

ean FORTEROCHE

[Accueil](#)
[Biblio](#)
[A propos](#)
[Accès](#)



This is all about Jean

A propos de JEAN :

Après avoir remis sa baguette magique, son sac à malice et ses bottes de sept lieux, Jean décida de mettre à profit ses multiples talents pour Ecrire -avant un grand E.


Sous de multiples noms de plume, elle vous livre des histoires -plus et moins primées, des songes de fantaisies urbaines, historiques ou extra-logiques. Carburant au chocolat et au jus d'orange, elle n'a de cesse de s'échapper toujours hors des sentiers battus. Il suffit pour s'en convaincre de demander à sa famille ou à son chat zombie.

Après une décennie d'écriture de manuels techniques et de plans de projets ennuyeux, JEAN a décidé de devenir auteur, explorateur, et puisqu'il faudra probablement en passer par là: acteur d'un monde meilleur...

Elle met ainsi à profit son imagination débridée, ses relations, ses talents d'écrivain et plus quand affinités. Ses compétences en matière d'élaboration de plans et de listes de tâches s'adaptent également à ses activités non fictives, car elle travaille aussi avec d'autres, présente des ateliers populaires en ligne et participe à des conférences d'écriture. Elle a écrit de nombreux articles sur son blog et ailleurs, sur l'économie, la politique, les affaires et la vie de l'écriture, et son site sera bientôt nommé l'un des 100 meilleurs sites Web pour les écrivains.

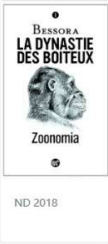



Sur le plan personnel, elle a déménagé une dizaine de fois sur les 20 dernières années; a vécu dans plus de cinq pays et aux quatre coins de notre vaste monde, mais elle affirme ne pas avoir évité les autorités. (Uh-huh, bien sûr...) Pour preuve, elle fait remarquer qu'elle est maintenant installée au gai Paris, dans la cité lumière de cette bonne vieille France; avec sa famille et ses muses -dont divers hommes arrogants qui assurément finiront par la rendre plus folle qu'elle n'est déjà.

Merci pour votre visite!



Lady South

A découvrir en Librairie, sur Amazon ou ailleurs

[Accueil](#)
[Biblio](#)
[A propos](#)
[Accès](#)

- Copyright © 2019 <http://localhost/jfrblog/>- Politique de confidentialité- Design & développement

Auteur : RODRIGUES NGUEMA

Mentor : Sandy RAZZAFITRIMO

OPENCLASSROOM formation 2019

-Chef projet multimédias option Développement

INTRODUCTION.....	3
CONTEXTE	3
Donne initiale	3
Démarche étudiant.....	3
SPECIFICATION DES BESOINS.....	4
Besoins fonctionnels.....	4
Logiciels	4
Autres.....	4
SOLUTION PROPOSEE.....	5
Arborescence	5
Description fonctionnelle des écrans	5
ARCHITECTURE.....	6
BASE DE DONNEES MYSQL.....	6
Contraintes relationnelles	7
CONCEPTION GENERALE DE L'APPLICATION.....	8
Pré requis	8
Patron de conception MVC.....	8
POO Programmation orienté objet.....	9
Organisation retenue	9
MISE EN OEUVRE APPLICATIVE	10
Front controlleur, configuration, htaccess, autoloader	10
Détail des Classes et Objets créés.....	11
Objets liés au Router.....	11
L'objet View	11
Les objets Model.....	12
Entités.....	12
Managers	13
Les objets Controller.....	14
FOCUS : FORMULAIRES ET GESTION PHP	15
BILAN.....	17
AXES D'AMELIORATION.....	ERREUR ! SIGNET NON DEFINI.
Saisie et restitution des textes : l'Editeur en ligne.....	17
Fonctionnel ou technique.....	17

INTRODUCTION

Contexte

Donne initiale

Jean Forteroche -acteur et écrivain, travaille actuellement sur son prochain roman :

"Billet simple pour l'Alaska". Il souhaite innover et le publier par épisode en ligne sur son propre site.

L'objectif est de développer une application de blog simple avec une interface frontend (lecture des billets) et une interface backend (administration des billets pour l'écriture).

Chaque billet doit permettre l'ajout de commentaires, qui pourront être modérés dans l'interface d'administration au besoin.

Les lecteurs doivent pouvoir "signaler" les commentaires pour que ceux-ci remontent plus facilement dans l'interface d'administration pour être modérés.

L'interface d'administration sera protégée par mot de passe. La rédaction de billets se fera dans une interface WYSIWYG basée sur TinyMCE.

Préalables techniques

Développement en PHP.

Base de données MySQL

Code construit sur une architecture MVC et développé autant que possible en orienté objet.

Interface WYSIWYG basée sur TinyMCE

Sources actualisés du projet remontés en ligne sur GITHUB.

Démarche étudiant

D'une façon générale, les sites d'auteur offrent plus de fonctionnalités que strictement demandé par Jean. Articles et réflexions divers y côtoient souvent des liens « promotionnels » vers d'autres publications de l'auteur. Même si les problématiques de reprise de l'existant n'ont pas été abordées, on imagine que Jean -qui est également acteur, a déjà plusieurs romans à son actif et que son blog sera amené à évoluer.

La préoccupation principale a été de répondre au réel besoin de l'auteur et de mettre à sa disposition une application simple, fonctionnelle et évolutive.

Après traduction des besoins en fonctionnalités -restitutions écran, formatage et hiérarchisation des informations à traiter ; l'intégration des prérequis techniques -patern MVC, base de données MYSQL, programmation orientée objet ... a obligé dès le départ à structurer les fonctionnalités applicatives et le code de l'outil proposé.

Le rendu final est insatisfaisant à plus d'un égard, mais nous espérons que cette réalisation saura intégrer toutes fonctionnalités souhaitable -ou à implémenter avant présentation du projet à l'auteur -ou mise à disposition pour phase de pré recette étendue.

Besoins fonctionnels

Spécification des besoins

L'application web proposée permettra de gérer un catalogue restreint à présenter en ligne, et de gérer en parallèle les épisodes du roman en cours d'élaboration. Elle visera aussi à faciliter la circulation du visiteur dans le contenu mis à sa disposition.

Les visiteurs doivent accéder aux contenus publiés en ligne par l'auteur, et de faire remonter leurs

appréciations ; **l'application pourra donc mettre à disposition en « front » les fonctionnalités suivantes :**

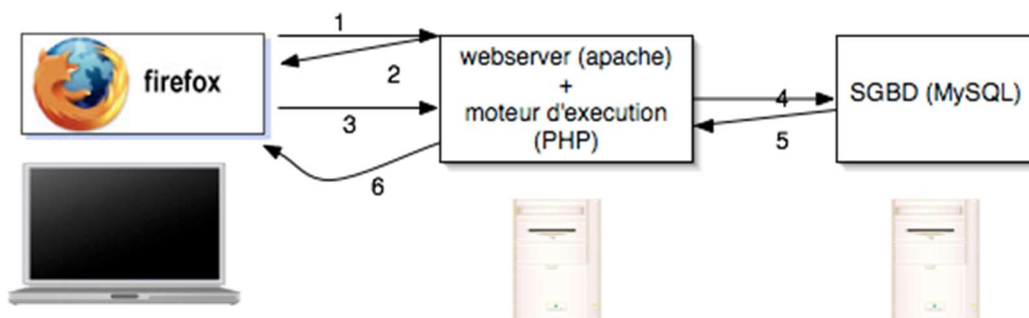
- Consultations des dernières publications mises en ligne - le ou les derniers épisodes du roman en cours ;
- Consultation des autres écrits en ligne - épisodes ou romans antérieurs, page à propos ...
- Saisie et signalement de commentaires.
- Enregistrement possible et accès tout public aux fonctionnalités de login et logout de l'application. (*)

L'auteur aura également besoin de fonctionnalités d'administration propres au back-office:

- Création, modification et suppression d'un livre -épisodes ou article ; pilotage des mises en ligne.
- Modération des commentaires et signalements émis par les visiteurs : validation ou suppression.
- Affectation de droits aux visiteurs enregistrés -abonnés amis. (RGPD à implémenter à terme) (*).

Environnement Matériel et Logiciels

Hors modalités d'hébergement, l'installation sera basée sur une architecture **Apache** (serveur Web) +**PHP** (langage de script)+**MySQL**(SGBD).



Bootstrap CDN en ligne -gère actuellement le *responsive design* du site ;

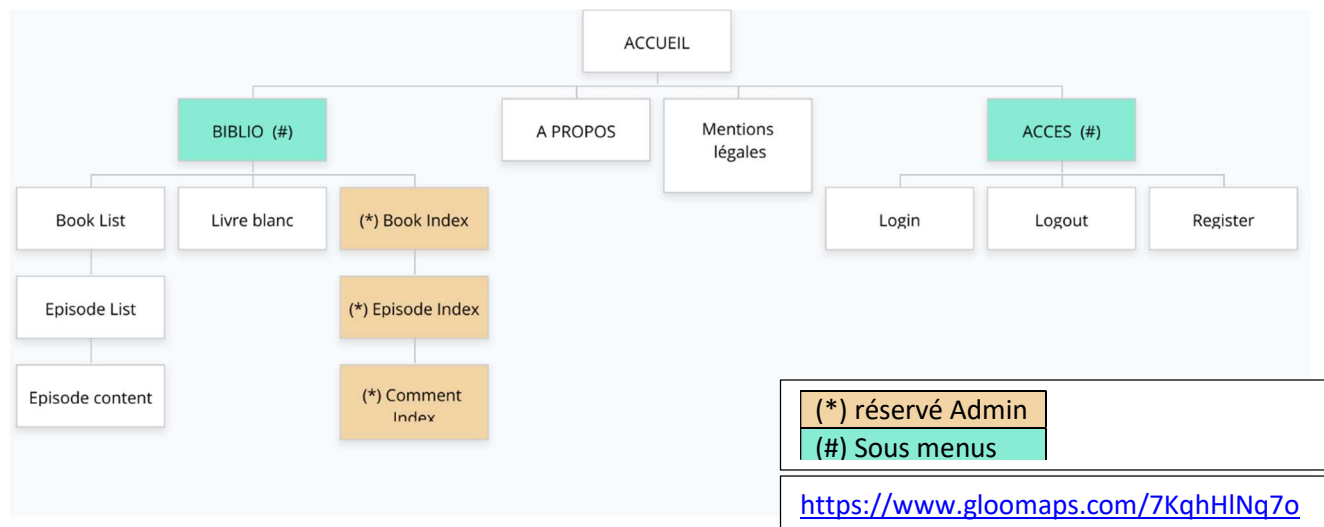
TinyMce -également en ligne, permettra à l'auteur de gérer sa mise en page.

Autres besoins

- Confort d'utilisation et de lecture : l'application doit être intuitive, et la lecture des textes facilitée par la présentation : intégration d'image pour aération, taille maximale des lignes de texte (90 caractères).
- Le temps de réponse de l'application doit être satisfaisant.
- L'application doit être portable, responsive et avoir des possibilités d'optimisation et d'évolution.
- L'application devra sécuriser les informations de ses utilisateurs (* cf RGPD et niveau de gestion des utilisateurs à intégrer).

Solution proposée

Arborescence



Description fonctionnelle des écrans

Désignation	Fonction principale	Action autorisées	Maj BD
ACCUEIL	Présentation du dernier épisode en ligne	Saisie commentaire et signalement	X
Book List	Liste des publications en ligne		
Episode List	Liste des épisodes par publication		
LIVRE BLANC	Liste des épisodes/article partie blog.		
Episode Content	Visuel du contenu d'un épisode + affichage liste des commentaires associés.	Saisie commentaire et signalement	X
A PROPOS	Présentation de l'auteur		
(*) Book Index	Liste de gestion des livres	création, mise à jour, suppression	X
Episode Index	Liste de gestion des épisodes	création, mise à jour, suppression	X
Comment Index	Modération des commentaire	Validation, suppression	X
Login/logout	Connexion/déconnexion		
Register	Inscription en ligne	Maj. Base de données	X

Architecture

L'application mise en place peut être découpée fonctionnellement en 3 couches distinctes.

- *L'interface utilisateur* : Pages HTML d'entêtes, menus, pieds de page, listes, formulaires etc.
- *La couche métier* : fonctions intégrées et "logique" de l'application elle-même. Si l'on crée une fiche ouvrage par ex. l'application prendra les données saisies par l'utilisateur dans le formulaire, vérifiera la cohérence des informations, mettra la requête dans un format adéquat, et la transmettra au gestionnaire des données (couche accès aux données).
- *La couche d'accès aux données* : le SGBD Mysql. Base de données Mysql

La base de données utilisée MYSQL est une base de données relationnelle. Elle permet de modéliser et d'organiser facilement les données nécessaires à l'application et leur transcription en tables.

La base de données » **jfrblog** » est encodée en utf-8.

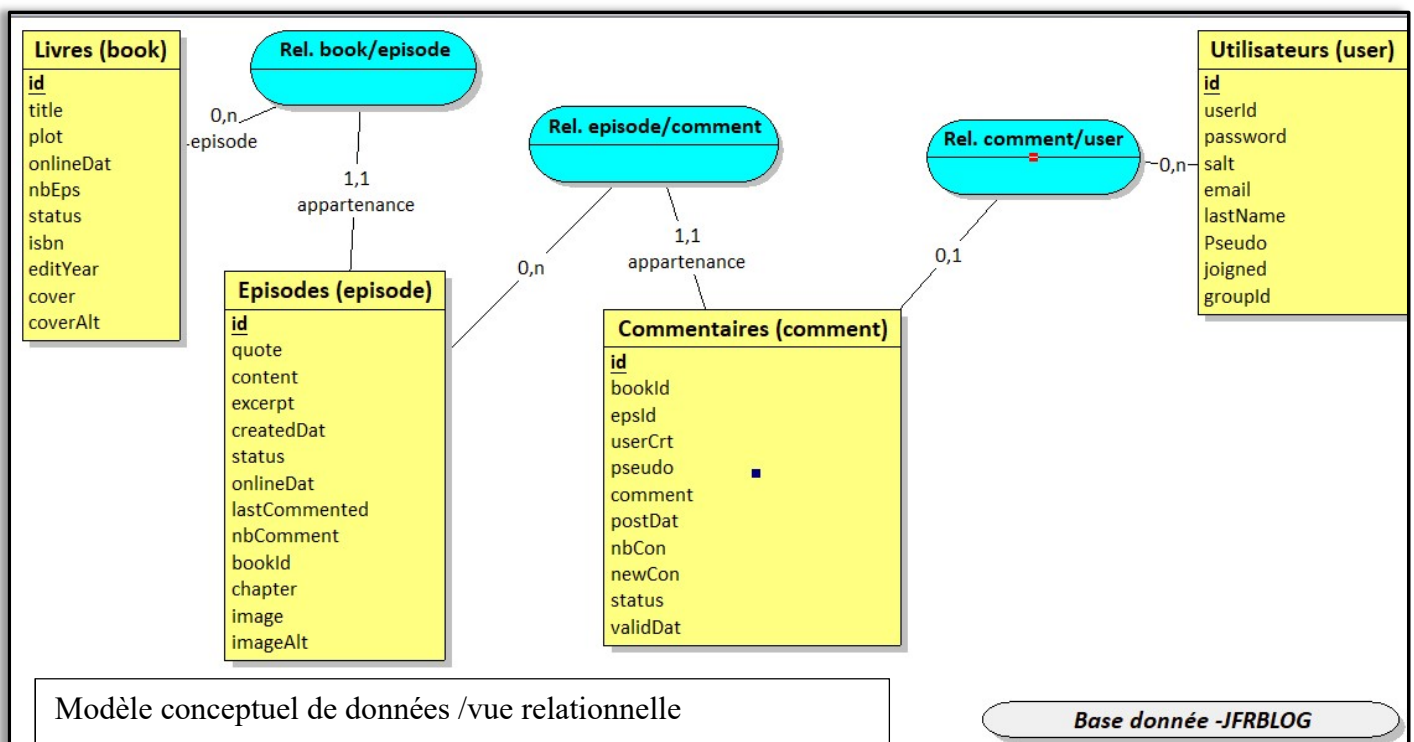
Le moteur de stockage retenu pour les tables est InnoDB.

- Son principal avantage est de permettre de valider ou d'invalider de façon logique des ensembles de transactions (commit Rollback) ; garantie de cohérence fonctionnelle.
- Les mises à jour et suppression sur les tables peuvent s'accompagner d'un verrouillage des tables -et non des enregistrements, en share Read non update.

Cette fonctionnalité sera surtout utile pour éviter les mises à jour concurrentes, ou intégrer un volume conséquent de modification -dans le cas de la reprise globale d'un livre et de ses épisodes par ex.

- En cas de crash du serveur, il est réputé récupérer automatiquement les données.

Dans le cas de la base « **jfrblog** », innoDB va permettre de gérer des clés étrangères, de sécuriser les transactions et éventuellement de déclencher des remises à niveau cohérentes.



Contraintes relationnelles

Les contraintes et règles de cohérence entre table ont été gérées pour partie par l'application. Certaines ont été ajoutée après coup dans la base de données relationnelle -grâce aux fonctionnalité innoDB.

Table relation / règle appareillage	Contraintes relationnelles : (*) si retranscrites dans la base sql.
book a de 0 à N episode Book.id = episode.bookId	Book/ Impossible de supprimer un livre s'il comporte des épisodes Episode/ l'identifiant du livre est requis et doit exister. (*)
episode a 0 à N comment episode.id = comment.epsId	La suppression d'un épisode doit entraîner la suppression de tous ses commentaires associés. (*)
book 0 à N comment book.id = comment.bookId	
user a 0 à N comment	La suppression d'un utilisateur doit entraîner la suppression de tous ses commentaires associés.

Contraintes formalisées dans la base de données sql ;

Propriétés de la contrainte			Contrainte de clé étrangère (INNODB)		
Nom	On Delete	On update	colonne	Table	colonne
episode_book	No action	No action	bookId	book	id
Comment_episode	cascade	No action	epsId	Episode	Id

(*) les possibilités d'activation de trigger interne à la base sql restent à étudier.

Conception générale de l'application

Pré requis

Patron de conception MVC

Un patron de conception ou un design pattern est un concept destiné à résoudre les problèmes récurrents du génie logiciel suivant le paradigme objet. Les patrons de conception décrivent des solutions standards qui répondent à des problèmes d'architecture et de conception des logiciels par une formalisation de bonnes pratiques.

Pour notre application web, les pré-requis engagent à utiliser le pattern MVC.

Le Modèle-Vue-Contrôleur (en abrégé MVC, de l'anglais Model-View-Controller) organise l'interface homme-machine (IHM) d'une application logicielle en 3 composants principaux :

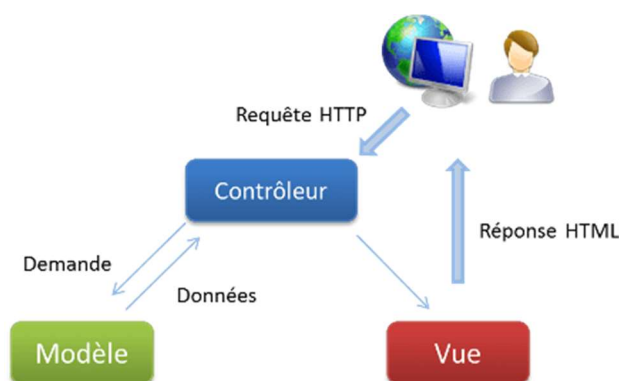
- **Modèle** : noyau de l'application, gère les données, permet de récupérer les informations dans la base de données, et de les organiser pour qu'elles puissent ensuite être traitées par le contrôleur ou la vue.
- **Vue** : composant graphique IHM ; permet de présenter les données du modèle à l'utilisateur.
- **Contrôleur** : composant en charge de la logique de contrôle, la gestion des événements et leur synchronisation. Il gère la logique du code et est l'intermédiaire entre le modèle et la vue.

Les avantages :

-Formalisation de l'application qui faciliter la compréhension de son mode de fonctionnement.

-La distinction des différents modules de l'application simplifie de leur création.

-La séparation des modules facilite les modifications sur l'un d'eux sans impacter le reste du code ou le fonctionnement global de l'application.



*NB : Une manière d'optimiser le MVC est d'utiliser par exemple le patron **DAO** (Data Access Object), pour mieux gérer la persistance des données en l'isolant du reste du traitement. Les objets en mémoire vive sont souvent liés à des données persistantes (stockées en base de données, fichiers, annuaires...). Le modèle DAO propose de regrouper les accès à ces données persistantes dans des classes à part, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes "métier", qui ne seront modifiées que si les règles de gestion métier changent.*

POO Programmation orienté objet

Les composants de l'application -ou objets, intègrent des « attributs » et des « méthodes ».

L'architecture orientée objet s'appuie sur 3 piliers : encapsulation, héritage et polymorphisme.

L'encapsulation concerne l'architecture détaillée de chaque objet. Les données sont protégées d'un accès direct par une couche d'interface (getters/setter). Les sous-fonctions, inutiles pour utiliser l'objet, sont masquées à l'utilisateur de l'objet.

L'héritage permet d'éviter la redondance de code et facilite l'extensibilité du logiciel, les fonctionnalités communes à plusieurs classes d'objets pouvant être regroupées dans un ancêtre commun.

Le polymorphisme permet d'utiliser des objets différents (possédant des comportements distincts) de manière identique, cette possibilité est réalisée par la définition d'interfaces à implémenter (classes abstraites).

Organisation retenue

La gestion de l'application jfrblog utilisera différents types d'objet pour gérer les livres, les épisodes, les commentaires et les utilisateurs, et également pour piloter ses fonctionnalités internes : gestion des routes, accès à la base de données, session etc...

Outre l'implémentation du MVC, les caractéristiques et fonctionnalités communes seront remontées autant que possible dans des objets de niveau supérieur ce qui a pour 1^{er} objectif de faciliter une gestion centralisée des fonctionnalités et du code.

L'application elle-même n'a pas été éclatée en application front-end et back-end, mais a été découpée en 5 modules fonctionnels principaux :

- La gestion de la page d'accueil
- La gestion des livres
- La gestion des épisodes
- La gestion des commentaires
- La gestion des utilisateurs

A chacun de ces modules a été affecté un contrôleur et un modèle de gestion des données comprenant un manager et une entité -quand il existe une table de référence en base de données.

Les objets « modèle » -comme les objets Controller, Manager, View, ou les objets partagés -comme DB, Session, Validate, sont regroupés dans un dossier « core » ;

les contrôleurs dans le dossier « controller », les manager et entités dans le dossier « manager ».

Un dossier image a été intégré à l'application avec des sous dossiers standardisés par taille d'image.

Les restitution écrans sont regroupées dans un dossier « view » et répartis dans les sous dossiers dédiés par module (book, episode, comment, user, home).

Les vues communes layout, header, footer, menu sont à la racine du dossier « view ».

Htaccess, Front contrôleur, configuration et autoloader

Quelque soit la demande -utilisateur ou système : accès par l'url, en GET ou en POST,

l'application renvoie au fichier **index.php** qui fait office de front contrôleur. -réécriture d'url par .htaccess.

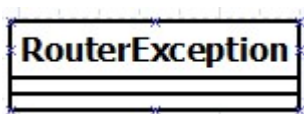
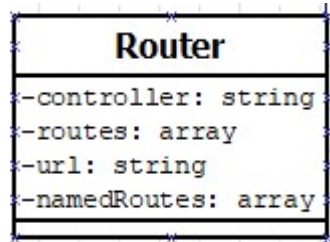
Index.php inclue le fichier de configuration de base (config.php), qui définit les constantes d'accès aux différents dossiers model, vue, controller, image etc... et met en place un autoloader (recherche et l'inclusion automatique des classes).



Détail des Classes et Objets créés

Objets liés au Router

Il est implémenté à l'aide de 2 classes principales et des méthodes ci -près :

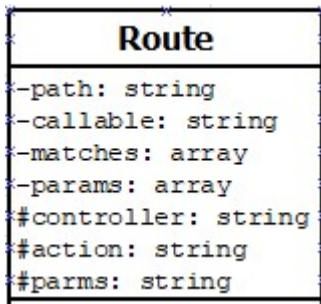


Méthodes/

addRoute() Instancie des objets **Route** décrits par index.php. Cette méthode crée un tableau des objets routes valides en GET et en "POST.

Run() Si la route est identifiée par l'appel de la méthode **Route->match()**, le routeur appelle la méthode **Route->run()**

Si problème il émet une erreur.



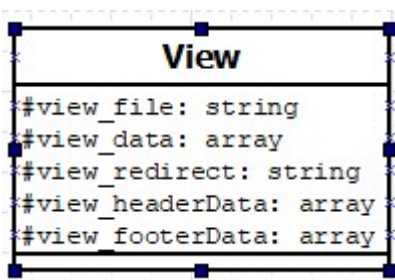
Méthodes/

paramMatch() et match() gèrent l'identification d'une route existante ;

run() instancie le contrôleur du module à traiter et appelle la méthode demandée (index, edit, show, liste etc...) du contrôleur dédié.

L'objet View

L'objet View gère l'affichage des informations ou des formulaires demandées.



Il est instancié par le contrôleur

L'objet **View** pilote la mise en forme de la page -méthode render().

temporisation et bufferisation des données propres à la vue ;

Chargement de la vue à traiter

Restitution de cette vue dans une variable \$content

Chargement des informations d'entête et pied de page

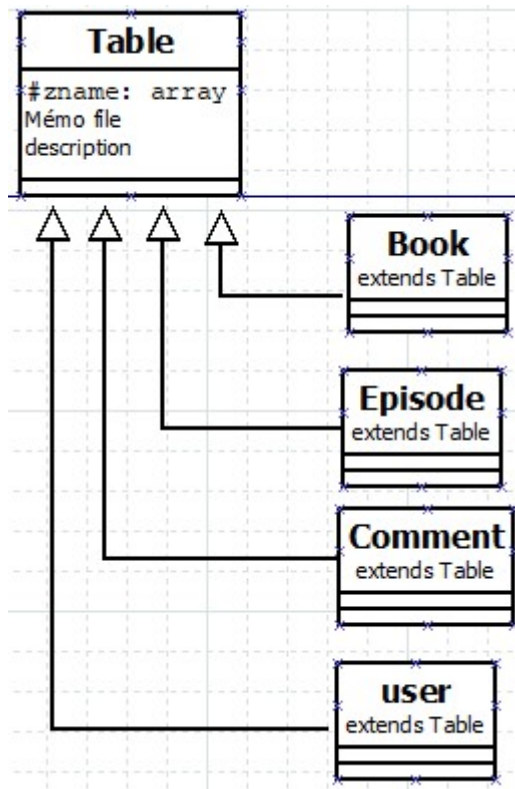
Inclusion du layout -dont header et footer.

En pratique les vues gèrent un certain nombre de règles d'affichage qui n'ont pas été intégrées aux niveaux entité par ex.

Les objets Model

Le model est divisé en 2 parties les entités et managers

Entités



L'objet [Table](#) centralise les attributs et méthodes communes aux entités :

- hydrate() – gestion transparente des appel au Setters
- getFfd() – récupération des descriptions fichiers (Field File Description) via appel méthode de l'objet [DB](#)
- createPost() – complément automatique des POST avec données initiales de la BD.

Les entités étendent l'objet Table.

Initialement limités aux attributs des tables de la base de données, et aux fonctions d'encapsulation correspondantes (setters et getter), les entités ont été complétés avec :

des attributs nécessaires à l'application : consolidation d'informations de niveau N-1 ou attributs spécifiques à transmettre aux vues (libellés par ex.), et les -getter et setter correspondants.

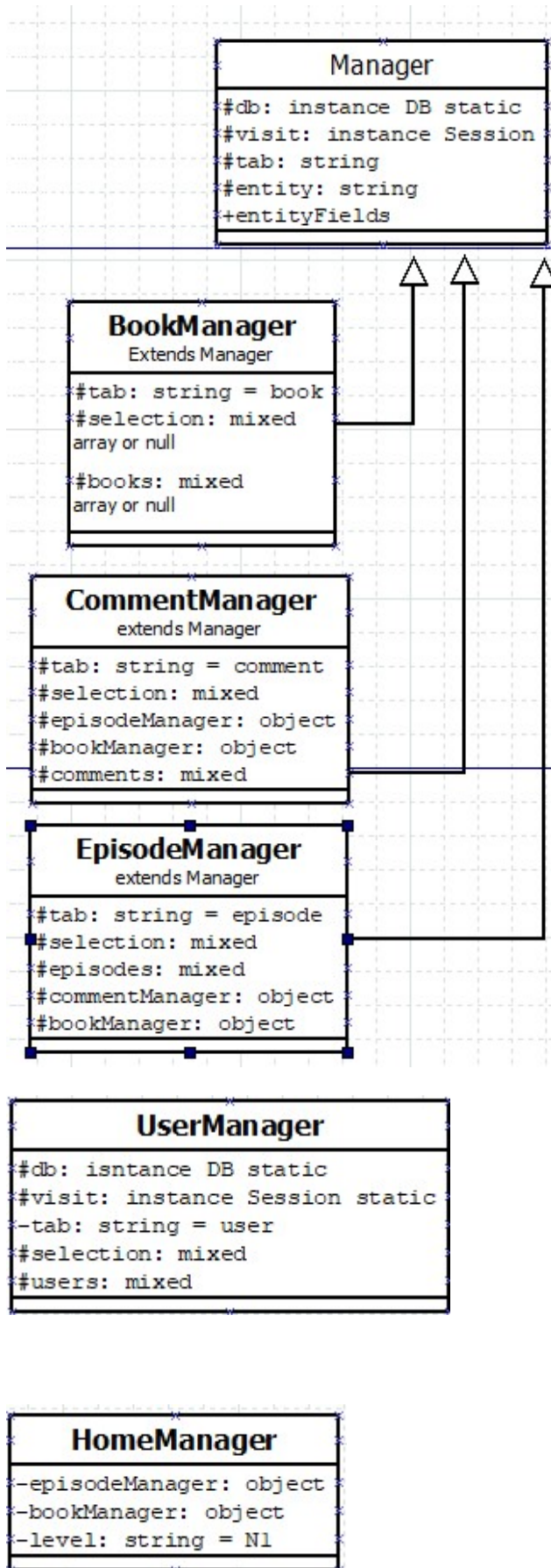
L'entité porte également des règles de contrôle unitaire ou de cohérence -préalables aux mises à jour en base de données. La méthode statique **validation()** fournit pour chaque entité : un tableau de contrôle par information identifiée. Le tableau est exploité ultérieurement par un objet [Validate](#) instancié lors des contrôles des saisies formulaires.

Ex :

```
$validTable = array(
    'id'      =>array( 'Reference' =>'Identifiant', 'required' => false ),
    'quote'   =>array( 'Reference' =>'Intro', 'required'   => true,
    'min'      => 3, 'max'         => 2000 ), .....etc.....
```

Managers

Les manager gèrent toutes les sélections de la base de données.



L'objet **Manager** centralise les attributs et méthodes communes aux différents manager :

`__construct()` – récupère les instances des objets **DB** et **Session**.

`setEntity($tableName)` – récupère la liste des zones table via `$this->_entity->getFfd($this->_tab)`

`majTab($class)` – centralise les appels aux méthodes de création, mise à jour et suppression de la base de donnée

- **Book...Episode...** et **CommentManager** ont été définis comme des extension de l'objet **Manager**.

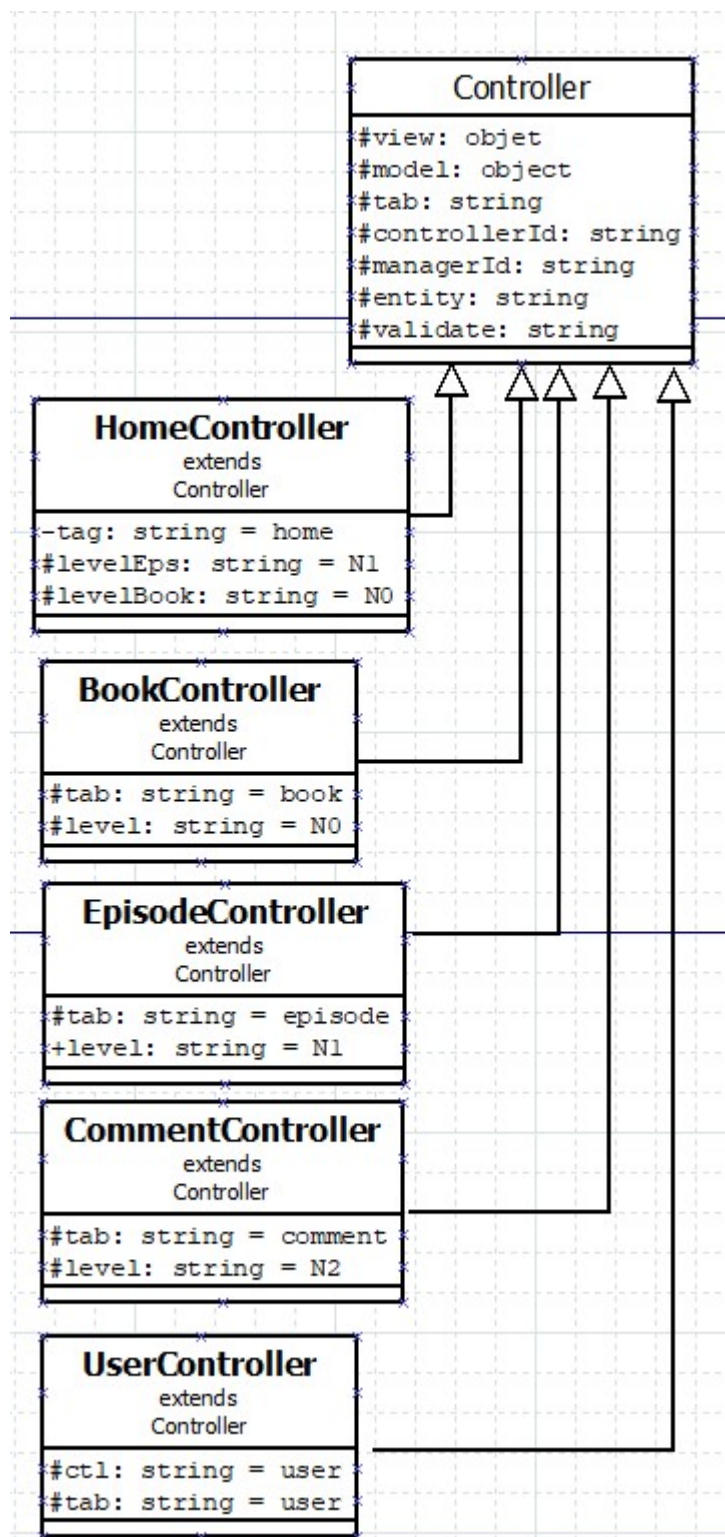
Leurs méthodes gèrent les différents cas de sélection et de tri pour affichage. Ils appellent des méthodes de l'instance **DB** en lui transmettant divers paramètres requis ou facultatifs (segments where, order by etc...)

`FormatSelection($level)` retourne le tableau d'objet correspondant à la sélection.

Suivant le niveau initial de la demande, les objets sont enrichis d'informations consolidés à partir des niveaux inférieurs et supérieurs.

L'objet **BookManager** par ex. s'adjoint une instance de **EpisodeManager** pour enrichir les objets **Book** retournés et actualiser des informations de consolidation (nombre et statut maximum des épisodes, date de 1ere mise en ligne etc...)

- **UserManager** aurait dû également étendre **Manager**. En plus des méthodes précédentes, il utilise des méthodes statiques de l'objet **Session**.
- **HomeManager** a été géré à part, car il instancie et utilise les méthodes de **BookManager** et **EpisodeManager**



L'objet **Controller** centralise les attributs et méthodes communes aux différents contrôleurs :

Il instancie le manager et la vue ; centralise les contrôles associés aux mises à jour de la Base de données (Validate) et gère des redirections.

Les contrôleurs dédiés gèrent des méthodes spécifiques par type de demande :

Index, Edit, show, list etc.... Ces types de demande correspondent aux méthodes appelées par le routeur et le front controller.

Ils gèrent également le niveau de demande initial, et des attributs spécifiques de la vue.

Ils peuvent enchaîner les appels de méthodes ex. **CommentController**, où la gestion des signalement requiert un complément d'information avant enchaînement sur la méthode de mise à jour centralisée par le Controller principal.

Le **HomeController** : pilote à la fois

la page d'Accueil,

la page AboutJFR,

et les informations d'en tête et pied de site présentes sur toutes les pages (publication à l'affiche, liste des autres publications).

Focus : formulaires et gestion PHP

Pour envoyer les données des formulaires on a privilégié la méthode POST et l'index comme page cible du formulaire (action) ➔ le passage obligé par le routeur assure le contrôle des accès et droits sur les path. -notamment les droits Administrateurs si création/modification ou suppression des bases book et episode.

Les variables des formulaires portent les noms des zones correspondantes de la base de données.

D'autres variables « input » transmettent option et url de destination pour gestion du chemin par le routeur.

Présence et validité de tous les paramètres

3 niveaux de contrôle

Contrôle via le formulaire (taille zone, requis...)

Contrôle attaché au niveau des modèles dans les entités. Une classe **Validate** centralise la méthode de contrôle et renvoie un `top isValid` avec l'objet correspondant à la table à mettre à jour dans la base de données.

Contrôle via l'encapsulation : `Htmlspecialchars` est appliqués aux infos « commentaire », « pseudo » et « utilisateur » saisies par le visiteur – par le setter de l'entité **Comment**.

NB Les saisies de l'auteur ne sont pas échappées (saisie tinymce)

Cas particulier :

La gestion des commentaires par l'auteur ne transite pas par un formulaire. Seuls le chemin, l'option et l'identifiant du poste à mettre à jour sont envoyés.

Pour normaliser le contrôle des données de la classe **Validate**, la méthode (`inzPost` du manager) permet de compléter `$_POST` à partir de l'objet existant en base de données.

La gestion du formulaire de signalement a été aligné sur le même schéma de fonctionnement.

Focus : gestion des requêtes SQL

Dans le contexte de mise en œuvre d'une programmation orientée objet, les requêtes SQL sont adressées au travers de l'objet PDO -interface portable vers autres SGBD.

Pour mettre en œuvre les sélections dans la base, mais également les mises à jour, l'application s'appuie sur un objet **DB** Database. C'est cet objet qui gère les instances de PDO et la crée initialement (__construct) avec les paramètres stockés dans l'objet **Config**.

DB
<pre>*-instance: object *-pdo: object *-query: string *+error: bool *-results: mixed *-count: int *#dspffd: array *+opt: string = select</pre>

Méthodes/

getInstance() : renvoie l'instance existante ou la crée.

Query() : appelle la méthode pdo->prepare, query->execute puis fetchAll(PDO::FETCH_ASSOC)

addClsRcd(\$table, \$class)

dltClsRcd(\$table, \$class)

updClsRcd(\$table, \$class)

gstFfd(\$table, \$class) : gère la construction dynamique des sql Add et Update.

action(\$action, \$table, \$join = null, \$where = array(), \$orderBy= null
gère la construction dynamique des sql Select.

Les managers de l'application jfrblog **Manager UserManager, BookManager** etc... utilisent tous les méthodes publiques de l'instance géré par la fonction statique getInstance de l'objet **DB**

```
$this->_selection = DB::getInstance()->get($this->_tab, $ksel, $orderBy, $action, $join);
```

```
$succes = DB::getInstance()->updClsRcd($this->_tab, $class);
```

Et récupèrent en général un tableau d'enregistrements qui est reformatté en tableau d'objets avant d'être retourné au contrôleur demandeur.

Bilan

Le bilan du projet mené et la synthèse des réalisations finalisées oblige à confronter ses choix aux solutions et alternatives multiples rencontrées par ailleurs. On se découvre forcément des lacunes.

Ces quelques mois de POO, PHP, MVC vous laissent malgré tout sidéré par l'ampleur du domaine et des connaissances à acquérir, et nombre de celles qu'on a croisé en chemin restent à tester ou à mettre en pratique à la première occasion.

axes d'amélioration

Saisie et restitution des textes : l'Editeur en ligne

Installer une version TinyMce qui permette vraiment de faire une mise en page. Actuellement la solution de facilité est de saisir ailleurs et de ramener le texte par copier/coller.

Fonctionnels

Concerne	Descriptif
Gestion des publications	Contrôler unicité du livre mis en avant ; Permettre les bascules de statut sans entrer en modification des postes
écran	Prévoir des ancres -
Mentions légales	Prévoir liens et Politique de confidentialité suivant niveau de gestion utilisateurs et présence O/N de Newsletter. Intégrer la page de mention légale dans la base de données -à gérer comme la page About (top ML).
Gestion utilisateur	Contrôler l'unicité du pseudo à l'inscription. Formulaire de gestion des données personnelles. protection des données perso/ cryptage MDP

Applicatif et techniques

Concerne	Descriptif
La remontée des informations de consolidation	Un système de trigger devrait permettrait la remontée et la mise à jour automatique des informations au niveau supérieur. (comment => episode ; episode -> book) Quid des pattern Observer ?
codage	Normaliser la gestion des redirections et la gestion des erreurs.
Gestion DB	Creuser les modalités d'application DAO