EIA-FR

TIC / I-3

SYSTÈMES D'INFORMATION 2

TP 12

Mini-projet

Auteurs :

Mathieu CLÉMENT

Bryan PILLER

Professeurs :

Omar ABOU KHALED

Stefano CARRINO

Joël DUMOULIN

19 janvier 2015

Introduction

Dans ce document, nous présentons le mini-projet créé durant le cours de systèmes d'information 2. Le but était de créer une application qui met en œuvre des technologies vues durant les séances de cours. Nous avons décidé d'implémenter un système de commande de pizzas. Le design est basé sur des sites connus comme *Domino's Pizzas*. Les fonctionnalités sont l'ajout de pizzas comme la *Quattro Formaggi* dans un panier, la sélection de la quantité voulue ainsi que la grandeur (petite ou grande pizza) et de confirmer la commande en ajoutant l'adresse de livraison.

Pour réaliser cette application nous avons utilisé différentes technologies, standards, bibliothèques et programmes :

— AngularJS,
— JSON,
— REST,
— Jersey,
— Hibernate,
— Tomcat,
— Maven,
— etc.

1 Conception

1.1 But de l'application

Cette application composée d'une partie frontend et backend (web service) permet à des clients affamés de consulter la carte (avec nom des pizzas, liste des ingrédients et prix) d'un service de livraison de pizzas à domicile, de faire son choix et d'effectuer une commande après avoir entré une adresse de livraison.

Une interface permet ensuite aux employés de consulter la liste des commandes pour préparer les pizzas et les faire livrer.

1.2 Architecture générale / MVC

Vue	HTML, CSS, AngularJS	La vue est constituée d'une page HTML unique mais permettant de passer de la carte des pizzas au formulaire de commande.
Contrôleur	AngularJS, Jersey	L'aspect gestion du panier du client et une partie de la validation est effectuée côté navigateur, tandis que la mise à jour de la base de données et une autre partie de la validation est réalisée du côté web service.
Modèle	Entités Hibernate, BDD	Le modèle est géré grâce à des entités utilisant les annotations JPA. Les requêtes à la base de données et l'implémentation de l'ORM font appel à Hibernate.

2 Réalisation du backend

2.1 Technologies utilisées

— Hibernate ORM
— Hibernate H2 in-memory database (pour le développement)

— PostgreSQL (en production)
— JPA annotations
— Jersey REST framework, implémentation de référence de JAX-RS
— Gson pour sérialisation JSON
— SLF4J pour les logs

2.2 Journaux (logging)

Si imprimer sur la console constitue une mauvaise idée, il est en revanche utile d'utiliser un *logger*. Nous avons apprécié SLF4J car il se marie bien avec le système de logging interne à Tomcat (qui utilise probablement Log4j puisque les deux sont issus du projet Apache). Le log courant de Tomcat est `$CATALINA_HOME/logs/catalina.out` .

En plus de la déclaration de la dépendance avec Maven, il faut avant de pouvoir utiliser le logger le déclarer dans la classe. Voici un exemple :

```
private static final Logger logger = LoggerFactory.getLogger(
                    "pizzaorders.webservice.HibernateListener");
```

Il est ensuite aisé d'écrire des entrées d'information, de déboggage ou des erreurs dans le journal.

2.3 Web Service

2.3.1 Chemins REST

Le fichier WADL permet bien entendu de voir comment intéragir avec l'application. Ici nous nous contenterons d'une courte explication et d'un exemple.

Les URLs sont raccourcies par question de lisibilité. On les préfixera par exemple avec `http://localhost:8080/pizzaorders-ws/rest`.

— `GET /pizzas` liste toutes les pizzas avec leurs ingrédients :

```
[
    {
        "name": "Margherita",
        "priceBig": 15,
        "priceSmall": 12,
        "ingredients": [
            {  "name": "tomates"          },
            {  "name": "basilic frais"    },
            {  "name": "mozzarella"       }
        ]
    },
    {
        "name": "Regina",
        etc.
    }
]
```

— `GET /pizzas/{pizzaName}` donne la même information mais pour une pizza précise ;
— `PUT /pizzas/{pizzaName}` permet d'ajouter une nouvelle pizza au système ;
— `PUT /ingredients/{ingredientName}` permet d'ajouter un nouvel ingrédient au système ;

— `POST /ingredients/forPizza/{pizzaName}/{ingredientName}/` ajoute un ingrédient à une pizza. La base de données étant normalisée (3FN) cette association fait l'objet d'une table séparée. Relation *Many-to-Many* ;

— `GET /orders` affiche la liste de toutes les commandes :

Note : on pourrait se contenter de reprendre uniquement le nom de la pizza mais cela peut amener des problèmes. Par exemple si on veut calculer le montant des commandes du mois mais que le prix de la pizza a changé entre-temps. Le même problème s'applique aux ingrédients.

On pourrait aussi améliorer le format de la date, utile à l'humain mais plus compliqué que d'autres formats à parser par la machine. Cela est dû au fait que le convertisseur JSON a vraisemblablement utilisé la méthode `toString()` de `Date`.

Enfin, un défaut : à cause de la manière dont est implémentée la clé composée (quadruplet numéroCommande, nomPizza, taillePizza, quantité) pour les pizzas commandées, on voit apparaître `"assocId"` qui n'apporte aucune information. Il s'agirait alors de coder une exception dans la sérialisation JSON pour éviter ce problème, déjà qu'il a fallu déclarer le numéro de la commande qui figurait dans le `"assocId"` comme *transient* sinon cela causait un cycle. Le même comportement peut être observé avec `toString()` ;

— `GET /orders/{orderId}` présente la même information mais pour une seule commande ;

— `POST /orders/begin` débute une transaction « Commande ». Ceci est détaillé plus loin dans ce document. Cette ressource REST retourne le numéro de la commande que l'on devra utiliser plus tard.

Pour adhérer à la philosophie REST nous mettons l'en-tête *Location* à `/orders/{orderId}`. Mais comme nous sommes sympas nous retournons aussi `{"id": 1}` pour faciliter le travail des développeurs ;

— `POST /orders/{orderId}/{qty}/{pizzaSize}/{pizzaName}` ajoute une pizza de la taille voulue dans la quantité voulue à la commande. On n'aurait peut-être pas dû passer la quantité, la taille de la pizza et son nom dans l'URL, car il n'y a pas réellement de « ressource » à cet endroit ;

— `POST /orders/{orderId}/confirm` permet de finaliser la commande. On passe dans la forme `x-www-form-urlencoded` (par défaut de Jersey) l'adresse de livraison ;

— `DELETE /orders/{orderId}` permet d'annuler la commande et interrompre la transaction.

2.3.2 `PUT` et `POST`

Dans la littérature, il est dit que l'on devrait utiliser `POST` pour mettre à jour une ressource. Si la ressource n'existe pas encore, alors il fait sens de préférer `PUT`.

https://jcalcote.wordpress.com/2009/08/06/restful-transactions/

2.3.3 `DELETE`

Ce serait une aberration de faire un `POST /orders/1/delete` au lieu de `DELETE /orders/`. Quel horreur ! Malheureusement de nombreux web services utilisent `POST` pour tout et n'importe quoi. Au final cela n'a plus grand chose à voir avec les principes REST, ni même HTTP pour certains « bricolages » (erreurs rapportées dans du XML / JSON avec code de status 200…)

2.3.4 Codes de statut HTTP

D'ailleurs, parlons de la bonne utilisation des codes de statut HTTP.

Nous avons notamment utilisé :

200 OK Opération réussie

201 Created Ressource créée avec succès (par exemple pizza, ingrédient, commande, etc.)

400 Bad Request Erreur dans les paramètres ou les données, faute à l'utilisateur

404 Not Found La ressource n'existe pas (par exemple pas de commande avec le numéro indiqué)

409 Conflict Nous avons utilisé ce code n'ayant pas trouvé la *best practice* si une ressource existe déjà.

410 Gone pour les cas où l'on sait pertinemment qu'il existait une ressource à l'adresse demandée mais qu'elle ne réapparaîtra pas. Indique au client de cesser de faire des requêtes sur cette ressource puisqu'elle est « partie ».

2.4 Accès aux données

Comme indiqué au début de cette section, nous avons fait appel à Hibernate, aux annotations JPA et à la base de données H2 d'Hibernate pour le stockage pendant la phase de développement.

2.4.1 Configuration Hibernate

Si Hibernate dans ses dernières versions permet d'utiliser les annotations à bien des égards, il est toujours nécessaire de créer un fichier de configuration.

Le voici :

```
<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"/>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <property name="current_session_context_class">thread</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.H2Dialect</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <!--
        <mapping resource="org/hibernate/tutorial/hbm/Event.hbm.xml"/>
        -->
        <!-- mapping package ? -->
        <mapping class="ch.eifr.lesdarons.pizzaorders.webservice.orm.entities.IngredientEntity"/>
        <mapping class="ch.eifr.lesdarons.pizzaorders.webservice.orm.entities.PizzaEntity"/>
        <mapping class="ch.eifr.lesdarons.pizzaorders.webservice.orm.entities.AddressEntity"/>
        <mapping class="ch.eifr.lesdarons.pizzaorders.webservice.orm.entities.OrderEntity"/>
        <mapping class="ch.eifr.lesdarons.pizzaorders.webservice ... PizzaToOrderAssocEntity"/>

    </session-factory>

</hibernate-configuration>
```

Le principe *Convention-over-Configuration* est adopté pour… l'emplacement de ce fichier. En le mettant dans le dossier `src/main/resources/` et en le nommant `hibernate.cfg.xml` alors il ne sera pas nécessaire de préciser où il se trouve à l'initialisation.

2.4.2 Bavardage

Dans le fichier de configuration nous avons activé l'affichage des requêtes SQL (et non pas HQL ici). C'est particulièrement utile non seulement pour déboguer en cas de problème, mais aussi pour vérifier que le framework fait bien ce que l'on attend de lui (pour contrôler que la cardinalité d'une relation est bien celle que l'on pense par exemple), ainsi que pour trouver la source d'éventuelles lenteurs.

Le journal de Tomcat affiche alors, par exemple :

```
    select
        ingredient0_.pizzas_name as pizzas_n1_5_0_,
        ingredient0_.ingredients_name as ingredie2_6_0_,
        ingredient1_.name as name1_2_1_
    from
        pizzas_ingredients ingredient0_
    inner join
        ingredients ingredient1_
            on ingredient0_.ingredients_name=ingredient1_.name
    where
        ingredient0_.pizzas_name=?
```

Notez la présence du point d'interrogation. Il permet de se prémunir d'attaques XSS et est géré entièrement par Hibernate pour autant que l'API appropriée soit appelée. Nous le verrons plus loin dans ce document lorsque nous évoquerons certaines requêtes particulières.

2.4.3 Insertion des données initiales

Plus pour l'exercice qu'autre chose (car le plus simple aurait été d'insérer directement les données dans la base de données) nous avons intégré au web service les méthodes pour insérer des pizzas, des ingrédients et associer les seconds aux premières.

Mais nous n'allions quand même pas coder l'insertion des données initiales en dur dans du code Java. Nous lisons plutôt le fichier XML suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<pizzas>
    <pizza>
        <name>Margherita</name>
        <price small="12" big="15"/>
        <ingredients>
            <ingredient>mozzarella</ingredient>
            <ingredient>basilic frais</ingredient>
            <ingredient>tomates</ingredient>
        </ingredients>
    </pizza>
    etc.
</pizzas>
```

Par-contre, pour être tout à fait honnête, il faut dire que le code qui lit le XML et effectue ensuite les appels au web service n'est pas des plus beaux. La faute à l'API vieillissante du DOM en Java. Peut-être utiliserons-nous JDOM la prochaine fois.

Exemple navrant :

```
XPathExpression ingredientXPath = XPathFactory.newInstance().newXPath().compile("//ingredient");
NodeList ingredientNodeList = (NodeList) ingredientXPath.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < ingredientNodeList.getLength(); i++) {
    Node node = ingredientNodeList.item(i);
    if(node.getNodeType() != Node.ELEMENT_NODE) continue;

    Element element = (Element) node;
    String ingredientName = element.getTextContent();
    // etc.
}
```

Notons le fait qu'une `NodeList` ne soit pas itérable. La classe devrait implémenter l'interface du même nom introduite par Java 5 pour permettre l'utilisation du sucre syntaxique qu'est *foreach*.

Les génériques, n'en parlons même pas, il n'y qu'à voir la présence de *casts* explicites et d'un odieux *getNodeType()* mais nous nous éloignons du sujet de ce travail pratique…

2.4.4 `web.xml`

Nous avons déjà vu le `web.xml` à utiliser avec Jersey lors du travail pratique qui a introduit le framework, mais nous devons y ajouter un moyen d'initialiser certains aspects d'Hibernate. Il est possible d'enregistrer un *listener* comme ceci :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                                           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>ch.eifr.lesdarons.pizzaorders.webservice.ws_resources</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Web Application</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
    <!-- ICI on peut "plugger" des classes pour executer du code au "deploy" et "undeploy"
         de l'application. -->
    <listener>
        <listener-class>ch.eifr.lesdarons.pizzaorders ... HibernateListener</listener-class>
    </listener>
</web-app>
```

2.4.5 Initialisation d'Hibernate : Création de la `SessionFactory`

Pour toute opération agissant sur la base de données, il est nécessaire de créer une session. Pour se faire, il faut une instance de la fabrique de sessions (`SessionFactory`) qui est configurée en fonction de `hibernate.cfg.xml`.

C'est le *listener* dont nous parlions avant. Il sera invoqué au déploiement de l'application (et non pas à la première requête, chose souvent rencontrée dans les web applications…)

Pour l'essentiel :

```
public class HibernateListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        HibernateUtil.getSessionFactory();  // call static initializer
        // populate database
        // etc.
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        HibernateUtil.getSessionFactory().close();
    }
}
```

Vous pouvez consulter l'implémentation de la classe `HibernateUtil`. Elle ne contient rien d'intéressant si ce n'est un `new Configuration().configure()` au milieu qui va lire le fichier de configuration d'Hibernate. L'objet retourné par la méthode permet alors de récupérer la `SessionFactory`. Celle-ci en main il suffit d'appeler `sessionFactory.openSession()` pour créer une session vers la base de données.

Notons l'importance de refermer la session. Les logs informent rapidement du problème en avertissant qu'une fuite de mémoire est probable. Il faut bien faire attention à traiter les exceptions correctement pour éviter cette situation.

2.4.6 Requêtes HQL

La plupart du temps, Hibernate fait bien son boulot et il n'y a pas grand chose à lui dire. Par exemple :

```
// Returns null if not found
public static Pizza findPizza(Session session, String pizzaName) {
    return (Pizza) session.load(PizzaEntity.class, pizzaName);
}
```

`pizzaName` doit être la clé de la table pour que ça fonctionne. Si par hasard nous avions un autre ID mais que nous savons que le nom de la pizza est unique, nous aurions pu faire

```
return (Pizza) session.createCriteria(PizzaEntity.class)
                      .add(Restrictions.eq("name", pizzaName))
                      .uniqueResult();
```

Dans certains scénarios plus compliqués, Hibernate est un peu perdu et on se retrouve avec des informations manquantes, il n'appelle pas les *setters* de certaines relations. Voici un *workaround* possible :

```
// Returns null if not found
public static Order findOrder(Session session, long orderId) {
    OrderEntity order = (OrderEntity) session.load(OrderEntity.class, orderId);
    List list = getAssociatedPizzas(session, order.getId()).list();
    Set<PizzaToOrderAssocEntity> associatedPizzas = new LinkedHashSet<>();
    for (Object o : list) {
        associatedPizzas.add((PizzaToOrderAssocEntity) o);
    }
```

```
    order.setPizzas(associatedPizzas);
    Order ret = order;
    return ret;
}

private static Query getAssociatedPizzas(Session session, long orderId) {
    return session
            .createQuery("from PizzaToOrderAssocEntity where assocId.order.id = :orderId")
            .setParameter("orderId", orderId);
}
```

Notons dans cette deuxième méthode la technique mise en place pour passer des paramètres et se protéger d'attaques XSS. On aurait aussi pu utiliser un *Critère* comme mentionné plus haut.

Ce langage de requêtes qui ressemble vaguement à du SQL est dénommé HQL pour, vous l'aurez deviné, le *Hibernate Query Language*. Son utilisation peut être un avantage comme un inconvénient. Pour réaliser des optimisations ou requêtes très compliquées ou tirant parti de certaines spécificité du SGBD, il est possible d'écrire des requêtes en SQL mais c'est comme le code natif, à n'utiliser qu'en dernier recours car le code est alors lié à une implémentation de SGBD particulière et cela complique la maintenance de l'application et la lisibilité du code.

2.4.7 `SessionFacade`

Les méthodes précédentes ne sont pas insérées dans les classes ressources du web service mais placées dans une classe `SessionFacade` qui masque une certaine complexité du code pour l'accès aux données. À part les deux courtes méthodes de `HibernateUtil` c'est une des seules classes à contenir du code spécifique à Hibernate. Pas grand chose d'ailleurs, juste `Query` et `Session`.

2.4.8 Clés composées

Nous avons d'un côté des pizzas et d'un autre côté les commandes avec l'adresse de livraison.

Finalement, un item de la commande ce n'est rien d'autre qu'une entité affublée de quelques informations supplémentaires comme la taille de la pizza commandée et la quantité. C'est un cas classique traduisible facilement dans le modèle entité-association étudié au cours *Bases de données 1.*

Par-contre, du point de vue JPA, c'est un vrai casse-tête !

Dans la classe `Order` :

```
    @MapsId("assocId")
    @OneToMany()
    public Set<PizzaToOrderAssocEntity> getPizzas() {
        return pizzas;
    }
```

À propos, un paramètre permet de demander un chargement façon *eagerly* (soit le contraire du *lazy loading*) des pizzas, mais cela n'a pas réglé le problème mentionné à la sous-section précédente.

Nous avons dû recourir à une entité d'association (`PizzaToOrderAssocEntity` que nous aurions aussi pu appeller `OrderItemEntity` par exemple) étant donné que nous avons ajouté deux champs qui ne font pas partie de la clé de l'association. Une chose que la classe `PizzaToOrderAssocEntity` doit contenir, c'est ça :

```
    @EmbeddedId
    public PizzaToOrderAssocId getAssocId() {
```

```
        return assocId;
    }
```

Il n'y a pas d'autre champ portant l'annotation `@Id`. L'`EmbeddedId` est la solution à adopter pour les clés composées.

Et enfin, `PizzaToOrderAssocId` :

```
@Embeddable
public class PizzaToOrderAssocId implements Serializable {
    @ManyToOne(targetEntity = OrderEntity.class, fetch = FetchType.EAGER)
    @JoinColumn(name = "order_id")
    public Order getOrder() {
        return order;
    }

    //@Id
    @ManyToOne(targetEntity = PizzaEntity.class, fetch = FetchType.EAGER)
    @JoinColumn(name = "pizza_name")
    public Pizza getPizza() {
        return pizza;
    }
}
```

Voilà un code non trivial qui nous a donné quelques sueurs froides. S'il fonctionne parfaitement et fait bien ce que l'on attend de lui, le système possède un gros défaut : en utilisant la sérialisation JSON que nous avons mis en place, le champ `assocId` apparaîtra aux clients, ce qui n'est ni élégant ni même utile.

Conclusion : Il vaut peut-être mieux laisser le framework utiliser des clés qui sont des numéros de séquence et imposer des contraintes sur les tables pour respecter les restrictions du modèle. Les *best practices* voudraient pourtant que les contraintes soient exprimées grâce à la structure des tables autant que possible.

2.4.9 Une clé qui n'est pas un numéro de séquence

Puisque nous parlons de numéros de séquence, nous avons déjà remarqué que les frameworks tels que *django* utilisent un ID numérique à tout bout de champ. Un bon design de base de données en décourage pourtant l'usage dans la plupart des cas. On peut facilement, avec JPA, demander d'autres types d'identifiant pour les entités.

Voyons par exemple comment définir une chaîne de caractères comme clé d'une table :

```
    @Id
    @Basic(optional = false)
    @Column(name = "name", unique = true, nullable = false)
    public String getName() {
        return name;
    }
```

Une chose importante à noter, si avec un numéro de séquence nous trouvons le code suivant :

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public long getId() {
        return id;
    }
```

il est important de remarquer que dans le premier cas non seulement il faut introduire manuellement les contraintes garantissant l'unicité de la clé et le fait qu'elle soit non nulle, mais aussi ne pas oublier de toujours « setter » une valeur, sinon, comme prévu, l'insertion d'une telle entité dans la table va échouer.

Petite astuce concernant l'ID auto-généré. L'auto-génération se fait au moment où l'entité est enregistrée dans la table. Pour mettre l'objet local représentant l'entité à jour avec le numéro de séquence attribué, on peut appeler `session.flush()` qui aurait pour effet de mettre à jour tous les objets concernés par la transaction en cours.

2.4.10 Gestion des transactions

Des transactions, nous en avons dans notre application. On aurait pu les éviter, mais nous trouvions intéressant d'étudier comment celles-ci pouvaient être mises en place dans un web service REST. Dans tous les cas, notons que SOAP est bien plus adapté dans de tels cas de figure puisque le support des transactions fait partie intégrante des spécifications.

Une solution que l'on trouve dans la littérature pour REST consiste à suivre l'algorithme suivant, ici en version simplifiée :

1. Le client indique au serveur qu'il veut commencer une nouvelle transaction. Il obtient en retour un identifiant pour la transaction qu'il devra utiliser ensuite. Cet identifiant devrait, dans l'idéal, être transmis sous la forme de l'en-tête HTTP `Location` puisque c'est bien cette information que ce dernier décrit.
2. En utilisant cet identifiant de transaction et l'URL indiquée à l'étape précédente, le client réalise des opérations dans le cadre de la transaction.
3. Lorsqu'il a terminé le client peut, à choix :
   a Annuler la transaction *(rollback)* ;
   b Clore la transaction *(commit)*.

Nous ne détaillerons pas ici comment nous avons géré les commandes commencées mais pas terminées. Vous pouvez néanmoins consulter le code de la classe `OrderManager` si cela vous intéresse.

Le petit soucis à régler du point de vue du serveur est de poser un délai *(timeout)* pour éviter de garder éternellement des transactions ni confirmées, ni annulées. Il s'agirait ensuite de réaliser une sorte de ramasse-miettes qui irait de temps en temps faire un *rollback* des transactions restées inactives trop longtemps et fermer les sessions correspondantes.

2.5 JSON

Nous avons utilisé la bibliothèque Gson de Google pour sérialiser les objets en JSON. Un problème que l'on rencontre immédiatement avec cette solution est que quand on pense récupérer les classes entité que l'on a mis dans la base de données lorsqu'on interroge cette dernière, on récupère en réalité une sous-classe implémentant `HibernateProxy` que Gson ne sait pas sérialiser.

Dans la classe `GsonForHibernate` qui est un Singleton, nous gardons une instance permanente de l'objet Gson configuré à notre sauce pour prendre en charge les entités.

Le secret consiste à enregistrer un *TypeAdapter* qui va aller visiter les objets référencés par l'entité de manière récursive. Cette approche amène par ailleurs différents problèmes ou limitations :

— Le JSON généré correspond *exactement* aux membres des entités. On est parfois amené à devoir écrire

du code dont la structure n'est pas idéale, souvent à cause de conventions adoptés par des frameworks. Voir `"assocId"`.

— La récursivité naïve va générer une boucle infinie en cas de cycle. Une solution consiste à casser cette récursivité en enlevant un attribut de la sérialisation. Une manière rapide de le faire est de déclarer l'attribut concerné comme `transient` mais de meilleures solutions existent [1].

3 Réalisation du frontend

L'interface graphique devra permettre de choisir des pizzas et de passer commande.

3.1 Technologies utilisées

Pour réaliser cette interface, nous avons choisi d'utiliser principalement AngularJS couplé à HTML5, CSS et consorts.

3.2 Incompatibilité avec le backend

L'interface utilisateur n'est pas terminée. Il reste des points d'amélioration comme le visuel ou encore quelques fonctionnalités présentes sur le serveur comme l'annulation de commande, l'ajout de pizzas et d'ingrédients.

Il y a également eu des changements à faire au niveau du code du frontend pour qu'il soit compatible avec le backend.

3.2.1 Changer les par défaut

Le but est de changer le type de contenu via les en-têtes transmis au serveur. Le code à insérer est le suivant :

```
$http.defaults.headers.post["Content-Type"] = "application/x-www-form-urlencoded";
```

Les autres possibilités est le format `form-data` et l'envoi des données sous forme brute. On peut modifier le *Content-Type* pour chaque requête individuellement si nécessaire.

Ce changement n'est pas suffisant. AngularJS ne peut pas deviner qu'il doit transformer le dictionnaire des *data* sous la forme « URL encodée ». Il faut le faire explicitement. AngularJS nous offre fort heureusement un *hook* pour le faire :

```
$http.defaults.transformRequest = function(obj) {
        var str = [];
        for(var p in obj)
            str.push(encodeURIComponent(p) + "=" + encodeURIComponent(obj[p]));
        return str.join("&");
    };
```

On critique souvent Javascript mais vous avouerez qu'il est difficile de faire plus court et plus clair.

4 Organisation

Cette section décrit la manière dont nous nous sommes organisés pour mener à bien ce projet.

1. http://stackoverflow.com/a/14489534/753136

4.1 Répartition des tâches

Piller Bryan	— Parties architecture et conception du projet — Création des beans
Mathieu Clément	— Implémentation du backend — Configuration Maven
Les deux	— Implémentation du frontend — Rédaction du rapport

4.2 Environnement de développement

— IntelliJ IDEA
— Eclipse
— Git
— GNU/Linux
— Tomcat
— Hibernate H2 in-memory database
— Chromium Developer Tools
— Extension de navigateur Postman (REST Client)

4.3 Déploiement de « production »

Vous pouvez trouver respectivement le web service et le frontend aux adresses suivantes :

`http://langid.tic.eia-fr.ch:8080/pizzaorders-ws/rest/application.wadl`

`http://langid.tic.eia-fr.ch:8080/pizzaorders-frontend`

Note : par manque de temps nous avons préféré nous concentrer sur la partie backend, c'est pourquoi le frontend est si simple et son design est si ravageur.

4.4 Maven

Apache Maven s'est avéré pratique pour ce projet. En plus de gérer impeccablement les dépendances, il nous a aussi permis de déployer le WAR sur Tomcat.

4.4.1 Téléchargement des sources

Il est quasiment indispensable de posséder au moins la documentation des APIs lorsque l'on utilise des bibliothèques et composants tiers.

Les sources peuvent aussi s'avérer utiles dans certains cas, lorsque la documentation est lacunaire ou pour comprendre d'où proviennent certains bugs et messages d'erreur.

Les IDE comme IntelliJ IDEA vont chercher les sources au même endroit qu'ils trouvent les artéfacts. On peut demander à Maven de télécharger les sources au moyen de la commande

```
mvn dependency:source
```

4.4.2 Déploiement sur Tomcat

Tomcat dispose d'un « accès script » (en fait via JMX) que l'on peut utiliser pour déployer des web applications. L'autre solution consiste à copier le WAR (ou son contenu, ce qui est par exemple dénommé *exploded WAR* dans IntelliJ IDEA) dans le dossier `webapps` surveillé par Tomcat, un mécanisme commun à de nombreux *servlet containers*.

Voici la configuration à ajouter dans le `pom.xml` :

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <executions>
                    <!-- Run before integration tests -->
                    <execution>
                        <phase>pre-integration-test</phase>
                        <goals>
                            <goal>redeploy</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <path>/pizzaorders-ws</path>
                    <update>true</update>
                    <url>http://localhost:8080/manager/text</url>
                    <!--
                    In Tomcat7 installation directory, add this to conf/tomcat-users.xml:

                    <role rolename="manager-script"/>
                    <user username="my-script-username"
                          password="my-secret"
                             roles="manager-script"/>

                    NEVER give both manager-script and manager-gui roles
                    as the text and JMX interfaces
                    are not protected against CSRF!

                    -->
                    <username>langid-script</username>
                    <!-- or use tomcat.username property -->
                    <password>secret-langid</password>
                    <!-- or use tomcat.password property -->
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Voir le commentaire XML concernant la configuration à ajouter à Tomcat et les précautions à prendre.

Ici nous avons indiqué vouloir que l'application soit déployée dans tous les cas afin de pouvoir exécuter les

tests d'intégration. Si on retire la section `<executions>` on peut alors demander le déploiement sur Tomcat avec `mvn tomcat7:deploy`.

Notez que bien que le plugin ait été développé à l'origine pour Tomcat 7.0, il fonctionne toujours avec Tomcat 8.0.

4.4.3 Propriétés

Il est courant de devoir indiquer une information à de multiples reprises dans le `pom.xml`. Pour éviter la répétition, il est de coutume d'utiliser une *propriété.*

Exemple :

```
<project>
    <properties>
        <jersey.version>2.15</jersey.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-servlet-core</artifactId>
            <version>${jersey.version}</version>
    </dependencies>
</project>
```

Conclusion

Ce travail nous a beaucoup plus. La liberté de choisir le thème et l'absence de restrictions sur les technologies à employer ont fait de ce travail pratique une expérience agréable. Il est très formateur de pouvoir faire soi-même des choix, bons ou mauvais, qu'il s'agit ensuite de devoir défendre et assumer.

Concernant le travail en lui-même, on aurait souhaité passer plus de temps sur le frontend. Malheureusement en raison du rendu de multiples autres travaux et des examens ces dernières semaines, et en ayant utilisé tout le temps accordé sur les heures de cours et bien plus en dehors, quelques heures de plus nous ont tout de même manqué.

Nous nous sommes particulièrement intéressés à la partie serveur et web service. Deux raisons à cela : la première est que nous avons fait le choix d'avoir quelque chose de solide en arrière-plan car c'est souvent la source de beaucoup d'erreurs et c'est là que le risque (intégrité des données, consistance, aspects légaux) est le plus important ; la deuxième raison est que nous avons déjà par le passé créé des interfaces web mais nous manquons d'expérience dans la gestion d'un backend, plus complexe et en ce qui nous concerne, plus intéressant à faire que le choix — aussi cornélien soit-il — de la couleur d'un bouton.

Si vous le permettez, nous souhaiterions émettre la proposition suivante : il pourrait être plus judicieux de débuter le mini-projet plus tôt dans le semestre. En particulier, le TP 11 est finalement très similaire à celui-ci excepté le fait que cette fois-ci le thème était laissé libre.

D'avance, nous nous réjouissons de continuer l'exploration des frameworks et de mettre en pratique dans le futur les connaissances acquises dans le cadre de ce cours. Au terme de ce semestre, nous vous remercions pour vos conseils avisés et vous adressons nos meilleurs sentiments.