Rapport de projet de Service
Réalisation d'une application de boutique marque blanche en Domain Driven Design

Lenny Lucas et Thomas Minier
Master 2 ALMA 2016/2017

2 Décembre 2016

Table des matières

1 Introduction

Le Domain Driven Design, ou Conception pilotée par le domaine en français, est une approche de développement logiciel développée par Eric Evans en 2004 [1]. Elle a pour but de placer le métier au coeur du développement, dans une industrie où c'est plus souvent le développement qui pilote le métier au lieu de l'inverse. Cela permet de créer une synergie créative entre les experts du métier et ceux du développement.

Le Domain Driven Design sépare un logiciel en quatre couches : l'infrastructure, le domaine, l'application et la présentation. Chacune d'entre elles gère une partie précise du logiciel : le métier est isolé dans la couche domaine, ses fonctionnalités spécifiques sont implémentées dans l'infrastructure, l'application connecte les deux et la présentation gère l'affichage global. Nous reviendrons dans les détails de chaque couche plus loin.

Dans ce projet, il nous a été demandé de réaliser une application de gestion de boutique en marque blanche selon le Domain Driven Design. Il s'accompagne également d'un fournisseur, qui sera développé par un autre groupe et auquel nous nous connecterons. Nous avons développé notre application en Java 1.8, avec Maven comme système de build. Le projet est géré sous Git, hébergé sur Github [2], et nous utilisons également Sonarqube comme serveur de revue de code, afin de calculer la couverture de code automatiquement à chaque commit et l'afficher en ligne.

Dans ce rapport, nous détaillerons notre spécification et implémentation des différentes couches. Nous parlerons également de nos choix au niveau du déploiement et des services tiers utilisés par notre application.

1. Evans, E. (2004). Domain-driven design : tackling complexity in the heart of software. Addison-Wesley Professional.
2. Dépôt de notre projet : https ://github.com/XyLnEn/ServiceBoutique

2 Couche API

Cette couche, non prévue par Eric Evans dans sa conception initiale du Domain Driven Design, a été rajouté afin de permettre un niveau d'abstraction supplémentaire entre les différentes couches du logiciel. Elle contient les interfaces implémentées par l'infrastructure et qui sont injectées dans le domaine par l'application.

Notre API va donc contenir trois types d'interfaces, toutes issus des concepts décrits par Eric Evans. La première est celle des **fabriques**. Les objets du domaine sont généralement assez vastes et leur instanciation peut être trop complexe et fastidieuse. De plus, elle dépend parfois de l'infrastructure, et ne peut donc pas être gérée par le domaine, car ce processus de création ne dépend pas toujours du métier. Les fabriques ont donc pour rôle de gérer ce processus de création et de l'encapsuler.

Le deuxième type d'interface est celui des entrepôts. Il s'agit d'objets dont le rôle est de gérer l'archivage, la mise à jour et la suppression des objets du système. Ils sont souvent liés aux modèles dits CRUD (Create Read Update Delete) ou BREAD (Browse Read Edit Add Delete). Ils permettent de découpler le métier de la gestion du cycle de vie des objets.

Enfin, le dernier type d'interface contenu dans l'API est celle des services. Ils représentent tous les actions du système qui sont indépendantes et que l'on ne peut pas lier au métier, aux fabriques ou aux entrepôts. Habituellement, il s'agit des services, voir de webservices, externes à l'application.

La figure 1 montre l'agencement de nos différentes couches entre elles. Les flèches représentent les dépendances, physiques ou logiques, entre les couches.

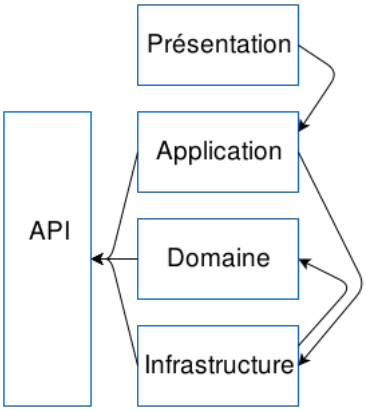

FIGURE 1 – Agencement des couches du logiciel entre elles

3 Couche Domaine

La couche domaine a un objectif précis : définir les terminologies, les fonctionnalités et les besoins pour représenter un métier. Elle a pour but de résoudre les problèmes du métier : dans le cas d'une boutique par exemple, comment acheter, vendre des produits et comment sauvegarder ces transactions. Il s'agit du coeur du programme.

3.1 Analyse et implémentation

Une boutique a pour but d'acheter et de vendre des produits. On doit donc avoir un concept de Produit. De plus, ces produits doivent être achetés par un Client. Il faut aussi que cette boutique ce restocke chez un Fournisseur. La boutique doit aussi garder un historique des Transactions avec les Clients et Fournisseurs afin de pouvoir analyser l'état des finances de la boutique ainsi que pour détecter des problèmes éventuels sur le stock. Enfin, lorsque la boutique effectue une Transaction, elle le fait sur un groupe de produits que nous appellerons Order.

D'un point de vue implémentation, nous avons séparés les classes en 2 grands types : les entités et les objets-valeurs. Les objets-valeurs sont immuables : lorsqu'ils sont instanciés, leur valeur ne changera plus jamais. Ils ne sont que des conteneurs à données sans comportement. En revanche, les entités sont des instances uniques ; ces objets gardent une identité propre durant toute la durée de vie du programme et même au-delà. Il est important de leur donner un identifiant unique qui permet de le reconnaître car il est parfaitement possible d'avoir 2 instances d'une de ces entités avec les mêmes attributs : il est important de les différencier. Toutes ces entités sont des Javabeans pour qu'ils soient sérialisables afin de pouvoir les stocker et les extraire d'une base de données.

Nous avons fait le choix de rassembler les Fournisseurs et les Clients dans une seule classe : ThirdPerson. En effet, ces 2 concepts sont très semblables : il s'agit d'une entité avec un nom, une adresse et un numéro de téléphone. Elle aura aussi un historique des Orders qu'elle a fait afin d'avoir un historique local qui peut nous apporter des métadonnées sur ces entités. Enfin le stock d'un fournisseur n'est pas lié à celui-ci : la méthode avec laquelle l'atteindre peut différer au cours du temps et donc elle ne doit pas être dans le domaine. Enfin nous avons fait le choix de représenter le gérant du magasin par un ThirdParty pour plus de simplicité.

La représentation des produits est très simple : un nom, une description, un prix et une catégorie. Étant donné qu'il s'agit de la base de toute la boutique, il est important que ce concept soit simple. Nous avons fait le choix d'avoir des instances de produits au lieu d'une relation Produit-quantité car cela permet d'avoir un contrôle plus fin sur ces Produits tout en représentant plus fidèlement

un stock de boutique en ligne.

Les Orders sont simplement des listes de produits. Nous avons fait ce choix car cela permet de déplacer les objets achetés dans une liste annexe pour pouvoir les supprimer dans la base de données sans les perdre. Ces Orders sont aussi composés du nom de l'entreprise chargée de la délivrer et de l'état de la commande. En effet, la boutique que nous devions créer étant une boutique en ligne, nous avons choisi de représenter les différentes étapes de la livraison.

Enfin, les Transactions sont simplement des relations entre les id uniques du ThirdParty, du gérant du magasin et de l'Order qui a été effectué. Cette entité nous permet de représenter un achat, que ce soit d'un client ou du magasin vers un fournisseur.

Le domaine existe aussi pour résoudre les problèmes du métier en utilisant ces concepts. Pour ce faire, nous exposons des services via la classe Shop du domaine. Elle est capable, entre autres, d'effectuer un achat et d'approvisionner le stock du magasin.

Le domaine n'est pas responsable de la création et de la gestion de ces concepts, car il ne fait que les définir et les utiliser. Il est donc nécessaire de lui injecter des conteneurs de données et des créateurs de données afin que cette exigence soit déplacée vers l'extérieur du domaine. C'est dans cette optique que nous utilisons les classes IFactory et IReporitory de l'api.

3.2 Librairies utilisées

Le domaine ne définissant que les termes du métier ainsi que les services pour manipuler ces concepts, il n'utilise que peu de librairies externes. La grande majorité des classes du domaine étant des JavaBeans, il y a beaucoup de getters et setters à tester ainsi que des tests de services pour monter le coverage des tests dans SonarQube.

— La première librairie utilisée ici est **OpenPojo**, une bibliothèque de tests unitaires qui permet les tests automatiques de tous les getters et setters d'une classe.
— La dernière librairie utilisée ici est **AssertJ** [3]. Cette librairie permet de faire des tests unitaires de manière plus agréable et d'avoir des messages clairs lors de leurs échecs.

3. AssertJ : http ://joel-costigliola.github.io/assertj/

4 Couche Infrastructure

La couche infrastructure a pour objectif d'implémenter les interfaces de l'API utilisées par le domaine. Elle permet de fournir une implémentation à l'application qui peut varier à l'exécution. Par exemple, on pourrait changer la base de données pour passer d'un système SQL vers NoSQL.

4.1 Analyse et implémentation

Nous devons implémenter de multiples services venant de notre API : les fabriques, les entrepôts et les services.

Les entrepôts seront implémentés au travers d'une base MongoDB. Nous avons choisi cette base de données car nous sommes dans un cas où nous allons effectuer beaucoup de lectures et d'insertion, mais peu de mise à jour. Dans ce contexte, une base NoSQL est mieux adaptée qu'une base SQL, car cette dernière repose sur des protocoles de validation atomique qui ralentirait notre application dans des conditions réelles d'utilisations. De plus, une base MongoDB effectue son stockage au format JSON, et nos objets étant prévus pour se sérialiser vers du JSON (et inversement), le stockage de nos objets en sera plus simple.

Concernant les fabriques, elles sont toutes implémentées selon le pattern Factory Method. A part une, elles se contentent toutes d'instancier des objets de manière classique. Seule la fabrique des objets venant du fournisseur effectue un traitement plus complexe, car elle va faire un appel au webservice du fournisseur. Cette dernière est au format JSON, car c'est le format de données qui a été choisi par les différents fournisseurs avec lesquels nous avons testé notre logiciel.

Pour les services, ils consultent tous des services externes :

— Le service qui permet de valider les cartes bancaires consulte un webservice SOAP que nous avons créé pour les besoins de ce projet. Nous en discuterons plus loin dans le rapport.
— Le service qui permet de consulter les produits du catalogue du fournisseur fonctionne de la même manière que la fabrique dont nous avons précédemment parlé.
— Le service qui effectue la conversion des devises selon un taux de change utilise un webservice JSON externe : Fixer.io [4].

Pour utiliser ses deux types de webservices (SOAP et JSON) et consulter la base de données, nous avons préféré développer des façades pour les librairies de décodage, afin de pouvoir les utiliser de manière plus transparente et plus découplée. Ainsi, si nous décidons de changer les librairies utilisées dans

4. Fixer.io : http ://fixer.io/

l'infrastructure, il nous suffira d'adapter les façades sans avoir à toucher à l'implémentation des services, entrepôts et fabriques.

4.2 Librairies utilisées

Dans cette couche, nous avons utilisés les librairies suivantes :

— **Jackson** [5] : cette librairie permet de sérialiser des objets aux normes JavaBean au format JSON, et inversement. Elle nous a été utile car elle permet de gérer ses opérations de manière transparente, sans avoir à réinventer un décodeur JSON, ce qui aurait été une tâche fastidieuse.

— **Le driver Java MongoDB** [6] : cette librairie est obligatoire pour communiquer avec une base MongoDB depuis un programme Java. Son inclusion a donc été naturelle dans notre projet.

— **OpenPojo** et **AssertJ** : comme dit précédemment pour la couche domaine, ces librairies nous ont été utiles pour réaliser les tests unitaires de la couche infrastructure.

5. Jackson : https ://github.com/FasterXML/jackson-databind
6. Driver java pour MongoDB : https ://mongodb.github.io/mongo-java-driver/

5 Couche Application

La couche Application a un rôle précis : être le chef d'orchestre du reste du programme. Elle utilise l'infrastructure pour accéder aux dépôts de données et pour les créateurs d'instances afin de les injecter dans le domaine. Elle expose aussi les méthodes fonctionnelles et non fonctionnelles à une présentation via une API JSON.

5.1 Analyse et implémentation

Cette couche utilise 2 concepts majeurs de l'infrastructure : les IRepository, et les IFactory. Elle les fournit au domaine lorsque nécessaire, tout en instanciant les IFactory avec les bonnes valeurs. C'est pour cela que nous utilisons des injections lors de l'instanciation des contrôleurs. Les contrôleurs, quant à eux, servent à exposer les services du domaine et les méthodes non fonctionnelles via des API JSON.
Il est important de noter que ces API peuvent servir de marque blanche : il est possible d'accéder à tous les produits et d'acheter un produit via ces API, exactement comme pour la présentation.

5.2 Librairies utilisées

Cette couche utilise 3 librairies :

— **SparkJava** [7], une librairie qui permet d'exposer des API sur des URLS. Nous avons fait le choix d'utiliser cette librairie au lieu de librairies plus communes (telle que Spring) car elle est très légère et extrêmement simple à utiliser. Elle correspond exactement à nos besoins sans en faire trop.
— **OpenPojo** encore une fois pour les tests unitaires automatisés.
— **Jackson** pour les mêmes raisons que précédemment dans la partie Infrastructure.

7. Spark Java : http ://sparkjava.com/

6 Couche Présentation

La couche présentation a pour objectif de fournir une interface aux utilisateurs pour interagir avec la couche application.

6.1 Analyse et implémentation

Étant donné que notre couche application s'expose comme une API JSON, nous avons choisi de développer la couche présentation comme une webapplication selon le modèle Frontend-Backend : elle va communiquer avec l'application via des requêtes HTTP pour récupérer les données dont elle a besoin et effectuer la plupart des interactions.

Notre présentation est donc physiquement indépendante de l'application, car elle n'entretient qu'une dépendance purement logique avec elle. Pour mettre l'accent sur cela, nous avons décidé de la séparer du reste des couches du logiciel et de la déployer sur une autre machine. La figure
2 illustre la répartition du logiciel d'un point de vue serveur.

Actuellement, notre présentation utilise les fonctionnalités suivantes de l'application :

— Consultation et achats des produits de la boutique.
— Consultation et achats des produits du fournisseur.
— Consultation de l'historique des transactions de la boutique.

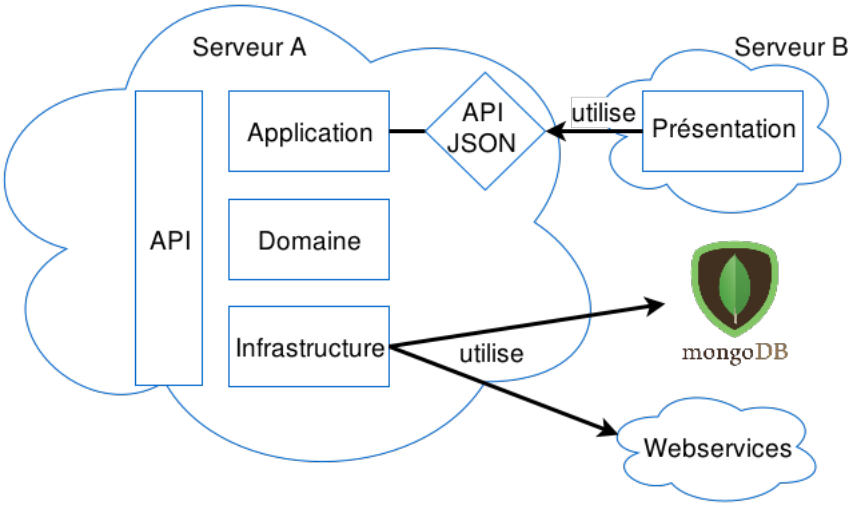

FIGURE 2 – Répartition du logiciel d'un point de vue serveur

6.2 Librairies utilisées

Dans cette couche, nous avons utilisé les librairies suivantes :

— **Spark Java** : pour les mêmes raisons qui nous ont poussé à choisir cette librairie pour l'application, Spark Java nous permet de déployer simplement un servlet qui sert une webappplication Javascript et qui peut par la suite être déployé sur n'importe quel type de serveur (Tomcat, Jetty, Glassfish, ...).

— **Vue.js**[8] : cette librairie permet de créer des webapplications Javascript sur le modèle Frontend-Backend. Nous l'avons choisi, plutôt qu'une librairie comme AngularJS ou React, car elle ne gère que la partie rendue de l'application, ce qui nous suffit amplement dans le cadre de la couche présentation.

— **Axios**[9] : cette libraire Javascript permet de gérer la partie échanges HTTP entre le Frontend et le Backend. Nous l'avons choisi car elle convient exactement à nos besoins et qu'elle est très légère.

7 Service de validation de cartes bancaires

Étant donné que nous avions besoin d'un service de validation de cartes bancaires, et que les spécifications du projet nous imposaient d'utiliser au moins un webservice en SOAP, nous avons décidé de mêler les deux en créant notre propre système de validation sous la forme d'un webservice SOAP. Pour ce faire, nous avons développé un servlet Java classique qui reçoit un numéro de carte bancaire, effectue une vérification simple dessus, puis retourne le résultat sous la forme d'une enveloppe SOAP. Ce servlet peut ainsi être facilement déployé sur n'importe quel type de serveur. dans notre cas, nous le déployons sur un Tomcat.

8. Vue.js : https ://vuejs.org/
9. Axios : https ://www.npmjs.com/package/axios

8 Conclusion

En conclusion, ce projet a été pour nous une occasion de mettre en oeuvre le Domain Driven Design dans le cadre d'un véritable projet. Nous avons pu constater que cette technique de développement présente des avantages certains : en séparant le logiciel en couches, il est aisé de répartir le développement des différentes couches entre les développeurs de l'équipe. De plus, tant que le contrat entre les différentes couches est correctement défini, le développement peut se faire sans friction et bien plus rapidement. C'est d'ailleurs de cette manière que nous avons travaillée durant le projet, et nous avons pu constater l'efficacité de cette méthode de travail.

Néanmoins, nous avons aussi pu constater que le Domain Driven Design n'est pas forcément adapté dans le cadre d'un projet de petite envergure, car il ajoute beaucoup de complexité au niveau logiciel. Il est donc plus à recommander dans le cas où le métier est vaste et complexe et qu'il y a un vrai besoin d'isolation entre les différentes couches.