Description

Programme Pédagogique National 2013 -- TCM4105C

Compléments d’informatique en vue d’une insertion immédiate.

Objectifs du module

Approfondissement technologique - Découvrir ou compléter l’apprentissage d’une compétence en développement informatique utilisée dans l’environnement professionnel.

Compétences visées

Compétences citées dans le référentiel d’activités et de compétences pour les activités :

- FA1-B : Conception technique d'une solution informatique.
- FA1-C : Réalisation d’une solution informatique.
- FA1-D : Tests de validation d’une solution informatique.

Enseignants

- Romain Giot -- S4p1A
- Marc-Antoine Tessier -- S4p2B
- Hélèna Berger -- S4p2C
- Pierre PECHOT-Maffre-- S4p2D

Format de la matière

- 1 cours d'introduction de 2 heures (01/02/2022) -- Marc-Antoine Tessier
- 5 TD de 4 heures -- tout le monde:
  - 00h30 à 01h00 de cours / 02h30 à 03h00 d'exercices / 30min de pause
  - Idéalement : 1 ou 2 concepts par séance

Attention, ça va TRÈS vite, il est important de suivre et faire les exercices sous peine d'être perdu rapidement

Contenu des séances

Nous essaierons de suivre le planning suivant

TD 1 : Prérequis / Validation de Bean

- Semaine du 31 Janvier 2022
- Prérequis Java :
  - Nécessaire car il faut maîtriser Java pour la suite
  - Distribution des pré-requis
  - Temps estimé : 3h (peut être moins car Java doit être maîtrisé)
- Validation :
  - Présentation du cours au tableau
  - Distribution du document exercice
  - But :
    - Utilisation de contraintes et validations
    - Définition de contraintes et validations
  - Temps estimé : 3h

TD 2 : Injection de dépendences et Java Persistence API / ORM

- Semaine du 07 février 2022
- Injection de dépendances :
  - Présentation cours + exercices Injection de dépendances
  - Présentation de Google Guice
  - Temps estimé : 2h
- JPA :
  - Présentation de l'intérêt de JPA
  - Utilisation de Hibernate et Google Guice

- Distribution du code `room-manager` et réalisation des exercices
- Temps estimé : >2h Le dernier exercice peut être terminé lors de la séance suivante

TD 3 : JPA fin

- Semaine du 14 février 2022
- JPA

  - Fin de l'exercice de la séance précédente

TD 4 : Conteneurs

- Semaine du 28 février 2022
- Conteneur

  - Présentation cours : Conteneur + Conteneur WEB - Optionnel : JSF, JSP, Ming - Obligatoire : Authentification, Injection de dépendances

- Discussion avec les étudiants pour revenir sur les points non maîtrisés
- Terminaison des exercices

TD 5 : API rest

- Semaine du 07 mars 2022

Notation

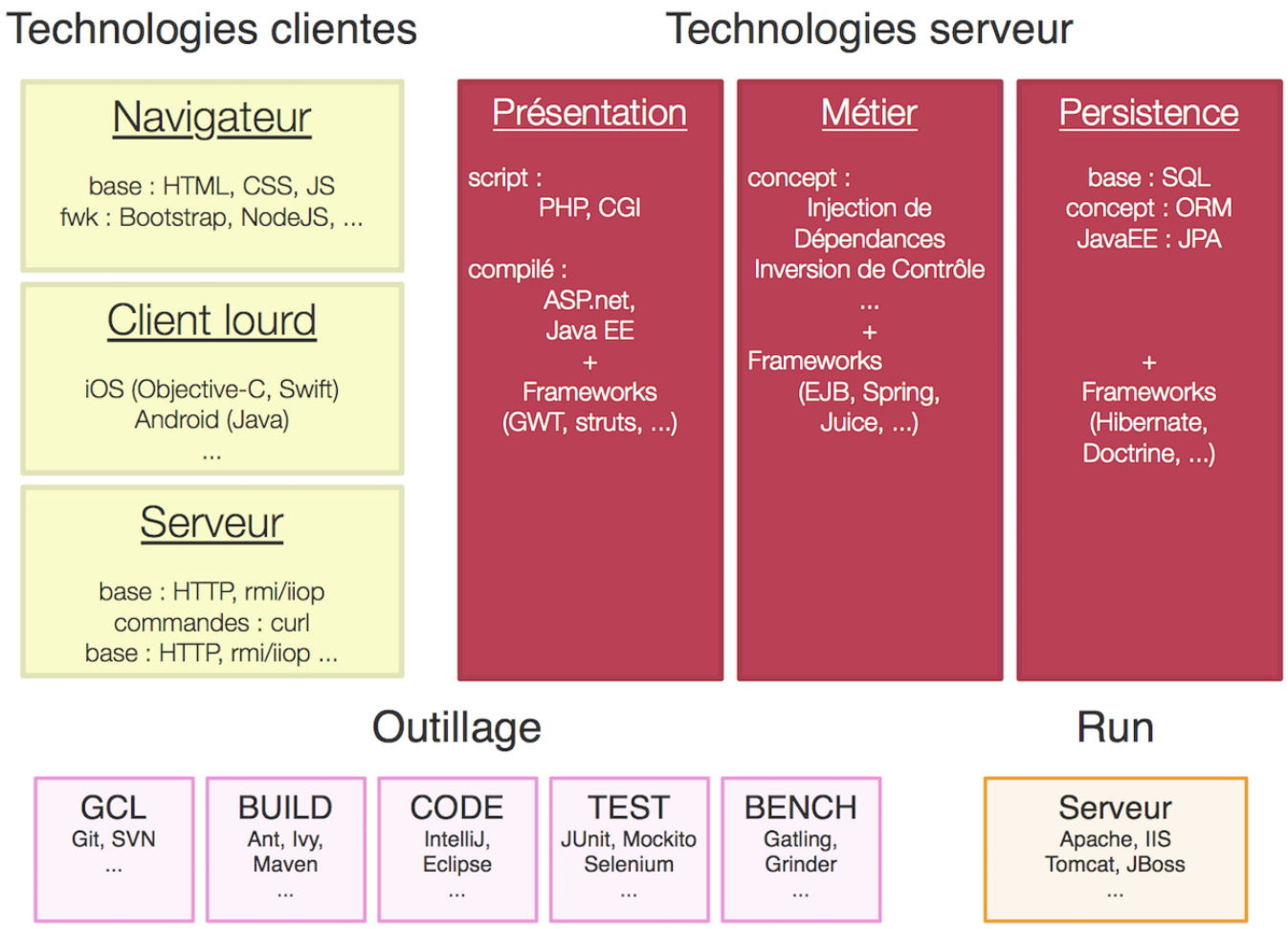
- 1 note de TD :
  - Travail effectué durant les 5 séances.
  - commits réguliers tout au long des séances.
  - **Le travail est impérativement en binôme.** Un unique trinôme est accepté dans les groupes impaires.

- 1 note de devoir sur table (**1 feuille A4 manuscrite autorisée**) :
  - Questions de cours
  - Exercices de développement

- Analyse de traces

Prérequis

De nombreux prérequis sont nécessaires pour pouvoir faire les exercices et suivre le cours. La figure suivante vous permet de situer dans quelle partie du *développement web* vous travaillez.



Les minimums de l'administration système

Rappels Système S1

Plusieurs outils manipulés pour faire du Java EE lisent leur configuration dans des variables d'environnement. Il faut donc savoir manipuler les variables d'environnement.

- Les variables sont stockées dans le ficher `~/.bashrc`
- Une variable est lue en préfixant son nom par `$` (ex. `echo $HOME` affiche le contenu de la variable `$HOME`)

- La valeur d'une variable est spécifiée avec la syntaxe `export VARIABLE="LES VALEURS"` sans espaces avant/après le =
- Lorsqu'une variable contient une liste de chemins (comme `PATH`), ces chemins sont séparés par le caractère `:`
- La modification du `.bashrc` n'est pas prise en compte instantanément ; il faut soit ouvrir un nouveau terminal, soit faire `source ~/.bashrc`

Variables à connaître

- `HOME` : répertoire personnel de l'utilisateur (ne saisissez jamais le chemin manuellement)
- `JAVA_HOME` : répertoire d'installation de la machine virtuelle Java
- `PATH` : liste des chemins parcourus par le shell lors de sa recherche d'exécutables

Vous pouvez créer des alias de commande à l'aide de la syntaxe `alias nom="la commande a executer"`

Java 7 (minimum)

Au fil du temps, `Java` évolue pour être plus puissant et agréable à utiliser. Les évolutions suivantes sont nécessaires pour le Java moderne :

Utilisation d'annotations

1. Structuration des sources

```
exo
 |
 |- src
  |
  |- fr
   |
   |- iut
    |
    |- DeprecatedAnnotationSample.java
    |- Main.java
```

1. Code source

Fichier `DeprecatedAnnotationSample.java` qui utilise une annotation qui indique la dépréciation de méthodes :

```
package fr.iut;

/**
* This class declares a deprecated method.
*/
```

```
public class DeprecatedAnnotationSample {
    /**
     * @deprecated
     * explanation of why deprecatedMethod() was deprecated
     */
    @Deprecated
    public void deprecatedMethod() {
        System.out.println("calling deprecatedMethod is deprecated");
    }

    /**
     * explanation of nonDeprecatedMethod()
     */
    public void nonDeprecatedMethod() {
        System.out.println("calling nonDeprecatedMethod is well supported");
    }
}
```

Fichier `Main.java` qui utilise les méthodes dépréciées :

```
package fr.iut;

/**
    * Main entry point for demo
    */
public class Main {
    public static void main(String[] args) {
        DeprecatedAnnotationSample sample = new DeprecatedAnnotationSample();
        sample.deprecatedMethod();
        sample.nonDeprecatedMethod();
    }
}
```

1. Commandes de compilation et d'exécution

```
mkdir -p target/classes
cd src && javac -d ../target/classes fr/iut/*.java
cd ..  && java -classpath target/classes fr.iut.Main
```

Exercice (15 min)

Objectif : Comprendre la construction d'un projet `java`. Coder les 2 classes (`Main` et `DeprecatedAnnotationSample`) et utiliser `javac` et `java` en ligne de commande.

- créer les deux classes (attention à l'organisation dossier/package)
- compiler manuellement sans utiliser d'IDE `javac...`
- exécuter manuellement sans utiliser d'IDE `java...`
- observer l'impact des annotations (voir le warning)

Types Génériques et boucles `for-each`

Equivalent des *template* en C++, ils évitent le recours à `java.lang.Object` et au casting

```
// Sans génériques
List oldNames = new ArrayList(); // Conteneur objets quelconques
oldNames.add("Jean");
oldNames.add("Raoul");

Iterator i = oldNames.iterator(); // Boucles objets avec itérateurs
while (i.hasNext()) {
    String name = (String)i.next(); // Conversion nécessaire
    System.out.println(name);
}


// Avec génériques
List<String> names = new ArrayList(); // Conteneur String
names.add("Bobine");
names.add("Ducable");
for (String name : names) { // Boucle String sans itérateurs
    System.out.println(name);
}
```

Autoboxing / Unboxing

Les types primitifs ne peuvent pas être manipulés comme des objets. Il faut donc les encapsuler pour les manipuler comme des objets grâce à leur proxie. Cependant, sans `autoboxing` et `unboxing` la manipulation de ces proxies est fastidieuse.

```
Integer i = new Integer(9); // Création manuelle du proxie
Integer l = 9; // Création automatique du proxie (autoboxing)
```

```
Integer k = new Integer(4);
int l = k.intValue(); // Récupération manuelle de la valeur
int m = k;            //  Récupération automatique de la valeur (unboxing)
```

L'outillage

Gestionnaire de projet Java - Maven

Les projets `Java EE` sont relativement complexes et comportent de nombreuses dépendances. `maven` est un outil, parmi tant d'autres, qui permet de simplifier la gestion des projets java en

adoptant la devise `convention over configuration` (tant que vous ne spécifiez pas le contraire, vous devez respecter un formalisme précis):

- Il génère automatiquement le squelette `standard` d'un projet `java`
- Il construit le programme final (compilation + archivage)
    - préparation de la construction
        - nettoyage
        - identification de l'emplacement du code source
        - identification de la destination
        - identification transitive des dependances
        - téléchargement et mise en cache des dépendances
        - construction des options de la compilation
    - compilation
    - test unitaires
    - packaging (JAR, WAR, EAR)
    - L'architecture en `plugin` de `maven` facilite l'intégration d'un grand nombre d'utilitaires (cf. plus bas JUnit, Cobertura, etc.)

Voici quelques commandes utiles (**à retenir**) :

- `mvn archetype:generate` pour créer l'arborescence d'un projet
- `mvn compile` pour compiler un projet
- `mvn test` pour lancer les tests unitaires
- `mvn package` pour générer l'archive du projet
- `mvn clean` pour supprimer les fichiers générés
- `mvn site` pour générer le *site* (Javadoc, etc.) du projet ainsi que différents reports (Tests unitaire, Couverture, etc.)
- `mvn assembly:assembly` pour construire le projet final distribuable.

  La page http://maven.apache.org/guides/getting-started/index.html est une courte introduction à maven et contient tout ce que vous avez besoin dans ce TD (**vous devez la lire pour la séance suivante**).

- **BONUS** La configuration de `bash` à l'IUT vous permet bénéficier de l'autocomplétion des arguments de maven à l'aide de la touche `<tab>`.

- **MALUS1** La version de `maven` installée à l'IUT est obsolète ; il est nécessaire que vous installiez votre propre version sur votre compte (ou /tmp si vous automatisez son téléchargement/installation).
- **MALUS2** `maven` (tout comme eclipse/intellij/android studio) télécharge beaucoup de données. Vous devrez faire de la place sur votre compte pour ne pas dépasser votre quota. Convernant `maven`, le cache est dans le dossier `~/.m2`.
- **MALUS3** La configuration `maven` obsolète de l'IUT force à utiliser un dépôt sur un répertoire en lecture seule. Si `maven` refuse de télécharger les dépendences, c'est que cette configuraiton est activée sur votre compte. Supprimez le dossier `~/.m2` pour résoudre le soucis.

Exercice

Objectif : Configurer les variables d'environnement (**impératif pour ne pas perdre de temps aux prochaines séances!**) et créer un premier projet `Maven`

Installer `Maven` sur votre compte personnel

1. Télécharger sur le site https://archive.apache.org/dist/maven/maven-3/3.5.2/binaries/apache-maven-3.5.2-bin.tar.gz le **binaire** de la version **3.5.2** (dans les archives).
2. Décompresser le dans le dossier `~/opt`.
3. Vérifier les pré-requis (`JAVA_HOME` spécifié et `java` dans le `$PATH`) ; corriger le `.bashrc` si besoin.
4. Modifier le `$PATH` pour rajouter la commande `mvn`.
5. Vérifier son bon fonctionnement.

```
$ echo $JAVA_HOME ; java -version ; mvn -version
/opt/oracle-jdk
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d; 2017-10-18T09:58:13+02:00)
Maven home: /mnt/roost/users/rgiot/opt/apache-maven-3.5.2
Java version: 1.8.0_181, vendor: Oracle Corporation
Java home: /opt/jdk1.8.0_181/jre
Default locale: fr_FR, platform encoding: UTF-8
OS name: "linux", version: "4.18.0-0.bpo.1-amd64", arch: "amd64", family: "unix"
```

1. Après la configuration, le `.bashrc` devrait contenir les instructions suivantes :

```
export MVN_HOME=~/opt/apache-maven-3.5.2
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
export PATH=$JAVA_HOME/bin:$MVN_HOME/bin:~/.local/bin/:$PATH
```

1. Dans un nouveau dossier de travail (`~/cours/javaee/prerequis`), créer le premier projet à l'aide de la commande `mvn` (**penser à l'autocomplétion**)

   ```
   $ mvn archetype:generate -DgroupId=fr.iut \
     -DartifactId=exo2 \
     -DarchetypeGroupId=uk.co.markg.archetypes \
     -DarchetypeArtifactId=java11-junit5 \
     -DinteractiveMode=false
   ```

2. Si la commande précédente a échoué, c'est que maven essaye d'installer les fichiers dans un répertoire auquel vous n'avez pas accès. Supprimez son dossier de configuration, le problème devrait être résolu (il est possible que ce problème soit récurrent en chaque début de séance).

   ```
   $ rm -rf ~/.m2
   ```

3. Aller dans le projet `exo2` nouvellement créé et le construire

   ```
   cd exo2
    mvn clean package
   ```

4. Créer un dépôt git pour la matière dans `~/cours/javaee` et y ajouter les fichiers sources. Ayez le réflexe de :

   - toujours ajouter les fichiers sources - jamais ajouter les fichiers générés - commiter régulièrement votre travail avec des messages informatifs - faire des tags en fin de séance pour suivre votre avancement

ATTENTION Avec Java EE et la programmation mobile vous risquez de remplir rapidement vos quotas. Il faudra surement qu'en fin de séance progmobile vous effaciez les dossiers générés par Android Studio et gradle et qu'en fin de séance Java EE vous effaciez les dossiers générés par maven.

Environnement de Développement Intégré pour Java

Bien qu'un simple éditeur de texte suffise pour faire du Java EE, il est préférable d'utiliser un IDE de qualité afin d'accélérer le développement.

Dans le cadre de ce TD, nous avons choisi `IntellijIdea` [1] qui est une alternative à `Eclipse` . A souligner que l'intégration de `maven` dans `IntelliJ` est bien plus performante que dans `Eclipse` [3].

À l'IUT, l'exécutable est `idea` .

Exercice

Objectif : Apprendre à utiliser l'IDE et la documentation. À l'aide de l'IDE, développer une classe utilitaire qui produit une chaîne *localisée* de montant à partir d'un double et de la locale.

1. Nous allons à présent ouvrir le projet précédemment (exo2) créé avec `Maven` dans `IntelliJ` (les screenshots correspondent à une version différente, mais le principe reste très similaire). Il faut dans `IntelliJ` créer un nouveau projet à partir de source existantes (ensuite il faudra indiquer qu'il s'agit d'un projet `Maven`).
2. Créer la nouvelle classe `StringUtil`
3. Ajouter la méthode `prettyCurrencyPrint` dont voici le prototype :

```
public static String prettyCurrencyPrint
            (final double amount, final Locale locale) {
   // faites votre implémentation
}
```

Voici un exemple d'utilisation de la méthode :

```
...
StringUtil.prettyCurrencyPrint(21500.390, Locale.FRANCE);
...
```

Dont la valeur renvoyée est `21 500,39 €`. *Astuce : Jettez un œil à la classe Java* `NumberFormat` *et particulièrement la méthode* `getCurrencyInstance` *et 'format`.* [2]

1. Ne pas oublier la *Javadoc*

Les tests unitaires

`JUnit` [4] est un *framework* `Java` pour la mise en place de tests unitaires (rappelez vous de BOOST en C++). L'utilisation de tests unitaires est primordiale lors du développement d'application Java EE en raison de leur complexité et de la grande difficulté de correction de bugs s'ils sont détectés trop tard. Elle fait même l'objet d'une méthodologie de développement, le `TDD`, *Test Driven Development*, où le développement du test unitaire est préalable au développement de la fonction métier. **Ne pas utiliser de test unitaire pour vérifier son code peut être considéré comme étant une faute professionnelle**.

L'emplacement et l'environnement d'exécution des tests sont déjà prévus dans la template *Maven*.

- dossier `src/test/java`
- dépendance JUnit dans le `pom.xml`

Exemple de test :

```
package com.example.foo;

import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;

/**
 * Tests for {@link Foo}.
 *
 * @author user@example.com (John Doe)
 */
public class FooTest {

    @Test
    public void thisAlwaysPasses() {
      assertTrue(true);
    }

}
```

Exercice

Objectif : Apprendre à implémenter des tests unitaires. Mise en place de tests unitaires associé(s) au code précédent.

1. Créer la classe `StringUtilTest` dans le dossier approprié
2. Proposer une implémentation de test unitaire de la fonction
3. Ajouter le test au dépôt.
4. Exécuter le test unitaire depuis `IntelliJ`
5. Déboguer le test unitaire depuis `IntelliJ` (Il y a 99% de chances que votre implémentation de test unitaire soit mauvaise. Installer un point d'arrêt et observer les variables vous aidera à comprendre pourquoi).
6. Exécuter le test unitaire depuis la ligne de commande (`mvn test`) et `mvn` qui doit retourner -si tout va bien- quelque chose comme ça :

```
[INFO] --- maven-surefire-plugin:2.7.2:test (default-test) @ exo2 ---
[INFO] Surefire report directory: /Users/fred/workspace_iut/teaching.s4_jee/00_Prerequis/exo2/target/surefire-reports

-------------------------------------------------------
   T E S T S
-------------------------------------------------------
Running fr.iut.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.046 sec
Running fr.iut.StringUtilTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.022 sec
```

```
Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.312s
[INFO] Finished at: Mon Feb 09 21:29:13 CET 2015
[INFO] Final Memory: 13M/247M
[INFO] ------------------------------------------------------------------------
```

1. Ajouter la correction au dépôt

Dans la suite du sujet nous ne spécifierons pas la nécessité d'ajouter des fichiers/commiter. À vous d'avoir le réflexe.

Limites

La mise en place de *vrais tests unitaires* est moins simple qu'il n'y parait en raison de différents points :

- L'exhaustivité des tests.
- La pertinence des tests à prouver (quid de tester des getter/setters ?).
- L'identification du périmètre à tester.
- L'isolation du test vis-à-vis des couches inférieures.
- L'environnement de test autonome et équivalent d'une exécution à l'autre.

L'utilisation de `mock objects` facilite l'isolation des couches en simulant et facilitant la vérification du comportement des objets développés. Mockito est un *framework* qui facilite l'utilisation des mock objects. Plus de détails ici : http://blog.soat.fr/2013/07/mockito-ou-comment-faciliter-lecriture-de-tests-unitaires/

Autres tests

Enfin, il convient de distinguer :

1. les tests unitaires (unit testing)
2. les tests fonctionnels (functional testing)
3. les tests d'intégration (ingration testing)

Couverture de tests unitaires

Les outils de mesure de la `couverture des tests` permettent d'identifier les parties du code qui échappent aux tests. Bien souvent, on a tendance à ne tester que les **cas passants** laissant

intactes une bonne partie du code.

`Cobertura` et `Jacoco` sont des outils open source qui facilitent la mise en œuvre de la mesure de la couverture des tests unitaires.

Dans le cadre des TP nous allons utiliser Jacoco

Exercice

Objectif : Mesurer le taux de couverture des tests unitaires.

1. modifier le `pom.xml` pour intéger automatiquement Jacoco à l'exécution des tests unitaires

```
<project ...>
  <build>
    <plugins>
...

        <plugin>
          <groupId>org.jacoco</groupId>
          <artifactId>jacoco-maven-plugin</artifactId>
          <version>0.8.7</version>
          <executions>
            <execution>
              <goals>
                <goal>prepare-agent</goal>
              </goals>
```

```
            </execution>
            <execution>
               <id>report</id>
               <!-- Précise que le plugin s'executera lors du cyle maven "verify" -->
               <phase>verify</phase>
               <goals>
                  <goal>report</goal>
               </goals>
            </execution>
         </executions>
      </plugin>

...
      </plugins>
   </build>
</project>
```

1. Lancer la vérification du projet et la génération du rapport grace à Maven

```
$ mvn verify
```

1. Observer la couverture dans le site sous `target/site/jacoco/index.html` 1. compléter la couverture si nécessaire

Complément de lecture https://www.jacoco.org

À retenir (non exhaustif)

1. Les variables d'environnement Linux, modification du fichier `~/.bashrc`
2. Les commandes de bases de Maven et leur utilité
3. Le rôle du fichier `pom.xml` dans un projet Maven
4. L'architecture de base d'un projet Maven (rôle des répertoires : `site`, `src`, `target`, `test`, ...), structuration du répertoire `src`.
5. L'utilité de Jacoco
6. La création d'un projet Maven spécifique aidé de l'example suivant :

```
$ mvn archetype:generate -DgroupId=fr.iut \
  -DartifactId=exo2 \
  -DarchetypeGroupId=uk.co.markg.archetypes \
  -DarchetypeArtifactId=java11-junit5 \
  -DinteractiveMode=false
```

1. https://www.jetbrains.com/idea/

2. http://docs.oracle.com/javase/7/docs/api/java/text/NumberFormat.html
3. https://eclipse.org/
4. http://junit.org/

Validation des objets

But

- Les applications Java EE doivent manipuler des objets dont les attributs sont renseignés par des humains depuis l'IHM ou des services externes.
- Les humains font NÉCESSAIREMENT des erreurs.
- Les services externes sont potentiellement buggés.
- Il est nécessaire de valider ses données tout au long de leur parcours :
  - saisie HTML/javascript, SWING, ou autre
  - stockage base
- C'est une opération normalement fastidieuse, répétitive, sujette à erreur
- MAIS standardisée (Bean Validation (JSR 303)[1]) et relativement facile à utiliser à l'aide de quelques annotations:
  - `@AssertFalse`, `@AssertTrue`
  - `@DecimalMax`, `@DecimalMin`
  - `@Future`, `@Past`
  - `@Max`, `@Min`
  - `@Null`, `@NotNull`
  - `@Pattern`
  - `@Digits`
  - `@Size`

Le chapitre «Declaring and validating bean constraints» de la documentation contient la liste des contraintes disponibles dans la section 2.3.1 : https://docs.jboss.org/hibernate/validator/5.0/reference/en-US/html/chapter-bean-constraints.html.

Utilisation d'une annotation de validation

- Les contraintes de validation sont exprimées sous la forme d'annotations

- Appliquées sur :
  - types
  - méthodes
  - champs
  - autres contraintes
  - Prise en compte à l'**exécution** et pas à la **compilation**.

- Exemple :

```
import jakarta.validation.constraints.*;

public class PersonToday {
  @NotNull
  private String name;
  @NotNull
  private String surname;
  @NotNull @Min(0)
  private int age;
  @Size(max = 2000)
  private String currentJob;
}
```

Composition de contraintes existantes

- La composition de contraintes permet de créer des contraintes complexes composées de plusieurs contraintes validées individuellement
- S'exprime sous la forme d'une annotation
- Il n'y a pas d'implémentation à écrire
- Une annotation est considérée comme étant une contrainte si :
  - sa politique de rétention contient `RUNTIME`
  - l'annotation est elle-même annotée avec `jakarta.validation.Constraint`

```
@NotNull // Contrainte existante
@Size(min=7) // Contrainte existante
@Pattern(regexp = "...") // Contrainte existante
@Constraint(validatedBy={}) // Pas besoin d'implémenter du code de validation
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Email{
  String message() default "Email incorrect"; // Message si pas valide
  Class<?>[] groups() default {}; // Groupe de validation partiel
  Class<? extends Payload>[] payload() default{}; // Metadonnée de la contrainte
```

```
}
```

Création d'une nouvelle contrainte

Contrainte NotNull

Déclaration de la contrainte :

```
// Cibles de l'annotation
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
//Quand l'annotation est utilisée
@Retention({RUNTIME})
// Inclusion dans la javadoc
@Documented
// Classe qui contient l'algo de validation
@Constraint(validatedBy = NotNullValidator.class)
public @interface NotNull {
  // Message si pas valide
  String message() default "{jakarta.validation.constraints.NotNull.message}";
  // Groupe de validation partiel
  Class<?>[] groups() default {};
  // Metadonnée de la contrainte
  Class<? extends Payload>[] payload() default {};
}
```

Définition de la contrainte :

```
// <Nom annotation, type de paramètre (pour spécialiser la contrainte)>
public class NotNullValidator implements ConstraintValidator<NotNull, Object>{
  public void initialize(NotNull parameters){
  } // appelé avant d'utiliser la contrainte
  public boolean isValid(Object object,
                         ConstraintValidatorContext context){
    return object != null;
  } // Code de validation
}
```

Documentation de `ConstraintValidator` :

```
public interface ConstraintValidator<A extends Annotation,T>

Defines the logic to validate a given constraint A for a given object type T.

Implementations must comply to the following restriction:

    T must resolve to a non parameterized type
    or generic parameters of T must be unbounded wildcard types
```

Autre exemple de contrainte

Cette contrainte est à comparer avec la contrainte *NotNull*.

Déclaration de la contrainte :

```
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = AddressValidator.class)

public @interface Address {
   String  message() default "{address should be less than 500 aphanumerics chars}";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default{};
}
```

Définition de la contrainte :

```
public class AddressValidator implements ConstraintValidator <Address, String> {
   @Override
   public void initialize(Address address) {  }

   @Override
   public boolean isValid(String s,
                  ConstraintValidatorContext constraintValidatorContext) {
      // TO WRITE
      return true;
   }

}
```

Spécialisation de contraintes

Les contraintes peuvent contenir des arguments afin de spécifier leur comportement.

Utilisation de la contrainte :

```
@URL
private String resourceURL;
@NotNull @URL(protocol="http", host="www.iut.u-bordeaux1.fr");
private String httpServerUrl;
@URL(protocol="ftp", port=21)
private String ftpServerUrl;
```

Déclaration de la contrainte :

```
@Constraint(validatedBy={URLValidator.class})
@Target({ElementType.METHOD, ElementType.FIELD,
   ElementType.ANNOTATION_TYPE, ElementType.CONSTRUCTOR, ElementType.PARAMETER})
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface URL{
  String message() default "URL incorrecte"; // Message si pas valide
  Class<?>[] groups() default {}; // Groupe de validation partiel
  Class<? extends Payload>[] payload() default{}; // Metadonnée de la contrainte

  String protocol() default "";
  String host() default "";
  int port() default -1;
}
```

Implémentation de la contrainte :

```
public class URLValidator implements ConstraintValidator<URL, String>{
  private String protocol;
  private String host;
  private int port;

  public void initialize(URL url){
    this.protocol = url.protocol();
    this.host = url.host();
    this.port = url.port();
  }

  public boolean isValid(String value, ConstraintValidatorContext context){
    if (value == null || value.lenght ==0){
      return true;
    }

    java.net.URL url;
    try{
      url = new java.net.URL(value);
    } catch(MalformedURLException e){
      return false;
    }

    if (protocol != null &&
        protocol.length() > 0 &&
        !url.getProtocol().equals(protocol)){
      return false;
    }
    if (host != null &&
        host.length() > 0 &&
    !url.getHost().equals(host)){
      return false;
    }
    if (port !=-1 && url.getPort() != port){
      return false;
    }

    return true;
  }
}
```

Utilisation

- Configuration : `META-INF/validation.xml`
- Bibliothèque : `annotations.jar`

Illustration

Les exemples du projet permettent de comprendre le fonctionnement de la validation de dépendances (à ouvrir avec `Intellij idea`) :

- ajouter des contraintes prédéfinies (via des annotations)
- créer des contraintes
- tester le mécanisme de validation via des tests unitaires (classe `TestConstraints.java` de `Exemples/Validation`)

Lisez attentivement le code pour bien intégrer l'utilisation de ce mécanisme.

Exercices

Objectif

Dans le cadre de cet exercice, on se propose d'implémenter les classes qui modélisent une application de gestion de projet (voir diagramme UML plus loin). Nous ne nous préoccupons que de la validation ; pas du reste.

Organisation

À partir de maintenant, vous devez utiliser un gestionnaire de sources pour stocker vos travaux et les partager avec votre encadrant de TD. La disposition des salles ne permettant pas toujours d'avoir une machine par étudiant, les TD se font en binôme. Dans le cadre de cette matière nous utilisons un dépôt `git` hébergé par le serveur gitlab de l'iut https://gitlab-ce.iut.u-bordeaux.fr/ (pour les étudiants avec C. Johnen et R. Giot) et https://bitbucket.org/dashboard/ (pour les autres étudiants). Vous avez déjà créé votre dépôt en local sur votre compte. Il faut maintenant le synchroniser avec le dépôt distant et le partager avec votre binôme et enseignant.

Le rôle de l'enseignant par rapport à ce dépôt gitLab doit être `reporter`, pour qu'il puisse télécharger vos projets JEE (la remise des projets se fait via `gitLab`).

Choisissez le compte d'un membre du binôme et créez un dépôt en respectant **STRICTEMENT** le nommage suivant : **jee_td_NOM1_NOM2** (avec NOM1 et NOM2 les noms de famille des membres du binôme) et en partageant le dépôt avec votre encadrant de TD et votre binôme. **Pensez à commiter et pousser plusieurs fois par séance votre travail sur le dépôt** (nous ne le

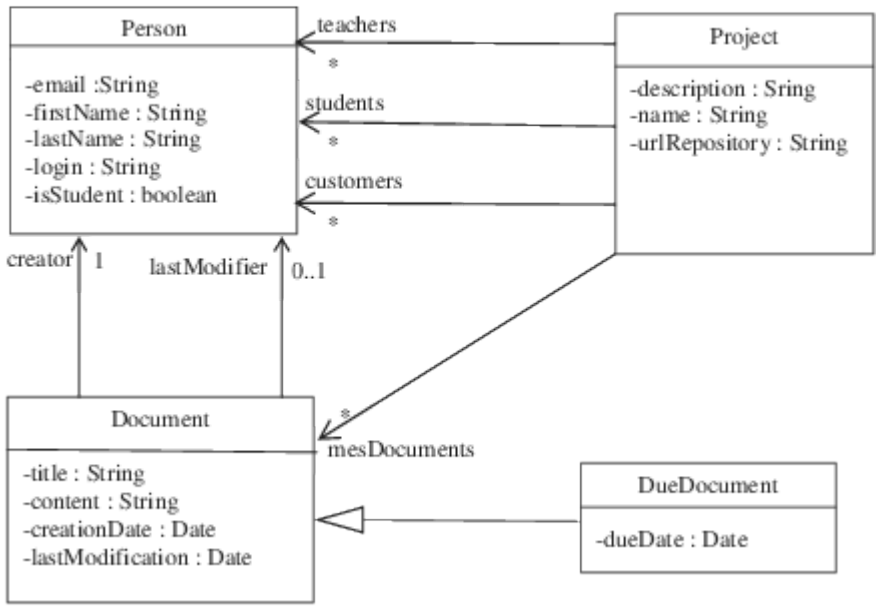
répéterons pas, mais regarder l'historique du dépôt fait parti de la notation TD). Chaque étudiant travaille dans un dossier portant son nom lorque la salle permet d'avoir 1 étudiant par machine.

Avant le TD2, créez un tag *TD1* (nous ne le répéterons pas, pensez à créer systématiquement un tag avant chaque nouveau TD).

```
$ git tag -a  "TD1" # pour ajouter le tag
$ git push --tags # pour l'envoyer sur le dépôt distant
```

Travail à faire et terminer pour la séance suivante

1. Dans votre dépôt créer un dossier *validation* dans lequel vous allez lancer la commande `mvn archetype:generate ...` pour créer votre exercice `validation`.
2. Créez et ajouter les classes suivantes **en utilisant les fonctionnalités de votre IDE** (mettez les dans un paquetage java nommé `bean`). C'est l'IDE qui génère le code de la classe avec ses getters/setters, pas vous (apprenez les fonctionnalités du menu `Code`).



1. Ajoutez les dépendances à la bibliothèque de validation dans le pom.xml :

```
<dependency>
   <groupId>jakarta.validation</groupId>
   <artifactId>jakarta.validation-api</artifactId>
   <version>3.0.1</version>
</dependency>

<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>7.0.2.Final</version>
</dependency>
```

2. Écrivez les classes de validation `validation.Login` et `validation.LoginValidator` qui permettent de valider un identifiant à l'aide des contraintes suivantes :
   - Taille minimum : 2 caractères
   - Taille maximum : 8 caractères
   - Utilisation uniquement des lettres de l'alphabet latin (majuscule ou minuscule)

À noter que cette validation est également implémentable à l'aide de l'annotation `@Pattern` que nous n'utiliserons pas pour cet exercice.

1. Pour chaque classe du projet, écrivez les annotations nécessaires à la validation des données (elles sont triviales à trouver). Le chapitre "Declaring and validating bean constraints" de la documentation contient la liste des contraintes disponibles dans la section 2.3.1 et 2.3.2 (https://docs.jboss.org/hibernate/validator/5.0/reference/en-US/html/chapter-bean-constraints.html) :
   - Gestion temporelle des dates
   - Gestion des champs non nuls
   - Gestion d'emails
2. Vérifiez votre implémentation en écrivant des tests unitaires :
   - Utilisez plusieurs petites fonctions
   - Regroupez les fonctions de manière thématique

```
private  static Validator validator;
private Set<ConstraintViolation<Person>> violations;

@BeforeAll
public static void setUp() {
  ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
  PersonConstraintTest.validator = factory.getValidator();
}
```

```
@Test
public void testNOk() {
  Person p2 = new Person();
  p2.setFirstName("21");
  p2.setLastName("a");
  p2.setEmail("21");
  p2.setLogin(null);
  violations = validator.validate(p2);
```

```
    assertEquals(4, violations.size());
}
```

1. Vérifier votre travail avec `mvn test`
2. FACULTATIF Pour utiliser l'annotation @pattern, comme dans la classe suivante :

```
public class Person  {
 @NotNull
 @Email(message = "email incorrect format")
 private String email;
   // firstName est un chaine de taille quelconque de lettres.
 @Pattern(regexp="[a-zA-Z]*", message="firstName syntax error")
 private String firstName;
  // lastName est un chaine de  2 à 12 lettres.
 @Pattern(regexp="[a-zA-Z]{2,12}",  message="lastName syntax error")
 private String lastName;
 @Login
 private String login;
 private boolean isStudent;
}
```

Il faut inclure dans le fichier de configuration de maven (*pom.xml*) la dépendance suivante :

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>jakarta.el</artifactId>
    <version>4.0.1</version>
</dependency>
```

À reternir (non exhaustif)

1. Usage des annotations "Bean Validation" (JSR 303) (sauf `@Pattern`).
2. Commandes de base de `git` (`git add`, `git commit`, `git pull`, `git push`, ...).
3. Création de nouvelles contraintes.

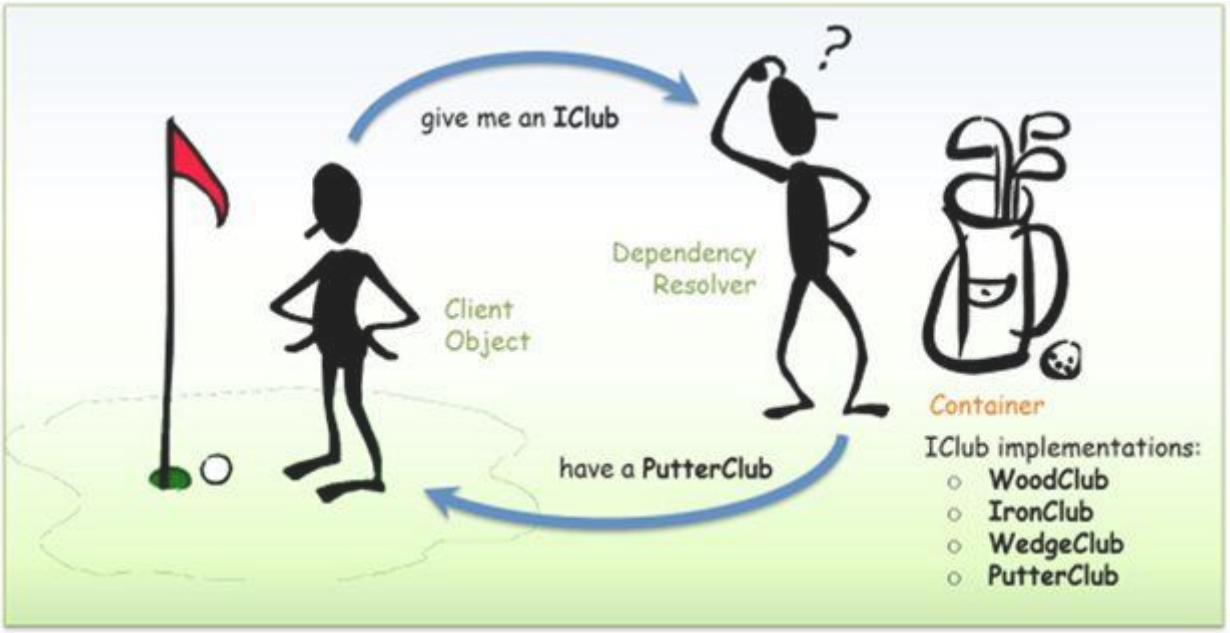
1. https://www.jcp.org/en/jsr/detail?id=303

Injection de dépendances

Concept

- Le but de cette présentation est de faire comprendre le fonctionnement et l'intérêt de l'injection de dépendances.
- L'intérêt est :
  - d'améliorer la modularité d'un projet ;
  - de faciliter les tests ;
  - de ne pas créer ses objets soi-même.
- Nous l'illustrons avec la modélisation d'un jeu de golf (inspiration de https://docs.microsoft.com/fr-fr/aspnet/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-dependency-injection).

Présentation d'un jeu de Golf standard



Le *golfeur* n'a pas à se soucier du club qu'il va utiliser car :

- au début du parcours, un caddy accompagne le golfeur. Celui-ci possède un *sac de golf* avec plusieurs clubs.
- Pendant le parcours, le caddy lui fournit le club adapté à la situation

De cette façon :

- Le caddy est **fournisseur de services**.
- Le golfeur est **consommateur** des services fournis.

Le concept d'**injection de dépendances** est un `pattern`[1] qui est implémenté dans de nombreux *frameworks* Java parmi lesquels :

- JavaEE avec les EJBs (Enterprise Java Beans)
- Spring framework
- Google Guice[2] (que nous utiliserons par la suite)

Quelques explications : https://github.com/google/guice/wiki/Motivation (prenez le temps de lire cette page plus tard, elle est très instructive).

Nous allons voir comment organiser les classes sans et avec injection de dépendances.

Principe de découplage et abstraction (SANS injection de dépendances)

Le principe de *découplage* est important dans le développement logiciel, car il permet une bonne évolutivité du code tout en facilitant la maintenance et diminuant le risque d'*effets de bord*.

Interface Club (déclaration de service)

En Java, l'abstraction est gérée en déclarant des `interfaces` de services.

Notez que dans une interface, toutes les méthodes sont implicitement publiques. Ainsi, on ne déclare pas le mot-clé *public* devant les méthodes. Pour de plus amples détails : http://docs.oracle.com/javase/tutorial/java/concepts/interface.html.

Il est nécessaire de fournir des implémentation de ce service.

Classe Putter, implémentation d'un club (Implémentation de service)

L'implémentation est effectuée comme suit. À noter qu'elle est placée dans un sous-package afin de mieux illustrer le `couplage` qui apparaîtra dans la classe *Caddy*.

D'autres implémentations sont nécessaires pour les autres services (clubs).

Classe Ball

La classe `Ball` représente une balle de golf à une position sur le parcours (attribut *position*).

Classe du Caddy (Fournisseur de services)

La classe du caddy :

- possède les différents clubs ;
- sait quel club utiliser en fonction des conditions.

À noter le `couplage` entre `Caddy` et les deux clubs `Putter` et `Wood`. La dépendance est marquée à deux endroits :

1. dans la section d'imports :

---

2. dans les attributs, sur le `new` :

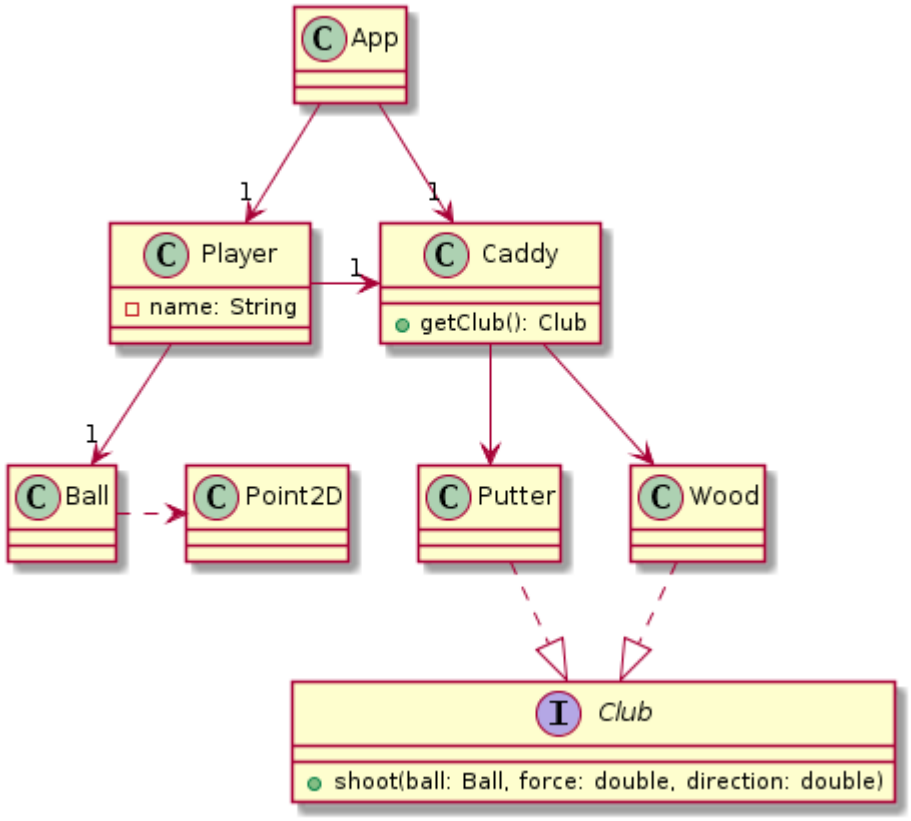
Enum des Conditions

La classe d'`enum` des différentes conditions de jeu.

Classe Player

Classe App

Pour finir, le point d'entrée principal du programme qui fait avancer une balle de golf sur un parcours.



Exercice

Objectif : Coder le mini-projet de Golf (15 minutes) qui servira de base à la version avec injection de dépendances.

1. À l'aide de *Maven* (dans le dossier `injection` du TP)
   - créer la structure du projet (cf. cours Prérequis et `archetype`)
   - nom de projet: *golf*
2. Importer le projet dans IntelliJ (cf. cours Prérequis)
3. Copier l'interface Club
4. Copier l'implémentation Putter qui fait avancer la balle de golf de 10 mètres
5. Coder l'implémentation Wood qui fait avancer la balle de golf de 100 mètres
6. Coder les autres classes du projet
7. Dans une classe App, coder la fonction `main`
8. Commiter/pousser

C'est rapide, tout le code est dans le support !

Explications

La classe *Caddy* dans l'exemple précédent joue 2 rôles :

1. le rôle de ***conteneur*** en *cachant* les implémentations de Club (en attribut)
2. le rôle de ***résolveur*** de dépendances en *connaissant* les implémentations d'interfaces à utiliser selon les conditions du terrain

L'**abstraction** est illustrée dans la classe *Player* où il n'y a aucune trace des classes *Putter* et *Wood* ! Les classes concrètes sont manipulées à travers l'interface Club.

Pour autant, il y a un **couplage** au niveau de la classe *Caddy* puisque cette dernière est obligée d'instancier les classes concrètes.

Utilisation d'injection de dépendances

Injection de dépendances avec Google Guice

- `Google Guice` est un *framework* d'injection de dépendances.
- L'injection permet de *supprimer* le couplage entre le golfeur et son caddy ainsi qu'entre le caddy et ses clubs.
- Techniquement, l'opération est rendue possible grâce à la JVM qui permet la manipulation du bytecode *à la volée*.

L'instanciation des objets qui ont des dépendances va être déléguée au `conteneur` Guice. C'est à ce dernier que l'on va devoir s'adresser pour récupérer des instances.

Classe App avec google Guice

Référence :

- https://google.github.io/guice/api-docs/4.2.2/javadoc/com/google/inject/Guice.html
- https://google.github.io/guice/api-docs/4.2.2/javadoc/com/google/inject/Injector.html

Classe MainModule

Tout le code de résolution des dépendances est dans la classe *MainModule* qui indique quelle classe utiliser pour quel service.

Référence :

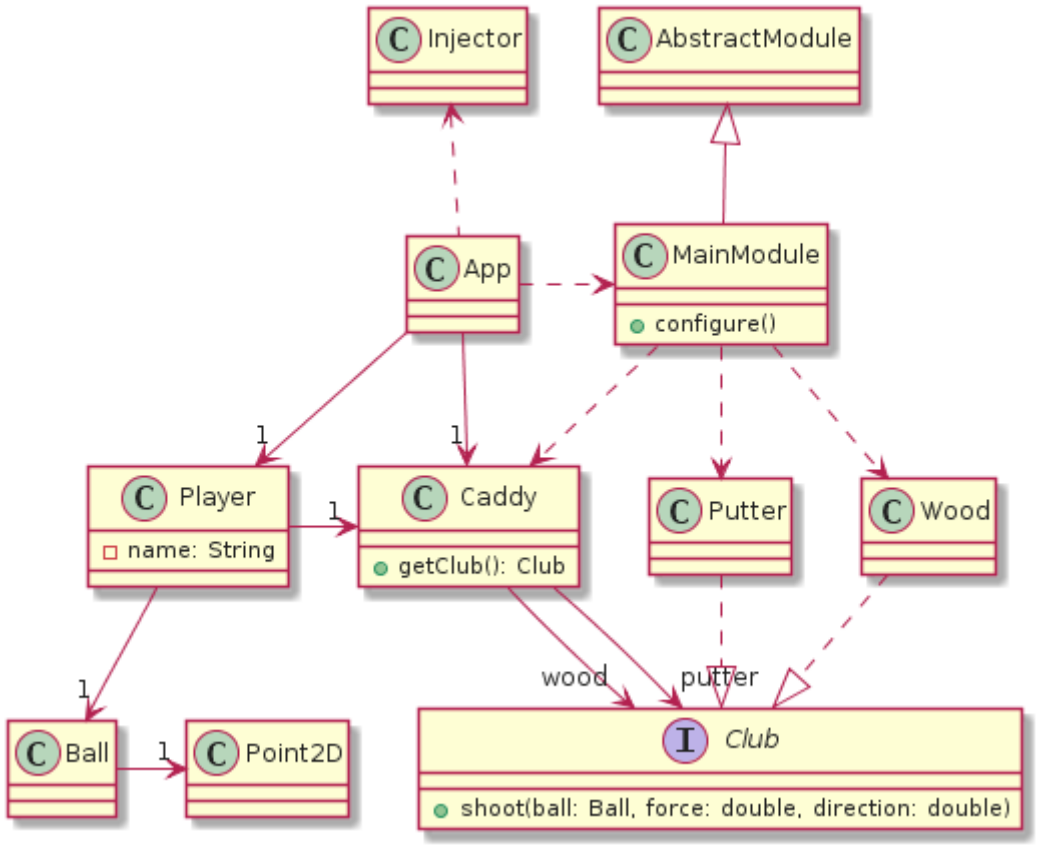
- https://google.github.io/guice/api-docs/4.2.2/javadoc/com/google/inject/AbstractModule.html
- https://google.github.io/guice/api-docs/4.2.2/javadoc/com/google/inject/name/Names.html

Classe Caddy

Dès lors, l'utilisation de Guice permet d' `injecter les dependances` comme dans le code ci-dessous.

Référence :

- https://google.github.io/guice/api-docs/4.2.2/javadoc/com/google/inject/name/Named.html
- https://google.github.io/guice/api-docs/latest/javadoc/com/google/inject/Inject.html
- On note qu'il n'y a plus aucun couplage avec une implémentation de `Club` dans `Caddy` (Suppression des `import` et `new`).
- On peut ajouter de nouvelles implémentations de `Club` sans recompiler `Caddy`.



Exercice

Objectif : Mise en place d'un système d'injection de dépendances. Intégrer google Guice au mini-projet de Golf (30 minutes)

1. Pensez à commiter et pousser la version précédente du golf
2. Rajouter la dépendance dans le `pom.xml`

```
<dependencies>
   ...
   <dependency>
      <groupId>com.google.inject</groupId>
      <artifactId>guice</artifactId>
      <version>5.0.1</version>
   </dependency>
   ...
</dependencies>
```

1. Rajouter la classe `MainModule`
2. Modifier la classe `Caddy`
3. Modifier le `main` pour exécuter ce programme
4. Coder une nouvelle implémentation de l'interface Club nommé `PutterExperimental`.
5. Ajouter une deuxième classe avec un `main` et un `MainModuleExperimental` pour utiliser la nouvelle implémentation de `Club` (`ExperimentalPutter`) pour illustrer la possibilité de créer une autre application qui utilise un autre type de Putter sans avoir à modifier `Caddy`.
6. Commiter régulièrement
7. Ajouter un tag `injection` à la fin du TD
8. Pousser

Nous verrons plus tard comment lancer ce type d'application manuellement. Contentez-vous d'utiliser l'IDE pour le moment.

Les architectures en couches ont un recours massif à l'injection de dépendances. Elles limitent -entre autres- les risques de dépendances circulaires, signe d'une erreur de conception.

1. http://martinfowler.com/articles/injection.html
2. https://github.com/google/guice

Persistance des objets

Dans cette partie, nous illustrons la persistance à travers l'application `room-manager` que vous allez devoir compléter. La première partie de cette séance présente l'application et les différents concepts à apprendre pour la comprendre. La seconde partie présente la persistance (que vous avez déjà vu dans d'autres matières) en Java EE.

Présentation du projet Room-Manager

- v1.5 : version ligne de commande.
- Une version de base vous est fournie ; vous devrez la compléter.

Objectifs pédagogique

- Manipuler les concepts logiciels modernes :
  - `injection de dépendances` avec `Google Guice`
  - persistance JavaEE `JPA` avec `Hibernate`
- Utiliser l'outillage JavaEE (`Maven` , `CheckSyle` , `PMD` , `JUnit` , `Cobertura` , etc...)
- Utiliser des bibliothèques tierces open-source (`slf4j` , etc...)

Démarrer l'application avec le livrable

Le binaire de l'application est également livrée sous forme d'archive `.zip` : room-manager-1.5-bin.zip`. Pour l'installer, il suffit de décompresser l'archive (voir le contenu ci-après).

Sous Windows : double-clic sur l'icône du fichier `.jar` (si installation de Java par *l'installeur*).

En ligne de commande (bash ou Dos) :

```
$ cd <room-manager directory>
$ java -jar room-manager.jar
```

*Notez qu'il n'y a aucune indication de **Class-Path** ni de Main-Class dans la ligne de commande !*

Interface *ligne de commandes* :

```
INFO  21:35:34.870 f.i.rm.App [App.java:119] - Room-Manager version 1.0 started
INFO  21:35:35.115 f.i.rm.App [App.java:122] - starting persistency service
usage: room-manager.jar
 -c <name>   Create new room
```

```
-h          Display help message
-l          List all rooms
-q          Quit
```

Contenu de l'archive du livrable

Toutes les bibliothèques dépendantes sont stockées dans le dossier `lib`, sous l'archive JAR.

```
<room-manager directory>
   |
   - room-manager.jar
   |
   - lib
       - antlr-2.7.6.jar
       - aopalliance-1.0.jar
       - asm-3.1.jar
       - cglib-2.2.jar
       - commons-cli-1.2.jar
       - commons-collections-3.1.jar
       - dom4j-1.6.1.jar
       - guice-3.0.jar
       - guice-persist-3.0.jar
       - h2-1.3.160.jar
       - hibernate-commons-annotations-3.2.0.Final.jar
       - hibernate-core-3.6.0.Final.jar
       - hibernate-entitymanager-3.6.0.Final.jar
       - hibernate-jpa-2.0-api-1.0.0.Final.jar
       - javassist-3.12.0.GA.jar
       - javax.inject-1.jar
       - jta-1.1.jar
       - logback-classic-1.0.13.jar
       - logback-core-1.0.13.jar
       - slf4j-api-1.6.1.jar
```

Voici le contenu du fichier MANIFEST.MF qui informe la JVM des dépendances à l'exécution du programme :

```
$ unzip -c room-manager.jar META-INF/MANIFEST.MF
```

```
Archive:  /Users/fred/workspace_iut/teaching.s4_jee/03_persistence/room-manager/target/room-manager-1.0/
room-manager.jar
  inflating: META-INF/MANIFEST.MF
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: fred
Build-Jdk: 1.7.0_25
Main-Class: fr.iut.rm.App
Class-Path: lib/guice-3.0.jar lib/javax.inject-1.jar lib/aopalliance-1
 .0.jar lib/guice-persist-3.0.jar lib/slf4j-api-1.6.1.jar lib/logback-
 classic-1.0.13.jar lib/logback-core-1.0.13.jar lib/hibernate-jpa-2.0-
 api-1.0.0.Final.jar lib/hibernate-entitymanager-3.6.0.Final.jar lib/h
```

```
ibernate-core-3.6.0.Final.jar lib/antlr-2.7.6.jar lib/commons-collect
ions-3.1.jar lib/dom4j-1.6.1.jar lib/hibernate-commons-annotations-3.
2.0.Final.jar lib/jta-1.1.jar lib/cglib-2.2.jar lib/asm-3.1.jar lib/j
avassist-3.12.0.GA.jar lib/h2-1.3.160.jar lib/commons-cli-1.2.jar
```

Les informations de l'entrée `Class-Path` sont stockées par convention à 80 colonnes, chaque bibliothèque (.jar) séparée d'un caractère espace !

Construction des Livrables

Le projet est outillé sous *Maven* ; on tire ainsi profit des déclarations de dépendances du `pom.xml` pour automatiquement construire le livrable (nommé assembly).

Les sources sont fournies : room-manager-1.5-src.zip.

Construction du livrable : `plugin Maven Assembly`

La commande `mvn assembly:assembly` [1] permet de fabriquer entièrement le programme a livrer. Le résultat est un fichier `.zip` dans le répertoire `target` qui contient (cf. section précédente) :

- l'archive Java (.JAR) de notre code
- l'ensemble des bibliothèques nécessaires au fonctionnement du programme
  - Injection
  - Google Guice
  - JPA
  - Hibernate (implémentation JPA)
  - Module d'interprétation de commande en ligne
  - Gestionnaire de traces applicatives
  - Parseur XML

Comment `maven` a-t-il connaissance des informations à inclure ?

Le livrable est défini dans le fichier binary.xml du dossier src/main/assembly[2]

```
<?xml version="1.0" encoding="UTF-8"?>
<assembly
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" \
      xsi:schemaLocation="http://maven.apache.org/xsd/assembly-1.1.0.xsd">
   <id>bin</id>
   <formats>
      <format>zip</format>
   </formats>
   <files>
```

```
      <file>
        <source>target/${project.artifactId}-${project.version}.jar</source>
        <outputDirectory>/</outputDirectory>
        <destName>${project.artifactId}.jar</destName>
      </file>
    </files>
    <dependencySets>
      <dependencySet>
        <outputDirectory>/lib</outputDirectory>
        <unpack>false</unpack>
        <scope>runtime</scope>
        <useProjectArtifact>false</useProjectArtifact>
      </dependencySet>
    </dependencySets>
</assembly>
```

Construction du site web du projet : `plugin Maven Site`

La commande `mvn site` [3] génère le site web du projet qui contient :

- la liste des dépendances et licences logicielles associées
- différents rapports d'audit
  - Surfire : tests unitaires
  - Cobertura : couverture des tests unitaires
  - FindBugs : outil d'analyse statique de code
  - PMD : outil de recherche des erreurs *communes* de programmation
  - Checkstyle, outil d'audit des conventions de codage
  - JavaDocs (applicatif et tests unitaires)

Certains des plugins mis en œuvre sont configurés dans le dossier `config` du projet.

Lancement de l'application depuis les sources

```
$ mvn assembly:assembly && cp target/room-manager-1.5-bin.zip /tmp && (cd /tmp && unzip -o room-manager-1.5-bin.zip && cd room-manager-1.5 && java -jar room-manager.jar)
```

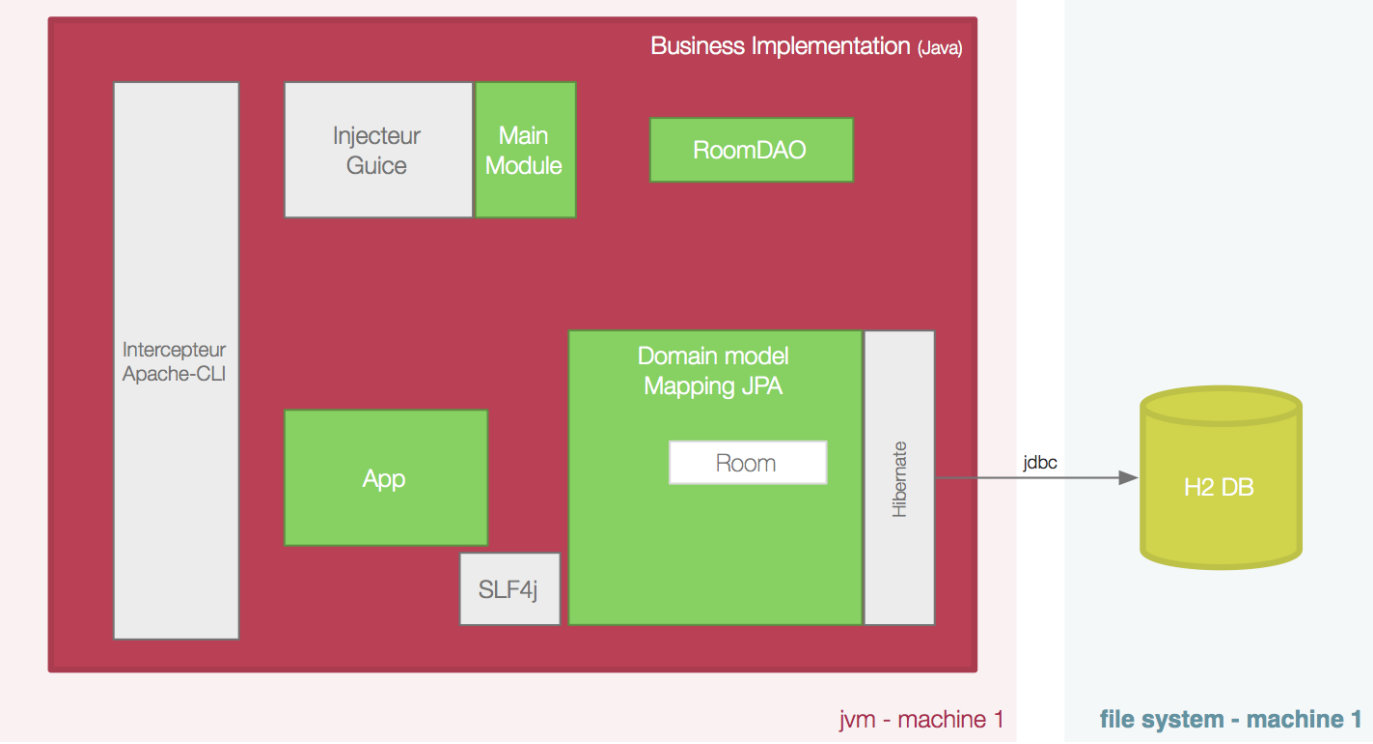
Exercice

Objectif : Re-construire les `livrables` de l'application et enregistrer une salle (15 minutes).

1. récupérer le code source dans la bibliothèque
2. le décompresser dans un dossier `room-manager` du dépôt utilisé pour les TDs
3. Commiter et pousser

4. construire le livrable. L'archive du livrable est dans le dossier 'target'.
5. générer le site. Il faut que la version du plugin `findBugs` soit 3.0.1 (faire les modifications nécessaires dans le fichier `pom.xml`)
6. faire fonctionner l'application via le livrable
   a. lister les salles
   b. créer une salle
   c. lister à nouveau les salles
   d. quitter l'application
   e. relancer l'application
   f. lister les salles

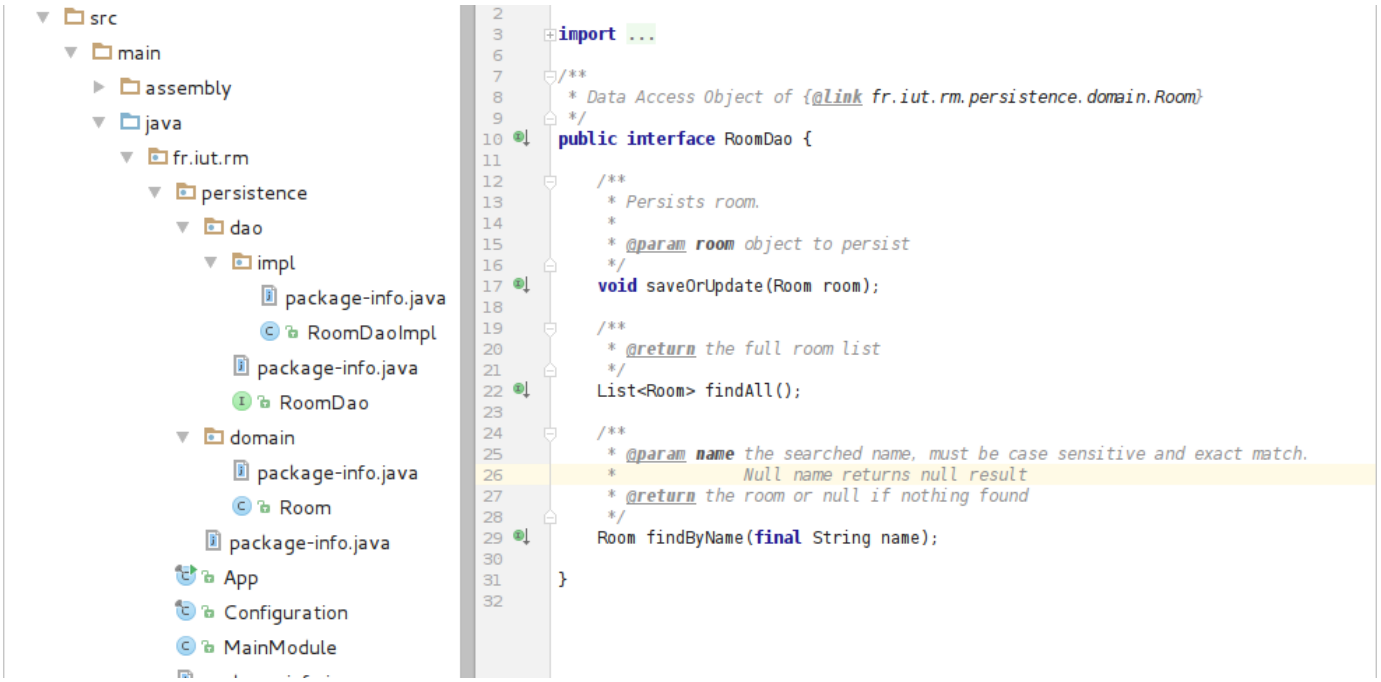
Architecture technique



Le patron de conception DAO (Data Access Object) est utilisé dans ce type d'application.

A savoir une claire séparation entre les classes métiers et les fonctions d'accès à la base de données (lecture/écriture/modification/supression des données de la BD). Le paquet `persistence.domain` contient les classes métiers. Le paquet `persistence.dao` contient les classes réalisant la persistance des données (objets des classes métiers), c'est à dire les accès à la base de données.

Seules les classes du paquet `persistence.dao.imp` dépendent de la technologique de stockage utilisée (JDBS, JPA, ...). En cas de changement de technologie, seules ces classes sont à réécrire.



Les fonctionalités de mise à jour de la BD sont déclarées dans des `interfaces` (*RoomDAO* en étant un exemple). Ces interfaces définissent les accès à la base de données permettant la persistance des "objets". L'acronyme informatique anglais CRUD (pour create, read, update, delete) (parfois appelé SCRUD avec un "S" pour search) désigne les quatre opérations de base pour la persistance des données, en particulier le stockage d'informations en base de données.

L'interface *RoomDAO* définit les fonctionnalités de persistance relatives aux objets de la classe *Room* : sauvegarde, récupération, modification. L'interface RoomDAO est incomplète (l'opération de suppression n'est pas proposée).

Composants tiers

Interpréteur Apache-Commons CLI

Apache-Commons CLI[4] est une bibliothèque utilitaire pour l'interprétation des commandes saisies.

```
...
// build options command line options
options.addOption(
  OptionBuilder
  .withDescription("List all rooms")
  .create(LIST));
options.addOption(
  OptionBuilder
  .withArgName("name")
  .hasArg()
```

```
  .withDescription("Create new room")
  .create(CREATE));
options.addOption(
  OptionBuilder
  .withDescription("Display help message")
  .create(HELP));
options.addOption(
  OptionBuilder
  .withDescription("Quit")
  .create(QUIT));
...
```

Google Guice

Guice permet la mise en œuvre de l'injection de dépendances.

```
  @Override
  protected final void configure() {
    logger.debug("MainModule configuration started");
    bind(App.class);
    bind(RoomDao.class).to(RoomDaoImpl.class);
    logger.debug("MainModule configuration done");
  }
```

*Remarquez comment sont **liées** l'application (App) ainsi que l'implémentation de DAO.*

Traces applicative

Un framework d'abstraction des traces applicatives est utilisé dans le programme : `slf4j`. Plusieurs *vrais* frameworks d'implémentation des traces peuvent ainsi être utilisés (intégré à Java, log4j, etc.). Changer d'implémentation devient alors transparent afin que l'application puisse facilement s'intégrer à d'autres systèmes possédant déjà une solution de traces applicatives (cas des Conteneurs Java EE).

La configuration des traces est réalisée dans le fichier `logback.xml`.

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-5level %d{HH:mm:ss.SSS} %logger{10} [%file:%line] - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="${log.root.level}">
    <appender-ref ref="STDOUT"/>
  </root>

  <logger name="fr.iut.rm" level="info"/>
  <logger name="org.hibernate" level="warn"/>
  <logger name="org.hibernate.SQL" level="debug"/>
  <logger name="cli" level="info">
```

```
    <appender-ref ref="FILE"/>
  </logger>
</configuration>
```

Pour que les traces soit enregistrées dans le fichier `~/.log`

```
<configuration>
   <appender name="FILE" class="ch.qos.logback.core.FileAppender">
      <file>${user.home}/.log</file>
      <encoder>
         <pattern>%-5level %d{HH:mm:ss.SSS} %logger{10} [%file:%line] - %msg%n</pattern>
      </encoder>
   </appender>

  <root level="${log.root.level}">
    <appender-ref ref="FILE"/>
  </root>

  <logger name="fr.iut.rm" level="info"/>
  <logger name="org.hibernate" level="warn"/>
  <logger name="org.hibernate.SQL" level="debug"/>
  <logger name="cli" level="info">
    <appender-ref ref="FILE"/>
  </logger>
</configuration>
```

H2

H2[5] est une base de données SQL implémentée en Java et accessible en `JDBC`[6] (*JDBC est un standard Java de connexion aux bases de données*).

H2 supporte des modes `serveur` (stand-alone) ou `embarqué` (embedded). Le stockage peut aussi bien être `en mémoire` (in memory) ou sur `système de fichiers` (filesystem).

Modèle métier

Composé simplement d'une classe `Room` qui respecte la norme `Bean` aussi nommé `javaBean` :

- un constructeur *vide* publique
- des `accesseurs`

ainsi que des attributs (*id* et *name*).

À souligner la présence d'annotations `JPA` (présenté dans les sections suivantes) :

- à la déclaration de la classe
- à la déclaration des attributs de la classe

```
package fr.iut.rm.persistence.domain;

import javax.persistence.*;

/**
 * A classic room
 */
@Entity
@Table(name = "room")
public class Room {
    /**
     * sequence generated id
     */
    @Id
    @GeneratedValue
    private long id;

    /**
     * Room's name
     */
    @Column(nullable = false, unique = true)
    private String name;

    /**
     * Default constructor (do nothing)
     */
    public Room() {
        // do nothing
    }

...
```

JPA, Java Persistance API

Notons qu'en français, on dit **persistance** et en anglais, **persistence**.

Il est nécessaire de rendre certains objets de l'application accessibles en permanence (même après un redémarrage, une mise à jour, un changement de machine, etc.)

Une solution est la persistance des objets dans des bases de données relationnelles :

- le stockage se fait donc sous forme de lignes stockées (appelé enregistrements) dans une ou plusieurs tables composées de différentes colonnes
- une clé primaire permet d'identifier les données (elle est stockée dans une colonne particulière)

Problème : Le monde `objet\Classe` est différent du monde `relationnel`

Les `classes` ne sont pas des tables (les `objets` ne sont pas des enregistrements) !

- il n'y a pas de notion d'héritage dans le monde relationnel ;

- les références aux collections sont *inversées* ;
- il n'y a pas la même notion d'égalité (primary key vs. instance)

Solution : JPA et Mapping Relationnel-Objet (ou `ORM`, Object Relationnal Mapping)

- Illusion de manipuler une base orientée objet dans une base relationnelle : il s'agit de créer une correpondance des classes à une base de données relationnelles.
- Différents ORM *framework propriétaires* le permettent (Hibernate, Ibatis, etc.). Cependant, les dernières versions de JavaEE ont publié un standard : `JPA` pour `Java Pesistence API`. Une partie des *frameworks propriétaires* ont implémenté le standard, dont `Hibernate`, utilisé dans l'application *Room-Manager*.

Littéralement « Java Persistence API », il s'agit d'un standard faisant partie intégrante de la plate-forme Java EE, une spécification qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données, ou autrement formulé la gestion de la persistance. les ORM plateformes réalisent cette correspondance.

Configuration de ORM

L'élément de premier niveau pour faire fonctionner JPA est l'`EntityManager`[7]. Ce dernier est configuré au moyen du fichier `persistence.xml` qui doit être placé dans le dossier `META-INF` de l'archive (`resources/META-INF` dans les sources).

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

  <!-- JPA Test Unit -->
  <persistence-unit name="room-manager" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- Need to add all classes due to JPA persistence file scope problem -->
    <class>fr.iut.rm.persistence.domain.Room</class>

    <properties>
      <property name="hibernate.hbm2ddl.auto"
            value="update"/>
      <property name="hibernate.showSql"
            value="true"/>

      <!-- H2 -->
      <property name="hibernate.dialect"
            value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.connection.driver_class"
            value="org.h2.Driver"/>
      <property name="hibernate.connection.url"
            value="jdbc:h2:room-manager;DB_CLOSE_DELAY=-1"/>
```

```
  </properties>
 </persistence-unit>

</persistence>
```

Parmi les informations importantes contenues dans les balises, on peut noter :

- le nom de l'unité de persistance "base de données" (ici la base de données est stockée dans le répertoire courant sous le nom *room-manager*). D'autres possibilités plus fiables existent :
- la base de données est en mémoire vive (les données ne sont pas réellement persistantes) :

```
<property name="hibernate.connection.url"
      value="jdbc:h2:mem:room-manager;DB_CLOSE_DELAY=-1"/>
```

  - la base de données est stockée dans le répertoire ~/JEE/BD/ sous le nom *room-manager* :

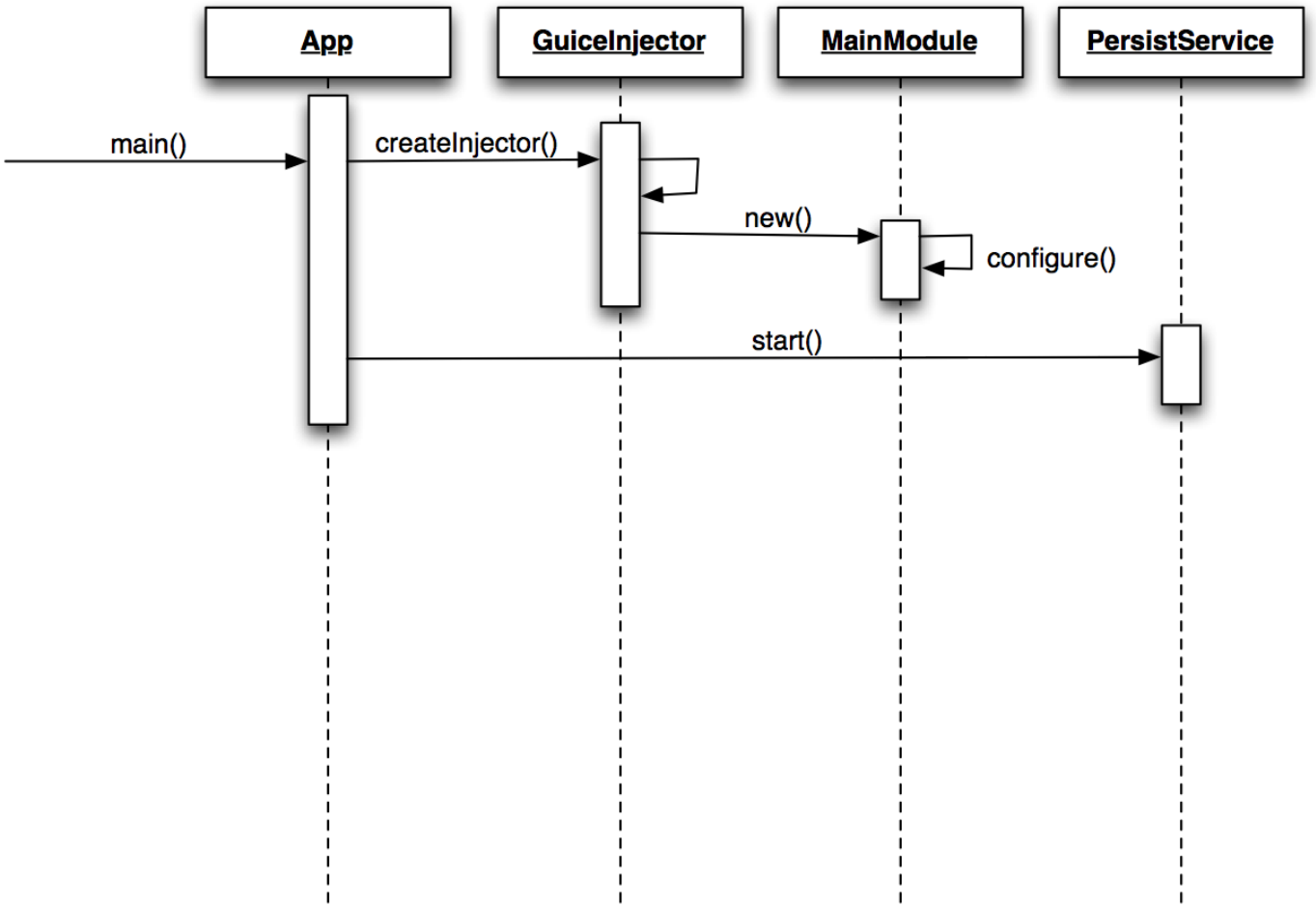
```
<property name="hibernate.connection.url"
    value="jdbc:h2:~/JEE/BD/room-manager;DB_CLOSE_DELAY=-1"/>
```

- le framework d'implémentation, *org.hibernate.ejb.HibernatePersistence*
- les classes Beans annotées `@Entity`
- `hibernate.hbm2ddl.auto`, pour créer automatiquement le schéma
  - **validate** : valide le schéma, ne réalise aucune modification sur le schéma de la base.
  - **update** : met à jour le schema.
  - **create** : crée le schéma, détruisant au préalable les anciennes données.
  - **create-drop** : supprime le schéma à la fin de la session.

Dans un programme, l'`EntityManager` est instancié pendant la phase de `bootstrap` (démarrage)

```
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("room-manager");
EntityManager em = emf.createEntityManager();
...
```

Dans le cas de l'application *Room-Manager*, l'instanciation est réalisée par le biais du `PersistService` afin que ce dernier puisse être injecté.



Utilisation de JPA

Décrire le Mapping entre les classes et les tables de la base

Afin qu'une classe Java puisse être considérée comme une `entité` une table, c'est à dire une classe qui est gérée par JPA, elle doit être déclarée dans le `persistence.xml` et `annotée` comme tel.

Pour Rappel :

```
...
@Entity
@Table(name = "room")
public class Room {
    /**
     * sequence generated id
     */
    @Id
    @GeneratedValue
    private long id;

    /**
     * Room's name
     */
```

```
    @Column(nullable = false, unique = true)
    private String name;

    /**
     * Default constructor (do nothing)
     */
    public Room() {
        // do nothing
    }
```

- La classe doit être un `bean`. Constructeur public sans arguments/getters/setters - L'entité ne doit pas être finale, ses attributs non plus - `@javax.persistence.Entity` permet au gestionnaire de persistance de ne pas reconnaitre la classe comme un simple POJO et de permettre de la persister - `@javax.persistence.Id` indique l'identifiant unique de l'objet (cf. clé primaire) - `@javax.persistence.GeneratedValue` indique que le gestionnaire de persistance se charge de générer cette valeur - Par défaut : - L'entité est stockée dans une table - Chaque attribut est stocké dans une colonne

Exercice (40 minutes)

Objectif : Apprendre à utiliser JPA. Rajouter un champ *description* de type *String*, de taille maximale de 10 caractères à l'entité *Room*.

1. Modifier la classe *Room*.
2. Modifier temporairement le *hibernate.hbm2ddl.auto* pour les besoins du développement (fichier `persistence.xml`)
3. Rajouter les annotations nécessaires (penser à la *Validation* vu la séance précédente).
4. Vérifier que la description de la class Room générée par le plugin de Maven contient bien le champs description (fichier Room_ dans le répertoire target). SI nécessaire compile le projet via la commande en ligne de Maven
5. Tester la création d'un Objet "Room" dont la description a plus de 10 caractères.
6. Modifier les commandes de création et construction de liste.

Relations entre les entités

Les relations entre les entités peuvent avoir différentes cardinalités; ce qui doit être spécifié au travers d'annotations.

- 1-1 (`@OneToOne`) [8]
- 1-n (`@OneToMany`) [9]
- n-1 (`@ManyToOne`) [10]

- n-n (`@ManyToMany`) [11]

Relations entre les entités - exemple «OneToOne»

extrait de http://blog.paumard.org/cours/jpa/chap03-entite-relation.html

```
//
// Entité Commune
//
@Entity
public class Commune implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length=40)
    private String nom ;

    @OneToOne
    private Personne maire ;

    // suite de la classe
}
```

En base, les choses se passent assez simplement : une colonne `maire_id` va être créée dans la table `Commune` . Cette colonne sera une clé étrangère référençant la colonne `id` de la table `Maire.` On peut imposer le nom de cette colonne en ajoutant l'annotation `@JoinColumn(name="...")` sur la relation `maire.`

```
//
// Entité Maire
//
@Entity
public  class Personne  implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length=40)
    private String nom ;

    @OneToOne(mappedBy="maire")  // référence la relation dans la classe Commune
    private Commune commune ;

    // suite de la classe
}
```

Exemple de relation bidirectionnelle entre les entités - «OneToMany» et «ManyToOne»

```
//
// Entité Marin
```

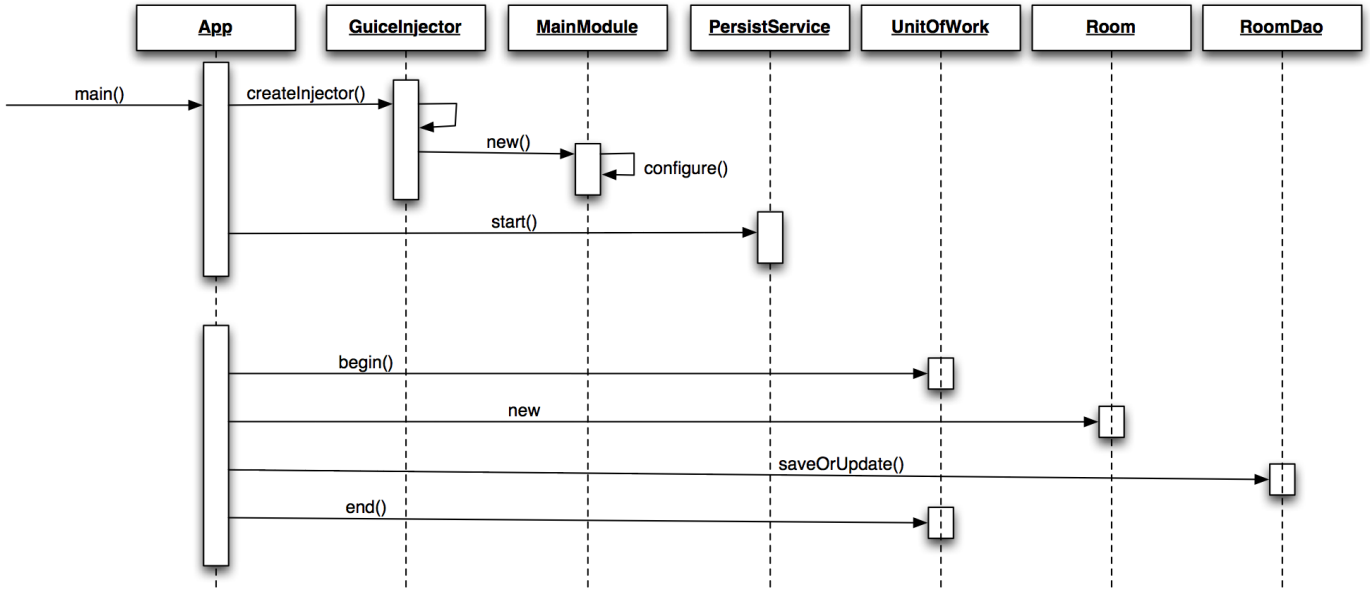
```
//
@Entity
public  class Marin  implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    private Bateau monBateau ;

    // reste de la classe
}

//
// Entité Bateau
//
@Entity
public  class Bateau  implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @OneToMany(mappedBy="monBateau")
    private Collection<Marin> marins ;

    // reste de la classe
}
```

Utilisation des objets mappés

Dans *Room-Manager*, l'utilisation des objets mappés est encadrée dans une *UnitOfWork* [12]. Cet élément permet de gérer la connexion et les transactions avec la base de données :

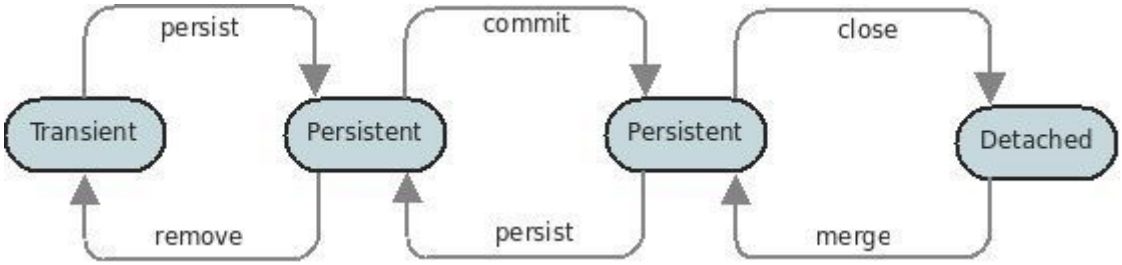
- sauvegarde en base rapide (1 grosse modification plutôt que plusieurs petites)
- état de la base toujours valide (tout ou rien)

Il est possible d'interdire des opérations JPA en dehors d'un cadre transactionnel.



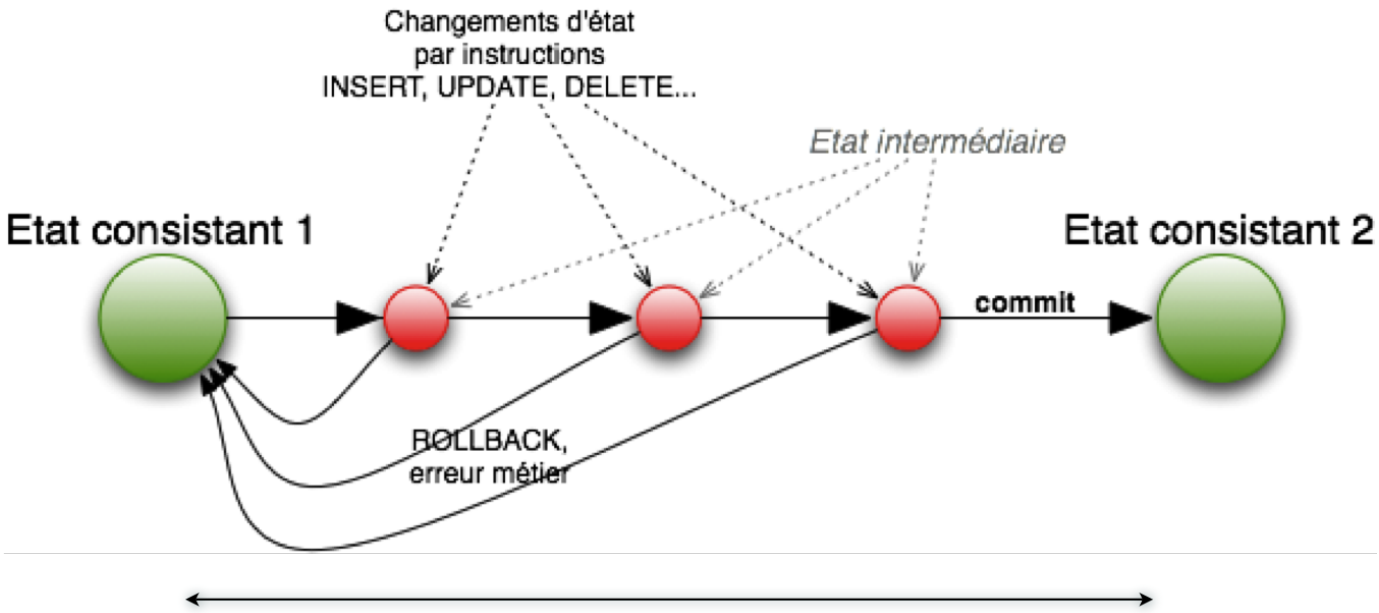
Cycle de vie d'une entité

Un objet instancié dans la JVM a plusieurs états vis-à-vis de l' `EntityManager` dont le rôle est d'assurer la synchronisation avec la base de données.



Gestion des transactions

Dans le monde des base de données relationnelles, les transactions sont essentielles à la conservation de l'intégrité du modèle de données.



Les propriétés `ACID` sont les quatre principaux attributs d'une transaction de données.

- **Atomicité** (transaction complète ou rien du tout)
- **Cohérence** (état nécessairement valide après transaction)
- **Isolation** (Pas de dépendences entre transactions)
- **Durabilité** (Si transaction confirmées : sur à 100% que les données sont enregistrées)

Pour autant, il n'y a pas de trace, *de prime abord* de la gestion des transactions dans le code source de *Room-Manager*. Elles sont en réalité gérées au moyen d'annotations, comme illustré dans l'extrait de code ci-après :

```
/**
 * @param room room to persist
 */
@Override
@Transactional
public void saveOrUpdate(final Room room) {
  this.em.get().persist(room);
  logger.debug("Room '{}' saved", room.getName());
}
```

L'annotation permet d'ouvrir et commiter automatiquement la transaction. Pour autant, le *vrai* commit n'aura lieu qu'à la fin de l'`UnitOfWork`. Si une exception est levée, la transaction sera automatiquement annulée (rollback).

Patron de conception DAO

Le programme *Room-Manager* utilise le `pattern` (modèle de conception logicielle) de `DAO` pour `Data Access Object` [13]. Son utilité est d'isoler au sein d'une seule et même couche tous les accès à la

persistance (à savoir les accès (CRUD) aux données stockées de manière permanente). Pour isoler, les accès aux données stockées, des interfaces telles que *RoomDao* sont créées; leur implémentation dépendra de la technique de stockage choisie (BD, fichier, ...).

```
package fr.iut.rm.persistence.dao;

import fr.iut.rm.persistence.domain.Room;

import java.util.List;

/**
 * Data Access Object of {@link fr.iut.rm.persistence.domain.Room}
 */
public interface RoomDao {

  /**
   * Persists room.
   *
   * @param room object to persist
   */
  void saveOrUpdate(Room room);
    /**
   * @param room object to remove from DB
   */
  void delete(Room room);

  /**
   * @return the full room list
   */
  List<Room> findAll();

  /**
   * @param name the searched name, must be case sensitive and exact match.
   *             Null name returns null result
   * @return the room or null if nothing found
   */
  Room findByName(final String name);

}
```

Java Persistence Query Language : JPQL

`JPQL` permet de créer des requêtes de recherche d'objets (aussi nommé entités) persistantes indépendamment du mécanisme utilisé pour stocker ces entités.

`JPQL` est donc portable et non limité à un type de base de données particulier.

`JPQL` est conçu pour combiner la syntaxe et la simplicité sémantique des requêtes SQL aux caractéristiques propres à Java.

L'implémentation des méthodes utilise des requêtes en `JPQL` (Java Persistence Query Language) [14]. Ce langage, est très proche de `SQL`, sauf qu'il utilise le nom des classes/attributs. Ainsi le programmeur n'a pas à connaître les noms des tables et des colonnes pour écrire une requête.

Afin d'éviter *d'écrire en dur* les noms des attributs (et de générer des requêtes erronnées), un plugin Maven génère du code Java de description des tables afin d'utiliser des constantes générées et vérifiables à la compilation [15] pour nommer les tables, colomnes , ...

Donc à chaque modification de la structure d'une `Entity`, il faut mettre à jour les méta-données : (c'est à dire recompiler via Maven le projet). Par exemple,

- `Room.class.getName()` est la table associée à la classe `Room`.
- `Room_.name.getName()` est le nom de la colonne correspondant à l'attribut "name" de la table associée à la classe `Room`.

```
/**
 * @param name  of the room
 * @return the corresponding room or null if nothing found
 */
@Override
@Transactional
public Room findByName(final String name) {
    StringBuilder query = new StringBuilder("from ");
    query.append(Room.class.getName()).append(" as room");
    query.append(" where room.").append(Room_.name.getName());
    query.append(" = :name");

    List<Room> resultList = em.get().createQuery(query.toString()).
                setParameter("name", name).
            getResultList();

    if (resultList.size() > 0) {
        logger.debug("Room  with name '{}' found", name);
        return (Room) resultList.get(0);
    }
    logger.debug("No room with name '{}' found", name);
    return null;
}
```

Enfin, l'utilisation de `PreparedStatement` (requêtage paramétré) offre de multiples avantages :

- cachabilité des plans d'exécution par les (gros) moteurs de base de données (Oracle, Postgres, etc.)
- insensibilité au risque d'attaque par injection SQL [16] avec l'échappement automatique des paramètres

Exercice

Objectif : Maîtriser le pattern DAO et les fonctions JPA, avoir des notions de JPQL. Ajouter une nouvelle entité *AccessEvent* afin de gérer les entrées/sorties dans une salle.

1. Ajouter dans *roomDao* la fonctionnalité de destruction d'un enregistrement *removeRoom(String name)*; ainsi les 4 opérations de CRUD seront implémentés.
2. Ajouter la commande dans l'application pour détruire un salle;
3. compléter le code de la méthode *createRoom* de la classe *App* : tester l'unicité des noms de salle;
4. Modifier temporairement le *hibernate.hbm2ddl.auto* pour les besoins du développement;
5. Développer la classe *AccessEvent* * ajouter une attribut d'identification auto-généré * ajouter un attribut de nom de personne * ajouter un attribut de type d'évènement (entrée/sortie) * ajouter un attrivut de data/heure d'évenement * la relation vers *Room* est unidirectionnelle et elle est de type many-to-one
6. Ajouter la commande dans l'application pour déclarer une entrée dans une salle;
7. Ajouter la commande dans l'application pour déclarer une sortie d'une salle;
8. Ajouter la commande de visualisation du journal des entrées/sorties d'une salle.
9. Réaliser une fonction qui affiche tous les movements d'une personne (désignée par son nom).

Aspects non abordés

JPA est une norme très complète et un TD de 4 heures est largement insuffisant pour la couvrir en totalité. Ainsi, on peut citer quelques points afin de se faire une idée :

- Les caches de premier et second niveaux (et les problématiques associées en cluster) - La gestion de la concurrence - La gestion du versionning des entités - La gestion des Callbacks - L'utilisation des intercepteurs - L'intégration avec les fonctionnalités natives de certaines base de données (trigger, etc.)

Finalisation du projet room manager

 **Exercice**

Objectif : Maîtriser les outils d'aide à la gestion de qualité du logiciel.

1. Modifiez votre code pour ne plus générer d'erreur dans FindBugs, PMD et checkstyle. Le code fourni a déjà des erreurs détectées par ces outils, il faut aussi les corriger.
2. Maximisez la couverture de tests.

À retenir (non exhaustif)

1. Usage, contenu, structure, et réalisation d'un livrable via Maven.
2. Patron de conception DAO (Data Acces Object).
3. Usage de JPA pour construire la structure d'une base de données. Par exemple, savoir écrire la classe *Building* et modifier la classe *Room* (dont les objets seront enregistés dans la base de données) à l'aide des annotations JPA. Un building a une adresse (chaîne de moins de 500 caractéres alphanumériques), un nombre d'étages et il contient des pièces (*Room*). La relation entre *building* et *Room* est bidirectionnelle. L'annotation permettant de valider l'adresse d'un immeuble est aussi à écrire. Qu'elles sont les modifications à faire dans le fichier *persistence.xml* ?
4. Des notions de JPQL. Par exemple savoir écrire la requête paramétrée suivante en JPQL "lister les building ayant au moins X étages" à l'aide de la requête JPQL suivante :

```
StringBuilder query = new StringBuilder("from ");
query.append(Room.class.getName()).append(" as room");
query.append(" where room.").append(Room_.name.getName());
query.append(" = :name");
List<Room> resultList = em.get().createQuery(query.toString()).
                    setParameter("name", name).getResultList();
```

5. Utilité/usage de JPA, ORM, JPQL, ...
6. Des connaissances sur les composants utilisés : `slf4j`, `h2`, `common CLI`, `Google Guice`.
7. Connaissance du rôle des fichiers *binary.xml*, *persistence.xml* et *lockback.xml*.
8. Architecture logiciel (métier, dao, contrôle, IHM)
9. Connaissance du rôle de la classe *com.google.inject.persist.UnitOfWork* et des annotations @Transactional, @Inject ...

Ressources supplémentaires

- Support JPA
- Support JPQL

1. http://maven.apache.org/plugins/maven-assembly-plugin/
2. http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html
3. http://maven.apache.org/plugins/maven-site-plugin/
4. http://commons.apache.org/proper/commons-cli/
5. http://www.h2database.com/html/main.html

6. http://www.oracle.com/technetwork/java/javase/jdbc/index.html
7. http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html
8. http://docs.oracle.com/javaee/7/api/javax/persistence/OneToOne.html
9. http://docs.oracle.com/javaee/7/api/javax/persistence/OneToMany.html
10. http://docs.oracle.com/javaee/7/api/javax/persistence/ManyToOne.html
11. http://docs.oracle.com/javaee/7/api/javax/persistence/ManyToMany.html
12. http://martinfowler.com/eaaCatalog/unitOfWork.html
13. http://www.oracle.com/technetwork/java/dataaccessobject-138824.html
14. https://docs.oracle.com/html/E24396_01/ejb3_langref.html
15. https://docs.jboss.org/hibernate/jpamodelgen/1.0/reference/en-US/html_single/
16. http://fr.wikipedia.org/wiki/Injection_SQL

Notion de conteneur d'applications

Notion de conteneur

Java EE a pour vocation de définir un ensemble de règles normées, `les spécifications`, définies à travers le processus d'adoption des `JSR`, les *Java Specification Requests*.

Ce qui distingue Java EE de Java SE, c'est la nature de ces spécifications, qui sont *orientées pour résoudre les problématiques des logiciels d'entreprise* telles que :

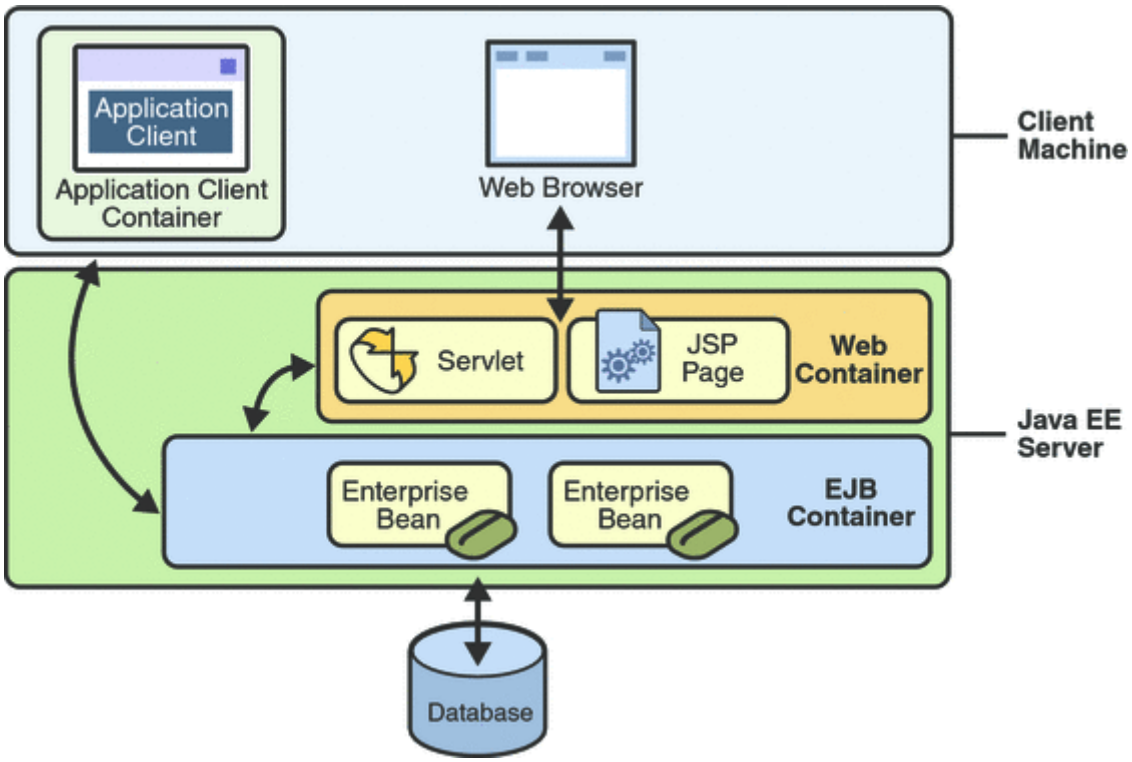
- accès concurents, *multithreading* - accès distants (*remote ou remoting*) - interfaces web (*servlets, JSP, JSF, etc).*
- intégrité des données (*transactions et XA*)
- fiabilité, *fail-over*
- performance et haute-disponibilité (*clustering* et *high availability*)
- sécurité (*role et realm*)

Dans le standard Java EE, l'exécution d'une *fonction métier* tire tout ou partie des services précédemment cités.

> Java EE n'est pas un logiciel ou un programme, c'est une norme.
> Ce sont les éditeurs de logiciels qui fournissent des implémentations du standard.
> On parle alors de conteneurs.

Depuis Java 5, l'utilisation des *annotations* dans la `norme EJB3` a permi d'alléger considérablement le développement (anciennement très lourd) d'EJBs. Cependant, les concepts manipulés n'en restent pas moins triviaux. En outre, la mise en œuvre des conteneurs d'application à base d'EJB s'avérant parfois complexe, cette technologie souffre d'une certaine désaffection de la part des industriels. En pratique, même si les conteneurs full Java EE sont choisis par les industriels, les applications utilisent très rarement la panoplie complète de la spécification.

Principe de fonctionnement

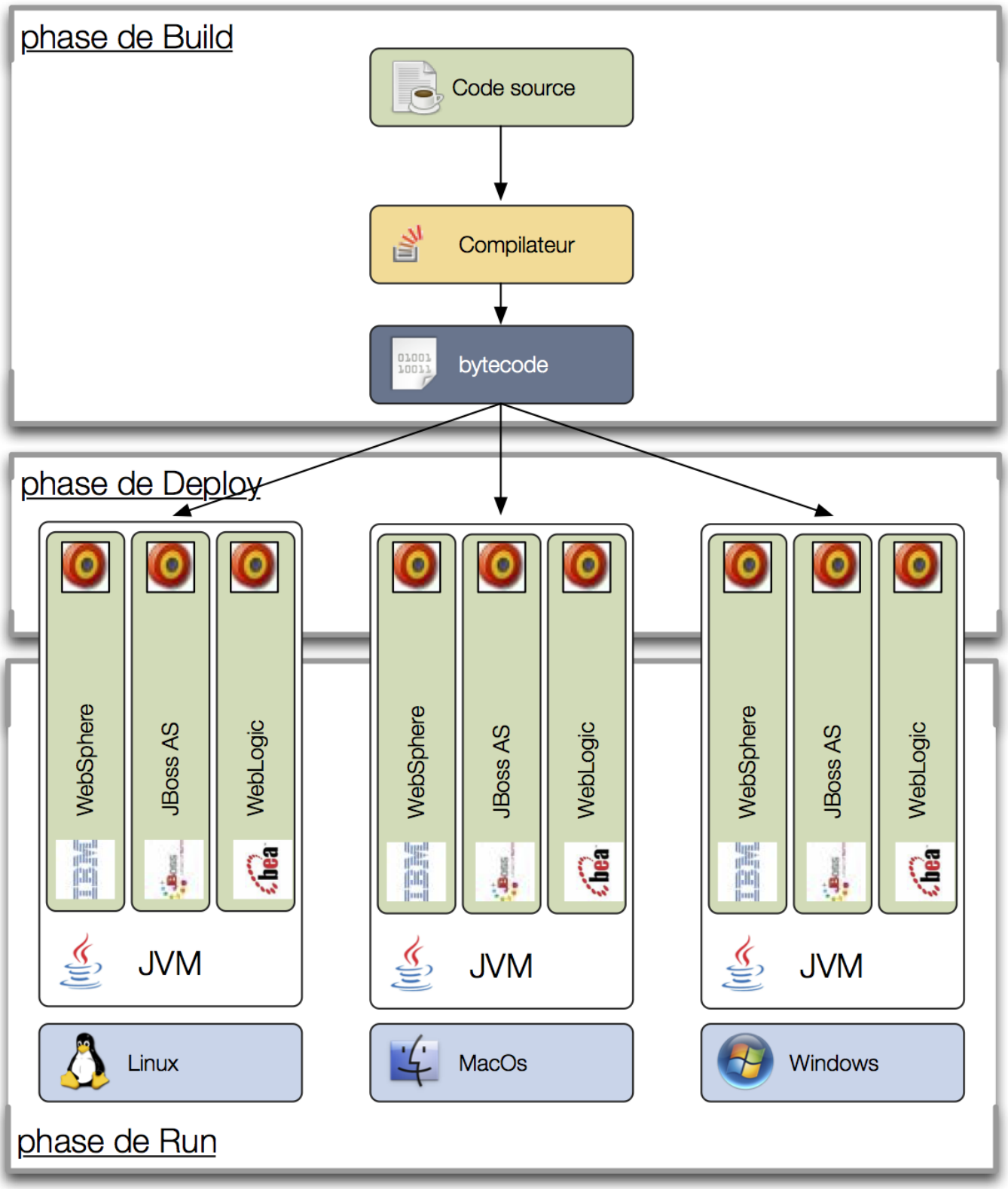


Différence majeure avec Java SE : plus de main

Processus de fabrication

Le processus de fabrication d'une application Java EE est en trois phases :

1. `build`, c'est la **compilation** du projet
2. `deploy`, comprend le **packaging**, c'est à dire l'archivage en `WAR` ou en `EAR` du code compilé. Une fois construite, l'archive est *déposée* (`déployée`) dans un dossier du conteneur prévu à cet effet.
3. `run`, c'est le démarrage par le serveur, autrement dit de la JVM qui lance le *main* du conteneur


phase de Build
Code source
Compilateur
bytecode
phase de Deploy
WebSphere
JBoss AS
WebLogic
JVM
Linux
MacOs
Windows
phase de Run

Quelques exemples de conteneurs

Les conteneurs ne couvrent pas forcément l'ensemble des spécifications de la norme Java EE. Ainsi, certains sont plus ou moins *légers* afin de mieux s'adapter aux besoins des développeurs. On peut néanmoins distinguer deux grandes familles de conteneurs :

1. Les conteneurs Full Java EE.
2. Les conteneurs légers, dits de servlet.

> Le mécanisme d'interface Java permet de découpler ses développements
> de l'implémentation fournie par l'éditeur du conteneur.

Autrement dit, une application développée sur le conteneur A devrait pouvoir être exécutée sur le conteneur B.

Conteneurs Full Java EE

Tous couvrent l'ensemble de la norme Java EE, mais ils se distinguent sur différents points comme l'interface d'administration, certaines facilités de *clusterisation* ou encore l'intégration plus ou moins poussée à un IDE.

- JBoss & Wildfly (RedHat)
- GlassFish (Sun)
- BEA Weblogic (Oracle)
- WebSphere (IBM)

Conteneurs de Servlets

On parle alors de *servlet container* ! Ils intègrent un sous-ensemble de la spécification, essentiellement pour les accès web. Les bibliothèques manquantes peuvent naturellement être intégrées à l'application.

Les principales implémentations disponibles sont :

- Tomcat (fondation Apache)
- Jetty (fondation Eclipse)
- Winstone (Sourceforge)

Pour autant, les problématiques d'entreprises encadrées par Java EE ne sont pas oubliées, mais confiées à des implémentations séparées.

Conteneurs, promesse tenue ?

Les besoins qui ont fait naître les spécifications Java EE sont toujours d'actualité. Les difficultés de mise en œuvre dans un premier temps, puis les batailles industrielles des éditeurs ont laissé le champ libre à de nombreuses initiatives open-source plus légères. Aujourd'hui, les dernières évolutions de la spécification ont permis de rattraper un certain retard sans pour autant détrôner les nombreux frameworks qui sont devenus des `standards d'usage` à défaut de respecter une norme de JSR. Dans la pratique, les solutions modernes n'utilisent que des bouts de la spécification Java EE !

Toutefois, Java EE a permis de définir les bases des architectures modernes. Les conteneurs ne demeurent pas moins incontournables pour les services de bas niveau (multi-threading, etc.), et Java, EE ou pas, reste une technologie de choix pour l'implémentation de services métiers.

Conteneur WEB

Présentation

Généralités

Un conteneur est un programme Java qui est respecte la norme Java EE. Il y a deux grandes familles de conteneurs :

1. conteneurs full Java EE (JBoss, Glassfish, WebSphere, etc.)
2. conteneurs *légers* dit `servlet container` (Tomcat, Winstone, etc.)

A noter qu'un conteneur *full java EE* intègre également les fonctions d'un conteneur *léger*.

Une `Servlet` est

- une classe Java qui implémente l'interface `javax.servlet.Servlet` :
  - par implémentation directe (rarement)
  - par `héritage` d'une des sous-classes disponibles dans l'implémentation Java EE
    - `GenericServlet` -- Servlet générique non liée au protocole HTTP
    - `HttpServlet` -- Classe abstraite pour les servlets HTTP
    - `FacesServlet` -- Classe pour les applications utilisant JavaServer Faces
- une application qui s'exécute sur un serveur en réponse à une requête.

Apache Tomcat

Nous utiliserons le projet `Tomcat` de la *fondation Apache* :

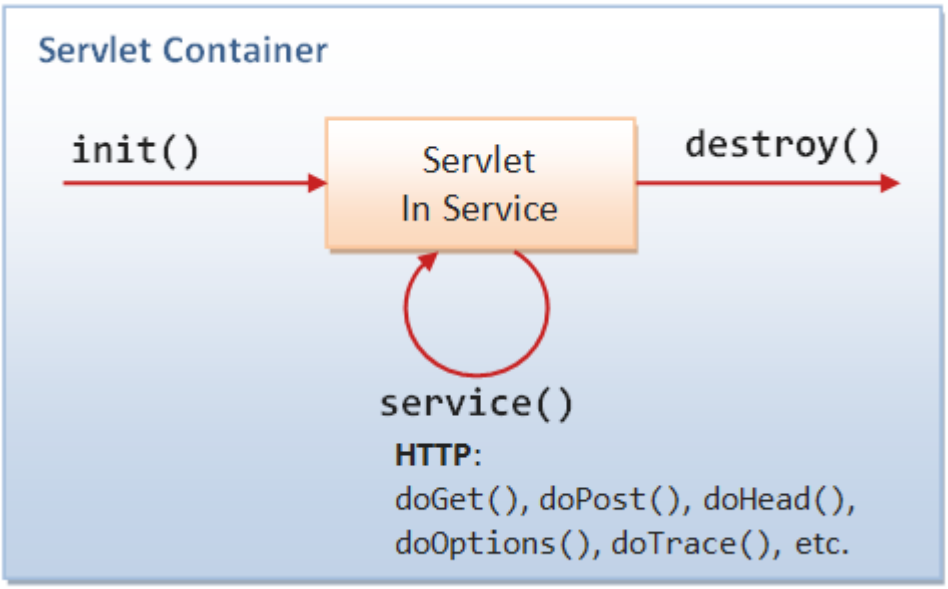
- Projet Open Source de la fondation `Apache`
- Implémente les *servlets*[9]
- Implémente les *JSP* (JavaServer Pages)[10]
- Héberge un serveur *HTTP* (pour des questions de '*tout-en-un*' mais est adossé à un serveur *de production* type Apache et peut alors être raccordé en proxy ou en `protocole JK`, pour Jakarta)

Principe de fonctionnement des servlets

Une servlet est une classe Java utilisée pour étendre les capacités du serveur. Dans notre cas, elles sont accessibles par le biais de requêtes HTTP.

Le développeur n'a pas à gérer le cycle de vie d'une servlet, c'est le conteneur qui s'en charge.

- Instanciation (pool de démarrage)
- Initialisation (appel à la méthode *init()*)
- Démarrage des threads d'écoute (du processus de la JVM)



Une servlet n'est instanciée qu'une seule fois. Ce sont les threads de la JVM qui rendent possible la gestion des appels concurrents. Selon les sollicitations, le conteneur appelle alors les méthodes d'implémentation des `servlets` correspondantes au protocole HTTP :

- **doGet** pour le GET
- **doPost** pour le POST
- etc.

Les instances de servlets sont conservées d'un appel à l'autre et les écueils de la programmation multi-threads doivent être manipulés avec précaution :

- **stockage d'états dans des variables membres (stateless vs. statefull)**
- **gestion de sous-threads**
- **durée de rétention**
- ***pseudo* singletons**
- **et bien d'autres encore...**

En effet, la servlet peut être partagée avec plusieurs threads.

HelloWorld

Distribution d'une application WEB

Pour rappel, les applications Java SE sont diffusées à l'aide d'une archive `.jar` qui inclue les classes compilées de l'application et un fichier de méta-données `MANIFEST.MF` contenant entre autre :

- le nom du la classe principale (entrée *Main-Class*)
- la liste des noms dépendances (entrée *Class-Path*)

Les applications Java EE sont également diffusées à l'aide d'une archive. Celle-ci est au format `WAR` (Web application ARchive) et est ensuite manipulée par le conteneur. Comme le `.jar`, le `.war` est une archive compressée au format `.zip`. **Le contenu est normalisé pour n'importe quel conteneur** :

- fichiers statiques ou ressources à partir de la racine (accessibles directement)
- `WEB-INF` (espace réservé au conteneur)

  - `web.xml`, qui décrit l'application (par analogie au `MANIFEST.MF`, mais dont le contenu est également normalisé) - `classes`, qui contient toutes les **classes compilées** de l'application (format .class) - `lib` qui contient **les dépendances** (les vrais fichiers, pas les chemins) de l'application

Dans tous les cas, les dépendances d'implémentation de la spéc. Java EE, propre au conteneur, sont stockées *ailleurs* dans l'arborescence du serveur, pas dans l'application... Attention donc à vérifier le **scope** dans les dépendances Maven. [12]

Une application Java packagée en `.jar` **peut être lancée** par n'importe quelle `JVM`, n'importe quelle application packagée en `.war` **peut être déployée** dans n'importe quel conteneur Java EE... *moyennant d'éventuels ajouts spécifiques au conteneur, distincts toutefois de l'application*.

HelloWorld pour une servlet

HelloServlet.java, la servlet

C'est une classe Java qui doit respecter le contrat *Servlet*.

- Elle récupère les informations fournies lors d'une requête HTTP (voir le paramètre *request* dans la signature des méthodes *doGet*)
- Elle écrit la réponse de la requête HTTP (voir le paramètre *response* dans la signature des méthodes *doGet*)

- Elle peut générer du code HTML (ou n'importe quoi d'autre) \pagebreak

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest request,
      HttpServletResponse response)
          throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println( "<HTML>" );
    out.println( "<HEAD>");
    out.println( "<TITLE>Page generee par une servlet</TITLE>" );
    out.println( "</HEAD>" );
    out.println( "<BODY>" );
    out.println( "<H1>Hello world !</H1>" );
    out.println( "</BODY>" );
    out.println( "</HTML>" );
    out.close();
  }
}
```

Sortie

```
<HTML>
<HEAD>
<TITLE>Page generee par une servlet</TITLE>
</HEAD>
<BODY>
<H1>Hello world !</H1>
</BODY>
</HTML>
```

Le descripteur de déploiement WEB-INF/web.xml

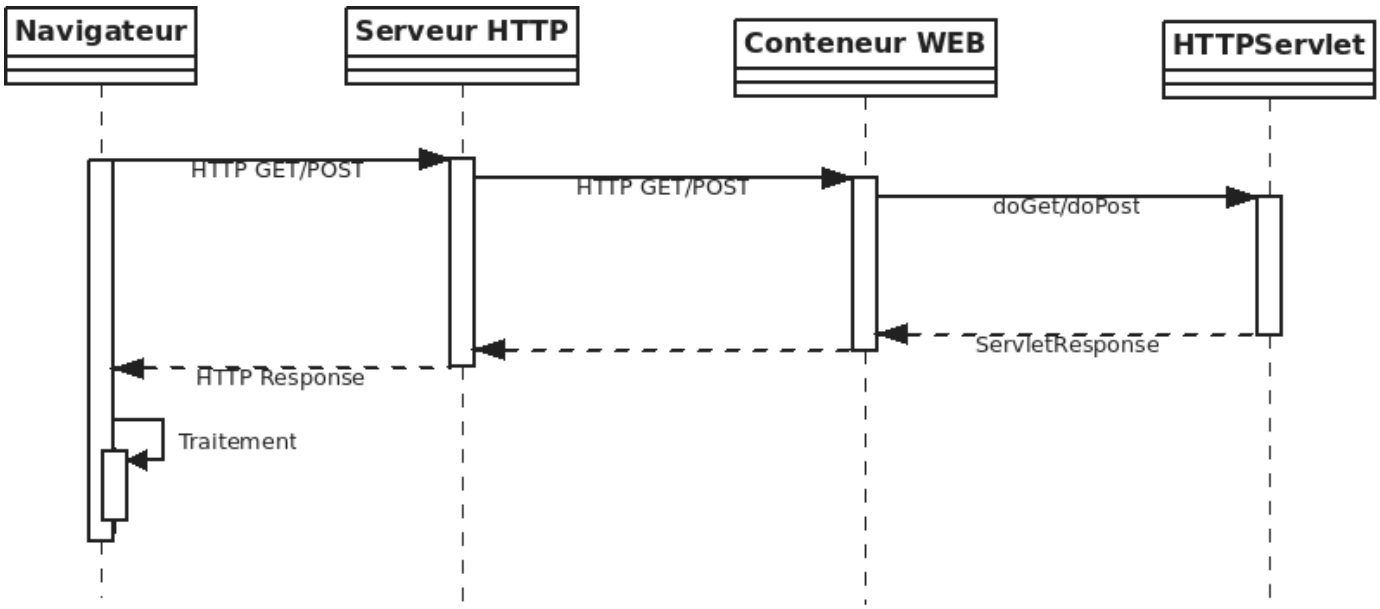
C'est **LE** fichier *névralgique* de l'archive. Le fichier `web.xml` fournit des informations de configuration et de déploiement pour les composants d'une application Web. Deux sections sont particulièrement importantes :

1. `<servlet\>` qui déclare auprès du conteneur les classes de Servlet mises à disposition
2. `<servlet-mapping\>` qui indique au conteneur quelle URL affecter à quelle servlet

```
<!DOCTYPE web-app PUBLIC
      "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
      "http://java.sun.com/dtd/web-app_2_3.dtd" >
```

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>fr.iut.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Déroulement d'une requête HTTP



Exercice

Objectif : Comprendre l'utilisation d'une servlet et créer une application web à l'aide de Maven (30 minutes)

1. Dans votre dépôt, construisez une application WEB `helloServlet` :

```
mvn archetype:generate -DgroupId=fr.iut \
  -DartifactId=helloServlet \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DinteractiveMode=false
```

1. Importer le projet dans IntelliJ
2. Ajouter dans le `pom.xml` la dépendance vers la librairie Java EE (pour connaître les classes de base des servlets) :

```
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>3.1.0</version>
  <scope>provided</scope>
   <!-- pour compilation / non inclu dans l'archive générée -->
</dependency>
```

1. Ajouter dans le `pom.xml` les informations pour compiler avec une version de java précise

```
  <properties>
   <maven.compiler.source>1.8</maven.compiler.source>
   <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

1. *Rafraîchir le projet si nécessaire pour que IntelliJ utilise la nouvelle dépendance*
2. Créer une servlet `fr.iut.HelloServlet` accessible depuis le chemin `/hello` qui affiche un message de bienvenue (pensez à créer le dossier `src/main/java`)

   1. Surchargez la méthode doGet 2. Réaliser une implémentation *inspirée* de l'extrait de code précédent

3. Créer le fichier `web.xml` dans le dossier *src/main/webapp/WEB-INF/* en s'inspirant de l'extrait de code précédent
4. Renommer le fichier *index.jsp* en ***index.html***
5. Remplacer le contenu de *index.html* par

```
<html>
<head>
   <meta http-equiv="Refresh" content="0; URL=hello"/>
</head>
<body></body>
</html>
```

1. Packager le projet avec *Maven* : `mvn package` (et s'assurer du résultat dans le dossier *target/*)
2. Commiter et pousser

Apache Tomcat

Installation

Tomcat est relativement léger, vous pouvez l'installer sur votre compte ou dans le répertoire `/tmp`. Mode opératoire d'installation :

1. Télécharger les **binaires** de la version 9.058 (.tar.gz sous linux ou .zip sous Windows): https://tomcat.apache.org/download-90.cgi
2. Désarchiver dans le dossier de votre choix
3. Pour les démarrages en ligne de commande, déclarer une variable d'environnement `CATALINA_HOME` (éviter les espaces dans les chemins)
4. * linux `export CATALINA_HOME=~/opt/apache-tomcat-9.0.58` * windows `set CATALINA_HOME="C:\opt\apache-tomcat-9.0.58"`
5. Pour les déploiements par la console web, ajouter un administrateur dans le fichier `apache-tomcat-9.0.58/conf/tomcat-users.xml` :

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">
<user username="admin" password="admin" roles="manager-gui"/>
</tomcat-users>
```

1. Tester l'installation en démarrant (voir ci-après) et corriger si nécessaire

Démarrer Tomcat

Tomcat est une application Java. C'est un *serveur/demon* dans le sens où il est prévu pour fonctionner *en tâche de fond* sans interaction humaine. On distinguera le démarrage en mode production du mode *foreground*, adapté au développement. Il est possible que vous deviez modifier les modes d'accès au fichier `catalina.sh` (script shell) : `chmod 544 catalina.sh`.

catalina.sh (ou .bat) pour le mode développement

```
$ cd $CATALINA_HOME/bin
$ ./catalina.sh
Using CATALINA_BASE:   /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4
Using CATALINA_HOME:   /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4
Using CATALINA_TMPDIR: /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4/temp
Using JRE_HOME:        /opt/oracle_jdk
Using CLASSPATH:       /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4/bin/bootstrap.jar:/mnt/roon/users/
```

```
cjohnen/opt/apache-tomcat-9.0.4/bin/tomcat-juli.jar
Usage: catalina.sh ( commands ... )
commands:
  debug             Start Catalina in a debugger
  debug -security   Debug Catalina with a security manager
  jpda start        Start Catalina under JPDA debugger
  run               Start Catalina in the current window
  run -security     Start in the current window with security manager
  start             Start Catalina in a separate window
  start -security   Start in a separate window with security manager
  stop              Stop Catalina, waiting up to 5 seconds for the process to end
  stop n            Stop Catalina, waiting up to n seconds for the process to end
  stop -force       Stop Catalina, wait up to 5 seconds and then use kill -KILL if still running
  stop n -force     Stop Catalina, wait up to n seconds and then use kill -KILL if still running
  configtest        Run a basic syntax check on server.xml - check exit code for result
  version           What version of tomcat are you running?
Note: Waiting for the process to end and use of the -force option require that $CATALINA_PID is defined
```

startup.sh (ou .bat) pour le mode serveur

également disponible en faisant ./catalina.sh start

```
$ ./startup.sh
Using CATALINA_BASE:   /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4
Using CATALINA_HOME:   /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4
Using CATALINA_TMPDIR: /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4/temp
Using JRE_HOME:        /opt/oracle_jdk
Using CLASSPATH:       /mnt/roon/users/cjohnen/opt/apache-tomcat-9.0.4/bin/bootstrap.jar:/mnt/roon/users/
cjohnen/opt/apache-tomcat-9.0.4/bin/tomcat-juli.jar
Tomcat started.

$
```

Vous y avez accès à l'adresse : http://127.0.0.1:8080/

Éteindre Tomcat

Trois façons d'éteindre Tomcat :

1. Ctrl-C : directement en stopant la JVM... suffisant pour le développement, mais termine *brutalement* l'exécution
2. `$ $CATALINA_HOME/bin/catalina.sh stop`
3. avec `shutdown.sh` :

```
$ cd $CATALINA_HOME/bin
$ ./shutdown.sh
```

Déploiement d'applications dans Tomcat

Les opérations à effectuer lors d'un déploiement dans un conteneur Java EE ne sont pas normées. Chaque conteneur a son propre mode de fonctionnement.

Dans Tomcat, on peut déployer les applications selon deux manières :

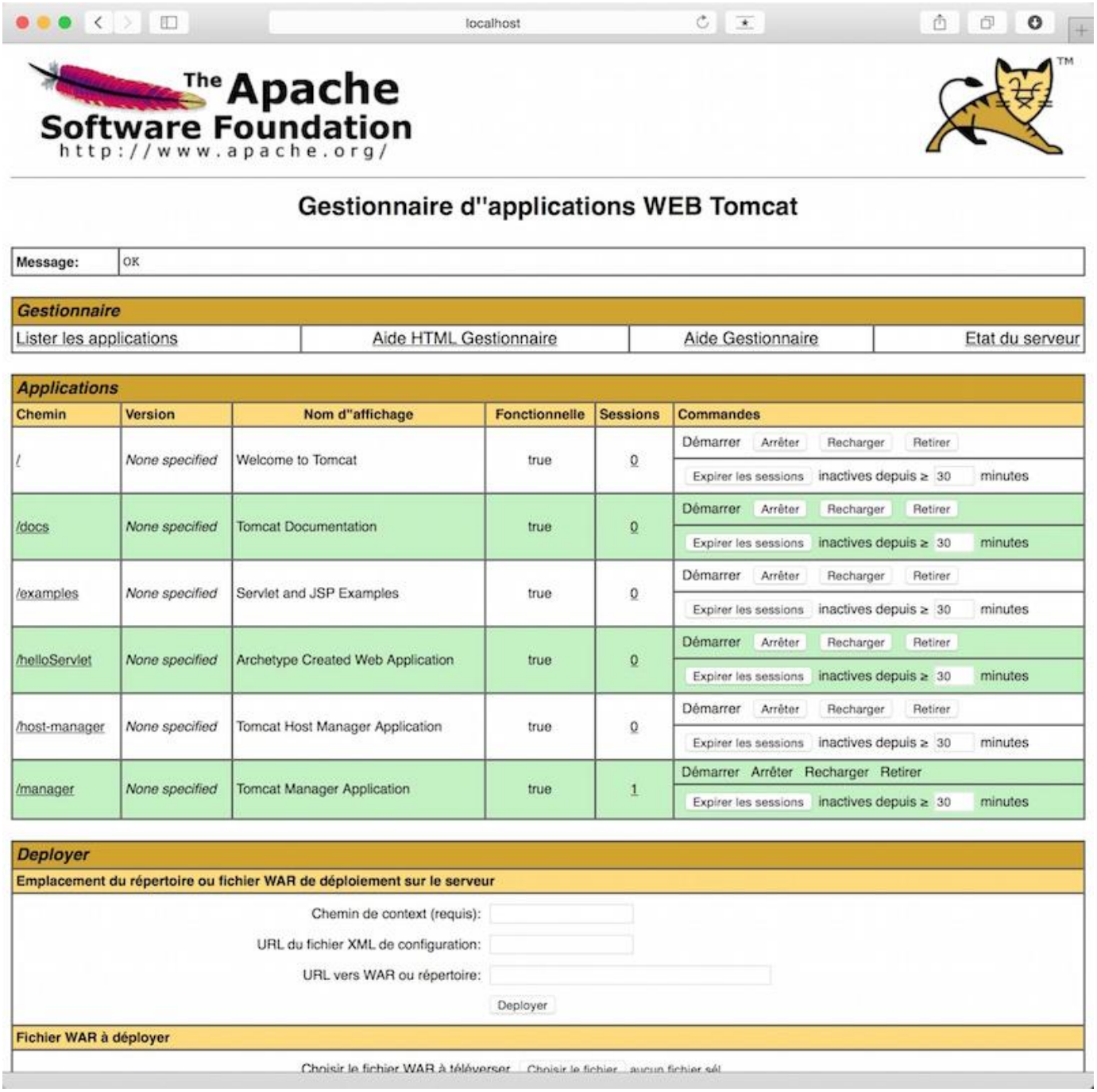
1. graphiquement, par console (web) d'administration, l'installation des application se fait à l'aide du formulaire approprié.
2. manuellement, en copiant/collant l'archive dans le dossier spécifique de déploiement

Pour faciliter le travail, l'utilisation de liens symboliques permet d'éviter de nombreux copier/coller :-)

Déploiement par l'application web *manager*

L'URL d'accès à la gestion des applications contenues dans Tomcat est http://localhost:8080/manager/.

Pour des raisons de sécurité, vous devez utiliser l'identifiant/mot de passe précédemment spécifié pour y accéder.



Exercice

Objectif : Utiliser et configurer Tomcat. Installer Apache Tomcat et déployer l'application HelloServlet (10 minutes)

1. Packager le projet avec *Maven* : `mvn package` (et s'assurer du résultat dans le dossier *target/*)
2. Démarrer Tomcat depuis la console `$CATALINA_HOME/bin/catalina.sh run`

3. Déployer l'application dans Tomcat à l'aide du gestionnaire d'applications Web de Tomcat

   1. Aller sur l'URL http://localhost:8080/manager/ 2. Choisir le *Fichier WAR à déployer* 3. Testez le fonctionnement

```
- de la servlet : <http://localhost:8080/helloServlet/hello>
- de la redirection automatique : <http://localhost:8080/helloServlet/>
```

Alternatives aux servlets

- Les servlets doivent construire manuellement la réponse à la requête HTTP (avec des prints) - Leur écriture peut donc être rébarbative et sujette à erreur - Différentes techniques permettent d'utiliser un formalisme intermédiaire qui :

```
- soit génère à l'exécution une servlet à partir d'un langage de description (JSP[^jsp])
- soit interprète un langage de description à l'aide d'une servlet existante (JSF[^jsf])
```

- ... afin de simplifier le développement de la partie graphique d'une application

Exemple JSF - JavaServer Faces

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

   <h:head>
      <title>JSF 2.0 Hello World</title>
   </h:head>
   <h:body>
      <h3>JSF 2.0 Hello World Example - hello.xhtml</h3>
      <h:form>
         <h:inputText value="#{helloBean.name}"></h:inputText>
         <h:commandButton value="Welcome Me" action="welcome">
      </h:commandButton>
      </h:form>
   </h:body>
</html>
```

- La courbe d'apprentissage de JSF est relativement longue - Le langage est très évolué - manipulation des Beans à l'aide du langage EL - Pour des raisons historique, **qui ne sont plus justifiées**, JSF n'est pas utilisé intensivement dans l'industrie - Nous ne l'utiliserons pas en TD

Exemple JSP - JavaServer Pages

Le JavaServer Pages ou JSP est une technique basée sur Java qui permet aux développeurs de créer dynamiquement du code HTML, XML ou tout autre type de page web.

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
      <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
      <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

- La courbe d'apprentissage de JSP est relativement simple - Le langage est minimaliste - Syntaxe proche de PHP - Nous l'utiliserons dans les TD

Frameworks MVC

JSP comme vue de servlet

Les *servlets* sont à la base de tout développement web en Java EE. Pour résumer, on pourrait dire que c'est du code HTML dans du Java. À l'usage, il s'est rapidement avéré que le mélange n'était pas adapté aux conditions réelle de développement : un web designer n'est pas à l'aise dans du code !

Très vite, la technologie JSP (JavaServer Pages) a tenté de résoudre la problématique. Une JSP est une *servlet à l'envers*, c'est à dire que c'est du code HTML qui contient des balises spécifiques.

Rappel de l'exemple précédent :

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
      <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
```

```
    } else {
  %>
     <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

À la première invocation, le conteneur détecte que c'est une JSP qui est demandée et **génère à la volée du code source Java** puis le **compile** afin de faire le rendu.

L'apparition des JSPs a cette fois mécontenté les développeurs qui se sont vite rendu compte des difficultés de maintenance du code ainsi produit, le manque d'IDE (integrated development environment) n'ayant pas arrangé les choses.

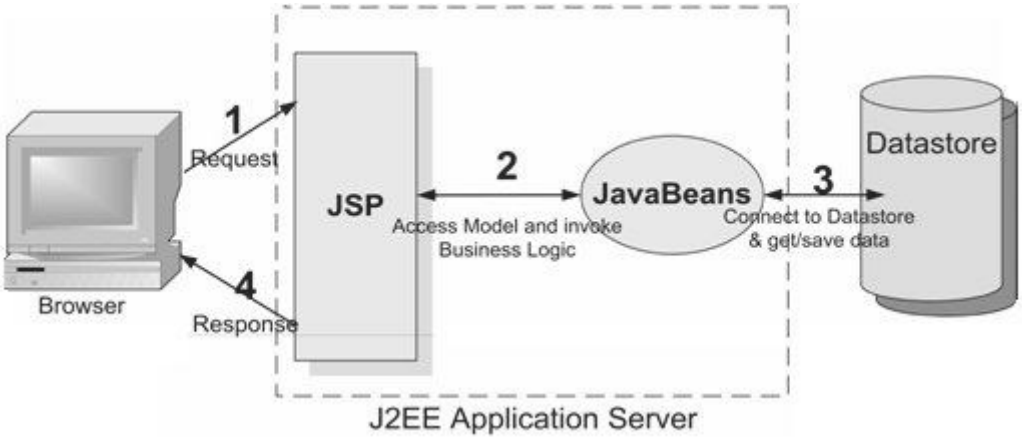
Les Tags, ou taglibs, sont alors venus *réconcilier* les deux mondes. Un tag est une *balise* à insérer dans une JSP. Cette balise est reliée à du code Java, séparant ainsi clairement les deux mondes.

MVC à la rescousse

Très vite, les `patterns` ont été mis en place pour favoriser la maintenabilité du code source des applications. MVC, Model View Controller a très vite rencontré un franc succès, d'autant plus que les initiatives open source ont pris de cours les industriels des JSR (Java Specification Requests).

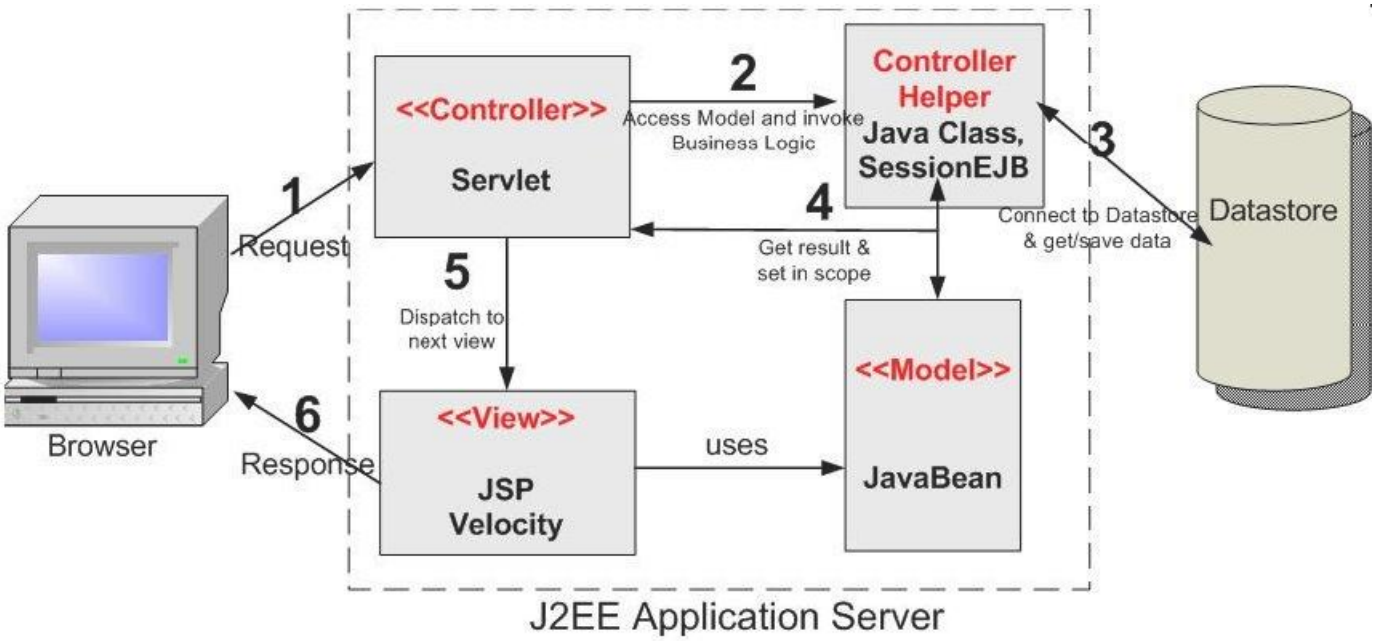
Dans les deux cas, l'idée est de séparer la logique de rendu (la vue), de la logique de récupération des données (le modèle) de la logique de contrôle du flux des demandes (le controlleur). Une fois mis en place, l'architecture MVC permet de remplacer une technologie d'un composant par une autre. L'exemple le plus courant est de remplacer les JSPs par un `moteur de templating` générique, type *Freemarker*[13] ou *Velocity*[14].

MVC modèle 1



Un controleur par vue.

MVC modèle 2



Un controleur qui redirige les requêtes à d'autres controleurs spécialisés.

Liste de frameworks à connaître

Ceci n'est pas une liste exhaustive. Chaque framework tend à améliorer ou résoudre une problématique du web (portabilité du code, rapidité de développement, dette technique, etc.)... Mais tous ces frameworks s'appuient sur des *servlets* et reposent sur les mêmes principes !

- `JSTL` , Javaserver page Standard Tag Library, bibliothèque de Tags
- `Struts v.1` , a eu beaucoup de succès car répondant aux problématiques
- `Struts v.2` , moins verbeux que la précédente version
- `Spring MVC` , évolution du framework d'injections de dépendances éponyme
- `JSF` , JavaServer Faces, la version *normalisée* par les JSR, mais qui s'est fait prendre de vitesse par les initiatives open source

HelloWorld template Freemarker

Objectif : Manipuler un framework de templating, Freemarker dans une servlet (30 minutes)

Ressource d'aide : [1]

1. Ajouter la dépendance de Freemarker dans le `pom.xml` :

```
      ...
<dependency>
  <groupId>org.freemarker</groupId>
  <artifactId>freemarker</artifactId>
  <version>2.3.20</version>
</dependency>
      ...
```

1. Ajouter une nouvelle `classe de servlet`, cette fois, déclarée par annotation (plutôt que dans le fichier `web.xml`)

```
package fr.iut;

import freemarker.template.DefaultObjectWrapper;
import freemarker.template.Template;
import freemarker.template.TemplateException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

@WebServlet(name = "home", urlPatterns = {"/home"})
public class HomeServlet extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
              throws ServletException, IOException {
    Template freemarkerTemplate = null;
    freemarker.template.Configuration freemarkerConfiguration =
    new freemarker.template.Configuration();
    freemarkerConfiguration.setClassForTemplateLoading(this.getClass(), "/");
    freemarkerConfiguration.setObjectWrapper(new DefaultObjectWrapper());
    try {
       freemarkerTemplate =
          freemarkerConfiguration.getTemplate("templates/home.ftl");
    } catch (IOException e) {
      System.out.println("Unable to process request,
           error during freemarker template retrieval.");  }

    Map<String, Object> root = new HashMap<String, Object>();
        // navigation data and links
    root.put("title", "freemarker Servlet");
    root.put("now",
           SimpleDateFormat.getDateTimeInstance().format(new Date()));
    PrintWriter out = response.getWriter();
    assert freemarkerTemplate != null;
```

```
    try {
        freemarkerTemplate.process(root, out);
      out.close();}
    catch (TemplateException e) { e.printStackTrace(); }
        // set mime type
    response.setContentType("text/html");
    }
}
```

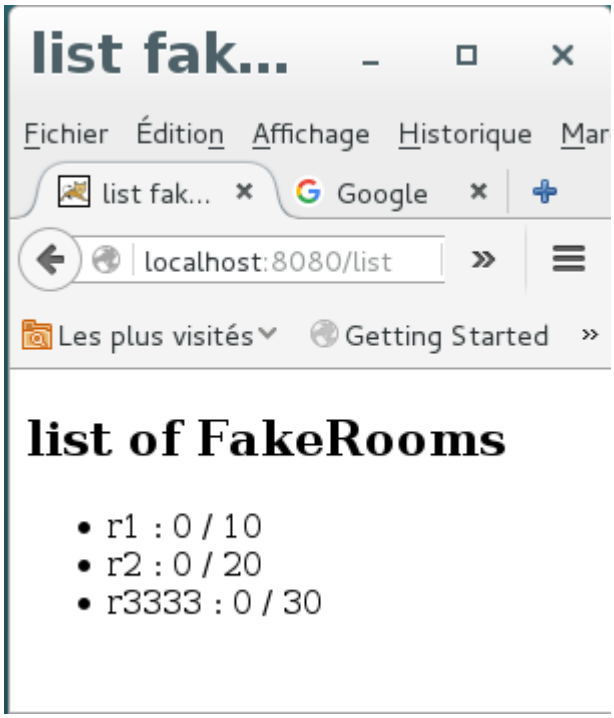
1. Ajouter le dossier `templates` aux *resources* du projet 1. Ajouter le *template freemarker* `home.ftl`

```
<!DOCTYPE html>
<html lang="fr">

<head>
  <title>${title}</title>
</head>
<body>
<h2>${now}</h2>
</body>
</html>
```

1. `Packager` et `Déployer` l'application dans Tomcat à l'aide du gestionnaire d'applications Web de Tomcat 1. Aller sur l'URL http://localhost:8080/manager/ 1. Choisir le *Fichier WAR à déployer* 1. Testez le fonctionnement

2. Créer un autre servlet affichant une liste d'objets de taille quelconque en utilisant le *template freemarker* `listRoom.ftl` ou un template similaire

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8"/>
<head>
   <title>${title}</title>
</head>
<body>
<h2> list of FakeRooms</h2>
<p></p>
<ul>
   <#list fakeRooms as room>
      <li>
         ${room.name} : ${room.occupation} / ${room.capacity}
      </li>
   </#list>
</ul>
</body>
</html>
```



1. Commiter et pousser

QR Code avec zxing

Les QR Code sont devenus incontournables dans les applications modernes.

Objectif : Intégrer la librairie de génération de QR Codes zxing (30 minutes)

1. Ajouter les dépendances de zxing dans le `pom.xml`

```
    ...
<!-- QR Code management -->
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>core</artifactId>
  <version>3.2.0</version>
</dependency>
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>javase</artifactId>
  <version>3.2.0</version>
</dependency>
...
```

1. Rajouter la servlet de génération de QR

```
@WebServlet(name = "qrcode", urlPatterns = {"/qrcode"})
public class QRCodeServlet extends HttpServlet {
  @Override
  protected void doGet(final HttpServletRequest request,
    final HttpServletResponse response)
        throws ServletException, IOException {
    QRCodeWriter writer = new QRCodeWriter();
  BitMatrix bitMatrix = null;
  try {
    String url = "coucou";
    bitMatrix = writer.encode(url, BarcodeFormat.QR_CODE, 300, 300);
  } catch (WriterException e) { e.printStackTrace(); }
  response.setContentType("image/png");
  MatrixToImageWriter.writeToStream(bitMatrix, "png",
      response.getOutputStream());
  }
}
```

1. Commiter et pousser. 2.

Authentification

- Comme dans toute application de type entreprise, il est nécessaire d'authentifier les utilisateurs.
- En effet ils ne peuvent pas effectuer toutes les actions possibles (*i.e.* appeler toutes les méthodes de tous les objets).
- Les conteneurs Java EE permettent donc d'utiliser différentes méthodes d'authentification.

Plusieurs couches de technologies interviennent dans le mécansime d'authentification.

Contraintes de sécurité

- Les contraintes de sécurité permettent de faire un mapping entre une URL et des privilèges d'accès
- Utilisation à l'aide d'annotations `@HttpConstraint` ou `@HttpMethodConstraint` si on utilise des servlets.

```
// Seuls les utilisateurs avec le rôle adminApp ont accès à la servlet
@WebServlet("/manage")
@ServletSecurity(@HttpConstraint(rolesAllowed = "adminApp"))
public class AdminServlet extends HttpServlet {
     // servlet code...
}


// Seuls les utilisateurs avec le rôle adminApp peuvent faire du GET et du POST
// Le POST est nécessairement sur une connexion chiffrée
@WebServlet("/manage")
@ServletSecurity(
  httpMethodConstraints = {
    @HttpMethodConstraint(value = "GET", rolesAllowed = "adminApp"),
    @HttpMethodConstraint(value = "POST", rolesAllowed = "adminApp",
              transportGuarantee = TransportGuarantee.CONFIDENTIAL),
  }
)
public class AdminServlet extends HttpServlet {
   // servlet code...
}
```

- Ou bien à l'aide du fichier `web.xml` et de différentes directives que l'on utilise des servlets ou pas (`security-constraint` , `web-resource-collection` , `auth-constraint` , `user-data-constraint` [2]

```
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint> <!-- spécifie l'utilisation de contraintes -->
   <web-resource-collection> <!-- URLs concernées -->
      <web-resource-name>wholesale</web-resource-name> <!-- nom -->
      <url-pattern>/company/wholesale/*</url-pattern> <!-- motif -->
   </web-resource-collection>
   <auth-constraint> <!-- contrainte d'authentification (qui peut) -->
      <role-name>PARTNER</role-name>
   </auth-constraint>
   <user-data-constraint> <!-- contrainte de transport (chiffrement) -->
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>

<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
   <web-resource-collection>
      <web-resource-name>retail</web-resource-name>
      <url-pattern>/company/retail/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
```

```
      <role-name>CLIENT</role-name>
   </auth-constraint>
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

- EJB3 peux gérer les contraintes de sécurité de façon plus fine (`@PermitAll` , `@DenyAll` , `@RolesAllowed` , `@DeclareRoles` , `@RunAs`)

Modalités d'authentification

- Afin de gérer les accès, il est nécessaire que le conteur web dispose d'un mécanisme d'authentification. - Plusieurs modes d'authentification sont disponibles :

```
- *Basic authentication* : le serveur demande un identifiant/mot de passe au navigateur
- *Form-based authentication* : le développeur de l'application peut proposer un formulaire intégré dans son application (*i.e.*, page d'authentification)
- *Digest authentication* : similaire à Basic mais sans faire transiter le mot de passe en clair
- *Client authentication* : le serveur authentifie le client grâce à un certificat (comme bitbucket par ssh)
- *Mutual authentication* : le client authentifie le serveur grâce à un certificat puis le serveur authentifie le client grâce à un certificat (ou un identifiant mot de passe)
```

- Les modalités d'authentification sont précisées dans le fichier `web.xml` :

```
<!-- Authentification par formulaire -->
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>file</realm-name>
   <form-login-config>
      <form-login-page>/login.xhtml</form-login-page>
      <form-error-page>/error.xhtml</form-error-page>
   </form-login-config>
</login-config>

<!-- Authentification de type Digest -->
<login-config>
   <auth-method>DIGEST</auth-method>
</login-config>
```

Gestion des utilisateurs et des rôles

Rôles

La liste des rôles requis pour utiliser l'application est listée de la façon suivante dans le `web.xml` :

```
<security-role>
   <role-name>manager</role-name>
</security-role>
<security-role>
```

```
    <role-name>employee</role-name>
</security-role>
```

Royaume

- Les utilisateurs appartiennent à un royaume (realm)[3]
- Chaque utilisateur est associé à une liste de rôles
- La déclaration des contraintes de sécurité est normalisée, mais pas le mécanisme d'accès aux utilisateurs
- Tomcat propose les méthodes suivantes :

  - `JDBCRealm` : informations stockées dans une base de données relationnelle accédée par JDBC - `DataSourceRealm` : informations stockées dans une base de données relationnelle accédée par une ressoource de type JNDI JDBC DataSource. - `JNDIRealm` : informations stockées dans un serveur LDAP - `UserDatabaseRealm` : informations stockées dans une ressource UserDatabase JNDI - `MemoryRealm` : informations stockées dans une collection en mémoire, intialisée depuis un document XML (conf/tomcat-users.xml). **À ne pas utiliser en production** - `JAASRealm` - informations récupérées depuis le framework Java Authentication & Authorization Service (JAAS)

- Il est possible de créer son propre connecteur pour utiliser d'autres sources de données

Exercice

Objectif : Installer une authentification simple et créer un formulaire en utilisant Freemarker, JSP ou HTML.

- Créer un formulaire permettant la saisie d'un nom. La balise `form` doit contenir les attributs suivants : `method="POST" action="j_security_check"` . Les champs de nom et mot de passe doivent avoir respectivement l'atrtibut `name` à la valeur `j_username` et `j_password` .
- Ajouter un utilisateur avec le rôle `application` .
- Sécuriser une servlet (nouvelle ou existante) en autorisant les utilisateurs du rôle `application` .
- Installer une authentification simple (memoryRealm Tomcat) sur le formulaire. L'authentification sera assurée à l'aide de Java EE et du conteneur Tomcat.
- Commiter et pousser
-

Injection de dépendances

- Contrairement aux conteneurs Full Java EE, les conteneurs WEB de type Tomcat n'hébergent pas de bibliothèque d'injection de dépendences - Il est donc nécessaire d'importer une implémentation dans le .war - Nous choisissons Google Guice ; d'autres librairies ont un fonctionnement différent [4]

Mapping de servlets

- En raison de l'injection de dépendances, il n'est plus possible de demander au conteneur WEB d'instancier les servlets - C'est l'injecteur qui doit s'en charger afin qu'il puisse injecter les éventuelles dépendences - Conséquence :

```
- Il n'y a plus de mapping de servlets dans le `web.xml`
- On demande au conteneur de router toutes les requêtes dans un objet de type  `GuiceFilter` via les lignes suivantes dans `web.xml`.
```

```
<filter>
  <filter-name>guiceFilter</filter-name>
  <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- Le mapping est définit dans un `Module` de `Guice` (classe qui hérite de `ServletModule`)

```
public class Module extends ServletModule {
  /*Logger*/
  private static final Logger logger =
       LoggerFactory.getLogger(Module.class);

  @Override
  protected void configureServlets() {
     super.configureServlets();
     serve("/list").with(ListServlet.class);
     serve("/delete").with(DeleteServlet.class);
     logger.info("WebModule configureServlets ended.");
    }
  }
```

Il faut ajouter la dépendance suivante dans `pom.xml` en sus de la dépendance à `guice` et celle au logger.

```
<dependency>
  <groupId>com.google.inject.extensions</groupId>
  <artifactId>guice-servlet</artifactId>
```

```
  <version>5.0.1</version>
</dependency>
```

Les classes `ListServlet` et `DeleteServlet` doivent être des *singleton bean*.

Pour définir une classe comme étant un singleton bean il faut l'annoter avec l'annotation `@Singleton`.

Il y a une seule instance d'un singleton bean par application, cette instance est créée par le container EJB.

Les valeurs des variables d'instance du singleton bean sont conservées entre différents appels du singleton bean.

- L'injecteur est créé et configuré grâce une classe qui hérite de *GuiceServletContextListener*. l'injecteur est créé dès que l'application est déployée avant la moindre requête[5]

```
public class MyGuiceServletConfig extends GuiceServletContextListener {

    @Override
    protected Injector getInjector() {
    return Guice.createInjector(new Module());
  }
}
```

- Cette classe est spécifiée dans le `web.xml` :

```
<listener>
  <listener-class>fr.iut.MyGuiceServletConfig</listener-class>
</listener>
```

Exercice

Objectif : Utiliser un injecteur Guice pour instancier les servlets des exercices précédents. (remarque : les contraintes de sécurité doivent être définies dans le fichier `web.xml`).

- Faites un copier-coller de votre dossier contenant les servelts créées lors de la séance.
- Modifier une version pour utiliser Google guice.
- Commiter et pousser.

1. http://freemarker.org/docs/dgui_template_exp.html
2. http://docs.oracle.com/javaee/6/tutorial/doc/gkbaa.html

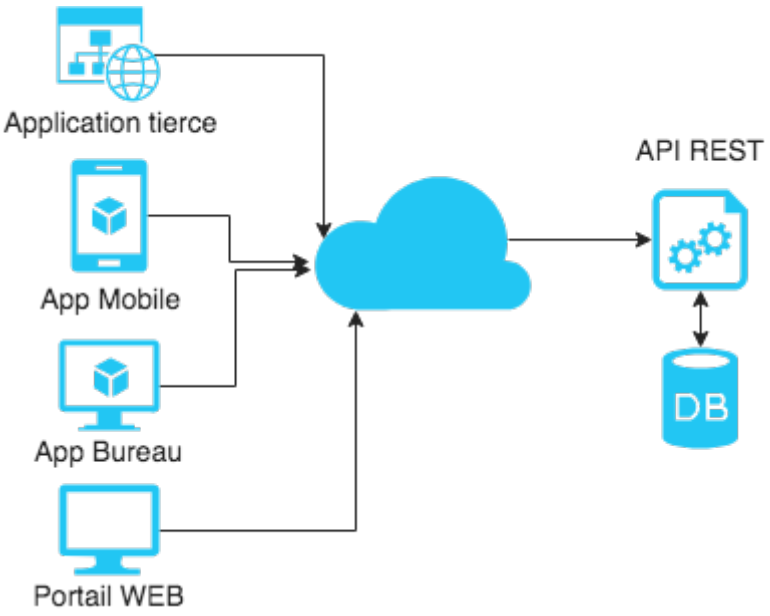
3. http://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html
4. https://github.com/google/guice/wiki/ServletModule
5. http://tomcat.apache.org/tomcat-8.0-doc/config/listeners.html
6. http://fr.wikipedia.org/wiki/Remote_method_invocation_%28Java%29
7. http://fr.wikipedia.org/wiki/Hypertext_Markup_Language
8. http://fr.wikipedia.org/wiki/Representational_State_Transfer
9. http://fr.wikipedia.org/wiki/Servlet
10. http://fr.wikipedia.org/wiki/JavaServer_Pages
11. http://www.oracle.com/technetwork/topics/jsf-092015.html
12. http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope
13. http://freemarker.org/
14. http://velocity.apache.org/engine/releases/velocity-1.4/veltag.html

Web services - API REST

Introduction aux API REST

Une `API` ou Application Programming Interface est une interface permettant d'exposer les fonctions d'un logiciel au monde exterieur.

Une `API` référence les différentes fonctions (services) de l'application qui sont accessibles par le consommateur.



Representational State Transfert[1] (`REST`) est un type d'architecture pour les systèmes distribués qui propose de représenter le modèle de données et les services sous la forme de ressources (web services).

L'architecture `REST` définie un ensemble de conventions et de bonnes pratiques à respecter, il ne s'agit pas d'une technologie à part entière.

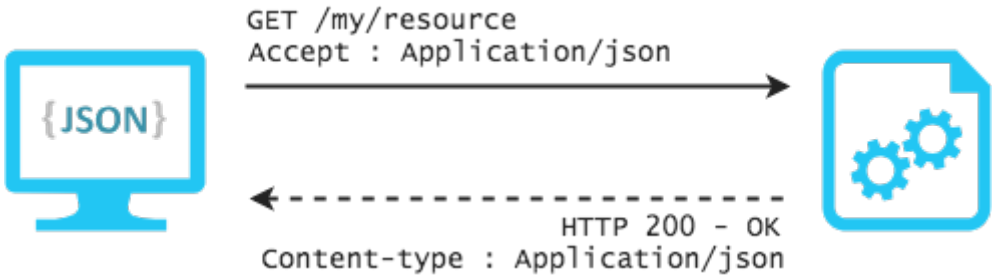
L'architecture `REST` est simple et se repose sur les méthodes standards du protocole `HTTP`, à l'instar des protocoles `SOAP` et `XML-RPC` qui sont des surcouches basées sur le procole `HTTP`.

`REST` est une architecture sans état (stateless) du côté du serveur. Le fait d'être «sans état» signifie que le serveur ne mémorise pas l'état du client entre deux requêtes. Du point de vue du serveur, chaque requête est une entité distincte des autres.

Le format d'échange entre le client et le serveur le plus couramment utilisé pour les API `REST` est le `JSON`[2](JavaScript Object Notation).

Une ressource `REST` c'est :

- Une `URI` : représente une ressource unique sur le système.
- Un verbe `HTTP` : identifie l'opération à réaliser.
- Des entêtes `HTTP` : défini le format du contenu d'échange, le jeton d'authentification, la version de la ressource à utiliser etc...
- Des paramètres de requêtes : définissent des paramètres facultatifs à la ressource.
- Une réponse `HTTP` : représente l'état de la ressource (via un code `HTTP` et du contenu).



L'`URI` comme identifiant des ressources

L'apporche `REST` se base sur les `URI` (Uniform Resource Identifier) afin d'identifier une ressource.

Les ressources (`URI`) doivent rester simple cf. KISS (Keep It Simple, Stupid) :

Une opération = une ressource.

Les ressources sont basées sur des noms et non des verbes, c'est la méthode `HTTP` qui determine l'opération à réaliser.

Quelques exemples de construction d'`URL` `REST` correspondant aux fonctionnalités de l'application Room Manager vue dans les séances précédentes :

Cible toutes les salles :

```
http://api.example.com/rooms
                ^^^^^
```

Cible une salle (d'identifiant 12):

```
http://api.example.com/rooms/12
                ^^^^^^^^
```

Cible les évènements d'entrée/sortie pour une salle :

```
http://api.example.com/rooms/12/events
                ^^^^^^^^^^^^^^^^
```

Cible l'évènement d'entrée/sortie (d'identifiant 345) pour une salle :

```
http://api.example.com/rooms/12/events/345
                ^^^^^^^^^^^^^^^^^^^^
```

Les verbes `HTTP` comme identifiant des opérations

L'approche `REST` se base sur la méthode `HTTP` pour determiner l'opération à réaliser.

L'utilisation des verbes `HTTP` rend l'API intuitive et permet d'éviter que le développeur n'ait à consulter une documentation verbeuse pour comprendre comment manipuler les ressources.

Les verbes `HTTP` correspondant aux opérations de type `CRUD` :

- Créer (Create) : `POST`
- Afficher (Read) : `GET`
- Mettre à jour (Update) : `PUT`
- Mettre à jour partiellement (Update) : `PATCH`
- Supprimer (Delete) : `DELETE`

Ressources et actions associées en fonction du verbe `HTTP` utilisé dans room manager

Ressource\Action	**POST**	**GET**	**PUT**	**DELETE**
`/rooms`	Crée une salle	Liste toutes les salles	Met à jour toutes les salles	Supprime toutes les salles
`/rooms/12`	-	Lit la salle 12	Met à jour la salle 12	Supprime la salle 12
`/rooms/12/events`	Ajoute un évènement à la salle 12	Liste tous les évènements de la salle 12	Met à jour tous les évènements de la salle 12	Supprime tous les évènements de la salle 12
`/rooms/12/events/345`	-	Lit l'évènement	Met à jour l'évènement	Supprime l'évènement

Ressource\Action	POST	GET	PUT	DELETE
		345 de la salle 12	345 de la salle 12	345 de la salle 12

Ainsi pour une même `URI` plusieurs actions sont disponibles :

Afficher la liste des salles

```
GET http://api.example.com/rooms
```

Contenu de retour :

```
[
   {
      "id" : 12,
      "name" : "Salle 307"
   },
   {
      "id" : 13,
      "name" : "Salle 311",
      "description" : "La salle de TD JEE"
   }
]
```

Code retour

```
200
```

Ajouter une salle

```
POST http://api.example.com/rooms
```

Contenu de reqûete :

```
{
   "name" : "Salle 42"
}
```

Code retour

```
200
```

Contenu de retour

```
{
  "id" : 14
}
```

Les paramètres de requetes (Query strings)

Les paramètres de reqûetes viennent en complément de l'URI, ils peuvent nottament servir à ajouter des fonctions :

- De recherche / filtre,
- De tri,
- De pagination

Dans l'exemple suivant :

```
GET http://api.example.com/rooms?search=informatique&order=asc
```

- Le paramètre `search` permet de filtrer les résultats contenant le mot clé fourni : `informatique`.
- Le paramètre `order` permet de trier les résultats par ordre croissant/décroissant.

Les entêtes (headers)

Les entêtes `HTTP` permettent de fournir des informations complémentaires sur la nature de l'échange entre le client et l'API.

Par exemple, l'appelant doit définir quel format de réponse il souhaite reçevoir via l'entête `Accept` (à condition que le format de réponse demandé soit supporté par l'API) :

- Entête permettant de demander le contenu de réponse au format JSON : `Accept: application/json`,
- Entête permettant de demander le contenu de réponse au format XML : `Accept: application/xml`

L'entête `Accept-Language` permet de préciser dans quelle langue le client souhaite recevoir le contenu.

L'entête `Authorization` permet de définir les informations d'authentification.

Les codes retour

La méthode `REST` se base sur les codes retour `HTTP` pour préciser le statut d'une ressource suite à l'exécution d'une requête.

Il existe plus de 70 codes `HTTP` regroupés en 5 grandes familles, ils sont composés de trois chiffres et sont catégorisés en fonction du chiffre des centaines :

- `1xx` - Réponses informatives.
- `2xx` - Réponses de succès.
- `3xx` - Réponses de redirection.
- `4xx` - Réponses d'erreur côté client.
- `5xx` - Réponses d'erreur côté serveur.

Les codes retour les plus couramment utilisés dans les API `REST sont les suivants :

- `200` - OK - code de base indiquant la réussite d'une requête.
- `201` - Created - indique que la requête a réussi et qu'une ressource a été créée en conséquence.
- `204` - No Content - indique que la requête a réussi et que le serveur ne renvoie pas de contenu (requête `PUT`).
- `400` - Bad Request - indique que le serveur ne peut pas comprendre la requête en raison d'une syntaxe invalide.
- `404` – Not Found - indique qu'un serveur ne peut pas trouver la ressource demandée.
- `401` - Unauthorized - indique que la requête n'a pas été effectuée car il manque des informations d'authentification valides.
- `403` - Forbidden - indique qu'un serveur comprend la requête mais refuse de l'autoriser (problème de droit d'accès).
- `415` - Unsupported Media Type - indique que le serveur refuse la requête car le format de contenu demandé n'est pas pris en charge.
- `500` - Internal Server Error - indique que le serveur a rencontré un problème inattendu qui l'empêche de répondre à la requête.

API `REST` en JAVA : JSR-311 JAX-RS

La spécification concernant les API `REST` en JEE est la JSR 311[4]: *JAX-RS: The JavaTM API for `REST`ful Web Services*

Les JSR 339 et 370 sont des mises à jour de la JSR 311.

La construction d'une ressource `REST` en java à l'aide de JAX-RS est réalisé de la façon suivante :

- L'`URI` / le chemin de la ressource est défini par l'annotation `@Path`
- Les verbes `HTTP` sont définis par les annotations suivantes * `@GET` * `@POST` * `@PUT` * `@DELETE` * `@HEAD` * `@PATCH`

- Les variables déclarées dans l'`URI` de la ressource `@Path` sont définies via l'annotation `@PathParam`
- Les paramètres de requêtes (query parameters) sont définis via l'annotation `@QueryParam`
- Les types de médias[3] (MIME types) consommés en entrée et délivrés en sortie sont définis respectivement par les annotations `@Consumes` et `@Produces`

```
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import com.sun.jersey.api.NotFoundException;

@Path("/rooms") // Le point d'entrée de toutes les ressources déclarées dans la classe
public class RoomResource {

    /**
     * Retrieve a room by its name.
     *
     * @param roomId room identifier
     * @return found room
     */
    @GET // Précise qu'il s'agit d'une ressource de lecture (READ)
    @Path("/{roomId}") // Le point d'entrée déclare le PathParam roomId
    @Produces({MediaType.APPLICATION_JSON + "; charset=UTF-8"}) // (Response Header) la resource GET renvoie du contenu de type "application/json"
    public RoomDetailsVO getRoom(@PathParam(value = "roomId") long roomId) { // Le Path Param roomId est de type "long", une ressource valide est donc : /rooms/12
        logger.debug("Retrieve room with id {}", roomId);
        //Retrieve rooms from DB
        final Room room = roomDao.get(roomId);
        if (room == null) {
            throw new NotFoundException("Room with id " + roomId + " does not exist");
        }


        // TODO : Convert rooms into visual object (select only necessary fields)
        RoomDetailsVO roomVO = new RoomDetailsVO();

        //Return room
        return roomVO;
    }

}
```

Plusieurs librairies implémentent la spécification JAX-RS.
Dans le cadre du TD nous utiliserons le projet JERSEY 1.19[5] fourni par Oracle / Glassfish

Les dépendances Jersey à ajouter dans un projet WEB géré par Guice sont les suivantes :

```
<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-server</artifactId>
   <version>1.9</version>
</dependency>

<dependency>
   <groupId>com.sun.jersey.contribs</groupId>
   <artifactId>jersey-guice</artifactId>
   <version>1.9</version>
</dependency>

<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-json</artifactId>
   <version>1.9</version>
</dependency>
```

A noter que la compatibilité entre Jersey et Guice est assurée par une librairie spécifique : `com.sun.jersey.contribs:jersey-guice`
Celle-ci contient entre-autres une classe déclarant un module Guice spécifique pour Jersey : `JerseyServletModule`

En effet, c'est le filtre de servlet Guice qui intercepte toutes les requêtes `HTTP` (cf. configuration dans le fichier de description `web.xml`) et qui déclare le listener de contexte de Servlet Guice `fr.iut.rm.web.WebGuiceServletConfig`

Le listener crée l'injecteur Guice et les différents modules du projet dont le module spécifique pour la couche d'API : `RestServletModule`

```
    this.currentInjector = Guice.createInjector(new MainModule(), new WebModule(interceptors), new
RestServletModule());
```

Contenu du module Guice de configuration de Jersey `RestServletModule` :

```
import com.google.inject.persist.PersistFilter;
import com.google.inject.persist.jpa.JpaPersistModule;
import com.sun.jersey.guice.JerseyServletModule;
import com.sun.jersey.guice.spi.container.servlet.GuiceContainer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.servlet.http.HttpServlet;

/**
 * Servlet module used to serve REST resources
 */
public class RestServletModule extends JerseyServletModule {

    /**
```

```
     * Logger
     */
    private static final Logger logger = LoggerFactory.getLogger(RestServletModule.class);

    @Override
    protected void configureServlets() {
        logger.debug("Configuration started");

        //JPA management
        install(new JpaPersistModule("room-manager"));
        filter("/*").through(PersistFilter.class);

        // Route all requests through GuiceContainer
        serve("/*").with((Class<? extends HttpServlet>) GuiceContainer.class);

        logger.debug("Configuration done");
    }

}
```

Le module démarre également le moteur de persistance JPA (`JpaPersistModule`)

Exercice

L'exercice consiste à récupérer le projet room-manager-api-src.tar.gz puis d'enrichir les ressources d'API contenues dans la classe `RoomResource`

1 - Enrichir la méthode existante de création de salle `createRoom`, pour permettre l'enregistrement d'une salle en base de données.

Utiliser la commande `curl` suivante pour valider l'enregistrement d'une salle :

```
curl -X POST \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-d '{"name":"Salle 311","description":"La salle de TD JEE"}' \
"http://localhost:8080/room-manager-api/rooms"
```

Utiliser la commande `curl` suivante pour Lister toutes les salles

```
curl -H "Accept: application/json" \
"http://localhost:8080/room-manager-api/rooms"
```

2 - Enrichir la méthode de lecture d'une salle `getRoom`

Utiliser la commande `curl` suivante pour la récupération d'une salle en prenant soin de selectionner un identifiant valide

```
curl -H "Accept: application/json" \
"http://localhost:8080/room-manager-api/rooms/12"
```

3 - Enrichir la ressource permettant de lister toutes les salles `listRooms` en y ajoutant un paramètre de filtre sur le nom de la salle

Choisissez un nom de paramètre (exemple `q`) puis testez le à l'aide la commande suivante :

```
curl -H "Accept: application/json" \
"http://localhost:8080/room-manager-api/rooms?q=marecherche"
```

4 - Ajouter la ressource permettant d'ajouter un accès à une salle existante

Un nouvel évènement du point de vue l'API est composé des informations suivantes :

- nom de l'utilisateur
- type d'accès (IN/OUT)

L'identifiant de la salle concernée est récupérée dans l'`URI`, La date de l'accès est positionné dans la méthode lors de la création de l'évènement en base.

La ressource doit renvoyer un code retour http 200 et l'identifiant de l'accès nouvellement créé.

La ressource doit renvoyer un code retour http 404 si l'accès porte sur une salle inexistante.

Utiliser la commande `curl` suivante pour valider l'enregistrement d'un évènement :

```
curl -X POST \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-d '{"userName":"John DOE","type":"IN"}' \
"http://localhost:8080/room-manager-api/rooms/12/events"
```

5 - Ajouter la ressource permettant de lister tous les évènement d'accès à une salle

Utiliser la commande `curl` suivante pour tester

```
curl -H "Accept: application/json" \
"http://localhost:8080/room-manager-api/rooms/12/events"
```

Ajouter un filtre sur le type d'évènement `type` puis tester avec la commande suivante

```
curl -H "Accept: application/json" \
"http://localhost:8080/room-manager-api/rooms/12/events?type=OUT"
```

6 - Ajouter la ressource permettant de supprimer un évènement d'accès à une salle

Utiliser la commande `curl` suivante pour tester

```
curl -X DELETE \
"http://localhost:8080/room-manager-api/rooms/12/events/42"
```

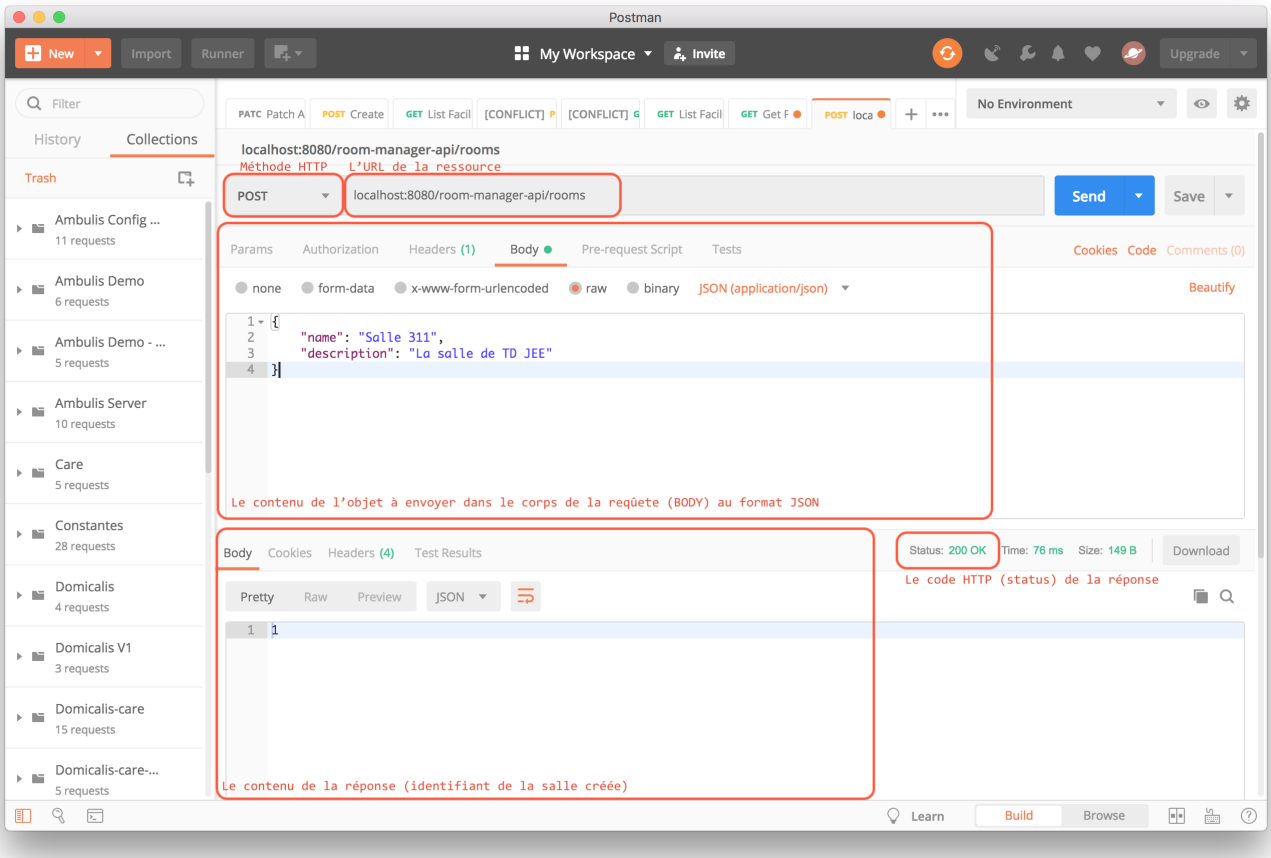
Utilisation d'un client graphique

L'utilisation d'un client d'API graphique peut s'avérer plus confortable que l'outil en ligne de commande cURL.

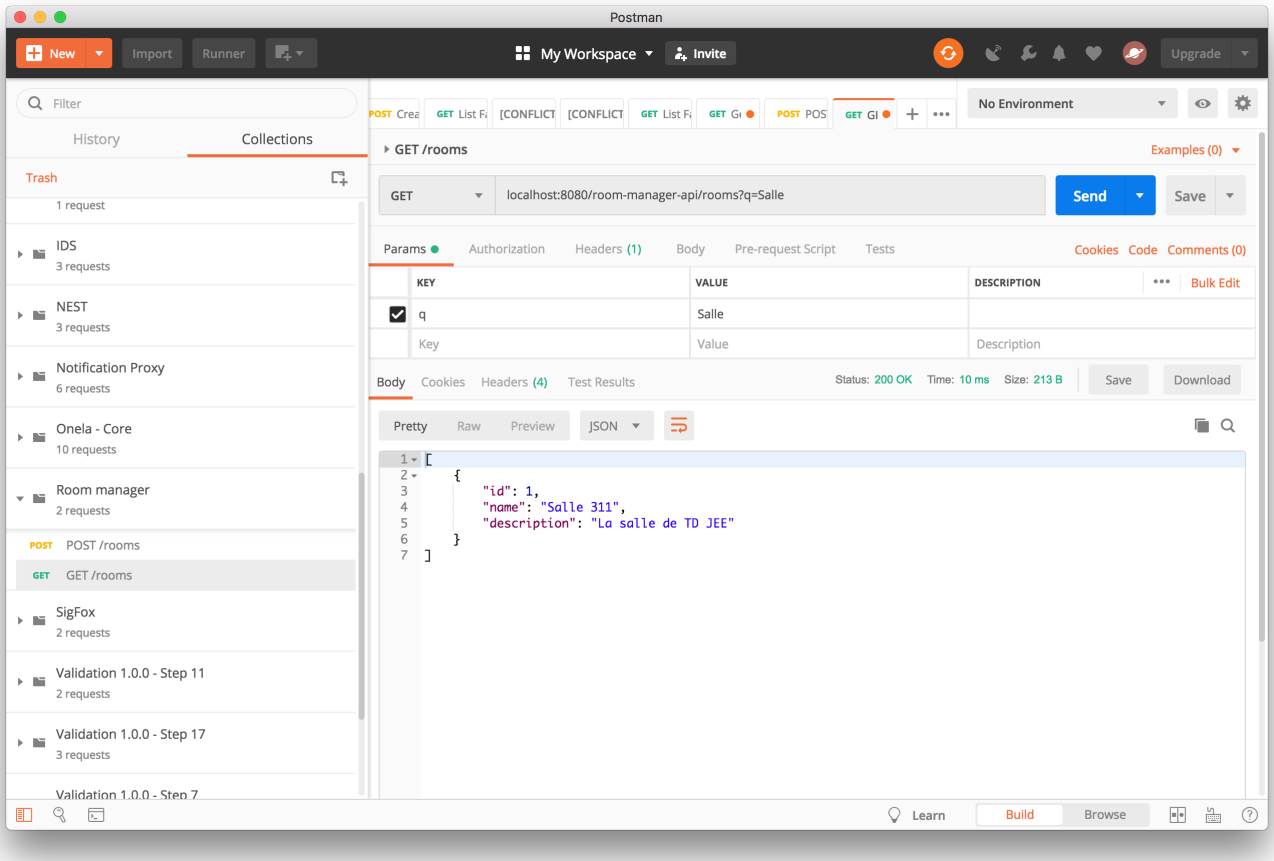
POSTMAN est l'outil graphique de référence permettant de gérer l'ensemble des interactions avec les API REST.

Quelques exemples d'utilisation

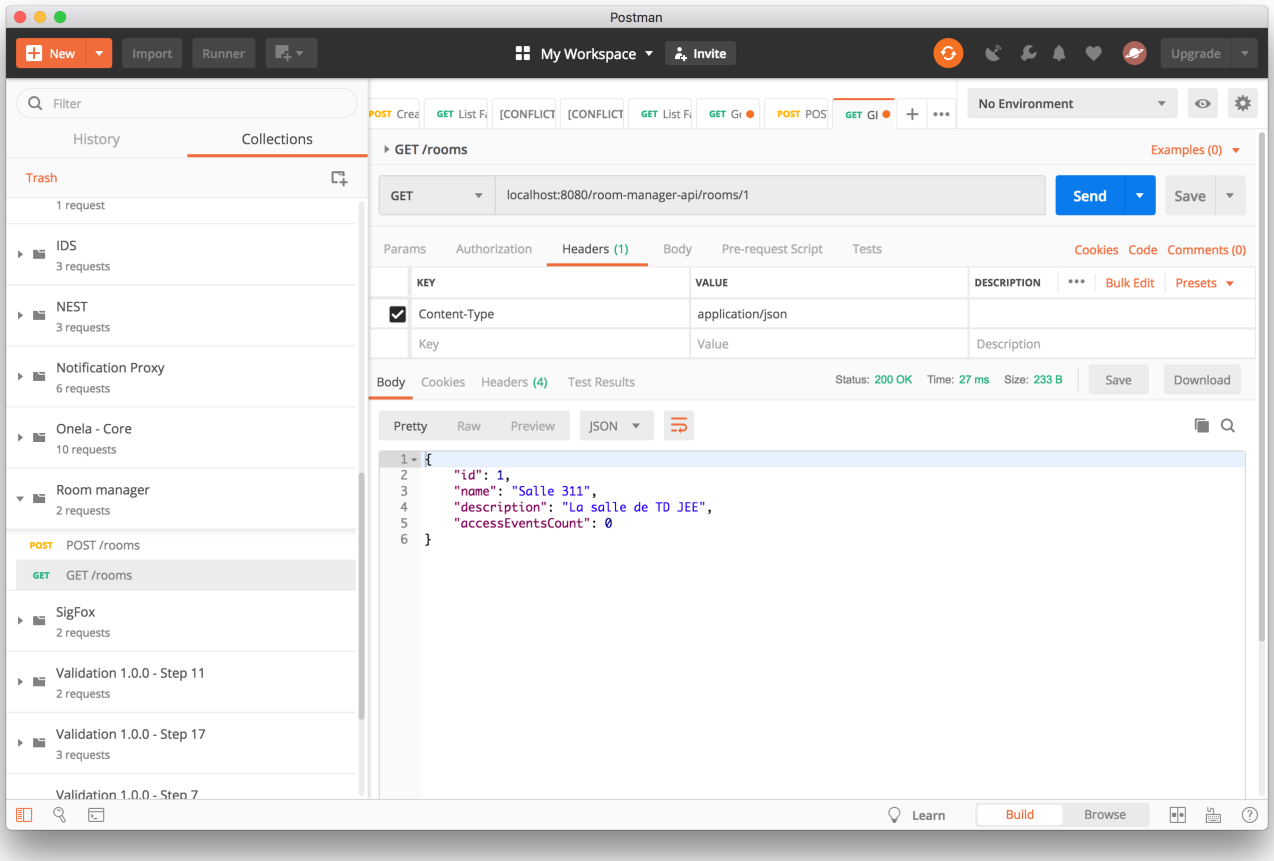
Création d'une salle `POST /rooms` :



Lecture de la liste des salles `GET /rooms` :



Lecture de la salle #1 `GET /rooms/1` :



1. https://fr.wikipedia.org/wiki/Representational_state_transfer
2. https://www.json.org/json-fr.html
3. https://developer.mozilla.org/fr/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types
4. https://jcp.org/en/jsr/detail?id=311
5. https://jersey.github.io/
6. https://www.getpostman.com/downloads/
7. https://developer.paypal.com/docs/api/overview/
8. https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol#M%C3%A9thodes
9. https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cp1v/index.html
10. https://openclassrooms.com/fr/courses/3449001-utilisez-des-api-rest-dans-vos-projets-web/3449008-quest-ce-quune-api
11. https://blog.octo.com/designer-une-api-rest/