

EI5 AGI 1415 - ISTIA

TP Java EE n° 1 - septembre 2014

Objectifs :

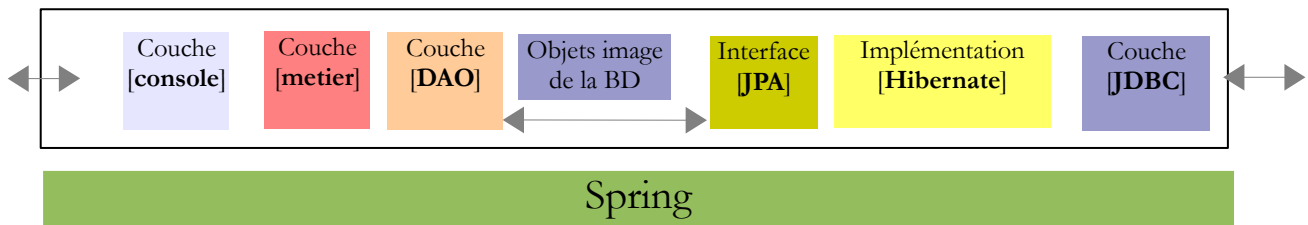
- renforcer les connaissances acquises dans le TD en abordant celui-ci sous un autre angle ;
- découvrir une architecture client / serveur, où le serveur est un service web / JSON ;
- découvrir comment configurer Spring sans fichiers XML ;
- découvrir comment accéder aux bases de données avec Spring Data ;
- découvrir l'IDE IntelliJ IDEA Community Edition ;

Références :

Ce document utilise des technologies décrites dans le document « Un exemple de client / serveur AngularJS - Spring 4 » à l'URL [\[http://tahe.developpez.com/angularjs-spring4\]](http://tahe.developpez.com/angularjs-spring4). On notera ce document par la suite [\[ref1\]](#).

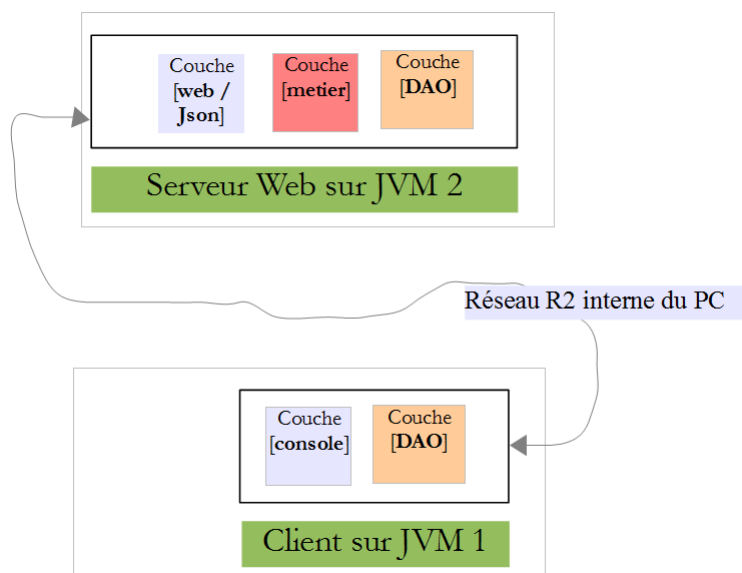
1 Le problème

Il s'agit de refaire l'application du paragraphe 4.11 du TD :



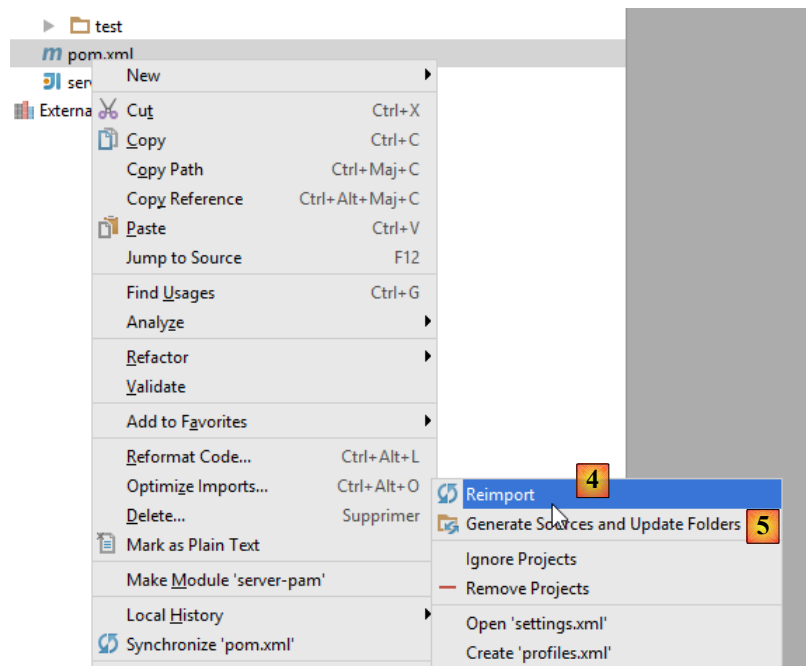
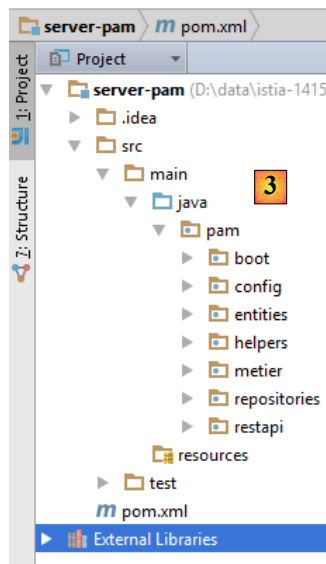
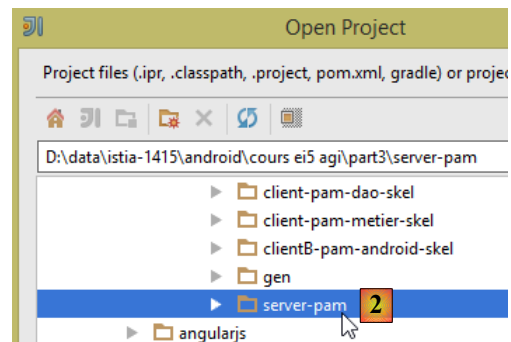
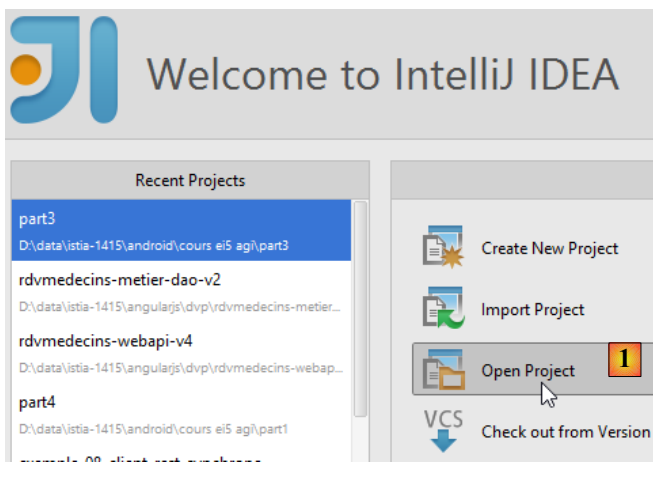
L'architecture ci-dessus s'exécute dans une unique JVM (Java Virtual Machine). Nous allons la porter vers une architecture client / serveur où le client et le serveur s'exécuteront dans deux JVM différentes.

La nouvelle architecture sera la suivante :



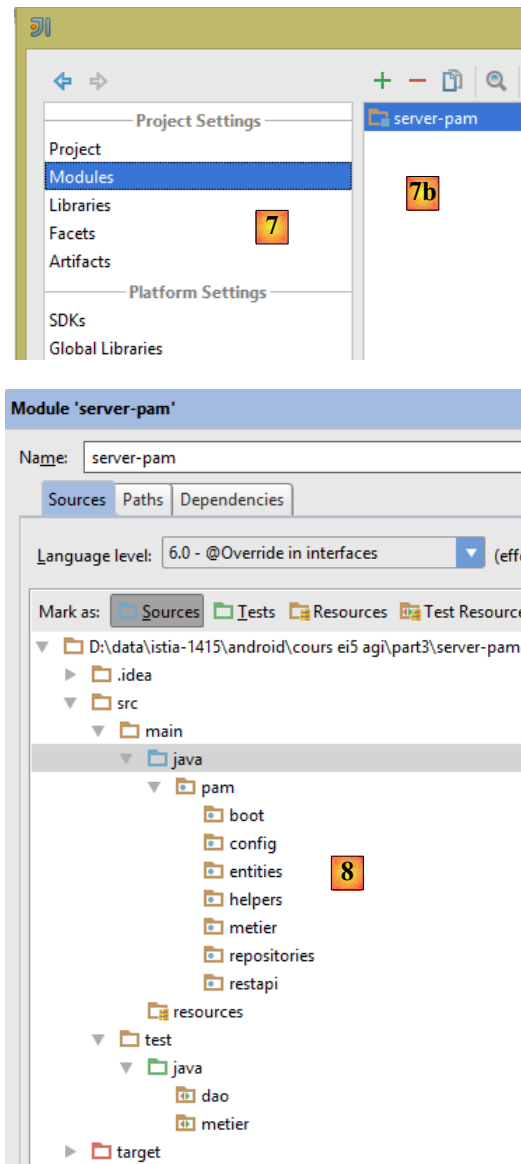
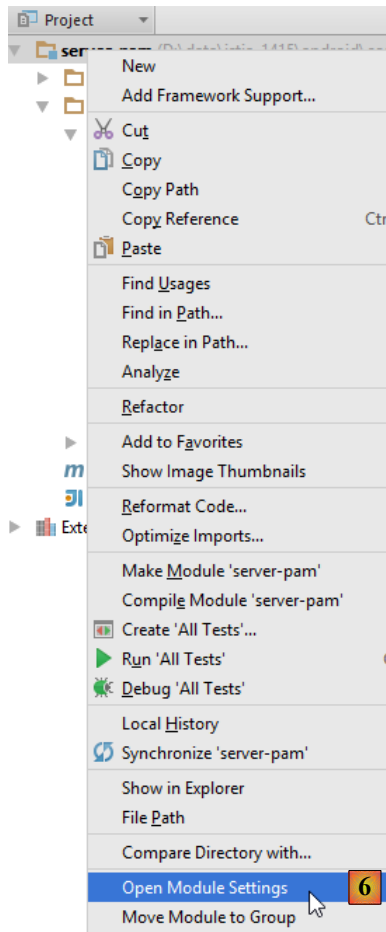
2 Introduction à l'IDE IntelliJ IDEA Community Edition

Nous allons donner ici quelques informations sur l'IDE IntelliJ IDEA Community Edition. Une fois lancé l'IDE, on charge le projet [server-pam] qui vous sera donné :



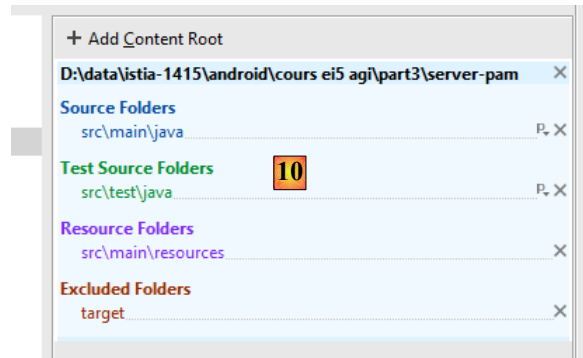
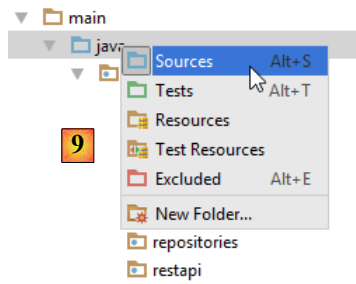
- en [3], on a un projet Maven (présence d'un fichier [pom.xml]). Lorsqu'il est modifié ou qu'on a l'impression que les dépendances du [pom.xml] n'ont pas été importées, on peut réimporter celles-ci [4] voire de régénérer certains fichiers source gérés par Maven [5].

Dans un projet Maven, il est important de spécifier les dossiers qui contiennent le code Java :



- en [6], on va dans les propriétés du projet (clic droit) et on sélectionne [Open Module Settings] [6] ;
- en [7], on a un aperçu des modules (= projets) ouverts. Ici il n'y en a qu'un [7b] ;
- en [8], l'arborescence du projet :
 - en bleu, les dossiers des codes source,
 - en vert, les dossiers des codes source de la branche [test],

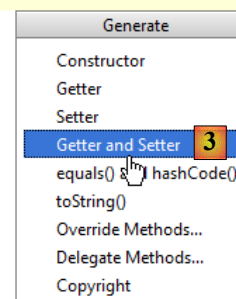
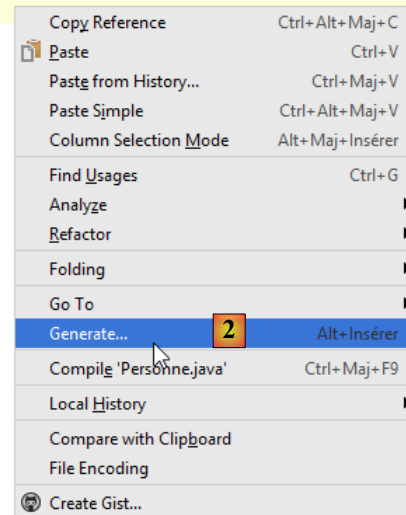
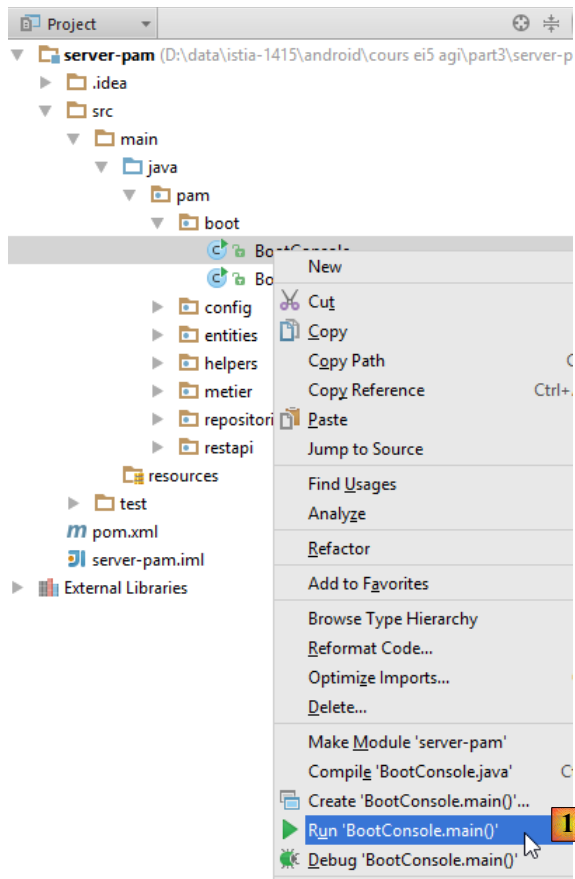
Pour fixer la nature d'un dossier, on clique droit dessus et on choisit sa nature [9] :



- en [10] : un résumé des différents types de dossier. Ici, on voit que :
 - les sources Java du projet sont dans le dossier [src/main/java],
 - les sources Java des tests sont dans le dossier [src/test/java],
 - les ressources sont dans le dossier [src/main/resources]. Une ressource est un fichier qui doit être placé dans le Classpath du projet mais qui n'est pas un fichier de code Java. Un exemple est le fichier XML de configuration de Spring. S'il y en a un, c'est là qu'il doit être placé,
 - le dossier [target] est le dossier dans lequel Maven place le produit de la compilation (build, make) du projet. Il est noté [excluded] car son contenu ne doit pas participer aux builds ;

Lorsqu'on importe un projet Maven (Eclipse, Netbeans) dans IntelliJ IDEA, il est ensuite important de vérifier cette configuration. Le projet importé peut ne pas fonctionner tout simplement parce que sa configuration de build (sources, tests, ressources) est incorrecte.

Pour exécuter une classe exécutable, il suffit de cliquer droit dessus comme montré ci-dessous en [1] :



Pour générer les getters / setters d'une classe on utilisera la procédure [2] et [3].

Pour mettre en forme le code : Ctrl-Maj-F.

3 Manipuler du JSON

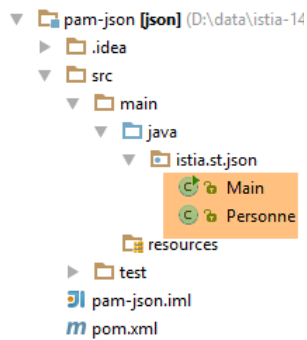
Le TP utilise la bibliothèque JSON [Jackson]. Cette bibliothèque est utilisée de façon transparente pour le développeur par le framework [Spring MVC]. Vous n'avez jamais dans ce TP à manipuler du JSON vous-mêmes. Pour illustrer ce qu'est le JSON (JavaScript Object Notation), nous présentons un programme qui sérialise des objets en JSON et fait l'inverse en désérialisant les chaînes JSON produites pour recréer les objets initiaux.

Il existe diverses bibliothèques capable de traiter du JSON. Nous utilisons celle appelée 'Jackson'.

La bibliothèque 'Jackson' permet de construire :

- la chaîne JSON d'un objet : `new ObjectMapper().writeValueAsString(object)` ;
- un objet à partir d'un chaîne JSON : `new ObjectMapper().readValue(jsonString, Object.class)`.

Les deux méthodes sont susceptibles de lancer une `IOException`. Voici un exemple.



Le projet ci-dessus est un projet Maven avec le fichier [pom.xml] suivant ;

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.   <modelVersion>4.0.0</modelVersion>
6.
7.   <groupId>istia.st.pam</groupId>
8.   <artifactId>json</artifactId>
9.   <version>1.0-SNAPSHOT</version>
10.
11.  <dependencies>
12.    <dependency>
13.      <groupId>com.fasterxml.jackson.core</groupId>
14.      <artifactId>jackson-databind</artifactId>
15.      <version>2.3.3</version>
16.    </dependency>
17.  </dependencies>
18. </project>
```

- lignes 12-16 : la dépendance qui amène la bibliothèque 'Jackson' ;

La classe [Personne] est la suivante :

```
1. package istia.st.json;
2.
3. public class Personne {
4.     // data
5.     private String nom;
6.     private String prenom;
7.     private int age;
8.
9.     // constructeurs
10.    public Personne() {
11.
12.    }
13.
14.    public Personne(String nom, String prénom, int âge) {
15.        this.nom = nom;
16.        this.prenom = prénom;
17.        this.age = âge;
18.    }
19.
20.    // signature
21.    public String toString() {
22.        return String.format("Personne[%s, %s, %d]", nom, prenom, age);
23.    }
```

```

24.
25.     // getters et setters
26. ...
27. }

```

La classe [Main] est la suivante :

```

1. package istia.st.json;
2.
3. import com.fasterxml.jackson.databind.ObjectMapper;
4.
5. import java.io.IOException;
6. import java.util.HashMap;
7. import java.util.Map;
8.
9. public class Main {
10.     // l'outil de sérialisation / désérialisation
11.     static ObjectMapper mapper = new ObjectMapper();
12.
13.     public static void main(String[] args) throws IOException {
14.         // création d'une personne
15.         Personne paul = new Personne("Denis", "Paul", 40);
16.         // affichage Json
17.         String json = mapper.writeValueAsString(paul);
18.         System.out.println("Json=" + json);
19.         // instanciation Personne à partir du Json
20.         Personne p = mapper.readValue(json, Personne.class);
21.         // affichage personne
22.         System.out.println("Personne=" + p);
23.         // un tableau
24.         Personne virginie = new Personne("Radot", "Virginie", 20);
25.         Personne[] personnes = new Personne[]{paul, virginie};
26.         // affichage Json
27.         json = mapper.writeValueAsString(personnes);
28.         System.out.println("Json personnes=" + json);
29.         // dictionnaire
30.         Map<String, Personne> hpersonnes = new HashMap<String, Personne>();
31.         hpersonnes.put("1", paul);
32.         hpersonnes.put("2", virginie);
33.         // affichage Json
34.         json = mapper.writeValueAsString(hpersonnes);
35.         System.out.println("Json hpersonnes=" + json);
36.     }
37. }

```

L'exécution de cette classe produit l'affichage écran suivant :

```

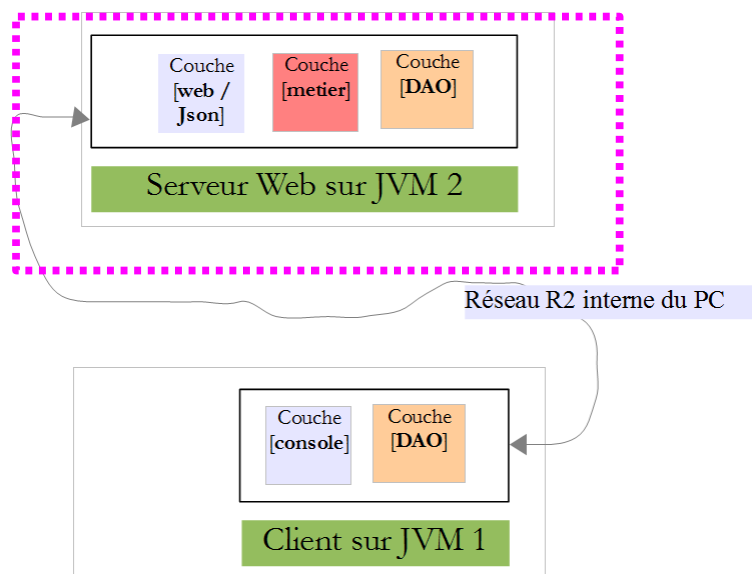
1. Json={"nom":"Denis","prenom":"Paul","age":40}
2. Personne=Personne[Denis, Paul, 40]
3. Json personnes=[{"nom":"Denis","prenom":"Paul","age":40},
  {"nom":"Radot","prenom":"Virginie","age":20}]
4. Json hpersonnes={"2":{"nom":"Radot","prenom":"Virginie","age":20},"1":
  {"nom":"Denis","prenom":"Paul","age":40}}

```

De l'exemple on retiendra :

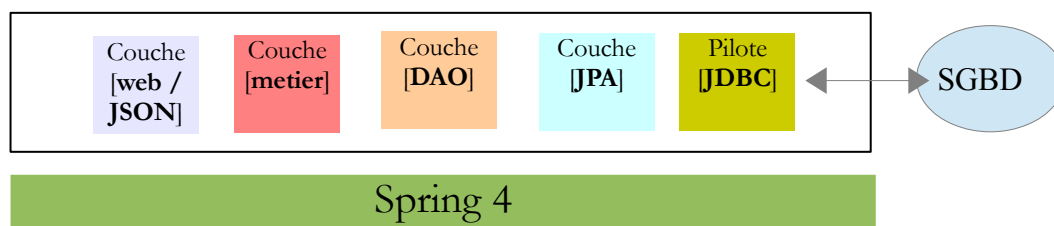
- l'objet [ObjectMapper] nécessaire aux transformations JSON / Object : ligne 11 ;
- la transformation [Personne] --> JSON : ligne 17 ;
- la transformation JSON --> [Personne] : ligne 20 ;
- l'exception [IOException] lancée par les deux méthodes : ligne 13.

4 Le serveur web/JSON



Lectures nécessaires : chapitre 2 de [\[ref1\]](#).

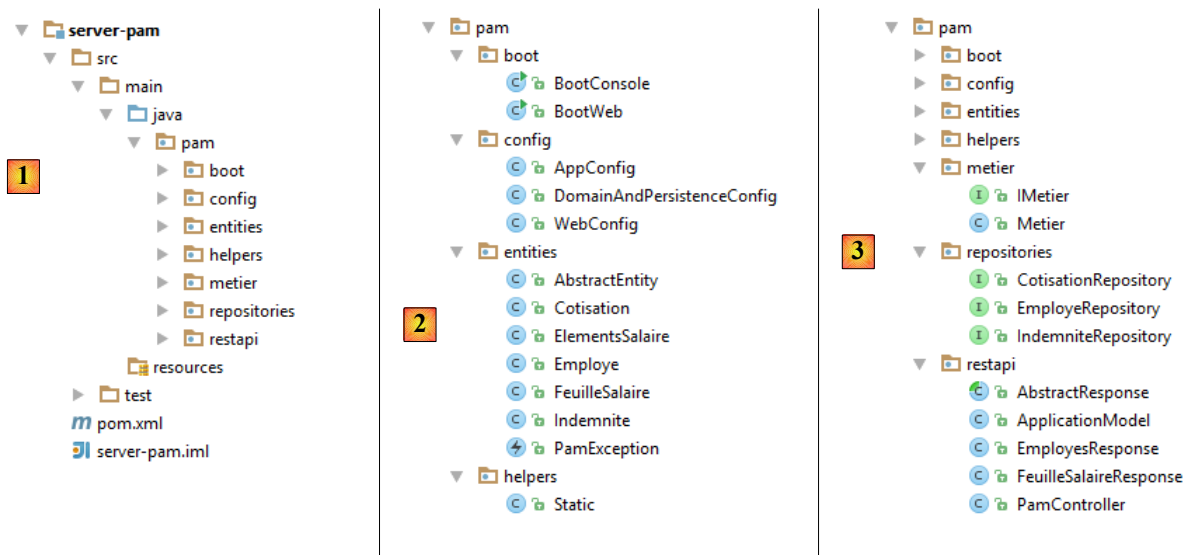
Le serveur aura l'architecture suivante :



La base de données est la base [dbpam_hibernate] du TD.

4.1 Le projet IntelliJ IDEA du serveur

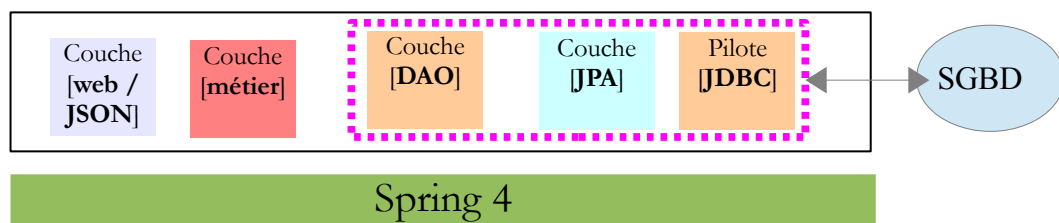
Une fois complété, le projet [server-pam] qui vous aura été donné aura l'allure suivante :



- [1] : le projet dans son ensemble ;
- [2] : contient des éléments qui vous sont donnés :
 - le package [boot] contient les classes exécutables : [BootConsole] est un programme console qui teste la couche [métier] ainsi que les interfaces du dossier [repositories] [3]. [BootWeb] est le programme qui lance le service web/JSON ;
 - le package [config] contient les classes de configuration de Spring :
 - [DomainAndPersistenceConfig] configure la couche [métier] ainsi que les interfaces du dossier [repositories],
 - [WebConfig] configure la couche [web/JSON],
 - [AppConfig] utilise les deux fichiers de configuration précédents pour configurer l'ensemble de l'application ;
 - le package [entities] contient :
 - les entités JPA,
 - les entités nécessaires à la couche [métier],
 - l'exception [PamException] ;
 - le package [helpers] contient une classe [Static] contenant une méthode utilitaire ;
- [3] : contient les éléments à compléter :
 - le package [metier] contient la couche [métier],
 - le package [repositories] contient les interfaces d'accès aux différentes tables de la base de données. On aurait pu appeler ce package [dao] mais on a préféré garder la terminologie de Spring. Ces interfaces implémentent la couche [DAO],
 - le package [restapi] implémente la couche [web/JSON] ;

4.2 Introduction à Spring Data

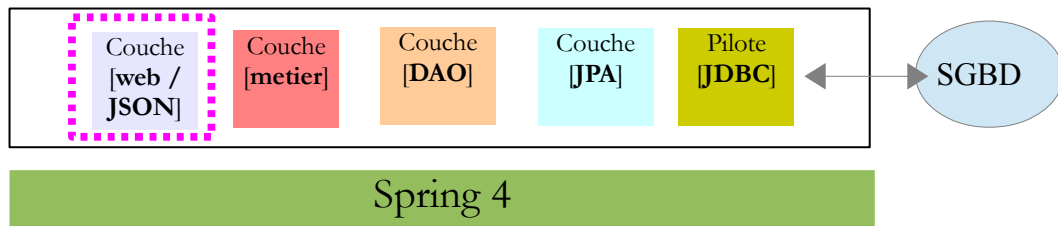
Nous allons implémenter la couche [DAO] du projet avec Spring Data, une branche de l'écosystème Spring.



On lira les paragraphes 2.2 à 2.2.4 (pages 20-29) du document [\[ref1\]](#). Utilisez STS (Spring Tool Suite) pour suivre le tutoriel.

4.3 Introduction à Spring MVC

Nous allons implémenter la couche [web] du projet avec Spring MVC, une branche de l'écosystème Spring.



On lira le paragraphe 2.11 (pages 60-73) du document [\[ref1\]](#). Utilisez STS (Spring Tool Suite) pour suivre le tutorial.

4.4 Le fichier [pom.xml]

Le projet [server-pam] est un projet Maven dont le fichier [pom.xml] des dépendances est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5.     <modelVersion>4.0.0</modelVersion>
6.     <groupId>istia.st.pam</groupId>
7.     <artifactId>server-pam</artifactId>
8.     <version>0.0.1-SNAPSHOT</version>
9.     <parent>
10.         <groupId>org.springframework.boot</groupId>
11.         <artifactId>spring-boot-starter-parent</artifactId>
12.         <version>1.1.1.RELEASE</version>
13.     </parent>
14.     <dependencies>
15.         <dependency>
16.             <groupId>org.springframework.boot</groupId>
17.             <artifactId>spring-boot-starter-web</artifactId>
18.         </dependency>
19.         <dependency>
20.             <groupId>org.springframework.boot</groupId>
21.             <artifactId>spring-boot-starter-data-jpa</artifactId>
22.         </dependency>
23.         <dependency>
24.             <groupId>org.springframework.boot</groupId>
25.             <artifactId>spring-boot-starter-test</artifactId>
26.             <scope>test</scope>
27.         </dependency>
28.         <dependency>
29.             <groupId>mysql</groupId>
30.             <artifactId>mysql-connector-java</artifactId>
31.         </dependency>
32.         <dependency>
33.             <groupId>commons-dbcp</groupId>
34.             <artifactId>commons-dbcp</artifactId>
35.         </dependency>
36.         <dependency>
37.             <groupId>commons-pool</groupId>
38.             <artifactId>commons-pool</artifactId>
39.         </dependency>
40.         <dependency>
41.             <groupId>com.fasterxml.jackson.core</groupId>
42.             <artifactId>jackson-databind</artifactId>
43.         </dependency>
44.         <dependency>
45.             <groupId>com.fasterxml.jackson.core</groupId>

```

```

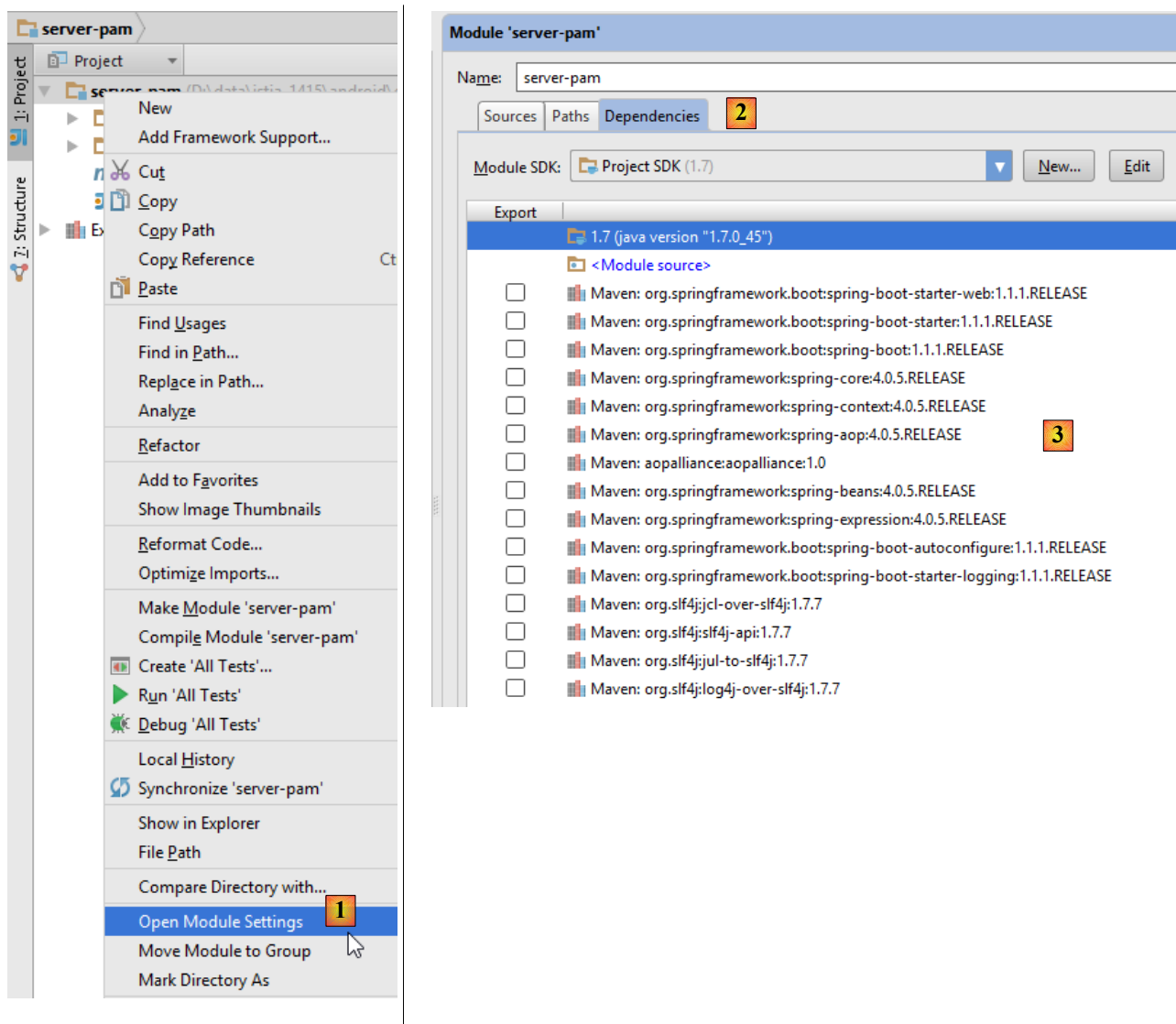
46.     <artifactId>jackson-annotations</artifactId>
47. </dependency>
48. <dependency>
49.     <groupId>com.google.guava</groupId>
50.     <artifactId>guava</artifactId>
51.     <version>16.0.1</version>
52. </dependency>
53. </dependencies>
54. <properties>
55.     <!-- use UTF-8 for everything -->
56.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
57.     <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
58.     <start-class>pam.boot.BootConsole</start-class>
59. </properties>
60. <build>
61.     <plugins>
62.         <plugin>
63.             <groupId>org.springframework.boot</groupId>
64.             <artifactId>spring-boot-maven-plugin</artifactId>
65.         </plugin>
66.     </plugins>
67. </build>
68. <repositories>
69.     <repository>
70.         <id>spring-milestones</id>
71.         <name>Spring Milestones</name>
72.         <url>http://repo.spring.io/libs-milestone</url>
73.         <snapshots>
74.             <enabled>false</enabled>
75.         </snapshots>
76.     </repository>
77. </repositories>
78. </project>

```

- lignes 9-13 : une dépendance sur le framework [Spring Boot], une branche de l'écosystème Spring. Ce framework permet une configuration minimale de Spring. Selon les archives présentes dans le Classpath du projet, [Spring Boot] infère une configuration plausible ou probable pour celui-ci. Ainsi si Hibernate est dans le Classpath du projet alors [Spring Boot] infèrera que l'implémentation JPA utilisée va être Hibernate et configurera Spring dans ce sens. Le développeur n'a plus à le faire. Il ne lui reste alors à faire que les configurations que [Spring Boot] n'a pas faites par défaut ou celles que [Spring Boot] a faites par défaut mais qui doivent être précisées. Dans tous les cas c'est la configuration faite par le développeur qui a le dernier mot ;
- lignes 9-13 : définissent un projet Maven parent. Ce dernier, [Spring Boot], définit les dépendances les plus courantes pour les divers types de projets Spring qui peuvent exister (cf page 41 de [ref1](#)). Lorsqu'on utilise l'une d'elles dans le fichier [pom.xml], sa version n'a pas à être précisée. C'est la version définie dans le projet parent qui sera utilisée ;
- lignes 15-18 : définissent une dépendance sur l'artefact [spring-boot-starter-web]. Cet artefact amène avec lui toutes les archives nécessaires à un projet Spring MVC. Parmi celles-ci on trouve l'archive d'un serveur Tomcat. C'est lui qui sera utilisé pour déployer l'application web. On notera que la version de la dépendance n'a pas été mentionnée. C'est celle mentionnée dans le projet parent qui sera utilisée ;
- lignes 19-22 : définissent une dépendance sur [spring-boot-starter-data-jpa]. Cet artefact amène avec lui les archives nécessaires à l'implémentation d'une couche [DAO] s'appuyant sur une interface JPA. L'implémentation JPA utilisée par défaut est Hibernate ;
- lignes 23-27 : définissent une dépendance sur [spring-boot-starter-test]. Cet artefact amène avec lui les archives nécessaires à l'intégration de tests unitaires JUnit avec Spring. On notera ligne 26, que le [scope] est *test*, ce qui signifie que la dépendance n'est nécessaire que pour les tests. Elle ne sera pas incluse dans l'archive produite par la compilation du projet ;
- lignes 28-31 : le pilote JDBC du SGBD MySQL ;
- lignes 32-39 : pool de connexions Commons DBCP. Un pool de connexions est un pool de connexions ouvertes. L'ouverture / fermeture d'une connexion à un SGBD a un coût (temps, mémoire) et le nombre de connexions ouvertes à un moment donné est limité. Le pool de connexions tente de résoudre ces deux problèmes : un certain nombre de connexions au SGBD sont faites dès l'instanciation du pool. Ensuite, de façon transparente, lorsqu'un code ouvre une connexion, une connexion du pool lui est donnée (pas de temps d'ouverture) et lorsque ce même code ferme la connexion, celle-ci est rendue au pool (elle n'est pas fermée). Ceci est fait de façon transparente pour le développeur. Le code n'a pas à être modifié ;

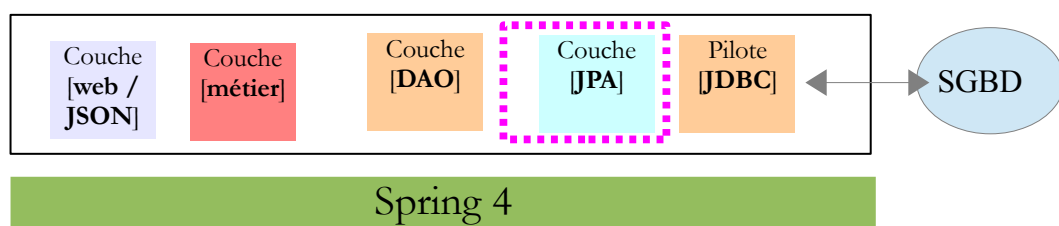
- lignes 40-47 : bibliothèque Jackson de gestion du JSON ;
- lignes 48-52 : bibliothèque Google de gestion des collections ;

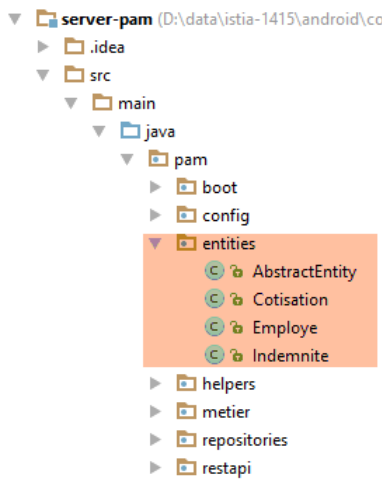
L'ensemble des dépendances ainsi amenée dans le projet peut être visualisé :



- en [1], les propriétés du projet ;
- en [2], on sélectionne l'onglet des dépendances ;
- en [3], la liste des dépendances (vue partielle). Elles sont très nombreuses. L'intérêt de [Spring Boot] est qu'on peut configurer un projet Spring sans avoir à les connaître toutes ;

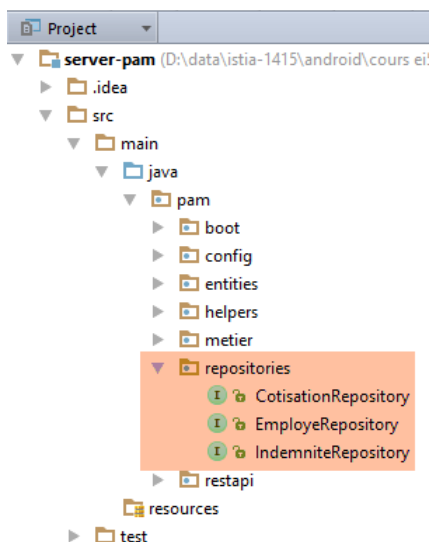
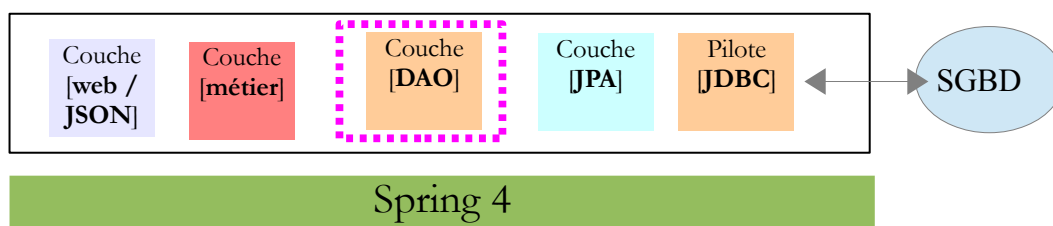
4.5 Les entités JPA





Travail : en suivant le paragraphe 2.5 page 42 de [ref1](#) ainsi que ce qui a été fait en TD, construire les entités JPA [Cotisation, Employe, Indemnité]. Le champ annoté par `[@ManyToOne]` de la classe [Employe] sera cherché en base en mode *Lazy*. Les classes JPA vous sont données sans les annotations JPA. Ce sont ces dernières que vous devez créer.

4.6 La couche [DAO]



Travail : en suivant le paragraphe 2.6 page 47 de [refl], construire les interfaces [CotisationRepository, EmployeRepository, IndemniteRepository] de la couche [DAO].

Notes :

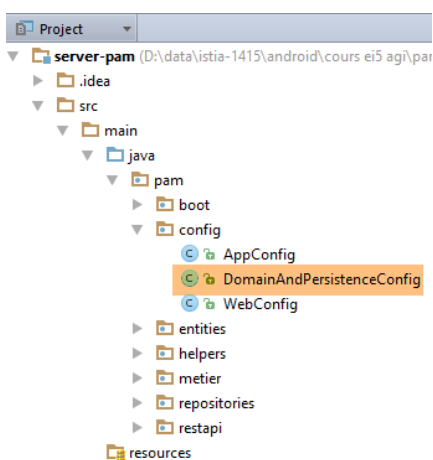
- toutes les interfaces étendront l'interface [CrudRepository] de [Spring Data]. L'interface [EmployeRepository] proposera en plus la méthode suivante :

```
// recherche d'un employé via son n° SS
public Iterable<Employe> findBySS(String SS);
```

L'employé renvoyé aura son champ [indemnité] rempli. Il faut se rappeler ici que ce champ a une annotation [ManyToOne] avec un mode de recherche *Lazy*. Donc par défaut, ce champ n'est pas initialisé lorsqu'on demande un employé à l'interface [EmployeRepository].

- il peut être utile de lire la classe de test [JUnitDao] qui teste les trois interfaces. Vous comprendrez mieux leur utilisation.

4.7 Configuration de la couche de persistance



La couche de persistance est configurée par la classe [DomainAndPersistenceConfig] suivante :

```
1. package pam.config;
2.
3. import org.apache.commons.dbcp.BasicDataSource;
4. import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5. import org.springframework.boot.orm.jpa.EntityScan;
6. import org.springframework.context.annotation.Bean;
7. import org.springframework.context.annotation.ComponentScan;
8. import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
9. import org.springframework.orm.jpa.JpaVendorAdapter;
10. import org.springframework.orm.jpa.vendor.Database;
11. import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
12. import org.springframework.transaction.annotation.EnableTransactionManagement;
13.
14. import javax.sql.DataSource;
15.
16. @EnableJpaRepositories(basePackages = { "pam.repositories" })
17. @EnableAutoConfiguration
18. @ComponentScan(basePackages = { "pam.metier" })
19. @EntityScan(basePackages = { "pam.entities" })
20. @EnableTransactionManagement
21. public class DomainAndPersistenceConfig {
22.
23.     // la source de données MySQL
```

```

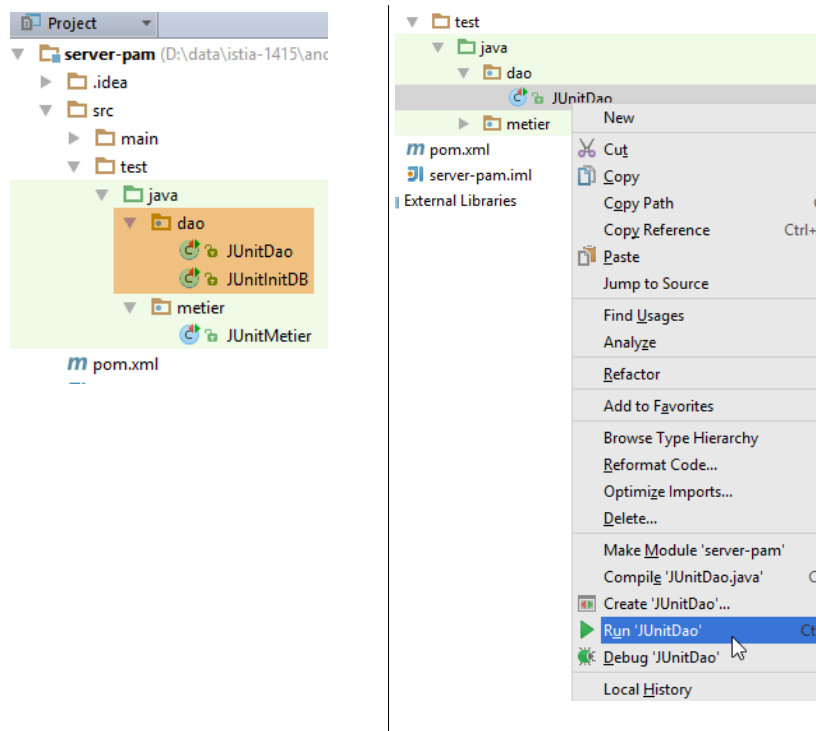
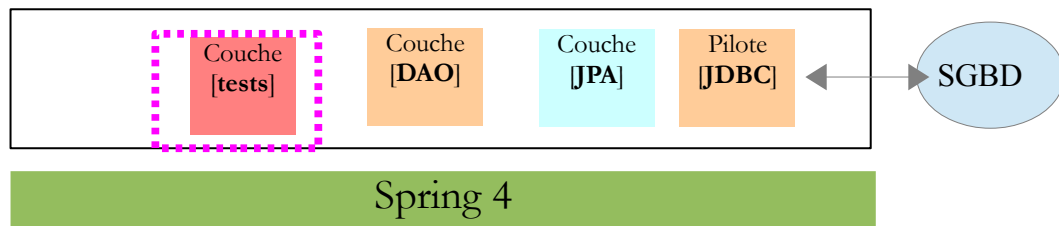
24. @Bean
25. public DataSource dataSource() {
26.     BasicDataSource dataSource = new BasicDataSource();
27.     dataSource.setDriverClassName("com.mysql.jdbc.Driver");
28.     dataSource.setUrl("jdbc:mysql://localhost:3306/dbpam_hibernate");
29.     dataSource.setUsername("root");
30.     dataSource.setPassword("");
31.     return dataSource;
32. }
33.
34. // le provider JPA - n'est pas nécessaire si on est satisfait des valeurs par
35. // défaut utilisées par Spring boot
36. // ici on le définit pour activer / désactiver les logs SQL
37. @Bean
38. public JpaVendorAdapter jpaVendorAdapter() {
39.     HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new HibernateJpaVendorAdapter();
40.     hibernateJpaVendorAdapter.setShowSql(false);
41.     hibernateJpaVendorAdapter.setGenerateDdl(false);
42.     hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
43.     return hibernateJpaVendorAdapter;
44. }
45.
46. // l'EntityManagerFactory et le TransactionManager sont définis avec des
47. // valeurs par défaut par Spring boot
48.
49. }

```

Depuis les versions 3.x, il est possible de configurer Spring avec des classes Java plutôt qu'avec des fichiers XML. C'est la voie qui a été suivie dans ce projet.

- ligne 17 : l'annotation `[@EnableAutoConfiguration]` a un double effet :
 - elle déclare que la classe annotée est une classe de configuration Spring,
 - elle autorise une auto-configuration faite par le framework [Spring Boot]. Ligne 4, on voit que l'annotation fait partie du framework [Spring Boot]. On rappelle que cette auto-configuration va être faite à partir des archives présentes dans le Classpath du projet, donc ici à partir des dépendances Maven que nous avons présentées ;
- ligne 16 : l'annotation `[EnableJpaRepositories]` est une annotation du framework [Spring Data] (ligne 8). Elle permet d'indiquer le ou les dossiers dans lesquels on trouvera les interfaces étendant l'interface `[CrudRepository]` de [Spring Data] ;
- ligne 19 : l'annotation `[EntityScan]` est une annotation du framework [Spring Boot] (ligne 5). Elle permet d'indiquer le ou les dossiers dans lesquels on trouvera les entités JPA ;
- ligne 20 : l'annotation `[EnableTransactionManagement]` est une annotation du framework [Spring] (ligne 12). Elle permet de demander à ce que les méthodes des interfaces `[CrudRepository]` de [Spring Data] soient exécutées dans des transactions ;
- ligne 18 : l'annotation `[ComponentScan]` est une annotation du framework [Spring] (ligne 7). Elle permet d'indiquer le ou les dossiers dans lesquels on trouvera des composants Spring (`@Component`, `@Service`, `@Controller`, ...). Ici l'annotation sera utilisée ultérieurement pour la couche [métier] ;
- ligne 24 : l'annotation `[Bean]` est une annotation du framework [Spring] (ligne 6). Elle désigne un objet géré par Spring. Cet objet est instancié une fois (singleton) lors de l'exploitation initiale de la classe de configuration. Par défaut, le bean porte le nom de la méthode annotée. On peut également lui donner explicitement un nom ;
- lignes 25-32 : la méthode `[dataSource]` définit le bean nommé 'dataSource'. Ce nom est prédéfini et ne doit pas être changé. Il sert à définir les quatre informations JDBC nécessaires pour accéder à la base de données ;
- on pourrait s'arrêter là. Les autres informations nécessaires (JPA provider, `EntityManagerFactory`, gestionnaire de transactions) vont être inférées par [Spring Boot] à partir du Classpath du projet (auto-configuration) ;
- lignes 38-44 : définissent le provider JPA, ici Hibernate. La méthode doit obligatoirement s'appeler `[jpaVendorAdapter]`. Par défaut, [Spring Boot] trouve Hibernate dans le Classpath du projet et configure un bean `[jpaVendorAdapter]` avec. Seulement, il active par défaut les logs SQL. Parce qu'on ne veut pas ces logs, on redéfinit le bean `[jpaVendorAdapter]`. On rappelle que c'est la configuration du développeur qui a le dernier mot ;
- ligne 40 : on inhibe les logs SQL ;
- ligne 41 : le provider JPA ne doit pas régénérer la base ;
- ligne 42 : le provider JPA est associé au SGBD MySQL. A noter que [Spring Boot] pouvait inférer cette information parce que le pilote JDBC de MySQL était dans le Classpath du projet ;
- lignes 46-47 : on garde l'auto-configuration de [Spring Boot] pour les autres éléments de configuration de la couche de persistance ;

4.8 Tests de la couche [DAO]



Le test [JUnitInitDB] est le suivant :

```

1. package dao;
2.
3. import org.junit.Before;
4. import org.junit.Test;
5. import org.junit.runner.RunWith;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.boot.test.SpringApplicationConfiguration;
8. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
9. import pam.config.DomainAndPersistenceConfig;
10. import pam.entities.Cotisation;
11. import pam.entities.Employe;
12. import pam.entities.Indemnité;
13. import pam.repositories.CotisationRepository;
14. import pam.repositories.EmployeRepository;
15. import pam.repositories.IndemnitéRepository;
16.
17. @SpringApplicationConfiguration(classes = DomainAndPersistenceConfig.class)
18. @RunWith(SpringJUnit4ClassRunner.class)
19. public class JUnitInitDB {
20.
21.     // repositories

```



```

22. @Autowired
23. private EmployeeRepository employeeRepository;
24. @Autowired
25. private IndemnityRepository indemnityRepository;
26. @Autowired
27. private CotisationRepository cotisationRepository;
28.
29. @Before()
30. public void clean() {
31.     log("clean-----");
32.     // on vide la base
33.     employeeRepository.deleteAll();
34.     cotisationRepository.deleteAll();
35.     indemnityRepository.deleteAll();
36. }
37.
38. // logs
39. private static void log(String message) {
40.     System.out.println("----- " + message);
41. }
42.
43. // tests
44. @Test
45. public void test01() {
46. ...
47. }
48. }

```

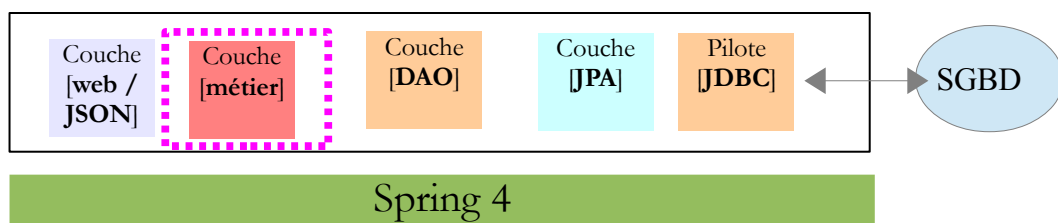
On a là un test JUnit intégré avec Spring. C'est l'annotation `@RunWith` (ligne 18), une annotation JUnit (ligne 5), qui assure cette intégration.

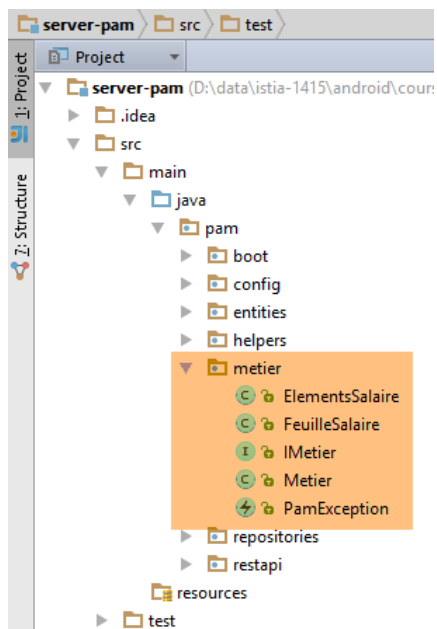
- ligne 17 : l'annotation `[SpringApplicationConfiguration]` permet d'indiquer la ou les classes de configuration à exploiter pour faire le test. Ici la classe `[DomainAndPersistenceConfig]` que nous avons détaillée va être exploitée. La couche `[DAO]` va alors être instanciée et reliée au SGBD. Les tests peuvent alors se dérouler ;
- ligne 22 : l'annotation `[Autowired]` sert à injecter un composant Spring après instanciation de la classe de test. Il y a différents types de composants Spring et les interfaces du dossier `[repositories]` en font partie (cf la classe `[DomainAndPersistenceConfig]`). On peut donc les injecter (lignes 22-27). Elles implémentent la couche `[DAO]` qui va alors pouvoir être testée ;

Travail : exécutez la classe `[JUnitInitDB]` qui régénère la base de données. Vérifiez que le test passe et que la base a bien été régénérée.

Travail : exécutez la classe `[JUnitDao]` composée de 11 tests. Vérifiez que tous les tests passent.

4.9 La couche [métier]





L'interface [IMetier] de la couche [métier] est la suivante :

```

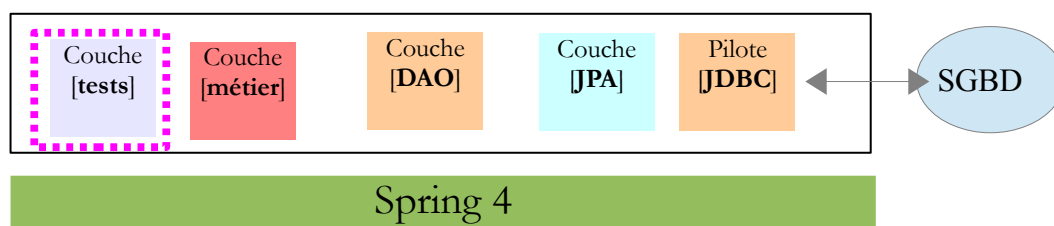
1. package pam.metier;
2.
3. import pam.entities.Employe;
4.
5. import java.util.List;
6.
7. public interface IMetier {
8.
9.     // liste des employés
10.    public List<Employe> getEmployes();
11.    // feuille de salaire
12.    public FeuilleSalaire getFeuilleSalaire(String ss, double ht, int jt);
13.
14. }

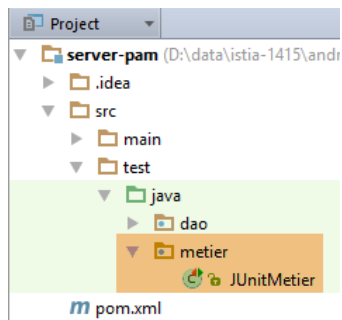
```

- ligne 10 : la méthode [getEmployes] renvoie la liste de tous les employés en base, sans remplir le champ [Indemnité indemnité] ;
- ligne 12 : la méthode [getFeuilleSalaire] renvoie la feuille de salaire de l'employé de n° SS [ss], ayant travaillé [ht] heures sur [jt] jours. Cette feuille de salaire inclut l'employé avec le champ [Indemnité indemnité] rempli. Si l'employé de n° SS [ss] n'existe pas, une exception de type [PamException] sera lancée ;

Travail : écrire la classe [Metier] qui implémente l'interface [IMetier].

4.10 Tests JUnit de la couche [métier]





Dans la classe [JUnitMetier], on trouve le code suivant :

```

1. package metier;
2.
3. import org.junit.Assert;
4. import org.junit.Test;
5. import org.junit.runner.RunWith;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.beans.factory.annotation.Qualifier;
8. import org.springframework.boot.test.SpringApplicationConfiguration;
9. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10. import pam.config.DomainAndPersistenceConfig;
11. import pam.metier.PamException;
12. import pam.metier.IMetier;
13.
14. @SpringApplicationConfiguration(classes = DomainAndPersistenceConfig.class)
15. @RunWith(SpringJUnit4ClassRunner.class)
16. public class JUnitMetier {
17.
18.     @Autowired
19.     @Qualifier("métier")
20.     private IMetier metier;
21.
22.     @Test
23.     public void test1() {
24. ...
25.     }
26. }

```

- lignes 18-20 : on injecte une référence sur la couche [métier]. Pour que cela fonctionne, il faut faire plusieurs choses :
 - il faut faire de la classe [Metier] un composant Spring :

```

@Service("métier")
public class Metier implements IMetier {

```

C'est l'annotation [@Service] qui fait de la classe [Metier] un composant géré par Spring. On aurait pu utiliser également l'annotation [Component]. On peut écrire [Service] ou [Service(« nom »)]. Dans le premier cas, le service n'est pas nommé, dans le second il l'est. Le nommage n'est nécessaire que lorsqu'il y a plusieurs composants Spring de même type, ici [IMetier] (Spring travaille avec les interfaces plutôt qu'avec les classes).

- il faut déclarer ce composant dans la classe de configuration [DomainAndPersistenceConfig] :

```

@ComponentScan(basePackages = {"pam.metier"})

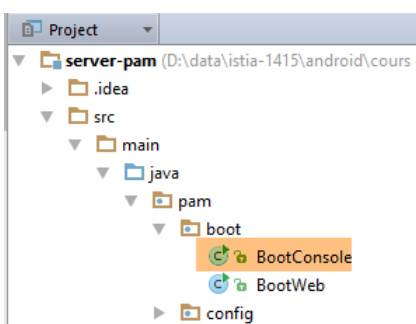
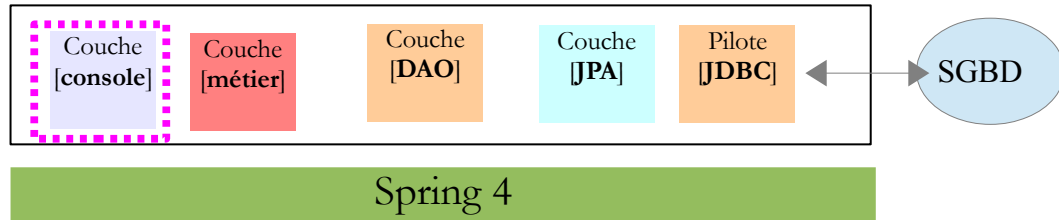
```

Ici, on dit à Spring de consulter le package [pam.metier] pour y trouver des composants. Il y trouvera la classe [Metier] avec son annotation [Service] ;

- ligne 19 : l'annotation `[@Qualifier]` sert à nommer le bean à injecter. N'est utile que si plusieurs beans ont le type `[IMetier]` de la ligne 20. C'est le cas dans ce projet ;

Travail : exécutez la classe `[UnitMetier]`. Vérifiez que le test passe. Régénérez auparavant la base avec le test `[UnitInitDB]`.

4.11 Tests console de la couche [métier]



La classe `[BootConsole]` est la suivante :

```

1. package pam.boot;
2.
3. import org.springframework.context.annotation.AnnotationConfigApplicationContext;
4. import pam.config.DomainAndPersistenceConfig;
5. import pam.entities.Employe;
6. import pam.metier.IMetier;
7.
8. import java.util.List;
9.
10. public class BootConsole {
11.     // le boot
12.     public static void main(String[] args) {
13.         // configuration Spring
14.         AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(DomainAndPersistenceConfig.class);
15.         // couche [métier]
16.         IMetier métier = context.getBean("métier", IMetier.class);
17.         // liste des employés
18.         List<Employe> employés = métier.getEmployes();
19.         display("Employés", employés);
20.         // salaire
21.         Employe employé = employés.get(0);
22.         String SS = employé.getSS();
23.         double ht = 150;
24.         int jt = 20;
25.         System.out.println(String.format("Salaire de %s %s (%s, %s, %s) : %s",
employé.getPrenom(), employé.getNom(), SS, ht, jt,

```

```

26.     métier.getFeuilleSalaire(SS, ht, jt));
27.     // fermeture du contexte Spring
28.     context.close();
29. }
30.
31. // méthode utilitaire - affiche les éléments d'une collection
32. private static <T> void display(String message, Iterable<T> elements) {
33.     System.out.println(message);
34.     for (T element : elements) {
35.         System.out.println(element);
36.     }
37. }
38.
39. }

```

- ligne 14 : exploitation du fichier de configuration [DomainAndPersistenceConfig] ;
- ligne 16 : récupération du bean appelé [métier] et de type [IMetier]. Il s'agit de la référence sur la couche [métier] ;
- ligne 18 : utilisation de la méthode [getEmployes] de l'interface [IMetier] ;
- ligne 26 : utilisation de la méthode [getSalaire] de l'interface [IMetier] ;

Travail : exécutez la classe [BootConsole].

Le résultat est noyé dans un flot de logs. On arrive néanmoins à trouver les lignes suivantes :

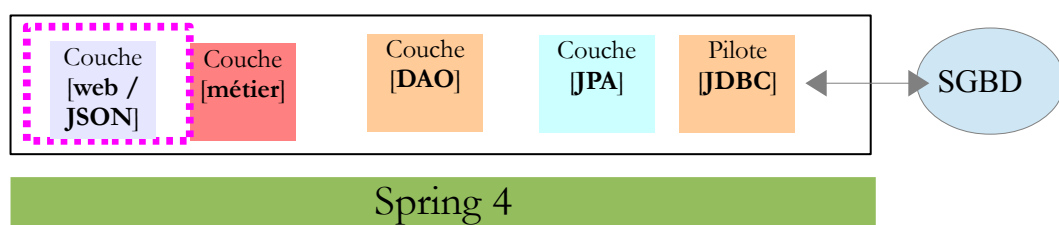
```

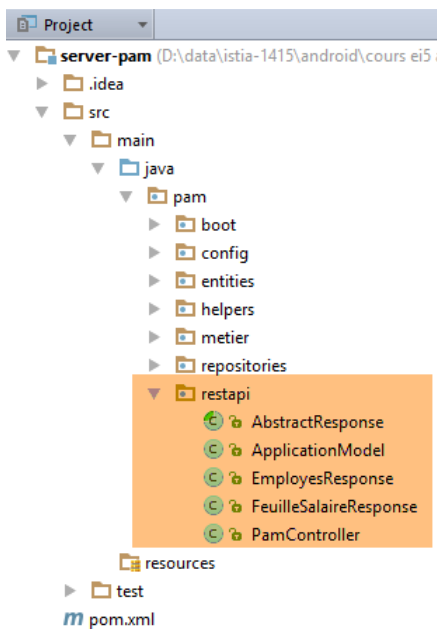
1. ...
2. Employés
3. jpa.Employe[id=27,version=0,SS=254104940426058,nom=Jouveinal,prenom=Marie,adresse=5 rue des
oiseaux,ville=St Corentin,code postal=49203,idIndemnité=36]
4. jpa.Employe[id=28,version=0,SS=260124402111742,nom=Laverti,prenom=Justine,adresse=La
braderie,ville=St Marcel,code postal=49014,idIndemnité=35]
5. ...
6. Salaire de Marie Jouveinal (254104940426058, 150.0, 20) :
[jpa.Employe[id=27,version=0,SS=254104940426058,nom=Jouveinal,prenom=Marie,adresse=5 rue
des oiseaux,ville=St Corentin,code
postal=49203,indemnité=jpa.Indemnité[id=36,version=0,indice=2,base heure=2.1,entretien
jour2.1,repas jour=3.1,indemnités
CP=15.0]],jpa.Cotisation[id=7,version=0,csgrds=3.49,csgd=6.15,secu=9.39,retraite=7.88],
[salaire base=362.25,cotisations sociales=97.48,indemnités d'entretien=42.0,indemnités de
repas=62.0,salaire net=368.77]]

```

- lignes 3-4 : on remarquera que les employés affichés n'ont pas de champ [Indemnité indemnité] mais ont le champ [long idIndemnité] qui est la clé étrangère de l'employé vers son indemnité ;
- ligne 6 : l'employé de la feuille de salaire a son champ [Indemnité indemnité] rempli ;

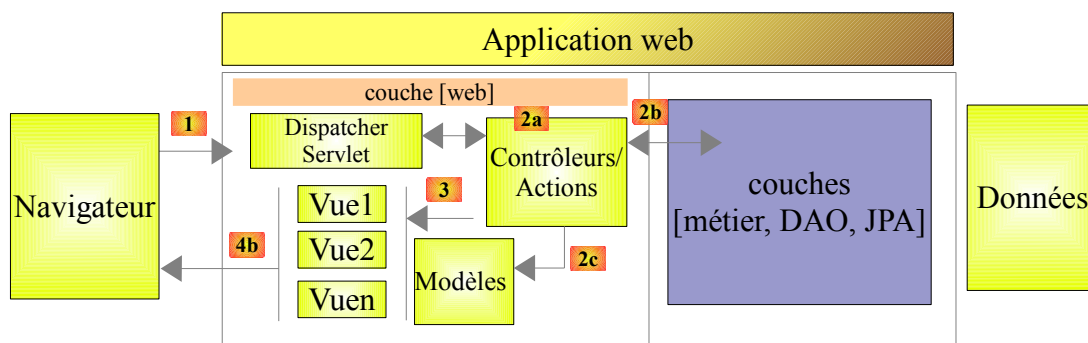
4.12 La couche [web/JSON]





4.12.1 L'architecture d'un service web/JSON implémenté par Spring MVC

Spring MVC implémente le modèle d'architecture dit MVC (Modèle – Vue – Contrôleur) de la façon suivante :

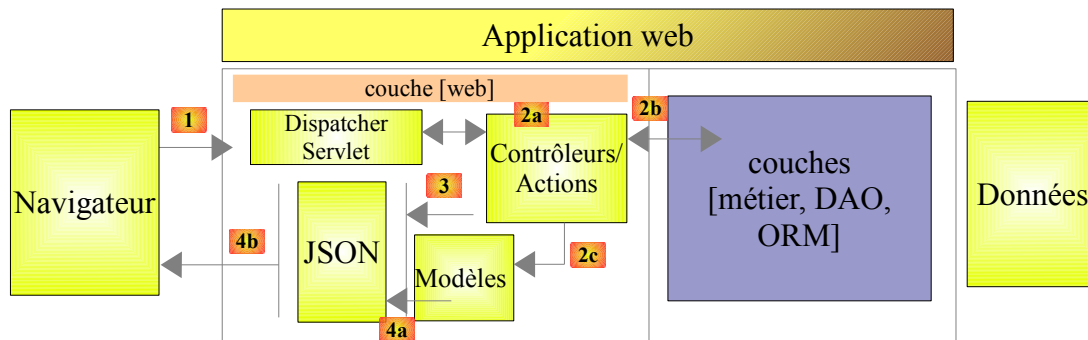


Le traitement d'une demande d'un client se déroule de la façon suivante :

1. **demande** - les URL demandées sont de la forme `http://machine:port/contexte/Action/param1/param2/....?p1=v1&p2=v2&...`. La [Dispatcher Servlet] est la classe de Spring qui traite les URL entrantes. Elle "route" l'URL vers l'action qui doit la traiter. Ces actions sont des méthodes de classes particulières appelées [Contrôleurs]. Le C de MVC est ici la chaîne [Dispatcher Servlet, Contrôleur, Action]. Si aucune action n'a été configurée pour traiter l'URL entrante, la servlet [Dispatcher Servlet] répondra que l'URL demandée n'a pas été trouvée (erreur 404 NOT FOUND) ;
2. **traitement**
 - l'action choisie peut exploiter les paramètres *parami* que la servlet [Dispatcher Servlet] lui a transmis. Ceux-ci peuvent provenir de plusieurs sources :
 - du chemin [/param1/param2/...] de l'URL,
 - des paramètres [p1=v1&p2=v2] de l'URL,
 - de paramètres postés par le navigateur avec sa demande ;
 - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche [métier] [2b]. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
 - une page d'erreur si la demande n'a pu être traitée correctement
 - une page de confirmation sinon
 - l'action demande à une certaine vue de s'afficher [3]. Cette vue va afficher des données qu'on appelle le **modèle de la vue**. C'est le M de MVC. L'action va créer ce modèle M [2c] et demander à une vue V de s'afficher [3] ;

3. **réponse** - la vue **V** choisie utilise le modèle **M** construit par l'action pour initialiser les parties dynamiques de la réponse HTML qu'elle doit envoyer au client puis envoie cette réponse.

Pour un service web / JSON, l'architecture précédente est légèrement modifiée :



- en [4a], le modèle qui est une classe Java est transformé en chaîne JSON par une bibliothèque JSON ;
- en [4b], cette chaîne JSON est envoyée au navigateur ;

4.12.2 Les URL du service web/JSON

Le service web expose les deux méthodes de la couche [métier]. Il accepte les deux URL suivantes :

- **/employes** : pour avoir la liste des employés ;
- **/salaire/SS/ht/jt** : pour avoir la feuille de salaire de l'employé de n° [SS] ayant travaillé [ht] heures pendant [jt] jours ;

Voici des copies d'écran montrant cela :

On demande les employés :

```

{
  "erreur": 0,
  "messages": null,
  "employés": [
    {
      "id": 27,
      "version": 0,
      "nom": "Jouveinal",
      "prenom": "Marie",
      "adresse": "5 rue des oiseaux",
      "ville": "St Corentin",
      "codePostal": "49203",
      "idIndemnité": 36,
      "ss": "254104940426058"
    },
    {
      "id": 28,
      "version": 0,
      "nom": "Laverti",
      "prenom": "Justine",
      "adresse": "La broderie",
      "ville": "St Marcel",
      "codePostal": "49014",
      "idIndemnité": 35,
      "ss": "260124402111742"
    }
  ]
}

```

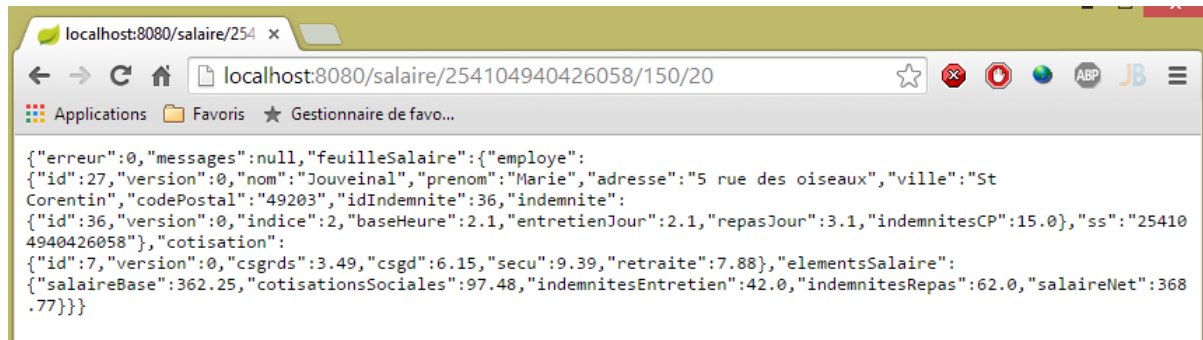
On coupe la base, on relance le serveur et on demande les employés :

```

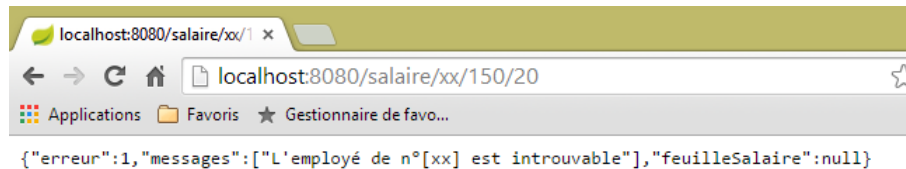
{
  "erreur": -1,
  "messages": [
    "Could not open JPA EntityManager for transaction; nested exception is javax.persistence.PersistenceException: org.hibernate.exception.GenericJDBCException: Could not open connection",
    "org.hibernate.exception.GenericJDBCException: Could not open connection",
    "Could not open connection",
    "Cannot create PoolableConnectionFactory (Communications link failure\n\nThe last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.)",
    "Communications link failure\n\nThe last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.",
    "Connection refused: connect"
  ],
  "employés": null
}

```

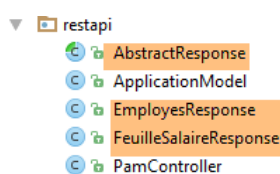
On demande un salaire :



On demande le salaire d'une personne inexistante :



4.12.3 Les réponses JSON du service web/JSON



Il y a deux types de réponse JSON selon l'URL demandée. Lorsque l'URL [/employees] est demandée, la réponse a trois champs :

1. `int erreur ;` // un code d'erreur - 0 si pas d'erreur
2. `List<String> messages ;` // une liste de messages d'erreur - null si pas d'erreur
3. `List<Employe> employes ;` // la liste des employés - null si erreur

Lorsque l'URL [/salaire] est demandée, la réponse a également trois champs :

1. `int erreur ;` // un code d'erreur - 0 si pas d'erreur
2. `List<String> messages ;` // une liste de messages d'erreur - null si pas d'erreur
3. `FeuilleSalaire feuilleSalaire ;` // la feuille de salaire - null si erreur

On factorise les champs [erreur, messages] dans la classe abstraite [AbstractResponse] (on déclare une classe *abstraite* lorsqu'elle ne peut être instanciée ou qu'on ne veut pas qu'elle soit instanciée. C'est cette dernière raison qui a prévalu ici) :

```

1. package pam.restapi;
2.
3. import java.util.List;
4.
5. public abstract class AbstractResponse {
6.
7.     // erreur
8.     private int erreur;
9.     // message d'erreur
10.    private List<String> messages;
11.
12.    // getters et setters

```



```
13. ...  
14. }
```

La classe [EmployeesResponse] est la réponse de l'URL [/employees] :

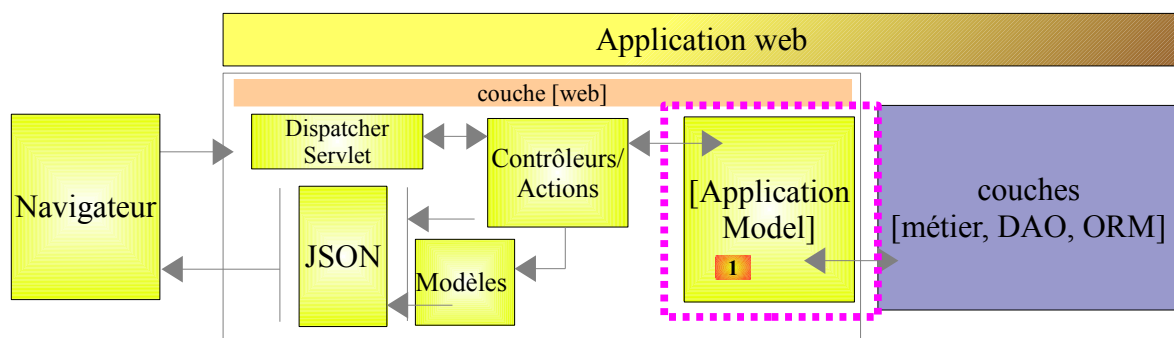
```
1. package pam.restapi;  
2.  
3. import pam.entities.Employe;  
4.  
5. import java.util.List;  
6.  
7. public class EmployeesResponse extends AbstractResponse {  
8.  
9.     // la liste des employés  
10.    private List<Employe> employés;  
11.  
12.    // getters et setters  
13. ...  
14. }
```

La classe [SalaireResponse] est la réponse de l'URL [/employees] :

```
1. package pam.restapi;  
2.  
3. import pam.metier.FeuilleSalaire;  
4.  
5. public class FeuilleSalaireResponse extends AbstractResponse {  
6.  
7.     // la feuille de salaire  
8.    private FeuilleSalaire feuilleSalaire;  
9.  
10.    // getters et setters  
11. ...  
12. }
```

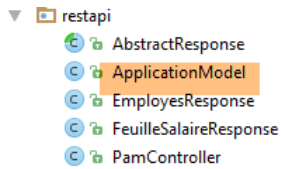
4.12.4 L'interface avec la couche [métier]

La couche [web/JSON] aura l'architecture suivante :



La classe [ApplicationModel] [1] aura deux fonctions :

- elle servira de cache à l'application web. Au démarrage de celle-ci, on mémorisera dans [ApplicationModel] la liste des employés. On suppose que celle-ci bouge rarement en base ;
- elle servira d'interface aux actions du contrôleur. Pour cela elle implémentera l'interface [IMetier] de la couche [métier] ;



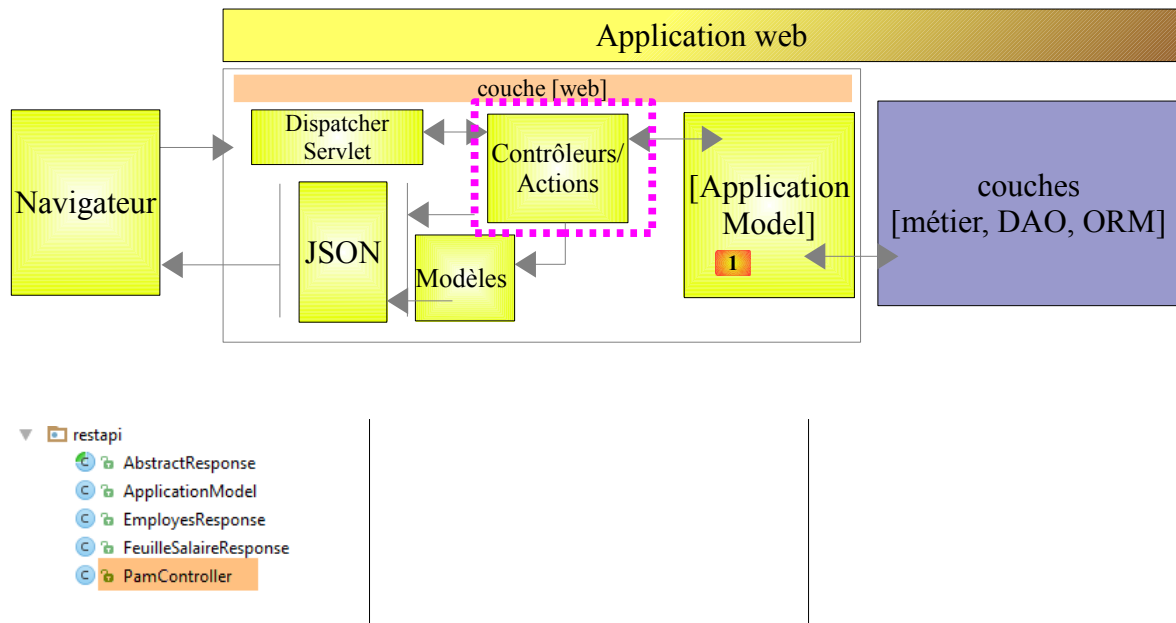
La classe [ApplicationModel] sera la suivante :

```
1. package pam.restapi;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.stereotype.Component;
5. import pam.entities.Employe;
6. import pam.metier.FeuilleSalaire;
7. import pam.helpers.Static;
8. import pam.metier.IMetier;
9.
10. import javax.annotation.PostConstruct;
11. import java.util.List;
12.
13. @Component
14. public class ApplicationModel implements IMetier {
15.
16.     // la couche [métier]
17.     @Autowired
18.     private IMetier métier;
19.
20.     // liste des employés
21.     private List<Employe> employés = null;
22.     // liste des messages d'erreur
23.     private List<String> messages = null;
24.
25.     @PostConstruct
26.     public void init() {
27.         // on récupère les employés
28.         try {
29.             employés = métier.getEmployes();
30.         } catch (Exception ex) {
31.             messages = Static.getErreursForException(ex);
32.         }
33.     }
34.
35.     // implémentation interface [IMetier]
36.     @Override
37.     public List<Employe> getEmployes() {
38.         return employés;
39.     }
40.
41.     @Override
42.     public FeuilleSalaire getFeuilleSalaire(String ss, double ht, int jt) {
43.         return métier.getFeuilleSalaire(ss, ht, jt);
44.     }
45.     // getters et setters
46.     ...
47.
48. }
```

- ligne 13 : la classe [ApplicationModel] sera un composant géré par Spring. Il sera instancié en un unique exemplaire (singleton) lors de l'exploitation de la classe de configuration de l'application web ;
- ligne 14 : la classe [ApplicationModel] implémente l'interface [IMetier] ;

- lignes 17-18 : une référence de la couche [métier] sera injectée ici. A noter que le champ annoté par [Autowired] ne nécessite pas la présence d'un *setter* ;
- ligne 25 : l'annotation [PostConstruct] désigne une méthode qui doit être exécutée après l'instanciation de la classe. Lorsqu'elle s'exécute, les injections Spring ont déjà eu lieu ;
- lignes 26-39 : la méthode [init] a pour but de stocker la liste des employés dans le champ de la ligne 21. Si elle n'y arrive pas, elle stocke la liste des messages d'erreur de l'exception dans le champ de la ligne 23. La classe [Static] utilisée ici vous est donnée ;
- lignes 36-44 : implémentent les méthodes de l'interface [IMetier] :
 - lignes 36-29 : la méthode [getEmployes] se contente de rendre la liste des employés stockée localement. Ainsi si cette liste change en base, l'utilisateur ne verra pas ce changement. On suppose ici que ce cas est rare ;
 - lignes 41-44 : la méthode [getEmployes] se contente d'appeler la méthode de même nom dans la couche [métier] ;

4.12.5 Le contrôleur



Le contrôleur [PamController] a la structure suivante :

```

1. package pam.restapi;
2.
3. import com.fasterxml.jackson.databind.ObjectMapper;
4. ...
5.
6. @RestController
7. public class PamController {
8.
9.     @Autowired
10.    private ApplicationModel application;
11.    @Autowired
12.    MappingJackson2HttpMessageConverter converter;
13.
14.    // liste de messages
15.    private List<String> messages;
16.    // mapper JSON
17.    private ObjectMapper mapper;
18.
19.    @PostConstruct
20.    public void init() {
21.        // messages d'erreur de l'application
22.        messages = application.getMessages();
23.        // mapper JSON

```

```

24.     mapper=converter.getMapper();
25. }
26.
27. ...
28. public EmployesResponse getEmployes() {
29. ...
30.     return response;
31. }
32.
33. ...
34. public FeuilleSalaireResponse getFeuilleSalaire(...) {
35. ...
36.     return response;
37. }
38. }

```

Travail : en suivant la démarche du paragraphe 2.12 de [refl](#), construisez le contrôleur [PamController].

Il y a une difficulté importante dans le code :

- la méthode [getEmployes] rend un objet de type [EmployesResponse] qui sera automatiquement sérialisé en JSON ;
- la méthode [getSalaire] rend un objet de type [SalaireResponse] qui sera automatiquement sérialisé en JSON ;

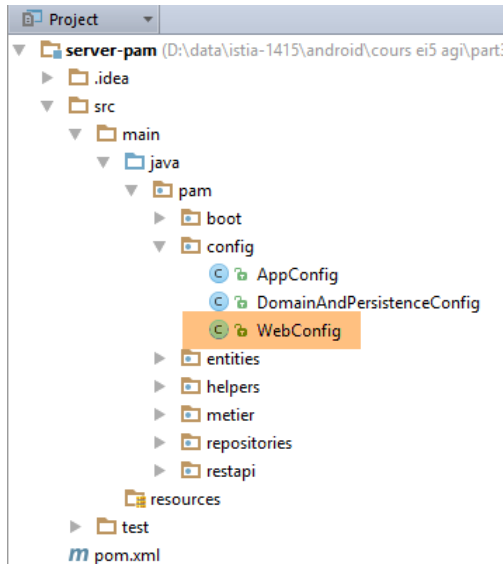
La sérialisation automatique sérialise tous les champs des objets [EmployesResponse] et [SalaireResponse].

Il y a un problème avec le type [Employe]. Il a un champ [Indemnité indemnité] annoté par [@ManyToOne(fetch = FetchType.LAZY)]. Ainsi la méthode [getEmployes] de la couche [métier] rend une liste d'employés avec le champ [Indemnité indemnité] non initialisé. Si on ne fait rien, la sérialisation par défaut de Spring va tenter de sérialiser le champ [Indemnité indemnité] en appelant la méthode [Employe.getIndemnité()]. Or cette méthode est redéfinie par le provider JPA pour aller chercher l'indemnité dans le contexte de persistance JPA. Or ici, nous sommes dans la couche web et le contexte de persistance n'existe pas. On obtient alors une exception. Il faut donc trouver un moyen de dire que le champ [Indemnité indemnité] du type [Employe] ne doit pas être sérialisé en JSON.

Par ailleurs, la méthode [getSalaire] rend une feuille de salaire qui contient un employé avec son indemnité. Donc celle-ci doit être sérialisée en JSON.

Résumons : on a un problème avec l'entité [Employe]. Son champ [Indemnité indemnité] doit parfois être sérialisé (méthode [getSalaire]) parfois pas (méthode [getEmployes]).

Il est possible d'indiquer au dernier moment quels champs on veut sérialiser mais pour cela on doit avoir accès au sérialiseur JSON, ce qui n'est pas le cas par défaut. Il nous faut donc configurer nous-mêmes la couche [web/JSON]. Ceci est fait avec la classe de configuration [WebConfig] :



La classe [WebConfig] est la suivante :

```

1. package pam.config;
2.
3. import com.fasterxml.jackson.databind.ObjectMapper;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.ComponentScan;
6. import org.springframework.context.annotation.Configuration;
7. import org.springframework.http.converter.HttpMessageConverter;
8. import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
9. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
10.
11. import java.util.List;
12.
13. @Configuration
14. @ComponentScan(basePackages = {"pam.restapi"})
15. public class WebConfig extends WebMvcConfigurerAdapter {
16.
17.     @Bean
18.     public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
19.         final MappingJackson2HttpMessageConverter converter = new
MappingJackson2HttpMessageConverter();
20.         final ObjectMapper objectMapper = new ObjectMapper();
21.         converter.setObjectMapper(objectMapper);
22.         return converter;
23.     }
24.
25.
26.     @Override
27.     public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
28.         converters.add(mappingJackson2HttpMessageConverter());
29.         super.configureMessageConverters(converters);
30.     }
31. }

```

- ligne 13: la classe [WebConfig] est une classe de configuration Spring. On avait rencontré l'annotation [EnableAutoConfiguration] qui faisait intervenir une auto-configuration faite par [Spring Boot]. Lorsqu'on n'a pas besoin de cette auto-configuration, on utilise simplement l'annotation [Configuration] ;
- ligne 14: on dit à Spring de chercher des composants dans le dossier [pam.restapi]. Il y trouvera le composant [PamController] annoté avec [@RestController]. Celui-ci sera donc géré par Spring, ce qui permettra de lui injecter des

références sur d'autres composants. Sera trouvé également le composant [ApplicationModel] annoté par [@Component]. Grâce à cette configuration, on pourra injecter le composant [ApplicationModel] dans le composant [PamController] ;

- ligne 15 : parce qu'on veut redéfinir le comportement par défaut de la couche [web/JSON], la classe [WebConfig] doit étendre la classe [WebMvcConfigurerAdapter]. On redéfinira la méthode [configureMessageConverters] (ligne 27) qui sert à définir les 'convertisseurs' des flux HTTP en entrée et sortie du service web ;
- ligne 17 : on définit un composant Spring ;
- lignes 18-23 : définissent un convertisseur JSON. C'est celui qui est utilisé par défaut lorsqu'on ne fait pas de configuration. Le fait de le définir explicitement comme un bean Spring va nous permettre de l'injecter dans le contrôleur et de paramétrer la sérialisation ou non des champs des objets qui vont être convertis en JSON ;
- lignes 26-30 : redéfinissent la méthode [configureMessageConverters] de la classe de base [WebMvcConfigurerAdapter]. Cette méthode sert à définir la liste des convertisseurs de l'application web. Ici, nous n'avons que le convertisseur JSON défini aux lignes 17-23 ;

Avec cette configuration, nous pouvons désormais injecter la référence du convertisseur dans le contrôleur. Revenons au code déjà présenté :

```
1. package pam.restapi;
2.
3. import com.fasterxml.jackson.databind.ObjectMapper;
4. ...
5.
6. @RestController
7. public class PamController {
8.
9.     @Autowired
10.    private ApplicationModel application;
11.    @Autowired
12.    MappingJackson2HttpMessageConverter converter;
13.
14.    // liste de messages
15.    private List<String> messages;
16.    // mapper JSON
17.    private ObjectMapper mapper;
18.
19.    @PostConstruct
20.    public void init() {
21.        // messages d'erreur de l'application
22.        messages = application.getMessages();
23.        // mapper JSON
24.        mapper=converter.getObjectMapper();
25.    }
26.
27.    ...
28.    public EmployeesResponse getEmployes() {
29.    ...
30.        return response;
31.    }
32.
33.    ...
34.    public FeuilleSalaireResponse getFeuilleSalaire(...) {
35.    ...
36.        return response;
37.    }
38. }
```

- le convertisseur JSON est injecté aux lignes 11-12 ;
- ligne 24 : la sérialisation est faite par l'objet de type [ObjectMapper] de la ligne 17 et initialisé ligne 24 à partir du convertisseur injecté ;

Le contrôle de la sérialisation de la classe [Employe] qui pose problème se fait en deux temps :

- il faut annoter la classe [Employe] avec des annotations de la bibliothèque JSON utilisée ici (il en existe d'autres) ;
- indiquer dans le code quels champs doivent être sérialisés ;

Le premier point se fait en ajoutant une annotation de filtre sur la classe [Employe] ;

```
1. import com.fasterxml.jackson.annotation.JsonFilter;
2. ...
3.
4. ...
5. @JsonFilter("employeeFilter")
6. public class Employe extends AbstractEntity implements Serializable {
```

- ligne 5 : on déclare que la sérialisation de la classe [Employe] est contrôlée par un filtre nommé [employeeFilter]. Ce filtre est défini dans le code Java et c'est grâce à lui qu'on définit les champs à sérialiser ;

Dans le code de la méthode [getEmployes], on écrira :

```
1. import com.fasterxml.jackson.databind.ser.impl.SimpleBeanPropertyFilter;
2. import com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;
3. ...
4. // sérialisation de la réponse
5. SimpleBeanPropertyFilter employeeFilter = SimpleBeanPropertyFilter.serializeAllExcept("indemnité");
6. mapper.setFilters(new SimpleFilterProvider().addFilter("employeeFilter", employeeFilter));
7. return response;
```

- ligne 5 : on définit un filtre qui sérialise tous les champs de l'objet auquel il sera appliqué sauf le champ appelé [indemnité] ;
- ligne 6 : ce filtre est ajouté aux filtres du convertisseur JSON et appelé [employeeFilter] ;

Ici la réponse est de type [EmployesResponse] :

```
1. package pam.restapi;
2.
3. ...
4.
5. public class EmployesResponse extends AbstractResponse {
6.
7. // la liste des employés
8. private List<Employe> employés;
9.
10. // getters et setters
11. ...
12. }
```

Le champ [List<Employe> employés] va être sérialisé donc chaque employé va être sérialisé. Comme la classe [Employe] utilise le filtre JSON [employeeFilter] et que ce filtre ne sérialise pas le champ [Indemnité indemnité], on aura donc une liste d'employés sans leurs indemnités.

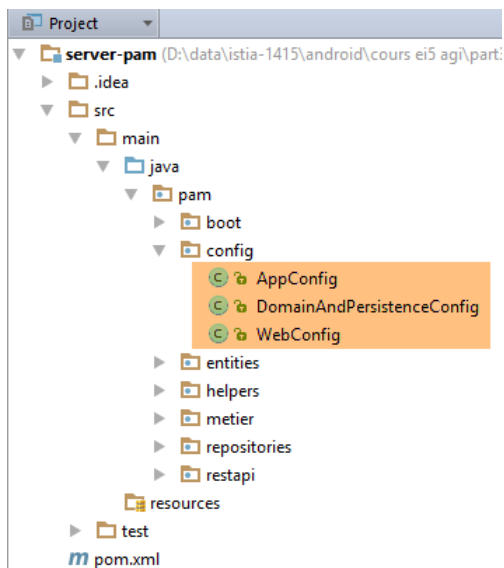
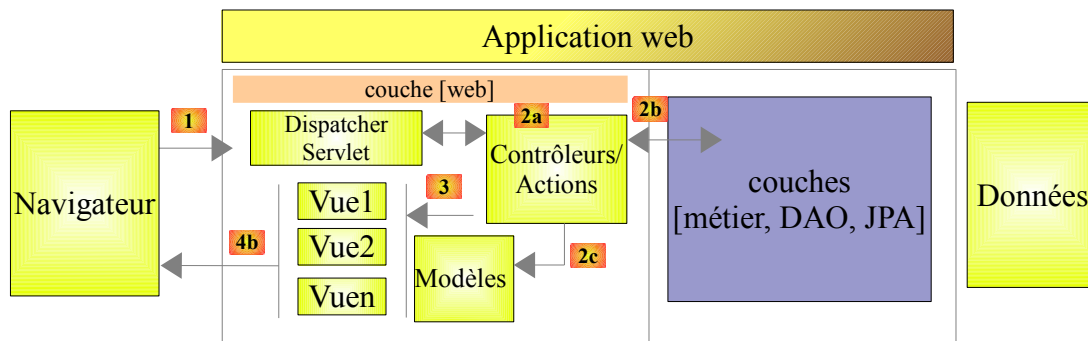
Dans le code de la méthode [getSalaire], on a un code similaire :

```
8. import com.fasterxml.jackson.databind.ser.impl.SimpleBeanPropertyFilter;
9. import com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;
10. ...
11. // sérialisation de la réponse
12. SimpleBeanPropertyFilter employeeFilter = SimpleBeanPropertyFilter.serializeAllExcept();
13. mapper.setFilters(new SimpleFilterProvider().addFilter("employeeFilter", employeeFilter));
14. return response;
```

- ligne 12 : on crée un filtre qui n'exclut aucun champ de la sérialisation ;

Ici la réponse est de type [SalaireResponse] qui contient un champ de type [Employe]. Ce champ sera donc sérialisé en JSON avec le champ [Indemnité indemnité].

4.13 Configuration de l'application web



L'application web dans sa globalité est configurée avec trois fichiers :

- le fichier [DomainAndPersistenceConfig] qui configure les couches [métier, DAO, JPA] ;
- le fichier [WebConfig] qui configure la couche [web/JSON] ;
- le fichier [AppConfig] qui configure la totalité de l'application ;

Le code de la classe [AppConfig] est le suivant :

```
1. package pam.config;
2.
3. import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4. import org.springframework.context.annotation.Import;
5.
6. @EnableAutoConfiguration
7. @Import({ DomainAndPersistenceConfig.class, WebConfig.class})
8. public class AppConfig {
9.
10. }
```

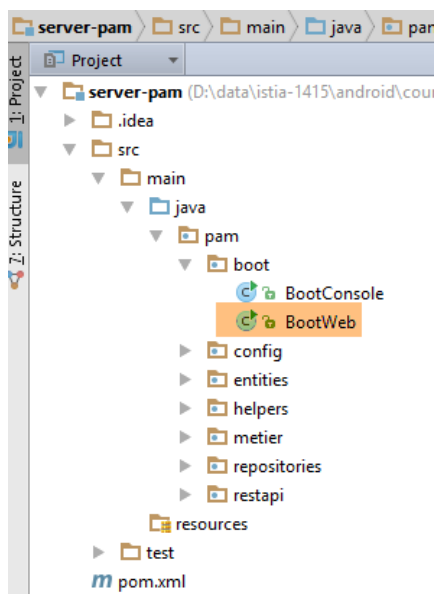
- ligne 6 : l'annotation [EnableAutoConfiguration] fait de la classe [AppConfig] une classe de configuration Spring. Par ailleurs, elle autorise une auto-configuration faite par [Spring Boot] à partir des dépendances du projet. Parce que dans celles-ci, on trouve les archives du serveur Tomcat ainsi que celles de la bibliothèque JSON [Jackson] :
 - l'application web sera déployée sur le serveur Tomcat ;

- la sérialisation des objets produits par les actions du contrôleur [PamController] sera faite par la bibliothèque JSON [Jackson] ;

De même parce que dans les archives du projet, on trouve celles de [Spring MVC], c'est ce framework qui sera utilisé pour traiter les requêtes des clients. Ces différentes informations rendent inutiles la création du fichier [web.xml] qui normalement configure toute application web JEE.

- ligne 7 : cette ligne permet d'indiquer les autres fichiers de configuration à exploiter :
 - [DomainAndPersistenceConfig] qui configure les couches [métier, DAO, JPA],
 - [WebConfig] qui configure la couche [web/JSON] ;

4.14 La classe exécutable de l'application web



La classe [BootWeb] est la classe exécutable qui lance le service web. Son code est le suivant :

```
1. package pam.boot;
2.
3. import org.springframework.boot.SpringApplication;
4. import pam.config.AppConfig;
5.
6. public class BootWeb {
7.     public static void main(String[] args) {
8.         SpringApplication.run(AppConfig.class, args);
9.     }
10.
11. }
```

- ligne 8 : la méthode [SpringApplication.run] est une méthode statique de [Spring Boot] (ligne 3). Elle permet de lancer une application Spring en lui donnant comme paramètres :
 - la classe de configuration de l'ensemble du projet ;
 - les éventuels arguments passés à la méthode [main] ;

Parce qu'ici le fichier [AppConfig] configure à la fois une couche de persistance et une couche web, le serveur Tomcat embarqué dans les dépendances va être lancé et l'application web déployée dessus. Les logs console sont alors les suivants :

```
1. ...
2. 2014-09-18 11:57:00.260 INFO 10560 --- [          main]
   o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations
   {4.0.4.Final}
3. 2014-09-18 11:57:00.360 INFO 10560 --- [          main] org.hibernate.dialect.Dialect
   : HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
```

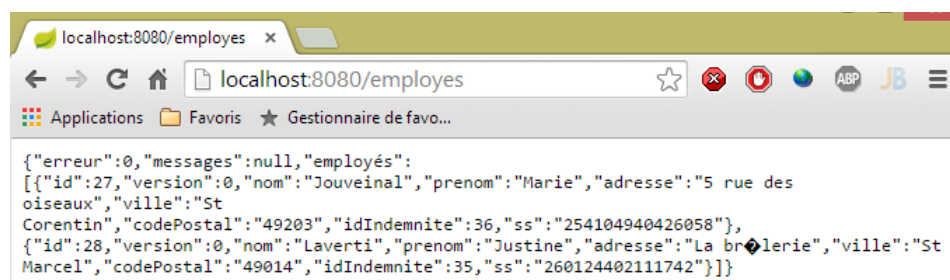
```

4. 2014-09-18 11:57:00.627 INFO 10560 --- [          main]
   o.h.h.i.ast.ASTQueryTranslatorFactory : HHH000397: Using ASTQueryTranslatorFactory
5. 2014-09-18 11:57:01.907 INFO 10560 --- [          main]
   o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler
   of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
6. 2014-09-18 11:57:02.028 INFO 10560 --- [          main]
   s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
   "{[/employees],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
   public pam.restapi.EmployeesResponse pam.restapi.PamController.getEmployees()
7. 2014-09-18 11:57:02.029 INFO 10560 --- [          main]
   s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/salaire/{SS}/{ht}/{jt}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
   pam.restapi.FeuilleSalaireResponse
   pam.restapi.PamController.getFeuilleSalaire(java.lang.String,double,int)
8. 2014-09-18 11:57:02.033 INFO 10560 --- [          main]
   s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
   "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
   org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter.handle(org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>)
   org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
9. 2014-09-18 11:57:02.033 INFO 10560 --- [          main]
   s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
   "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[text/html],custom=[]}"
   onto public org.springframework.web.servlet.ModelAndView
   org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest)
10. 2014-09-18 11:57:02.061 INFO 10560 --- [          main]
   o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type
   [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
11. 2014-09-18 11:57:02.061 INFO 10560 --- [          main]
   o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of
   type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
12. 2014-09-18 11:57:02.429 INFO 10560 --- [          main] o.s.j.e.a.AnnotationMBeanExporter
   : Registering beans for JMX exposure on startup
13. 2014-09-18 11:57:02.562 INFO 10560 --- [          main]
   s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080/http
14. 2014-09-18 11:57:02.565 INFO 10560 --- [          main] pam.boot.BootWeb
   : Started BootWeb in 8.801 seconds (JVM running for 9.397)

```

- lignes 1-4 : les logs de la création de la couche de persistance ;
- lignes 5-14 : les logs du déploiement de la couche web ;
- ligne 6 : l'URL [/employees] du service web est découverte ;
- ligne 7 : l'URL [/salaire/{SS}/{ht}/{jt}] du service web est découverte ;
- ligne 13 : le serveur Tomcat est lancé et attend des requêtes sur le port 8080 ;

On prend alors un navigateur et on demande l'URL [http://localhost:8080/employees] pour avoir la liste des employés sous forme JSON :



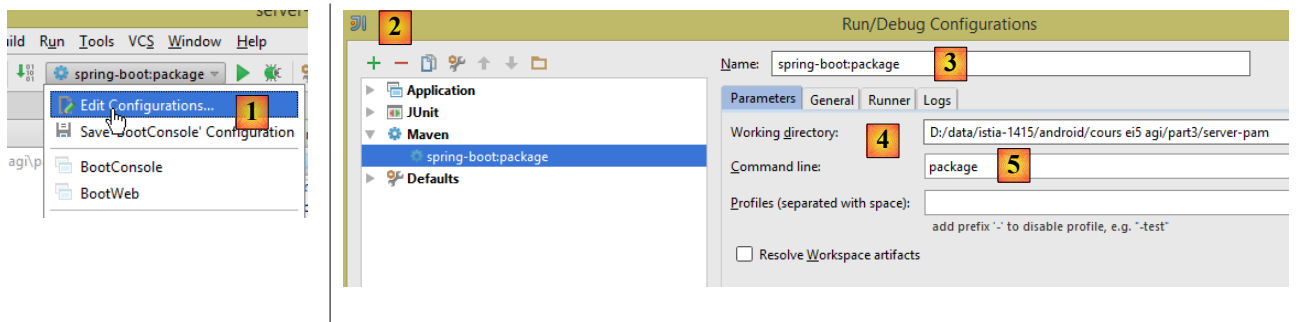
```

{"erreur":0,"messages":null,"employés":
[{"id":27,"version":0,"nom":"Jouveinal","prenom":"Marie","adresse":"5 rue des
oiseaux","ville":"St
Corentin","codePostal":"49203","idIndemnite":36,"ss":"254104940426058"},
{"id":28,"version":0,"nom":"Laverti","prenom":"Justine","adresse":"La broderie","ville":"St
Marcel","codePostal":"49014","idIndemnite":35,"ss":"260124402111742"}]}

```

4.15 Création d'une archive exécutable

Nous montrons ici comment créer une archive exécutable du projet permettant d'exécuter l'application web en-dehors d'un IDE. Nous commençons par créer une nouvelle configuration d'exécution :



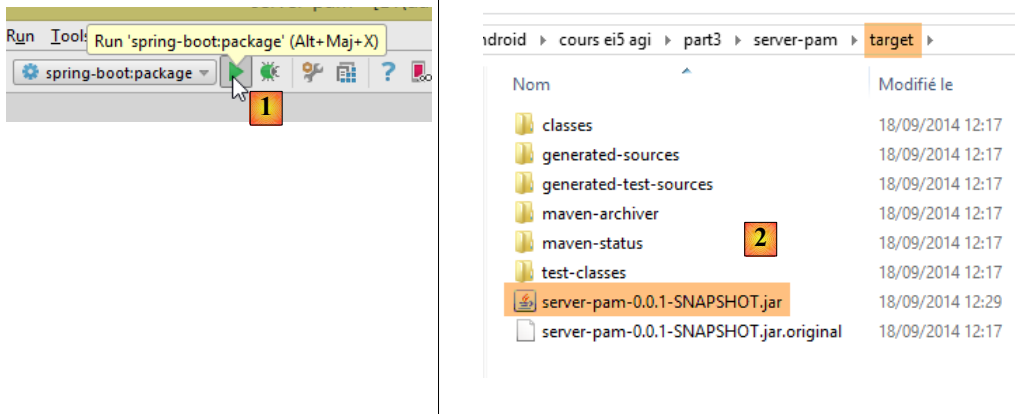
- en [1], on crée une nouvelle configuration d'exécution ;
- en [2], on crée une configuration Maven (sélectionner Maven et faire +) ;
- en [3], on donne un nom à la configuration ;
- en [4], on se positionne sur le répertoire du projet (c'est lui qui contient le fichier pom.xml qui va être exécuté) ;
- en [5], la commande Maven à exécuter est la commande [package] ;

La partie du fichier [pom.xml] exploitée est la suivante :

```
1. <properties>
2.     ....
3.     <start-class>pam.boot.BootWeb</start-class>
4. </properties>
5. <build>
6.     <plugins>
7.         <plugin>
8.             <groupId>org.springframework.boot</groupId>
9.             <artifactId>spring-boot-maven-plugin</artifactId>
10.        </plugin>
11.    </plugins>
12. </build>
13. <repositories>
14.     <repository>
15.         <id>spring-milestones</id>
16.         <name>Spring Milestones</name>
17.         <url>http://repo.spring.io/libs-milestone</url>
18.         <snapshots>
19.             <enabled>false</enabled>
20.         </snapshots>
21.     </repository>
22. </repositories>
```

C'est le plugin [spring-boot-maven-plugin] qui va produire l'archive exécutable. Pour qu'une archive jar soit exécutable, il faut indiquer quelle est la classe à exécuter dans l'archive. Cette information est donnée par la ligne 3.

Ceci fait, on exécute [1] cette configuration d'exécution (il faut auparavant arrêter le serveur web s'il est en cours d'exécution) :



Si l'exécution se passe bien une archive a été créée dans le dossier [target] du projet [2]. On l'exécute dans une fenêtre DOS :

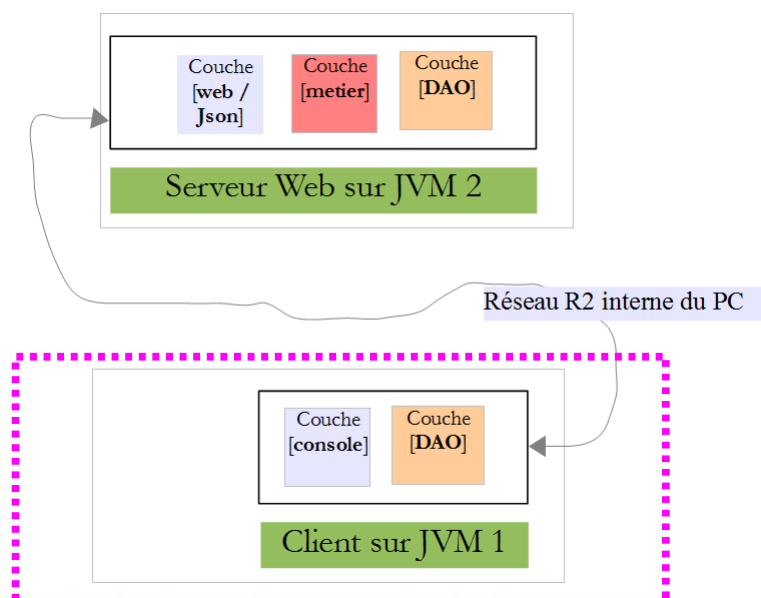
```
...\server-pam\target>java -jar server-pam-0.0.1-SNAPSHOT.jar
```

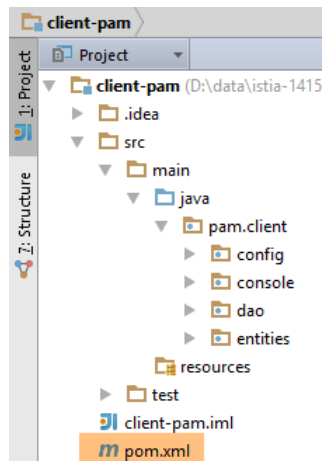
Il faut être positionné sur le dossier [target]. Le service web est alors lancé :

```
1. ....
2. 2014-09-18 12:39:06.050 INFO 8552 --- [          main]
   o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type
   [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
3. 2014-09-18 12:39:06.050 INFO 8552 --- [          main]
   o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of
   type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
4. 2014-09-18 12:39:06.568 INFO 8552 --- [          main] o.s.j.e.a.AnnotationMBeanExporter
   : Registering beans for JMX exposure on startup
5. 2014-09-18 12:39:06.712 INFO 8552 --- [          main]
   s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080/http
6. 2014-09-18 12:39:06.715 INFO 8552 --- [          main] pam.boot.BootWeb : Started
   BootWeb in 11.163 seconds (JVM running for 11.981)
```

On peut alors demander avec un navigateur les URL du service web.

5 Le client





5.1 Dépendances Maven

Le projet est un projet Maven avec le fichier [pom.xml] suivant :

```

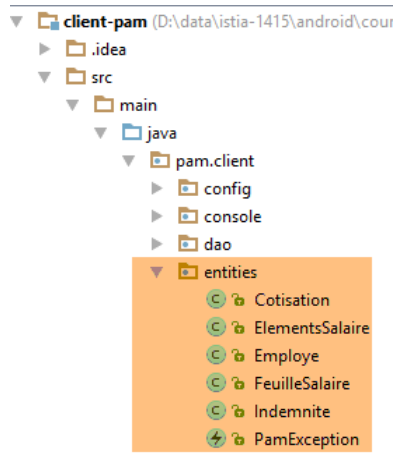
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.     <modelVersion>4.0.0</modelVersion>
6.
7.     <groupId>istia.st.pam</groupId>
8.     <artifactId>client-pam</artifactId>
9.     <version>1.0-SNAPSHOT</version>
10.
11.     <properties>
12.         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13.     </properties>
14.     <parent>
15.         <groupId>org.springframework.boot</groupId>
16.         <artifactId>spring-boot-starter-parent</artifactId>
17.         <version>1.1.1.RELEASE</version>
18.     </parent>
19.     <dependencies>
20.         <dependency>
21.             <groupId>org.springframework.boot</groupId>
22.             <artifactId>spring-boot-starter-web</artifactId>
23.         </dependency>
24.         <dependency>
25.             <groupId>com.fasterxml.jackson.core</groupId>
26.             <artifactId>jackson-databind</artifactId>
27.             <version>2.3.3</version>
28.         </dependency>
29.         <dependency>
30.             <groupId>com.fasterxml.jackson.core</groupId>
31.             <artifactId>jackson-annotations</artifactId>
32.             <version>2.3.3</version>
33.         </dependency>
34.         <dependency>
35.             <groupId>org.springframework.boot</groupId>
36.             <artifactId>spring-boot-starter-test</artifactId>
37.             <scope>test</scope>
38.         </dependency>

```

```
39. </dependencies>
40.
41. </project>
```

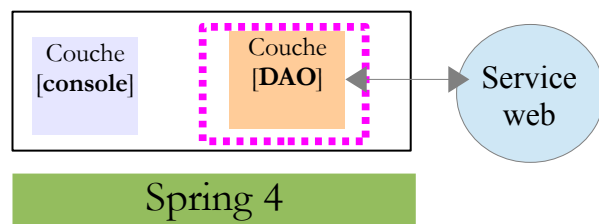
- lignes 20-23 : une dépendance sur le framework [Spring MVC]. Ce framework fournit la classe [RestTemplate] qui permet de dialoguer avec un serveur web/JSON. La version utilisée est celle définie par le projet Maven parent des lignes 14-18 ;
- lignes 24-33 : les dépendances sur la bibliothèque JSON [Jackson]. Elle est utilisée ici pour désérialiser les chaînes JSON envoyées par le service web. La version utilisée est celle définie par le projet Maven parent des lignes 14-18 ;
- lignes 34-38 : les dépendances sur les bibliothèques d'intégration Spring / JUnit ;

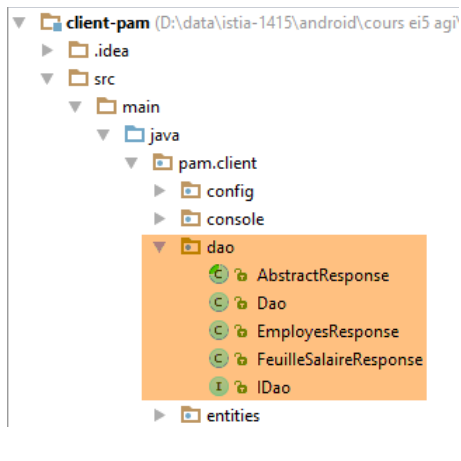
5.2 Les entités manipulées par le client



Les éléments du dossier [entities] sont identiques à leurs versions de mêmes nom côté serveur. Cependant, pour les entités JPA [Cotisation, Employe, Indemnite] on n'a pas conservé les annotations JPA, inutiles côté client.

5.3 La couche [DAO]





Les classes [AbstractResponse, EmployesResponse, SalaireResponse] sont les réponses du service web déjà utilisées côté serveur.

La couche [DAO] présente l'interface [IDao] suivante :

```

1. package pam.client.dao;
2.
3. import pam.client.entities.Employe;
4. import pam.client.entities.FeuilleSalaire;
5.
6. import java.util.List;
7.
8. public interface IDao {
9.     // liste des employés
10.    public List<Employe> getEmployes();
11.
12.    // feuille de salaire
13.    public FeuilleSalaire getFeuilleSalaire(String ss, double ht, int jt);
14.
15.    // timeout
16.    public void setTimeout(int millis);
17.
18.    // URL service REST
19.    public void setUrlServiceRest(String urlServiceRest);
20. }

```

- les méthodes des lignes 10 et 13 reprennent les mêmes noms et signatures que leurs équivalentes dans la couche [métier] du serveur. C'est normal. Tout se passe comme si la couche [console] du client dialoguait de façon transparente avec la couche [métier] du service web ;
- ligne 16 : sert à fixer un délai maximal d'attente d'une réponse du service web. Un délai par défaut existe mais il est de plusieurs dizaines de secondes. En général, on ne veut pas attendre aussi longtemps ;
- ligne 19 : sert à fixer l'URL racine du service web. Dans nos exemples précédents, cette URL serait [http://localhost:8080] ;

L'interface [IDao] est implémentée par la classe [Dao] suivante :

```

1. package pam.client.dao;
2.
3. import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
4. import org.springframework.stereotype.Service;
5. import org.springframework.web.client.RestTemplate;
6. import pam.client.entities.Employe;
7. import pam.client.entities.FeuilleSalaire;
8. import pam.client.entities.PamException;
9.
10. import java.io.IOException;
11. import java.net.InetSocketAddress;

```

```

12. import java.net.Socket;
13. import java.net.URI;
14. import java.net.URISyntaxException;
15. import java.util.List;
16. import java.util.logging.Level;
17. import java.util.logging.Logger;
18.
19. @Service
20. public class Dao implements IDao {
21.
22.     // client REST
23.     private final RestTemplate restTemplate;
24.     // délai d'attente maximal
25.     private int timeout;
26.     // URL du service REST
27.     private String urlServiceRest;
28.
29.     // constructeur
30.     public Dao() {
31.         // on crée un objet [RestTemplate]
32.         restTemplate = new RestTemplate();
33.         // on le configure - il doit être capable de gérer la chaîne JSON qu'il va recevoir
34.         restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());
35.     }
36.
37.     private void checkResponsiveness(String urlService) {
38.         // on crée l'URI du service distant
39.         String url = urlService.replace("{", "").replace("}", "");
40.         // on vérifie que [url] ressemble à une URL
41.         URI service = null;
42.         boolean erreur = false;
43.         try {
44.             service = new URI(url);
45.             erreur = service.getHost() == null && service.getPort() == -1;
46.         } catch (URISyntaxException ex) {
47.             erreur = true;
48.         }
49.         // erreur ?
50.         if (erreur) {
51.             throw new PamException(String.format("Format d'URL incorrect [%s]", url));
52.         }
53.         // on se connecte au service
54.         Socket client = null;
55.         try {
56.             // on se connecte au service avec attente maximale définie par configuration
57.             client = new Socket();
58.             client.connect(new InetSocketAddress(service.getHost(), service.getPort()), timeout);
59.         } catch (Exception e) {
60.             throw new PamException("Le service distant n'a pas répondu assez vite", e);
61.         } finally {
62.             // on libère les ressources
63.             if (client != null) {
64.                 try {
65.                     client.close();
66.                 } catch (IOException ex) {
67.                     Logger.getLogger(Dao.class.getName()).log(Level.SEVERE, null, ex);
68.                 }
69.             }
70.         }
71.     }
72.
73.
74.     @Override

```



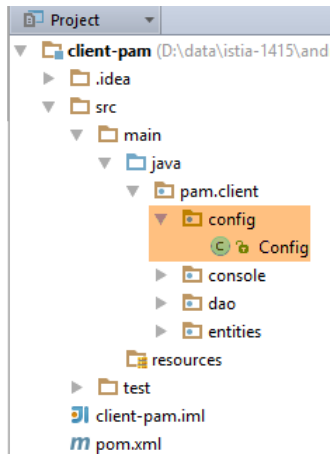
```

75. public List<Employe> getEmployes() {
76.     // on vérifie que le serveur distant répond assez vite
77.     // une exception est lancée sinon
78.     checkResponsiveness(urlServiceRest);
79.     // on interroge le service
80.     String urlService = String.format("%s/%s", urlServiceRest, "employes");
81.     EmployesResponse response = restTemplate.getForObject(urlService,
EmployesResponse.class);
82.     // on analyse la réponse
83.     int erreur = response.getErreur();
84.     if (erreur == 0) {
85.         return response.getEmployés();
86.     } else {
87.         throw new PamException(getMessageFromMessages(response.getMessages()));
88.     }
89. }
90.
91. private String getMessageFromMessages(List<String> messages) {
92.     StringBuffer buffer = new StringBuffer();
93.     for (String message : messages) {
94.         buffer.append(message + "\n");
95.     }
96.     return buffer.toString();
97. }
98.
99. @Override
100. public FeuilleSalaire getFeuilleSalaire(String SS, double ht, int jt) {
101.     ...
102. }
103.
104. // délai d'attente maximal
105. public void setTimeout(int millis) {
106.     this.timeout = millis;
107. }
108.
109. @Override
110. public void setUrlServiceRest(String urlServiceRest) {
111.     this.urlServiceRest = urlServiceRest;
112. }
113.
114. }

```

Travail : comprenez ce code et écrivez la méthode [getFeuilleSalaire].

5.4 Configuration de la couche [DAO]



La classe [Config] est la suivante :

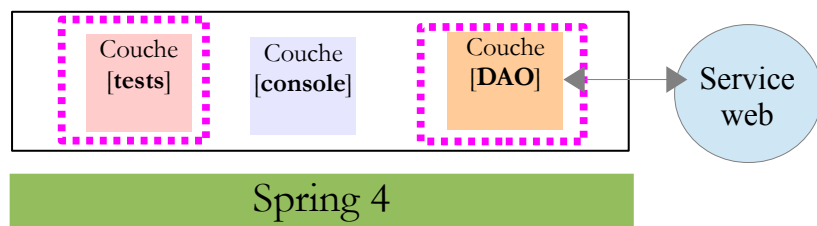
```

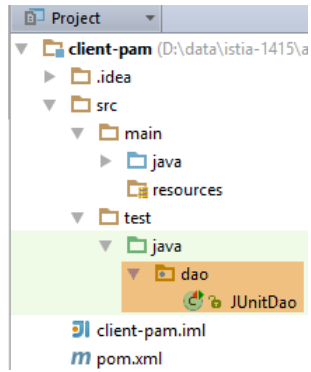
1. package pam.client.config;
2.
3. import org.springframework.context.annotation.ComponentScan;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. @ComponentScan(basePackages = {"pam.client.dao"})
8. public class Config {
9. }

```

- ligne 6 : fait de la classe [Config] une classe de configuration Spring ;
- ligne 7 : demande à ce que le package [pam.client.dao] soit exploré pour trouver des composants Spring. Un seul sera trouvé : la classe [Dao] annotée par l'annotation Spring [@Service]. La classe [Dao] devient ainsi un composant Spring qui peut être injecté dans d'autres composants Spring ;

5.5 Les tests de la couche [DAO]





La classe [JUnitDao] teste la classe [IDao] de la façon suivante :

```

1. package dao;
2.
3. import org.junit.Assert;
4. import org.junit.Before;
5. import org.junit.Test;
6. import org.junit.runner.RunWith;
7. import org.springframework.beans.factory.annotation.Autowired;
8. import org.springframework.boot.test.SpringApplicationConfiguration;
9. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10. import pam.client.config.Config;
11. import pam.client.dao.IDao;
12. import pam.client.entities.Employe;
13. import pam.client.entities.PamException;
14.
15. @SpringApplicationConfiguration(classes = Config.class)
16. @RunWith(SpringJUnit4ClassRunner.class)
17. public class JUnitDao {
18.
19.     @Autowired
20.     private IDao dao;
21.
22.     @Before()
23.     public void init() {
24.         dao.setTimeout(1000);
25.         dao.setUrlServiceRest("http://localhost:8080");
26.     }
27.
28.     @Test
29.     public void test1() {
30.         Assert.assertEquals(72.4, dao.getFeuilleSalaire("260124402111742", 30,
31.             5).getElementsSalaire().getSalaireNet(), 1e-6);
32.         Assert.assertEquals(368.77, dao.getFeuilleSalaire("254104940426058", 150,
33.             20).getElementsSalaire().getSalaireNet(), 1e-6);
34.         boolean erreur = false;
35.         try {
36.             dao.getFeuilleSalaire("xx", 150, 20);
37.         } catch (PamException ex) {
38.             erreur = true;
39.         }
40.         Assert.assertTrue(erreur);
41.     }
42.
43.     @Test
44.     public void test2() {
45.         System.out.println("Liste des employés");
46.         for (Employe employe : dao.getEmployes()) {

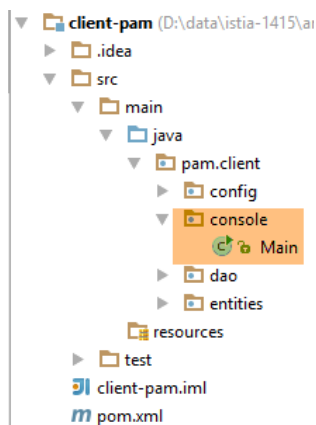
```

```
45.     System.out.println(employe);
46.   }
47. }
48. }
```

- ligne 15 : le test est configuré avec la classe de configuration [Config] ce qui permet au composant [Dao] d'être injecté aux lignes 19-20 ;
- lignes 22-26 : avant chaque test (annotation `@Before`), on fixe le *timeout* des connexions HTTP et l'URL du service web/JSON ;
-

Travail : exécutez ces tests et vérifiez qu'ils passent.

5.6 La couche [console]



Travail : écrire la classe [Main]. Elle doit faire la même chose que dans le TD (cf paragraphe 4.11.1).

6 A rendre

A rendre : les projets IntelliJ IDEA zippés du client et du serveur.

Table des matières

1 LE PROBLÈME.....	1
2 INTRODUCTION À L'IDE INTELLIJ IDEA COMMUNITY EDITION.....	2
3 MANIPULER DU JSON.....	5
4 LE SERVEUR WEB/JSON.....	8
4.1 LE PROJET INTELLIJ IDEA DU SERVEUR.....	8
4.2 INTRODUCTION À SPRING DATA.....	9
4.3 INTRODUCTION À SPRING MVC.....	9
4.4 LE FICHIER [POM.XML].....	10
4.5 LES ENTITÉS JPA.....	12
4.6 LA COUCHE [DAO].....	13
4.7 CONFIGURATION DE LA COUCHE DE PERSISTANCE.....	14
4.8 TESTS DE LA COUCHE [DAO].....	16
4.9 LA COUCHE [MÉTIER].....	17
4.10 TESTS JUNIT DE LA COUCHE [MÉTIER].....	18
4.11 TESTS CONSOLE DE LA COUCHE [MÉTIER].....	20
4.12 LA COUCHE [WEB/JSON].....	21
4.12.1 L'ARCHITECTURE D'UN SERVICE WEB/JSON IMPLÉMENTÉ PAR SPRING MVC.....	22
4.12.2 LES URL DU SERVICE WEB/JSON.....	23
4.12.3 LES RÉPONSES JSON DU SERVICE WEB/JSON.....	24
4.12.4 L'INTERFACE AVEC LA COUCHE [MÉTIER].....	25
4.12.5 LE CONTRÔLEUR.....	27
4.13 CONFIGURATION DE L'APPLICATION WEB.....	31
4.14 LA CLASSE EXÉCUTABLE DE L'APPLICATION WEB.....	32
4.15 CRÉATION D'UNE ARCHIVE EXÉCUTABLE.....	34
5 LE CLIENT.....	36
5.1 DÉPENDANCES MAVEN.....	36
5.2 LES ENTITÉS MANIPULÉES PAR LE CLIENT.....	37
5.3 LA COUCHE [DAO].....	37
5.4 CONFIGURATION DE LA COUCHE [DAO].....	40
5.5 LES TESTS DE LA COUCHE [DAO].....	41
5.6 LA COUCHE [CONSOLE].....	42