POLYTECH©
NICE-SOPHIA

INTRODUCTION À L'ARCHITECTURE LOGICIELLE

RAPPORT FINAL

Vendredi 20 mai 2016
Enseignant responsable : Sébastien MOSSER

Groupe AA
Nicolas Hory
Lucas Martinez
Lucas Soumille

Introduction

Avec l'essor des smartphones et des technologies toujours plus innovantes, la société d'exploitation des remontées mécaniques Isola 3000 a décidé de moderniser son système d'information. Pour cela, elle a lancé un appel d'offre indiquant dans les grandes lignes, les nouvelles fonctionnalités attendues.

Tout au long de ce rapport, nous allons présenter l'architecture de notre plateforme. Nous commencerons par comparer l'architecture "finale" par rapport à celle prévue initialement. Puis nous présenterons les modifications faites depuis la démonstration. Enfin nous analyserons l'impact de l'ajout de la couche de persistance et les futures évolutions de notre système.

Description de notre implémentation

D'un point de vue extérieur, l'ensemble du système est composé de trois entités. La première est la partie application cliente qui permet d'interagir avec le serveur d'Isola 3000, la deuxième est un serveur implémenté en .NET qui permet de simuler le fonctionnement de services externes (paiement, notification, etc...), enfin la partie centrale de l'application, implémentée en J2E, rassemble toutes les fonctionnalités nécessaires au bon fonctionnement de la station telles que l'achat de forfaits et la gestion de l'accès aux remontées mécaniques.

Cette dernière partie s'articule autour d'une architecture 3-tiers composée des couches suivantes :

- **Présentation** qui expose les fonctionnalités aux clients qui peuvent les consommer. Ici, nous avons utilisé à la fois les webservices (SOAP) pour interagir et des pages XHTML (JSF) pour visualiser les charges des remontées. Elle est responsable de l'interaction avec les utilisateurs
- **Domaine** qui implémente les fonctionnalités métiers de notre application. Cette couche est consommée par la couche "Présentation". Nous avons utilisé les EJB pour l'implémenter.
- **Persistance** qui gère le stockage des informations dans la base de données. Cette couche est appelée par la couche "Domaine" dès qu'elle souhaite interroger ou modifier la base. Ici, nous nous sommes servis de JPA.

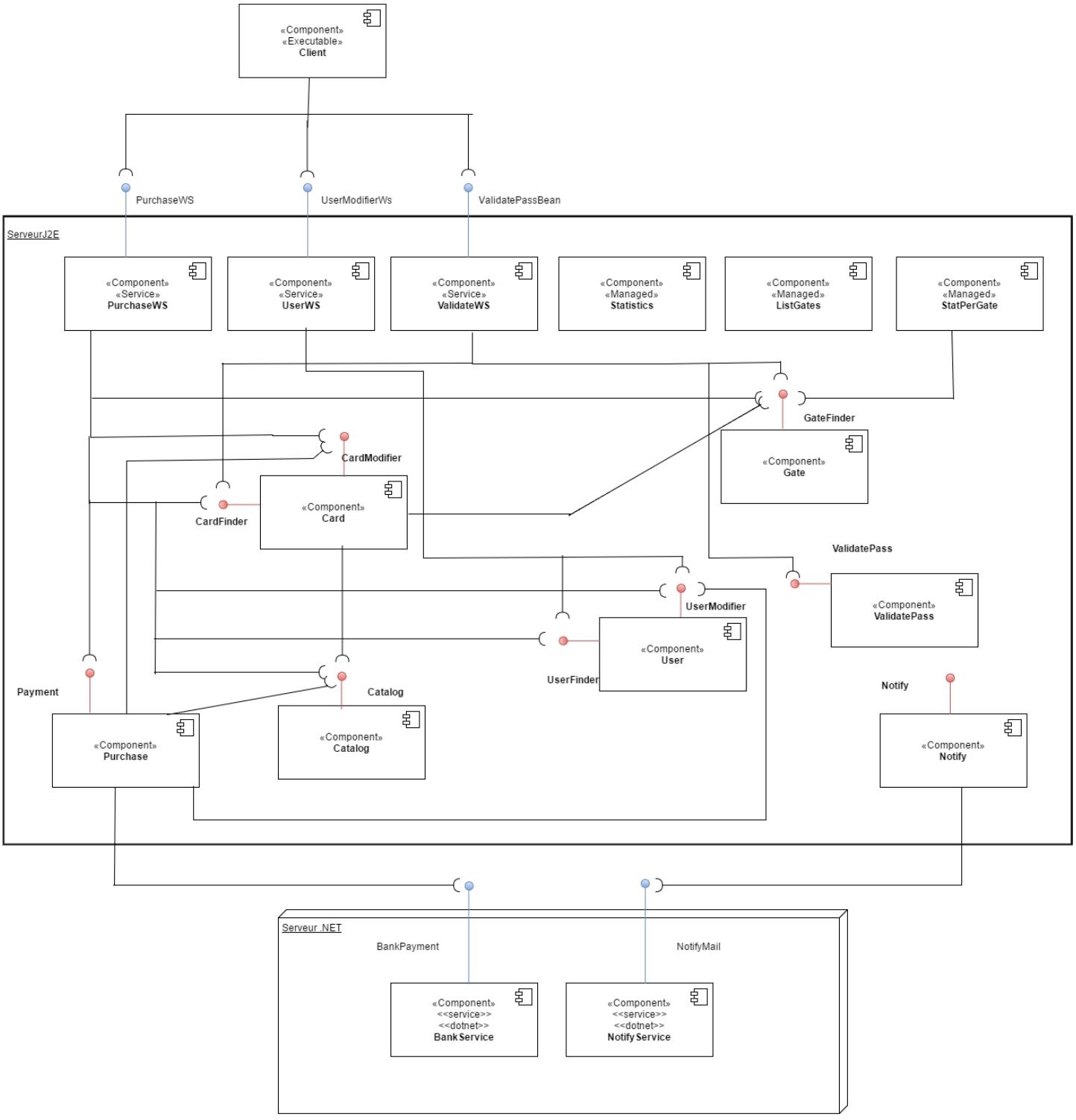

Figure 1: Diagramme de composants

Description des interfaces

Interfaces Web Services J2E – Client

PurchaseWS	*addCardForUser(nom, prenom)*	Ajoute une carte à la personne en utilisant ses informations bancaires pour la payer
	addPassForCard(idCarte, age, type, zone)	Ajoute le forfait sur la carte en utilisant les informations bancaires du propriétaire de la carte
	subscribeFidelicime(idCarte)	Transforme la carte en un pass Fidélicime en utilisant les informations bancaires du propriétaire de la carte
	listAllPass()	Retourne le catalogue de forfait
	getCard(numCB)	Achète une carte avec la carte de crédit
	addPassForCardWithCredential(numCB, idCarte, age, type, zone)	Ajoute le forfait sur la carte en utilisant la carte de crédit
UserModifierWS	*register(nom, prenom, age, numCB, mail)*	Ajoute l'utilisateur
	getHistory(nom, prenom)	Retourne les transactions de l'utilisateur
ValidatePassWS	*validatePass(idCarte, idGate)*	Vérifie la validité du forfait

Interfaces Composants

CardFinder	*findById(idCarte)*	Retourne la carte correspondant à l'id
CardModifier	*create(carte)*	Ajoute la carte à la base
	add(carte, forfait)	Ajoute le forfait sur la carte
	remove(carte, forfait)	Supprime le forfait sur la carte
	contents(carte)	Retourne le forfait contenu sur la carte
Catalog	*getAllTypesOfPass()*	Retourne la liste des forfaits disponibles
	getPriceForPass(forfait)	Retourne le prix du forfait
	getPriceCard()	Retourne le prix d'une carte
	getPriceSubFidelicime()	Retourne le prix de la souscription à fidélicime
GateFinder	*findById(idGate)*	Retourne la remontée correspondante
	getGatesByZone(zone)	Retourne les remontées de la zone
Notify	*notifyByAge(objet, message, intervalle)*	Envoie un message aux skieurs dont leur âge est dans l'intervalle
Payment	*payCardForUser(utilisateur, carte)*	Paye la carte et ajoute la transaction à l'utilisateur
	payCard(numCB, carte)	Paye la carte
	payPassForUser(utilisateur, forfait)	Paye le forfait et ajoute la transaction à l'utilisateur
	payPass(numCB, forfait)	Paye le forfait
	paySubFidelicime(utilisateur)	Paye la souscription et ajoute la transaction à l'utilisateur
UserFinder	*getByNameAndFirstName(nom, prenom)*	Retourne l'utilisateur correspondant
	getByIdCard(idCarte)	Retourne l'utilisateur possédant la carte
UserModifier	*add(utilisateur)*	Ajoute l'utilisateur

	update(utilisateur, nouveau)	Met à jour l'utilisateur
	remove(utilisateur)	Supprime l'utilisateur
	addCardForUser(utilisateur, carte)	Ajoute la carte à l'utilisateur
ValidatePass	*validate(carte, remontee)*	Vérifie la validité du forfait

Interfaces Web Services J2E - .NET

PaymentService	*receiveRequest()*	Demande un paiement
	findPaymentById(id)	Retourne le paiement correspondant
	getAllPaymentIds()	Retourne l'ensemble des paiements
NotifyService	*mailRequest()*	Demande l'envoi de mails

Comparaison entre le diagramme initial et final

Avant de commencer le projet, nous avions peu de vision sur l'organisation d'une architecture 3-tiers. De plus, nous n'avions pas encore appréhendé toutes les fonctionnalités demandées par le client. C'est principalement pour ces deux raisons que notre diagramme initial n'est pas au final ce qui a été implémenté. *Vous pouvez voir notre diagramme initial en annexe.*

Il est tout de même intéressant d'analyser les premiers choix que nous avions faits et pourquoi ils n'étaient pas adaptés : le premier constat que l'on peut faire c'est que notre nombre de composants était trop faible. En effet, dans le serveur J2E, il y avait seulement trois composants. En plus, ces composants n'étaient pas orientés vers des fonctionnalités attendues par le client, nous avions fait par exemple un composant qui devait être utilisé pour l'analyse des requêtes. Egalement, nous avions positionné des composants d'affichage à l'extérieur du J2E, alors qu'au contraire ce sont les composants essentiels de notre architecture, en particulier de notre couche présentation. Nous pensions que le serveur .NET devait interagir directement avec les clients notamment pour ce qui est de la notification. Nous n'avions pas compris que la partie implémentée en C# devait représenter des services externes consommés par le serveur J2E et que le client allait se connecter seulement aux web services exposés par le J2E. Enfin, nous ne savions pas que la connexion client-serveur allait être un web service. Nous imaginions plus une connexion bas niveau en utilisant directement les sockets.

C'est en partant de ces hypothèses que nous avions fait le diagramme de composants, ce qui a entraîné les différences présentes entre le diagramme initial et final.

Résolution du point bloquant

Le point bloquant lors de la démonstration était l'ajout initial des valeurs "constantes", c'est à dire le l'ensemble des remontées et des forfaits disponibles, au démarrage du serveur J2E. Lors de la mise en fonction de ce dernier, cet ajout était automatique peu importe que les données étaient déjà présentes ou non. Pour corriger ce problème, nous vérifions avant de lancer la fonction d'ajout dans le postconstruct que les tables de remontées et des prix de la base sont bien vides.

Impact de l'ajout de la persistance

L'utilisation d'une couche de persistance est obligatoire dans la majorité des applications modernes. Dans notre architecture la gestion de la persistance est une partie importante car c'est une couche complète sur laquelle se reposent nos composants. Au début de notre implémentation de la plateforme Isola 3000, la couche de persistance était modélisée par un composant de type singleton où les données étaient perdues à l'extinction. Suite à la démonstration du MVP, il a fallu débouchonner ce composant tout en apportant de nouvelles fonctionnalités.

Pour cela, nous avons fait le choix de diviser notre équipe en deux, pour avoir un membre qui travaille sur les attentes émises par le client lors de la démonstration tandis que les deux autres ont rendu le MVP persistant. Cette dernière tâche a rendu le code du projet un peu plus complexe. Nous avons dû modifier légèrement les objets métiers pour que la modélisation en base soit cohérente, en particulier la gestion de l'héritage des forfaits n'était pas adaptée (mauvaise gestion de l'ensemble des types). De plus l'implémentation des "finders" a dû être refaite, la mécanique de requête n'étant pas forcément intuitive cela nous a pris du temps.

Suite à cela, nous avons fusionné la partie de chacun. Puis quand les nouvelles fonctionnalités ont été persistantes, nous avons fait le choix d'implémenter un cache de lecture pour la vérification des forfaits en bas des remontées dans le but que le passage des skieurs soit le plus fluide et le plus rapide possible. Pour cela dès qu'un utilisateur badge pour la première fois de la journée, nous retenons ses informations étant donné qu'il va repasser de multiples fois. Par contre, dès qu'il y a une modification (achat d'un nouveau forfait, d'une carte, etc…), nous répercutons directement le changement dans la base et dans le cache pour que la perte d'information ne soit pas trop importante en cas d'un arrêt inopiné du serveur.

Évolution du système avec l'ensemble de la grille tarifaire

Notre implémentation permet de gérer les forfaits de type classique comme par exemple les forfaits journée. Comme chaque forfait possède un compteur de journée restante, il est possible de souscrire des forfaits sur des jours non consécutifs. Ainsi lorsqu'un skieur possédant un forfait 8 jours passe à une remontée pour la première fois d'une journée le compteur est décrémenté.

Aussi, notre application gère le Fidéli'cime : lorsqu'un utilisateur achète ce type de forfait, nous lui faisons payer la cotisation puis dès qu'il badge une journée, son solde est augmenté d'un forfait journée. Nous pensions faire tourner un script lors de la fermeture de la station chaque soir qui réalise les encaissements.

Si la société de gestion des remontées décide d'ajouter de nouveaux forfaits, dans notre code il faut ajouter dans l'entrée dans la base, puis modifier l'héritage des forfaits pour ajouter les nouvelles informations nécessaires à ce type puis enfin gérer son traitement lors de son passage aux portiques.

Annexe

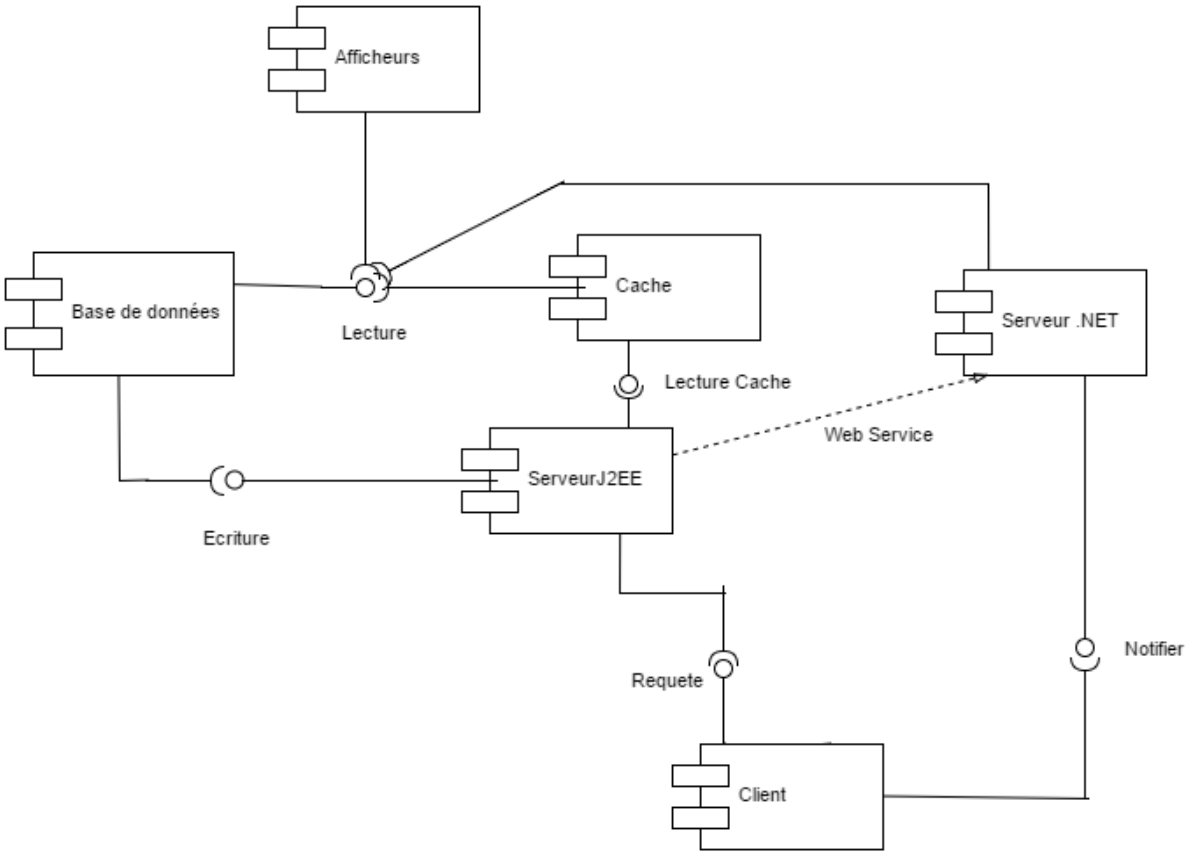

Figure 2: Diagramme de composants initial