@HeaderParam

– permet d'associer une valeur d'un champ HTTP à un champ ou un paramètre d'une méthode de la classe de ressources.

Exemple:

```
@Path("/header")
public class EmployeeService {
    @GET
    @Path("/getemp")
    @Produces("application/json")
    @Formatted
    public Response getEmp(@HeaderParam("User1") String user1) {
        Map<String,String> map = new HashMap<String,String>();
        map.put("User1", user1);
        return Response.ok(map).build();
    }
}
```


Systèmes distribués

Filière INDIA (3ème année)

Pr. Abderrahim EL QADI

Département Mathématique appliquée et Génie Informatique

ENSAM-Université Mohammed V de Rabat

A.U. 2023/2024

Partie 2:

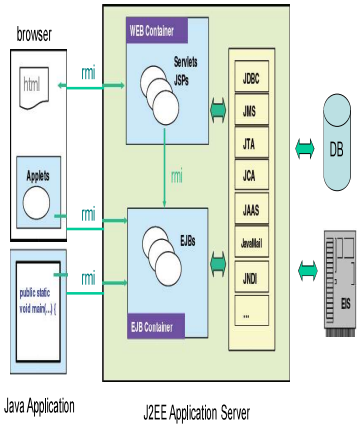
1. J2EE- Java 2ème Edition Entreprise
2. Hibernate & JPA
3. Spring Boot
4. Microservice

Références

1. Polycopie Introduction au langage Java et à l'écosystème Spring. Serge Tahé, mars 2017
2. JPA : Java Persistence API, Olivier Perrin.
3. http://www.info.univ-angers.fr/~richer/ens/m2cdsii/crs_orm.pdf
4. Pro JPA 2, 2nd Edition: Mike Keith, Merrick Schincariol Publisher: Apress, Year: 2013
5. https://o7planning.org/fr/11267/le-tutoriel-de-spring-boot-pour-les-debutants
6. https://www.supinfo.com/articles/single/5676-qu-est-ce-que-architecture-microservices
7. http://microservices.io/
8. Microservices Pattern. Chris Richardson. Chapiter 8
9. https://www.supinfo.com/articles/single/5676-qu-est-ce-que-architecture-microservices
10. http://microservices.io/
11. Java Persistence with Hibernate, 2nd Edition; Christian Bauer, Gavin King, Gary Gregory Publisher: Manning, Year: 2015.
12. http://courses.coreservlets.com/Course-Materials/hibernate.html
13. https://arodrigues.developpez.com/tutoriels/java/performance/hibernate-performance-part1-strategies-chargement/
14. Hibernate et la gestion de persistance (http://litis.univ-lehavre.fr/~duvallet/).
15. JPA: Java Persistance API, Olivier Perrin.
16. http://www.info.univ-angers.fr/~richer/ens/m2cdsii/crs_orm.pdf
17. Pro JPA 2, 2nd Edition; Mike Keith, Merrick Schincariol Publisher: Apress, Year: 2013

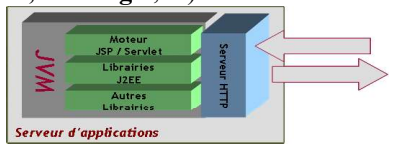
1. J2EE – Java Edition Entreprise

- Est une plate–forme fortement orientée serveur pour le développement et l'exécution des applications distribuées.
- Elle est composée de deux parties essentielles :
  - Ensemble **de spécifications** pour une infrastructure dans laquelle s'exécute les composants écrits en java
  - Ensemble **d'API** (Application Program Interface): permettant de s'interfacer avec le système d'information.
- Les API de J2EE regroupées en trois grandes catégories :
  - Les composants : Servlet, JSP, EJB
  - Les services : JDBC, JTA/JTS, JNDI, JCA, JAAS
  - la communication : RMI–IIOP, JMS, Java Mail.

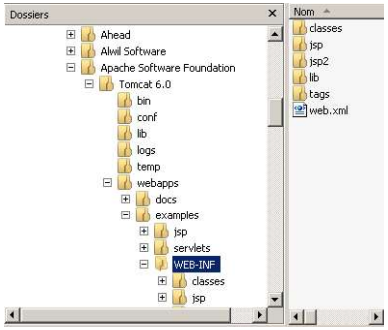


Execution des applications J2EE:

- Pour exécuter des applications web, il faut utiliser un **conteneur web ou serveur d'application** :
  Les conteneurs fournissent des services qui peuvent être utilisés par les applications lors de leur exécution.
- Il existe plusieurs conteneurs définit par J2EE :
  - **conteneur web** : pour exécuter les servlets et les JSP
  - **conteneur d'EJB** : pour exécuter les EJB
  - **conteneur client** : pour exécuter des applications standalone sur les postes qui utilisent des composants J2EE
- Les serveurs d'applications peuvent fournir un conteneur web uniquement (exemple : **Tomcat**) ou un conteneur d'EJB uniquement (exemple : **JBoss, Jonas**, ...) ou les deux (exemple : **Websphere, Weblogic**, ...).



- Une application web doit suivre certaines règles pour être **déployée** au sein d'un **conteneur de web**.
  Soit <webapp> le dossier d'une application web. Une application web est composée de :
  Classes --------> dans le dossier <webapp>\WEB-INF\classes
  archives java --------> dans le dossier <webapp>\WEB-INF\lib
  vues, ressources (.jsp, .html, ...) ------> dans le dossier <webapp> ou des sous-dossiers



- L'application web est configurée par un fichier XML : **<webapp>\WEB-INF\web.xml**

(Description de déploiement du contexte)

- Ces fichiers XML peuvent être créés à la main car leur structure est simple.

Exemple de web.xml : La présence du fichier web.xml est obligatoire pour que les servlets web fonctionnent

En-tête du fichier **web.xml**
Balise principale
Balise de description de l'application WEB
Balise de description d'une Servlet
Nom de la Servlet "Identification"
Classe de la Servlet
Définition d'un chemin virtuel
Nom de la Servlet considéré "Identification"
Définition du chemin virtuel

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
     instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>Application WEB affichant
                  HelloWorld</display-name>
    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/msg.hello</url-pattern>
    </servlet-mapping>
</web-app>
```

- **Jakarta** JAKARTA EE

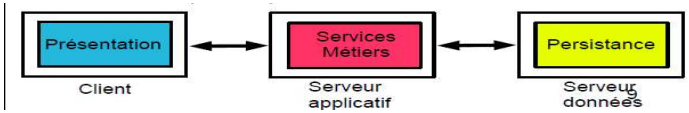
· Jakarta EE (anciennement Java 2 Platform, Enterprise Edition, ou J2EE, puis Java Platform, Enterprise Edition ou Java EE), est une spécification pour la plate-forme Java d'Oracle, destinée aux applications d'entreprise et de cloud.

· Il fournit de nouvelles technologies et fonctionnalités qui accélèrent le développement et l'élargissement des applications d'affaires pour le déploiement dans le cloud.

· La plate-forme étend Java Platform, Standard Edition (Java SE) en fournissant une API de mapping objet-relationnel, des architectures distribuées et multitiers, et des services web.

· La plate-forme se fonde principalement sur des composants modulaires exécutés sur un serveur d'applications.

2. Hibernate et la gestion de persistance



– **Hibernate** est un logiciel, écrit en java, qui permet de faire le mapping entre Objets Java et Objets stockés en base relationnelle.

– est une solution open source de type **ORM** (**Object Relational Mapping**) qui permet de faciliter le développement de la couche persistance d'une application.

– **Hibernate** propose son propre langage d'interrogation **HQL**

– Le site officiel http://www.hibernate.org

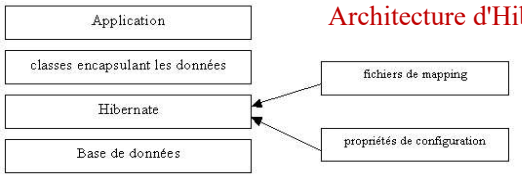
– La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :

- le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
- les relations entre les entités des deux modèles sont différentes.
- les objets ne possèdent pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre).
- Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique.

Hibernate a besoin de plusieurs éléments pour fonctionner :

- **Une classe de type javabean** qui encapsule les données d'une occurrence d'une table (ex. Classe Employe.java)
- **Un fichier de correspondance** qui configure la correspondance entre la classe et la table (ex. EmployeDAO.java et HibernateSessionFactory.java)
- **Des propriétés de configuration** notamment des informations concernant la connexion à la base de données (ex. hibernate.cfg.xml et Employe.hbm.xml)

Architecture d'Hibernate



1.1. Fichier de correspondance

– Pour assurer le mapping, Hibernate a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.

– Même si cela est possible, il n'est pas recommandé de définir un fichier de mapping pour plusieurs classes.

– Le plus simple est de définir un fichier de mapping par classe, nommé du nom de la classe suivi par **".hbm.xml"**.

– Ce fichier doit être situé dans le même répertoire que la classe correspondante ou dans la même archive pour les applications packagées.

– Différents éléments sont précisés dans ce document XML :

- la classe qui va encapsuler les données,

- L'identifiant dans la base de données et son mode de génération,
- le mapping entre les propriétés de classe et les champs de la base de données,
- les relations, etc.

– Le fichier débute par un prologue et une définition de la DTD utilisée par le fichier XML.

– Le tag racine du document XML est le tag **<hibernate-mapping>**

Ce tag peut contenir un ou plusieurs tag <class> : il est cependant préférable de n'utiliser qu'un seul tag <class> et de définir autant de fichiers de correspondance que de classes.

Exemple : **Employe.hbm.xml**

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd ">

<hibernate-mapping>
<class name="Employe" table="Employe">
    <id name="id" type="integer">
        <column name="idEmploye" sql-type="int(4)" />
        <generator class="increment" />
    </id>
    <property name="nom" column="nomEmploye"></property>
    <property name="prenom" column="prenomEmploye"></property>
    <property name="age" column="age"></property>
</class>
</hibernate-mapping>
```


1.2. Fichier de configuration

– Les propriétés de configuration pour la connexion à la base de données, peuvent être fournies sous plusieurs formes :

- un fichier de configuration nommé **hibernate.properties** et stocké dans un répertoire inclus dans le **classpath**
- un fichier de configuration au format XML nommé hibernate.cfg.xml
- utiliser la méthode setProperties() de la classe Configuration
- définir des propriétés dans la JVM en utilisant l'option propriété=valeur

– Les principales propriétés pour configurer la connexion JDBC sont :

Nom de la propriété	Rôle
hibernate.connection.driver_class	nom pleinement qualifié de la classe du pilote JDBC
hibernate.connection.url	URL JDBC désignant la base de données
hibernate.connection.username	nom de l'utilisateur pour la connexion
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.connection.pool_size	nombre maximum de connexions dans le pool
hibernate.dialect	nom de la classe pleinement qualifiée qui assure le dialogue avec la base de données
hibernate.show_sql	booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console

Exemple : **hibernate.cfg.xml**

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd ">

<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
      <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:bd1</property>
       <property name="hibernate.connection.username">system</property>
       <property name="hibernate.connection.password">bd1</property>
       <property name="hibernate.connection.pool_size">1</property>
       <property name="hibernate.current_session_context_class">thread</property>
       <property name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>

       <property name="hibernate.show_sql">false</property>
       <property
name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
       <!-- Mapping file  -->
             <mapping resource="Employe.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```


1.3. Utilisation d'Hibernate

– Pour utiliser le fichier **hibernate.cfg.xml**, il faut :

- créer une occurrence de la classe Configuration,
- appeler sa méthode configure() qui va lire le fichier XML,
- appeler la méthode **buildSessionFactory()** de l'objet renvoyer par la méthode **configure()**

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateSessionFactory {
   private static final SessionFactory sessionFactory = buildSessionFactory();
   private static SessionFactory buildSessionFactory() {
      try {
         SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
         return sessionFactory;
      } catch (Throwable ex) {
         System.err.println("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);       }   }
```



```
   public static SessionFactory getSessionFactory() {
      return sessionFactory;
   }
   public static void closeSession() {
      getSessionFactory().close();
   }
   public static Session currentSession() {
         Session session =sessionFactory.openSession();
         return session;
   }
}
```

1.3.1. La persistance d'une nouvelle occurrence

– Pour créer une nouvelle occurrence dans la source de données, il suffit de :
  - créer une nouvelle instance de la classe encapsulant les données,
  - valoriser ses propriétés
  - et d'appeler la méthode **save()** de la session en lui passant en paramètre l'objet encapsulant les données.

```
Session session = HibernateSessionFactory.getSession(); // create session
Transaction tr = session.beginTransaction();   // create and begin transaction
Employe emp = new Employe();
emp.setNom("Alaoui");
emp.setPrenom("Said");
emp.setAge(new Integer(20));
session.save(emp);
tr.commit();       session.close();
```


1.3.2. Recherche une occurrence à partir de son identifiant

```
Session session = HibernateSessionFactory.getSession();
try {
      Employe emp = (Employe) session.load(Employe.class, new Integer(1));
      System.out.println("nom = " + emp.getNom());
} finally {
     session.close();
 }
```

```
List list = session.createQuery("from Employe").list();
Iterator it = list.iterator();
while(it.hasNext())
{
     Employe e = (Employe)it.next();
     System.out.println(e.getId()+" "+e.getPrenom()+" "+ e.getNom()+": "+
              e.getAge()+" ans");
}
```


1.3.3. La mise à jour d'une occurrence

– Pour mettre à jour une occurrence dans la source de données, il suffit d'appeler la méthode **update()** de la session en lui passant en paramètre l'objet encapsulant les données.

– Le mode de fonctionnement de cette méthode est similaire à celui de la méthode save().

– La méthode **saveOrUpdate()** laisse Hibernate choisir entre l'utilisation de la méthode save() ou update() en fonction de la valeur de l'identifiant dans la classe encapsulant les données.

Exemple:

```
String hql = "update Employe set nom = :nom,  prenom=:prenom, age=:age
 where id = :id";
 Query query = session.createQuery(hql);
```


1.3.4. La suppression d'une ou plusieurs occurrences

– La méthode **delete()** de la classe Session permet de supprimer une ou plusieurs occurrences en fonction de la version surchargée de la méthode utilisée.

– Pour supprimer une occurrence encapsulée dans une classe, il suffit d'invoquer la méthode delete() en lui passant en paramètre l'instance de la classe

Exemple: suppression de toutes les occurrences de la table

```
session.delete("from Employe");

String hql = "delete from Employe where id = :id";
Query query = session.createQuery(hql);
```

2. Java Persistence API (JPA)

– JPA Simplifie le modèle de persistance
– permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC
– repose sur des entités qui sont de simples POJOs (Plain Old Java Object) annotés
  • Un POJO est une classe Java qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.
– Un objet Java de type POJO mappé vers une table de la base de données grâce à des méta data via l'API Java Persistence est nommé bean entité (Entity bean).

2.1. Les Entités

– Ensemble d'annotations permettant de définir les entités
  • attributs
  • clé
  • relations
  • …
– permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables.
  – Deux possibilités pour définir le mapping
– annotations
– ou fichier de mapping
  – L'API propose plusieurs annotations pour supporter un mapping O/R assez complet.

Annotation	Rôle
@javax.persistence.Table	Préciser le nom de la table concernée par le mapping
@javax.persistence.Column	Associer un champ de la table à la propriété (à utiliser sur un getter)
@javax.persistence.Id	Associer un champ de la table à la propriété en tant que clé primaire (à utiliser sur un getter)
@javax.persistence.GeneratedValue	Demander la génération automatique de la clé primaire au besoin
@javax.persistence.Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
@javax.persistence.Transient	Demander de ne pas tenir compte du champ lors du mapping

Exemple de mapping simple:

– Pour qu'une classe puisse être persistante, il faut
  • qu'elle soit identifiée comme une entité (**entity**) en utilisant l'annotation **@java.persistence.Entity**
  • qu'elle possède un attribut identifiant en utilisant l'annotation **@javax.persistence.Id**
  • qu'elle ait un **constructeur sans argument**

Employe
id : Integer nom: String prenom: String age : Integer

Relationnel

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Employe {
@Id
private int id;
private String nom;
private String prenom;
private int age;
…
// constructeurs/setter/getter
}
```

Java

2.2. Clé primaire

- Une entité doit avoir un attribut qui correspond à la clé primaire de la table associée
- L'attribut clé primaire est désigné par l'annotation **@Id**
- Si la clé est de type numérique:
  - @GeneratedValue indique que la clé sera automatiquement générée par le SGBD
  - l'annotation peut avoir un attribut strategy qui indique comment la clé sera générée:


```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
```

- **AUTO**: le SGBD choisit (valeur par défaut)
- **SEQUENCE**: il utilise une séquence SQL
- **IDENTITY**: il utilise un générateur de type IDENTITY (auto increment dans MySQL par exemple)
- **TABLE**: il utilise une table qui contient la prochaine valeur de l'identificateur.

2.3. Attributs

- L'annotation **@Column** possède plusieurs attributs :
  - name(): nom de l'attribut
  - unique(): la valeur est-elle unique ?
  - nullable(): accepte une valeur nulle ?
  - insertable(): autorise ou non l'attribut à être mis à jour
  - columnDefinition(): définition DDL de l'attribut
  - table(): lorsque l'attribut est utilisé dans plusieurs tables
  - length(): longueur max
  - precision(): précision pour les valeurs numériques

Exemple:

```
@Entity
public class Employe {
   @Id
   @Column(name="ID")
  //@GeneratedValue(strategy = GenerationType.AUTO)
   private int id;

   @Column(name = "NOM", nullable = false, length = 30)
   private String nom;

   @Column(name = "PRENOM", length = 20)
   private String prenom;

   @Column(name = "AGE")
   private int age;
// constructeurs
//getters et setters
}
```

2.4. Clé composite primaire

- Relation existante avec clé multi-attributs
- 2 possibilités :
  - **@IdClass**: correspond à plusieurs attributs Id dans la classe entité
  - **@EmbeddedId** et **@Embeddable**: un seul attribut Id dans la classe entité
- Dans les 2 cas, la clé doit être représentée par une classe Java
  - les attributs correspondent aux composants de la clé
  - la classe doit être publique
  - la classe doit avoir un constructeur sans paramètre
  - la classe doit être sérializable et redéfinir equals et hashcode

-

a. @EmbeddedId

- Correspond au cas où la classe entité comprend un seul attribut annoté @EmbeddedId
- La classe clé primaire est annotée par @Embeddable

```
@Entity
public class Personne implements Serializable {
    @EmbeddedId
    private PersonnePK clePrimaire;
    …
}
```


b. @IdClass

- Correspond au cas où la classe entité comprend plusieurs attributs annotés par @Id
- La classe entité est annotée par @IdClass qui prend en paramètre le nom de la classe clé primaire
- La classe clé primaire n'est pas annotée
- ses attributs ont les mêmes noms et mêmes types que les attributs annotés @Id dans la classe entité.

Exemple:

```
@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
    private String prenom;
    private String nom;
....
}
```


2.5. Les relations

- Hibernate et JPA proposent de transcrire les relations du modèle relationnel dans le modèle objet. Il supporte plusieurs types de relations :
  - relation de type 1 - 1 (one-to-one).
  - relation de type 1 - n (many-to-one).
  - relation de type n - n (many-to-many).
- Dans le fichier de mapping, il est nécessaire de définir les relations entre la table concernée et les tables avec lesquelles elle possède des relations.

Exemple de relation un / un:

Table Etudiant (
Id int(11) primary key, Nom varchar(50), Prenom varchar(50), age int,
numero_adr varchar(20), cp_adr varchar(5), ville_adr varchar(80)
)

```
public class Etudiant {
    private Long id;
    private String nom;
    private String prenom;
    private int age;
    private Adresse adresse;
    // constructeurs
    // getter et setter sur les champs de la classe
  }
```

```
public class Adresse {
    private String numero;
    private String cp;
    private String ville;
    public Adresse(String numero, String cp, String ville) {
        super();
        this.numero = numero;   this.cp = cp;
        this.ville = ville; }
    public Adresse() { }
    // // getter et setter sur les champs de la classe //
    }
```


- Le fichier de mapping de l'entité Adresse (**Adresse.hbm.xml**) possède plusieurs caractéristiques liées au type de la relation utilisée avec l'entité Etudiant :
  - Le champ identifiant **id** est défini avec un générateur de type **foreign** avec un paramètre qui précise que la valeur sera celle de l'identifiant du champ etudiant
  - La relation inverse avec Etudiant est définie avec un tag <one-to-one> avec l'attribut **constrained** ayant la valeur true


```
<?xml version="1.0"?> <!DOCTYPE hibernate-mapping PUBLIC "-
//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name=« Etudiant" table=« Etudiant">
     <id name="id" column="id">
       <generator class="increment" /> </id>
     <property name="nom" column="Nom" />
     <property name="prenom" column="Prenom" />
     <property name="age" column="age" />
     <component name="adresse" class=«Adresse">
       <property name=« numero" column=« numero_adr" />
       <property name="cp" column="cp_adr" />
       <property name="ville" column="ville_adr" />
   </class>
</hibernate-mapping>
```

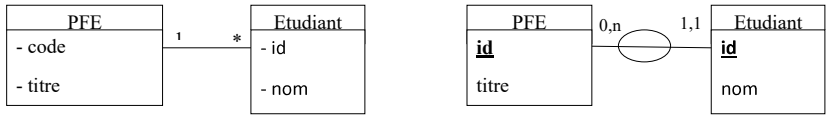

- Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux fichiers de mapping des entités.

```
<?xml version='1.0' encoding='UTF-8'?> <!DOCTYPE hibernateconfiguration
PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration3.0.dtd">
<hibernate-configuration>
   <session-factory>
       ……
       <mapping resource=« Etudiant.hbm.xml"></mapping>
       <mapping resource=« Adresse.hbm.xml"></mapping>
     </session-factory>
</hibernate-configuration>
```

- L'application de test est basique :
  - créer une instance de type Adresse,
  - créer une instance de type Etudiant,
  - assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Etudiant à l'instance de l'adresse. Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels
  - et sauvegarder l'éleve dans la base de données


```
int index = 3;
Session session = sessionFactory.openSession();
try {
   transaction = session.beginTransaction();
   Adresse adresse = new Adresse(« numero" + index, "cp_" + index,
      "ville" + index);
   Etudiant eleve = new Etudiant("nom" + index, "prenom_" +
     index, null, age);
   adresse.setEtudiant(eleve);
   session.save(eleve);
   transaction.commit();
    System.out.println(« le nouveau etudiant a ete enregistre");
  }
  catch (Exception e) { transaction.rollback(); e.printStackTrace(); }
  finally { session.close();
  }
```


Exemple de mapping Relation many-to-one:



Modèle orienté objet

```
@Entity
public class Etudiant {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private long id;   // (clé primaire)
   private String nom;
   private String prenom;
   private int age
// constructeurs
//getters et setters
@ManyToOne
  Private Pfe pfe;
}
```

Modèle relationnel

```
import javax.persistan e.Id
@Entity
public class Pfe {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private long id;    // (clé primaire)
   private String titre;
   private String auteur;
// deux constructeurs: sans et avec paramètres
//getters et setters
  @OneToMany(mappedBy="pfe")
      Private Collection <Etudiant> listetudiant;
}
```


3. Spring

3.1. Architecture Spring MVC

– Un conteneur J2EE est un environnement d'exécution chargé de gérer des composants applicatifs et leur donner accès aux API J2EE.

– Il fournit des services qui peuvent être utilisés par les applications lors de leur exécution.

– Pour exécuter des applications web, il faut utiliser un conteneur web ou serveur d'application

– Il existe de nombreuses versions :
  - Commerciales tel que **IBM Webpshere** ou **BEA WebLogic**
  - Libres tel que **Tomcat** du projet GNU

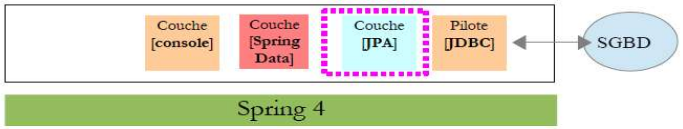
– **Spring est un conteneur dit « léger** », c'est-à-dire une infrastructure similaire à un serveur d'application J2EE.

- Il prend en charge la création d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets.
- Le gros avantage par rapport aux serveurs d'application est qu'avec Spring, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le Framework (au contraire des serveurs d'applications J2EE). C'est en ce sens que Spring est qualifié de **conteneur « léger** ».

– **Spring** est un Framework de développement d'applications Java, qui apporte plusieurs fonctionnalités :

- Spring Security, Spring MVC, Spring Batch, Spring Ioc, Spring Data, etc.
- Ces Framework ont pour objectif de faciliter la tâche aux développeurs.
- Malheureusement, leurs mises en œuvre deviennent très complexes à travers les fichiers de configuration XML qui ne cessent de grossir.
- Solution : Utilisation de Spring Boot

Exemple d'architecture MVC Spring:



1. La couche JPA : L'accès à la base de données se fait au travers d'une couche [JPA], Java Persistence API

```
@Entity  // couche JPA
public class Etudiant {
      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private long id;
      private String nom;
      private String prenom;
      private String reffil;
      // ajouter le constructeur sans et avec des paramètres
      //ajouter les méthodes getters et setters
      // ajouter la méthode ToString
}
```


2. La couche [Spring Data]: La classe [EtudiantRepository] implémente la couche d'accès à la table [Etudiant]. Son code est le suivant :

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EtudiantRepository extends JpaRepository<Etudiant, Long>{
      List<Etudiant> findByReffil(String reffil);
}
```

L'interface JpaRepository définit les opérations CRUD (Create – Read – Update – Delete) qu'on peut faire sur un type JPA.

@Repository : cette annotation est appliquée à la classe afin d'indiquer à Spring qu'il s'agit d'une classe qui gère les données, ce qui nous permettra de profiter de certaines fonctionnalités comme les translations des erreurs.

3. La couche [console] : Exemple de la classe [Application]

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired    // le bean [EtudiantRepository] est injecté dans le code de la classe principale
    EtudiantRepository dao;
    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }
    @Override
    public void run(String... strings) throws Exception {
        // save des etudiants
        dao.save(new Etudiant("Ali", "Said",'IDDL"));
        dao.save(new Etudiant("Mark", "Said",'IDDL"));
```



```
        System.out.println("Etudiants found with findAll():");
        System.out.println("-------------------------------");
        for (Etudiant e : dao.findAll()) {              // recherche tous les étudiants
            System.out.println(e);
        }
        Etudiant e = dao.findOne(1L);              // rech des étudiants par ID
        System.out.println("Etudiant found with findOne(1L):");
        System.out.println("-------------------------------");
        System.out.println(e);
        System.out.println("Etudiant found with findByLastName('Ali'):");
        System.out.println("-------------------------------------------");
        for (Etudiant e1 : dao.findByLastName("Ali")) {  // rech étudiant par nom
            System.out.println(e1i);
        }
    }
}
```


@SpringBootApplication est une annotation regroupant plusieurs annotations [Spring Boot] :

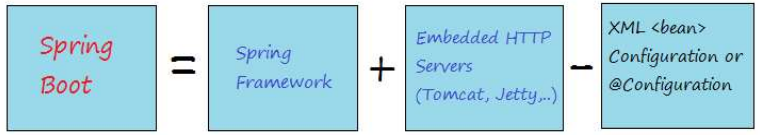
- **@Configuration** : indique que la classe est une classe de configuration ;
- **@EnableAutoConfiguration** : demande à [Spring Boot] de créer lui-même un certain nombre de beans en fonction de diverses propriétés, en particulier le contenu du Classpath du projet :
  - Parce que les bibliothèques Hibernate sont dans le Classpath, le bean [entityManagerFactory] sera implémenté avec Hibernate.
  - Parce que la bibliothèque du SGBD H2 est dans le Classpath, le bean [dataSource] sera implémenté avec H2.
  - Dans le bean [dataSource], on doit définir également l'utilisateur et son mot de passe. Ici Spring Boot utilisera l'administrateur par défaut de H2, "**sa**" sans mot de passe.
  - Parce que la bibliothèque [spring-tx] est dans le Classpath, c'est le gestionnaire de transactions de Spring qui sera utilisé ;

- **@EnableWebMvc** : si dans le Classpath se trouve la bibliothèque [spring-mvc].
  - Dans ce cas, une auto-configuration est faite pour l'application web ;
- **@ComponentScan** : qui dit à Spring où chercher les autres beans, configurations et services.
  - Ici ils sont cherchés par défaut dans le package contenant la classe taguée, ç-à-d le package du projet.
  - Ainsi les classes [Etudiant] et [EtudiantRepository] vont-elles être trouvées.
  - Parce que la première a l'annotation [@Entity] elle sera cataloguée comme entité à gérer par Hibernate.
  - Parce que la seconde étend l'interface [CrudRepository] elle sera enregistrée comme bean Spring ;

- La méthode statique [SpringApplication.run] est exécutée.
  - Son premier paramètre est une classe de configuration Spring, ici la classe [SpringbootBiblioApplication].
  - Son deuxième paramètre est ici la liste des arguments passés à la méthode [main].
  - La méthode statique [SpringApplication.run] a pour rôle de créer le contexte Spring, ç-à-d créer les différents beans trouvés soit dans les classes de configuration soit dans les dossiers explorés par l'annotation [@ComponentScan].

3.2. Spring Boot

– **Spring Boot** est un sous projet de Spring qui vise à rendre Spring plus facile d'utilisation en élimant plusieurs étapes de configuration.
- Il permet aux développeurs de se concentrer sur des tâches techniques et non des tâches de configurations, de déploiements, etc.
- Il fournit des serveurs intégrés (Embedded HTTP servers) comme Tomcat, Jetty .... afin de développer et de tester des applications web à la manière la plus facilement.
- Spring boot permet de créer l'application Web Java qui exécute par la ligne de commande **'java -jar'** ou exporter le fichier war pour déployé sur le Web Server comme d'habitude.



– **Spring Boot** diminue énormément du temps et augmente la productivité.
- Il évite d'écrire plusieurs codes d'expressions standard, des Annotations et des configurations XML (**On n'a plus besoin des fichiers XML à configurer** (pas besoin du fichier du descripteur de déploiement web.xml dans le cas d'une application web)).
- Il fournit beaucoup de plugins afin de :
  – développer et de tester des applications Spring Boot rapidement en utilisant les outils de Build comme Maven et Gradle
  – travailler avec des serveurs intégrés et la base de données stockées dans la mémoire (database H2) facilement.
- Il permet de **déployer très facilement une application dans plusieurs environnements sans avoir à écrire des scripts**.

Pour ce faire, une simple indication de l'environnement (développement ou production) dans le fichier de propriétés **(.properties)** suffit à déployer l'application dans l'un ou l'autre environnement.


```
// Exemple : l'accès à la base H2

spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:biblio2;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.name=biblio
#The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.datasource.username=sa
spring.datasource.password=123456

spring.jpa.hibernate.ddl-auto=update
                                                    //
```

Exemple : l'accès à la base MySql

```
spring.datasource.url== jdbc:mysql://localhost:3306/hibernatedb
spring.db.driver= com.mysql.jdbc.Driver
spring.datasource.username = su
spring.datasource.password =
spring.jpa.show-sql = true
#spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.strategy =
org.hibernate.cfg.ImprovedNamingStrategy
```


- Spring Boot possède **un serveur d'application Tomcat embarqué** afin de faciliter le déploiement d'une application web.

Il est possible d'utiliser un serveur autre ou externe, grâce à une simple déclaration dans le fichier pom.xml.

- Spring Boot permet de mettre en place **un suivi métrique de l'application** une fois déployée sur le serveur afin de suivre en temps réel l'activité du serveur, ceci grâce à **spring-boot-starter-actuator**.

Les dépendances:

- La dépendance **spring-boot-starter-parent** permet de rapatrier la plupart des dépendances du projet. Sans elle, le fichier pom.xml serait plus complexe.
- La dépendance **spring-boot-starter-web** indique à Spring Boot qu'il s'agit d'une application web, ce qui permet à Spring Boot de rapatrier les dépendances comme **SpringMVC, SpringContext**, et même le serveur d'application **Tomcat**, etc.

- Spring Boot propose 2 fonctionnalités principales:
  - l'auto-configuration,
  - les starters.

L'auto-configuration

- Spring Boot permet de **configurer automatiquement** l'application à partir des *jar* trouvés dans Classpath.
- Il consulte la liste des dépendances importées, puis produira la configuration nécessaire pour que tout fonctionne correctement.
  - L'annotation [@SpringBootApplication] est un raccourci pour les trois annotations **[@Configuration**, **@EnableAutoConfiguration**, **@ComponentScan**]

Les Starters

– Les starters viennent compléter l'auto-configuration et font gagner énormément de temps.

– Un starter va apporter au projet un **ensemble de dépendances**, communément utilisées pour un type de projet donné. Ceci va permettre de créer un **"squelette" prêt à l'emploi** très rapidement.

– L'autre énorme avantage est la **gestion des versions**.
  - Plus besoin de chercher quelles versions sont compatibles puis de les ajouter une à une dans le *pom.xml*
  - Il suffit d'ajouter une simple dépendance à un starter. Cette dépendance va alors ajouter, à son tour, les éléments dont elle dépend, avec les bonnes versions.

Exemple : Pour créer un Microservice. En temps normal, on aura besoin des dépendances suivantes :

- Spring ;
- Spring MVC ;
- Jackson (pour json) ;
- Tomcat ;
- ...

Avec Spring Boot, il suffit d'avoir une seule dépendance dans le fichier pom.xml :

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```


Tous les starters de Spring Boot sont au format ***spring-boot-starter-***

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starters</artifactId>
      <version>1.5.9.RELEASE</version>
   </parent>
   <artifactId>spring-boot-starter-web</artifactId>
   <name>Spring Boot Web Starter</name>
   <description>Starter for building web, including RESTful, applications using Spring MVC.
   </description>
    <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter</artifactId>
      </dependency>
```



```
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
      </dependency>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-validator</artifactId>
      </dependency>
      <dependency>
         <groupId>com.fasterxml.jackson.core</groupId>
         <artifactId>jackson-databind</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-webmvc</artifactId>
      </dependency>
   </dependencies>
</project>
<parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>
```

Ce tag, ajouté en haut du pom.xml, permet au fichier pom d'hériter des propriétés d'un autre pom (qui lui-même hérite d'un autre pom : *spring-boot-dependencies*).
Il permet de définir principalement :

- La version de Java à utiliser.
- Une liste complète des versions des dépendances prises en charge.

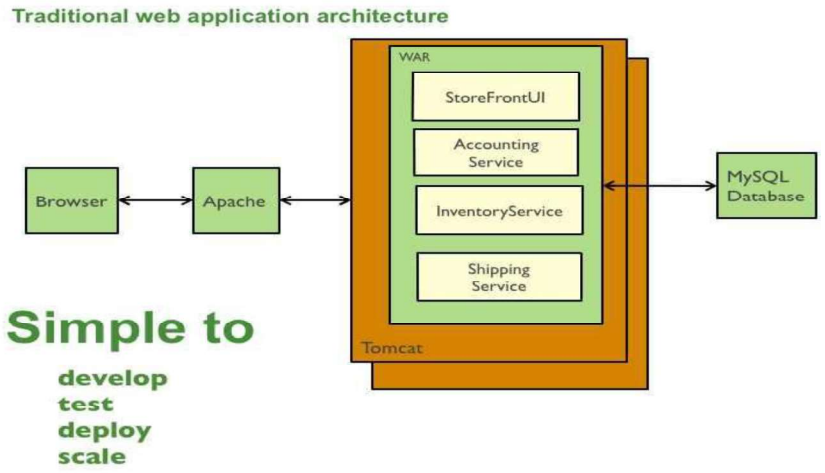
4. MicroServices

– Les microservices, également appelés architecture de microservices, sont un style architectural qui structure une application sous la forme d'un ensemble de services

– L'architecture des microservices permet la livraison rapide, fréquente et fiable de grandes applications complexes. Cela permet également à une organisation de faire évoluer sa pile technologique.

– Netflix, eBay, Amazon, Twitter, et bien d' autres sites Web à grande échelle et les applications ont tous évolué de l'architecture monolithique à microservices.

– Dans une architecture traditionnelle, l'application est au format WAR qui comporte tous les composants.



– L'architecture traditionnelle présente de nombreux avantages :

– Simple à développer - l'objectif des outils de développement et des IDE actuels est de soutenir le développement d'applications monolithiques ;

– Simple à déployer - il suffit de déployer le fichier WAR (ou la hiérarchie de répertoires) sur le runtime approprié ;

– Simple à mettre à l'échelle l'application en exécutant plusieurs copies de l'application derrière un équilibreur de charge.

– Cependant, une fois que l'application devient volumineuse et que l'équipe s'agrandit, cette approche présente un certain nombre d'inconvénients qui deviennent de plus en plus importants :

  – La grande base de code monolithique intimide les développeurs, en particulier ceux qui sont nouveaux dans l'équipe. L'application peut être difficile à comprendre et à modifier.

  – IDE surchargé - plus la base de code est grande, plus l'EDI est lent et les développeurs sont moins productifs.

  – Conteneur Web surchargé - plus l'application est volumineuse, plus le démarrage est long. Cela a eu un impact énorme sur la productivité des développeurs en raison du temps perdu à attendre le démarrage du conteneur. Cela a également un impact sur le déploiement.

– L'architecture microservices propose une solution en principe simple :

  • Découper une application en petits services, appelés microservices, parfaitement autonomes qui exposent une API REST que les autres microservices pourront consommer ;

  • Chaque microservice a sa propre base de données, son propre serveur d'application (tomcat, jetty, etc.), ses propres librairies, administré par une petite équipe, hautement maintenable et testable, et être déployé indépendamment des autres services,

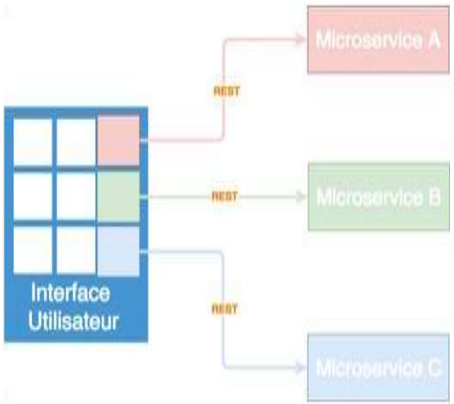
  • La plupart du temps ces microservices sont chacun dans un container Docker, ils sont donc totalement indépendants y compris vis-à-vis de la machine sur laquelle ils tournent.

Exemple d'application : Affichage d'un produit à vendre

– Dans le schéma, l'interface utilisateur fait appel à un microservice pour chaque composant à renseigner.

  • Ainsi, celle-ci peut faire une requête *REST* au **Microservice A**, qui s'occupe de la gestion des photos des produits, afin d'obtenir celles correspondant au produit à afficher.

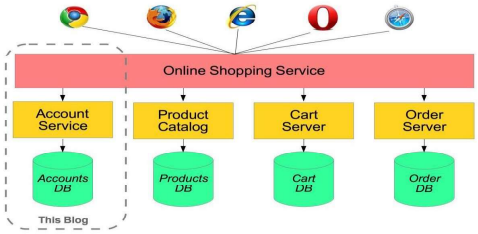
  • De même, les **Microservices B et C** s'occupent respectivement des descriptifs et des prix



– Lorsqu'un utilisateur demande une fiche de produit, l'application applique sa logique interne et va puiser dans une base de données, puis produit un HTML final.

– L'architecture de microservices est considérée comme particulièrement idéale lorsqu'on veut gérer la compatibilité avec une gamme de plates-formes et de périphériques, couvrant le web, le mobile, l'IOT, etc.

  • Par exemple, imaginons une boutique en ligne avec des microservices distincts pour les comptes d'utilisateurs, le traitement des commandes de catalogue de produits et les paniers d'achat :



Il y a inévitablement un certain nombre de solutions mobiles à installer et configurer pour construire un tel système.

4.1. Micro services Pourquoi ?

Problème n°1 :

Les entreprises se livrent à une "**guerre de la mise à jour**". Il faut que l'entreprise soit capable de faire évoluer son application de façon très fréquente et de répondre rapidement aux nouvelles fonctionnalités que propose la concurrence.

- Or, avec une application traditionnelle, le processus de mise à jour est long et compliqué car il faut mettre à jour **toute l'application**.
- Les entreprises sont donc obligées de passer à des architectures qui permettent de faire des mises à jour qui visent des **composants ciblés**, très rapidement, de façon fiable et sans se soucier des éventuelles conséquences sur le reste de l'application.

Problème n°2 :

Les technologies utilisées pour développer ces applications **évoluent très vite** et les nouveautés offrent parfois des avantages énormes. Comment les entreprises peuvent-elles s'adapter rapidement pour tirer profit de ces évolutions ?

- Une solution est d'adopter une architecture qui lui permettra de **passer d'une technologie à l'autre sur des portions individuelles** de son application très facilement.

Problème n°3 :

Comment faire en sorte que les applications proposées en ligne soient **toujours disponibles** et ne souffrent jamais de coupure ou de ralentissement, quelle que soit l'affluence des utilisateurs sur celles-ci?

- Une solution est d'utiliser le **cloud**. Celui-ci permet d'augmenter et de diminuer le nombre de ressources nécessaires au fonctionnement d'une application à la demande.
- Il faut envisager, la conception de l'application afin que celle-ci soit "**Cloud-native**", l'application conçue de façon à ce qu'elle puisse être **scalable**, c'est-à-dire **capable de s'étendre sur plus de ressources** (plus de serveurs, de disques durs, de bases de données), tout en gardant une parfaite consistance dans ses données et une cohérence dans son comportement.

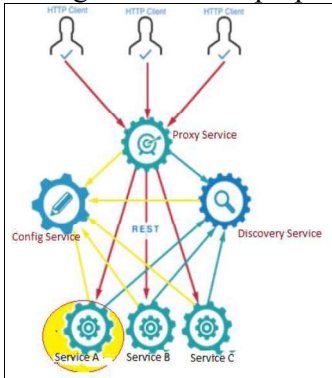
Problème n°4 :

La sécurité, qui devient très compliquée à gérer dans les applications complexes, ainsi que les difficultés liées à la coordination entre développeurs, et bien d'autres !

- Il faut donc que les entreprises conçoivent leurs applications dès le départ à partir d'architectures répondant à ces besoins et qui sont **parfaitement adaptées au cloud**.

4.2. Mise en œuvre l'architecture microservices

– Dans une architecture microservices, plusieurs services s'exécutent en même temps, sur des processus différents, avec chacun sa propre configuration et ses propres paramètres.

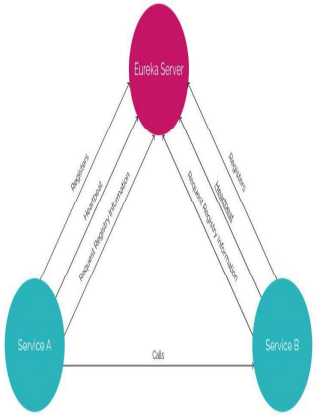


– Service A : Service principal, qui offre une API REST pour lister une liste de clients.
– Service B : Service principal, qui offre une API REST pour traiter les commandes des clients.
– *Config Service* : Service de configuration, dont le rôle est de centraliser les fichiers de configuration des différents microservices dans un endroit unique.
– *proxy Service* : Passerelle se chargeant du routage d'une requête vers l'une des instances d'un service, de manière à gérer automatiquement la distribution de charge.
– *Discovery Service*: Service permettant l'enregistrement des instances de services en vue d'être découvertes par d'autres services

– Spring Cloud fournit des outils pour les développeurs pour construire rapidement et facilement des patrons communs de systèmes répartis (tel que des services de configuration, de découverte ou de routage intelligent).

– **Spring Cloud Config** fournit un support côté serveur et côté client pour externaliser les configurations dans un système distribué. Grâce au service de configuration, il est possible d'avoir un endroit centralisé pour gérer les propriétés de chacun de ces services.

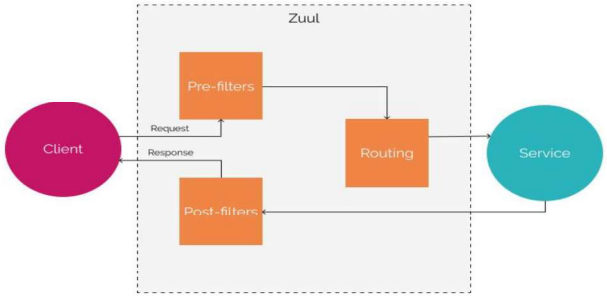
– **Eureka, registre de services au cœur de l'architecture :**



- Eureka est une application permettant la localisation d'instances de services.
- Elle se caractérise par une partie serveur et une partie cliente.
- La communication entre les parties se fait via les API Web exposées par le composant serveur.
- Ces services doivent être créés en tant que clients Eureka, ayant pour objectif de se connecter et s'enregistrer sur un serveur Eureka.
- De ce fait, les clients vont pouvoir s'enregistrer auprès du serveur et périodiquement donner des signes de vie.

Le service Eureka (composant serveur) va pouvoir conserver les informations de localisation desdits clients afin de les mettre à disposition des autres services (service registry).

– **Routage et équilibrage de charge côté serveur avec Zuul**



- Dans une architecture de microservices, il peut y avoir des dizaines, des centaines voire des milliers de services. Beaucoup sont privés et internes, mais certains doivent être exposés au monde extérieur. Nous avons besoin d'un point d'entrée unique dans le système pour nous permettre de câbler et d'exposer certains services au monde extérieur.