

Framework PrototypePHP

Comment l'utiliser ?

Éric Quinton

27 septembre 2016

Table des matières

1	Présentation	1
1.1	Historique	1
1.2	Gestion des versions	2
1.3	plugins utilisés	2
1.4	Modèle MVC	3
1.5	Licences	3
I	Le contrôleur	7
2	Fonctionnement général	9
2.1	Synopsis	9
2.2	Organisation des dossiers	10
2.3	Paramètres	12
2.3.1	Paramètres généraux	12
2.3.2	Identification	13
2.3.3	Connexions aux bases de données	15
2.4	Gestion des messages	16
3	Décrire les actions	19
4	Identifier les utilisateurs et gérer les droits	23
4.1	Identifier les utilisateurs	23
4.2	Gérer les droits	24
II	Le modèle	27
5	ObjetBDD - accéder aux bases de données	29
5.1	Présentation	29
5.2	Fonctionnalités générales	29
5.2.1	Formatage des dates	29

5.2.2	Opérations d'écriture en base de données	30
5.2.3	Gestion des erreurs	30
5.3	Variables générales utilisables	30
5.4	Héritage	31
5.5	Fonctions principales	31
5.5.1	Constructeur de la classe	31
5.5.2	lire	33
5.5.3	ecrire	33
5.5.4	supprimer	34
5.5.5	supprimerChamp	34
5.5.6	getListe	34
5.5.7	getListFromParent	34
5.5.8	getListParamAsPrepared	34
5.5.9	getListeParam	34
5.5.10	ecrireTableNN	34
5.5.11	getBlobReference	35
5.5.12	encodeData	36
5.5.13	executeAsPrepared	36
5.5.14	executeSQL	36
5.5.15	formatDateDBversLocal	36
5.5.16	formatDateLocaleVersDB	36
5.5.17	utilDatesDBVersLocale	37
5.6	Utilisation avancée	37
5.6.1	Requête multi-table contenant des champs date	37
5.7	Le tableau de paramètres ObjetBDDParam	37
6	Exécuter les actions	39
6.1	Les actions standard	39
6.1.1	list	42
6.1.2	display	43
6.1.3	change	43
6.1.4	write	43
6.1.5	delete	44
III	Les vues	45
7	Les vues	47
7.1	La vue Smarty	48
7.1.1	Fonctions disponibles	48
7.1.2	Organisation de l'écran	48

TABLE DES MATIÈRES

7.1.3	Conventions de nommage	49
7.2	La vue Ajax	50
7.3	La vue CSV	50
7.4	La vue binaire	50
7.5	La vue PDF	51
8	Génération du menu	53
8.1	Fichier de description	53
8.2	Génération en mode développement	54
9	Gestion des langues	55
9.1	Formatage des dates	55
10	Compléments sur Smarty	57
10.1	Syntaxe générale	57
10.1.1	Cohabitation Javascript et Smarty	57
10.1.2	Affichage d'une variable	57
10.1.3	Affichage d'une liste	58
10.1.4	Les tests	60
10.1.5	Les variables internes	60
10.2	Affichage des libellés en fonction de la langue	61
10.3	Organisation des formulaires de saisie	61
IV	Sécurité et implémentation	63
11	Mécanismes de sécurité et mise en production	65
11.1	Intégrer le transcodage des clés	65
12	Mise en production	67
12.1	Configuration et installation générale	67
12.1.1	Configuration du serveur web	67
12.1.2	Nettoyage de l'application et contrôles à réaliser	67
12.1.3	Installation de la base de données des droits	67
12.1.4	Nettoyage des comptes par défaut	67
12.2	Travailler avec plusieurs applications différentes à partir du même code	67

Chapitre 1

Présentation

Historique

Au début des années 2000, PHP commençait à être largement utilisé pour créer des applications web. Certains frameworks étaient déjà présents, mais ils présentaient souvent des difficultés pour les appréhender et n'étaient pas forcément adaptés aux besoins de l'époque (performance souvent insuffisante en raison d'un chargement systématique de toutes les classes, fonctionnement exclusivement objet, etc.). De plus, ils ne permettaient que difficilement de remplacer certains composants par d'autres.

Des outils comme Smarty, un moteur de templates qui permet de séparer le code HTML du code PHP commençaient à se faire une place. On trouvait également des bibliothèques assez élaborées comme PHPGACL pour gérer les droits de manière particulièrement pertinente.

La gestion des bases de données n'était pas des plus optimales, et un souvent un peu trop conceptuelle.

PrototypePHP a été créé pour assembler divers outils disponibles, selon la conception qu'en avait l'auteur à l'époque. Il était loin d'être parfait et a évolué de multiples fois, pour intégrer une approche mvc, puis des contraintes de sécurité, etc. Toutefois, les fondements de départ sont restés quasiment identiques, même si certaines évolutions ont été intégrées :

- des actions décrites dans un fichier xml, qui est utilisé pour générer le menu en fonction des droits détenus par l'utilisateur ;
- une gestion des droits basée sur PHPGACL. Si le produit initial a été abandonné, sa philosophie a été conservée ;
- une séparation du code PHP et HTML avec l'utilisation de SMARTY ;
- un accès aux tables de la base de données réalisé par l'intermédiaire d'une classe dédiée à cet usage, ObjetBDD, qui contient des fonctions très simples à manipuler, comme `ecrire($data)`, `lire($id)`, `supprimer($id)`. La connexion à

la base de données, à l'époque réalisée en utilisant la bibliothèque ADODB, a été remplacée par PDO ;

- un support de l'identification selon quatre modalités : base de données, annuaire LDAP, annuaire LDAP puis base de données, et connexion via un serveur CAS ;
- un souci permanent de la performance, lié au passé de son concepteur ¹.

La première version publiée l'a été en 2008, dans sourceforge (<https://sourceforge.net/projects/prototypephp/>). Depuis quelques années, elle est disponible dans github (<https://github.com/equinton/prototypephp>), la branche active étant la branche *bootstrap*, créée au moment du basculement de l'affichage en utilisant les fonctionnalités de ce produit.

Si le principe général d'une conception MVC a prévalu depuis plusieurs années, des améliorations récentes, notamment dans la gestion des vues, a été apportée. À partir de septembre 2016, une meilleure gestion des droits a été implémentée, notamment dans les contextes de travail avec un annuaire d'entreprise LDAP. Il n'est pas impossible également que le support de Shibboleth puisse être intégré dans le futur, notamment quand des bibliothèques prêtes à l'emploi seront disponibles.

Gestion des versions

Le framework est mis à jour en parallèle aux développements de logiciels bâtis à partir de celui-ci. Le code disponible reflète donc les retranscriptions des modifications apportées au gré des évolutions envisagées par son concepteur.

Il n'existe ainsi plus depuis plusieurs mois de gestion de version : le plus simple est de se référer à la date du commit, en utilisant la branche *bootstrap*, qui est celle de travail actuel.

plugins utilisés

Les bibliothèques suivantes sont installées dans le framework :

- pour le code PHP :
 - ObjetBDD (conçu par le développeur du framework), qui gère l'interface avec la base de données ;
 - SMARTY (<http://www.smarty.net>), le moteur de templates ;
 - phpCAS (<https://wiki.jasig.org/display/CASC/phpCAS>), pour la connexion par l'intermédiaire d'un serveur CAS ;

1. il a commencé sa carrière à une époque où les ressources informatiques étaient rares, chères, et dont la puissance était limitée

- et d'autres bibliothèques disponibles dans le framework, mais utilisées uniquement si nécessaire, comme tcpdf (<https://sourceforge.net/projects/tcpdf/files/>), odtphp (<https://sourceforge.net/projects/odtphp/>), openoffice_generation, phpExcelReader (sourceforge.net/projects/phpexcelreader/)...
- pour l'affichage et la conception des pages web, le recours au javascript est omniprésent :
 - JQuery, JQueryUI, et des plugins pour les sélections des dates ;
 - DataTables et ses plugins ;
 - OpenLayers pour l'affichage des cartes ;
 - bootstrap pour la prise en compte de l'affichage sur le mode *responsive* ;
 - ...

Elles sont mises à jour régulièrement, mais il est préférable de vérifier si de nouvelles versions sont disponibles avant de procéder à une mise en production.

Modèle MVC

Le framework est basé sur un modèle MVC, qui présente les caractéristiques suivantes :

- le contrôleur est unique, les actions et les droits associés sont décrits dans un fichier unique ;
- les vues sont héritées d'une classe non instanciable, avec des classes dédiées à l'usage (html via Smarty, ajax, csv pour le moment) ;
- le modèle est constitué de deux types d'objets : des classes héritées d'ObjetBDD pour gérer les échanges avec la base de données, et des fichiers de script exécutant les modules (ou actions) demandés.

Le framework n'a pas une philosophie « tout objet », comme peuvent l'être d'autres, pour tirer parti de la souplesse du php. De nombreuses fonctions permettent de faciliter et limiter le code à écrire.

Quelques classes génériques sont utilisées (une classe Message, par exemple), et l'application recourt fortement aux variables de session.

Licences

Le framework est distribué sous licence LGPL v2 et CECILL-C.

Voici la liste des licences des composants utilisés :

Framework PrototypePHP

Composant - version	Site web et usage	Langage	Licence
ObjetBDD 3.2	Accès aux tables des bases de données (ORM)	PHP	LGPL
SMARTY 3.1.24	http://www.smarty.net Moteur de templates	PHP	LGPL
PHPCAS 1.3.3	https://wiki.jasig.org/display/CASC/phpCAS Identification via un serveur CAS	PHP	Apache 2.0
fpdf17	http://www.fpdf.org/ Génération de fichiers PDF	PHP	Aucune restriction d'usage
odtphp	URL publipostage dans des fichiers ODT	PHP	
openoffice-generation	URL Génération de fichiers ODS	PHP	
phpExcelReader	URL Lecture de fichiers XLS	PHP	
tcpdf	URL Génération de fichiers PDF	PHP	
Php-clamav 0.15.8	http://php-clamav.sourceforge.net/	PHP	GPL v2
bootstrap	URL Affichage HTML	CSS et Javascript	
carhartl-jquery-cookie 1.5.1	https://github.com/carhartl/jquery-cookie Gestion des cookies dans le navigateur	Javascript	MIT
Datatables 1.10.12	http://www.datatables.net/ Affichage des tables	Javascript	MIT
datetime-moment	URL Gestion du tri des dates dans Datatables	Javascript	
moment	URL bibliothèque utilisée par le composant précédent pour le tri des dates	Javascript	
Jquery	http://jquery.com/ Fonctions d'encapsulation de Javascript	Javascript	Équivalent BSD
JqueryUI	http://jqueryui.com/ Composants graphiques associés à Jquery	Javascript	Équivalent BSD

CHAPITRE 1. PRÉSENTATION

Composant - version	Site web et usage	Langage	Licence
jquery- timepicker- addon	URL saisie de la date/heure	Javascript	
magnific- popup 0.9.9	http://dimsemenov.com/plugins/magnific-popup/ Affichage des images sous forme de pop-up	Javascript	MIT
smartmenus	URL Affichage des menus dans bootstrap	Javascript	
c3js0.4.10	http://c3js.org/ Création de graphiques	Javascript	MIT
openlayers 3.12.1	http://openlayers.org/ Affichage de cartes	Javascript	

TABLE 1.1: Liste des licences des composants utilisés

Les composants PHP sont stockés dans le dossier *plugins*, les composants Javascript dans *display/javascript*.

Première partie

Le contrôleur

Chapitre 2

Fonctionnement général

Synopsis

L'appel de toute page dans l'application passe nécessairement par l'ensemble de ces étapes :

- vérification que l'encodage des caractères transmis respecte bien l'encodage utf-8
- lecture des paramètres
- chargement des classes génériques utilisées systématiquement
- démarrage de la session, et ajout de contrôles (durée de la session ouverte...)
- lecture des paramètres en sur-écrasement, ce qui permet des implémentations multiples avec le même code
- initialisation de l'identification
- contrôles de cohérence IP (vérification que, pour une même session, l'adresse IP ne change pas)
- lancement des connexions aux bases de données (par défaut, deux connexions : une pour la base des droits, l'autre pour les données applicatives)
- décodage des variables HTML encodées (protection contre les attaques de type XSS)
- traitement du module demandé :
 - initialisation, le cas échéant, de la vue associée
 - vérification de l'identification, ou déclenchement des procédures d'identification
 - vérification des droits nécessaires pour accéder au module
 - vérification, le cas échéant, de la cinématique : les opérations de modification ne devraient être possibles que si l'opération précédente correspond à l'affichage du formulaire de saisie
 - exécution du module
 - analyse du code de retour du module, et enchaînement le cas échéant sur un autre module

- déclenchement de la vue
- enregistrement, le cas échéant, des messages destinés à SYSLOG (messages systèmes)

Organisation des dossiers

Les fichiers sont organisés selon cette arborescence :

- **database** : dossier de travail contenant la description de la base de données, la documentation pour les développeurs, les scripts. Le dossier doit être supprimé lors de la mise en production
- **display** : le seul dossier accessible. Il contient tous les fichiers nécessaires pour gérer l’affichage :
 - **CSS** : les feuilles de style
 - **images** : les icônes et images utilisées dans l’affichage des pages
 - **javascript** : l’ensemble des bibliothèques Javascript utilisées
 - **templates** : les modèles de documents utilisés par Smarty (cf. 10, *Compléments sur Smarty*, page 57)
 - **templates_c** : dossier utilisé par Smarty pour compiler les templates. Ce dossier doit être accessible en écriture par le serveur Web
- **doc** : ancien dossier, contenant un mécanisme de gestion de la documentation en ligne. N’est plus utilisé actuellement, mais pourrait être employé le cas échéant
- **framework** : le code de base du framework. Il comprend :
 - **droits** : dossier permettant de gérer les droits
 - **identification** : gestion de la connexion des utilisateurs
 - **import/import.class.php** : classe créée il y a quelques années pour gérer les imports (obsolète en grande partie)
 - **ldap/ldap.class.php** : connexion à un annuaire LDAP et récupération d’informations
 - **navigation** : programmes utilisés pour générer le menu et décoder les actions demandées à partir du fichier XML les contenant
 - **translateId/translateId.class.php** : classe permettant de transcoder les identifiants des enregistrements de la base de données, pour éviter les attaques par forçage de clé
 - de nombreux fichiers utilisés par le framework, dont le contrôleur (controller.php), des fonctions génériques (fonctions.php)...
 - **vue.class.php** : les classes utilisées pour les vues (cf. 7 *Les vues*, page 47)

- **install** : contient des scripts d'installation de la base de données (normalement à déplacer dans *database*), et le fichier **readme.txt**, décrivant les dernières nouveautés
- **locales** : dossier contenant les fichiers de langue (fr.php et en.php)
- **modules** : dossier contenant le code spécifique de l'application. Il est organisé ainsi :
 - **classes** : les classes nécessaires pour l'application
 - **example** : des exemples de codage
 - les autres dossiers sont libres et contiennent les modules de l'application
 - **beforeDisplay.php** : fichier appelé systématiquement avant l'affichage des pages HTML
 - **beforeSession.inc.php** : fichier appelé systématiquement avant le démarrage de la session. Il permet de déclarer les librairies qui sont nécessaires pour instancier des classes stockées en variables de session
 - **common.inc.php** : fichier appelé systématiquement avant le traitement des modules
 - **fonctions.php** : fonctions déclarées par le programmeur et disponibles dans toute l'application
 - **postLogin.php** : script exécuté uniquement quand un utilisateur s'est identifié
- **param** : dossier contenant les paramètres de l'application :
 - **actions.xml** : fichier contenant la description de l'ensemble des modules utilisables, avec les droits associés et le type de vue à utiliser
 - **menu.xml** : description du menu qui sera généré
 - **param.default.inc.php** : les paramètres par défaut
 - **param.inc.php** : paramètres en écrasement, spécifiques de l'implémentation. Ce fichier n'est jamais livré lors des mises à jour, pour éviter la suppression des paramètres de base de données, par exemple
 - **param.inc.php.dist** : fichier d'exemple de *param.inc.php*, à renommer et à mettre à jour lors de l'installation d'une nouvelle implémentation
- **plugins** : dossier contenant les bibliothèques tierces, comme Smarty, ObjetBDD (maintenant intégré au framework)...
- **temp** : dossier de stockage temporaire, qui doit être accessible en écriture au serveur web. Les fichiers présents dans celui-ci ont une durée de vie de 24 heures (suppression lors de la connexion d'un utilisateur)
- **test** : dossier utilisé pour réaliser certains tests. Doit être systématiquement supprimé lors de la mise en production

Seuls le fichier *index.php*, à la racine, les dossiers *display* et *test* sont accessibles directement. Les autres dossiers sont protégés par des fichiers *.htaccess*.

Paramètres

Les paramètres utilisés dans l'application sont gérés avec 3 fichiers différents :

- **param/param.default.inc.php** : contient l'ensemble des paramètres utilisés ;
- **param/param.inc.php** : contient ceux issus du fichier précédent, qui sont adaptés à l'implémentation ;
- **param.ini** : fichier contenant les paramètres spécifiques du nom DNS de l'application (par exemple, schéma particulier associé au nom du site). Pour plus d'informations sur ce point, consultez le chapitre 12.2 *Travailler avec plusieurs applications différentes à partir du même code*, page 67.

Voici la description de l'ensemble des paramètres :

Paramètres généraux

Variable	Signification
APPLI_version	Numéro de version de l'application
APPLI_versiondate	Date de la version
language	Langue par défaut
DEFAULT_formatdate	Format par défaut d'affichage des dates
navigationxml	nom du fichier XML contenant la description des modules exécutables
APPLI_session_ttl	durée de la session, en secondes
APPLI_cookie_ttl	durée de vie par défaut des cookies, en secondes
APPLI_path_stockage_session	obsolète
LOG_duree	Durée de conservation des traces des actions réalisées, en jours
APPLI_mail	Adresse pour déclarer les incidents (mail ou non)
APPLI_titre	Nom de l'application qui sera affiché (cas où le code est utilisé par plusieurs entrées différentes)
APPLI_code	Code interne de l'application. Utilisé dans certains cas
APPLI_fds	Feuille de style utilisée par défaut (obsolète)
APPLI_address	Adresse DNS de l'application. Utilisée en cas d'identification CAS (adresse de retour)
APPLI_modeDeveloppement	si à true, certaines opérations sont réalisées dans un contexte de développement (affichage de messages, recalcul systématique du menu...)

CHAPITRE 2. FONCTIONNEMENT GÉNÉRAL

Variable	Signification
APPLI_notSSL	utilisé en développement, si l'application ne fonctionne pas en mode SSL (déconseillé)
APPLI_utf8	systématiquement à true (plus de support des autres encodages)
APPLI_menufile	nom du fichier XML contenant la description du menu
APPLI_temp	nom du dossier utilisé pour stocker les fichiers temporaires
APPLI_moduleDroitKO	nom du module appelé en cas de refus d'accès pour un problème de droits
APPLI_moduleErrorBefore	nom du module appelé en cas de problème lié à la cinématique de l'application
APPLI_moduleNoLogin	nom du module appelé en cas d'échec d'identification
paramIniFile	nom du fichier contenant les paramètres spécifiques liés au DNS utilisé (<i>cf. 12.2 Travailler avec plusieurs applications différentes à partir du même code</i> , page 67)
SMARTY_param	Paramètres utilisés par le moteur de templates SMARTY
SMARTY_variables	variables systématiquement transmises à SMARTY et utilisées lors de l'affichage général
ERROR_display	Affiche les erreurs à l'écran (mode développement)
OBJETBDD_debugmode	0 : pas d'affichage de message d'erreur, 1, affichage des messages d'erreur, 2 : affichage de toutes les commandes SQL générées par ObjetBDD
ADODB_debugmode	obsolète

TABLE 2.1: Variables générales de l'application

Identification

Variable	Signification
ident_type	Type d'identification supporté. L'application peut gérer BDD (uniquement en base de données), LDAP (uniquement à partir d'un annuaire LDAP) LDAP-BDD (d'abord identification en annuaire LDAP, puis en base de données), et CAS (serveur d'identification <i>Common Access Service</i>)
CAS_plugin	Nom du plugin utilisé pour une connexion CAS
CAS_address	Adresse du serveur CAS
CAS_port	Systématiquement 443 (connexion chiffrée)
LDAP	tableau contenant tous les paramètres nécessaires pour une identification LDAP
privateKey	clé privée utilisée pour générer les jetons d'identification
pubKey	clé publique utilisée pour générer les jetons d'identification
tokenIdentityValidity	durée de validité, en secondes, des jetons d'identification

TABLE 2.2: Variables utilisées pour paramétrer l'identification

Voici le contenu des variables du tableau LDAP :

Variable	Signification
address	adresse de l'annuaire
port	389 en mode non chiffré, 636 en mode chiffré
rdn	compte de connexion, si nécessaire
basedn	base de recherche des utilisateurs
user_attrib	nom du champ contenant le login à tester
v3	toujours à <i>true</i>
tls	<i>true</i> en mode chiffré
groupSupport	true si l'application recherche les groupes d'appartenance du login dans l'annuaire
groupAttrib	Nom de l'attribut contenant la liste des groupes d'appartenance
commonNameAttrib	Nom de l'attribut contenant le nom de l'utilisateur
mailAttrib	Nom de l'attribut contenant l'adresse mail de l'utilisateur

CHAPITRE 2. FONCTIONNEMENT GÉNÉRAL

Variable	Signification
attributgroupname	Attribut contenant le nom du groupe lors de la recherche des groupes (cn par défaut)
attributloginname	attribut contenant les membres d'un groupe
basedngroup	base de recherche des groupes

TABLE 2.3: Variables utilisées pour paramétrer l'accès à l'annuaire LDAP

Connexions aux bases de données

Deux connexions sont systématiquement implémentées : l'une à la base de données contenant la gestion des droits, et l'autre à celle contenant les données propres à l'application.

Variable	Signification
BDD_login	compte de connexion à la base de données
BDD_passwd	mot de passe associé
BDD_dsn	adresse de la base de données sous forme normalisée
BDD_schema	schéma utilisé (plusieurs schémas peuvent être décrits, en les séparant par une virgule - fonctionnement propre à Postgresql)
GACL_dblogin	compte de connexion à la base de données des droits
GACL_dbpasswd	mot de passe associé
GACL_dsn	adresse normalisée
GACL_schema	schéma utilisé
GACL_aco	nom du code de l'application utilisé dans la gestion des droits (<i>cf. 4 Identifier les utilisateurs et gérer les droits, page 23</i>)

TABLE 2.4: Variables utilisées pour paramétrer les connexions

Il est possible de créer des comptes séparés, voire de ne donner accès qu'en lecture à la base des droits (à l'exception de la table *log*, qui contient la trace de toutes les actions demandées).

Gestion des messages

Une classe est instanciée systématiquement pour gérer les messages, la classe *Message*. Deux types de messages sont pris en compte :

- les messages envoyés au navigateur, à destination de l'utilisateur ;
- les messages enregistrés dans Syslog, le mécanisme de gestion des messages systèmes de Linux.

Les messages sont enregistrés dans un tableau, qui sera ensuite dépilé pour générer les textes soit à afficher, soit à stocker dans Syslog.

La classe dispose des fonctions suivantes :

fonction	Objectif
<code>__construct(\$displaySyslog = false)</code>	Constructeur de la classe. La variable permet d'indiquer si les messages destinés à Syslog sont également affichés à l'écran (mode par défaut en développement)
<code>set(\$value)</code>	Ajoute un nouveau libellé utilisateur
<code>setSyslog(\$value)</code>	Ajout un nouveau message système
<code>get()</code>	Retourne le tableau contenant l'ensemble des messages, avec ou sans les messages systèmes, selon le mode indiqué dans le constructeur
<code>getAsHtml()</code>	Formate les messages pour les envoyer au navigateur. Chaque message est séparé par un retour à la ligne. Les libellés sont encodés en HTML avant d'être envoyés
<code>sendSyslog()</code>	Génère un message dans Syslog. Actuellement, le message est toujours de type NOTICE.

TABLE 2.5: Fonctions utilisables dans la classe Message

Les messages sont systématiquement transmis à la vue Smarty, et l'envoi des messages systèmes est la dernière action réalisée avant l'affichage de la vue.

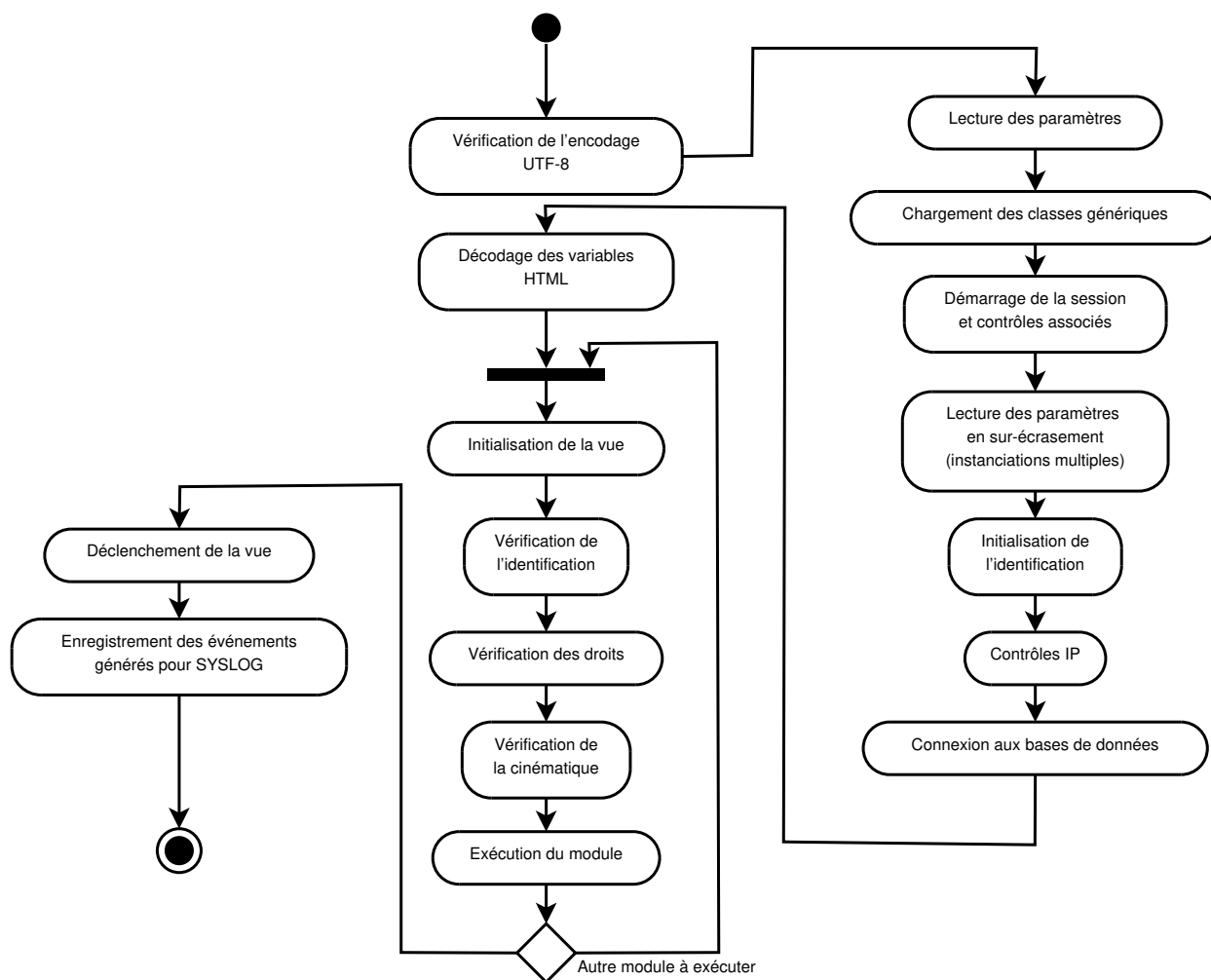


FIGURE 2.1 – Synopsis général de fonctionnement du contrôleur

Chapitre 3

Décrire les actions

Les actions possibles dans le logiciel sont décrites dans un fichier, par défaut *param/actions.xml*. C'est un fichier XML dont la racine s'appelle *navigation*.

Une action est la conjonction entre un contexte et une opération, par exemple *poissonList* pour afficher la liste des poissons, *poissonChange* pour afficher la page de modification d'un poisson, *poissonWrite* ou *poissonDelete* pour déclencher l'écriture en base de données.

Dans le contexte de ce framework, l'action s'appelle *module* (nom du champ transmis depuis le navigateur). L'attribut *action* contient le nom du fichier PHP appelé. Il est associé à l'attribut *param*, qui permet d'indiquer le détail de l'action à réaliser (par exemple, *list* ou *change*).

Voici la liste des attributs disponibles pour un module (ou une action) :

Attribut	Requis	Signification
action	X	nom de la page PHP à exécuter (accès relatif depuis la racine de l'application)
param		paramètre analysé dans la page, pour savoir quelle action doit être réalisée. Par convention, les actions possibles sont les suivantes : list, read, change, write, delete, ou autre action
droits		Liste des droits nécessaires pour exécuter l'action. Si plusieurs droits sont possibles, ils doivent être séparés par une virgule
loginrequis		Indique, en l'absence de droits spécifiques, si l'action nécessite d'être connecté. Vaut 1 si la connexion est requise

Attribut	Requis	Signification
modulebefore		Pour les opérations d'écriture, permet d'indiquer le nom du module qui doit impérativement être exécuté avant. Cela limite les risques d'attaques de type CSRF et les rafraîchissements intempestifs dans les formulaires. Plusieurs modules peuvent être indiqués, en les séparant par une virgule
retourok		indique le nom du module qui sera exécuté si le code de retour (variable \$module_coderetour) vaut 1
retourko		indique le nom du module qui sera exécuté si le code de retour (variable \$module_coderetour) vaut -1 (échec d'exécution)
type	(X)	pour les modules envoyant des données au navigateur, indique le type de la vue qui sera utilisée. Les valeurs possibles sont smarty ou html (même vue), ajax, pdf, csv
droitko		nom du module appelé si les droits ne sont pas suffisants pour exécuter l'action demandée

TABLE 3.1: Liste des attributs utilisables pour décrire une action

Le module *model* n'est pas analysé, il sert à montrer l'ensemble des options possibles.

Le module *default* correspond au module appelé par défaut, si l'application est appelée sans indiquer de nom de module (variable *module* non transmise soit dans le lien, soit dans le formulaire).

Voici quelques exemples d'utilisation :

```
<appliList action="framework/droits/appli.php"
  param="list" droits="admin" retourlogin="1"
  type="smarty" />
<appliDisplay action="framework/droits/appli.php"
  param="display" droits="admin" type="smarty"
  />
<appliChange action="framework/droits/appli.php"
  param="change" droits="admin" type="smarty"/>
<appliWrite action="framework/droits/appli.php"
  param="write" droits="admin" retourok="
  appliDisplay" retourko="appliChange" modulebefore
  ="appliChange" />
```

```
<appliDelete action="framework/droits/appli.php"
  param="delete" droits="admin" retourko="
  appliList" retourko="appliChange" modulebefore
  ="appliChange"/>
```

Il s'agit des modules utilisés dans la gestion des droits. Ils nécessitent tous que l'utilisateur dispose du droit *admin*. Une seule page est appelée (*appli.php*), l'action à réaliser étant analysée à partir de l'attribut *param*.

Les modules *appliWrite* et *appliDelete* ne génèrent pas directement d'affichage : ils sont là uniquement pour écrire des informations dans la base de données. Par contre, ils enchaînent, en fonction de leur code de retour, soit sur le réaffichage du formulaire de saisie, soit sur le retour au détail ou à la liste. Ces deux modules ne peuvent être exécutés que si le précédent est *appliChange*, c'est à dire si le formulaire de saisie a été affiché.

Chapitre 4

Identifier les utilisateurs et gérer les droits

Identifier les utilisateurs

Quatre modes d'identification sont prévus dans le logiciel :

- uniquement dans le logiciel (comptes stockés dans la base de données) ;
- à partir d'un annuaire LDAP ;
- d'abord en recherchant dans l'annuaire LDAP, puis ensuite dans la base des comptes de l'application (mode mixte) ;
- auprès d'un serveur CAS.

L'identification retenue est déclarée dans les paramètres généraux (*cf.* 2.3.2 *Identification*, page 13).

Si l'identification à partir d'un annuaire LDAP ou d'un serveur CAS ne nécessite guère d'autres informations que celles décrites dans les paramètres, l'identification par base de données comprend des mécanismes particuliers pour protéger les mots de passe et limiter les risques associés. Voici les règles imposées lors de la création d'un mot de passe :

- il doit avoir une longueur minimale de 8 caractères ;
- il doit comporter 3 jeux de caractères différents (minuscules, majuscules, chiffres et autres caractères) ;
- il n'est pas possible de réutiliser un mot de passe pour le même compte.

Les mots de passe sont salés (utilisation du login) et chiffrés (chiffrement SHA-256).

L'écran de création propose un bouton de génération automatique d'un mot de passe, qui devra être transmis à l'utilisateur, en lui demandant d'en changer (il peut modifier son mot de passe une fois connecté).

Dans sa version actuelle, le framework ne dispose pas d'un mécanisme de récupération par envoi de mails.

Les mots de passe n'expirent pas.

Gérer les droits

Les droits sont gérés selon le principe initialement utilisé dans la bibliothèque PHPGACL, aujourd'hui obsolète.

Les logins sont déclarés dans des groupes organisés de manière hiérarchique : un groupe hérite des droits attribués à ses parents.

Les droits utilisés dans le logiciel sont associés à des groupes. Il est possible d'attribuer plusieurs droits à un même groupe, et un droit peut être détenu par des groupes différents.

Si le paramètre `$LDAP["groupSupport"]` est positionné à `true`, les groupes dont fait partie le compte LDAP sont également récupérés, et peuvent être détenteurs de droits dans le logiciel (le nom des groupes est sensible à la casse).

Voici le schéma des tables utilisées pour gérer les droits :

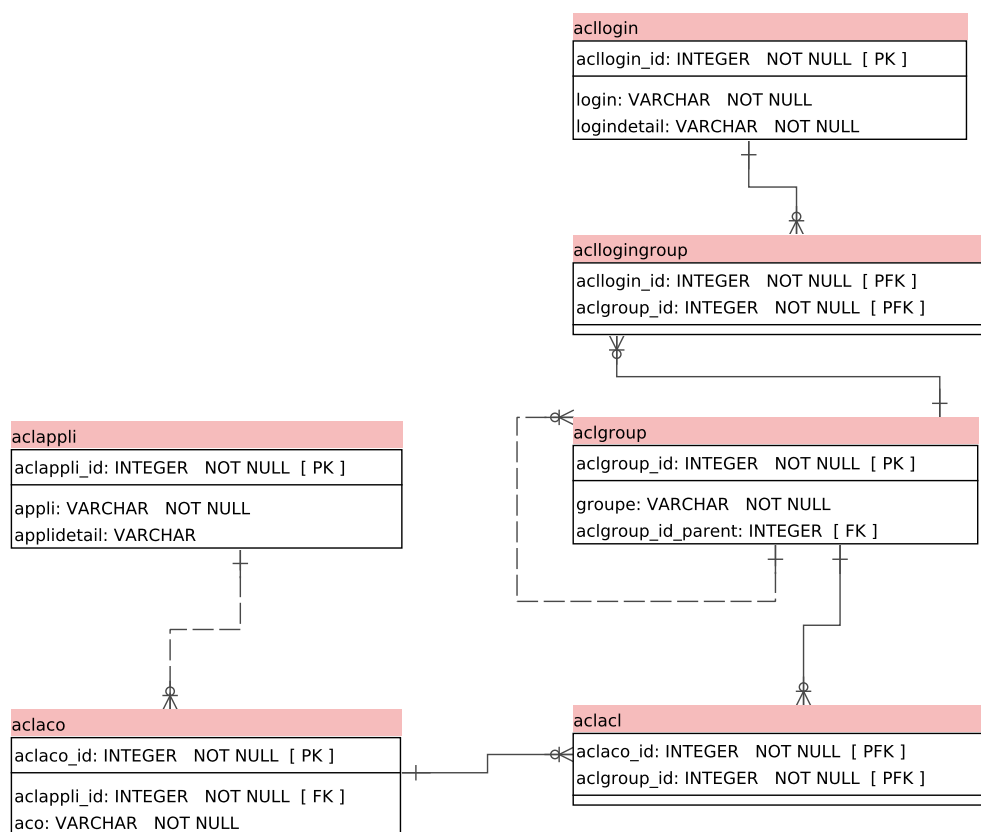


FIGURE 4.1 – Schéma des tables utilisées pour gérer les droits

CHAPITRE 4. IDENTIFIER LES UTILISATEURS ET GÉRER LES DROITS

Voici la description des tables :

aclogin Liste des logins utilisés. Si un compte est créé dans la base locale d'identification, un enregistrement est également créé dans cette table. Pour les identifications LDAP ou CAS, ils doivent être identiques. Si seuls les groupes LDAP sont utilisés pour un compte, il n'a pas besoin d'être décrit ici

aclappli Liste des applications gérées. Il est ainsi possible de gérer, à partir de la même base de données, plusieurs ensembles de droits, qui utilisent les mêmes logins

aclaco liste des droits déclarés dans l'application

aclgroup Liste des groupes contenant les logins, et qui détiennent les droits. Un groupe peut hériter d'un autre groupe. Les droits associés au groupe parent sont également attribués au groupe hérité.

acloggingroup table permettant de déclarer les logins associés à un groupe

aclacl table décrivant les droits détenus par un groupe

Le module d'administration permet de saisir toutes ces informations. Il faut que l'utilisateur dispose du droit *admin*, c'est à dire faire partie du groupe *admin* (configuration par défaut à l'initialisation de la base des droits) pour pouvoir accéder à ces fonctions.

Deuxième partie

Le modèle

Chapitre 5

ObjetBDD - accéder aux bases de données

Présentation

ObjetBDD est une classe qui sert d'interface entre l'application et la base de données. Elle a été créée pour simplifier les requêtes, seules celles d'interrogation spécifiques devront être écrites.

Historiquement, ObjetBDD travaillait avec ADODB, une classe qui encapsulait la connexion à la base de données. Avec la sortie de PDO, la classe a été adaptée pour utiliser des connexions PDO. Elle était également prévue pour fonctionner initialement avec Sybase ASE et MySQL. Les récentes évolutions ont porté sur le support des bases PostgreSQL : il n'est pas certain que toutes les fonctionnalités soient disponibles pour MySQL ou Sybase ASE.

Les fonctions initiales ont été modifiées ou complétées pour supporter maintenant les requêtes préparées.

Fonctionnalités générales

Formatage des dates

Les dates stockées dans les bases de données sont dans un format difficilement utilisable. La classe transforme automatiquement les dates dans le format français par défaut (mais d'autres formats possibles).

Elle est également capable de transformer les dates reçues du navigateur au format de stockage. Le format de saisie est libre : la plupart des séparateurs sont supportés, l'année est rajoutée automatiquement, etc.

Le formatage des date inclut également les dates/heures.

Opérations d'écriture en base de données

La classe dispose de deux fonctions pour écrire les informations : `ecrire()` et `supprimer()`. La fonction `ecrire()` va décider s'il faut réaliser un insert ou un update, en fonction de la clé fournie. Par convention, si la clé vaut 0, un insert sera réalisé.

Gestion des erreurs

En cas d'échec d'exécution d'une requête SQL, la classe génère une exception.

Variables générales utilisables

Ces variables sont toutes publiques.

Variable	Type	Signification
connection	PDO	instance PDO. Peut être utilisée pour instancier une nouvelle classe basée sur <code>ObjetBDD</code> à l'intérieur d'une fonction
id_auto	entier	Si à 1, la classe gère la création automatique des identifiants. Si à 2, l'identifiant est généré manuellement, avec une requête de type <i>max(id)</i>
formatDate	entier	0 : amj, 1 : jma (défaut), 2 : mja
debug_mode	entier	0 : pas de mode de débogage, 1 : affichage des messages d'erreur, 2 : affichage de toutes les commandes SQL générées
error_data	tableau	liste de toutes les erreurs détectées lors de la vérification des données. \$errorData[["code"]] : code d'erreur : 0 : non précisé 1 : champ non numérique 2 : champ texte trop grand 3 : masque (pattern) non conforme 4 : champ obligatoire vide \$errorData[["colonne"]] : champ concerne \$errorData[["valeur"]] : valeur initiale
srid	numérique	Valeur du srid pour les variables de type Postgis
quoteIdentifier	caractère	caractère utilisé pour encadrer les noms des colonnes dans les requêtes (pour les colonnes contenant une majuscule ou un accent)

Variable	Type	Signification
transformComma	entier	Si à 1 (défaut), les virgules sont transformées en points pour les nombres décimaux

TABLE 5.1: Liste des variables utilisables dans ObjetBDD

En principe, les variables sont initialisées lors de l’instanciation de la classe, mais peuvent être modifiées à la volée, si nécessaire.

La classe est conçue pour fonctionner en UTF8.

Héritage

La classe `ObjetBDD` n’est pas instanciable, et doit donc être héritée. En particulier, le constructeur de la classe doit être surchargé pour rendre la classe opérante.

Fonctions principales

Constructeur de la classe

```
function __construct(PDO &$p_connection, array
    $param = array())
```

Paramètres

La fonction doit recevoir une instance PDO, correspondant à une connexion déjà réalisée à la base de données. Cette instance PDO est stockée ensuite dans la variable *connection*, qui peut être réutilisée si d’autres classes héritées sont à instancier à l’intérieur du code.

Le tableau *param* comprend, si nécessaire, l’ensemble des variables globales à mettre à jour.

Surcharge

Le constructeur doit être impérativement être surchargé, avec le code minimal suivant (exemple) :

```
function __construct($bdd, $param = array()) {
    $this->table = "acllogin";
    $this->colonnes = array (
        "acllogin_id" => array (
            "type" => 1,
```

```

        "key" => 1,
        "requis" => 1,
        "defaultValue" => 0
    ),
    "login" => array (
        "requis" => 1
    ),
    "logindetail" => array (
        "type" => 0,
        "requis" => 1
    )
);
parent::__construct ( $bdd, $param );
}

```

table doit correspondre au nom de la table (sans tenir compte du schéma, traité lors de la connexion à la base de données).

colonnes contient la description des colonnes de la table. Chaque colonne doit être nommée, et contient un tableau, dont les attributs possibles sont les suivants :

Variable	Signification
type	0 : varchar 1 : numérique (y compris décimaux) 2 : date 3 : datetime 4 : champ Postgis
requis	Si à 1, le contenu doit être fourni pour réaliser l'écriture
key	Si à 1, l'attribut est utilisé comme clé primaire (en principe, n'utiliser que des clés mono-attributs, même si la classe devrait être capable de gérer des clés multiples)
defaultValue	valeur par défaut. Il est possible d'indiquer le nom d'une fonction (entre guillemets). Parmi celles-ci, il est possible d'utiliser : getDateJour : retourne la date du jour getDateHeure : retourne la date et l'heure courante getLogin : retourne la valeur de la variable \$_SESSION["login"]
parentAttrib	si vaut 1, la valeur est utilisée comme clé étrangère principale de l'enregistrement

Variable	Signification
longueur	pour les champs de type varchar, indique la longueur maximale autorisée (attention au codage UTF-8, les caractères accentués étant comptés pour 2)
pattern	pattern traité par expression régulière, pour tester la correspondance de l'information fournie au modèle décrit

TABLE 5.2: Liste des attributs permettant de décrire les colonnes de la table

Les deux derniers attributs sont toujours utilisables, mais en rarement employés.

lire

```
lire($id, $getDefault = true, $parentValue = 0)
```

Fonction permettant de récupérer un enregistrement. Elle accepte les paramètres suivants :

Variable	Signification
id	clé de l'enregistrement
getDefault	si à <i>true</i> , récupère les valeurs par défaut si l'enregistrement n'existe pas dans la base (initialisation d'une saisie, par exemple)
parentValue	clé de l'enregistrement parent. Si <i>getDefault</i> vaut <i>true</i> , pré-remplit l'attribut qui contient la valeur <i>parentAttrib</i> avec la clé fournie dans <i>parentValue</i>

TABLE 5.3: Liste des paramètres de la fonction lire

La fonction retourne le tableau associatif correspondant.

ecrire

```
ecrire($data)
```

Déclenche l'écriture des informations dans la base de données. *\$data* doit être un tableau qui comprend les attributs à écrire (au minimum, les attributs déclarés comme obligatoires).

Le nom des attributs fournis doit correspondre exactement au nom des colonnes.

En principe, ce tableau correspond à la variable `$_REQUEST`.

La fonction génère soit une commande insert, soit une commande update. En principe, la commande insert est générée si la clé fournie vaut 0.

Elle retourne la clé modifiée ou créée.

supprimer

```
supprimer($id)
```

Permet de supprimer un enregistrement, à partir de sa clé. Attention : la fonction ne gère pas les suppressions en cascade, si ce n'est pas prévu directement dans la base de données.

supprimerChamp

```
supprimerChamp($id, $champ)
```

Fonction très pratique pour supprimer tous les enregistrements fils. Elle génère une requête du type :

```
delete from table where :champ = :id;
```

getListe

```
getListe($order = "")
```

Fonction récupérant l'ensemble des enregistrements d'une table, triés ou non selon le contenu de la variable \$order.

getListFromParent

```
function getListFromParent($parentId, $order = "")
```

Retourne la liste des enregistrements fils correspondant à la clé étrangère \$parentId. Le résultat peut ou non être trié selon les paramètres définis dans la seconde variable.

getListParamAsPrepared

```
function getListeParamAsPrepared($sql, $data)
```

Permet de récupérer une liste d'enregistrements à partir de la requête SQL fournie et du tableau des données à insérer (requêtes préparées PDO), avec transformation des dates

getListeParam

```
function getListeParam($sql)
```

Exécute la requête et retourne la liste des enregistrements correspondants, avec transformation des dates.

Attention : cette fonction ne gère pas la préparation des requêtes : il importe au codeur d'en tenir compte pour éviter les risques d'injection de code. Elle ne devrait être utilisée que dans les cas où une requête préparée ne peut être utilisée.

ecrireTableNN


```
ecrireTableNN($nomTable, $nomCle1, $nomCle2, $id,
    $lignes)
```

Fonction permettant de mettre à jour les tables de relation NN, selon le schéma suivant :

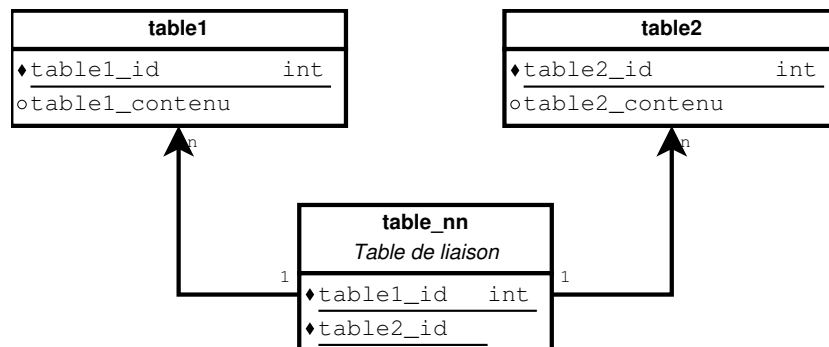


FIGURE 5.1 – Structure d’une liaison N-N

En général, la saisie de ce type de liaisons est effectuée par des cases à cocher, ce qui permet de récupérer un tableau contenant la liste des clés de la table2 (champs html `<input type="checkbox" name="attribut[]">`).

Les arguments à indiquer sont les suivants :

Variable	Signification
nomTable	Nom de la table NN (table_nn dans notre exemple)
nomCle1	nom de l’attribut contenant la clé de la table principale
nomCle2	nom de l’attribut contenant les clés de la table secondaire
id	valeur de la clé de la table principale
lignes	tableau contenant les valeurs de la table secondaire à conserver ou à rajouter

TABLE 5.4: Liste des paramètres de la fonction `ecrireTableNN`

La fonction ne génère que les requêtes de modification nécessaires (insertion ou suppression). Elle permet d’éviter de déclarer une instantiation d’ObjetBDD pour la table_nn.

getBlobReference

```
function getBlobReference($id, $fieldName)
```

Fonction permettant de récupérer un champ binaire stocké dans la base de données. PDO retourne l'identifiant interne PHP du fichier temporaire contenant l'information binaire lu.

Arguments :

Variable	Signification
id	Clé de l'enregistrement
fieldName	nom de la colonne contenant l'information binaire

TABLE 5.5: Liste des paramètres de la fonction getBlobReference

encodeData

```
encodeData($data)
```

Fonction encodant les quotes comprises dans les champs du tableau *data*, pour toutes les requêtes SQL directes (exécution ne passant pas par le mécanisme des requêtes préparées).

executeAsPrepared

```
function executeAsPrepared($sql, $data, $onlyExecute  
    = false)
```

Fonction exécutant la requête fournie sous forme de requête préparée. Les variables à insérer sont décrites dans le tableau *data*. Si l'attribut *\$onlyExecute* vaut true, la fonction ne retourne pas de résultat.

executeSQL

```
function executeSQL($ls_sql) {
```

Exécute la commande SQL, sans précaution particulière (attention aux risques d'injection).

formatDateDBversLocal

```
function formatDateDBversLocal($date, $type = 2)
```

Transforme la date, au format de la base de données, vers le format lisible pour l'utilisateur.

Si *\$type* vaut 3, la fonction retourne le champ au format date/heure.

formatDateLocaleVersDB

```
function formatDateLocaleVersDB($date, $type = 2)
```

Transforme la date fournie en format géré par la base de données. Si le type vaut 3, un champ de type date/heure est attendu.

utilDatesDBVersLocale

```
function utilDatesDBVersLocale($data)
```

Transforme les dates présentes dans le tableau joint à un format lisible par l'utilisateur.

```
function utilDatesLocaleVersDB($data)
```

Transforme les dates présentes dans le tableau joint au format supporté par la base de données.

Utilisation avancée

Requête multi-table contenant des champs date

Un des cas fréquents posé par ObjetBDD est celui des requêtes manuelles qui retournent des dates. L'objectif est de les formater pour les mettre dans le même état que les dates décrites dans la table associée à la classe.

Le plus simple consiste à rajouter la colonne date complémentaire à la liste des variables juste avant d'exécuter la requête. Voici un exemple :

```
$sql = "select t1.id, t1.date1, t2.id2, t2.date2
from table1 t1
join table2 t2 using (id)";
$this->colonnes["date2"]=array("type"=>2);
return $this->getListeParam($sql);
```

Une fois le contenu de la requête récupéré, la classe pourra alors appliquer la transformation de date sur la colonne issue de la seconde table.

Le tableau de paramètres ObjetBDDParam

ObjetBDDParam est une variable contenant les paramètres par défaut utilisés pour initialiser les instances issues d'ObjetBDD. C'est en particulier assez pratique pour gérer le choix de la langue d'affichage pour les dates. Il est également possible de modifier, de manière globale dans le logiciel, le fonctionnement d'ObjetBDD.

Ce tableau devrait être passé systématiquement en paramètre de toute instantiation d'objet basé sur ObjetBDD.

Chapitre 6

Exécuter les actions

Le framework est conçu pour n'exécuter que les actions décrites dans un fichier XML (*cf. 3 Décrire les actions*, page 19).

Les actions sont identifiées par deux informations : d'une part, le nom du fichier PHP à exécuter, et d'autre part un paramètre permettant de décrire ce qu'il faut réaliser précisément. Ce second paramètre pourrait ne pas être utilisé, mais cela impliquerait un fichier par action. Le framework a été conçu pour obtenir un bon équilibre entre le nombre de fichiers et la navigation dans le code.

Les actions standard

En général, on identifie facilement les actions suivants sur un type d'objet :

- l'affichage d'une boîte de recherche et de la liste des dossiers correspondants ;
- l'affichage du détail d'un enregistrement ;
- l'affichage de la page permettant de créer ou de modifier un enregistrement ;
- l'écriture des informations en base de données ;
- la suppression d'une fiche.

Par convention, ces actions sont nommées **list**, **display**, **change**, **write** et **delete**.

Voici un exemple de code standard utilisé pour traiter tous les modules (fichier `modules/example/example.php`, à recopier et à adapter...) :

```
include_once 'modules/example/example.class.php';
$dataClass = new Example($bdd,$ObjetBDDParam);
$keyName = "example_id";
$id = $_REQUEST[$keyName];

switch ($t_module["param"]) {
    case "list":
        /*
```

```

    * Display the list of all records of the
      table
    */
    /*
    * $searchExample must be defined into modules
      /beforeSession.inc.php :
    * include_once 'modules/classes/searchParam.
      class.php';
    * and into modules/common.inc.php :
    * if (!isset($_SESSION["searchExample"])) {
    * $searchExample = new SearchExample();
    * $_SESSION["searchExample"] = $searchExample
      ;
    * } else {
    * $searchExample = $_SESSION["searchExample
      "];
    * }
    * and, also, into modules/classes/searchParam
      .class.php...
    */
    $searchExample->setParam ( $_REQUEST );
    $dataSearch = $searchExample->getParam ();
    if ($searchExample->isSearch () == 1) {
        $data = $dataClass->getListeSearch (
            $dataExample );
    $vue->set($data , "data");
    $vue->set(1, "isSearch");
    }
    $vue->set($dataSearch , "exampleSearch");
    $vue->set("example/exampleList.tpl","corps" );
    break;
case "display":
    /*
    * Display the detail of the record
    */
    $data = $dataClass->lire($id);
    $vue->set($data,"data");
    /*
    * Assignment du modele d'affichage
    */

```

```
$vue->set( "example/exampleDisplay.tpl", "
    corps");

break;
case "change":
/*
 * open the form to modify the record
 * If is a new record, generate a new record
   with default value :
 * $_REQUEST["idParent"] contains the
   identifiant of the parent record
 */
dataRead($dataClass, $id, "example/
    exampleChange.tpl", $_REQUEST["idParent"]);
break;
case "write":
/*
 * write record in database
 */
$id = dataWrite($dataClass, $_REQUEST);
if ($id > 0) {
    $_REQUEST[$keyName] = $id;
}
break;
case "delete":
/*
 * delete record
 */
dataDelete($dataClass, $id);
break;
}
```

Quelques explications...

Le code commence par charger la classe héritée d'ObjetBDD, puis celle-ci est instanciée avec, en paramètres, la connexion PDO à utiliser et le tableau *ObjetBDDParam*, qui contient la configuration par défaut d'ObjetBDD (cf. 5.7 *Le tableau de paramètres ObjetBDDParam*, page 37).

Le nom de la clé associée est indiqué, ce qui permet d'obtenir une code plus générique.

Ensuite, le tableau contenant la description de l'action (*t_param*) est analysé, et plus particulièrement sa valeur *param*, qui correspond à ce qu'on attend du module, avec une instruction *switch*.

Quelques précisions sur ce qui est implémenté par défaut.

list

Permet d'afficher une boîte de recherche et la liste des dossiers associés.

La classe de gestion des critères de recherche

Les paramètres de recherche peuvent être stockés en variable de session, pour que l'utilisateur récupère la liste des dossiers affichés précédemment quand il revient sur cet écran.

Chaque jeu de paramètres est déclaré dans une instance héritée de la classe *SearchParam*, dans le fichier *modules/classes/search.class.php*.

Voici un exemple d'instanciation de cette classe de recherche :

```
class SearchExample extends SearchParam {
    function __construct() {
        $this->param = array (
            "comment" => "",
            "numero" => 0,
            "numero1" => "",
            "dateExample" => date ( 'd/m/Y' )
        );
        $this->paramNum = array (
            "numero",
            "numero1"
        );
        parent::__construct ();
    }
}
```

Deux tableaux doivent être déclarés. Le premier correspond aux attributs utilisables dans la recherche, et leur valeur par défaut doit être indiquée. Le second indique quels sont les champs qui sont numériques.

Deux fonctions sont utilisés couramment :

- *setParam(\$_REQUEST)* : met à jour les paramètres de recherche ;
- *getParam()* : retourne le tableau avec les paramètres de recherche.

De plus, une fonction utilitaire permet de savoir s'il s'agit de la première fois que la recherche est utilisée. Si le formulaire contient un champ configuré ainsi :


```
<input type="hidden" name="isSearch" value="1">
```

Il est alors facile de savoir si le formulaire de recherche a déjà ou non été appelé (la variable *isSearch* est initialisée à 0).

Affichage de la liste

L’affichage de la liste va être traité par la vue Smarty. Le contenu va être transmis (par convention, dans une variable nommée *data*, mais dans certains cas, il vaut mieux utiliser d’autres libellés, surtout si plusieurs informations doivent être affichées en parallèle).

Il est important également de transmettre le nom du *template* qui devra être utilisé, en assignant la valeur à la variable normalisée *corps*.

display

L’entrée *display* permet d’afficher le détail des informations relatives à un enregistrement. Le fonctionnement est beaucoup plus simple que pour l’affichage de la liste : il suffit de transmettre à la vue le contenu de l’enregistrement lu à partir d’ObjetBDD.

Toutefois, si la table contient de nombreuses tables filles, il faudra adapter le code pour transmettre également à la vue l’ensemble des listes d’informations associées.

change

Cette entrée va déclencher l’affichage du formulaire de modification. Par défaut, si la clé vaut 0, on considère qu’il s’agit d’une création.

Pour simplifier l’écriture, une fonction générique est utilisée :

```
dataRead($dataClass, $id, "example/exampleChange.tpl",  
        $_REQUEST["idParent"]);
```

Cette fonction va réaliser automatiquement la lecture de l’information dans la classe ObjetBDD. Si la clé vaut 0, les valeurs par défaut seront récupérées. S’il s’agit d’une table fille, la valeur de la clé parente sera également ajoutée, si la classe a été décrite en ce sens (*cf.* 5.2 *Surcharge*, page 33).

Le troisième paramètre correspond au nom du *template* Smarty à utiliser, qui sera assigné automatiquement.

write

Cette entrée est celle qui est utilisée pour traiter les écritures en base de données. Elle utilise également une fonction générique :

```
$id = dataWrite($dataClass , $_REQUEST);
```

Cette fonction gère l'écriture dans la base de données, traite les erreurs, et retourne l'identifiant, qui doit être supérieur à 0 si l'opération a abouti.

Une fois l'écriture réalisée, la valeur de l'identifiant est mise à jour, pour que le module appelé après dispose de la bonne valeur (affichage du détail après une création, par exemple).

delete

Cette dernière opération, qui traite la suppression d'un enregistrement, utilise également une fonction générique :

```
dataDelete($dataClass , $id);
```

Cette fonction gère également les erreurs et les enchaînements en fonction du résultat de l'opération.

Attention : le framework n'a pas été conçu pour gérer les suppressions en cascade. Il faut donc soit coder les effacements dans les tables filles à partir d'une surcharge de la fonction *supprimer()* d'ObjetBDD, soit configurer la base de données pour qu'elle réalise l'opération elle-même.

Troisième partie

Les vues

Chapitre 7

Les vues

D'une manière générale, toute action demandée se termine par l'exécution d'une vue : envoi d'une page HTML – le cas le plus fréquent –, envoi d'un fichier au format JSON pour les requêtes de lecture AJAX, envoi de fichiers dans des formats variés : fichiers PDF, CSV, des images...

Chaque type d'envoi nécessite une vue différente. Les actions demandées (les modules appelés) décrivent quelle vue doit être utilisée (*cf.* 3.1 *Liste des attributs utilisables pour décrire une action*, page 20). Toutefois, certains modules ne sont pas associés à des vues : ce sont ceux qui vont écrire des informations dans la base de données, et qui enchaîneront systématiquement sur un autre module qui, lui, déclenchera un affichage.

Les vues sont toutes héritées d'une classe de base, **Vue**, qui ne devrait pas être instanciée. Cette classe contient les fonctions génériques suivantes :

fonction	Objectif
set(\$value, \$variable = "")	stocke une valeur dans la vue. Le nom de la variable n'est fourni que pour certains types de vues. Si une seule valeur est stockée sans indiquer de nom, elle peut être utilisée telle qu'elle
send(\$param = "")	déclenche l'envoi du contenu. Elle doit être systématiquement réécrite (vide par défaut).
encodehtml(\$data)	encode la variable fournie avant un envoi vers le navigateur. C'est une fonction récursive capable de traiter les tableaux imbriqués

TABLE 7.1: Fonctions déclarées dans la classe non instanciable Vue

La vue Smarty

Il s'agit de la vue la plus utilisée dans le Framework. Elle permet de générer les pages web. Smarty (<http://smarty.net>) est un moteur de *templates*, dont le principal avantage est qu'il permet une séparation simple du code entre PHP et HTML : cela simplifie l'écriture et la relecture du code.

Les templates de Smarty comprennent le code HTML et le code spécifique qui sera interprété par la classe. Ce code est compris entre accolades, et un fichier PHP intermédiaire est généré automatiquement. Pour plus d'informations sur l'utilisation de Smarty, vous pouvez consulter le site du projet, mais également les quelques informations regroupées dans le chapitre 10 *Compléments sur Smarty*, page 57.

Fonctions disponibles

fonction	Objectif
__construct(\$param, \$var)	Constructeur de la classe. Il nécessite deux tableaux : <i>param</i> : contient l'ensemble des paramètres nécessaires au bon fonctionnement de Smarty (cf. 2.3 Paramètres, page 12) <i>var</i> : variables pré-assignées à Smarty. Ce sont principalement le nom des sous-templates utilisés pour le menu, le pied de page...
set(\$contenu, \$variable)	assigne une valeur à Smarty. <i>contenu</i> peut être tout type de contenu, comme un tableau associatif
send()	Déclenche l'affichage

TABLE 7.2: Fonctions déclarées dans la classe VueSmarty

Organisation de l'écran

Smarty dispose d'une fonction très intéressante, qui permet d'inclure des sous-templates dans un template. Cela permet d'afficher systématiquement le même template, avec simplement certaines parties qui évoluent selon les besoins.

Voici comment est structuré le framework :

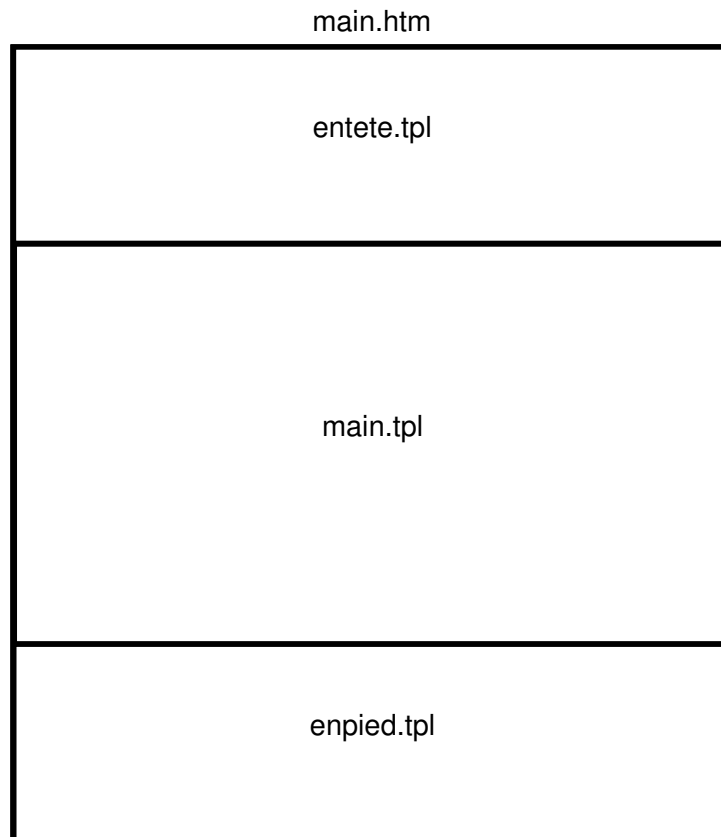


FIGURE 7.1 –

En principe, seul le modèle *main.tpl* change de page en page. Il est modifié en utilisant l'assignation :

```
$set ->("nom_du_modele.tpl", "corps");
```

Un autre modèle est incorporé systématiquement : *main_js.tpl*. Il contient toutes les assignations de feuilles de style, de classes ou fonctions Javascript complémentaires utilisées dans l'application.

Conventions de nommage

Les templates sont stockés dans le dossier `display/templates`. Le dossier `display/templates_c` est utilisé par Smarty pour préparer l'affichage, et doit donc être accessible en écriture au serveur web.

Pour simplifier la navigation, les templates doivent être stockés dans un sous-dossier dont le nom doit être identique au nom du sous-dossier contenant les modules PHP. Les fichiers doivent commencer par le nom du module, et se terminer par l'action correspondante, par exemple *poissonList.tpl*, *poissonChange.tpl*,

etc. Cela facilite la recherche des templates en regroupant par ordre alphabétique les modèles qui portent sur le même sujet.

La vue Ajax

Nom de la classe : **VueAjaxJson**.

Elle encode le tableau fourni par la fonction *set()* au format Json, et transmet la chaîne générée au navigateur, après avoir nettoyé le cache.

La vue CSV

Nom de la classe : **VueCsv**.

Fonctions disponibles :

fonction	Objectif
<code>setFilename(\$filename)</code>	indique le nom à utiliser pour générer le fichier
<code>send(\$param = "")</code>	Déclenche l'envoi du tableau vers le navigateur, au format CSV. <i>param</i> peut contenir le nom du fichier souhaité. S'il est vide, le nom du fichier transmis par la fonction précédente est utilisé. Sinon, un nom de fichier, contenant la date, est généré.

TABLE 7.3: Fonctions déclarées dans la classe VueCsv

La fonction *set()* doit être utilisée pour indiquer le tableau à transformer en CSV. La classe va générer automatiquement une ligne d'entête à partir du nom des colonnes de la première ligne.

En l'état actuel, il n'est pas possible de définir des options particulières pour la génération du fichier CSV.

La vue binaire

Nom de la classe : **VueBinaire**.

Cette vue est utilisée pour envoyer des données sous forme binaire au navigateur (images, par exemple). Les données doivent avoir été auparavant générées dans un fichier du serveur web : c'est le contenu du fichier qui est transmis.

Fonctions disponibles :

fonction	Objectif
setParam(array \$param)	transmet un tableau contenant l'ensemble des paramètres à utiliser pour générer le fichier. Les paramètres sont les suivants : <i>filename</i> : nom du fichier tel qu'il apparaîtra dans le navigateur <i>disposition</i> : <i>attachment</i> (fichier joint) ou <i>inline</i> (affichage direct dans le navigateur) <i>tmp_name</i> : nom du fichier dans le serveur <i>content_type</i> : type mime. S'il n'est pas indiqué, le programme essaiera de le déterminer à partir du contenu du fichier
send()	Envoie le fichier au navigateur, en fonction des paramètres indiqués

TABLE 7.4: Fonctions déclarées dans la classe VueBinaire

La vue PDF

Nom de la classe : **VuePdf**.

Il s'agit d'une variante de la vue précédente. Elle accepte non pas le nom d'un fichier, mais la référence correspondant à une fonction *fopen()* ou équivalente. Cette approche est nécessaire si le fichier PDF à envoyer a été stocké dans une base de données ouverte avec PDO.

Fonctions disponibles :

fonction	Objectif
setFileReference(\$ref)	indique la référence du fichier à traiter (résultat de <i>fopen()</i> ou d'une lecture PDO)
setFilename(\$filename)	Nom du fichier tel qu'il sera transmis au navigateur. S'il n'est pas précisé, il sera généré (en cas d'attachement)
setDisposition(\$disp = "attachment")	Indique la manière d'envoyer le fichier au navigateur. Valeurs acceptées : <i>attachment</i> ou <i>inline</i>
send()	Envoie le fichier au navigateur, en fonction des paramètres indiqués

TABLE 7.5: Fonctions déclarées dans la classe VuePdf

La classe peut générer des exceptions en cas de problème.

Chapitre 8

Génération du menu

Pour les pages web, le menu est généré de manière dynamique :

- lors du premier appel à l'application ;
- après toute opération de connexion ou de déconnexion.

Le menu est stocké en variable de session, pour accélérer l'affichage.

Il est structuré sous la forme d'une liste non ordonnée (balises `ul` et `li`), et contient les classes utilisées par bootstrap pour son affichage.

Fichier de description

Le menu est généré à partir du fichier **param/menu.xml**. La branche principale s'appelle `<menu>`. Voici un exemple d'entrée, qui correspond au menu d'administration :

```
<item module="administration" value="4" title="5"
  droits="admin">
  <item module="loginList" droits="admin" title="3"
    value="2"/>
  <item module="appliList" droits="admin" value="
    appliliste" title="applilistetitle"/>
  <item module="aclloginList" droits="admin" value
    ="aclloginliste" title="aclloginlistetitle"/>
  <item module="groupList" droits="admin" value="
    groupliste" title="grouplistetitle"/>
  <item module="phpinfo" droits="admin" value="
    phpinfo" title="phpinfotitle"/>
</item>
```

Les entrées du menu sont déclarées dans des balises **item**. Voici les attributs utilisables :

Attribut	Requis	Signification
module	X	Nom du module à exécuter, tel que décrit dans le fichier actions.xml (cf. 3.1 <i>Liste des attributs utilisables pour décrire une action</i> , page 20)
droits		Droit nécessaire pour afficher l'entrée du menu. Il est possible d'indiquer plusieurs droits, en les séparant par une virgule
loginrequis		Si vaut 1, l'entrée ne sera affichée que si l'utilisateur est connecté
onlynoconnect		Si vaut 1, l'entrée ne sera affichée que si l'utilisateur n'est pas connecté
value	X	nom de la sous-variable du tableau \$LANG["menu"], qui contient le libellé à afficher (cf. 9 <i>Gestion des langues</i> , page 55)
title	X	nom de la sous-variable du tableau \$LANG["menu"], qui contient le libellé à afficher au survol de la souris (attribut HTML <i>title</i>) (cf. 9 <i>Gestion des langues</i> , page 55)

TABLE 8.1: Liste des attributs utilisables pour décrire les entrées du menu

Une entrée *item* peut contenir d'autres entrées *item*, ce qui permet de décrire les menus en cascade. Actuellement, le menu n'a été testé qu'avec 2 niveaux (menu principal horizontal, et menus verticaux associés).

L'ordre d'affichage est celui décrit dans le fichier xml.

Génération en mode développement

Si la variable `APPLI_modeDeveloppement` est positionnée à `true`, le menu est généré à chaque appel.

Chapitre 9

Gestion des langues

Le framework a été conçu pour supporter plusieurs langues européennes. Pour cela, les libellés à afficher peuvent être stockés dans un tableau, `$LANG`, qui sera chargé en fonction de la langue demandée.

Les fichiers contenant les libellés (les entrées du tableau) sont placés dans le dossier *locales*.

La variable `$language`, dans le fichier *param.default.inc.php* (cf. 2.3 Paramètres, page 12) contient le nom de la langue par défaut. Le fichier correspondant au code (fr.php) est systématiquement chargé. Si une autre langue est demandée, le second fichier sera alors lu, et les nouveaux libellés seront traités en remplacement : cela permet de conserver les libellés d'origine s'ils n'ont pas été traduits.

Le tableau `$LANG` est organisé en plusieurs sous-tableaux :

- menu : contient tous les libellés utilisés dans les menus ;
- message : libellés généraux utilisés dans l'ensemble de l'application ;
- login : libellés utilisés dans le module de gestion des droits et des utilisateurs ;
- `ObjetBDDError` : messages correspondants aux anomalies détectées par la classe `ObjetBDD` (cf. 5 *ObjetBDD - accéder aux bases de données*, page 29) ;
- les autres entrées sont libres pour l'application.

Les libellés peuvent inclure des balises HTML, qui seront envoyées telles qu'elles au navigateur. Cela permet d'insérer, par exemple, un retour à la ligne (`
`).

La variable `$LANG` est transmise systématiquement à la vue Smarty.

Formatage des dates

Le fichier de langue commence par la définition du format de date, qui est ensuite transmis à `ObjetBDD` pour que la classe en tienne compte lors du formatage des informations.

Chapitre 10

Compléments sur Smarty

Syntaxe générale

Smarty est basé sur le langage PHP : il est possible d'utiliser beaucoup de mécanismes du langage, notamment les fonctions de test.

Les commandes Smarty sont encadrées dans des accolades. L'analyseur est capable de rechercher des balises dans des chaînes encadrées par des guillemets ou des cottes. Toute commande ou ordre Smarty doit toucher l'accolade ouvrante.

Les variables, comme en PHP, doivent commencer par le caractère *dollar*.

Cohabitation Javascript et Smarty

Les fonctions Javascript sont, elles aussi, encadrées par des accolades. Pour permettre d'insérer des variables Smarty dans du code Javascript, il suffit de respecter les règles suivantes :

- toute commande Smarty doit impérativement toucher l'accolade ouvrante ;
- tout code Javascript doit être séparé de l'accolade ouvrante par un espace.

Exemple :

```
function fonctionJavascript() {  
var varJavascript = {$variableSmarty};  
}
```

La variable *varJavascript* sera assignée avec le contenu de la variable *\$variableSmarty*, transmise par Smarty.

Affichage d'une variable

Une variable assignée à Smarty est affichée ainsi :

Code PHP :

```
$vue->set("contenu", "varSmarty");
```

Dans le template :

```
{ $varSmarty }
```

affichera la valeur *contenu*.

Affichage d'une liste

En général, l'interrogation de la base de données ramène un tableau associatif, chaque ligne contient un tableau avec le nom des attributs comme clé, et la valeur de l'attribut associé.

Smarty dispose d'un mécanisme permettant de traiter un tableau facilement.

Voici d'abord le code PHP :

```
$vue->set($instance_objetBDD->getListe(), "data");
```

Et le code permettant de l'afficher dans le template :

```
{section name=lst loop=$data}
{$data[lst].attr1} {$data[lst].attr2}<br>
{/section}
```

Les attributs *attr1* et *attr2* de toutes les lignes seront affichés les uns au dessous des autres.

Dans un tableau - Datatables

Le framework utilise le composant Datatables (datatables) pour l'affichage des tableaux. Datatables a été paramétré pour trier correctement les dates (plugin qui reconnaît automatiquement les libellés de type date).

Pour qu'une table soit reconnue comme un composant Datatables, il suffit de lui rajouter la classe datatable. Voici un exemple d'utilisation :

```
<table id="exampleList" class="table table-bordered
    table-hover datatable " >
<thead>
<tr>
<th>Date</th>
<th>Comments</th>
<th>status</th>
</tr>
</thead><tbody>
{section name=lst loop=$data}
<tr>
<td>
{if $droits["gestion"] == 1}
```



```
<a href="index.php?module=exampleChange&example_id={
    $data[1st].example_id}">
{$data[1st].example_date}
</a>
{else}
{$data[1st].example_date}
{/if}
</td>
<td>{$data[1st].comment}</td>
<td><span class="textareadisplay">{$data[1st].
    example_comment}</span></td>
</tr>
{/section}
</tbody>
</table>
{/if}
```

Chaque ligne du corps est traité dans la section. Cet exemple rajoute en plus l'accès à la page de modification, selon les droits définis (*cf. 4 Identifier les utilisateurs et gérer les droits*, page 23).

Pour modifier l'ordre de tri ou d'autres paramètres du composant Datatables, le plus simple est de rajouter ce code après l'affichage du tableau :

```
$(document).ready(function() {
    var exempleList = $("#exempleList").DataTable();
    exempleList.order([[1, 'desc'], [0, 'desc']]).
        draw();
});
</script>
```

L'ordre des colonnes commence à 0.

Dans un select

Dans les formulaires, des champs de type *select* est souvent utilisé pour proposer une liste fermée de choix à l'utilisateur.

Voici comment l'implémenter dans un template :

```
<select id="container_type_id" name="
    container_type_id" class="form-control">
<option value="" {if $data.container_type_id == ""}
    selected{/if}>Selectionnez...</option>
{section name=1st loop=$container_type}
```

```
<option value="{ $container_type[1st].
    container_type_id}" {if $container_type[1st].
    container_type_id == $data.container_type_id}selected
{/if}>
{ $container_type[1st].container_type_name}
</option>
{/section}
</select>
```

Dans cet exemple, le champ `container_container_id` doit être sélectionné à partir de la liste contenue dans `container_type`. Ici, l'utilisateur peut ne pas renseigner l'information : la première option peut être vide. Si le champ doit être obligatoire, il suffit de supprimer la première option.

Des tests sont réalisés pour positionner correctement l'indicateur *selected* lors de l'affichage en modification.

Les tests

Les tests sont classiques, sous la forme :

```
{if condition_de_test}

{else}

{/if}
```

Les conditions de test sont celles de PHP, par exemple :

```
{if strlen($variable) > 0}
...
{/if}
```

Les variables internes

Les sections disposent de variables permettant de connaître le nombre d'occurrences d'un tableau, l'occurrence courante... Avec des composants comme Datatables, on ne les utilise guère.

Par contre, il est parfois nécessaire de réaliser quelques calculs : il est possible d'assigner des variables facilement (le code a été simplifié par rapport aux premières versions) :

```
{ $variable = 0}
{ $section name=1st loop=$data}
{ $variable = $variable + $data[1st].montant}
```

```
{/section}  
Montant total : {$variable}
```

Affichage des libellés en fonction de la langue

Comme nous l'avons vu précédemment (*cf. 9 Gestion des langues*, page 55), les libellés sont stockés dans la variable `$LANG`. Il est alors assez facile de les afficher dans le template. Voici un exemple correspondant à l'affichage du libellé *Rechercher* :

```
<button name="{ $LANG[message].21}">
```

Organisation des formulaires de saisie

En raison du parti-pris du framework de séparer toutes les actions et de pouvoir leur imposer des droits différents, la gestion d'un formulaire de saisie est un peu compliquée. À partir du même formulaire, il faut pouvoir aussi bien déclencher l'écriture d'une information (*write*) que supprimer l'enregistrement (*delete*).

Il est possible de gérer deux formulaires différents, l'un étant dédié uniquement au bouton *Supprimer*. L'approche actuelle est plutôt d'utiliser du javascript pour générer automatiquement l'action correspondante.

Voici un exemple de formulaire implémentant ce fonctionnement :

```
<form class="form-horizontal protoform" id="
  exampleForm" method="post" action="index.php">
<input type="hidden" name="example_id" value="{ $data
  .example_id}">
<input type="hidden" name="moduleBase" value="
  example">
<input type="hidden" name="action" value="Write">

<div class="form-group center">
  <button type="submit" class="btn btn-primary
    button-valid">{$LANG["message"].19}</button
  >
  {if $data.example_id > 0 }
  <button class="btn btn-danger button-delete">{
    $LANG["message"].20}</button>
  {/if}
</div>
```

`</form>`

Le formulaire doit comprendre la classe *protoform*, et deux champs cachés : *moduleBase* et *action*. Les boutons doivent être des classes *button-valid* ou *button-delete*, pour respectivement déclencher l'écriture ou la suppression de l'enregistrement.

Le code Javascript associé (déjà déclaré dans le framework) va permettre de modifier le contenu du champ *action* si le bouton *Supprimer* est actionné. Le contrôleur reconstituera le module demandé en associant les deux champs.

Quatrième partie

Sécurité et implémentation

Chapitre 11

Mécanismes de sécurité et mise en production

Intégrer le transcodage des clés

Chapitre 12

Mise en production

Configuration et installation générale

Configuration du serveur web

Nettoyage de l'application et contrôles à réaliser

Installation de la base de données des droits

Nettoyage des comptes par défaut

Travailler avec plusieurs applications différentes à partir du même code

Dans certains cas, l'application réalisée doit permettre de travailler avec des bases de données différentes selon le contexte, pour éviter de mélanger les informations. La première solution consiste à créer autant de copies que nécessaire du logiciel.

La seconde consiste à n'utiliser qu'un seul code, mais en paramétrant les informations spécifiques à chaque base de données.

Voici le principe général (cf. schéma 12.2) :

Dans le paramétrage de l'alias DNS (en principe, dans **/etc/apache2/sites-available**), l'application pointe vers le dossier **/var/www/appliApp/appli1/bin**. */var/www* correspond à la racine du site web, *appliApp* au dossier racine de l'application, *appli1* au dossier spécifique de l'alias DNS.

Ce dossier *appli1* ne contient que deux fichiers : **param.ini**, qui contient les paramètres spécifiques, et **bin**, qui est un lien symbolique vers le dossier **../bin**.

Le dossier **../bin** (donc, dans */var/www/appliApp*) est lui aussi un alias qui pointe vers le code réel de l'application, ici **code_appli**.

Le fichier **param.inc.php** décrit l'entrée suivante :

```
$paramIniFile = "../param.ini";
```

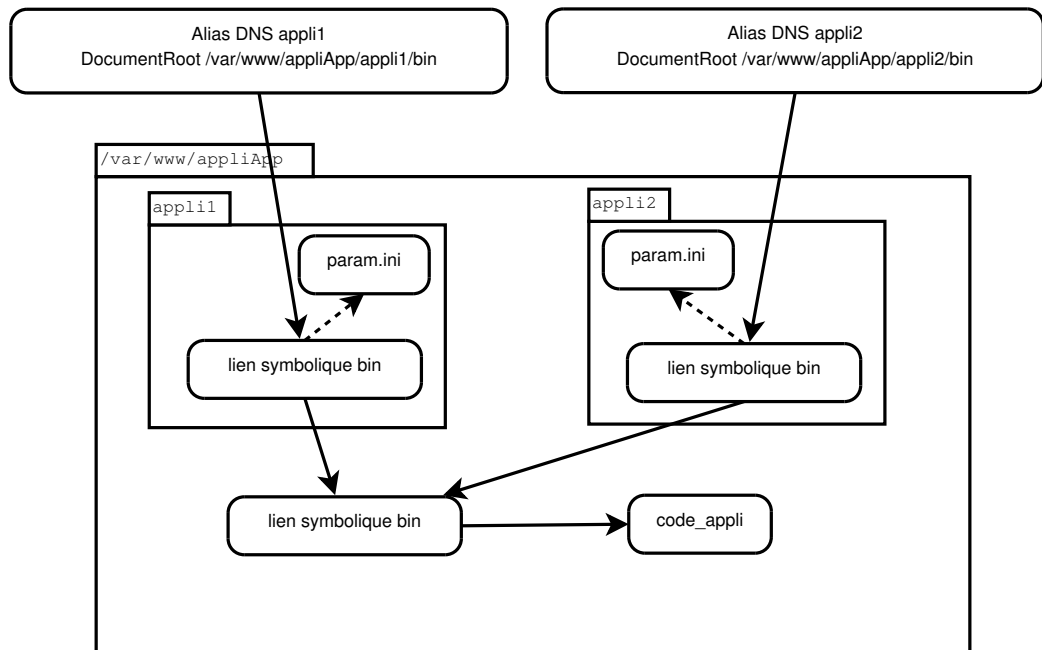


FIGURE 12.1 – Schéma général d’implémentation pour utiliser le même code avec des noms d’application et des jeux de données différents

Le fichier **param.ini** sera cherché dans le dossier parent du code de l’application, c’est à dire soit dans *appli1*, soit dans *appli2* dans cet exemple.

Il suffit qu’il contienne les paramètres adéquats pour rendre l’application utilisable dans des contextes différents à partir du même code initial.