# Partie 2

# Travaux effectués

## Sommaire

1. **Etude de mise en œuvre de l’approche Domain Driven disign**

Les travaux réalisés dans cette partie sont les plus intéressants par rapport aux autres tâches qui m’ont été confié pendant le stage. En effet, les conceptions réalisées à la fin de cette partie vont être appliqué sur les différents modules développés par la suite.

* 1. **Objectifs**

Afin d’avoir une démarche unifiée de construction des modules et prendre en compte la complexité croissante dans la future, il faudra adopter une approche de conception de logiciel plus agile, modulaire et simple à évoluer. L’équipe R&D CRM ont mis d’accord d’adopter l’approche Domain Driven Disign à travers le design pattern « CQRS-Event Sourcing » pour répondre aux besoins mentionnées. Le premier travail qui m’a été attribué est de faire la conception à partir d’un module exemplaire en appliquant l’approche DDD en prenant compte différentes types de déploiements.

A fin de viser le plus grand nombre de client du marché, et en se basent sur les expériences de la direction client de coheris, les modules de la nouvelle offre doient être plus flexible en terme de déploiement et de mise en œuvre. En effet, les modes adoptées de mise en œuvre du module étaient :

* Mode 1 : le Frontend et le Backend dans le même serveur
* Mode 2 : le Frontend et le Backend sur deux différents serveurs
* Mode 3 : Backend en mode API REST

Il faudra aussi prendre en compte, d’une part que la solution peut être soit en mode saas soit en mode on-premise, et d’autre part la mise en cluster du module.

Pour que les travaux soient bien cadrés, une spécification fonctionnelle a été réalisée par le Product Manager qui décrit les fonctionnalités attendus du module.

L’objectif de cette première étape d’une durée de 2 semaines est multiple : la documentation de la conception de déférents types de déploiement et bien comprendre l’approche Design Driven Developpment, la mise en œuvre du design pattern « CQRS-Event Sourcing » en respectant les modes de déploiement.

1. **Etude et Réalisation**

L’architecture du module est en multicouche (N-Layered). Elle assure les concepts d’inversion de contrôle et d’injection de dépendances du socle technique JEE de Coheris. En suivant les besoins mentionnées par rapport aux modes de déploiements, le module va être divisé en 7 couches. Chaque couche est encapsuler dans un « .jar » à part.

#### Les différentes Couches du module

L’architecture du module et ses difféentes couches est d´détaillée en annexes (Annexe B, page 80). Il faut cependant savoir que des refontes ont était faite dans la partie présentation sur l’approche DDD pour qu’on puisse répondre aux différentes types de déploiements. En effet, la couche « GUI », qui est basées sur le Framework MVC standard de de la spécification JEE7 « JavaServer Faces 2.2 », exploite les services exposées dans la couche applications soit via un appel direct vers ce dernier ou par un appel à une autre couche « api-client » qui implémente les mêmes fonctionnalités que l’application mais via des appels REST (Fig. 2.4).

Figure

Pour assurer la mise en œuvre du mode saas du module, un composant « module-saas » doit être rattaché au module lors du processus de build qui assure l’architecture multi-tenant du module. Ce composant repose sur le module core-saas qui gére le partitionnement virtuellement des données et la configuration requise.

1. ***Modes de déploiements du module Coheris***

Comme expliqué dans l’objectif de partie des travaux, le déploiement de l’ensemble des ressources du module doit respecter le mode choisie. On a rencontré des difficultés pendant cette phase du à la séparation des encapsulations des différentes couche du module et dans quelle extensions livrable doit-on les mettre.

En effet, un serveur d’application JEE (conteneur Web + conteneur EJB) nous permet de déployer 3 différentes types de livrable « .ear », « .war » ou « .jar ». Le choix était sur un livrable « .war » pour qu’on puisse assurer la mise en œuvre d’une part, la couche *module-gui* comme frontend et d’autre part la couche *module-api* qui expose les fonctionnalités du module via un api REST comme backend. En fin, chaque mode déploiement est définit dans un script de Build Gradle.

Pour le moment et dans le cadre de mon stage, il a été convenu de déployer le en mode 1 et en on-premise.

Pour le moment et dans le cadre de mon stage, il a été convenu de le déployer le en mode 1 et en on-premise.

Figure mode 1

Figure mode 2

Figure mode 3

1. **Bilan**

L’objectif de cette première étape a ´été tenu et a constitué une excellente introduction au sujet principal de mon stage. La conception qui a été réalisé va me servir à organiser le développement selon les différentes couches pour chaque module. En effet, afin de réaliser dans de bonnes conditions les développements des modules avec la nouvelle architecture, il était indispensable que je sois formé sur l’outil de build gradle pour écrire taches de build avec un langage de scripts Groovy.

1. **Module Scheduler**
   1. **Objectifs**

Afin de planifiés des traitements souvent longs dans les différentes modules de la nouvelle offre, on a besoin d’un module Scheduler. C’est un gestionnaire de taches, paramétrable et intégrable selon le besoin. En effet, le module social a besoin de planifié des taches de fond qui collecte les posts et les commentaires Facebook dans les différentes Fanpage pour les qualifié après.

L’objectif de ce travail planifié sur 2 mois est d’une part de maitriser la nouvelle architecture adopté pour les modules et d’autre part le développement d’un module prototype complet qui peut être déployé sur les différents modes.

Avant de commencer les travaux, une étude complète et détaillée doit être réalisée en prenant soin d’effectuer des comparatifs avec les solutions existantes des Framework qui gère les systèmes de planification. Dans cette étude, il faut mettre en valeur les points jugés intéressants et les éventuelles régressions pour chaque solution. De plus, un certain nombre de points techniques doivent être vérifiés.

La mission a été d´découpée en plusieurs tâches. L’étude a été réalisée sur 1 semaines, l’installation et la manipulation la nouvelle architecture a également dure 1 semaines, enfin l’intégration du framework et le développement d’un prototype a été accompli en 2 étapes de 3 semaines.

* 1. **Etude détaillée**

Etude du choix du Framework

### Dans la version Java Entreprise Edition 7, l'API Concurrency Utilities1.0 (JSR 236) de Java propose un système de planification de tâches riches et simple à utiliser. On peut spécifier des annotations pour définir une tache avec l’expression cron []. En revanche, Il existe plusieurs schedulers disponibles pour la plateforme J2EE qui sont plus évolué et qui sont opensource. Quartz est un projet terracota qui propose des composants orientés entreprise JAVA. Mon rôle a donc a été de réaliser une veille technologique afin d’acquérir des informations techniques et d’établir la liste des points forts et faibles pour chaque Solution.

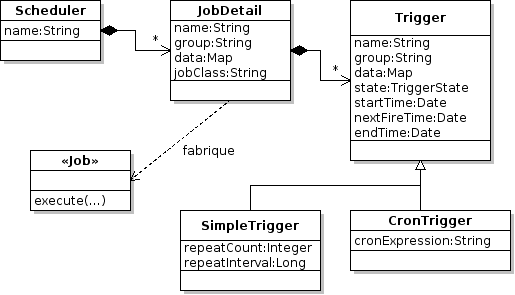
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Framework | documentation | Communauté autour | Clusturing | shema de donnée | configuration | Standard JEE |
| **Concurrency Utilities 1.0** | Peu de documumentation:  documentation standard | Encore nouveau pas beaucoup | Non | non | Pas de configuration | oui |
| Quartz Scheduler | Documumentation riche : exemples, tutoriaux… | Beaucoup plus | OUI | Inclus et gérer automatiquement | Un fichier .properties | Non |

Le choix a été sur Quartz Scheduler vu qu’il gére lui-même sa base de données des taches et il supporte le mode Clusturing qui est une contrainte impérative pour les modules nouvelles offre de Coheris.

A noter, Quartz ne gére pas l’historique des taches qui ont été déjà exécuté mais il fournit un listener qui va nous servir servir à sauvgarder l’historique de chaque éxécution d’une tache.

Architecture

En se basant sur l’architecture type du module d’un côté, et sur le modèle métier du framework Quartz, il nous faudra alors de trouver une approche pour branché l’API Quartz dans cette architecture. En effet, quartz se base sur la structure de donnés suivante :



* **Trigger:** le *quand exécuter*, ça décrit une ou série de moments, il y a plusieurs variantes: Cron, périodique, one shot...
* **Job:** le *quoi exécuter*, c'est un bout de code
* **Job Detail:** l'instance de Job, elle a un ou plusieurs triggers et éventuellement des paramètres
* **Job Group:** un ensemble de Job Details ou de Triggers que l'on manipule ensemble: arrêt, démarrage, annulation...
* **Scheduler:** le moteur chargé de la planification et de l'exécution des tâches planifiées.

On constate que le modèle de données quartz nous permet une manipulation libre et flexible du framework. Selon les spécifications fonctionnelles, le module doit nous permettre de

* planifier des taches : quotidienne, hebdomadaire, mensuelle et annuelle.
* on peut définir un traitement qui pourra être exécuté par plusieurs taches planifiées.
* on peut consulter l’historique de chaque tache exécuté et son état d’exécution.

Cela ramène à définir 5 domains de fonctionnalités dans le module

* **JobInfo :** l'entité qui contient les informations nécessaire pour créer un job
* **TriggerInfo :** l'entité qui collecte les informations nécessaire pour créer un Trigger quartz
* **SchedulerManager:** l'entitité qui représente l'entité scheduler du quartz
* **ExecHistory :** l'entité qui contienne les informations catchées du listener du quartz sur les exécutions
* **Process :** l'entité qui représente les infos nécessaire pour le traitement à exécuter

Chaque modèle a son propre service (couche application et api), son propre repository(couche Infrastructure), ses propres commandes et queries et enfin ces composants IHM(couche gui).

image

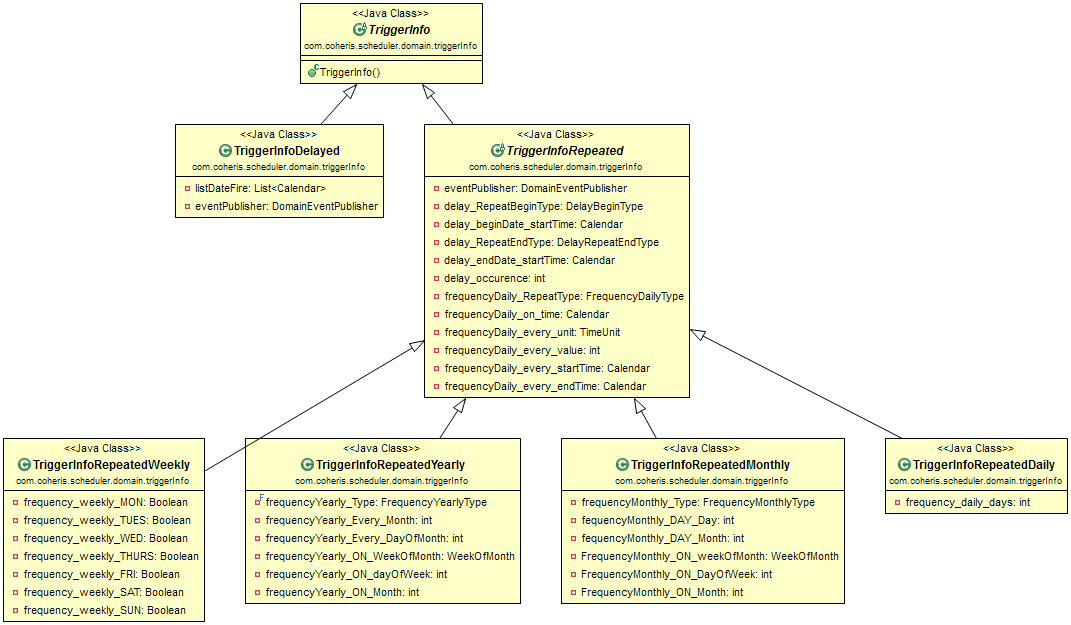
Ce qui concerne le branchement du framework dans le module, la meilleure approche était de mettre ses fonctionnalités, dans la couche infrastructure du module. D’abord, car suivant l’approche DDD la couche infrastructure est le composants qui assure le dialecte avec des ressources externe exp : base de données, l’exploitation d’un api rest ou soap externe. Alors, on peut considérer quartz comme étant une ressource externe du module en traduisant les modèles de la couche domain en modèle de données Quartz.

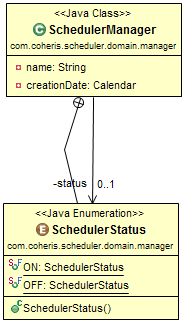
Finalement, il m’a été demandé aussi d’externalisée la configuration du framework d’une façon programmatique à partir des paramètres définie dans le fichier standalone.xml du serveur.

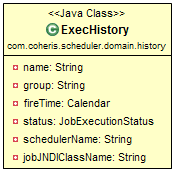
* 1. **Réalisation**

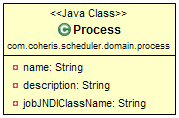
Les réalisations relatives à cette mission correspondent en grande partie à brancher le framework Quartz dans le module scheduler et développer les classes java nécessaires en se basant sur un module exemple [voir lien] qui applique l’approche DDD avec CQRS event sourcing. Dans une autre partie des travaux, j’ai été mené à réaliser un prototype IHM qui expose les fonctionnalités nécessaire du module.











Les composants IHM

Enfin, pour tester le prototype du module, il faut à réaliser une Interface Homme Machine pour qu’on puisse visualiser les taches. J’ai été mené à désigner des maquettes, qui ont été validé par le Product Manager, qui exploite les différentes fonctionnalités développer dans le module. Il m’a été demandé de se baser d’une part sur le Framework Bootstrap pour que les composant soit résponsives, et d’autre part sur Primefaces qui propose une bibliothèque riche de composnant qui se base sur JSF facile à intégrer.

Tableau de bord : où on peut trouver d’une part, les taches en cours d’exécution et son état de chaque d’autre part une vue global sur l’état du scheduler par des statistiques faite par un diagramme.

Gestionnaire de taches : C’est dans cet vue on peut ajouter des taches selon un formulaire dynamique. Et Aussi on trouve plus des détails sur chaque tache.

Gestionnaire des traitements : C’est dans cet vue on peut ajouter des traitements selon un formulaire dynamique.

Gestionnaire d’Historiques : où se trouve l’historique de tous les taches qui ont été exécuter

Conclusion

* 1. **Bilan**

1. **Module Social et l’Intégration SPAD RealTime**
   1. **Objectifs**
   2. **Etude de l’existant**
   3. **Réalisation**
   4. **Bilan**

Découplage entre le couche domain qui contient et infrastructure par la mise en œuvre du pattern Event Sourcing. En effet, la couche application est la chef d’orchestre entre les autres couches. Elle recoit les commande ou les queries.

Un diagramme qui explique les dépendances entre le module core et n modules après

Application : C’est la couche qui expose les services exploitable du module. Ces services sont des implémentations des interfaces définies dans la partie shared-kernel. L’interaction se fait soit par des commandes pour un traitement d’écriture, soit par des queries pour les traitements de lecture via un composant appelé finder. Ce dernier, n’est pas besoin de traversé les autres couches pour appliquer son traitement de lecture. Il cherche directement ses résultats par des requêtes criteria sur le modèle de données via l’ORM JPA. En fin, et pour que les autres couche peuvent se communiquer entre eux via des événements, des listeners sont mis en place pour Handler les évènements qui circule dans le module.

Domain : c’est le centre du module. Cette couche comporte toutes les classes correspondant aux éléments du modèle du domaine. Il encapsule les entités et le logique métier du module. Aussi. Afin que la couche infrastructure implémente les fonctions nécessaires on trouve les définitions des repositories qui sont

Infrastructure

La couche api ou les contrôleurs (managed bean) des composants JSF de la couche Gui.

Shared-kernel

Api : Une API Restful pour la partie API de l’architecture doit être mise en place mis en place pour pouvoir exploiter le module comme étant une API service.

Api-client :

Gui :