# Base de donnée

# Postgresql

Les recherches menées pour obtenir une instance de base de donnée accessible de façons centralisée on pris beaucoup de temps. Nous avons fait des recherches sur des site comme ovh qui fourni un VPS (virtual private serveur) accessible depuis internet. Le problème c’est que les offres de ces sites ne sont pas suffisamment détaillée et les caractéristique technique insuffisantes. Chez ovh par exemple nous n’avons pas eu de détails exacte sur ce qu’est un core

Nous avons pris une année de test chez amazon pour un serveur cloud afin d’éberger une base de données centralisé, le serveur se trouve en Amérique et de ce fait les temps de latences sont considérables. Nous avons ouvert un ticker chez infomaniaque pour obtenir également une machine sur le cloud, il nous font une offre gratuite uniquement pour un mois. Nous avons donc décliné. Comme nous ne développons pas de services pour le logiciel il est impératif qu’un serveur dans l’intranet de l’école ou en suisse soit accessible car le trafic entre les clients et le serveur de base de donnée est conséquent et pas optimisé par l’intermédiaire d’un service. Comme nous accédons pas au données à travers un service web mais directement avec un Object Relational Maping intégrée dans les clients nous ne pouvons pas optimiser le trafic de données et limitée la quantité d’information qui transite entre le serveur et le client.

Nous avons réalisé un script postgresql à partir du model UML générer avec MySQL, nous avons choisi de simplifier le script au maximum pour ne pas avoir des soucis d’adaptation entre les deux systèmes de base de données. Etant donnée que nôtre schéma de base de données ne comporte pas des relation 1,1 à 1,1 ou 1,1 à 1,N ou 1,N à 1,N. Nous n’avons pas nom plus besoins d’activer ou désactiver des contraintes temporairement d’où le fait que nous ne les renommons pas comme nous n’allons pas travailler dessus.

# SQLite

Nous avons installer un browser sqlite pour générer une base de donnée d’après un script SQL et le tester avec Hibernate et JPA

Le browser SQLite que nous avons utilisé peut se télécharger à l’adresse suivante : <http://sqlitebrowser.org/>

Suit à l’utilisation de SQLite avec hibernate nous avons rencontrer plusieurs problème pendant la mise en place du projet template. Ces problèmes provenait essentiellement des librairies qui ne supportent pas la base de donnée. Suite à la consultation de la page web <https://stackoverflow.com/questions/17587753/does-hibernate-fully-support-sqlite> Nous avons choisi de prendre Derby pour notre projet, Derby étant développée directement en Java

# Projet template

La création d’un projet Template avec maeven nous à pris du temps, nous avons du importer toutes les dépendances afin qu’Hibernate puisse fonctionner correctement avec la version 9 de java. Les premier projet template n’était pas fait avec maven, cela afin de comprendre comment configurer les mapping et hibernate. Comme nous avons deux sources de données différentes PostgreSQL et Derby nous avons passé du temps à comprendre comment les intégrer au projet.

Avec netbeans l’intégration de la spécification JPA et hibernate se fait beaucoup plus aisément que sur intellj, netBeans met directement tous les fichiers de mapping, de configurations aux bonnes place avec toutes les dépendances nécessaire à la bonne utilisation d’hibernate.

# Projet template

Nous avons créer un projet template pour hibernate afin d’avoir toutes les bonnes dépendences et la bonne configuration. Le première choses qu’on a essayer de réaliser sur ce projet template est l’utilisation et la gestion d’un même shéma de base de donne sur deux bases de données diférentes. En gros utiliser le mapping d’un même modèle d’entitié sur deux système de base de donnée différents

Avant de créer ce projet Template nous avons installé deux système de base de donnée différent, PostgreSQL et Derby, ensuite nous avons exécuté le même script de création de tables.

Pour la création de ce projet template nous avons fait un nouveau projet maven appelé MoneyThoring ensuite dans project structure nous avons ajouter un nouveau module nomé hibernate, nous avons télécharger la librairies hibernate depuis l’utilitaire d’ajout du module. Afin de configurer et générer les classe à partir de la base de données nous avons utiliser l’outils persistance fourni dans intellij. Attention cette outils ne fait pas tout automatiquement, par la suite nouverons que nous devrons ajouter des choses dans les xml. Avec l’outil de persistances nous avons générer le modèle depuis le shéma de bases de données. Dans le dossier ressources nous allons placer une fichier de mapping et le fichier de configuration pour hibernate.

hibernate.pgsql.cfg.xml pour le fichier de configuration d’Hibernate pour postgresql

hibernate.pgsql.mapping.xml pourle fichier de mappage des entitée d’Hibernate pour postgresql



Une fois la configuration effectuées pour postgresql nous avons une arborecences comem celle-ci-dessous



Nous avons procéder déjà a un premier test de hibernate cer nous n’avons pas encore toutes les librairies nécessaire au bon fonctionnement. Nous pouvons ajouter un fichier de maping d’entié pour jpa que nous allons faire par la suite pour tester mas d’abor nous teston juste avec hibernate et un petit bout de code qui utilise une session.

Voici le petit code de tests que nous avons utilisé

**public class** Main {  
 **private static final** SessionFactory ***ourSessionFactory***;  
  
 **static** {  
 **try** {  
 Configuration configuration = **new** Configuration();  
 configuration.configure();  
  
 ***ourSessionFactory*** = configuration.buildSessionFactory();  
 } **catch** (Throwable ex) {  
 **throw new** ExceptionInInitializerError(ex);  
 }  
 }  
  
 **public static** Session getSession() **throws** HibernateException {  
 **return *ourSessionFactory***.openSession();  
 }  
  
 **public static void** main(**final** String[] args) **throws** Exception {  
 **final** Session session = *getSession*();  
 **try** {  
 Query query = session.createQuery(**"from mt.dal.entities.VoitureEntity"**);  
 List listess = query.list();  
 Iterator iterator = listess.iterator();  
 **while** (iterator.hasNext()) {  
 VoitureEntity user = (VoitureEntity) iterator.next();  
 System.***out***.println(user.getEmail());  
  
 }  
 }**finally** {  
 session.close();  
 }  
 System.***out***.println(**"End"**);  
 }  
}

Une fois que l’on a executer le petit code nous avons du importer plusieurs librairies avec maven pour éviter des problème de compatibilité et pour ajouter des dépendence manquante

Voici ce que nous avons obtenu pour le première exécution



Voici plus d’information pour cette erreur https://stackoverflow.com/questions/43574426/how-to-resolve-java-lang-noclassdeffounderror-javax-xml-bind-jaxbexception-in-j/46484873

Pour résoudre cette exception nous avons importé la libraire suivante :

javax.xml.bind

com.sun.xml.bind

en allant les prendres sur le site mvnreposistory.com

voici les ajouts au pom.xml à faire.

<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.3.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-core -->

<dependency>

<groupId>com.sun.xml.bind</groupId>

<artifactId>jaxb-core</artifactId>

<version>2.3.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-impl -->

<dependency>

<groupId>com.sun.xml.bind</groupId>

<artifactId>jaxb-impl</artifactId>

<version>2.3.0</version>

</dependency>

Nous devons spécifier le fichier xml de configuration dans le session configure car nous allons utiliser deux fichier de configuration différent un pour postgresql et l’autre pour Derby

Deuxième exécution nous obtenons des erreurs issue d’un manque de configuration dans le fichier de cofiguration hibernate

Se sont principalement les propriétes telle que le username et le password de la base de donnée

<**property name="connection.username"**>username</**property**>  
<**property name="connection.password"**>mdp</**property**>

Nous avons lancé le projet une troisièmes fois et avons obtenu l’erreur suivante



Pour remédier à cette erreur nous avons importé la librairies suivante :

<!-- https://mvnrepository.com/artifact/javax.activation/activation -->

<dependency>

<groupId>javax.activation</groupId>

<artifactId>activation</artifactId>

<version>1.1.1</version>

</dependency>

Puis ensuite ajouter le bon driver et dernier en date pour postgresql

<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.2.2</version>

</dependency>

Voici l’exécution du programme



Dans la suites nous allons réitérer les opérations pour ajouter et faire fonctionner hibernate avec derby. Et nous finiront par tester jpa

Pom.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>MoneyThoring</**groupId**>  
 <**artifactId**>MoneyThoring</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**dependencies**>  
  
 *<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->* <**dependency**>  
 <**groupId**>javax.xml.bind</**groupId**>  
 <**artifactId**>jaxb-api</**artifactId**>  
 <**version**>2.3.0</**version**>  
 </**dependency**>  
 *<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-core -->* <**dependency**>  
 <**groupId**>com.sun.xml.bind</**groupId**>  
 <**artifactId**>jaxb-core</**artifactId**>  
 <**version**>2.3.0</**version**>  
 </**dependency**>  
 *<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-impl -->* <**dependency**>  
 <**groupId**>com.sun.xml.bind</**groupId**>  
 <**artifactId**>jaxb-impl</**artifactId**>  
 <**version**>2.3.0</**version**>  
 </**dependency**>  
 *<!-- https://mvnrepository.com/artifact/javax.activation/activation -->* <**dependency**>  
 <**groupId**>javax.activation</**groupId**>  
 <**artifactId**>activation</**artifactId**>  
 <**version**>1.1.1</**version**>  
 </**dependency**>  
 *<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->* <**dependency**>  
 <**groupId**>org.postgresql</**groupId**>  
 <**artifactId**>postgresql</**artifactId**>  
 <**version**>42.2.2</**version**>  
 </**dependency**>  
  
  
  
 </**dependencies**>  
  
</**project**>

hibernate.pgsql.cfg.xml

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-configuration PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>***<**hibernate-configuration**>  
 <**session-factory**>  
 <**property name="connection.url"**>jdbc:postgresql://192.168.43.41:5432/moneythoring</**property**>  
 <**property name="connection.driver\_class"**>org.postgresql.Driver</**property**>  
 <**property name="connection.username"**>postgres</**property**>  
 <**property name="connection.password"**>Google2018$</**property**>  
 <**mapping class="mt.dal.entities.VoitureEntity"**/>  
 <**mapping resource="hibernate.pgsql.mapping.xml"**/>  
  
  
 *<!-- DB schema will be updated if needed -->  
 <!-- <property name="hbm2ddl.auto">update</property> -->* </**session-factory**>  
</**hibernate-configuration**>

Insertion dans postgresql

INSERT INTO moneythoring.client(

username, email, password, isactivated, activationkey, salt)

VALUES ( 'toto', 'toto@mt.com', 'pssword', true, 'sdafkey','sdafsalt');

# DAL

Nous avons créer un package DAL pour data access layer, cette couche va nous permettre d’interagir avec une base de donnée centralisé qui est postgresql et une base de donnée embaquée Derby.

Voici l’arborescence des packages



## dal.dalexception

Le pakage dalexception contiendra les exceptions que la couche d’accès aux données renverra au couches supérieur. Ainsi on pourra encapsuler toutes les exceptions générer par Hibernate et différentes parties du code de la couche d’accès au donnée. Depuis ce package nous pourront également définir les traitements à appliquer en cas d’exception.

## dal.entities

Ce package rassembles toutes les entités utiliser par Hibernate l’ors du mapping, ce package contiens deux sous, comme leurs nom l’indique, l’un regroupe les entités pour le mapping de derby et l’autre les entités pour le mapping de PostgreSQL. Chaque entités dans ces deux packages implémentes toutes les interfaces dans qui se trouve dans le package ientites. Nous avons fait ce choix pour factoriser plus facilement le code et exposer directement des interfaces à mapper au couches supérieurs, ainsi les couches supérieur n’on pas besoins de connaitre l’implémentation des entités qu’elles manipulent et de savoir si se sont bien des entités derby ou PostgreSQL. Pour le mappage les couches supérieurs utiliseront simplement un mapper qui convertira un objet d’un modèle d’entité provenant de la couche supérieur en le mappant soit à un objet du modèle d’entité derby ou du modèle d’entité PostgreSQL.

## dal.ientities

Ce package regroupe toutes les interfaces d’ont les entitées derby et pgsql implémentes, ces interfaces seront utilisées par les couches supérieures pour manipuler les objets

## dal.irepositories

Ce package contiens toutes les interfaces qui expose les méthode d’accès aux données, donc les opérations CRUD par exemple. L’implémentation de ces repositories se fait une fois pour derby et une fois pour PostgreSQL bien que le code reste assez semblable.

## dal.repositories

Ce package contiens deux sous package l’un appelé derby et l’autre pgsql. Ces deux sous packages contiennent des classe qui implémentes l’accès aux données.

## dal.orm

Le package dal.orm contiens en quelque sorte une unitOfWork qui sera charger d’instancier une seul fois chaque repositories, de les stocker et de les retourner pour pouvoir utiliser les méthodes d’accès aux données. L’objet qui regroupe tous ces repositories sera charger de les instancier avec la même session et de les faires travailler sur une même connexion. Cela permet de faire travailler tous les repositories en harmonie sur une même session partagées entre tous et dans une même transaction. Plus tard nous allons mettre en place les logique de commit, de rollback et de transaction directement dans cette objet. Nous serons forcer de le faire car Derby ne supporte pas plusieurs connexion simultanée.

Nous avons aussi créer un package de test afin de pouvoir s’assurer du bon fonctionnement et du bon comportement de la couche d’accès au donnée lorsque nous appelons des méthodes. Ceci prend du temps mais semble indispensable car nous mappons à chaque fois les objet au couche plus haut ce qui fait que la référence des objets qui ont été rechercher sont à chaque fois perdu. Donc l’ors d’une modification d’un objet venant d’une couche plus haut nous devons le convertir systématiquement en objet du modèle de la couche d’accès aux données. Les testes prennent également du tems car nous souhaitons voir ce qui se passe avec le chargement paresseux et les propriété de navigation et de plus au moment du mappage des objets entre la couche d’accès au donnée et les couches plus hautes

# Hibernate

Avec Hibernate nous avons perdu du temps pour bien comprendre comment se configurait le mapping xml de chaque entité. Certains choix aussi ont du se faire concernant les types de base ou des types plus évoluée donc simplement leur équivalent en objet. Nous avons choisit de travailler avec des types de bases. Les autres choix concernent les types des propriétés de navigation, car ces propriétés devront retourner les interfaces qu’implémente chaque classe affin de pouvoir garder des méthodes communes aux modèle postgresql et aux modèles derby.

Plus d’information suivront dans cette partie car pour le moment nous n’avons pas encore terminée certaines partie.