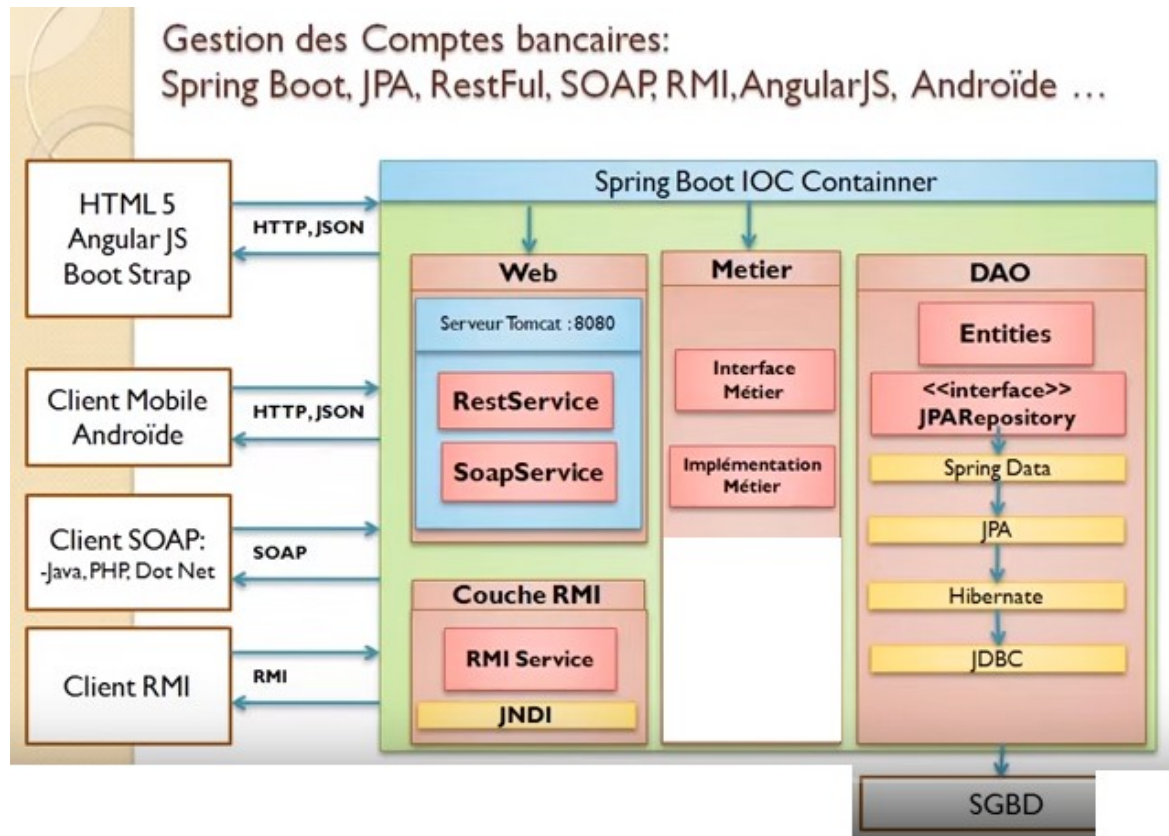


TP : Banque Spring Boot JPA Hibernate Restful et Angular JS



Projet Systèmes Distribués

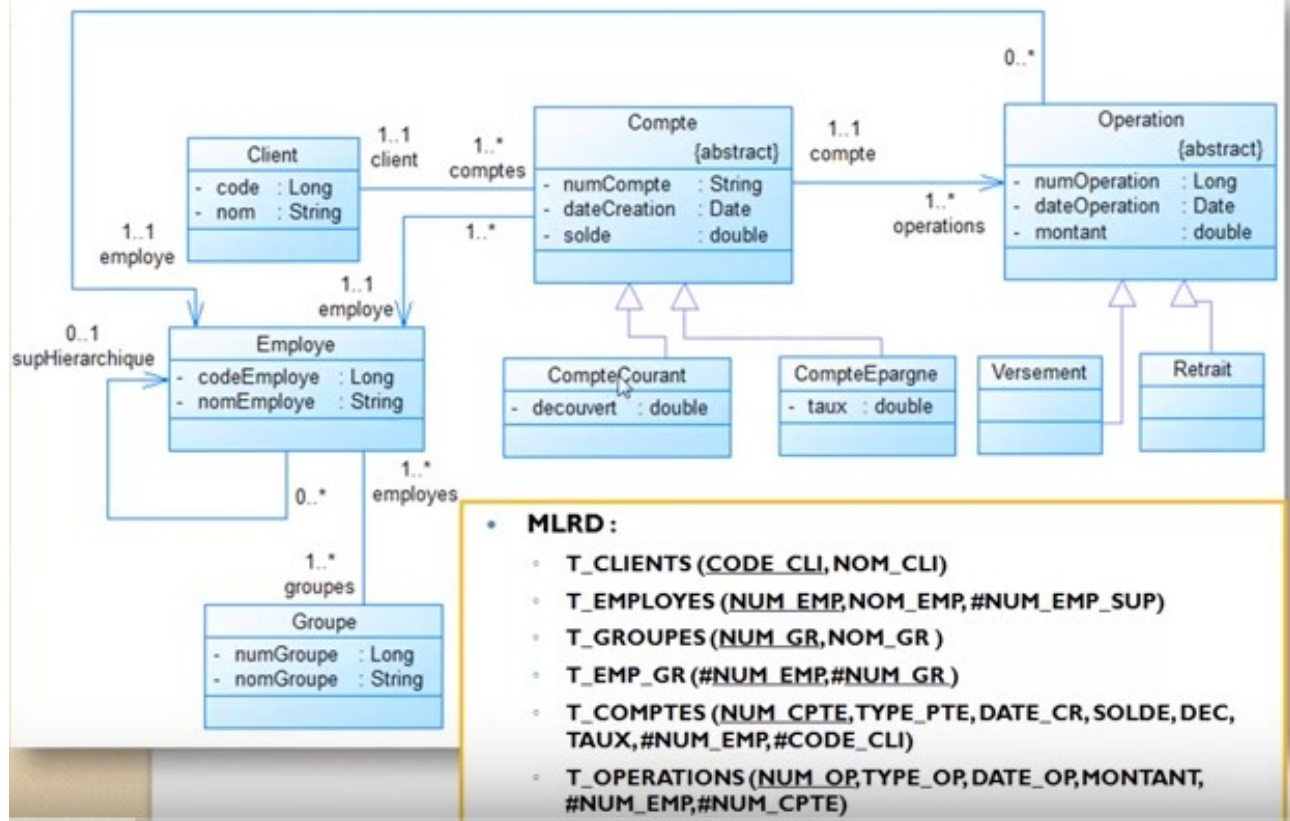
On souhaite créer une application qui permet de gérer des comptes bancaire.

- Chaque compte est défini un numéro, un solde et une date de création
- Un compte courant est un compte qui possède en plus un découvert
- Un compte épargne est un compte qui possède en plus un taux d'intérêt.
- Chaque compte appartient à un client et créé par un employé.
- Chaque client est défini par son code et son nom
- Un employé est défini par son code et sont nom.
- Chaque employé possède un supérieur hiérarchique.
- Chaque employé peut appartenir à plusieurs groupes
- Chaque groupe, défini par un code est un nom, peut contenir plusieurs employés.
- Chaque compte peut subir plusieurs opérations.
- Il existe deux types d'opérations :Versement et Retrait
- Chaque opération est effectuée par un employé.
- Une opération est définie par un numéro, une date et un montant.

Spécifications fonctionnelles

- L'application doit permettre de :
 - Ajouter des groupes
 - Ajouter des employés
 - Affecter les employés aux groupes
 - Ajouter des clients
 - Ajouter des comptes
 - Effectuer des versements dans un compte
 - Effectuer des retraits dans un compte
 - Effectuer des virements d'un compte vers un autre.
 - Consulter un compte
 - Consulter les comptes d'un client
 - Consulter une page une d'opérations concernant un compte.

Diagramme de classes et MLDR

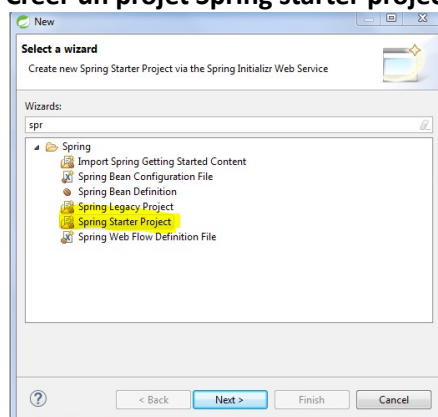


Association **bidirectionnelle** ?


Une association **uni directionnel** ?

On travailler avec **STS eclipse**


Créer un projet Spring starter project



Il se connecte au site de Spring pour récupérer les différents paramètres



New Spring Starter Project



Name

BanqueSI

☒ Use default location

Location

D:\dev-env\workspace\BanqueSI

Browse

Type:

Maven

Packaging:

Jar

Java Version:

1.8

Language:

Java

Group

org.issmi

Artifact

BanqueSI

Version

0.0.1-SNAPSHOT

Description

Demo project for Spring Boot

Package


org.issmi

Working sets

☐ Add project to working sets

Working sets:

Select...

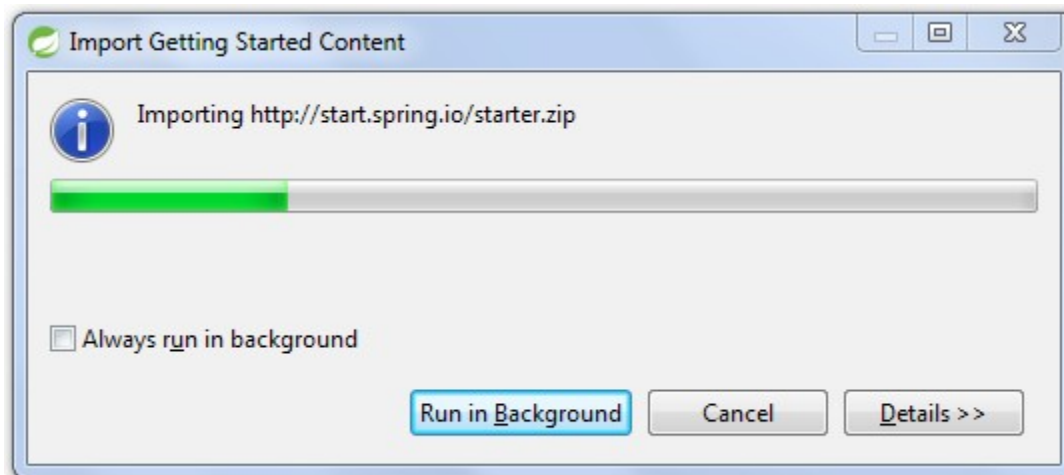
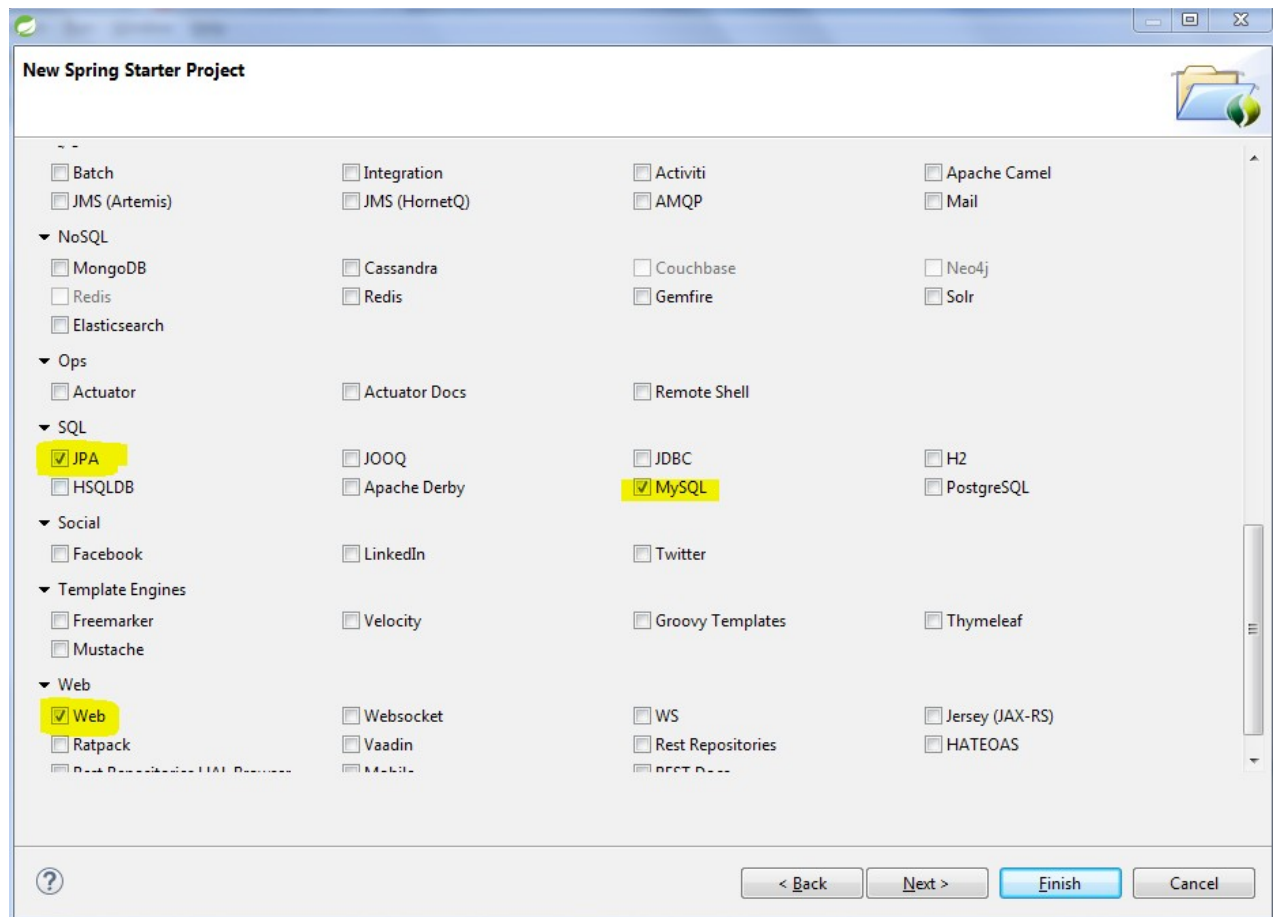


< Back

Next >











Finish

Cancel



Il va charger les **dépendances**

Maintenant on va créer les entités

- ▲  org.issmi.entites
 - ▷  Client.java
 - ▷  Compte.java
 - ▷  CompteCourant.java
 - ▷  CompteEpargne.java
 - ▷  Employe.java
 - ▷  Groupe.java
 - ▷  Operation.java
 - ▷  Retrait.java
 - ▷  Versement.java

```
package org.issmi.entites;

import java.io.Serializable;
import java.util.Collection;

public class Client implements Serializable{

    private Long codeClient;
    private String nomClient;
    private Collection<Compte> comptes;

    public Client() {
        super();
    }

    public Client(String nomClient) {
        super();
        this.nomClient = nomClient;
    }

    // Getteurs et setteurs
```

On ajoute les annotations JPA


```

@Entity
public class Client implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long codeClient;

    private String nomClient;

    @OneToMany(mappedBy="client", fetch=FetchType.LAZY)
    private Collection<Compte> comptes;

    public Client() {
        super();
    }

    public Client(String nomClient) {
        super();
        this.nomClient = nomClient;
    }

    // Getteurs et setteurs

```

la différence entre FetchType.Lazy et FetchType.EAGER ?

Fichier de config (Application.properties):

```

4 spring.datasource.url = jdbc:mysql://localhost:3306/db_banque_si
5
6 # Username and password
7 spring.datasource.username = root
8 spring.datasource.password = p4$$word
9
10 # Pilote
11 spring.datasource.driverClassName = com.mysql.jdbc.Driver
12
13 # Type de la base de données
14 spring.jpa.database = MYSQL
15 # =====
16 # = JPA / HIBERNATE
17 # =====
18
19 # Use spring.jpa.properties.* for Hibernate native properties (the prefix is
20 # stripped before adding them to the entity manager).
21
22 # Show or not log for each sql query
23 # Requete SQL il sont afficher au niveau de la console
24 spring.jpa.show-sql = true
25
26 # Hibernate ddl auto (create, create-drop, update): with "update" the database
27 # schema will be automatically updated accordingly to java entities found in
28 # the project
29 # update : si les tables n'existe pas c'est a lui de le créer
30 spring.jpa.hibernate.ddl-auto = update
31
32 # Naming strategy
33 # Strategie de nomage : Exemple codeClient va se traduire par CODE_CLIENT au niveau de la BD
34 spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
35
36 # Allows Hibernate to generate SQL optimized for a particular DBMS
37 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

```


Si on veut changer le port :

Ajouter :

```
server.port=${port:8080}
```

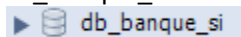
Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can also use placeholders in `application.properties`, e.g.

```
server.port=${port:8080}
```

Création de la base de donnée dans MySQL



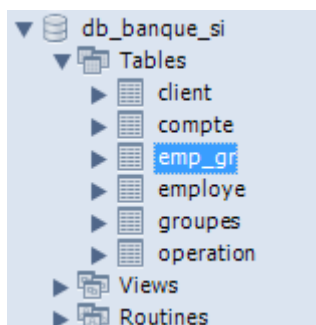
db_banque_si



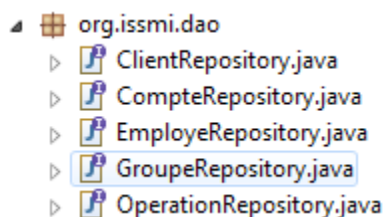
Lancement de l'application avec Spring boot permet de créer directement les tables grâce à notre fichier de configuration :

Application.properties

```
29 # update : si les tables n'existe pas c'est a lui de le créer
30 spring.jpa.hibernate.ddl-auto = update
```



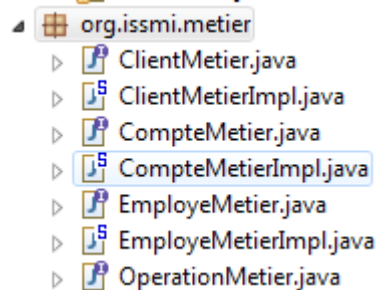
Maintenant on va créer les class DAO :



```
1 package org.issmi.dao;
2
3 import org.issmi.entites.Client;
4
5
6 public interface ClientRepository extends JpaRepository<Client, Long>{
7
8 }
9
```

Maintenant on va créer les class Métiers :

- Implémenter la logique métiers
- Normalement tous les méthodes métiers doivent être **transactionnel**
- c-à-d : soit toutes les opérations interagissant avec la BD fonctionnent et spring effectue le commit soit rien et il fait un rollback : **@Transactional**



Exemple :

```
package org.issmi.metier;

import java.util.List;

public interface ClientMetier {

    public Client saveClient(Client c);
    public List<Client> listeClient();
}

package org.issmi.metier;

import java.util.List;

@Component
@Service
public class ClientMetierImpl implements ClientMetier {

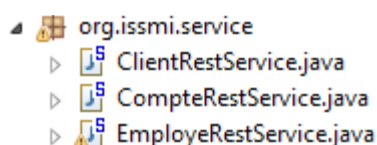
    @Autowired
    private ClientRepository clientRepository;

    @Override
    public Client saveClient(Client c) {
        return clientRepository.save(c);
    }

    @Override
    public List<Client> listeClient() {
        return clientRepository.findAll();
    }
}
```

spring : IOC , injection avec @Autowired

Maintenant on va créer les class Service (Web service Rest Avec Spring) :



```

package org.issmi.service;

import java.util.List;

// Service Restfull
@RestController
public class ClientRestService {

    @Autowired
    private ClientMetier clientMetier;

    @RequestMapping(value="/clients", method=RequestMethod.POST)
    // @ResponseBody //le retour c de JSON c'est implicite car on a utiliser @RestController
    public Client saveClient(@RequestBody Client c) {
        return clientMetier.saveClient(c);
    }

    @RequestMapping(value="/clients", method=RequestMethod.GET)
    public List<Client> listeClient() {
        return clientMetier.listeClient();
    }
}

```

Plugin Advanced REST client pour tester les web service