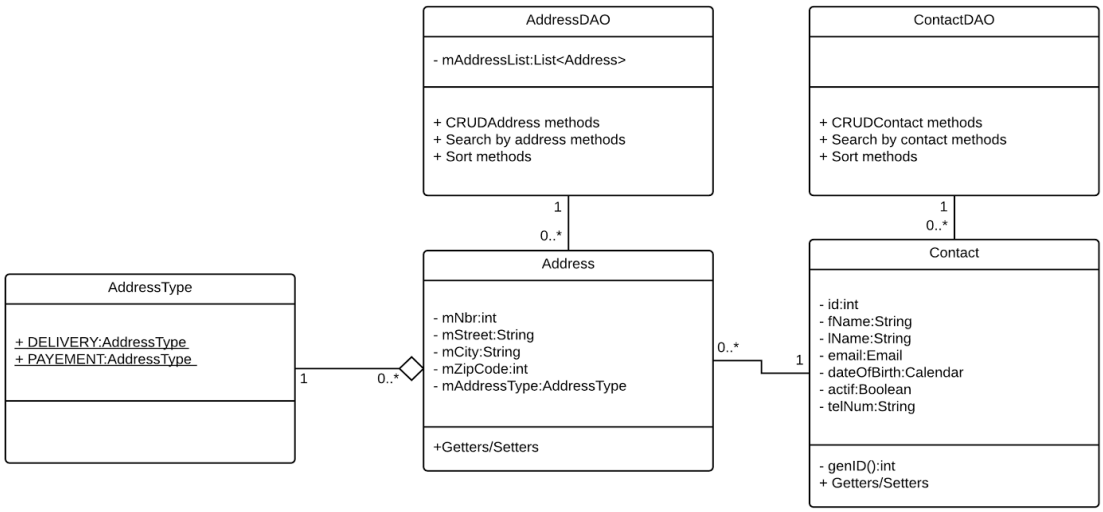
Projet IHM
Documentation

L'objectif de ce projet est la conception et la réalisation de l'IHM d'une application web de gestion de contacts.
Pour cela, nous utilisons SpringMVC ainsi que Maven pour la gestion des dépendances.
L'IDE est Eclipse Luna et le runtime Java est la version 7.

Architecture et technologies

En ce qui concerne la conception, nous avons commencé par concevoir la partie modèle de l'application avec l'UML suivant :



Puis, pour la conception de l'IHM, nous avons décidé d'utiliser les principes REST afin de structurer l'architecture de notre application. REST (REpresentational State Transfer) est un ensemble de pratiques et de contraintes permettant l'indépendance de la partie client et serveur et utilisant toutes les particularités du protocole HTTP dans le cas d'une application web.

L'architecture est donc la suivante:

Nous avons créé trois répertoires afin de bien séparer les différentes tâches de notre application.

Ainsi, nous avons :

- un répertoire "controller" dans lequel sont spécifiés les controllers.
- un répertoire "entity" qui contient nos entités.
- un répertoire "model" contenant deux sous répertoires : Service et DAO.

Contenu de nos répertoires :

- Le répertoire "controller" contient deux contrôleurs :

  - AddressController
  - ContactController

- Le répertoire "entity" contient nos entités :

  - Address
  - Contact
  - AddressType (qui est une énumération).

- Le répertoire "model" contient deux sous-répertoires :

  - dao

  Ces sous-répertoires contiennent eux-mêmes :

  - Le sous-répertoire dao :

  Deux interfaces : IAddressDAO et IContactDAO
  Leurs implémentations : AddressDAOImpl et ContactDAOImpl.

De plus, suivant REST, chaque ressource n'est accessible qu'à une seule URI.

Graphiquement, nos pages sont sous forme de JSP et sont stylisées grâce à Bootstrap.
La technologie proposée par défaut par SpringMVC pour les pages est JSP, ainsi nous avons préféré nous contenter de celle-ci.

La navigation a été pensé de façon simple et méthodique.
La page d'accueil liste les contacts entrés par l'utilisateur ainsi que les informations associés.
Les actions associées aux contacts sont contenues dans les pages sous l'arborescence /contact et celles associées aux adresses sous l'arborescence /contact/address.

Configuration du projet et technologies utilisées

Le projet Eclipse se présente sous la forme d'un projet Google AppEngine et est testé avec JUnit.
La gestion des dépendances a été faite avec Maven, ce qui a été d'une aide considérable à ce niveau. Le temps d'apprentissage du fonctionnement de Maven n'a pas été très long puisque le XML nous était familier mais il est vrai que son côté verbeux est quelque fois repoussant, en particulier lorsque les dépendances sont considérables ainsi que les paramètres de build et la gestion des versions de chaque librairies importée.
Cependant, le fait qu'il aille souvent de paire avec Eclipse nous a fait choisir Maven plutôt que Gradle.

Il peut être déployé sur Apache Tomcat, Google AppEngine et nous avons également utilisé Jetty version 9 car il fournit plus de logs que Tomcat et AppEngine, ce qui permet d'être plus efficace lors du débug.
Malgré le fait qu'Apache Tomcat soit directement intégré dans Eclipse, il se révèle quelque fois lent, capricieux et préfère le lancement en console depuis Maven plutôt qu'avec l'interface graphique d'Eclipse qui entraine quelques bugs de temps à autre.
Google AppEngine, quant à lui, nécessite une installation d'abord fastidieuse même après l'importation Maven car il est largement préférable (voire indispensable) d'ajouter une "nature" AppEngine au projet afin de faciliter son déploiement sur le cloud.
AppEngine permet toutefois un déploiement en local qui se révèle pratique et qui est basé sur Jetty (version 6.1 dans notre cas).
Jetty permet, entre autre, un déploiement "à chaud" du projet, ce qui veut dire qu'il n'est pas nécessaire de redémarrer le serveur lorsque les JSP sont mis à jour.

Cependant, si ce sont les classes Java telles que les contrôleurs qui sont mises à jour, il faut tout de même recompiler le projet et redémarrer le serveur.

Afin de débugger en profondeur le projet, nous avons également utilisé un plugin Chrome appelé Advanced Rest Client qui permet de forger ses propres requêtes et qui affiche plusieurs informations telles que le code de la page retournée, la réponse serveur ainsi que les headers de requête et de réponses.

Sur Github, seules les sources ainsi que les fichiers de configuration du projet associés à Maven et AppEngine sont disponibles.
Le code est commenté et possède une documentation Java.
Nous avons également fait des tests JUnit tout au long du projet afin de minimiser les bugs au fil du temps. Ces tests ont porté sur le modèle ainsi que les DAO.

Problèmes rencontrés

Le premier problème rencontré a été la mise en place du projet sur Eclipse.
En effet, celui-ci impliquant à la fois Maven, Google appengine et Spring, il a fallu se poser la question de créer un projet à nature Maven, Web dynamique, Google ou Spring.

La solution Spring s'avère pratique au début puisqu'il est possible ensuite de le convertir en projet Maven et ainsi d'importer les dépendances manquantes.
Cependant, le déploiement Google se fait de façon plutôt externe à Eclipse puisque les librairies appengine ne sont même pas incluses dans la vue Eclipse du projet, malgré le fait qu'elles soient présentes et qu'elles participent à la compilation du projet.

Cette difficulté nous a forcés à resynchroniser plusieurs fois nos projet car sur le répertoire Git, nous ne partagions que le code et les fichiers de configuration.
Au final, nous avons opté pour un projet à nature Google, même si en réalité plusieurs natures sont présente dans le fichier .project du projet Eclipse.