manager = null)
{
    [...]
    $this->setManager($manager);
    [...]
}
```

La méthode « `setManager()` » utilise comme instance le gestionnaire par défaut si nous ne lui avons fourni aucun objet de type Manager :

Zend/Session/Container.php

```
protected function setManager(Manager $manager = null)
{
    if (null === $manager) {
        $manager = self::getDefaultManager();
        [...]
    }
    $this->manager = $manager;
    return $this;
}
```

Cependant, dans le cas où nous fournissons notre gestionnaire personnalisé, les deux objets seront différents. Le gestionnaire de sessions doit respecter un contrat avec l'interface Zend\Session\ManagerInterface :

Zend/Session/ManagerInterface.php

```
interface ManagerInterface
{
    [...]
    public function start();
    public function destroy();
    [...]
    public function setId($id);
    public function getId();
    public function regenerateId();
    public function rememberMe($ttl = null);
    public function forgetMe();
    [...]
}
```

Nous comprenons que c'est ce gestionnaire qui va permettre de démarrer, arrêter ou régénérer l'identifiant de la session.

Le gestionnaire Zend\Session\SessionManager étend Zend\Session\AbstractManager qui définit entre autres une méthode « `getConfig()` » donnant accès à la configuration de la session sous la forme d'un objet de type Zend\Session\Configuration\SessionConfiguration, ainsi qu'une méthode « `setSaveHandler()` », qui offre la possibilité de modifier les fonctions de gestion de stockage des sessions.

Comme pour chacun des composants du framework, la gestion des sessions est divisée en de multiples classes où chacune est responsable d'une tâche précise (configuration, stockage, conteneur). Il est possible de remplacer chaque sous-composant par un composant personnalisé, à condition que celui-ci respecte le contrat défini par l'interface liée à ce sous-composant. Ainsi, si nous souhaitons étendre le composant de session, nous devrons utiliser les interfaces :

- Zend\Session\ManagerInterface qui définit un contrat avec le gestionnaire de sessions ;
- Zend\Session\ConfigurationInterface qui définit un contrat avec le gestionnaire d'options de nos sessions ;
- Zend\Session\SaveHandler\SaveHandlerInterface qui définit un contrat avec le gestionnaire de stockages des sessions ;
- Zend\Session\Storage\StorageInterface qui définit un contrat avec le gestionnaire de stockages des valeurs en session.

Nos composants personnalisés peuvent alors être injectés depuis une fabrique personnalisée ou depuis les méthodes d'altération du composant de session.

Zend\Stdlib

Le composant Zend\Stdlib est une bibliothèque de classes permettant de résoudre les problèmes récurrents que l'on rencontre sur les projets Web. Ce composant se veut être complémentaire à la SPL (bibliothèque standard de PHP) que l'on connaît. Les classes de ce composant s'appuient aussi souvent sur ce standard afin d'enrichir un concept ou objet existant. Le framework s'appuie beaucoup sur ce composant qui permet d'introduire une certaine homogénéité dans le framework. Nous allons présenter dans ce chapitre quelques classes que l'on retrouvera beaucoup au sein du framework et que vous pourrez utiliser dans de vos projets.

La classe AbstractOptions

La classe abstraite AbstractOptions fournit des méthodes de base aux classes permettant de gérer la configuration d'un composant. Cette classe offre un mécanisme d'initialisation qui, à partir d'un tableau d'options, permet de distribuer aux méthodes d'altérations, les données qu'elles doivent recevoir. Afin de pouvoir automatiser cette tâche, la classe formate les noms de clés du tableau de paramètres en nom de méthode correspondant à cette clé en la préfixant du mot « set » :

Zend/Stdlib/AbstractOptions.php

```
public function __construct($options = null)
{
    if (null !== $options) {
        $this->setFromArray($options);
    }
}
```

Zend/Stdlib/AbstractOptions.php

```

public function setFromArray($options)
{
    if (!is_array($options) && !$options instanceof Traversable) {
        [...]
    }
    foreach ($options as $key => $value) {
        $this->__set($key, $value);
    }
}

```

Zend/Stdlib/AbstractOptions.php

```

protected function assembleSetNameFromKey($key)
{
    $parts = explode('_', $key);
    $parts = array_map('ucfirst', $parts);
    $setter = 'set' . implode('', $parts);
    if (!method_exists($this, $setter)) {
        [...]
    }
    return $setter;
}

```

Le constructeur fait appel à la méthode « `setFromArray()` » qui s'occupe d'injecter les options depuis un tableau afin que chaque valeur de la configuration soit traitée par la méthode magique « `__set()` ». Cette méthode en recherche une autre du nom de « `setMaVariable()` » où « `ma_variable` » est une clé du tableau de configuration. Par exemple, la clé « `config_cache_key` » est injectée depuis la méthode « `setConfigCacheKey()` » dont le nom est formaté en « `CamelCase` » tout en supprimant les underscores du nom de la variable. Nous comprenons donc que le fait de ne pas respecter cette règle, en nommant ou en insérant une clé qui ne correspond à aucune méthode interne, lèvera une exception :

Zend/Stdlib/Options.php

```

protected function assembleSetNameFromKey($key)
{
    [...]
    if (!method_exists($this, $setter)) {
        throw new Exception\BadMethodCallException(
            'The option "' . $key . '" does not '
            . 'have a matching ' . $getter . ' getter method '
            . 'which must be defined'
        );
    }
    return $setter;
}

```

L'exception levée dans notre exemple

```

Fatal error: Uncaught exception 'Zend\Stdlib\Exception\BadMethodCallException' with message 'The option "ma_cle_inconnu" does not have a matching setMaCleInconnu'

```

Il devient alors très simple de composer une classe responsable de la gestion d'options d'un composant. Il suffit d'écrire les « `setters` » et « `getters` », méthodes de

récupération et de modification, correspondant aux noms des attributs et la classe devient fonctionnelle. Cette classe est utilisée à de nombreuses reprises dans le framework : Zend\Cache, Zend\Mail, Zend\Serializer, etc.

La classe ArrayUtils

La classe ArrayUtils offre de nombreuses fonctions afin de manipuler au mieux les tableaux. La classe offre des méthodes afin de connaître le type des clés d'un tableau, par exemple en testant s'il existe des clés du type de la chaîne de caractères :

Zend/Stdlib/ArrayUtils.php

```
public static function hasStringKeys($value, $allowEmpty = false)
{
    if (!is_array($value)) {
        return false;
    }
    if (!$value) {
        return $allowEmpty;
    }
    return count(array_filter(array_keys($value), 'is_string')) > 0;
}
```

La méthode « hasStringKeys() » applique un filtre sur la liste des clés du tableau afin de récupérer les clés de type chaîne de caractères. Deux autres méthodes similaires existent afin de tester s'il existe des clés de type entier ou numérique.

La méthode statique « iteratorToArray() » permet de compléter la fonction native de PHP « iterator_to_array() » en permettant de parcourir récursivement le tableau ou l'objet implémentant l'interface Traversable passé en paramètre. En effet, la méthode laisse la possibilité de passer un tableau en paramètre, auquel cas il retournera simplement le tableau si l'on n'a pas demandé de le parcourir récursivement :

Zend/Stdlib/ArrayUtils.php

```
public static function iteratorToArray($iterator, $recursive = true)
{
    if (!is_array($iterator) && !$iterator instanceof Traversable) {
        throw new Exception\InvalidArgumentException(__METHOD__ . '
expects an array or Traversable object');
    }
    if (!$recursive) {
        if (is_array($iterator)) {
            return $iterator;
        }
        return iterator_to_array($iterator);
    }
    [...]
}
```

La méthode vérifie ensuite si l'objet ne comporte pas une méthode « toArray() », auquel cas elle est utilisée avant de parcourir le tableau récursivement :

Zend/Stdlib/ArrayUtils.php

```

public static function iteratorToArray($iterator, $recursive = true)
{
[...]
if (method_exists($iterator, 'toArray')) {
    return $iterator->toArray();
}
$array = array();
foreach ($iterator as $key => $value) {
    if (is_scalar($value)) {
        $array[$key] = $value;
        continue;
    }
    if ($value instanceof Traversable) {
        $array[$key] = static::iteratorToArray($value,
$recursive);
        continue;
    }
    if (is_array($value)) {
        $array[$key] = static::iteratorToArray($value,
$recursive);
        continue;
    }
    $array[$key] = $value;
}
return $array;
}

```

À chaque fois que la boucle rencontre un tableau ou un objet de type Traversable, la méthode est appelée récursivement. Nous comprenons l'intérêt de la souplesse de la méthode sur le type en premier paramètre. Cette méthode permet de gérer facilement les compositions de variables entre tableaux, objets de type Traversable et objets implémentant la méthode « toArray() ».

La classe CallbackHandler

La classe CallbackHandler permet de surcharger le comportement des fonctions natives de PHP « call_user_func() » ou encore « call_user_func_array() » qui permettent d'appeler des fonctions de rappels. La classe offre la possibilité de prendre en paramètre tout ce qui peut être appelé en tant que fonction :

Zend/Stdlib/CallbackHandler.php

```

public function __construct($callback, array $metadata = array())
{
    $this->metadata = $metadata;
    $this->registerCallback($callback);
}

```

Zend/Stdlib/CallbackHandler.php

```

protected function registerCallback($callback)
{
    if (!is_callable($callback)) {
        throw new Exception\InvalidCallbackException('Invalid callback
provided; not callable');
}

```

```

    }
    [...]
}
```

Il est donc possible de passer de nombreux paramètres au constructeur de la classe : tableau, closure, chaîne de caractères, etc. Voici quelques exemples d'utilisation où nous définissons quelques classes qui serviront de méthodes de rappel :

Utilisation du CallbackHandler

```

class Callback
{
    public function doSomething()
    {
        echo "ok\n";
    }

    public function doSomethingWithArg($arg)
    {
        echo $arg . "\n";
    }
}
function doSomething()
{
    echo "ok\n";
}
```

Maintenant voici comment utiliser la classe CallbackHandler :

Utilisation du CallbackHandler

```

$callbackObject = new Callback();
$callback = new Stdlib\CallbackHandler(array('Callback',
    'doSomething'));
$callback->call();

$callback = new Stdlib\CallbackHandler(array($callbackObject,
    'doSomething'));
$callback->call();

$callback = new Stdlib\CallbackHandler(function() { echo "ok\n"; });
$callback->call();

$callback = new Stdlib\CallbackHandler(array($callbackObject,
    'doSomethingWithArg'));
$callback->call(array('ok'));

$callback = new Stdlib\CallbackHandler('doSomething');
$callback->call();

$callback = new Stdlib\CallbackHandler('\Callback::doSomething');
$callback->call();
```

Comme nous le voyons, la méthode « call() » permet de fournir des arguments à la méthode de callback. Évidemment, cette méthode s'appuie sur les fonctions « call_user_func() » et « call_user_func_array() » de PHP que l'on connaît :

Zend/Stdlib/CallbackHandler.php

```
|| public function call(array $args = array())
|| {
|| [...]
|| $argCount = count($args);
|| [...]
|| switch ($argCount) {
||     case 0:
||         if (self::$isPhp54) {
||             return $callback();
||         }
||         return call_user_func($callback);
||     case 1:
||         if (self::$isPhp54) {
||             return $callback(array_shift($args));
||         }
||         return call_user_func($callback, array_shift($args));
||     case 2:
||         $arg1 = array_shift($args);
||         $arg2 = array_shift($args);
||         if (self::$isPhp54) {
||             return $callback($arg1, $arg2);
||         }
||         return call_user_func($callback, $arg1, $arg2);
||     case 3:
||         $arg1 = array_shift($args);
||         $arg2 = array_shift($args);
||         $arg3 = array_shift($args);
||         if (self::$isPhp54) {
||             return $callback($arg1, $arg2, $arg3);
||         }
||         return call_user_func($callback, $arg1, $arg2, $arg3);
||     default:
||         return call_user_func_array($callback, $args);
||     }
|| }
```

La classe offre aussi la possibilité d'utiliser l'instance de l'objet CallbackHandler directement comme une fonction :

Zend/Stdlib/CallbackHandler.php

```
|| public function __invoke()
|| {
||     return $this->call(func_get_args());
|| }
```

Nous pouvons donc écrire ces instructions :

Utilisation de l'instance comme fonction

```
|| $callback = new Stdlib\CallbackHandler(array($callbackObject,
|| 'doSomething'));
|| $callback();
```

Ce composant peut s'avérer particulièrement utile pour nos gestions de fonctions de rappel afin de fournir un peu plus de flexibilité dans nos applications.

La classe ErrorHandler

La classe ErrorHandler fournit des méthodes de gestion pour capturer les erreurs de l'application. Voici un exemple de capture de notice :

Capture d'erreur

```
class ErrorHandlerTest
{
    public function doSomething()
    {
        trigger_error('lancement de notice', \E_USER_NOTICE);
    }
}
$errHandler = new ErrorHandlerTest();
$errHandler->doSomething();
$err = Stdlib\ErrorHandler::stop();

echo $err->getMessage();
```

Sortie à l'écran

```
lancement de notice
```

La capture d'erreur est démarrée par la méthode « `start()` » qui fait appel à la méthode « `set_error_handler()` » de PHP afin de modifier la fonction qui gère les erreurs :

Zend/Stdlib/ErrorHandler.php

```
public static function start($errorLevel = \E_WARNING)
{
    if (static::started() === true) {
        throw new Exception\LogicException('ErrorHandler already
started');
    }
    static::$started      = true;
    static::$errorException = null;
    set_error_handler(array(get_called_class(), 'addError'),
$errorLevel);
}
```

La méthode prend le type d'erreur à gérer en paramètre et enregistre sa méthode « `addError()` » comme méthode de gestion d'erreurs. Cette méthode se contente d'enregistrer l'erreur dans l'attribut « `$errorException` » que l'on peut récupérer automatiquement sous forme d'exception depuis la méthode statique « `stop()` » :

Zend/Stdlib/ErrorHandler.php

```
public static function stop($throw = false)
{
    if (static::started() === false) {
        throw new Exception\LogicException('ErrorHandler not started');
    }
    $errorException = static::$errorException;
    static::$started      = false;
    static::$errorException = null;
```

```
    restore_error_handler();

    if ($errorException && $throw) {
        throw $errorException;
    }
    return $errorException;
}
```

La classe ErrorHandler possède une API relativement facile et agréable à prendre en main qui permet de se détacher de la gestion des erreurs que l'on implémente dans chaque projet.

La classe Glob

La classe Glob permet d'utiliser la fonction native « glob() » avec les systèmes qui ne gèrent pas la méthode « glob() » avec l'utilisation du flag « GLOB_BRACE ». Une fonction a été spécialement portée de la bibliothèque GLIBC afin de rendre l'utilisation de « GLOB_BRACE » possible :

Zend/Stdlib/Glob.php

```
public static function glob($pattern, $flags, $forceFallback = false)
{
    if (!defined('GLOB_BRACE') || $forceFallback) {
        return self::fallbackGlob($pattern, $flags);
    } else {
        return self::systemGlob($pattern, $flags);
    }
}
```

Nous remarquons que cette fonction « fallbackGlob() » est utilisée si le système ne définit pas la constante « GLOB_BRACE ». Cette classe permet donc d'utiliser cette fonctionnalité sans bloquer les plates-formes qui ne sont pas compatibles. Il est évidemment conseillé d'utiliser ce composant lorsque vous souhaitez utiliser la fonction « glob() » avec le flag « GLOB_BRACE ».

Les hydrateurs

Les hydrateurs permettent d'extraire les données et de peupler les attributs d'un objet suivant une stratégie définie. Les hydrateurs de la Stdlib se trouvent dans le namespace Zend\Stdlib\Hydrator et offrent plusieurs stratégies :

- extraction et modification depuis les méthodes « getXXX() » et « setXXX() » de l'objet ;
- extraction et modification depuis les propriétés publiques de l'objet ;
- extraction et modification depuis une analyse, à l'aide de la classe reflectionClass, de l'objet ;
- extraction et modification depuis les méthodes « getArrayCopy() » et « exchangeArray() » propres aux objets travaillant avec les tableaux.

Voici un exemple d'utilisation de l'hydrateur basé sur les « getters » et « setters » de l'objet :

Utilisation de l'hydrateur ClassMethod :

```
class Exemple
{
protected $ma_propriete = 'ma valeur';

public function getMaPropriete()
{
    return $this->ma_propriete;
}

public function setMaPropriete($ma_propriete)
{
    $this->ma_propriete = $ma_propriete;
    return $this;
}
}

$exemple = new Exemple();
$hydrateur = new Stdlib\Hydrator\ClassMethods();
$datas = $hydrateur->extract($exemple);
```

L'hydrateur implémente l'interface Zend\Stdlib\Hydrator\HydratorInterface qui définit le contrat de base que doivent respecter les hydrateurs :

Zend/Stdlib/Hydrator/HydratorInterface.php

```
interface HydratorInterface
{
public function extract($object);
public function hydrate(array $data, $object);
}
```

La méthode « extract() » permet d'extraire les données de l'objet suivant la stratégie de l'hydrateur et la méthode « hydrate() » permet de peupler l'objet depuis les données fournies en paramètre.

Dans notre exemple, nous utilisons l'hydrateur basé sur les méthodes de l'objet. Les méthodes « getMaPropriete() » et « setMaPropriete() » vont lui permettre d'extraire de l'objet, un tableau avec comme entrée une clé « ma_propriete » avec la valeur « ma valeur ».

Examinons le fonctionnement de l'hydrateur Zend\Stdlib\Hydrator\ClassMethods :

Zend/Stdlib/Hydrator/ClassMethods.php

```
public function extract($object)
{
[...]
$transform = function($letters) {
    $letter = array_shift($letters);
    return '_' . strtolower($letter);
};
$attributes = array();
$methods = get_class_methods($object);
```

```

foreach ($methods as $method) {
    if (!preg_match('/^(get|has|is)[A-Z]\w*/', $method)) {
        continue;
    }
    if (preg_match('/^get/', $method)) {
        $setter = preg_replace('/^get/', 'set', $method);
        $attribute = substr($method, 3);
        $attribute = lcfirst($attribute);
    } else {
        $setter = 'set' . ucfirst($method);
        $attribute = $method;
    }
    if (!in_array($setter, $methods)) {
        continue;
    }
    if ($this->underscoreSeparatedKeys) {
        $attribute = preg_replace_callback('/([A-Z])/', '',
$transform, $attribute);
    }
    $attributes[$attribute] = $this->extractValue($attribute,
$object->$method());
}
return $attributes;
}

```

L'hydrateur récupère tout d'abord la liste des méthodes de l'objet avant de les parcourir :

Zend/Stdlib/Hydrator/ClassMethods.php

```

public function extract($object)
{
[...]
$methods = get_class_methods($object);
foreach ($methods as $method) {
    [...]
}
}

```

Comme nous souhaitons extraire des données, l'hydrateur se base alors sur le « getter » de l'objet et donc ses méthodes commençant par « get », tout en vérifiant que le « setter » correspondant existe bel et bien :

Zend/Stdlib/Hydrator/ClassMethods.php

```

public function extract($object)
{
foreach ($methods as $method) {
    if (!preg_match('/^(get|has|is)[A-Z]\w*/', $method)) {
        continue;
    }
    if (preg_match('/^get/', $method)) {
        [...]
    } else {
        $setter = 'set' . ucfirst($method);
        $attribute = $method;
    }
    if (!in_array($setter, $methods)) {
        continue;
    }
}
}

```

Nous remarquons que l'hydrateur récupère aussi les méthodes de récupération de données commençant par « has » ou « is ». La seule différence de ces méthodes avec celles commençant par « get » réside dans le nom de « setter ».

Une fois les vérifications faites, le nom de l'attribut est formaté afin d'être enregistré dans le tableau de retour :

Zend/Stdlib/Hydrator/ClassMethods.php

```
public function extract($object)
{
[...]
foreach ($methods as $method) {
    [...]
    if (preg_match('/^get/', $method)) {
        $setter = preg_replace('/^get/', 'set', $method);
        $attribute = substr($method, 3);
        $attribute = lcfirst($attribute);
    } else {
        $setter = 'set' . ucfirst($method);
        $attribute = $method;
    }
    [...]
    $attributes[$attribute] = $this->extractValue($attribute,
$object->$method());
}
[...]
}
```

Nous remarquons que nous avons la possibilité de transformer les noms de clés au format « CamelCase » ou un format avec underscore grâce à l'attribut « \$underscoreSeparatedKeys » que nous pouvons définir depuis le constructeur :

Zend/Stdlib/Hydrator/ClassMethods.php

```
public function __construct($underscoreSeparatedKeys = true)
{
parent::__construct();
$this->underscoreSeparatedKeys = $underscoreSeparatedKeys;
}
```

Par défaut, la classe utilise les underscores. Nous aurons donc la clé « ma_propriete » pour la méthode « getMaPropriete() ». Si nous passons le paramètre à « false », nous obtiendrons la clé « maPropriete ».

Lors de l'hydratation, ou peuplement de l'objet, la méthode « extract() » utilisera les méthodes commençant par « set » de l'objet si celles-ci existent :

Exemple d'hydratation

```
$hydrateur = new Stdlib\Hydrator\ClassMethods();
$data = array('ma_propriete' => 'nouvelle valeur');
$hydrateur->hydrate($data, $exemple);
```

Revenons à l'extraction et remarquons que l'hydrateur utilise la méthode « extractValue() » pour enregistrer la valeur plutôt qu'une affectation classique. Ceci permet

d'ajouter une surcouche lors de l'extraction ou de l'hydratation. Voici comment se comporte la méthode « `extractValue()` » :

Zend/Stdlib/Hydrator/AbstractHydrator.php

```
public function extractValue($name, $value)
{
    if ($this->hasStrategy($name)) {
        $strategy = $this->getStrategy($name);
        $value = $strategy->extract($value);
    }
    return $value;
}
```

L'`AbstractHydrator` dont héritent les hydrateurs vérifie si une stratégie particulière existe sur la propriété à hydrater afin de l'utiliser. Par défaut, la valeur est simplement retournée. Voici un exemple de stratégie ajoutée sur l'hydrateur :

Ajout de stratégie

```
class FilterStrategy implements StrategyInterface
{
    public function extract($value)
    {
        return strip_tags($value);
    }

    public function hydrate($value)
    {
        return "<p>" . $value . "</p>";
    }
}

$exemple->setMaPropriete('<p>ma valeur</p>');
$hydrateur = new Stdlib\Hydrator\ClassMethods();
$hydrateur->addStrategy('ma_propriete', new FilterStrategy());
$data = $hydrateur->extract($exemple);
```

La stratégie permet ici de supprimer les tags de la clé correspondant à « `ma_propriete` » lors de l'extraction et de les ajouter lors de l'hydratation.

Attention, nous utilisons les underscores pour les clés. Si nous passons au format CamelCase, cela ne fonctionnera plus, sauf si nous modifions la clé pour la stratégie :

Utilisation du format CamelCase

```
$exemple->setMaPropriete('<p>ma valeur</p>');
$hydrateur = new Stdlib\Hydrator\ClassMethods(false);
$hydrateur->addStrategy('maPropriete', new FilterStrategy());
$data = $hydrateur->extract($exemple);
```

Les hydrateurs sont très utilisés avec les formulaires et peuvent l'être dans de nombreuses applications. Ils sont très extensibles et très simples d'utilisation. Les autres hydrateurs fonctionnent de la même manière, ils ne diffèrent que dans la récupération des données de l'objet depuis ses méthodes de classes ou directement depuis ses propriétés.

16

Les exceptions

Le framework a également mis à jour son système d'exception afin de simplifier la gestion des erreurs lors du développement. Chaque composant implémente une interface correspondant à son espace de nom, par exemple avec le composant des contrôles d'accès :

`Zend/Permissions/Acl/ExceptionExceptionInterface.php`

```
|| interface ExceptionInterface
|| {}
```

Cette interface est ensuite implémentée par toutes les classes d'exception de cet espace de nom, ce qui laisse la possibilité de faire référence à une exception provenant d'un espace de nom ou composant précis, lorsque nous attrapons les exceptions. Le nom de la classe de chacune des exceptions de cet espace de nom permet de filtrer ensuite l'erreur plus précisément :

Filtrage des erreurs

```
|| try {
||   [...] // bloc d'actions
|| }
|| catch(Zend\Permissions\Acl\Exception\RuntimeException $e)
|| {
||   [...] // exception la plus spécialisée
|| }
|| catch(Zend\Permissions\Acl\Exception\ExceptionInterface $e)
|| {
||   [...] // exception la plus générique qui fait partie du composant
|| Zend\Permissions\Acl
|| }
|| catch(\RuntimeException $e)
|| {
||   [...] // exception de base
|| }
```

Le framework est maintenant plus riche en classes d'exception afin d'augmenter la qualité du code et la gestion des erreurs. Il devient plus facile de créer ses propres exceptions dans la bibliothèque de son application tout en conservant la règle d'une interface par composant et d'une ou plusieurs exceptions pour les sous-composants.

L'ajout d'une exception se résume alors à la création une classe vide possédant un

nom propre à l'exception, nom généralement assez explicite, ainsi que de l'implémentation de l'interface d'exception de l'espace de nom correspondant. Il convient ensuite de la faire hériter de la classe d'exception de base qui la décrit au mieux. Par exemple, l'exception de type Zend\Permissions\Acl\Exception\Runtime n'étiendra pas \Exception mais \RuntimeException qui lui correspond davantage :

Zend/Permissions/Acl/RuntimeException.php

```
class RuntimeException extends \RuntimeException implements
ExceptionInterface
{}
```

AVEC LE ZEND FRAMEWORK 1

Les exceptions étendent toutes Zend_Exception (cette classe étend elle-même \Exception), ce qui ne permet pas de pouvoir filtrer et gérer les exceptions avec précision.

Prendre les bonnes habitudes du Zend Framework 2 au niveau de la gestion des exceptions permet le maintien d'un code où la gestion des erreurs en sera plus simple, ce qui facilitera alors les corrections.

17

Cas d'utilisation

Afin de mettre en œuvre ce qui a été expliqué dans les chapitres précédents, nous allons construire un petit projet qui permettra de comprendre les étapes de la création d'une application Web avec le framework. Ce projet sera aussi l'occasion de découvrir des composants du framework qui n'ont pas été analysés dans les derniers chapitres.

La première partie présente le concept de l'application ainsi que les fonctionnalités que nous implémenterons et détaillerons lors de l'analyse. L'architecture, les bibliothèques utilisées, la configuration et les contrôleur suivront afin de faire un tour complet du code de l'application.

Vous pouvez également retrouver le code du projet sur mon compte github afin de visualiser le code en même temps que la lecture : <https://github.com/blanchonvincent/au-coeur-de-zend-framework-2>. Le code de ce projet pouvant évoluer, vous retrouverez le code utilisé lors de l'écriture de cet ouvrage avec le tag « 2.0.0 » qui est la version du framework utilisée.

Concept et fonctionnalités

Présentation du projet

Le projet étudié est une application dédiée à l'agrégation de contenus autour du Zend Framework 2 : articles, tutoriels, tweets, vidéos, développeurs, etc. J'ai choisi de réaliser ce projet afin de ne pas reprendre l'exemple classique de la gestion de livres que l'on connaît et qui reste limité en termes de fonctionnalité. Si ce projet n'a pas d'intérêt particulier, il va permettre d'utiliser de nombreux composants que l'on n'a peut-être pas l'occasion d'utiliser souvent comme l'API Twitter ou YouTube, et d'autres qui font partie intégrante de la vie d'un projet comme les bases de données ou les formulaires.

Ce projet est hébergé sur le nom de domaine « <http://zend-framework-2.fr> » et permettra de regrouper les différentes ressources sur le framework afin de pouvoir aider, je l'espère, quelques développeurs dans leur apprentissage.

Les besoins du projet

Afin de pouvoir développer ce projet, il est nécessaire de devoir télécharger les projets suivants :

- ZendService ;
- ZendGData ;
- ZendRest.

Ces projets sont maintenus séparément du framework pour des questions pratiques comme nous l'avons expliqué dans le chapitre dédié aux composants. Ils sont disponibles sur le compte Github du Zend Framework.

Le projet utilise aussi une base de données MySQL pour la sauvegarde des données et une version de PHP supérieure ou égale à PHP 5.3.3, prérequis pour l'utilisation du framework. À tous ceux qui se demandent pourquoi cette version précise, depuis PHP 5.3.3, les méthodes ayant le même nom que la classe dans laquelle elles se trouvent ne sont plus traitées comme des constructeurs. En effet, l'utilisation d'une méthode de fabrique « fabrique() » dans un fichier « Fabrique.php » poserait un problème avec l'utilisation d'une version antérieure, la méthode serait alors considérée comme son constructeur.

Afin d'étudier le projet, listons les fonctionnalités qui seront disponibles dans l'application :

- la page d'accueil listera les derniers tweets et tutoriels disponibles sur le site ;
- une page listera les tweets à propos du Zend Framework et sera paginée. Une autre fera la même chose avec les tutoriels suivant la langue française ou anglaise ;
- une page qui liste les données du framework sur les réseaux sociaux Facebook, Slideshare et YouTube ;
- une page listant les développeurs Zend Framework en France avec leurs liens Viadeo, Linkedin et Twitter, ainsi que leurs certifications ;
- un formulaire capable d'inscrire un nouveau développeur qui en ferait la demande ;
- un formulaire de contact qui permet l'envoi d'un simple e-mail.

Nous ferons souvent référence à ces besoins tout au long de l'étude du projet afin d'expliquer les solutions mises en œuvre.

Organisation et architecture

Pour mettre en œuvre notre projet, il est important de définir une arborescence. Trois modules sont présents :

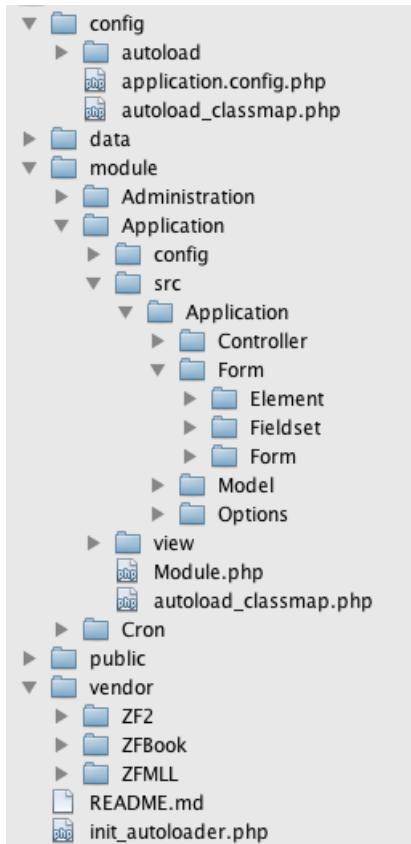
- « Application », qui représente le cœur de l'application (fabriques, configurations, modèles de données, etc.) et qui contient les contrôleurs et vues de la partie utilisateur ;

- « Administration » qui contient les contrôleurs et vue de l'administration du site. Dans notre projet, l'administration n'a pas été développée, mais le module est présent pour indiquer comment peut être faite la séparation entre les différentes fonctionnalités du site et sert aussi d'exemple pour la gestion de la restriction de chargement de modules que l'on verra plus loin ;

- « Cron » qui contient les contrôleurs pour la gestion des tâches automatisées.

L'avantage de séparer ces trois modules est qu'il est désormais possible de désactiver l'un d'entre eux temporairement sans affecter le cœur de l'application.

Le module « Application » contient l'ensemble des fabriques et modèles de l'application, ce module ne peut pas être désactivé. Voici la hiérarchie des fichiers :



Le dossier « vendor » contient trois dossiers :

- « ZF2 » qui est toutes les bibliothèques du Zend Framework : le framework, ZendService, ZendGData et ZendRest ;
- « ZFBook » qui est la bibliothèque propre au projet ;
- « ZFMLL » qui est la bibliothèque permettant de réaliser la restriction de chargement de modules que nous verrons dans la prochaine section.

Maintenant que notre architecture est définie, analysons la bibliothèque pour la restriction de chargement de modules qui va nous permettre de nous plonger au

œur du composant ModuleManager.

Personnalisation du chargement de modules

La restriction de chargement

Lors du chargement des modules du projet, le framework charge et initialise chacun des modules avant de garder la configuration en mémoire. Une fois cette étape réalisée, lorsqu'un nouvel utilisateur fait une requête sur la plateforme Web, le framework pourra supprimer l'étape de création de configuration qui est en cache et aura juste à initialiser les modules. Cependant, dans certains cas, le fait d'initialiser tous les modules peut être pénalisant. En effet, nous pouvons souhaiter restreindre l'accès à la zone d'administration à un certain nombre d'adresses IP ou à l'accès en HTTPS uniquement. Nous pouvons également envisager de charger le module de Cron uniquement si nous utilisons la SAPI CLI de PHP car ce module ne doit pas être accessible par un autre moyen. C'est à partir de ce raisonnement que j'ai mis en place une bibliothèque permettant le chargement à la demande en indiquant les conditions de chargement.

Ce module étend les fonctionnalités du gestionnaire de modules. Il peut être téléchargé sur mon compte Github. Il est conseillé de bien avoir assimilé le chapitre sur la gestion des modules avant de lire l'implémentation de cette bibliothèque afin de comprendre les explications fournies. Voyons maintenant son implémentation.

L'implémentation du chargement à la demande

Afin de gérer la restriction du chargement, la première étape est d'étendre la classe ListenerOptions qui gère le stockage des options liées aux modules. Cette nouvelle classe hérite de celle proposée par le framework et définit un nouvel attribut :

ZFMLL/ModuleManager/Listener/ListenerOptions.php

```
class ListenerOptions extends BaseListener
{
    protected $lazyLoading = array();

    public function getLazyLoading()
    {
        return $this->lazyLoading;
    }
    public function setLazyLoading(array $lazyLoading)
    {
        $this->lazyLoading = $lazyLoading;
        return $this;
    }
}
```

L'attribut « lazyLoading » contiendra toutes les restrictions de chargement que l'on aura définies. Voici un exemple d'utilisation depuis la configuration de l'application :

application.config.php

```

<?php
return array(
[...]
'module_listener_options' => array(
[...]
'lazy_loading' => array(
'Cron' => array(
'sapi' => 'cli',
),
'Administration' => array(
'port' => 443,
'remote_addr' => array('127.0.0.1'),
),
),
[...]
);

```

Comme pour les options, la classe `DefaultListenerAggregate` qui contient la liste des écouteurs du gestionnaire de modules a été étendue afin de gérer ses propres événements :

ZFMLL/ModuleManager/Listener/AuthListenerAggregate.php

```

class AuthListenerAggregate extends DefaultListenerAggregate
{
public function attach(EventManagerInterface $events)
{
    $options = $this->getOptions();
    $lazyLoading = $options->getLazyLoading();

    $listenerManager = new AuthManager($lazyLoading);
    $this->listeners[] = $events->attach(ModuleEvent::EVENT_LOAD_
MODULE_AUTH, array($listenerManager, 'authorize'));
    return parent::attach($events);
}
}

```

Un nouvel événement, avec pour identifiant « `loadModuleAuth` » va permettre de court-circuiter le chargement de base afin d'effectuer les vérifications nécessaires.

Une fois les classes de base redéfinies, il n'y a plus qu'à surcharger le comportement du gestionnaire de modules. La vérification des restrictions doit se faire juste avant le chargement afin de ne charger que ceux qui remplissent les conditions définies. Le nouvel événement peut donc être lancé avant la phase de chargement :

ZFMLL/ModuleManager/ModuleManager.php

```

public function loadModules()
{
$this->getEventManager()->trigger(ModuleEvent::EVENT_LOAD_MODULES_
AUTH, $this, $this->getEvent());
return parent::loadModules();
}

```

La méthode « `onLoadModulesAuth()` » qui écoute cet événement se contente de mettre à jour le tableau de module en fonction du besoin de chargement de chacun

d'entre eux :

ZFMLL\ModuleManager\ModuleManager.php

```
public function onLoadModulesAuth()
{
    [...]
    $modules = array();
    foreach ($this->getModules() as $moduleName) {
        $auth = $this->loadModuleAuth($moduleName);
        if($auth) {
            $modules[] = $moduleName;
        }
    }
    $this->setModules($modules);
}
```

La méthode « loadModuleAuth() » lance l'évènement « loadModuleAuth » qui permet de définir si le module doit être chargé ou non. Une fois que chacun des modules a été vérifié, le tableau est mis à jour afin de reprendre le cycle habituel de chargement. Plus le nombre de modules sera restreint, plus notre application Web se chargera vite. Dans cet exemple, les modules d'administration et de gestion des tâches automatisées ne seraient pas chargés lorsque l'on demande l'accès à la page d'accueil. D'après les benchmarks effectués, le temps de chargement des modules diminue de 5 à 70 % en fonction des règles et besoins définis dans la configuration. Cela peut aussi être intéressant d'un point de vue sécurité en ne laissant aucune possibilité d'agir sur la zone d'administration ou de gestion des crons depuis un accès sur le port 80 ou depuis un simple navigateur par exemple.

Cette bibliothèque utilise aussi un gestionnaire de plugins afin de gérer les classes de vérification de restriction, ce qui nous permet de consulter les possibilités offertes par la bibliothèque :

ZFMLL\ModuleManager\Listener\ListenerManager.php

```
class ListenerManager extends AbstractPluginManager
{
    protected $invokableClasses = array(
        'datetime' => 'ZFMLL\ModuleManager\Listener\Server\DateTime',
        'hostname' => 'ZFMLL\ModuleManager\Listener\Server\'
DomainListener',
        'getopt' => 'ZFMLL\ModuleManager\Listener\Environment\
GetoptListener',
        'http_method' => 'ZFMLL\ModuleManager\Listener\Server\
HttpMethod',
        'https' => 'ZFMLL\ModuleManager\Listener\Server\HttpsListener',
        'port' => 'ZFMLL\ModuleManager\Listener\Server\PortListener',
        'remoteaddr' => 'ZFMLL\ModuleManager\Listener\Server\
RemoteAddrListener',
        'sapi' => 'ZFMLL\ModuleManager\Listener\Environment\
SapiListener',
        'url' => 'ZFMLL\ModuleManager\Listener\Server\UrlListener',
        'user_agent' => 'ZFMLL\ModuleManager\Listener\Server\
UserAgent',
    );
    protected $aliases = array(
        'php_sapi' => 'sapi',
        'domain' => 'hostname',
    );
}
```

```
    'uri' => 'url',
    'remote_addr' => 'remoteaddr',
    'ip' => 'remoteaddr',
    'http_user_agent' => 'user_agent',
);
[...]
}
```

Nous remarquons que la restriction peut porter sur l'heure courante, la méthode HTTP utilisée, le nom de domaine, l'URL d'accès, le port utilisé, l'agent utilisateur, etc.

Une fois cette étape effectuée, le chargement reprend son cours normal et la configuration de l'application est créée. Analysons celle de notre projet.

La configuration

Notre projet ne comporte pas de variable d'environnement, car comme nous l'avons expliqué dans le chapitre sur les configurations, un fichier de configuration locale sera déployé sur les serveurs concernés en même temps que le code d'application. Adopter cette méthodologie permet de ne plus avoir à versionner les informations sensibles et permet de limiter les effets de bord avec l'utilisation d'une variable d'environnement. Le fichier « local.config.php » dans le dossier « config/autoload » contient donc les configurations propres à l'environnement. Dans notre projet, nous retrouverons les configurations des bases de données et paramètres SMTP :

config/autoload/local.config.php

```
<?php
return array(
    'db' => array(
        'driver' => 'Pdo',
        'dsn' => 'mysql:dbname=zf2book;host=127.0.0.1',
        'username' => 'root',
        'password' => 'root',
    ),
    'smtp_options' => array(
        'host' => '',
        'connection_config' => array(
            'username' => '',
            'password' => '',
        )
    ),
);
```

Un fichier, non versionné, est déployé sur le serveur de production avec les valeurs qui seront utilisées sur cet environnement. Aucune information sensible n'est donc versionnée.

En ce qui concerne le reste des configurations, le module « Application » définit une partie de sa configuration dans le fichier « module.config.php » :

```
module.config.php
<?php
return array(
    'router' => include 'routes.config.php',
    'service_manager' => array(
        'factories' => array(
            'DefaultNavigation' => 'Zend\Navigation\Service\DefaultNavigationFactory',
        ),
    ),
    'controllers' => array(
        'invokables' => array(
            'application-index' => 'Application\Controller\IndexController',
        ),
    ),
    'controller_plugins' => array(
        'invokables' => array(
            'flashmessenger' => 'ZFBook\Mvc\Controller\Plugin\FlashMessenger',
        ),
    ),
    'view_helpers' => array(
        'invokables' => array(
            'tags' => 'ZFBook\View\Helper\Tags',
            'messages' => 'ZFBook\View\Helper\FlashMessenger',
            'userTwitter' => 'ZFBook\View\Helper\UserTwitter',
        ),
    ),
    [...]
);
```

Ce fichier nous permet de définir nos propres aides de vue et d'action ainsi que nos contrôleurs. Le reste de la configuration est défini depuis la classe Module :

Module.php

```
class Module implements AutoloaderProviderInterface,
    ServiceProviderInterface,
    ConfigProviderInterface
{
    [...]
    public function getServiceConfig()
    {
        return array(
            'invokables' => array(
                'TweetService' => 'Application\Model\Service\TweetService',
                [...]
            ),
            'factories' => array(
                'SmtpOptions' => function($sm) {
                    $config = $sm->get('config');
                    return new SmtpOptions($config['smtp_options']);
                },
                'DbAdapter' => function($sm) {
                    $config = $sm->get('config');
                    $config = $config['db'];
                    $dbAdapter = new DbAdapter($config);
                    return $dbAdapter;
                }
            )
        );
    }
}
```

```

    },
    'TweetTable' => function($sm) {
        return new Table\TweetTable('tweet', $sm->get('DbAdapter'));
    },
    [...]
),
'aliases' => array(
    'TweetModel' => 'TweetTable',
    [...]
),
);
}

```

Chaque fabrique de modèles de table est définie dans la partie réservée au gestionnaire de services et une fabrique permet d'englober les options de notre application : « SmtpOptions ». Cette classe permet de piloter les options propres à l'envoi d'e-mails.

Toutes nos configurations et fabriques sont prêtes, il ne reste plus qu'à les utiliser dans nos contrôleurs.

Les contrôleurs de l'application

Les contrôleurs du module « Application » se contentent d'utiliser les modèles pour afficher les données avec le paginateur, dont voici quelques exemples :

Listing des tweets et tutoriels de la page d'accueil

```

public function indexAction()
{
    $sm = $this->getServiceLocator();
    return array(
        'tweets' => $sm->get('TweetModel')->fetchAllLastValid(25),
        'tutofr' => $sm->get('TutorialModel')->fetchAllLastValidByLang(
            'fr', 3),
        'tutoen' => $sm->get('TutorialModel')->fetchAllLastValidByLang(
            'en', 3),
    );
}

```

Voici une capture d'écran de la page d'accueil :



The screenshot shows the Zend Framework 2 homepage. At the top, there is a navigation bar with a 'Home' link. Below the navigation, there are two main sections: 'Tutorial Zend Framework 2 en français' and 'Tweet Zend Framework'. The 'Tutorial' section features a thumbnail for 'Livre Zend Framework 2' and a brief description. The 'Tweet' section lists several tweets from various users, each with a link to the tweet's details.

La page listant les tweets utilise le paginateur du framework :

Listing des tweets paginés

```
public function tweetAction()
{
    $page = $this->getRequest()->getQuery()->get('page', 1);
    $numByPage = 25;
    $sm = $this->getServiceLocator();
    $tweets = $sm->get('TweetModel')->getQueryLastValid();

    $paginator = new Paginator\Paginator(new Paginator\Adapter\DbSelect($tweets, $sm->get('DbAdapter')));
    $paginator->setItemCountPerPage($numByPage);
    $paginator->setCurrentPageNumber($page);
    $paginator->setPageRange(5);

    return array('tweets' => $paginator);
}
```

Chaque modèle est récupéré depuis le gestionnaire de services afin de toujours utiliser la même instance d'objet. En effet, il n'est pas nécessaire de créer un objet de modèle à chaque fois que l'on souhaite interroger la base de données. De plus, comme nous avons fait un alias de « TweetModel » vers « TweetTable », il devient facile de changer le mode de stockage des données si besoin en redéfinissant l'alias.

La méthode du modèle récupère simplement la liste des derniers tweets valides :

Application/Model/Table/TweetTable.php

```
public function getQueryLastValid()
{
    $select = $this->getSql()->select()
        ->columns(array('date', 'user', 'text'))
        ->join('language', 'language.id = tweet.language')
        ->where(array('moderate'=>1))
        ->order('date DESC');

    return $select;
}
```

Voici une capture d'écran de la page listant les tweets :

Home / Tweet ZF2

Tweet sur le Zend Framework

- @andrewx192** : Zend Framework 2.0 Documentation: <http://t.co/s8WFX50C>
- @tagmyjob** : Zend Framework 2 hello world example <http://t.co/wJxQip0>
- @inglisd101** : About to do the Zend Framework 2 Tutorial! Wish me luck.
- @phphindonesia** : rilis Zend Framework 2.0 stable <http://t.co/JXlrgrcb>
- @rufin** : Finally #Zend #Framework 2.0 is released (and #Symfony 2.1) <http://t.co/QTbWLhlc>
- @webandiphp** : Fact. ScnSocialAuth is a great example of how #ZF2 and its modules are easily extensible when designed correctly.
- @scollado** : Does anyone know a good tutorial on #ZF2 with #TDD ? [#phpunit #zend #ZendFramework](#)
- @efikaphp** : Why bother with #Zend Framework 2? <http://t.co/3GzTUVIR #php>
- @madmamor** : Zend Framework 2.0.0 STABLE Released! <http://t.co/7qAPKfve>
- @sajj89** : Zend Framework 2.0.0 has arrived <http://t.co/Kuy9hC9r>

Les formulaires sont utilisés dans l'action qui permet d'enregistrer un profil de développeur sur le site :

Formulaire d'ajout d'un utilisateur

```

public function registerDeveloperAction()
{
    $form = new Form\Developpeur();
    $request = $this->getRequest();
    if ($request->isPost()) {
        $formData = $request->getPost();
        $form->setData($formData);
        if ($form->isValid()) {
            $formData = $form->getData();
            $sm = $this->getServiceLocator();
            $sm->get('DeveloperModel')->register($formData);
            $this->plugin('flashmessenger')->addValidMessage('Merci
pour l\'inscription, celle-ci est généralement prise en compte sous
48h.');
            return $this->plugin('redirect')->toRoute('register-
developer');
        }
        $this->flashMessenger()->addError('Merci de corriger les
erreurs du formulaire.');
    }

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

```

Le formulaire d'enregistrement vérifie la validité des données avant de les enregistrer. Voici le formulaire d'enregistrement :

Application/Form/Form/Developpeur.php

```

class Developpeur extends AbstractForm
{
    public function init()
    {
        $this->add(new Fieldset\WebIdentityFieldset());
        $this->add(new Fieldset\PHPKnowledge());
        $this->add(new Fieldset\SocialLinks());
        $this->add(new Element\Button\Add());
    }
}

```

L'appel à la méthode « `init()` » est fait depuis la classe « `AbstractForm` », aucune méthode de ce type n'est prévue dans les formulaires du framework. Afin de pouvoir réutiliser les éléments et conteneurs d'éléments dans d'autres formulaires, des classes de base ont été créées afin de construire rapidement de simples formulaires. Ces classes fournissent un élément avec un type, label et nom par défaut. Le conteneur « `WebIdentityFieldset` » comporte des éléments de base :

Application/Form/Fieldset/WebIdentityFieldset.php

```

class WebIdentityFieldset extends AbstractFieldset
{
    public function __construct($name = null)
    {
        parent::__construct('web_identity');
        $this->add(new Personnal\Firstname());
        $this->add(new Personnal\Name());
    }
}

```

```

    $this->add(new Personnal\Email());
}
}

```

Tous les éléments de base ont été créés pour les besoins du projet car d'autres formulaires les utilisent. Cette manière de faire dépend des besoins du projet, celui-ci ne nécessite pas de personnalisation avancée et la création d'éléments génériques permet un gain de temps lors de la création de nouveaux formulaires.

Une fois les informations soumises, celles-ci sont enregistrées dans le modèle de la table des développeurs :

Application/Model/Table/DeveloperTable.php

```

public function register($datas)
{
    $data = array(
        'name' => $datas['web_identity']['name'],
        'email' => $datas['web_identity']['email'],
        'is_php_5_certified' => (boolean)$datas['php_knowledge']
        ['php5certification'],
        'is_php_53_certified' => (boolean)$datas['php_knowledge']
        ['php53certification'],
        'is_php_zf1_certified' => (boolean)$datas['php_knowledge']
        ['zf1certification'],
        'viadeo' => $datas['social_links']['viadeolink'],
        'linkedin' => $datas['social_links']['linkedinlink'],
        'twitter' => $datas['social_links']['twitterlink'],
        'valid' => 0,
    );
    return $this->insert($data);
}

```

Le rendu du formulaire est très simple :

Rendu du formulaire

```

<?php echo $this->messages(array('valid', 'error')); ?>

<?php
$form = $this->form;
$form->prepare();
$form->setAttribute('action', $this->url());
$form->setAttribute('method', 'post');
echo $this->form()->openTag($form);
?>

<dl class="zend_form">
<?php echo $this->formRow($form->get('web_identity')->get('name'));
?>
<?php echo $this->formRow($form->get('web_identity')-
>get('firstname')); ?>
<?php echo $this->formRow($form->get('web_identity')->get('email'));
?>
<?php echo $this->formRow($form->get('php_knowledge')-
>get('php5certification')); ?>
<?php echo $this->formRow($form->get('php_knowledge')-
>get('php53certification')); ?>

```

```
<?php echo $this->formRow($form->get('php_knowledge'))->get('zf1certification')); ?>
<?php echo $this->formRow($form->get('social_links'))->get('twitterlink')); ?>
<?php echo $this->formRow($form->get('social_links'))->get('viadeolink')); ?>
<?php echo $this->formRow($form->get('social_links'))->get('linkedinlink')); ?>
<?php echo $this->formRow($form->get('add'))); ?>
</dl>

<?php echo $this->form()->closeTag() ?>
```

Voici le rendu du formulaire :

The screenshot shows a web form titled "Ajouter son profil développeur Zend Framework". The form includes fields for Nom (Name), Prénom (First Name), Email (Email), and several checkboxes for certifications: Certifié PHP 5, Certifié PHP 5.3, and Certifié ZF1. It also has fields for Profil Twitter, Profil Viadeo, and Profil Linkedin, each with a text input field. At the bottom is an "Ajouter" (Add) button.

Home

Ajouter son profil développeur Zend Framework

Liste des développeurs »

Nom :

Prénom :

Email :

Certifié PHP 5 :

Certifié PHP 5.3 :

Certifié ZF1 :

Profil Twitter :

Profil Viadeo :

Profil Linkedin :

Le module « Application » contient la mise en forme des données. L'ajout de données, comme les tweets, se fait depuis le module « Cron » que l'on va maintenant expliquer.

Les tâches automatisées

Implémentation

La gestion des tâches automatisées utilise le composant de la console et son router afin de gérer le lancement des actions en ligne de commande. La configuration de ce module indique les options des services ainsi que les routes à utiliser :

module.config.php

```

<?php
return array(
    'cron_options' => array(
        'twitter_options' => array(
            'queries' => array(
                '#zf2f2', 'zend framework 2', '#zendframework2',
                '#zf2conf'
            ),
            'languages' => array(
                'fr', 'en',
            ),
        ),
    ),
    'controllers' => array(
        'invokables' => array(
            'cron-crawl' => 'Cron\Controller\CrawlController',
            'cron-publish' => 'Cron\Controller\PublishedController',
        ),
    ),
    'console' => array(
        'router' => array(
            'routes' => array(
                'crawl-tweet' => array(
                    'type' => 'simple',
                    'options' => array(
                        'route' => '--crawl-tweet',
                        'defaults' => array(
                            'controller' => 'cron-crawl',
                            'action' => 'tweet',
                        ),
                    ),
                ),
                [...]
            ),
        ),
    ),
);

```

Chaque action sera alors définie depuis les routes indiquées dans le router. Notons que nous avons enregistré un tableau de configuration sous la clé « cron_options » qui nous permettra d'utiliser les options définies depuis un objet créé par la fabrique « CronModuleOptions » :

Module.php

```

public function getServiceConfig()
{
    return array(
        'factories' => array(
            'CronModuleOptions' => function($sm) {
                $config = $sm->get('Config');
                return new Options\ModuleOptions($config['cron_options']);
            },
        ),
    );
}

```

Ces options vont pouvoir être utilisées facilement depuis les contrôleurs d'actions qui recherchent les nouveaux tweets ou vidéos à propos du Zend Framework. Cette recherche va être effectuée grâce au composant ZendService qui fournit des classes pour piloter les API de Twitter ou YouTube par exemple. Passons maintenant à leur implémentation.

Utilisation du composant ZendService

Le contrôleur Cron\Controller\CrawlController effectue la recherche de tweets, présentation sur Slideshare et vidéos sur YouTube, voici l'implémentation de la recherche sur Twitter :

Cron/Controller/CrawlController.php

```
public function tweetAction()
{
    $sm = $this->getServiceLocator();
    $twitterOptions = $sm->get('CronModuleOptions')->getTwitterOptions();
    $queries = $twitterOptions->getQueries();
    $langs = $twitterOptions->getLanguages();
    $results_type = array('recent', 'popular');

    $list = 0;
    foreach($langs as $lang) {
        foreach($queries as $query) {
            foreach($results_type as $result_type) {
                $search = new Twitter\Search();
                $search->setOptions(new Twitter\SearchOptions(array(
                    'rpp' => 25,
                    'include_entities' => true,
                    'result_type' => $result_type,
                    'lang' => $lang,
                )));
                $results = $search->execute($query);
                foreach($results['results'] as $result) {
                    $list += (integer)$sm->get('TweetService')-
                >addTweet($result);
                }
            }
        }
    }
    return $list . ' tweets ajoutés';
}
```

Les options de recherche sont récupérées depuis la fabrique « CronModuleOptions » et une boucle est effectuée pour couvrir l'ensemble des recherches. Une fois celles-ci exécutées, les tweets trouvés sont enregistrés dans le modèle et le nombre total est retourné. Voici un exemple de sortie après le lancement de cette action :

Sortie à l'écran

```
|| 5 tweets ajoutés
```

Les composants fournis par le framework pour piloter les API de services externes sont très simples d'utilisation et permettent un gain de temps non négligeable.

Les modules du framework

L'équipe de développement du framework met à disposition un site qui répertorie les modules créés pour le framework par les développeurs qui souhaitent partager leurs développements : <http://modules zendframework.com>.

De nombreux modules sont déjà disponibles, par exemple :

- intégration de Doctrine ;
- intégration de Google Analytics ;
- intégration de solutions de paiement ;
- gestion des API d'emailing comme Amazon, Mailchimp, etc. ;
- gestion du moteur de template TWIG.

Vous retrouverez la liste complète des modules sur le site, et il est très facile pour vous d'y contribuer. Pour cela, il suffit de laisser le code de votre module en téléchargement libre, sur Github par exemple, et de demander l'ajout de votre module à la liste. Vous ferez ainsi bénéficier la communauté de votre module.

18

Contribuer au framework

Beaucoup de développeurs se demandent ou aimeraient savoir comment contribuer au Zend Framework. S'il existe de nombreuses aides afin d'expliquer les étapes de la participation, il n'existe pas d'endroit où toutes les informations sont centralisées et expliquées de façon précise. Afin d'être le plus complet possible, nous verrons comment suivre le planning de développement du framework, savoir où trouver le code de développement qui évolue au fur et à mesure des changements effectués par les développeurs, comment reporter et suivre un bogue que l'on a rencontré sur le framework et enfin comment proposer une correction de bogue ou apporter une fonctionnalité.

Le planning

L'équipe de Zend utilise le service AgileZen comme planning de développement. AgileZen est un outil de gestion de projet en ligne, disponible à l'URL suivante : <http://www.agilezen.com>. Si vous désirez suivre le projet sur l'outil, il est nécessaire de demander à être invité sur le projet, vous trouverez les informations nécessaires sur le site devzone zend.com. Sur la page du projet, vous retrouverez les tâches en cours, ce qu'il reste à faire et ce qui est fait.

Voici un aperçu du projet sous AgileZen :

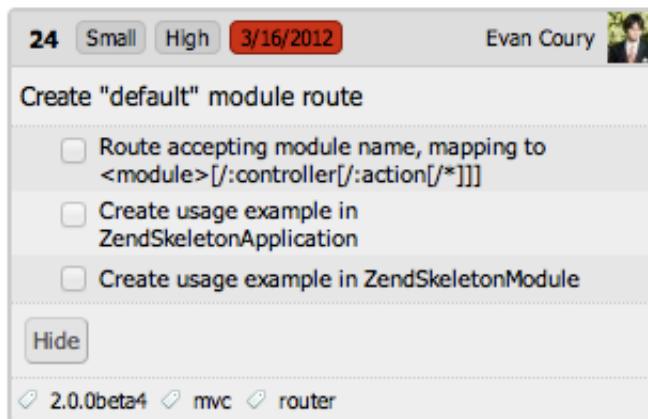
The screenshot shows the AgileZen application interface. At the top, there is a navigation bar with the title "AgileZen" and tabs for "Zend Framework 2", "Home", "Board", "Work", and "Performance".

The main area is a Kanban board with three columns:

- Ready for Work** (Left column): Contains four tasks:
 - Mass Coding Standards Fixes (49 tasks, 3 of 3 finished, 100%, 2.0.0beta5)
 - Replace file headers (43 tasks, 0 of 1 finished, 0%, 2.0.0beta5)
 - Create "default" module route (24 tasks, 0 of 3 finished, 0%, 2.0.0beta4, mvc, router)
 - Add zf2status tool link to website (28 tasks, 0 website, 2/29/2012, Matthew Weier O'Phinney)
- Work In Progress** (Middle column): Contains four tasks:
 - Zend\Di\Di instance compiler PO (51 tasks, 3 of 8 finished, 38%, 1 comment, code generation, dic, perform)
 - Console (2 tasks, 11 of 29 finished, 38%, 1 comment, 6 changesets, 2.0.0beta5, cli, mvc)
 - Escaper component (31 tasks, 5 of 11 finished, 45%, 1 comment, 2.0.0beta5, escaper, security)
 - i18n / L10n (18 tasks, 4 of 11 finished, 36%, 1 comment)
- Backlog** (Right column): Contains one task:
 - Add license files and docblocks to skeleton and Zend* modules (14 tasks, 1 of 3 finished, 33%)

Chaque colonne représente un état de tâche en cours. Cela permet de voir l'avancée de la version en cours du framework. Cette capture d'écran a été faite juste avant la sortie de la bêta 4, on remarque qu'il reste peu de choses avant la finalisation de cette version, et l'on remarque aussi le tag « 2.0.0beta5 » qui nous indique l'existence d'une bêta 5.

Il est possible aussi d'obtenir plus d'informations sur une tâche précise en cliquant sur celle-ci :



La liste des actions associées à la tâche à effectuer permet de connaître en détail ce qui va être fait et ce qu'il reste à faire. Le développeur indique parfois des commentaires liés à la tâche en cours afin de fournir plus d'informations à d'éventuels contributeurs.

Lorsqu'une tâche est terminée, celle-ci glisse dans la colonne « Complete » et est ensuite fusionnée avec le code du framework dans la branche de développement en cours. Il est alors possible d'avoir accès immédiatement à la fonctionnalité.

La gestion des versions

Le gestionnaire de versions utilisé par l'équipe de développement est Git. Le code est disponible sur Github à l'adresse suivante : <http://github.com/zendframework/zf2>. Sur ce dépôt vous retrouverez la dernière version du framework ainsi que la branche de développement mise à jour en continu. Vous y retrouverez aussi les tests unitaires et la documentation du framework.

Sur ce dépôt, vous pouvez aussi voir les demandes d'ajout de code faites par les développeurs de Zend comme celles faites par n'importe quel développeur souhaitant contribuer au framework. Ces demandes d'ajout sont appelées « Pull Requests » (PR) et sont visibles dans l'onglet du même nom : <http://github.com/zendframework/zf2/pulls>. C'est par les « Pulls Requests » que passera chaque code qui sera ajouté au framework.

L'onglet « Graphs » permet d'avoir des statistiques sur l'activité du dépôt ou encore des statistiques sur les développeurs du framework. Vous pouvez par exemple retrouver l'activité en fonction du nombre de commits effectués (<http://github.com/zendframework/zf2/graphs/commit-activity>), les statistiques de contribution des développeurs (<http://github.com/zendframework/zf2/graphs/contributors>), ou encore la liste des contributions d'un développeur, par exemple pour mon profil : <http://github.com/zendframework/zf2/commits?author=blanchonvincent>.

Généralement, les « Pulls Requests » sont divisées en trois catégories :

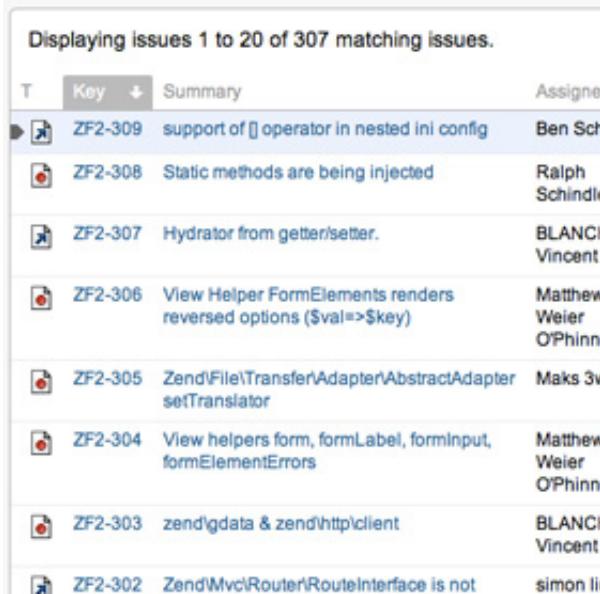
- Les bugfixs, ou corrections de bogue ;

- Les features ou ajouts de fonctionnalité ;
- La modification de documentation ou de règles de codage.

Pour les deux premiers éléments, il est impératif d'ouvrir un ticket sur le gestionnaire de bogues du framework avant de proposer une « Pull Request ».

Le bug tracker

Le gestionnaire de bogues, ou bug tracker, est disponible à l'adresse <http://framework zend.com/Issues>. Une fois inscrit, nous avons alors accès à l'ensemble des tickets ouverts pour le framework :



Displaying issues 1 to 20 of 307 matching issues.		
T	Key	Summary
	ZF2-309	support of [] operator in nested ini config
	ZF2-308	Static methods are being injected
	ZF2-307	Hydrator from getter/setter.
	ZF2-306	View Helper FormElements renders reversed options (\$val=>\$key)
	ZF2-305	Zend\File\Transfer\Adapter\AbstractAdapter::setTranslator
	ZF2-304	View helpers form, formLabel, formInput, formElementErrors
	ZF2-303	zend\gdata & zend\http\client
	ZF2-302	Zend\Mvc\Router\RouteInterface is not

Lorsque l'on souhaite ouvrir un ticket, il est seulement nécessaire de décrire le bogue ou la fonctionnalité ainsi que de choisir sur quel composant ce ticket sera relié. Le fait de relier un ticket à un composant permet d'affecter directement le ticket au développeur concerné et responsable du composant :

Create Issue

Project **Zend Framework 2.0**

Issue Type  **Bug**

Public Fields Internal Project Management Fields

Summary

Priority **Major**

Component/s Start typing to get a list of possible matches or pre

Description

Lorsque l'on écrit le ticket, si l'on souhaite proposer soi-même le correctif, il est nécessaire de l'indiquer dans le ticket afin que le travail ne soit pas fait deux fois. Le développeur responsable du composant affectera le ticket à son créateur et sera en attente de la « Pull Request ».

Corriger un bogue ou proposer une amélioration

Afin de proposer une correction de bogue ou une nouvelle fonctionnalité, il est important d'ouvrir un ticket afin d'en avertir la communauté, car ce bogue peut déjà être en cours de correction ou la fonctionnalité en cours d'implémentation par un autre développeur. Une fois cette étape faite, nous pouvons passer à la modification du code.

La première étape est de se créer un compte sur Github. En effet, sans compte, il ne va pas être possible de proposer ses modifications aux développeurs. Lorsque l'on possède son compte, il faut ensuite « fork » le projet de Zend Framework 2 depuis le bouton « fork » de la page du projet. Une fois le projet « forké », il faut récupérer le code sur son environnement de travail local. En fonction de votre poste de travail (Windows, Mac ou Linux) vous suivrez les instructions de Github. Notez qu'il existe un excellent logiciel pour Mac afin d'interagir avec celui-ci.

Lorsque le code du framework est sur notre machine locale, il est nécessaire de tirer une nouvelle branche afin d'y apporter les modifications que l'on souhaite ajouter. L'ajout d'une branche permettra de conserver la branche « master » à jour avec l'évolution du code du framework. Lorsque la branche est créée, nous allons pouvoir lui apporter toutes les modifications que l'on souhaite faire. Une fois celles-ci effectuées, il est indispensable de mettre à jour les tests unitaires liés à notre modification. Si les tests unitaires n'ont pas été modifiés en conséquence, les modifications seront refusées par l'équipe de développement.

Lorsque les modifications sont faites et que les tests unitaires sont à jour, il est alors temps de pousser notre code sur notre dépôt github. La nouvelle branche va alors être créée, et il est possible d'en faire une « Pull Request » depuis le bouton « Pull Request ». Une fois notre requête validée, nous pouvons la visualiser dans la liste des « Pull Requests » en attente d'ajout au framework : <https://github.com/zendframework/zf2/pulls>.

Récapitulatif des étapes

Voici le récapitulatif des étapes à mener avant de pouvoir contribuer au framework :

- créer un compte Github ;
- créer un compte sur le « bug tracker » ;
- ouvrir un ticket sur le « bug tracker » ;
- « fork » le projet Zend Framework 2 ;
- créer une branche pour sa modification ;
- faire ses modifications et mettre les tests unitaires à jour ;
- pousser son code local sur sa nouvelle branche ;
- faire une demande de « Pull Request » ;
- attendre et répondre aux commentaires éventuels sur sa « Pull Request » ;
- fermer le ticket une fois la « Pull Request » fusionnée dans le framework ;

Vous avez maintenant toutes les cartes en main pour contribuer au framework.

