Réalisation d'une application répartie sous Docker

Sources

Composant serveur : https://github.com/charleshuber/4trest

Composant client : https://github.com/charleshuber/4tfront

Architecture docker : https://github.com/charleshuber/4tswarms

Liens

Projet final : http://www.surfthevoid.com

Plateforme d'intégration continue : http://www.surfthevoid.com:8090

(**login** : cyril / **mot de passe** : smb215)

Introduction

L'objectif du projet est de construire les briques communes à une grande quantité d'applications réparties, afin de fournir une base de travail pour de futures réalisations.

Il existe des similitudes entre de nombreuses applications aux niveaux :

- de l'architecture applicative
- de la mise en place d'une intégration continue
- des technologies utilisées (JEE / JavaScript)

Le projet décrira comment ces trois aspects peuvent être factorisés dans l'optique de construire un environnement de développement solide et réutilisable, qui puisse me servir pour mon projet de mémoire d'ingénieur. Bien que toutes les fonctionnalités communes a de nombreuses applications n'ont pas pu être réalisées par manque de temps, l'ensemble des technologies employées a été validé par des exemples simples. Par exemple, le protocole https n'a pas été mis en place au niveau de l'architecture technique et la gestion de l'authentification n'a pas encore été implémentée au niveau applicatif.

Dans les sources du projet on retrouvera souvent le terme « **4t** ». Ce nom est destiné à l'application de gestion des congés qui sera réalisée dans le cadre de mon mémoire d'ingénieur.

Mise en place de l' architecture applicative

Présentation

La totalité des applications que j'ai eu a réaliser avaient 2 composants. Une base de donnée et une application web responsable à la fois des processus métiers et de la génération des pages HTML pour le client.

La volonté du projet est d'utiliser la technologie des conteneurs et celle des micro-services REST (representational state transfer) pour réaliser une véritable architecture MVC (Modèle / Vue / Contrôleur) en désolidarisant le contrôle des données de leur présentation au client. Cela permet de décharger le serveur métier de la génération des pages HTML et de pouvoir changer la présentation des données sans modifier le contrôleur.

Notre architecture sera donc basée sur trois services, qui seront chacun construit à partir d' une image docker différente :

- Une image MYSQL responsable de la persistance des données récupérée directement depuis le registre Docker public (https://hub.docker.com/_/mysql/).
- Une image Tomcat 8 que nous devons construire nous même car elle doit contenir l'exécutable de notre application. Le fichier « Dockerfile » contenant les instructions qui permettent de construire l'image, est joint avec les sources du composant serveur (https://github.com/charleshuber/4trest). Les sources contiennent également tous les autres fichiers nécessaires à la construction de l'image.
- Une image Apache que nous devons construire nous même car elle doit contenir les fichiers de notre application cliente. Le fichier « Dockerfile » contenant les instructions qui permettent de construire l'image, est joint avec les sources du composant client (https://github.com/charleshuber/4tfront). Les sources contiennent également tous les autres fichiers nécessaires à la construction de l'image.

Pour associer ces trois services en vue de construire l'application répartie, nous utilisons un outil différent en fonction de la configuration :

- La configuration locale, utilise « docker-compose », car il est possible de déployer les sources en cours de développement directement sur les conteneurs.
- La configuration de développement utilise « docker stack ». Cet outil assemble l'application à partir des images que nous avons construites et publiées sur le registre. En production, « docker stack » permettra la répartition des conteneurs sur plusieurs machines. En développement, il va nous permettre de mettre en place une intégration continue.

Les fichiers correspondant à ces deux configurations sont disponibles sur https://github.com/charleshuber/4tswarms.

La principale difficulté dans cette architecture est l'impossibilité pour un navigateur web, d'effectuer des requêtes JSON vers un nom de domaine différent de celui utilisé pour télécharger la page web (cross-origin requests)[1]. Il faut donc que la page HTML du client et les services du contrôleur soient exposés sous le même nom de domaine. Une autre difficulté est le choix d'implémenter une application SPA (Single Page Application) grâce au framework Angular4. Toute les requêtes venant des navigateurs doivent donc être redirigées vers la page « index.html », à l'exception des requêtes demandant des ressources statiques telles que les images. Ces problèmes ont été résolus par la mise en place d'un proxy Apache au niveau du conteneur client.

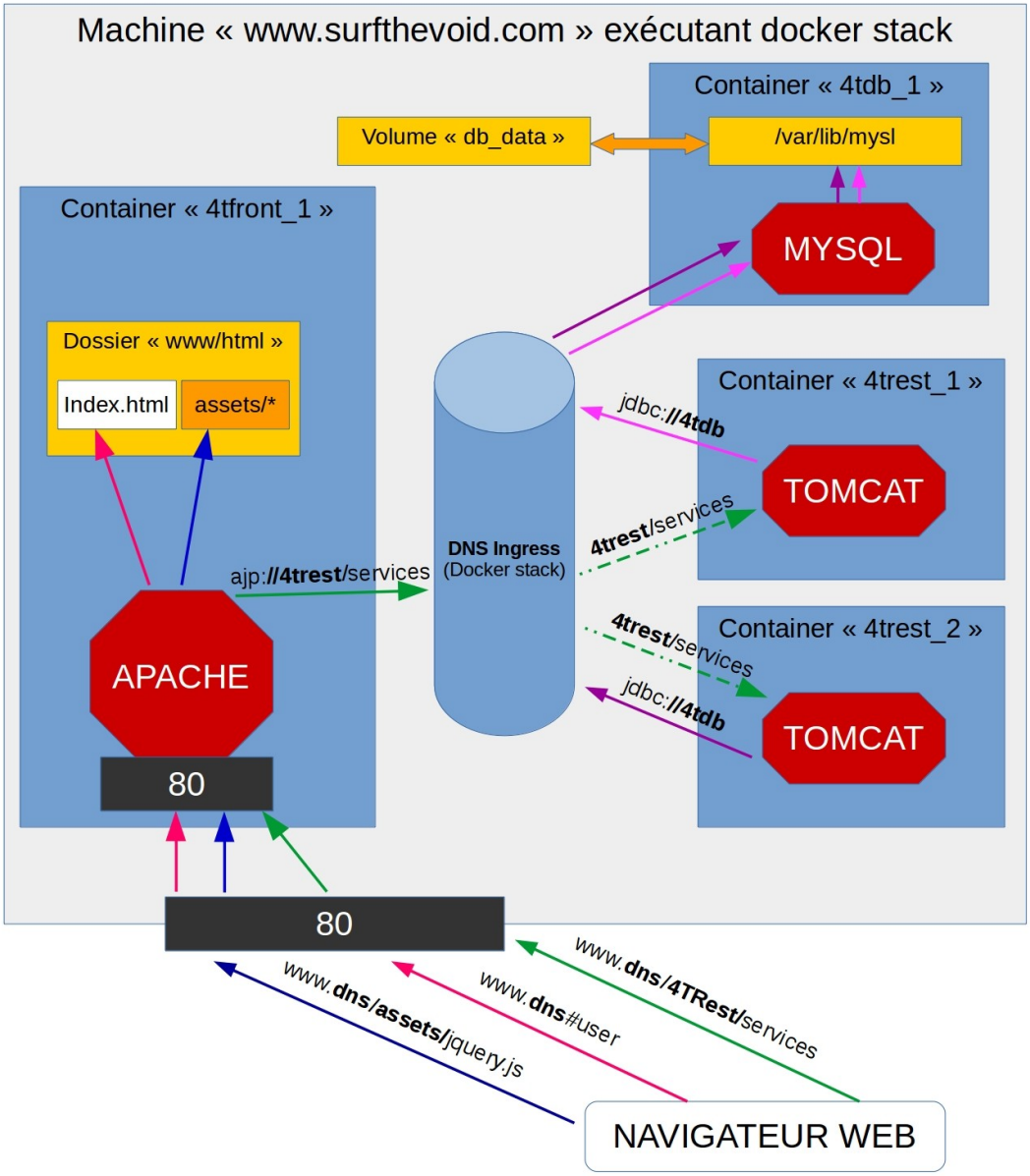

Illustration 1: Architecture de l'application sous "docker swarm"

1 En réalité il est possible de configurer le serveur pour permettre l'acceptation des requêtes depuis des noms de domaines étrangers, mais cela pose alors un réel problème de sécurité.

Installation de l'application

La configuration du proxy permettant de rediriger les requêtes, se situe dans le fichier «**000-default.conf** » présent dans les sources du composant client avec le fichier « Dockerfile ».

A la construction de l'image « **charleshuber/4tfront** », le fichier de configuration apache est ajouté à l'image en même temps que le script « **wait-tomcat.sh** », qui est également présent dans les sources. Ce script sera appelé au démarrage du conteneur pour s'assurer que le serveur Tomcat est opérationnel. De la même façon, les sources du composant serveur contiennent un fichier « **wait-mysql.sh** », qui permet au serveur Tomcat de ne pas démarrer tant que le base de donnée n'est pas opérationnelle.

Tout cet assemblage peut sembler compliqué ; c'est pourquoi avant d'expliquer en quoi consiste l'intégration continue et comment je l'ai mise en place, je vais décrire la façon dont l'application est exécutée en environnement de développement. Pour cela je vais rappeler les quelques étapes qui permettent d'installer et d'exécuter l'application sur n'importe quel poste :

- Installer la dernière version de docker (https://docs.docker.com/engine/installation/)
- Exécuter la commande **« sudo docker swarm init --advertise-addr [IP de la machine] »** pour indiquer à Docker de s'exécuter en mode « swarm ». (Dans notre cas, l'essaim de conteneurs n'a qu'un seul nœud).
- Télécharger la configuration de l'application docker en exécutant :

  « git clone https://github.com/charleshuber/4tswarms.git »
- Dans le dossier « **4tswarms**» téléchargé, aller dans le dossier « **dev** » et exécuter la commande **« sudo docker stack deploy -c docker-compose-stack.yml 4tdev ».**

  (Attention : le port 80 doit être libre)

A partir de ce point l'application est installée sur le poste concerné et est accessible en faisant une requête sur « localhost ». C'est le fichier « **docker-compose-stack.yml** » qui définit les images à utiliser, les variables d'environnement et les scripts a exécuter au démarrage des conteneurs.

Les scripts « **wait-mysql.sh** » et « **wait-tomcat.sh** » doivent donc être contenus dans les images téléchargées. Docker stack permet de définir un ordre de démarrage des services, mais cela n'est pas suffisant. Même si le fichier « **docker-compose-stack.yml** » indique que le service Mysql doit être démarré avant le service Tomcat, le processus Mysql n'est pas encore opérationnel au moment où les processus Tomcat démarrent. Ce qui oblige à démarrer les processus via ces scripts de sondage.

Le serveur de développement a été démarré une seule fois de la manière décrite ci-dessus. C'est à dire à partir de sources compilées et intégrées dans des images docker publiées et utilisables. L'intégration continue (CI) consiste à automatiser l'ensemble de la chaîne entre la publication d'une modification du code source et la mise à jour de l'application en environnement de développement.

Intégration continue

Le principe de l'intégration continue est simple. Lorsqu' un développeur soumet une modification sur le code, une plateforme compile le code modifié, exécute les tests unitaires pour confirmer qu'il est bon et, si tout est valide, déploie automatiquement le code sur le serveur de développement pour mettre à jour l'application. En pratique, l'intégration continue est quelque chose de compliquée à configurer sur des architectures standards, et de vraiment très compliquée à configurer sur une architecture docker .

Dans les cas d'une architecture standard, une plateforme d'intégration continue tel que « Jenkins » suffit à faire l'ensemble du travail. Dans notre cas, Jenkins va seulement exécuter une partie des tâches car il n'est pas capable de construire par lui même des images docker et de les publier. Il va se partager la tâche avec Docker Cloud et Docker Hub. Ces trois plateformes vont enchaîner leurs tâches par un système de notification intégré à GitHub.

Pour le moment la chaîne d'intégration continue n'est mise en place que sur les sources du service Tomcat. C'est à dire que la modification du code Angular4 ne met pas à jour automatiquement l'environnement de développement, alors qu'une modification des services REST le fait. C'est donc avec cet exemple que je vais décrire la chaîne d'intégration continue.

Comme il serait trop compliqué de détailler chaque point de configuration de Jenkins pour illustrer le diagramme d'activité qui va suivre, j'ai créé un compte sur la plateforme Jenkins qui a des droits de configuration sur les tâches. Il est donc possible de consulter la configuration de chacune des étapes à l'aide de ce compte.

Plateforme d'intégration continue : http://www.surfthevoid.com:8090

(**login** : cyril / **mot de passe** : smb215)

Concernant Docker Cloud, je n'ai pas réussi a créer d'autres utilisateurs. Le principe est qu'il est possible d'automatiser la construction d'une image à partir d'un repository GitHub, dès que celui-ci change. Plus généralement, en ce qui concerne GitHub et Docker Hub, les plateformes permettent de configurer l'appel d'une URL, dès qu'une action précise est effectuée (WebHook). Ces URL sont intégrées à la configuration des tâches comme élément de déclenchement, sur les plateformes cibles (étapes 2, 5 et 8).

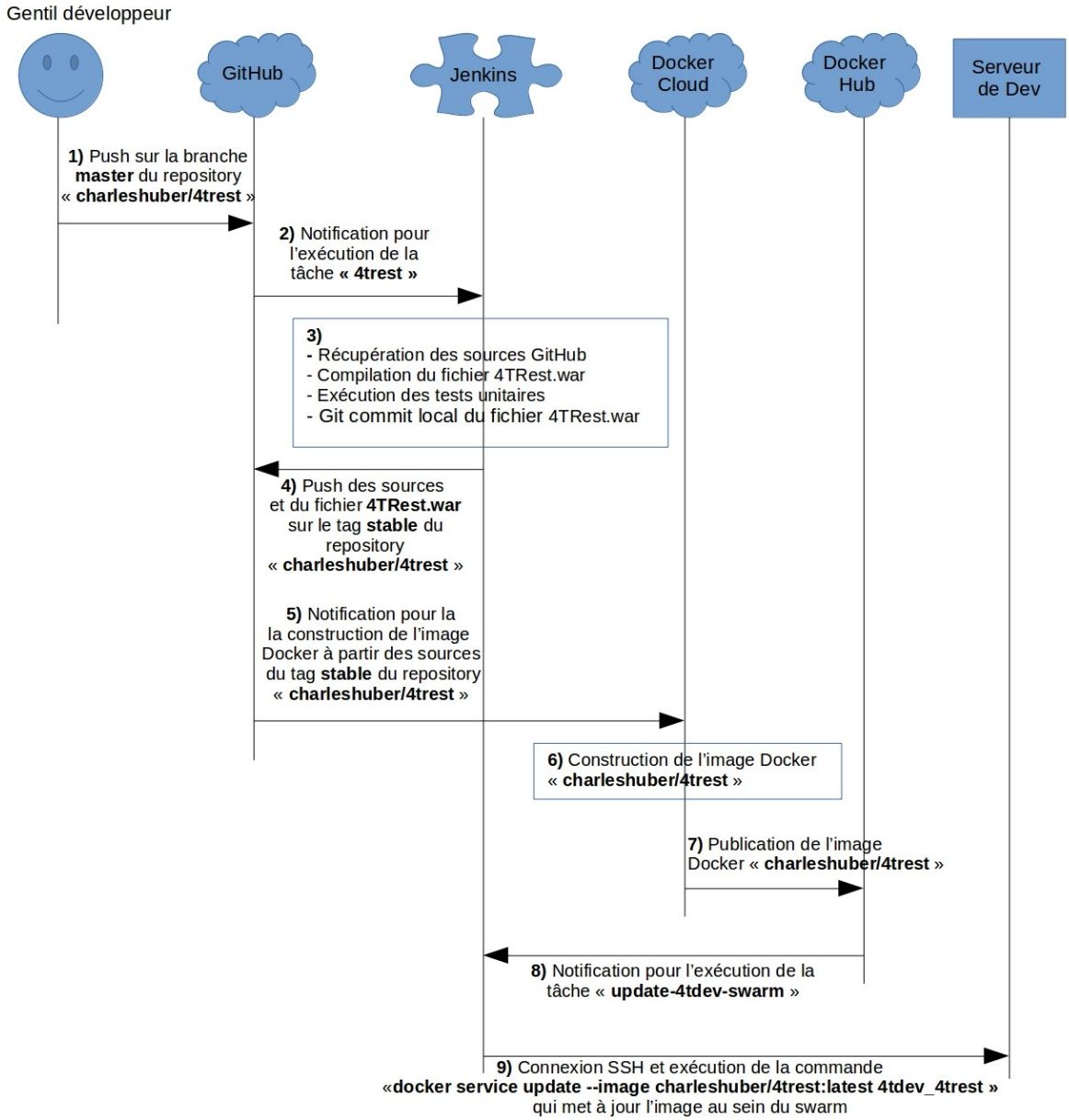

Illustration 2: Diagramme d'activité de la chaîne d'intégration continue

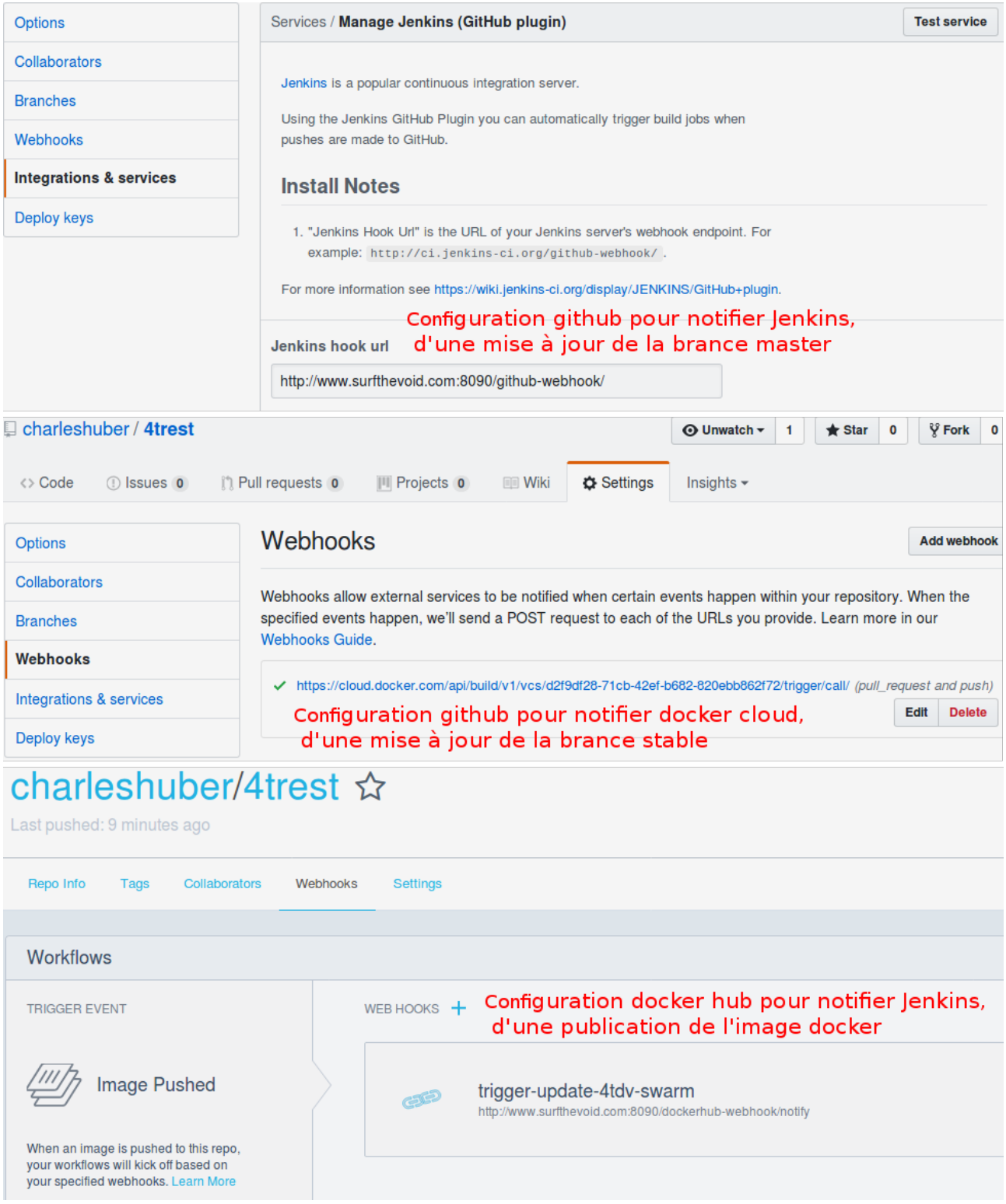

Illustration 3: Configuration des webhooks sur les plateformes GitHub et DockerHub

Technologies employées

Code client

L'avantage d'une application basée sur des micro-services REST, est l'indépendance complète du code client vis à vis du code serveur. La logique métier est située au niveau des services REST, alors que le code client est juste une couche de présentation qui peut évoluer ou se multiplier autant qu'on le souhaite. Compte tenu que le développement d'une application cliente était déjà le sujet principal de l'unité de valeur SMB214, j'ai pris la liberté d'expérimenter la technologie Angular4 en sachant que je ne pourrai pas trop rentrer dans les détails de son implémentation.

Les sources « charleshuber/4tfront », sont une branche du repository public « https://github.com/AngularClass/angular-starter » , destiné au commencement d'un projet angular4 compilé avec l'outil webpack.

Cet expérimentation est justifié par la volonté de réaliser une application cliente SPA (Single Page Application) pour offrir une expérience utilisateur plus agréable. Je ne donnerai aucun retour sur cet essai car je n'ai pas encore suffisamment de recule pour la juger.

Code serveur

Le code serveur est simplement l'exposition de web-services REST. Le principe est de proposer des actions élémentaires et atomiques pour chaque ressource en utilisant les 4 types de requête HTTP (pour ce projet, les services ont été développés pour les ressources « Group» et « User » bien que le code client n'exploite que la ressource « User »).

Le protocole HTTP propose les quatre types de requête POST, GET, PUT et DELETE , qui permettent respectivement de créer, lire, mettre à jour et supprimer une ressource. Le fondement des services REST est de dire que la disposition de ces quatre requêtes sur l'ensemble des ressources permet de faire toutes les actions possibles. La manipulation des ressources au niveau du client ressemble à la manipulation d'une base de donnée. La différence est que le serveur joue un rôle d'intermédiaire pouvant faire une authentification applicative, offrant une validation du modèle de donnée, adaptant le modèle de donnée à un objectif de présentation et pouvant effectuer des calculs métiers.

La description des services REST du projet est disponible à l'adresse http://www.surfthevoid.com/4TRest/services/resources?_wadl. Le projet est codé en Java 8. Il est compilé et préparer avec Maven. Le framework de web-services est CXF. Il se base sur l' API REST standard JAX-RS. Le framework de persistance est Hibernate. Il se base sur l'API standard JPA2. Le tout est orchestré par le framework d'injection de dépendance Spring.

Architecture

Chaque ressource est gérée par 6 classes organisées en couches :

- Une classe héritant de la classe abstraite « **Resource** », qui représente le modèle de donnée échangé avec le client. La ressource est échangée au format JSON.
- Une classe héritant de la classe abstraite « **ResourceBoundary** », qui représente le point d'accès à la manipulation de la ressource. Cette classe propose au minimum les quatre méthodes vues précédemment. Elle ne fait aucun traitement. Son rôle est de déléguer le traitement à une classe de service, de gérer le retour d'exception au client en cas d'erreur et de sérialiser la ressource au format JSON.
- Une classe de service héritant de la classe abstraite « **ResourceStore** » qui est responsable du traitement. Chacun des appels fait sur la classe de service par la classe héritant de « **ResourceBoundary** », est encapsulé dans une transaction au niveau de la base de donnée. C'est à dire que, soit la totalité de l'appel est exécuté, soit aucune modification n'est effectuée en base de donnée si une erreur à lieu au milieu du traitement. Son rôle principale et de valider les données entrantes et de les transformer vers le modèle de la base de donnée avant de les enregistrer. La validation et l'enregistrement sont déléguées à des classes spécifiques.
- Une classe héritant de la classe abstraite « **Validator** » responsable de la validation des ressources entrantes.
- Une classe héritant de la classe abstraite « **GenericDbo** » qui représente le modèle de donnée rattachée à la base de donnée.
- Une classe héritant de la classe abstraite « **GenericDao** » responsable de la lecture et de la sauvegarde de la ressource en base de donnée.

Dans le répertoire des sources, les classes destinées à être partagées avec d'éventuels clients écrits en Java, ne sont pas intégrées au code métier situé dans le dossier nommé « **business-rest** ». Toutes ces classes d'interfaces sont contenues dans un jar indépendant nommé « **business-rest-definition** » , qui pourra être partagé avec un client. Ce jar est composé de :

- la classe « **Resource** » et toutes les classes qui en héritent.
- La classe « **IResourceBoundary »** et toutes les classes qui en héritent. Ce sont les interfaces des classes héritant de « **ResourceBoundary** ». Elles définissent toutes les méthodes pouvant être appelées sur les ressources, ainsi que les URL relatives pour y accéder. Ce sont ces classes qui contiennent les annotations JAX-RS qui permettent au framework CXF d'exposer les web-services dans le contexte Spring ou de générer des classes clientes en fonction du contexte.

- Les classes « '**Resource'ValidationErrors** » qui contiennent les différents codes d'erreur pouvant être renvoyés suite à la validation d'une ressource, ainsi que leur signification en anglais.
- La classe « **ValidationException** » qui représente le modèle de donnée servant à véhiculer les erreurs de validation lorsqu'une requête échoue pour une raison identifiée. Comme les ressources, cette exception est renvoyée au format JSON dans la réponse lorsque celle-ci contient l'entête HTTP « **validation-failed** ».

L'organisation du code n'a pas été réalisé en suivant le modèle en couche qu'il implémente. Plutôt que de faire un dossier pour chaque type de classe, le choix a été fait de mettre en place un dossier pour chaque ressource. Ce dossier contient l'ensemble des classes nécessaires au traitement de la ressource (à l'exception des classes d'accès à la base de donnée qui sont d'avantage sujet à un changement de technologie). Grâce à l'utilisation des types génériques et au expressions lambda introduites dans Java 8 (pointeur de fonction), il a été possible de réaliser un très grand taux de factorisation du code. Si bien que la majorité du code est implémenté dans le package générique des classes abstraites.

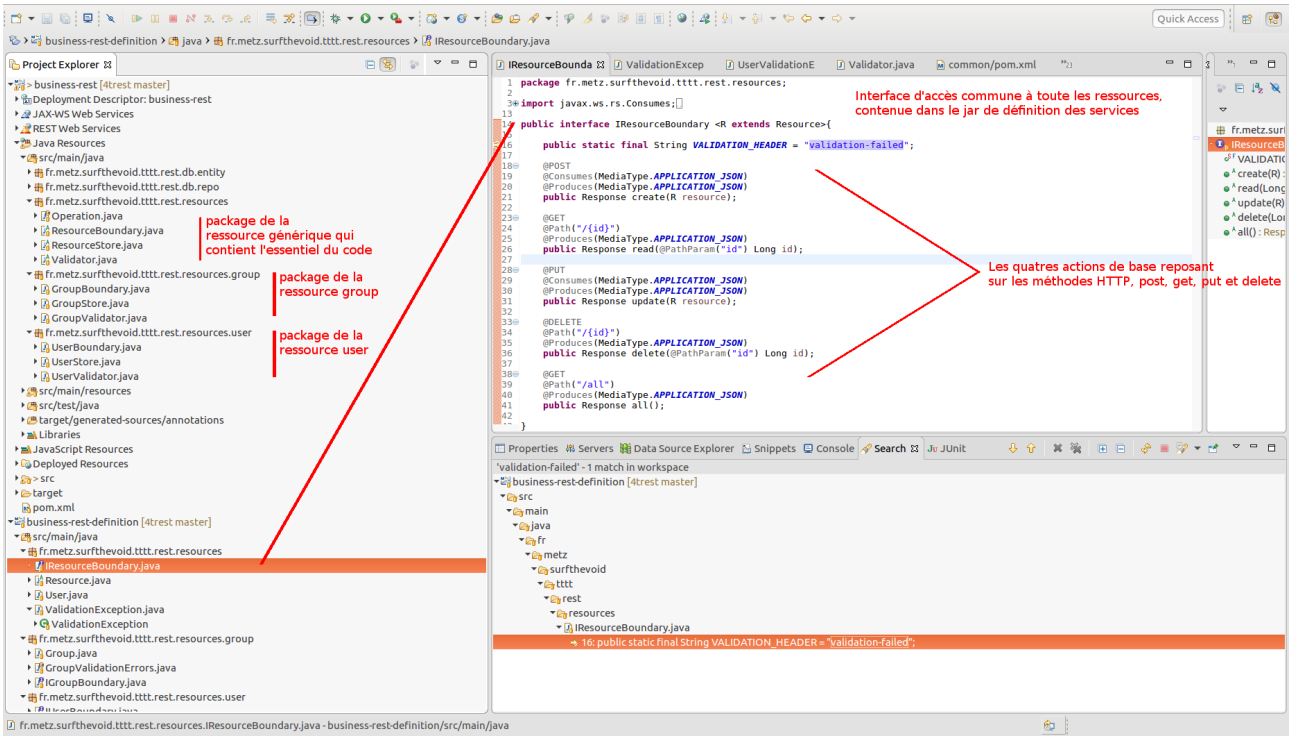

Illustration 4: Organisation du code de l'application

Il est difficile de continuer à décrire le fonctionnement des services REST sans rentrer dans le détail du code. Je vous invite donc à suivre le déroulement de la création d'un utilisateur en lisant les commentaires. La porte d'entrée de la visite commence avec la méthode `create(R resource)` de la classe **ResourceBoundary**, dont la classe **UserResourceBoundary** hérite de toutes les méthodes (cette dernière ne traitera que des appels spécifiques à la ressource **User**).

L'essentiel du code se trouve ensuite dans les classes abstraites **ResourceStore** et **Validator**. Les classes filles n'ont pour rôle que de surcharger des fonctions de traitement spécifiques à une ressource en particulier.

Note :

Les sources doivent être compilées avec Maven. Par défaut le code ne compile pas, car certaines classes utilitaires pour le framework de persistance sont générées par Maven lors de la compilation.

Pour compiler le projet il faut :

- Installer **Maven** (au minimum la version 3) et le **jdk8** (java development kit)
- lancer la commande « **mvn clean install** » dans le dossier « **sources/common** »
- lancer la commande « **mvn clean install** » dans le dossier « **sources/business-rest-definition**»
- lancer la commande « **mvn clean install** » dans le dossier « **sources/business-rest**»