Document version 0.8 – 18.01.2015
Plateforme EVE 2.0.1

Version temporaire destiné à la team de développement
&
Ressources externes travaillant sur le développement

Version définitive suivra

Préambule

Eve framework est distribué de façon libre sous licence GNU *(https://www.gnu.org/licenses/gpl.html)*. Le code source peut être téléchargé gratuitement sur eve-framework.com.

Vous désirez contribuer au développement de la framework ? Devenez membre de la team de développement « source one ». Toutes les informations d'inscriptions sur le site internet de eve-framework.com.

L'utilisation de la solution EVE framework, ses différentes ressources, les comptes gratuits ou tout autre type de fonction dérivé de EVE framework pour des activités illégales selon le droit Suisse (CH) ou contraire à l'éthique est strictement interdite. Des poursuites peuvent être entamées.

Table des matières

Introduction

Après avoir testé un certain nombre de CMS, la conclusion a toujours été la même. S'ils sont très efficaces pour la création rapide de petits sites web utilisant des modules existants, leur usage est très vite limité pour de gros modules dédiés à une tâche précise et qui ne peut pas réellement être généralisée. L'apprentissage pour la création de module est vite fastidieuse et impose un certain nombre de contraintes, un grand nombre de ressources sont mises en place sans réellement en avoir l'utilité, etc. De plus, après un certain nombre de recherches, il a été démontré que la sécurité n'était pas optimale sur ce genre de système. Ils sont prévus pour un grand nombre de plates-formes différentes, et dès lors ne sont pas spécialisés. Les failles sont vites comblées, mais impactent directement un grand nombre de systèmes différents. Elles sont donc très souvent exploitées, cela d'autant plus que les administrateurs peinent généralement à mettre à jour leur système.

Pour ces différentes raisons il a été décidé de créer notre propre système, sous la forme d'un Framework dans un premier temps, mais destiné à devenir un véritable CMS dans un second temps.

Développement

Ce système a été développé dans un premier temps pour mettre en place la plate-forme de commande de Parahoster. Rapidement, nous nous sommes rendu compte du potentiel de ce système. L'équipe de développement a donc proposé à la direction d'améliorer ce système pour en faire une plate-forme indépendante permettant aussi bien de créer plus facilement de nouvelles applications que d'offrir de nouvelles fonctionnalités à Parahoster. Après une discussion animée, la direction a accepté cette proposition. La décision a donc été prise de rendre cette plate-forme autonome et de plus en plus performante. Elle a ainsi pu être implémentée avec succès entre autre pour la plate-forme de commande de papeterie du groupe Optic2000 suisse. En parallèle, la plate-forme a continué d'être optimisée pour permettre un usage facilité de ces composants et de ses modules.

Les objectifs de cette plate-forme sont les suivants. Il va être nécessaire de créer les modules de contrôle afin d'en faire un réel CMS et permettre une gestion automatique des autres modules. Ce module de contrôle va permettre un tournant majeur du projet. En effet si ce système est propriétaire, l'objectif est d'en faire un système open-source aussi rapidement que possible. Ce module de contrôle permettra réellement de passer ce cap et de permettre aux contributeurs de prendre en main le système.

Remarque

Le Framework EVE est développé sur PHP 5.4 ou supérieur. Il utilise un certain nombre de fonctions qui sont propres à cette version. En particulier, il s'agit de programmation orientée objet et faisant appel aux namespace pour retrouver l'architecture du site. Ensuite le système utilise un certain nombre de Design Pattern pour fonctionner. Cet usage permet au système d'augmenter sa polyvalence et de réduire sa dépendance aux technologies.

Un avantage d'EVE est sa portabilité. Pour autant que certain prérequis soient suivis (version minimale de PHP en particulier) le Framework peut être placé sur n'importe quel système. Un effort a particulièrement été fait pour rendre le système indépendant de l'accès aux données. Pour autant que les requêtes spécifiques soient redéfinies, le système peut changer son comportement d'accès. Nativement l'équipe de développement a choisis un accès via PDO.

Fonctionnement

Model MCV

L'application fonctionne selon un système MCV. C'est à dire que les trois parties (model, contrôleur et vue) sont séparées les uns des autres. Le model, appelé Manager dans Eve, s'occupe de récupérer les données sur le serveur pour les transformer en entités. Le contrôleur manipule ces entités et les données de l'utilisateur pour traiter la demande de l'utilisateur. Il va ensuite créer de nouvelles entités pour le serveur et/ou envoyer des informations à la vue. La vue, également appelée action, quant à elle va interpréter les données transmises depuis le contrôleur pour générer le contenu que verra l'utilisateur.

Classes

Chargement

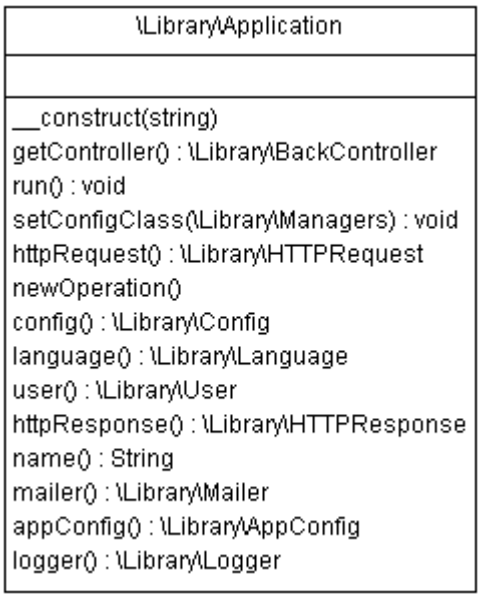
Le chargement des classes peut se faire soit de manière standard en incluant le fichier de la classe, soit de manière automatique avec l'autoloader mis à disposition. Cet autoloader charge les classes en convertissant le namespace en arboressance de fichier. C'est à dire que le namespace doit correspondre exactement au nom des fichiers depuis la racine du système. De plus, le nom du fichier doit être le nom de la class suivi de [.class.php].

Il y a une exception à cette règle. Dans la mesure où le les classes contenues dans Utils sont susceptibles d'être souvent chargées, leur instanciation est facilitée. Un contrôle supplémentaire est fait en indiquant le dossier Utils au début de l'arborescence du fichier.

Les classes sont séparées en deux grandes catégories. D'un côté les classes qui peuvent être crées ou modifiées par un utilisateur du système. De l'autre des classes qui ne servent qu'à transmettre de l'information.

Classes modifiables

Application



Définition

Il s'agit du cœur du système. C'est l'application qui est appelée en premier sur la page root de l'application. C'est également l'application qui doit créer le contrôleur ainsi que les éléments de base du système.

La marge de manœuvre du créateur de l'application n'est pas énorme pour l'application. Il doit principalement lui indiquer son nom (qui doit être le même que le dossier dans lequel elle se trouve) et utiliser les méthodes souhaitées. Il peut s'il le souhaite indiquer d'avantage d'informations.

Architecture

Chaque application doit se trouver dans un dossier portant le même nom que celui de l'application. Ce dossier doit lui se trouver dans le dossier /Applications/ qui se trouve à la racine du système.

Le dossier de l'application doit contenir les éléments suivants

- Un dossier nommé Config qui doit contenir un fichier nommé Config.class.php. Ce fichier doit être une class nommée Config. Cette class doit uniquement contenir les constantes de base du système. Il est nécessaire que le fichier de configuration contienne les informations suivantes
    - LOG : Boolean

Indique si oui ou non le système doit enregistrer les logs

- BDD_HOST : String

Le host de la base de données

- BDD_NAME : String

Le nom de la base de données

- BDD_USER : String

Le username de la base de données

- BDD_PASSWORD : String

Le mot de passe de la base de données

- MAX_ADMIN_LVL : int

Le niveau à partir duquel un utilisateur est administrateur

- DAO : String

Le type de DAO utilisé

D'autres informations peuvent être insérées si besoin.

- Un dossier nommé Lang qui lui-même doit contenir des fichier nommés base.[LANG].php, [LANG] devant être remplacés par chacun des codes langues valides pour l'application. Au minimum le fichier avec le code langue de base doit être créé. A l'intérieur de ces fichiers, les constantes de langues globales de l'application peuvent être créées.
- Un dossier nommé Templates contenant un fichier nommé layout.php. Lors de la génération d'une page HTML, tout le contenu dynamique de la page est chargé dans une variable nommée $content et la page layout.php est chargée. Le contenu va donc se placer à l'emplacement prévu.

Un certain nombre d'éléments peuvent également être insérés dynamiquement dans le fichier layout. Parmi lesquels :

- Le chemin pour accéder à la racine du site : $root
- Le chemin pour accéder à la racine du site avec les informations de langue : $rootLang
- Une instance de l'utilisateur : $user
- Une information sur la langue dans : $language

- Un fichier représentant l'application. Sa nomenclature doit être [NomDeLApplication]Application.class.php et il doit contenir une class [NomDeLApplication]Application qui hérite de \Library\Application. Ce fichier doit être à la racine de l'application.

Flow

1. L'application est instanciée.

Le constructeur doit être redéfini par le créateur de l'application afin que celle-ci déclare un nom pour l'application avant d'appeler le constructeur parent de l'application. Elle va générer un certain nombre d'éléments

a. Une instance de AppConfig
b. Une instance de Logger
c. Une instance de HTTPRequest
d. Une instance de HTTPResponse
e. Une instance de DataTyper
f. Une instance de User

2. Appel de la méthode run de l'application

Cette méthode est abstraite et est laissée à l'appréciation du créateur de l'application. Un certain nombre

d'actions sont néanmoins impératifs pour permettre à l'application de fonctionner. C'est-à-dire les actions suivantes :

a. Appel à la méthode interne getController pour charger une instance
b. Appel de la méthode execute du contrôleur qui doit exécuter l'action courante
c. Appel de la method setPage de HTTPResponse pour lui transmettre l'instance de la \Library\Page générée depuis le contrôleur
d. appel de la method send de HTTPResponse

3. Lors de l'appel à la méthode getControler, l'application doit générer un certain nombre d'informations pour finalement retourner le contrôleur souhaité par l'utilisateur en fonction de l'URI indiqué.
   a. Ajout si nécessaire de la page de langue globale de l'application
   b. création de l'URL de langue
   c. Si l'utilisateur souhaite se déconnecter, déconnexion
   d. On définit un routeur et on lui définit ses routes
   e. On regarde si l'une des routes correspond à l'URI
   f. Si la route nécessite une connexion et que l'on est pas connecté, on charge le module de connexion
   g. Si on a droit de charger la route, on la charge
   h. On contrôle que le contrôleur soit bien une instance de \Library\BackController
   i. On instancie le module de configuration
   j. On retourne le contrôleur

Si une erreur est retournée, un contrôle de l'erreur est fait et l'erreur est traitée.

ApplicationComponent

Définition

Les applications component sont tous les compléments de l'application. Leur seule caractéristique est de bénéficier d'une instance de l'application qui les as créées. De cette manière, ils peuvent toujours accéder aux méthodes de l'application.

Contrôleur

Définition

Le contrôleur d'une classe est la partie qui va traiter les données et les entrées de l'utilisateur pour créer de nouvelles données et envoyer des informations visuelles à la vue.

La majeure partie du contrôleur est à écrire par le créateur de l'application. C'est dans cette partie qu'il faut définir l'ensemble des actions que doit résoudre l'application étant donné l'action donnée et les informations transmises par l'utilisateur.

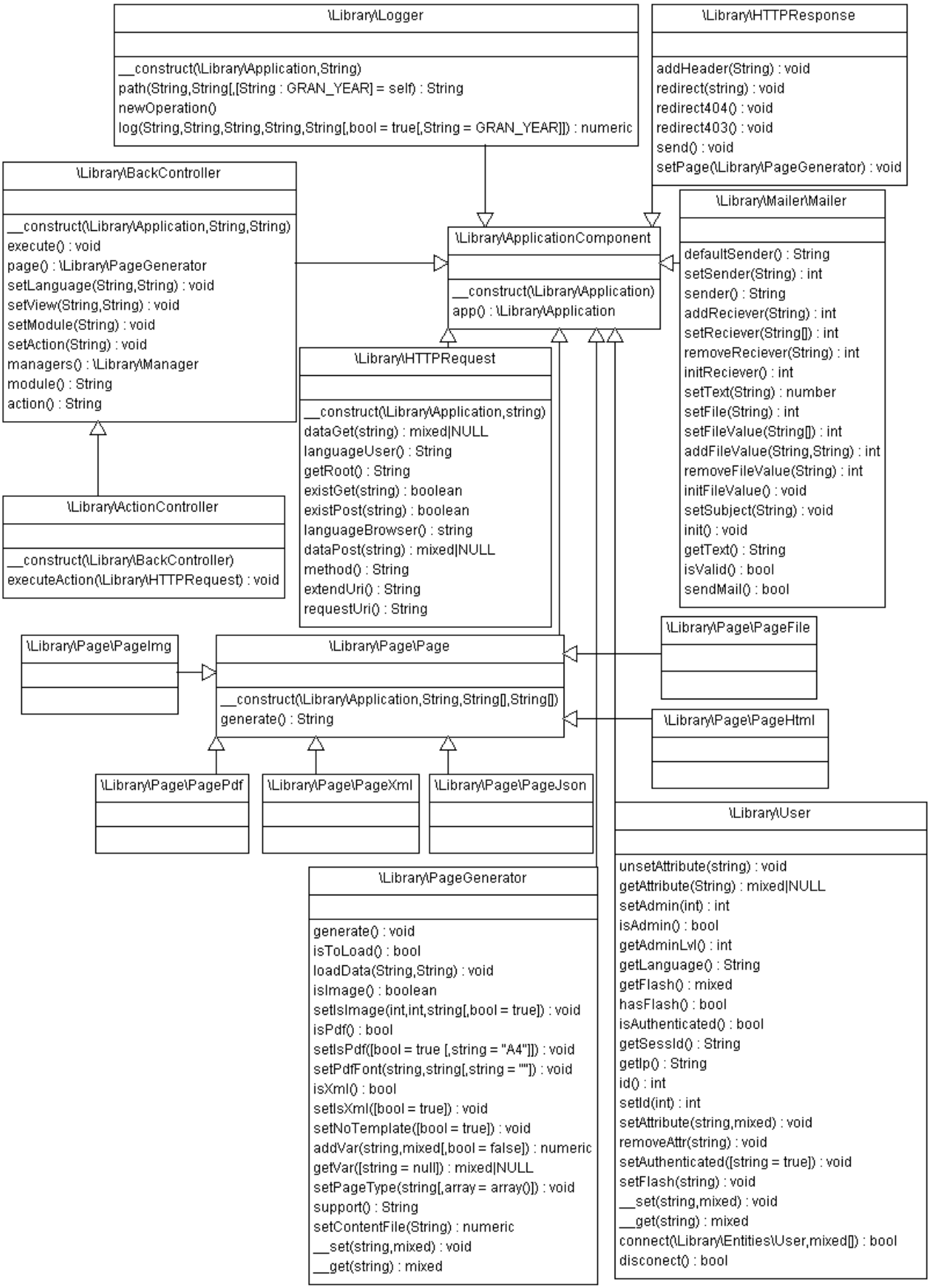
Si les actions sont assez courtes, il est conseillé d'utiliser la méthode dédiée. Dans le cas contraire, la classe dédiée permet d'avoir une meilleure structure de code. Dans le cas où les deux options sont proposées, le code va instancier la classe. Attention toutefois au fait que la classe dédiée de l'action n'hérite pas de la classe dédiée du module. Les deux classes ne peuvent pas utiliser les méthodes l'une de l'autre.

Architecture

Deux cas de figure sont possibles.

Contrôleur du module

Un fichier nommé [ModuleName]Controller.class.php doit être placé à la racine du module. Ce fichier doit contenir une classe nommée [ModuleName]Controller et doit hériter de \Library\BackController. Cette classe doit contenir des méthodes nommées execute[Action] et avoir en paramètre une instance de \Library\HTTPRequest. Ces méthodes sont les méthodes dédiées aux actions.



Contrôleur de l'action

Un fichier nommé [NomDeLAction]Controller.class.pgp doit être placé dans un dossier nommé Controller. Ce dossier doit lui être placé à la racine du module. Ce fichier doit contenir une classe nommée [NomDeLAction]Controller qui hérite de \Library\ActionController. Cette class doit alors implémenter une

méthode nommée executeAction qui contient en paramètre une instance de \Library\HTTPRequest. La classe hérite elle aussi de \Library\BackController, ce qui lui donne accès à toutes les informations de base du contrôleur.

Flow

1. Lors de l'instanciation du contrôleur ses propriétés de base sont initialisées
    a. Instanciation de la page
    b. Instanciation du manager
    c. Mise en place du module
    d. Mise en place de l'action
    e. Mise en place des fichiers de langue
        I. Chargement du fichier de langue général du module s'il existe
        II. Chargement du fichier de langue de l'action s'il existe
    f. mise en place de la vue
        I. Définition du fichier de contenu de la vue
2. Lors de l'exécution du contrôleur
    a. Si un contrôleur spécifique a été défini pour l'action courante, ce contrôleur est instancié et la méthode d'exécution est appelée.
    b. Dans le cas contraire la méthode spécifique de l'action est appelée.

Page (Vue)

Définition

Les vues sont l'ensemble de la partie traitée par le système et qui sera ensuite renvoyée à l'utilisateur. La vue ne doit traiter aucune donnée. Les données doivent toujours avoir été récupérées, traitées et transmises à la vue.

Elles sont partagées en deux parties. D'un côté les pages, et de l'autre côté leur contenu (vue).

Architecture

Les pages sont mises en place dans la librairie et possèdent une fabrique. La fabrique va prendre le type de page en paramètre, et retourner une page permettant de générer une page d'un type bien précis. Cette page va ensuite se charger de traiter les différents paramètres qui lui permettront de s'afficher pour le client.

Les vues doivent toujours être placées dans le dossier Views du module. De plus, chaque vue doit avoir le même nom que le nom de l'action à laquelle elle est associée. La vue n'a pas de lien direct avec l'application, ce qui veut également dire quelle n'a pas accès à celle-ci, ni aux managers.

Dans le cas d'une image ou d'un pdf, une précision peut être donnée pour indiquer une vue différente dans certains cas pour la page HTML ou les autres. Il suffit de nommer la vue [action].img.php ([action].pdf.php) pour que la page soit prise en priorité comme une image (respectivement un PDF). S'il n'y a pas de priorité, la même page sera utilisée que pour une page. HTML.

Les différents types de vue doivent respecter des fonctionnalités

- Les pages HTML peuvent être inscrites dans le template du site
- Les pages PDF sont mises en place grâce à la classe HTML2PDF
- Les images sont mises en place avec la méthode imagecreatetruecolor. Une instance de cette classe est mise à disposition dans une variable nommée $image
- Les pages File retournent simplement un fichier. Ce fichier doit impérativement se trouver dans un document spécifique pour le téléchargement.

HTTPRequest

Cette classe permet de retourner toutes les informations retournées par le client. Elle ne stocke pas d'information et ne permet pas de traiter d'information.

Cette classe est instanciée au lancement du système.

HTTPResponse

Cette classe contient les informations que le système veut renvoyer au client. Elle contient les différentes méthodes qui permettent de terminer le traitement pour renvoyer de l'information.

Cette classe est instanciée au lancement du système.

User

Cette classe contient toutes les informations sur l'utilisateur qui sont persistantes. C'est-à-dire les informations que le système doit conserver de page en page. Si pour l'instant le système fonctionne avec des SESSION, seule la classe User les utilises. Il n'est pas inenvisageable d'utiliser un autre système à l'avenir si bien qu'il est déconseillé d'utiliser des SESSION ailleurs.

Cette class est instanciée au lancement du système.

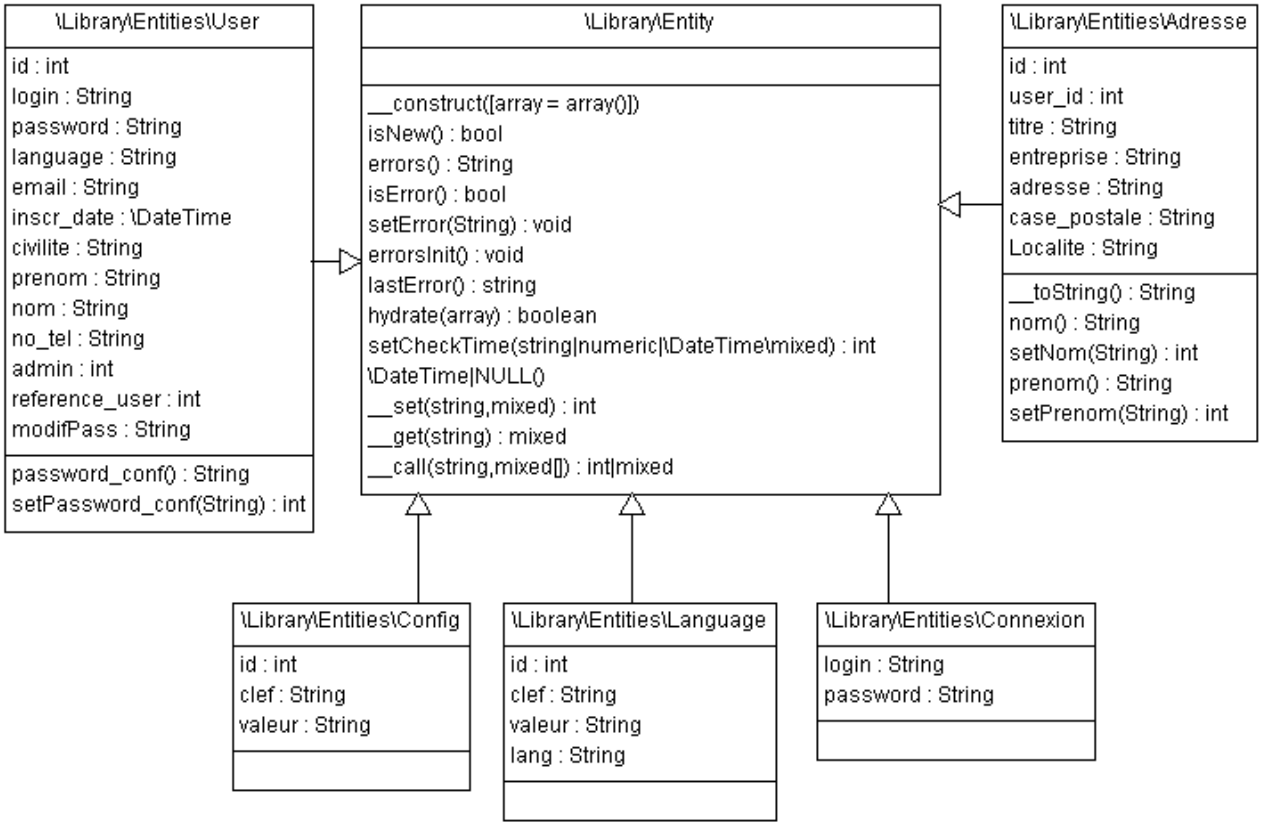
Mailer

Classe permettant d'envoyer des emails.

Cette classe va récupérer les différentes informations relatives à un email pour ensuite l'envoyer. Le contenu de l'email peut être textuel ou peut être un fichier texte (les balises HTML sont interprétées). S'il s'agit d'un fichier texte, des constantes peuvent être présentes, qui seront alors remplacées par leurs valeurs. Il doit n' y avoir qu'une personne qui envoie le message, mais un nombre indéterminé de personnes qui le reçoivent. De plus, un contrôle est fait avant de valider l'envoi pour savoir s'il peut être envoyé.

N'étant pas utile dans tous les cas, cette class n'est instanciée que la première fois où elle est appelée. La même instance est ensuite retournée. Il peut donc être nécessaire de réinitialiser le Mailer entre deux appels.

Entité



Définition

Une entité est la représentation dans le système d'une entrée stockée sur le serveur. Les différents paramètres de l'entité doivent correspondre aux informations sur le serveur.

Architecture

Les entités doivent être placées dans le dossier Entities du module dans laquelle elle doit servir.

Une entité doit impérativement hériter de \Library\Entity afin de bénéficier de toutes ses méthodes et de pouvoir être utilisée dans les Manager

Chacun des attributs de l'entité doit avoir exactement le même nom que le nom des attributs retournés par la class DataTyper. Si un des attributs n'a pas exactement le même nom ou si un des attributs est en trop/manque cela peut provoquer des erreurs lors de l'appel des geter ou des seter.

Dans le cas de PDO, une nomenclature spécifique a été mise en place afin de faciliter l'accès à la base de données. Le nom de chaque table doit être [nom du module]_[nom de l'entité] de manière à pouvoir automatiser certaines tâches. Le nom de l'entité doit donc correspondre à la seconde partie du nom de la table auquel elle doit correspondre.

Remarque

Certaines entités spéciales, nécessaires au fonctionnement du système ou qui ne sont pas spécifiques à un module, sont placées dans un autre dossier. Il s'agit des entités

- Config : Entité gérant les éléments de configuration du système enregistré sur le serveur
- User : Entité gérant les informations d'un utilisateur
- Adresse : Entité gérant les informations d'un utilisateur
- Language : Entité gérant les informations de langue d'un utilisateur

Ces entités sont placées dans les dossiers Library\Entities. Elles n'ont normalement pas à être

instanciées. Elles doivent généralement être appelées par un manager ou une fonction spécifique.

Flow

Une entité n'est qu'une instance de stockage. Il n'y a donc pas réellement de flow.

Lors de sa création, une entité peut automatiquement définir ses données grâce à la méthode d'hydratation.

L'entité doit également mettre en place un système de gestion d'erreur pour les données. C'est ce système qui va permettre aux formulaires de savoir si une erreur s'est produite pour une donnée.

Des fonctions sont automatiquement mises en place pour chaque entité.

- Une fonction d'hydratation
- Un setter et un getter pour chaque paramètre. Grâce à une classe retournant le type des données, le système sait de quel type de données chaque paramètre est, et peut donc contrôler son type, donner des valeurs par défaut, …

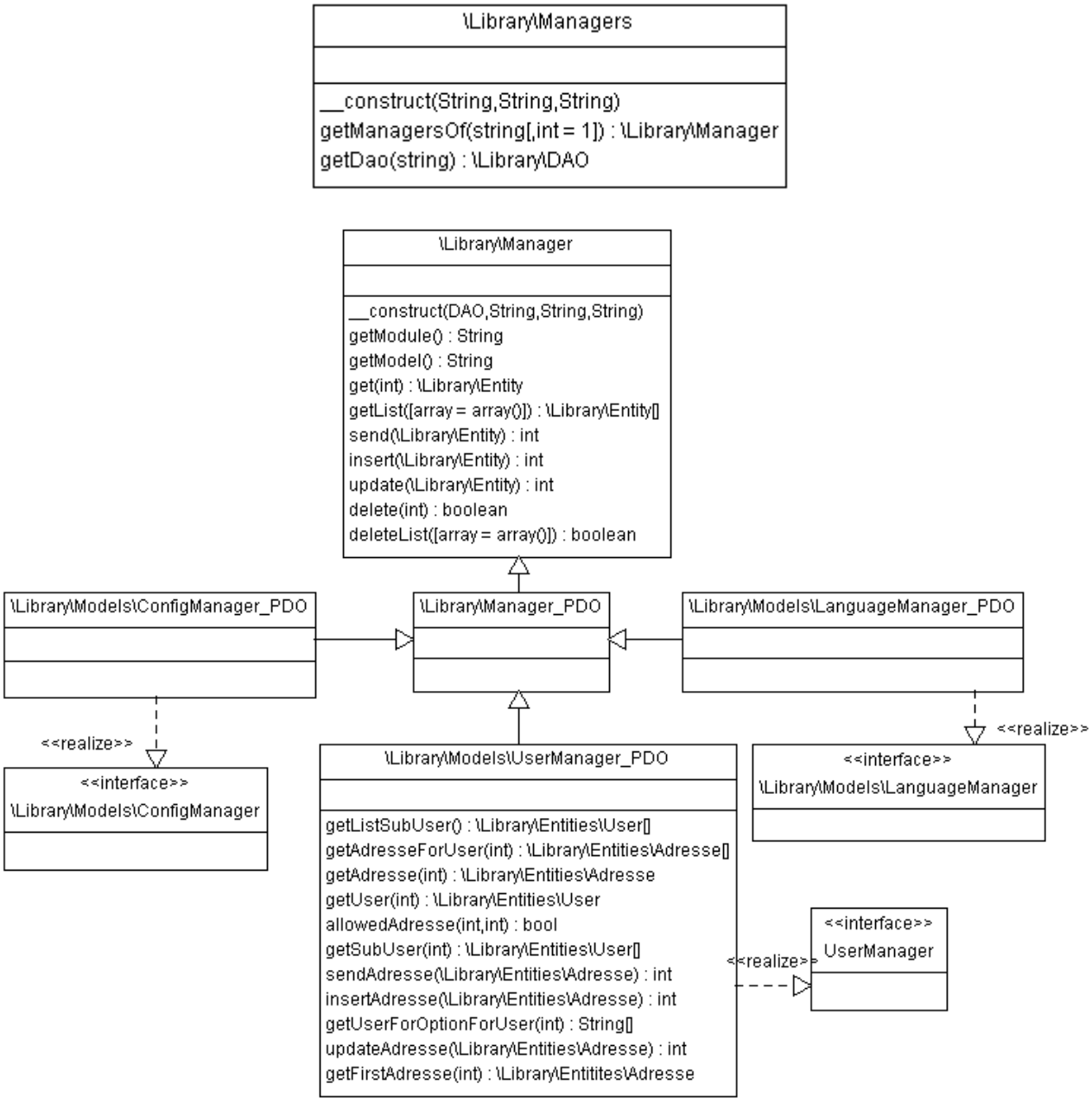
Les fonctions sont nommées set[Attribute]() pour le setter et [attribute]() pour le getter.

Il est important de noter que toutes ces fonctions peuvent toujours être redéfinies afin d'avoir un comportement moins standard ou pour une optimisation.

- Les fonctions __set et __get ont été définies, permettant d'utiliser les différents paramètres de la fonction comme des attributs publics. Ces fonctions font appel à set[Attribute] et [attribute]. Les fonctions redéfinies seront donc appelées si elles ont été créées.

Attention, la fonction magique __call a été utilisée pour définir ces setter et ces getter. Une fonction non définie va donc automatiquement appeler la fonction __call.

Managers



Définition

Afin de rendre les Models indépendants de l'objet d'accès aux données (DAO), ils fonctionnent avec un design pattern factory. Il y a donc d'un côté une usine de Model, nommé Managers, et de l'autre côté les différents models nommés Manager.

Le Managers n'est instancié qu'une seule fois par DAO (il ne s'agit pas d'un singleton et rien n'empêcherait de le faire, mais cela n'est pas utile). Il peut ensuite créer des Manager pour chacune des entités que l'on souhaite. Une méthode statique permet de retourner un DAO en fonction de son type.

Un Manager est une class utilisant un DAO pour récupérer des données sur le serveur pour les transformer en entité utilisable par le système ou d'enregistrer les informations d'une entité sur le serveur. Les Manager doivent hériter de la classe \Library\Manager_[DAO] qui eux-même doivent hériter de \Library\Manager. Un certain nombre de méthodes sont obligatoires.

Les fonctions obligatoires sont

- Une méthode permettant de récupérer une entité d'un élément étant donné son identifiant

- Une méthode permettant de récupérer une liste d'entité étant donné un certain nombre de conditions
- Une méthode permettant d'insérer les informations d'une entité sur le serveur
- Une méthode permettant de mettre à jour une entité sur le serveur étant donné son identifiant
- Une méthode permettant de supprimer une entrée du serveur étant donné son identifiant
- Une méthode permettant de supprimer une liste d'entrée du serveur étant donné un certain nombre de conditions.

Les retours des managers doivent toujours être des entités.

Dans un premier temps, un seul DAO est disponible. Il s'agit de PDO. D'autres systèmes peuvent parfaitement être mis en place, mais il faut alors créer toutes les instances nécessitant un accès aux données. Il s'agit pour les fonctions de base du système de

- Les managers
- Le système d'accès
- Le routeur

Architecture

Les managers doivent être créés dans le dossier Models du module. Un model doit impérativement hériter de \Library\Manager ou éventuellement de \Library\Manager_[DAO] si une classe spécifique pour le DAO a été mise en place.

Chaque Manager doit suivre une certaine nomenclature, c'est à dire [nom de l'entité]Manager_[DAO] et implémenter les différentes fonctions qui n'ont pas encore été définies. Il est également possible de redéfinir les fonctions qui ont besoin d'un traitement spécifique.

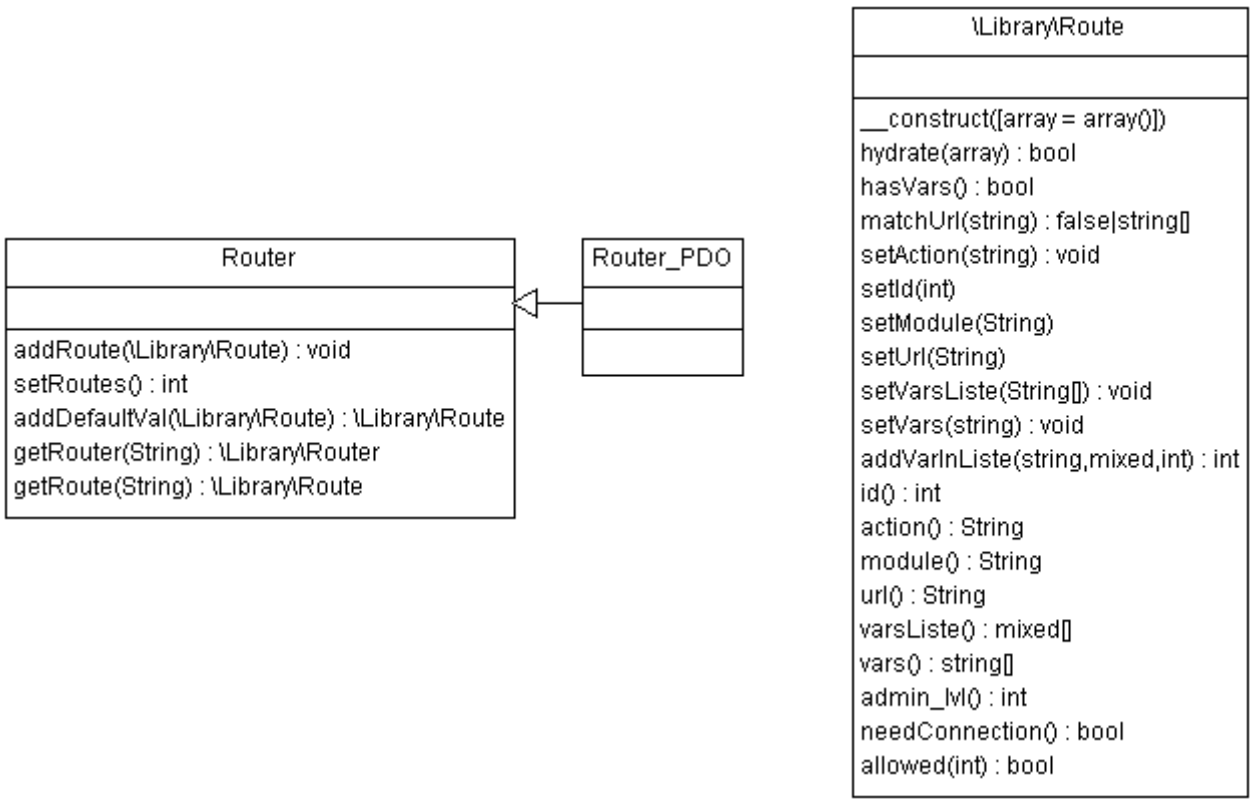
Flow

Le Managers doit être instancié en indiquant le type de DAO souhaité ainsi que le module courant. Lors de cette instanciation, la classe enregistre les informations du DAO souhaité dans une variable statique. Un nouveau Managers doit être créé pour chaque nouveau DAO et pour chaque nouveau module.

Lors de l'appel d'un Manager, le Managers contrôle si un Manager avec ce nom a déjà été instancié. Si c'est le cas il le retourne. Dans le cas contraire, il l'instancie, l'enregistre et finalement le retourne. Cela dans un souci d'optimisation des ressources du serveur.

Avant la génération de la page, les différentes variables qui lui ont été transmises sont mises à disposition de celle-ci. La page a donc accès à toutes les variables qui peuvent lui être nécessaires.

Router



Définition

Le routeur permet de définir, étant donné une URI, l'action et le module qui lui sont associés, ainsi que toutes les variables qui doivent être définies.

Les routes sont enregistrées sur le serveur sous la forme d'un pattern. Lorsqu'un URI est reçu de la part de l'utilisateur, il est comparé avec les différents patterns des routes. Si plusieurs routes ont un pattern qui correspond à cette class, la toute première de la liste qui match sera retournée.

Dans un souci d'optimalité, avant de faire un contrôle de pattern, un premier contrôle est fait pour contrôler simplement l'égalité entre une URI et une route. Si le résultat n'est pas concluant, alors un test de pattern est fait.

Dans un premier temps, le routeur prend le même DAO que le reste de l'application.

Fonctionnalité

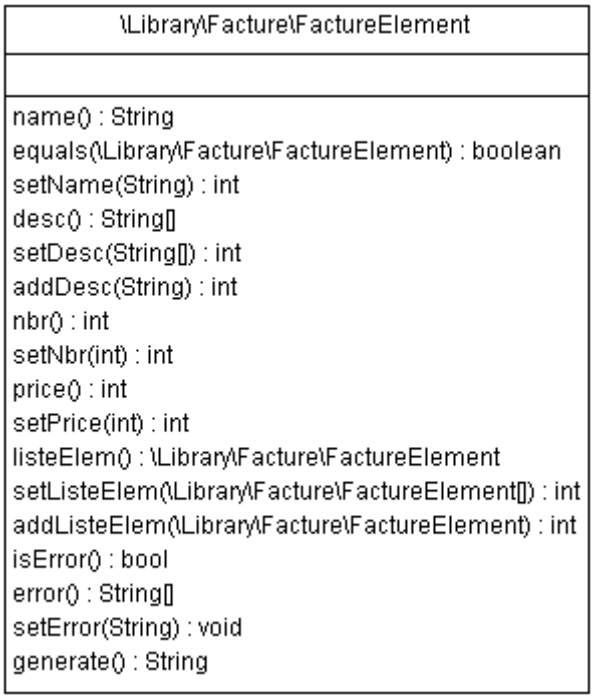
Chaque route doit être enregistrée sur le serveur pour pouvoir être utilisée doit définir chacun des attributs suivants

- Un URI sous la forme d'un pattern
- Le module à charger
- L'action qui doit être lancée
- La liste des variables possibles. Le nombre de variables doit être égal au nombre d'éléments du pattern. Les variables sont séparées par des virgules.
- Le niveau d'administration minimum pour accéder à la page. Le niveau 0 défini une page publique.

Dans le cas de PDO, les routes sont simplement des entrées en base de données, dans la table route.

Divers

Facture



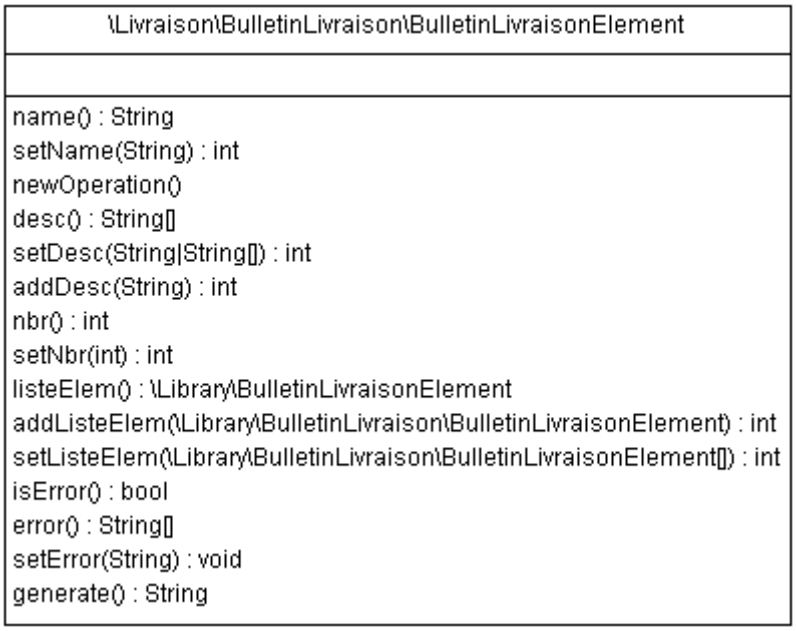
Les factures ayant toujours le même format, une classe spéciale a été mise en place pour les gérer le système de facture fonctionne avec deux éléments.

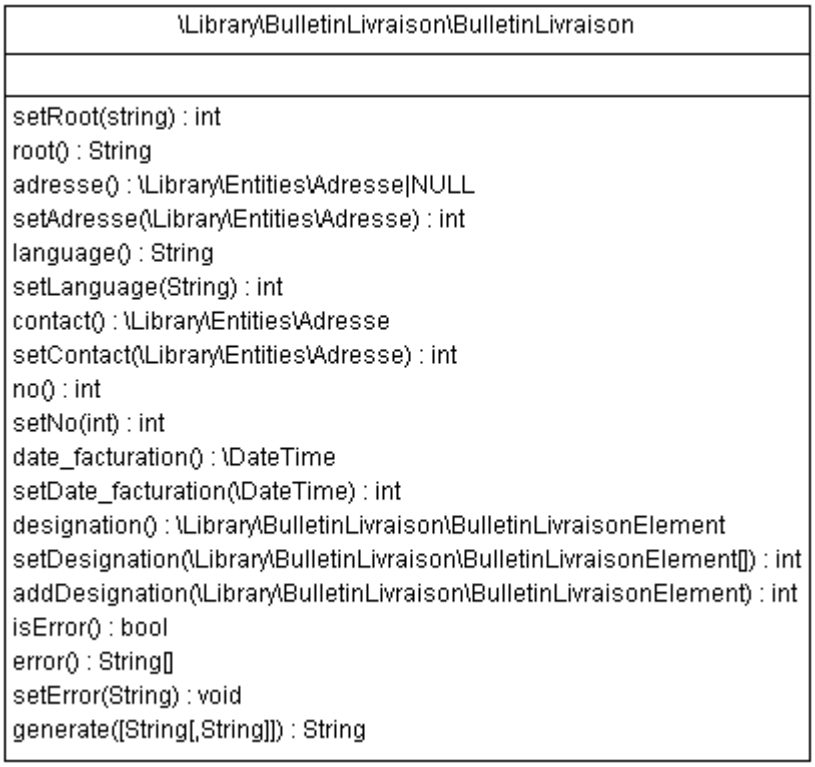
Premièrement, la classe Facture contient toutes les informations globales de la facture ainsi que différentes méthodes permettant d'obtenir les informations complémentaires. La facture contient ensuite une liste d'éléments de la facture.

Puis, un élément de la facture contient l'ensemble des informations relatives à un élément en particulier. Un élément de facture peut lui-même contenir une sous-liste d'éléments de facture. ils seront traités de manière récursive.

Pour changer le format d'une facture (d'un élément de facture) il est envisageable de créer une nouvelle classe héritant de Facture (respectivement de FactureElement) pour modifier la partie generate. Cela assure d'avoir toujours un contrôle sur les éléments de la facture.

Bulletin de livraison





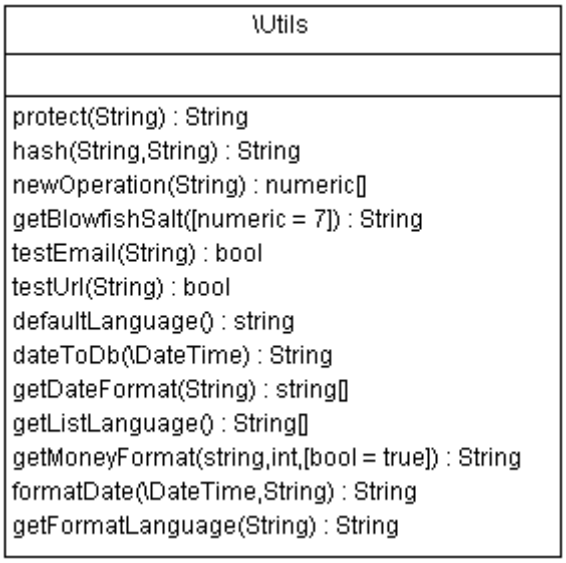
Les bulletins de livraison sont une composante assez importante pour notre société. C'est pourquoi un système a été spécialement mis en place pour les gérer. Ce système fonctionne en deux parties.

Premièrement, une classe Bulletin de Livraison, qui contient l'ensemble des informations globale du bulletin de livraison. Le bulletin contient également une liste des éléments que contient ce bulletin de livraison.

Ensuite, un élément du bulletin de livraison contient l'ensemble des informations relatives à un élément en particulier. Un élément de bulletin de livraison peut lui-même contenir une sous-liste d'éléments de bulletin de livraison. Ils seront traités de manière récursive.

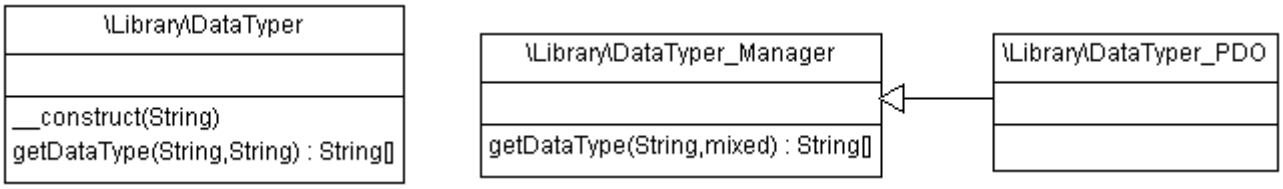
Pour changer le format d'un bulletin de livraison ou d'un élément de bulletin de livraison, il est envisageable de créer une nouvelle classe qui hérite de bulletin de livraison, respectivement d'un élément de bulletin de livraison, en réécrivant la méthode generate.

Utils



Une classe a été créée, regroupant un grand nombre de méthodes utilitaires. Toutes ses méthodes sont statiques, elles peuvent donc être utilisées depuis n'importe où. Ces méthodes ne doivent avoir aucun effet de bord. C'est-à-dire qu'elles ne doivent pas utiliser de ressources systèmes, si ce n'est éventuellement de récupérer des informations de configuration.

Format de données



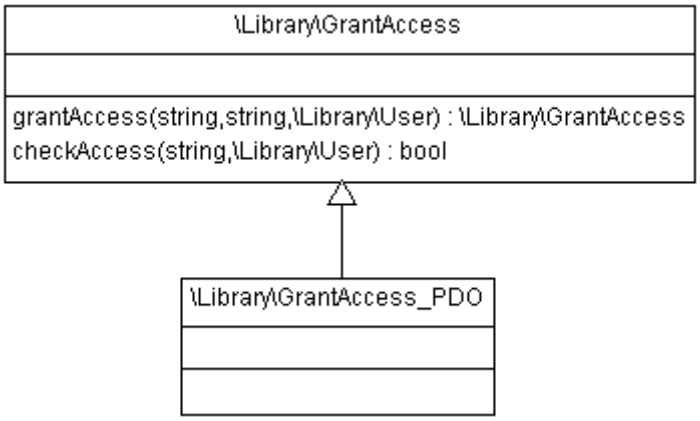
Dans la mesure où PHP est un langage non typé, mais que le type de données est souvent nécessaire, il était nécessaire de trouver un moyen de gérer ce typage de donnée. Une classe a donc été mise en place pour gérer cet aspect des choses. Il s'agit de la classe DataTyper, qui est instanciée depuis une fille de la classe DataTyper_Manager.

Dans un premier tempsa, le choix a été fait de récupérer le type des données directement depuis la base de données puisque les entités ne sont rien d'autre qu'une représentation des entrées de la base de donnée.s Malheureusement ce processus prend du temps. Une autre solution serait de créer un fichier XML simple contenant le type de chaque entité.

Si le choix s'est arrêté sur la base de données dans un premier temps, c'est parce que tout est automatique. Il n'y a rien à régénérer lors de la création d'une nouvelle entité, ou la modification d'une ancienne. De plus, le nombre d'accès au système n'est pas encore un problème. A court terme, il faudra malgré tout créer une nouvelle classe permettant de fonctionner indépendamment de la base de données.

Gestion des accès

Dans la mesure où un certain contrôle des accès est nécessaire pour un site internet, un système de



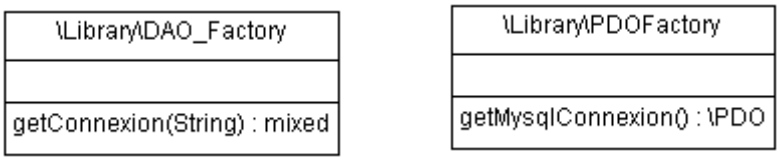
contrôle d'accès à été mis en place.

Un certain nombre d'aspects de sécurité se doivent d'être contrôlés. Cela peut être résumé comme : « Un utilisateur connecté doit avoir entré un login et un mot de passe, il doit avoir changé de page il y a suffisamment peu de temps et son IP ne peut pas avoir changé ». Ce système doit donc protéger des problèmes suivants

- Utilisation frauduleuse d'un identifiant
- Oubli de fermeture de sa session
- Vol d'identifiant de session

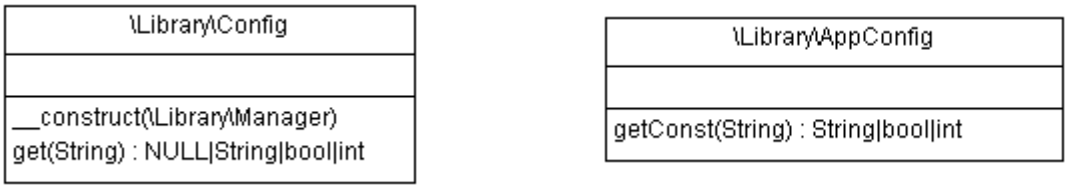
De plus, il faut demander une seconde authentification pour les actions critiques.

DAO



Un constructeur de DAO doit être mis à disposition sous la forme d'une usine. De cette manière, le système n'est pas dépendant d'un DAO spécifique. En remplaçant le DAO souhaité avec l'appel de cette fonction, on remplace le DAO du système.

Configuration

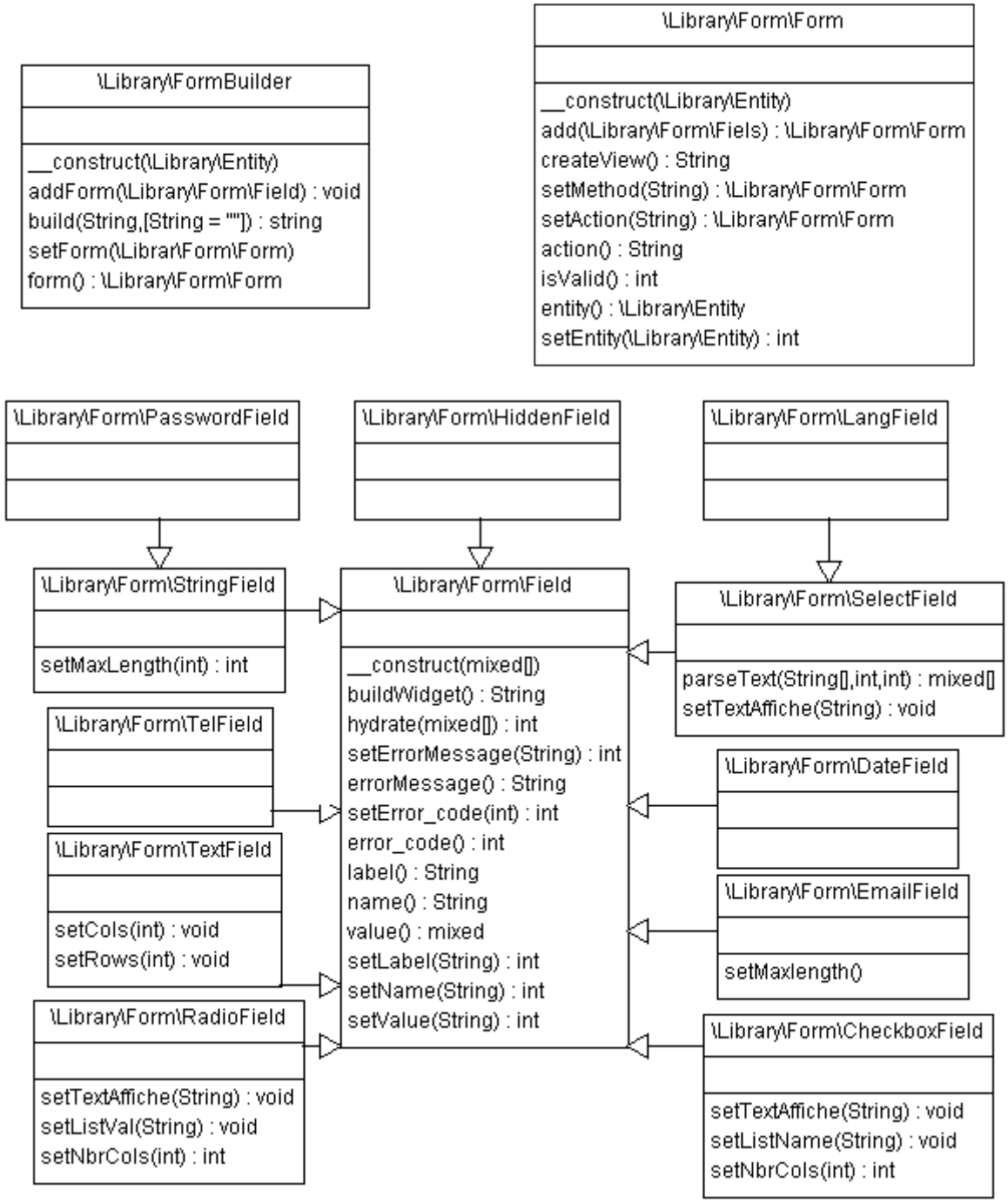


Il existe deux types de fichier de configuration

Premièrement, un fichier de configuration de l'application est placé dans le dossier config du dossier de l'application. Ce fichier contient des informations cruciales pour le fonctionnement de base du système. Sans ces informations, le système ne peut pas démarrer. Ces informations sont stockées en dur sur le serveur sous la forme de simples constantes de classe.

De plus, un deuxième élément de configuration existe qui va aller chercher en base de données des éléments en fonction de leur clé. Ce système est plus simple d'accès, mais nécessite de décider au préalable quel DAO doit être utilisé pour utiliser ces informations. Il nécessite donc le premier pour fonctionner.

Formulaire



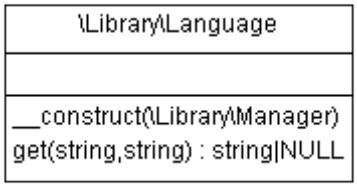
Un système a été mis en place pour permettre la génération automatique ainsi que le contrôle d'éléments du formulaire.

Pour créer un formulaire, il y a deux manières. Soit créer manuellement le formulaire en utilisant les classes Form et Field (ainsi que ses sous-classes). Cela permet de créer un formulaire contenant des champs particuliers, ou des éléments qui doivent être différents pour chaque utilisateur.

L'autre manière consiste à créer un XML et à l'envoyer à FormBuilder. Celui-ci va automatiquement parser le XML et retourner tous les éléments nécessaires. Il va finalement générer le formulaire ainsi que toutes ses erreurs éventuelles en fonction de l'entité source.

Le contrôle des données doit se faire dans les Entités. Le formulaire ne fait aucun contrôle lui-même sur le serveur. Il ne sert qu'à générer un formulaire HTML plus facilement.

Langue

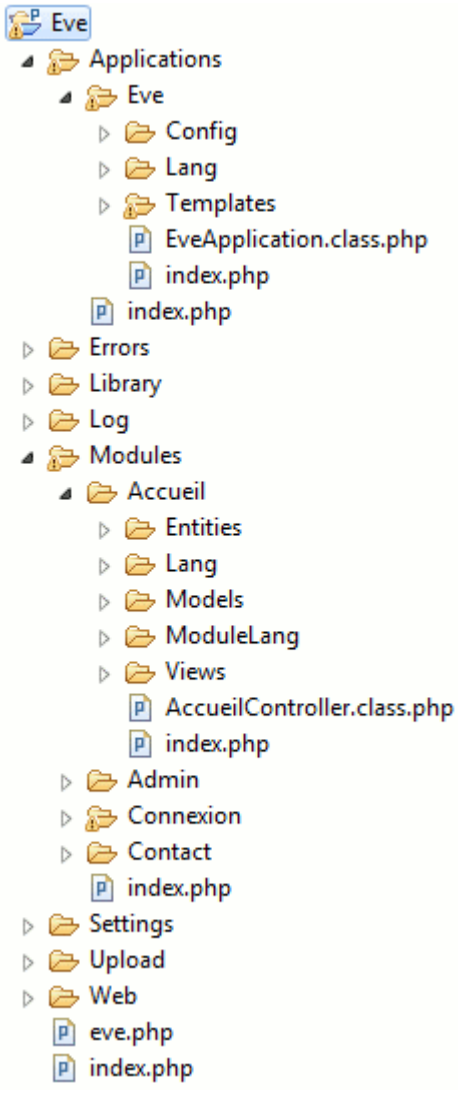


Les éléments de langue sont extrêmement importants mais apportent certaines contraintes. Dans la mesure où tout le texte doit pouvoir être traduit, la solution choisie a été de remplacer tout le texte par des constantes de langues. Cela apporte une plus grande flexibilité (il suffit de créer ces constantes dans une nouvelle langue pour traduire le site) car il n'est pas nécessaire de recréer toutes les pages pour avoir les langues.

La contrainte principale étant alors les éléments dynamiques. Il n'est pas envisageable de devoir faire des modifications sur le serveur pour chaque entrée dynamiquement insérée. De plus, il n'est pas possible d'inscrire dans une table toutes les traductions (il faudrait alors modifier toutes les tables pour ajouter une langue). Une nouvelle table de constante en fonction de la langue a donc été mise en place. Cette table permet donc en inscrivant une constante et une langue de retourner la valeur de la constante

Architecture du module

Lors de la création d'un site web utilisant le système EVE, l'architecture est la suivante.



En premier lieu, l'application doit être placée dans le dossier Applications. L'application est constituée d'un dossier dont le nom correspond au nom de l'application.

L'appel de cette fonction doit être faite depuis le fichier indiqué dans le .htaccess. Dans note exemple, le .htaccess envoie toutes les requêtes situées ailleurs que dans le dossier Web sur le fichier eve.php qui va lui-même faire appel à la classe \Applications\Eve\EveApplication.

Ensuite, l'ensemble des modules sont placés dans le dossier Modules. Leur accès se fait via l'URL et sa correspondance dans la table des routes. Le nom d'un module doit impérativement être en corrélation avec le nom du controller. C'est-à-dire que si le nom du module est [nom du module], le nom du contrôleur doit être [nom du module]Controller et le nom du fichier doit être le même suivi de .class.php

L'ensemble des autres dossiers contient des classes utiles, mais qui ne devraient pas être modifiées pour faire fonctionner le système. Elles doivent être suffisamment neutres pour pouvoir fonctionner sur tous les systèmes et en personnaliser une risquerait de rendre le système instable ou trop lourd pour d'autres applications.

S'il est impossible de faire fonctionner un module sans une version personnalisée d'une des différentes classes, alors il est conseillé de créer une sous-classe de la classe en question. La majorité des attributs et des méthodes non publics sont protected, ce qui permet de facilement en hériter sans perte de fonctionnalité.

L'ensemble des ressources graphiques, CSS, Javascript et autres qui doivent pouvoir être chargés par le client doivent impérativement se trouver dans le dossier Web. Il s'agit du seul dossier qui n'est pas protégé par le .htaccess. Tous les liens sur les autres dossiers étant automatiquement redirigés sur le fichier

eve.php

Sécurité

Un certain nombre d'éléments de sécurité ont été mis en place afin de protéger l'utilisateur et le système des attaques externes.

- Hashage des mots de passe

Tous les mots de passe en base de données sont hachés avec la fonction crypt de PHP et utilisent l'algorithme Blowfish. De plus, afin de se prémunir contre les rainbowtable un sel a été ajouté. Les mots de passe sont donc relativement sûrement protégés.

- Usage de PDO avec les requêtes préparées

Les requêtes sont pour la majorité préparée avec PDO, ce qui protège efficacement contre les injections SQL. **Pour les autres requêtes, un soin tout particulier doit être fait pour empêcher les injections SQL, mais elles sont normalement relativement peu nombreuses et leur contenu ne doit jamais être des informations qu'un utilisateur pourrait avoir manipulé sans contrôle.**

- Usage d'un .htaccess

Un fichier .htaccess est présent à la racine du système. Son rôle est de protéger toute intrusion dans l'un des dossiers non protégés. Il doit donc rediriger tout accès à l'un des sous-dossiers sur le fichier racine.

- Usage d'une constante prédéfinie

Une constante est définie sur le fichier racine, et chaque autre fichier doit tester la présence de cette constante ou s'arrêter. De cette manière, on peut s'assurer que les fichiers sont bien accédés depuis le fichier racine, et donc dans un schéma classique. On peut ainsi se prémunir d'accès direct aux différents documents.

- Usage d'un fichier index.php

Chaque dossier contient un fichier index.php qui s'affichera donc si quelqu'un parvient à accéder au dossier. Le dossier ne peut donc pas donner la liste des éléments du fichier.

- Usage de fichier de Log

Chaque erreur et chaque action critique doit être enregistrée dans le fichier de log. De cette manière il est possible de déceler les actions potentiellement litigieuses. Les fichiers de logs contiennent l'adresse IP ainsi que différentes autres informations à propos de l'utilisateur. Ce qui peut nous permettre de le retrouver le cas échéant, ou de détecter des tentatives de connexion depuis des IP inconnus (hameçonnage ou autre)

- Fichier d'erreur personnalisé

Lors d'une erreur 404 ou 403 un fichier d'erreur spécifique a été créé pour retourner l'erreur. Dans le cas d'une erreur, elle doit être catchée par le système pour être traitée à la fin, de manière à s'assurer que le système ne va pas afficher d'informations secrètes.

- URL personnalisée

Les URL personnalisées n'ont pas de lien avec l'architecture des dossiers. Il n'est donc pas possible de retrouver un fichier depuis un URL donné. De plus, l'usage de pattern pour la récupération des informations assure de ne pas avoir de caractère non prévu dans une URL.

- Contrôle de l'IP et de l'ID de session

A chaque chargement d'une page, un contrôle est fait que l'IP donné avec l'ID de session donné est bien le même que précédemment. De cette manière on peut s'assurer qu'il n'y a pas eu de vol de session.

- Contrôle de temps de connexion

A chaque changement de page un contrôle est fait que le précédent changement de page a eu lieu dans une durée spécifique. Cela afin de s'assurer que l'user n'a pas oublié de se déconnecter et qu'une tierce personne ne vienne lui voler son compte par la suite.

Création d'un nouveau module

Tout ajout de nouvelle fonctionnalité au système passe par la création de module. Il est donc impératif de pouvoir facilement en créer de nouveau. C'est en ce sens que l'architecture de l'application a été pensée. La création de nouveau module se veut très simple d'une part, et surtout chaque module se doit

d'être indépendant des autres dans sa construction. Tout appel d'un module à un autre doit donc se faire par l'appel d'une page, et non à l'instanciation d'une partie d'un autre module. De cette manière, si un module est supprimé, les autres modules peuvent toujours fonctionner.

Un module est constitué de différents éléments. C'est-à-dire :

- Le contrôleur du module

Ce module doit contenir une méthode pour chacune des actions n'ayant pas de contrôleur défini pour cette action.

- Un dossier contenant les vues

Ce dossier contient les vues de toutes les actions qui en nécessitent une. Les vues ne sont pas nécessaires pour afficher certaines pages.

- Un dossier contenant les entités

Ce dossier doit contenir toutes les entités nécessaires au bon fonctionnement du module. Le contenu de ce dossier est donc étroitement lié à celui du dossier model.

- Un dossier contenant les fichiers de langues d'action

Ce dossier doit contenir un fichier de langues par action et par langue supportée par l'application. Si une information n'est utilisée que par une des actions d'un des modules, alors cette information doit se trouver dans ce dossier. Si un fichier de langue n'existe pas il essaiera de charger le fichier de langue par défaut, et s'il n'existe pas non plus, il n'en chargera pas.

- Un dossier contenant les fichiers de langues globaux

Un dossier contenant les fichiers de langues globaux du module dans chacune des langues possibles de l'application. Les constantes présentes dans ces fichiers de langues devraient être toutes les constantes qui se trouvent dans plus de deux actions du module. Si le fichier de langue de la langue choisie n'existe pas, le système va essayer de prendre le fichier de langue de la langue par défaut du système.

- Un dossier contenant les models

Un dossier contient l'ensemble des models nécessaires pour le fonctionnement du module. Il est important de noter que le model doit être du même type de DAO que l'application. Si l'application change de DAO, il faudra alors réécrire des models correspondants.

- Un dossier contenant les actions

Si une action nécessite un contrôleur spécifique, il est nécessaire d'avoir un dossier contenant ce contrôleur.

De plus, pour fonctionner il faut que certaines routes utilisent ce module. Il faut donc inscrire dans la table des routes une URL spécifique, le nom de ce module, l'action de ce module qui doit être exécutée, les différentes variables que le pattern peut retourner et le niveau d'accréditation nécessaire pour accéder à cette route.

Détail

Pour l'explication de la création de ce module, nous allons admettre que le nom du module est [Module], que le nom de l'action est [Action], que cette action s'exécute lors de l'accès à la page monsite/part/part-2/page.html et qui indique que le paramètre ID vaut 2.

Dossier du module

Le dossier du module doit se placer dans le dossier Module et il doit impérativement se nommer [Module]. C'est le seul moyen que l'on a de savoir comment y accéder par la suite.

Contrôleur du module

Le contrôleur du module doit se nommer [Module]Controller dans un fichier [Module]Controller.class.php et il doit hériter de \Library\BackController. De plus, son namespace doit être Module\[Module].

S'il n'existe pas de controller spécifique pour l'action [Action] alors le controller global doit contenir une méthode nommée execute[Action] qui a un paramètre de type \Library\HTTPRequest et qui est le

HTTPRequest de l'application. C'est cette méthode qui sera appelée en premier.

Cette méthode peut parfaitement faire appel à d'autres méthodes, mais il y aurait alors un risque important de surcharger la class. Il est alors conseillé d'utiliser un controller spécifique pour cette action.

Controller

Comme exprimé précédemment, il y a deux cas de figure possibles pour le controller.

Contrôleur d'action spécifique

Le contrôleur spécifique est une classe dédiée au contrôle d'une action spécifique. Cela peut être utile pour une action qui doit effectuer un grand nombre de choses, de manière à ne pas avoir une seule grosse méthode, ou de multiples méthodes dans une seule classe.

Ce controller doit se trouver dans un dossier nommé Controller. Ce dossier doit se trouver à la racine du dossier du module. La classe doit se nommer [Action]Controller. Elle doit hériter de la classe \Library\ActionController et son namespace doit être Module\[Module]\Controller

Ensuite, cette classe doit contenir une méthode nommée executeAction qui a un paramètre de type \Library\HTTPRequest, l'instance HTTPRequest de l'application. C'est cette méthode qui sera appelée en premier.

Le controller peut utiliser des instances de Models pour récupérer des données ou pour les envoyer sur le serveur. Il doit ensuite inscrire différentes variables de l'instance de PageGenerator contenue dans l'application.

Exemple :

Le contrôleur génère une variable $content qui contient un certain nombre d'informations. Nous souhaitons pouvoir accéder à ce contenu dans une variable $var lorsque nous serons dans la vue. Pour ce faire, il nous suffit, depuis le controller, d'utiliser la méthode $this->app()->page()->addVar('var', $content);

Vue

Les vues sont le contenu qui sera retourné à l'utilisateur. Suivant le type de vue souhaité, le format de données à utiliser sera plus ou moins différent. Dans tous les cas, le fichier de la vue n'est jamais obligatoire. On peut admettre retourner une page blanche dans de nombreux cas.

- Json – XML

Le fonctionnement de ces deux types de données est similaire. Dans les deux cas, la page se contente de formater les différentes variables transmises à la page pour mettre ces données dans un format Json/XML. Il n'y a donc pas de vue pour ces types de page.

- HTML

Lors d'une page HTML, le contenu de la vue doit être au format HTML. Toutes les différents variables sont fournies depuis le controller en accès directe. De plus, certaines variables sont transmises automatiquement, c'est-à-dire :

  - $root : contient le lien d'accès pour le fichier langue, utile pour les images ou les fichiers dans Web
  - $rootLang : contient le lien suivis de la langue choisie, utile pour les liens
  - $user : une instance de \Library\User permettant d'accéder aux informations d'authentification, etc...

- PDF

Nous utilisons la classe Html2Pdf qui nous permet de formater automatiquement nos pages depuis du HTML en pdf. Il existe malgré tout quelques contraintes.

  - Créer une page PDF se fait entre deux balises <page format="[Format]">(...)</page> la balise format n'est pas obligatoire et vaut A4 par défaut.
  - La transformation des balises HTML en élément PDF n'est pas parfaite. Il est donc important de tester chaque page et d'être très stricte dans l'usage des fonction.

    - Le template n'est pas chargé. Les CSS ne sont donc pas chargsé non plus. En cas d'usage de feuille de style CSS, il faut les charger depuis cette page spécifiquement.

- IMG

une variable nommée $image représente l'image. Il s'agit d'une instance de imagecreatetruecolor. Il s'agit donc d'utiliser différentes méthodes sur cette image.

De plus, tout le texte qui sera produit lors de la vue sera ensuite placé sur l'image. Cela permettant entre autres choses d'afficher les erreurs éventuelles.

Il n'est pas possible de créer un contenu HTML pour l'afficher sur l'image automatiquement.

- File

Il s'agit ici d'un proxy pour charger automatiquement des fichiers sans que le lien du fichier ne soit inscrit dans l'URL, pour des raisons de sécurité. Il n'y a donc pas de vue pour de type de page.

La vue doit être un fichier nommé [Action].php ([Action].pdf.php, [Action].img.php) pour une page HTML (respectivement PDF et IMG, bien que si les compléments de nom de fichier ne sont pas inscrits, le fichier sans extension soit utilisé) et placé dans un dossier view lui-même placé à la racine du module. Il ne s'agit pas ici de classe mais d'éléments procéduraux. La page sera chargée par un include si elle existe.

Entité

Les entités doivent se placer dans le dossier Entities, dossier qui doit être placé à la racine du module. Ces entités doivent hériter de \Library\Entity et leur namespace doit être Module\[Module]\Entities . Le format de ces entités doit être la seconde partie du nom de la table de la base de données, et chacun de ses attributs doit correspondre aux entrées de colonnes de la base de données correspondante (du moins tant que l'on utilise PDO).

Pour rendre un élément obligatoire lors de l'enregistrement de l'information, il suffit également d'inscrire une constante de classe nommée INVALID_[NOM_DE_L_ARGUMENT] avec une valeur unique, c'est-à-dire qu'aucune autre constante de classe doit avoir le même argument. Cela implique automatiquement que lorsqu'on cherche à inscrire une valeur dans cet argument via le setter automatique, si la valeur est vide, alors une erreur sera retournée.

Exemple

Si une base de données se nomme [Module]_[Entity] et contient comme élément atr1 et atr2, notre entité doit s'appeler [Entity] dans un fichier nommé [Entity].class.php et elle doit contenir deux attributs, $atr1 et $atr2. Si atr1 est obligatoire pour le bon fonctionnement de l'application, l'entité doit également disposer d'une constante nommée INVALID_ATR1 avec une valeur qu'aucune autre constance de la classe ne possède.

Fichier de langue

Les fichiers de langue globaux doivent se trouver dans le dossier ModuleLang qui se trouve à la racine du module. Les fichiers de langues d'action doivent se trouver dans le dossier Lang à la racine du module. La nomenclature est la suivante. Les fichiers généraux doivent se nommer [Module].[CONST_LANG].php et les fichier de langue d'action doivent se nommer [Action].[CONST_LANG].php . Leur contenu doit simplement être une suite de fonction define de PHP

Exemple

Si l'on souhaite inscrire la valeur « ma doc » en français et « my doc » en anglais, il suffit de créer une constante define("CONST_DOC", "ma doc"); dans le fichier [Module].fr-FR.php et define("CONST_DOC", "my doc"); dans le fichier [Module].en-EN.php, et d'inscrire CONST_DOC dans la vue. De cette manière, la valeur de la constante changera en fonction de la langue.

Models

Pour chaque DAO et chaque entité, il doit exister une classe dans Model, plus une par entité. Il doit y

avoir une interface de class générale pour chaque entité, contenant toutes les méthodes que la classe doit posséder pour être certain que l'application n'aura pas de problème.

Tous les fichiers de models, classes ou entités, doivent se trouver dans le dossier Models qui se trouve à la racine du module.

Cette interface doit se nommer, pour une entité nommée [Entity], [Entity]Manager, et être dans un fichier nommé [Entity]Manager.class.php. Son namespace doit être Module\[Module]\Models .

De plus, pour chaque [DAO], il faut une classe supplémentaire pour pouvoir accéder aux différentes données. Cette classe doit hériter de \Library\Manager (ou \Library\Manager_[DAO] s'il existe) et avoir comme namespace Module\[Module]\Models . Cette classe doit se nommer [Entity]Manager_[DAO] dans un fichier [Entity]Manager_[DAO].class.php . On peut ainsi s'assurer que toutes les différentes méthodes nécessaires seront implémentées.

Il est impératif que tous les models soient redéfinis lorsque l'on change de DAO, sans quoi l'application ne parviendra pas à charger les classes.