Laboratoire sur le projet de départ

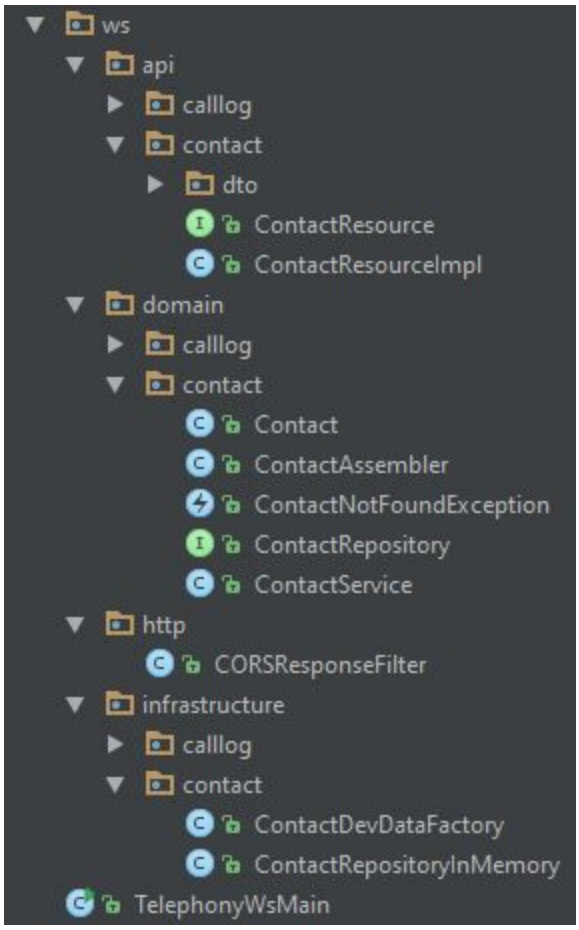
Automne 2016

Olivier Dugas

Thème du laboratoire

Présentation d'un projet de base qui sera utilisé pour votre travail d'équipe de session.

1. Structure



Le projet est structuré selon le modèle en couche avec la section Api, Domain et Infrastructure qui interagissent l'un avec l'autre.

2. Api

Le package *Api* contient le code responsable de la communication externe du logiciel. Il y aura donc un lien avec Jersey qui sert à faire la corrélation entre une méthode et une route HTTP. L'interface est utilisée pour placer les annotations de *routing* et l'implémentation répond aux appels.

Par exemple:

```
@Path("/telephony/contacts")
public interface ContactResource {

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  List<ContactDto> getContacts();
```

Lorsqu'un appel "GET" est fait à "http://<*URL du serveur*>/telephony/contacts/" la méthode qui implémente le "getContacts" de l'interface sera exécuté.

Le @Produces sert à spécifier que le format JSON sera utilisé comme support texte de retour vers le client ayant fait la requête. (À l'inverse du XML par exemple).

```
@Override
public List<ContactDto> getContacts() {
  return contactService.findAllContacts();
}
```

Le code exécuté sera celui de "ContactResourceImpl.getContacts()".

* (Anglais) Bon aperçu des verbes HTTP:
http://www.restapitutorial.com/lessons/httpmethods.html

3. Domain

Le package *Domain* contient le code responsable de la logique et de la définition des objets du domaine. C'est ici que sera toujours présent, peu importe l'implémentation ou le service externe utilisé, le code qui sera en mesure de répondre aux besoins des utilisateurs de l'application. Dans un monde idéal, ce code pourrait être utilisé derrière une API Web, une application native avec interface, en ligne de commande et utilisant une base de données SQL, MongoDB, en mémoire, etc.

Les parties du domaine qui impliquent une communication externe, l'utilisation d'une technologie particulière ou laisse le choix à l'utilisateur de l'implémenter seront des interfaces:

```
public interface ContactRepository {
  List<Contact> findAll();

  Contact findById(String id);

  void update(Contact contact)
          throws ContactNotFoundException;

  void save(Contact contact);

  void remove(String id);
}
```

4. HTTP

Package créé pour contenir les classes spécifiques à la configuration du serveur HTTP. En ce moment seulement le filtre pour rendre disponible le Cross Origin Resource Sharing est implémenté, mais d'autres filtres ou classes pourraient y être, tel que la validation de Header pour la sécurité, la compression des données de retour, etc.

* (Anglais) Source intéressante d'information sur le Cross Origin: http://enable-cors.org/index.html

5. Infrastructure

Ici se trouveront les implémentations des interfaces du package Domain qui sont propres à une technologie ou un environnement. Dans notre cas, vue que nous n'utiliserons pas de bases de données, l'implémentation "In Memory" est présente pour effectuer le travail de conserver les données de l'application.

```
public class ContactRepositoryInMemory implements ContactRepository {

  private Map<String, Contact> contacts = new HashMap<>();

  @Override
  public List<Contact> findAll() {
    if (!contacts.isEmpty()) {
      return Lists.newArrayList(contacts.values());
    } else {
      return new ArrayList<>();
    }
  }

  @Override
  public Contact findById(String id) {
    return contacts.get(id);
  }

  @Override
  public void update(Contact contact)
          throws ContactNotFoundException {
    Contact foundContact = contacts.get(contact.getId());
    if (foundContact != null) {
      contacts.put(contact.getId(), contact);
    } else {
      throw new ContactNotFoundException("Contact not found, cannot be updated");
    }
  }

  @Override
  public void save(Contact contact) {
    contacts.put(contact.getId(), contact);
  }

  @Override
  public void remove(String id) {
    contacts.remove(id);
  }
}
```

6. Main

La classe “TelephonyWsMain” va servir de pont entre les différentes couches, instancier tous les objets et les orchestrer afin de pouvoir bâtir l’application. Finalement, démarrer le serveur et attendre que les premières requêtes HTTP viennent interagir avec celui-ci.

```
public static void main(String[] args)
        throws Exception {

  // Setup resources (API)
  ContactResource contactResource = createContactResource();
  CallLogResource callLogResource = createCallLogResource();

  // Setup API context (JERSEY + JETTY)
  ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);
  context.setContextPath("/api/");
  ResourceConfig resourceConfig = ResourceConfig.forApplication(new Application() {
    @Override
    public Set<Object> getSingletons() {
      HashSet<Object> resources = new HashSet<>();
      // Add resources to context
      resources.add(contactResource);
      resources.add(callLogResource);
      return resources;
    }
  });
  resourceConfig.register(CORSResponseFilter.class);

  ServletContainer servletContainer = new ServletContainer(resourceConfig);
  ServletHolder servletHolder = new ServletHolder(servletContainer);
  context.addServlet(servletHolder, "/*");

  // Setup static file context (WEBAPP)
  WebAppContext webapp = new WebAppContext();
  webapp.setResourceBase("src/main/webapp");
  webapp.setContextPath("/");

  // Setup http server
  ContextHandlerCollection contexts = new ContextHandlerCollection();
  contexts.setHandlers(new Handler[] { context, webapp });
  Server server = new Server(8080);
  server.setHandler(contexts);

  try {
    server.start();
    server.join();
  } finally {
    server.destroy();
  }
}
```

* Le code est laissé en une longue méthode, mais cela pourrait facilement être séparé en sous-méthodes pour aider à la lecture et la clarté du code.

7. Exercice:

Utilisez le code disponible sur le site de l'ENA ou de la branche "ulaval-labo1" du répertoire Git:

- git checkout https://github.com/bgagnonadam/embedded-webapp-template.git
- git branch labo1 origin/ulaval-labo1
- git checkout labo1

Pour partir l'application:

- mvn compile
- mvn exec:java

Effectuez l'implémentation de la ressource CallLog de l'application, qui sert à afficher l'historique des appels de l'usager.

```
public interface CallLogResource {

  List<CallLogDto> getCallLogs();

  void deleteCallLog( String id);
}
```

Les verbes HTTP disponibles de la ressource sont GET et DELETE. POST et PUT ne seront pas implémentés car l'historique d'appel sera bâti par l'application en surveillant les interactions.

* Inspirez-vous de la ressource "contact" vue qu'il s'agit de la même structure, avec moins de fonctionnalités.
* La page web pour cette ressource web est déjà disponible et accessible en exécutant l'application.

Aide mémoire pour les remises

- Un jeu de données pour tester.
- Un rapport pour aider les correcteurs:
    - Instructions pour l’installation (prérequis et exécution du projet).
    - Diagrammes d'architecture et leur description.
    - Patrons retenus et non retenus avec justifications.
    - Liste des stories complète avec celles complétées identifiées.
    - Explications sur les choix qui ont été pris (Architecturaux, bris de règles de qualité et de design, etc.).
    - Faiblesses et forces de votre architecture selon vous.
- Les attentes lors de la correction:
    - Du code de qualité: utilisation de patrons, pas de répétition de code, bonne séparation des objets et des concepts, principes SOLID, etc.
    - Une bonne couverture et qualité des tests.
    - Pas une sécurité digne de production: utilisateur et mot de passe passés en clair est OK.
    - Pas une grosse attention sur le UI.
    - Le Front-End (/webapp/*) n’est **PAS** corrigé.