

**DNET**

Scrabble

*Rapport*

Debecker Bernard

Foncier Romain

2012 - 2013

# Introduction

Dans le cadre du cours de *Développement InterNET* dispensé en dernière année de bachelier en informatique, il a été proposé aux étudiants de développer une application web de *location de biens de vacances* à l'aide du Framework Java *Spring*.

Le développement est à réaliser par groupe de 2 à 3 personnes. Le notre est composé des membres suivants : *Bernard Debecker* et *Romain Foncier*.

Ce rapport a pour but de présenter notre travail. Il comportera trois parties. La première décrira le travail d'analyse mis en place avant et pendant le développement. Les deux parties suivantes décriront respectivement les structures établies côté serveur, puis côté client, ainsi que les techniques et technologies mises en place de chaque côté.

# Analyse

Le développement d'un site web s'articule de manière générale autour de deux pôles principaux : serveur et client. Ces derniers se calquent sur le *cycle requête-réponse* à la base de l'architecture client/serveur d'une application web. Le client envoie une requête au serveur qui lui renvoie une réponse sous forme de données formattées en HTML, texte, JSON, XML, ...

Notre analyse a donc pris en considération cette interaction entre chaque partie, afin de très vite définir les informations qui allaient devoir être échangées, leur type et le contexte au sein duquel elles pourraient être consultées et modifiées.

L'application concerne un site de location de biens. Plusieurs questions ont donc été posées :

* Quelles informations définissent un bien ?
* Comment les utilisateurs peuvent-ils les consulter ?
* Quels vont être les différentes sections du site ?
* Quelles informations vont être liées à l'utilisateur et comment ?
* ...

Nous allons commencer la présentation de notre analyse par le modèle de données, puis nous nous intéresserons au design et aux cas d'utilisations.

## Modèle de données

Notre application propose un service de location de biens. Elle s'articule donc essentiellement autour de trois environnements principales :

### User space

L'environnement *User* est composé d'une seule table : *User*.

***User***

Cette entité est importante et indispensable au fonctionnement général de l'application. Elle est composée des attributs suivants :

* **username** : Unique sur la plate-forme.
* **first\_name**
* **last\_name**
* **email**
* **password**
* **created** : Date de création du compte.
* **is\_staff** : Booléen spécifiant si l'utilisateur a accès à l'interface d'administration.

*User* est une entité clé. Une *property* ne peut être créée sans utilisateur. Si aucune propriété n'est définie, l'application ne peut fonctionner correctement. Ainsi, les utilisateurs de la plate-forme pourront créer un compte, se logguer, y ajouter un ou plusieurs biens en location et y réserver des biens. Mais principalement, un utilisateur enregistré pourra participer à l'activité du site au travers du système de commentaires et de notations, et ainsi interagir avec d'autres utilisateurs de l'application.

### Property space

L'environnement *Property* est également fondamental à notre site web. On y retrouve toutes les entités gravitant autour de *Property* qui viennent enrichir celle-ci et offrent les fonctionnalités d'interaction avec les utilisateurs. Il est composé des tables suivantes :

***Property***

Cette entité représente un bien. Afin de définir quels attributs doit posséder une propriété, il est important d'anticiper les informations qui seront affichées sur la plate-forme. Un utilisateur consultant la description d'une *property* devra y trouver les données suivantes :

* **owner :** Propriétaire du bien (Foreign Key vers l'entité *user*)
* **title**
* **short\_desc** : Courte description affichée dans les résulats de recherche.
* **long\_desc** : Description détaillée du bien fournit par le propriétaire.
* **price**
* **type** : Type du bien : FLAT, LOFT, HOUSE, ...
* **rooms**
* **country**
* **city**
* **rent\_period\_start** : Date de début de la période de location du bien.
* **rent\_period\_stop** : Date de fin de la période de location du bien.

Un certain nombre de champs s'ajoute à cette liste venant y apporter de nombreuses informations complémentaires utiles à l'application et à certaines fonctionnalités :

* **added** : Date de création du bien.
* **modifed** : Date de dernière modification du bien.
* **coordinates** : Latitude et longitude du bien permettant d'utilisé *l'API Google Maps*.
* **note** : Moyenne des appréciations postées par les utilisateurs ayant loué le bien.

Il est important d'apporter quelques précisions concernant certain de ces attributs. Par simplification, nous considérons par défaut qu'une chambre (*room*) est disponible pour deux personnes. Un bien possédant deux chambres sera considéré par l'application comme pouvant proposer une location pour quatre personnes.

Afin d'apporter plus d'interactivité à notre site, nous avions dès le départ envisagé d'utiliser l'*API Google Maps*. Il nous est de cette manière possible de proposer une localisation d'un bien lors de la consultation de celui-ci, mais également lors de la phase d'ajout. Le stockage des *coordonnées géographiques* (*lat/lng*) permet d'en faciliter l'utilisation et offre également des possibilités de recherches précises ou sur des zones de taille donnée. Ces éléments seront détaillés dans la section consacrée au système de recherche.

D'autre part un système de notation a été mis en place. Basé sur les critères d'appréciations suivants :

* **cleanliness** : Etat de propreté général du bien
* **confort** : Confort de la literie, des installations, ...
* **quality/price** : Rapport qualité-prix

Le processus de notation a lieu une fois le séjour passé. L'utilisateur à l'origine de la réservation sera invité à fournir ses données. Une moyenne de l'ensemble de ces critères pour un bien permettra d'en définir la *note globale*.

***Property Options***

Cette entité vient compléter *property* en y ajoutant un certain nombre d'options. Celles-ci ont été ajoutées dans la table *property\_options*, offrant plus de souplesse quant à la possibilité d'en ajouter de nouvelles. Elle est composée des attributs suivants :

* **target\_property** : Id de l'entité property cible.
* **parking**
* **swimming\_pool**
* **wifi**
* **laundry**

Toutes les options sont des booléens. Ces informations sont donc faciles à récupérer et grouper afin d'offrir un affichage des données pertinents. Par exemple, il est aisé d'afficher dans les résultats d'une recherche, le nombre de chacunes des options dans la liste de *property* trouvées.

***Evaluations***

Cette entité permet de stocker pour une property la moyenne de chacun des critères définis. Elle est composée des éléments suivants :

* **target\_property**
* **cleanliness**
* **confort**
* **qa\_price** : Rapport qualité/prix.

Chacun des critères stocké sous la forme d'une valeur entière comprise entre 0 et 100. Comme nous l'avons précisé précédemment, le calcul de la *note globale* d'une *property* s'effectue sur base des *moyennes de chaque critères*. Cette approche est triviale est présente l'avantage d'offrir une évaluation honnête d'un propriété.

Toutefois, il aurait pu être intéressant d'envisager une évaluation plus avancée en ajoutant un élément supplementaire au calcul : un *poids pour chacun des critères*. Ce dernier aurait pu être défini de sorte à maximiser certain critère. Par exemple, si le critère *confort* est le plus souvent le meilleur, son poids pourrait dans ce cas être évalué à une valeur supérieure à 33% (1 critère / 3). On pourrait alors avoir un poids de 40% pour le confort et 30% pour les deux autres. Cette approche, bien qu'intéressante, n'a pas été implémentée pour ce projet.

***Commentaires***

Cette entité permet d'ajouter un *aspect social* à l'application en proposant aux utilisateurs de partager leur avis et expériences sur les biens. Nous avons choisi de mettre en place un système de commentaires très simple ne stockant que les messages postés et basé sur une structure linéaire. Il n'est alors pas possible de créer des fils de conversations imbriqués. Cette table est composée des éléments suivants :

* **creator** : Utilisateur à l'origine du message.
* **target\_property**
* **date**
* **modified** : L'utilisateur pourra s'il le désire modifier un commentaire qu'il aura composé.
* **message** : Corps du commentaire.

A la fin de son séjour, l'utilisateur sera invité à laissé son appréciation. Celle-ci n'est pas obligatoire, mais encouragée. Elle permet aux utilisateurs intéressés par un bien, de pouvoir plus aisément se faire un avis sur celui-ci. Ces avis n'excluant pas un manque d'objectivité de la part de certain utilisateurs. D'autre part, tout commentaire ne respectant pas la chartre d'utilisation de la plate-forme se verra supprimé sans délais par les administrateurs.

***Pictures***

Cette entité permet de stocker les photos des property. Elle avait été envisagée dans un premier temps avant d'être remplacée par un système de fichier plus simple et rapide. Aucune information supplémentaire n'est alors stockée dans la base de données. Les fichiers images sont alors enregistrés dans un dossier dont le nom correspond à l'id du bien. Il est alors aisé de retrouver toutes les images d'une propriété et de les renvoyer au client lorsqu'ils sont demandés.

### Reservation space

Nous introduisons ici une nouvelle entité : *Reservation*. Celle permet aux utilisateurs de l'application de réserver une property sur une période donnée, pour un nombre de personnes défini et pour un montant fixé.

***Reservation***

Cette entité est composée des éléments suivants :

* **target\_user**
* **target\_property**
* **date\_rent\_start** : Date de début de la location.
* **date\_rent\_stop** : Date de fin de la location.
* **hosts** : Nombre de personnes pour cette réservation.
* **price** : Prix fixé pour le séjour.
* **evaluated** : Booléen précisant si la réservation a été évaluée.
* **commented** : Booléen précisant si un commentaire a été laissé pour cette réservation.
* **note** : Note globale du séjour.

La *note globale du séjour* est calculée sur base des trois critères rencontrés au sein de l'entité *evaluation*. Celle-ci n'a qu'un intérêt limité. Toutefois, elle sera affichée avec le commentaire de l'utilisateur ayant effectué le séjour et renseignera les autres utilisateurs de la plate-forme du résultat de l'évaluation faite par ce dernier pour le bien concerné.

*Note : Les id de ces entités sont tous de type Integer, à l'exception de l'entité User pour laquelle l'id est stocké sous forme d'une String représentant la valeur hexadécimale d'un UUID (Universaly Unique IDentifier).*

### Schéma du modèle

Le diagramme de la **Figure 1** présente l'ensemble des entités de notre modèle et offre une visualisation précise des interractions que celles-ci entretiennent entre-elles.

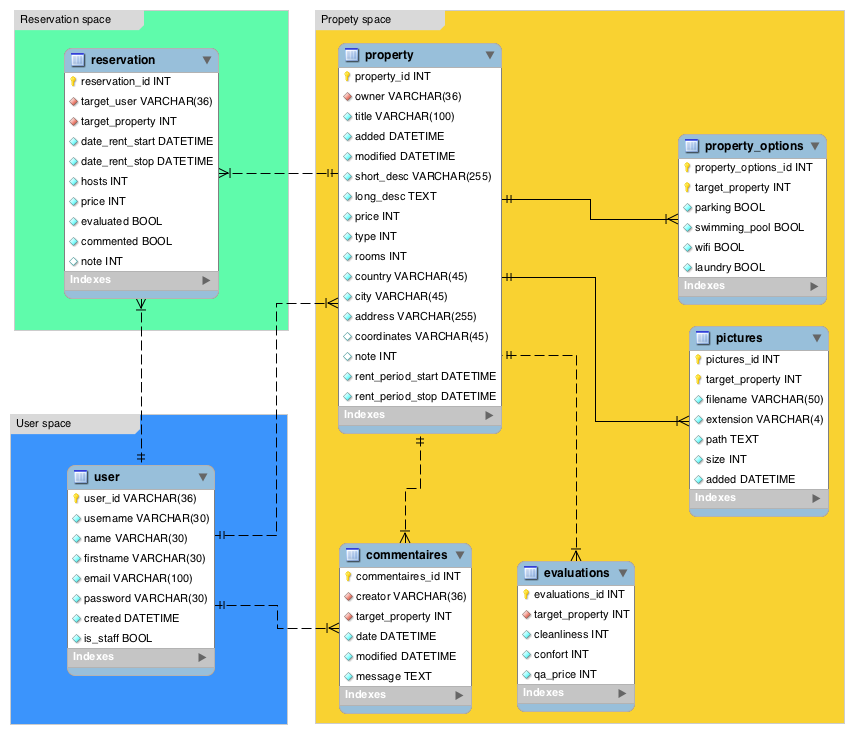


Figure 1 - Modèle de données

## Spring Framework

L'utilisation du *framework Spring* était une des prérogatives du projet. Il s'agit d'un framework libre (*licence Apache*), développé en 2004, sous l'initiative de *Rod Johnson*. Il offre une interface de construction et structuration d'un projet web en *Java J2EE*.

Il s'appuie principalement sur l'intégration de trois concepts majeurs:

### La recherche de dépendances et l'injection de dépendances. Il s'agit de design patterns dérivés d'un patron de conception très répendu dans les frameworks, *Inversion of Control*. Le concept est simple, le framework prend le contrôle de l'éxécution de l'application.

Il orchestre le flot d'opérations ayant lieu au sein de l'application.

### La programmation orientée aspect. Paradigme de programmation permettant d'améliorer les échanges entre les modules de l'application (gestion des utilisateurs, archivage des données, ...) et la logique métier. La séparation fonctionnelle des modules permet un développement et une maintenance plus aisée. Au sein de *Spring*, l'injection de dépendances va permettre de renforcer l'interraction entre ceux-ci et d'en faciliter le contrôle par le framework.

### Couche d'abstraction. Elle permet de faciliter l'intégration d'autres modules ou librairies au framework

### Composition de Spring

*Spring* se repose sur deux éléments fondamentaux :

* Une *fabrique* de *beans* (*JavaBeans*). Il s'agit de classes *Java*, représentant le plus souvent les entités clés du modèle, respectant certaines conventions sur le nommage, la construction (constructeur sans paramètre) et le comportement des méthodes (*getters* et *setters*).
* Un conteneur pour les *beans*.

Les beans sont définis de manière déclarative au sein de fichiers de configurations XML.

<bean id="templateResolver" class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">

<property name="prefix" value="/WEB-INF/templates/" />

<property name="templateMode" value="HTML5" />

</bean>

<bean id="templateEngine" class="org.thymeleaf.spring3.SpringTemplateEngine">

<property name="templateResolver" ref="templateResolver" />

<property name="additionalDialects">

<set>

<bean class="org.thymeleaf.extras.tiles2.dialect.TilesDialect"/>

<bean

class="org.thymeleaf.extras.springsecurity3.dialect.SpringSecurityDialect"/>

<bean

class="org.thymeleaf.extras.conditionalcomments.dialect.ConditionalCommentsDialect"/>

</set>

</property>

</bean>

Le snippet précédent illustre la manière dont les beans sont déclarées et la manière dont elles peuvent être injectées au sein d'un autre.

La classe *ServletContextTemplateResolver* est déclarée au sein de notre application par la bean *templateResolver*. Des propriétés spécifiques à celle-ci y sont définies au sein de balises *<property>*.

D'autre part, la classe *SpringTemplateEngine* appartenant au package *org.thymeleaf.spring3* est déclarée au sein de la bean *templateEngine*. On peut d'ailleurs remarquer que notre précédente bean *templateResolver* y est injecté via la propriété :

<property name="templateResolver" ref="templateResolver" />

Cette intégration est l'illustration de l'*injection de dépendances* permettant de contrôler l'appel de certains éléments et leurs imbrications.

Il s'agit ici d'objets appartenant à notre système de templates : *Thymeleaf*. Ce dernier est un framework *Java* permettant de remplacer les traditionnels fichiers *.jsp* par de véritables fichiers HTML au sein desquels seront injectées des données. Ce dernier sera présenté plus en détails dans les sections suivantes.

### Structure d'un projet WEB

Comme tout framework, *Spring* impose un structure particulière du projet reposant sur les bases de développement d'un projet WEB J2EE. L'arborescence de fichiers est donc répartie au sein de trois dossiers principaux :

* **/java/** : Contenant toutes les classes Java.
* **/resources/** : Contenant tous les fichiers de ressources. Il s'agit de l'ensemble des fichiers de configurations XML de l'application (*Spring, Hibernate*). Il peut également contenir des fichiers média ou encore des fichiers de traductions.
* **/webapp/** : Contenant tout les fichiers statiques de l'application (*CSS, Javascript*, templates) ainsi que le fichier de configuration du server *web.xml*.

## Data Access Object

La mise en place de la DAO est l'étape faisant suite au travail d'analyse du modèle de données. Il s'agira de créer les objets permettant à toutes les entités de l'application d'interragir avec la base de données.

### Classes DAO

Les objets d'accès aux données concerneront toutes les entités qui interragiront avec la logique métier de notre application. Elles ont été présentées au sein du modèle de données. Il s'agit des classes suivantes :

### Hibernate

### Classes Service

## Spring Security

# Developpement