SC Project

Master ALMA 2eme année

Rapport de projet

Encadrants : M.Oussalah & S.Thibaudeau

N.Boukra
L.Duringer
A.Marguerite
M.Ouairy
N.Sebary

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

Table des matières

Remerciements

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO

1 Introduction

1.1 L'entreprise Obeo

Obeo est une société de service et un éditeur de logiciels [obe]. Son expertise dans le domaine de l'ingénierie des modèles (démarche MDA) lui permet de proposer des solutions allant de la création à la refonte d'applications informatiques.

Ces solutions proposées permettent notamment de diminuer les délais de projets et de diminuer les risques d'erreurs. Les amélioration des performances d'adaptation et d'agilité font aussi parties des objectifs des outils et méthode de la société *Obeo*.

La société *Obeo* est aussi un membre actif dans le domaine Open Source et est membre de la fondation Eclipse. Elle est à l'initiative du projet Acceleo (*cf.* [acc]), un générateur de code basé sur le framework EMF.

1.2 Acceleo et démarche M2T

Acceleo est un projet Open Source de la fondation Eclipse dont *Obeo* est à l'origine. À partir de modèles basé sur le framework EMF (*cf.* [emf]), Acceleo permet de générer du code en mettant en œuvre l'approche Model driven architecture (MDA). Le générateur Acceleo est une implémentation de la norme de l'Object Management Group [omg] pour les transformations de modèle vers texte (Model to Text : M2T).

2 Objectifs et démarches

Dans le cadre de l'élaboration de notre modèle, trois méta-modèles ont été mis à notre disposition par *Obeo*. Dans ce chapitre, nous détaillerons le rôle de ces trois méta-modèles.

2.1 Objectifs

Dans le cadre du module *SC Project*, nous avons eu l'opportunité de travailler sur la conception d'outils de modélisation dédiés aux applications web. En partant de trois méta modèles mis à disposition par Obeo (*cf.* section 4) l'objectif était de créer des modèles et les générateurs de code associé pour produire le code d'une application. L'utilisateur final aura ainsi la possibilité de créer le modèle de son application web (à partir d'une vue en arbre par exemple) puis d'exécuter le générateur mis à disposition pour produire le code de l'application désirée.

2.2 Démarche

Les première semaines du projet furent consacrées à la prise en main des outils et des technologies. S.Thibaudeau nous a mis à disposition des exemples de générateurs, et les références vers les différentes initiations indispensables pour ce projet (Tutoriel Acceleo, ObeoDesigner ...). En parallèle nous avons aussi étudié le framework *Play!* et vérifié la compatibilité avec l'objectif de ce projet. Dans un second temps, nous avons entamé la conception d'un prototype de génération d'une application web avec *Play!*. Enfin, nous avons étudié les différents méta-modèles mis à disposition puis entamé la création des différents générateurs de code associés. Dans ce rapport nous détaillerons ces différents étapes du projet. Ainsi dans le chapitre 3 résume l'étude menée sur le *Play!* et le prototype crée. Dans le chapitre 4 nous reviendrons sur les différents méta-modèles ainsi que les générateurs de code associés.

3 Technologie cible et outils

Le premier objectif proposé a été d'étudier le framework *Play!*et de s'assurer de la compatibilité de sa philosophie avec un projet de génération de code. Dans la section 3.1 de ce chapitre, nous détaillerons les qualités de ce framework qui ont motivées la sélection de *Play!*pour ce projet. Dans un second temps (*cf.* section 3.2, nous aborderons le prototype que nous avons élaboré avec *Play!*.

3.1 Le Framework Play

Play! est un CMS (Content Management System) web basé sur les langages Java et Scala permettant de la création d'applications web. De plus en plus de développeurs choisissent cet outil, qui est relativement récent, et qui présente de nombreux avantages. L'aspect « prêt à l'emploi » (*Plug'n Play*) permis par ses fonctionnalités par défaut le rendent efficace et rapide à mettre en place et à configurer. Le développement également est facilité, d'une part par par des mécanismes de compilation à la volée (*shadow-build*) lors du chargement des pages, mais aussi par la mise à disposition de systèmes tests intégrées (JUnit, Selenium). Sa gestion des requêtes Web peut être bloquante ou non-bloquante (synchronisme). La génération des pages Web renvoyées peut être dynamisée grâce à un mécanismes de templating basé sur Scala. Enfin son architecture modulaire, composée de plugins et du *design pattern* MVC (Modèle-Vue-Contrôleur), permettent une répartition claire du code source ce qui est propice à la démarche MDA.

FIGURE 3.1 – Framework Play !

Le Framwork *Play!* propose de diviser l'implémentation d'un site web en trois composantes principales :

3.1.1 Les pages web

Pour la gestion de ses pages web, *Play!* propose une approche basée sur les templates. Les pages peuvent donc être écrites avec du HTML classique, et être enrichies avec du code *Scala* pour générer du contenu dynamiquement. Ainsi, il est possible d'insérer des clauses conditionnelles (if/else) ou des boucles (for/while) à l'intérieur du code HTML. Afin de conserver une architecture cohérente, il est possible de passer un certain nombre d'arguments/objets en paramètre de ces templates, afin que les pages puissent en afficher le contenu de manière formatée.

3.1.2 Les Données

La gestion des données dans *Play!* est laissée au choix de l'utilisateur - le développeur. Cependant, *Play!* embarque nativement l'ORM *Ebean*. Le principe d'un ORM (Object-Relational Mapping) est de fournir une couche d'abstraction dessus d'une Base de Données relationnelle. Cette couche d'abstraction doit être suffisante pour que les différents éléments de la base soient manipulables directement en tant qu'Objets. Cela permet aux développeurs de s'affranchir des contraintes techniques que peuvent présenter les Bases de Données relationnelles.
Dans *Play!*, avec *Ebean*, une classe Java pourra être gérée comme étant une entité/table, et ses attributs et références seront traitées comme des colonnes de cette table. (!!! Hésitez pas à annoter les points où c'est pas clair, car c'est un peu capilotracté des fois!!!)

3.1.3 Les Contrôleurs et les Routes

Dans *Play!* le/les contrôleur(s) joue(nt) un rôle central au sein l'application. Ce sont les contrôleurs qui se chargent de récupérer les requêtes des visiteurs (de type HTTP), d'effectuer les traitements (ajout d'un cookie, traitement dans la Base de Données, ...), et d'effectuer le rendu des pages web retournées au visiteur.
Dans le monde du Web, les requêtes se présentent la plupart du temps sous forme d'adresse dite URL (Uniform Resource Locator) appelée par le visiteur. Comme beaucoup d'autres CMS, *Play!* propose un fichier de Routes dans sa configuration. Le rôle d'un fichier de Routes est de convertir/résoudre les adresses/URL afin de les faire correspondre à une méthode du Contrôleur de l'application.

Ainsi, lorsqu'une requête arrive à l'application, *Play!* regarde d'abord dans son fichier de Routes, puis appelle la méthode correspondante, qui retournera une réponse/page vers l'émetteur de la requête.

3.2 Conception d'une maquette d'exemple

Nous nous sommes proposés de bâtir un mini-site Web basé sur *Play!*. L'ojectif est double :
- Nous familiariser avec le framework *Play!* et comprendre son fonctionnement.
- Obtenir une maquette qui sera utilisée comme objectif de code à générer.

Le prototype que nous avons proposé est une implémentation simpliée d'un site marchand. Ce type de site web est en effet très courant, et il implique différents aspects et fonctionnalités : Stockage persistant de données (produits vendus), utilisation de formulaires (inscription des clients), utilisation de sessions (authentification des clients sur le site, et achats).

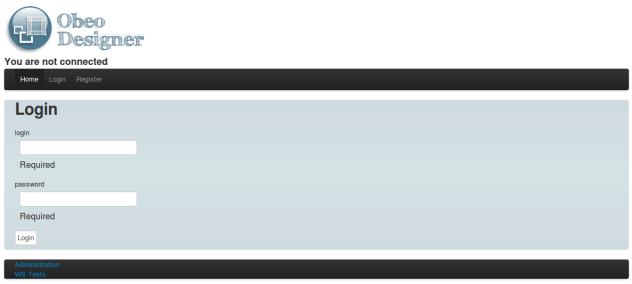

FIGURE 3.2 – Prototype Play_Shop

4 Travaux Effectués

4.1 Entity

Le métamodèle Entity représente une couche « métier ». Il permet notamment de modéliser la persistance des données :

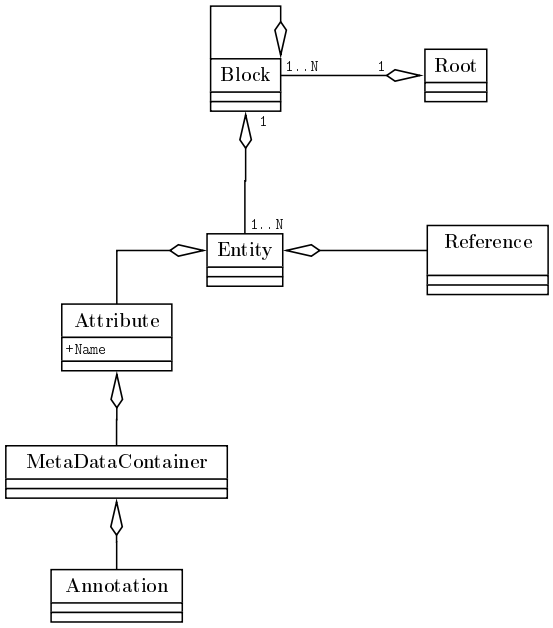

FIGURE 4.1 – Métamodèle Entity

4.2 *SOA*

SOA(Service Oriented Architecture) - ou Architecture Orientée Services - modélise le concept de Services et d'Opérations au travers d'un système basé sur les Composants. Concrêtement, un modèle basé sur *SOA*est constitué de Composants. Chaque Composant peut possèder des Services dont des Interfaces permettent de communiquer avec l'extérieur. Ces Interfaces définissent une une plusieurs Opérations, chaque opération étant caractérisée par un ensemble de Paramètres d'entrée et de sortie. Ces paramètres peuvent être des types relativement basiques (Integer, String, ...) mais également des « Entity » complexes, comme des informations complètes sur les utilisateurs, ou sur des produits.

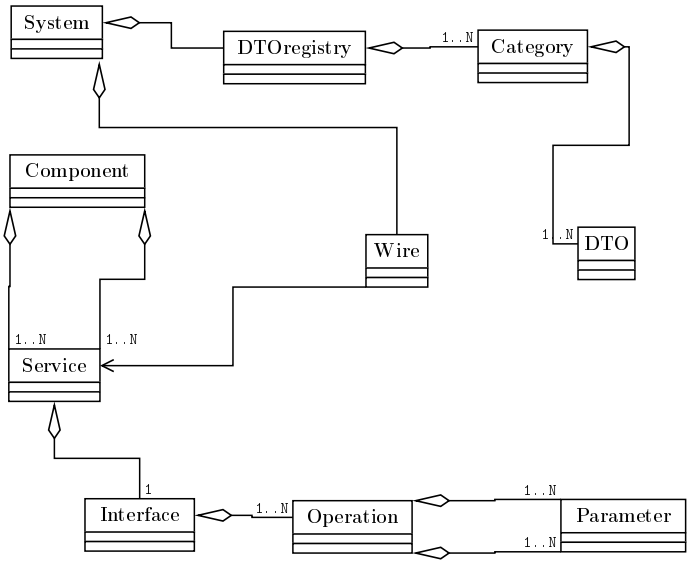

FIGURE 4.2 – Métamodèle SOA

4.2.1 Le concept de *SOA* dans *Play!*

Play!, de part son achitecture en MVC, ne propose pas d'implémentation concrête de la notion de Services. Nous avons donc fait le choix de déporter ces derniers dans des classes externes de type Singleton, ce qui permet de faire appel aux différents Services depuis n'importe quel Controleur de *Play!*. (*Services.getAKindOfService().do_service()*). Cela permet de séparer les différents concepts sans pour autant remettre en cause l'architecture originelle de *Play!*.

4.2.2 Conception du modèle

La conception du modèle SOA pour le contexte de notre magasin en ligne a été assez simple à convenir : Pour chaque cas d'utilisation ou action, nous avons modélisé un certain nombre d'opérations basiques. Ces opérations ont été réunies dans différents Composants et Services, en fonction de leur contexte, mais également en fonction du type d'utilisateur susceptible de faire appel à ces opérations.
Ainsi, les opérations consistant à inscrire ou connecter un utilisateur ont été regroupées au sein d'un *AuthentificationManager* accessible uniquement par l'application en elle-même.

4.2.3 Génération du code des services

Séparation interfaces/implémentations

Afin d'assurer une souplesse de l'application, la génération des différents Services systématiquement découpée en deux parties :

- Des interfaces, contenant les signatures des différentes opérations.
- Des classes d'implémentation, liées à ces interfaces.

Les classes d'implémentation sont générées avec le code des différentes opérations, lorsque celles-ci sont basiques et/ou facilement interprêtables (récupérer/éditer/détruire une entité). Dans tous les cas, des balises « user code » (à définir) sont également insérées afin de laisser libre choix à l'utilisateur pour les détails de l'implémentation.
Cette séparation interfaces/implémentations permet également d'avoir plusieurs types d'implémentation de Services pour une même interface.
Nous avons implémenté les différentes classes de Services comme étant des Singleton. Il est ainsi possible d'appeler un Service depuis n'importe quel endroit de l'application avec une syntaxe du type *Services.getMonPremierService().faireMonTravail()*.

Plus loin avec SOA : Les webservices

Ils nous a été proposé d'ajouter des couches supplémentaires par dessus les simples implémentations de Services SOA, notamment au niveau des moyens d'accéder à ces Services.
Nous avons donc mis en place des solutions permettant d'appeler certains Services depuis l'extérieur, sans passer par l'interface du site Web. Pour ce faire, nous avons procédé à la mise en place d'un service de type REST (REpresentational State Transfer).
Le principe est de recevoir des requêtes HTTP de type GET/POST/PUT/DELETE provenant d'autres programme que des simples navigateurs.
Par exemple, il devrait être possible d'appeler certains services depuis un logiciel de gestion, pour la gestion des ventes des produits, des factures, ou des comptes utilisateurs, ou pour collecter des informations.

Nous avons donc implémenté un tel système, en configurant - pour chaque opération accessible depuis l'extérieur - des Routes spécifiques, déclenchant l'exécution des opération, et retournant un résultat au format *JSON*.

4.3 Cinématique

4.3.1 Modélisation

Vue globale du métamodèle

Le métamodèle est organisé autour de trois principaux packages :

- View : représente les concepts liés à la définition des écrans IHM.
- Toolkit : représente les concepts liés à la définition des widgets[1] IHM.
- Flow : permet d'identifier le comportement dynamique des écrans IHM. Le flow peut être appréhendé comme une sorte de diagramme d'activités.

1. Elément visuel d'une interface graphique (bouton, ascenseur, liste déroulante, etc.)

5 Conclusion

5.1 Problèmes persistant

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO

5.2 Bilan

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO

Table des figures

Bibliographie

[acc] Projet Acceleo. http://www.acceleo.org/pages/accueil/fr.

[emf] Framework EMF. http://www.eclipse.org/modeling/emf/.

[obe] Societe Obeo. http://www.obeo.fr/.

[omg] Object Management Group. http://www.omg.org.