
	Atelier N°4 (Partie 2)	
---	------------------------------------	---

Objectifs	Temps Alloué	Outis
<ul style="list-style-type: none"> Exemples d'architectures logicielles modernes Mise en œuvre des architectures basées sur le modèle MVC: SpringMVC 	6 heures	<ul style="list-style-type: none"> IDE STS+ JDK 1.8(ou 11) MySQL Server Spring devtools SpringMVC Spring DATA JPA Spring Security Thymleaf

I. Descriptif du travail à faire :

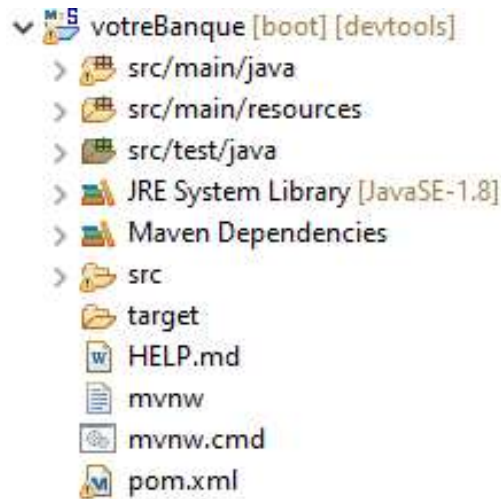
Le but de cet atelier est de créer une application Web JavaEE en utilisant le framework spring.

L'application devrait permettre de :

- Gérer des clients:
 - Ajouter un client
 - Consulter un client
 - Consulter des clients dont le nom contient un mot clé
- Gérer des comptes
 - Ajouter un compte
 - Consulter un compte
- Gérer des opérations
 - Effectuer un virement
 - Effectuer un retrait
 - Effectuer un virement d'un compte vers un autre.
 - Consulter les opérations d'un compte page par page.

On suppose que les données sont stockées dans une base de données MySQL.

L'application s'organise comme suit :



II. Démarche à suivre :

La démarche à suivre pour le développement de cette application se résume en trois étapes :

1. Créer la base de données MySQL nommée **Banque**.
2. Créer un projet SpringStarterProject nommé **BanqueMVC**
3. Développer la couche de données entités+DAO.
4. Développer la couche métier.
5. Développer la couche web.
6. Ajouter la composante sécurité.

Architecture Technique

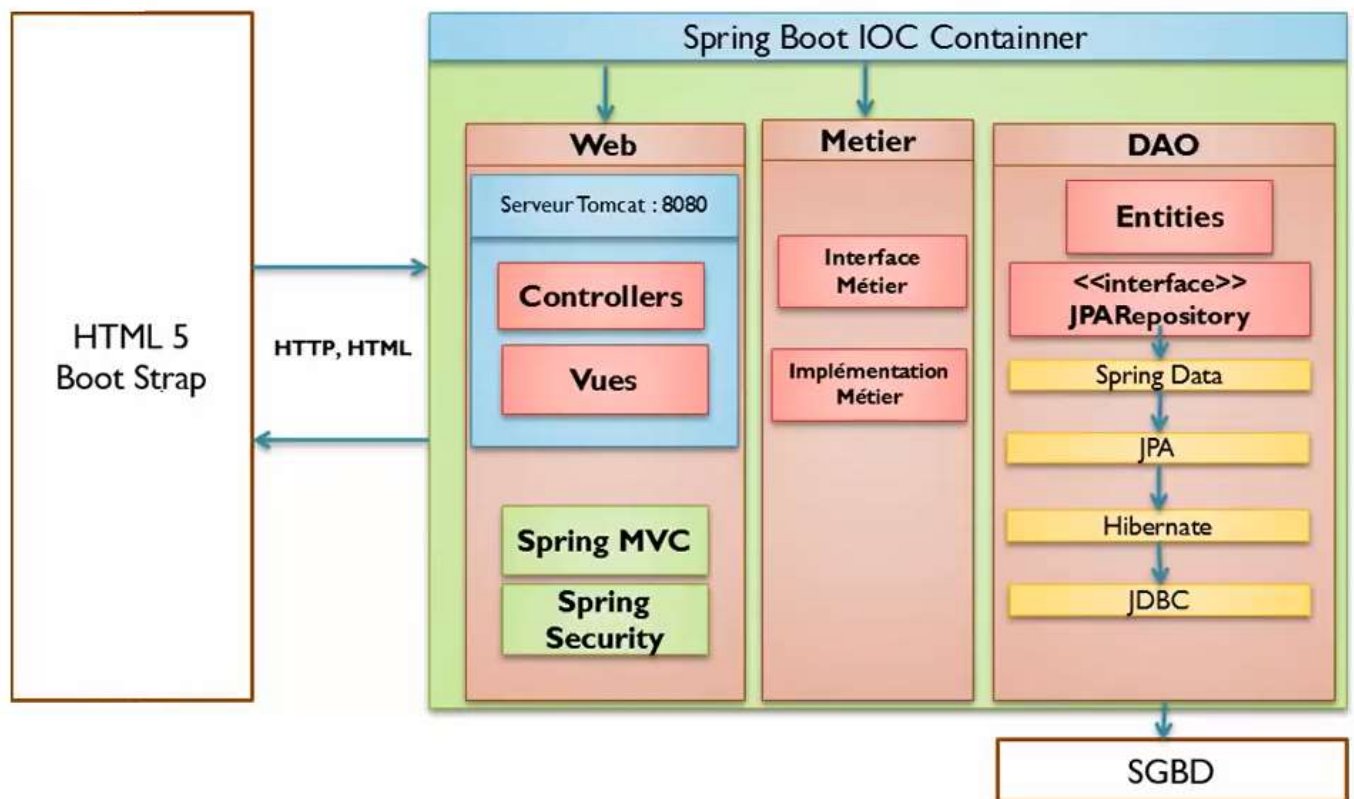


Figure 1 : L'architecture de l'application

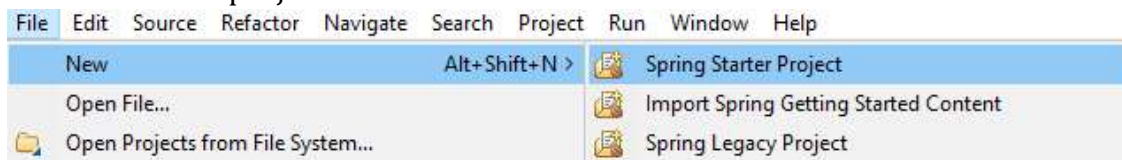
III. Réalisation :

1. La Base de données

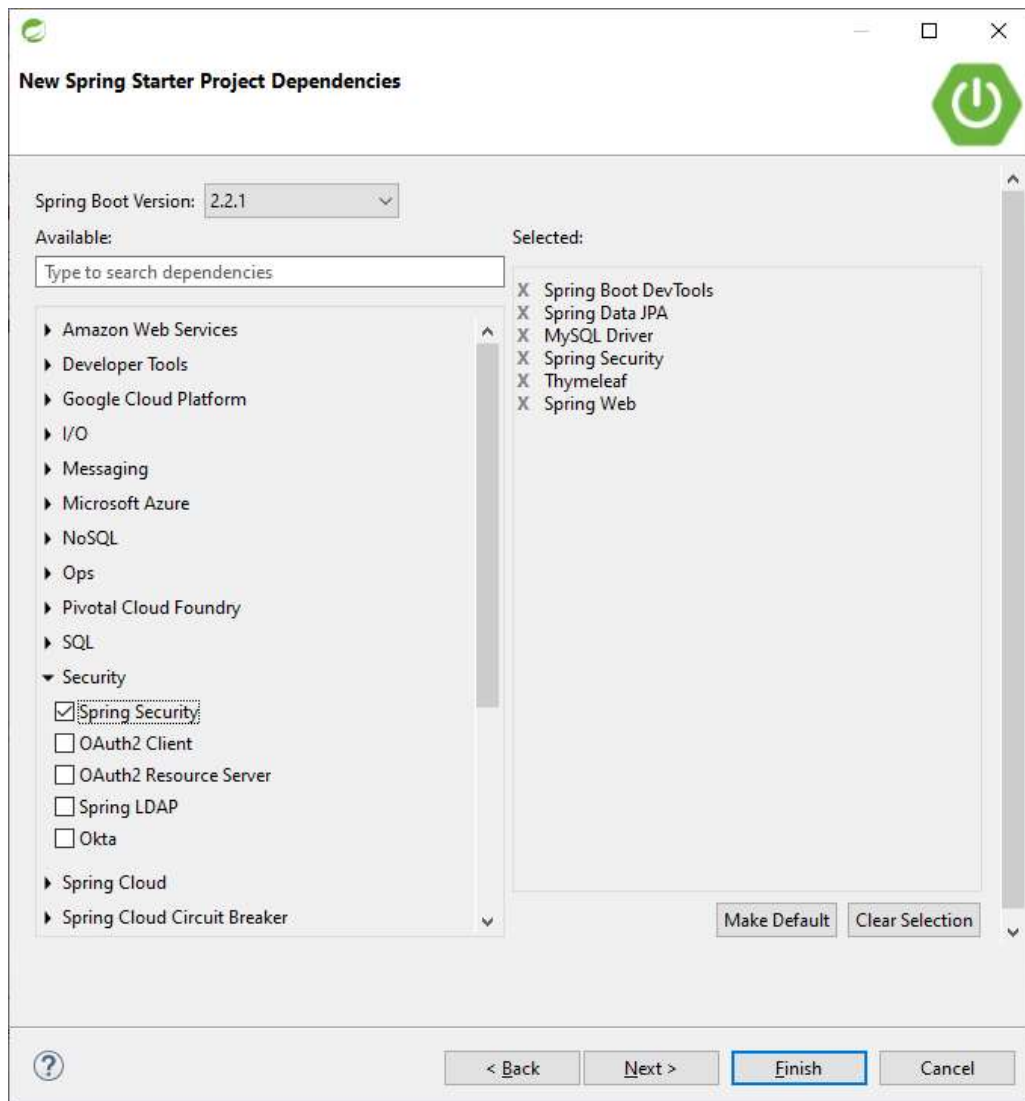
- Utiliser un serveur MySQL pour la création de la BDD.
- Nommer la BDD «**Banque**»

2. Configuration du projet

- Créer un projet

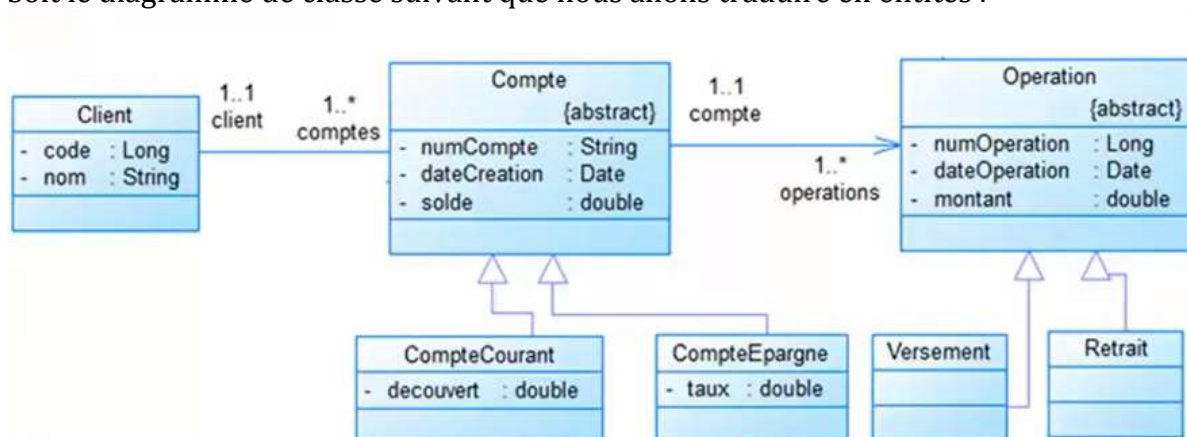


- Configurer les dépendances de ce projet :



3. les Entités

Soit le diagramme de classe suivant que nous allons traduire en entités :



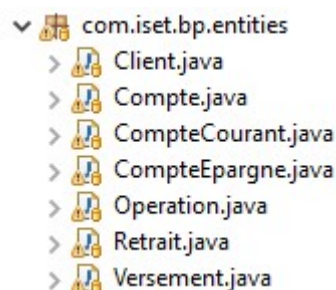
Pour obtenir le schéma de la base de données suivant :

- **MLRD** : En utilisant la stratégie Single Table pour l'héritage
 - **T_CLIENTS** (**CODE_CLI**, **NOM_CLI**)
 - **T_COMPTE** (**NUM_CPTE**, **TYPE_PTE**, **DATE_CR**, **SOLDE**, **DEC**, **TAUX**, **#CODE_CLI**)
 - **T_OPERATIONS** (**NUM_OP**, **TYPE_OP**, **DATE_OP**, **MONTANT**, **#NUM_CPTE**)

Sachant que :

- Chaque compte est défini par un code, 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.
- Chaque client est défini par son code et son nom.
- Chaque compte peut subir plusieurs opérations
- Il existe deux types d'opération ; versement et retrait
- Une opération est définie par un numéro, une date et un montant

Dans src créer un package com.iset.bp.entities



Pour vous aider dans la traduction des relations intertables, consultez cet URL :

Les relations :

https://gayerie.dev/epsi-b3-orm/javaee_orm/jpa_relations.html

L'héritage :

https://gayerie.dev/epsi-b3-orm/javaee_orm/jpa_inheritance.html

4. La couche DAO

Créer un package com.iset.bp.dao et créer les interfaces DAORepository

```
ClientRepository.java
1 package com.iset.bp.dao;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5
6 public interface ClientRepository extends JpaRepository<Client, Long>{
7
8
9 }
```

```

CompteRepository.java
1 package com.iset.bp.dao;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5
6
7 public interface CompteRepository extends JpaRepository<Compte, String> {
8
9 }

OperationRepository.java
1 package com.iset.bp.dao;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5
6 public interface OperationRepository extends JpaRepository<Operation, Long> {
7
8 }

```

- Dans application.properties ajouter les lignes suivantes :

```

#DB
spring.datasource.url=jdbc:mysql://localhost:3306/Banque
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

```

5. L'application principale :

La classe qui définit `SpringBootApplication` doit implémenter l'interface `CommandLineRunner` , dans sa méthode run écrire le code suivant pour tester la communication avec la base de données :

```

@SpringBootApplication
public class BanqueMVCAApplication implements CommandLineRunner{
    @Autowired
    private ClientRepository clientrepository;
    public static void main(String[] args) {
        SpringApplication.run(VotreBanqueApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        clientrepository.save(new Client("Najla", "allouche.najla@gmail.com"));
    }
}

```

6. La couche métier :

Sous le package com.iset.bp.metier et créer l'interfaces

```
public interface IBanqueMetier {  
  
    public Compte getCompte(String codeCompte);  
    public void versement(String codeCompte, double montant );  
    public void retrait(String codeCompte, double montant );  
    public void virement(String codeCompteRetrait,String codeCompteVersement,double montant);  
    public Page<Operation> listOperationsCompte(String codeCompte,int page,int sizePage);  
  
}
```

Sous le même package créer la classe qui implémente cette interface

```
@Service  
@Transactional //soit les opérations s'exécutent correctement soit on annule toutes les  
opérations  
public class BanqueMetierImpl implements IBanqueMetier {  
  
    @Autowired  
    //Repositories  
  
    @Override  
    public Compte getCompte(String codeCompte) {  
        //à compléter  
    }  
  
    @Override  
    public void versement(String codeCompte, double montant) {  
        //à compléter  
    }  
  
    @Override  
    public void retrait(String codeCompte, double montant) {  
  
        Compte compte = getCompte(codeCompte);  
        double facilitesCaisse = 0;  
  
        if (compte instanceof CompteCourant) {  
  
            facilitesCaisse = ((CompteCourant) compte).getDecouvert();  
  
            if ( compte.getSolde()+facilitesCaisse < montant )  
                throw new RuntimeException("Solde insuffisant");  
        }  
  
        Retrait retrait = new Retrait(new Date(), montant,compte);  
        operationRepository.save(retrait);  
        compte.setSolde(compte.getSolde() - montant);  
        compteRepository.save(compte);  
    }  
  
    @Override  
    public void virement(String codeCompteRetrait, String codeCompteVersement, double montant) {  
        //Impossible : pas de virement dans le meme compte  
        //sinon retrait du source puis versement dans la destination  
    }  
}
```



```

@Override
public Page<Operation> listOperationsCompte(String codeCompte, int page, int sizePage) {
    // page: numero de la page
    //sizePage : la taille de la page

    return operationRepository.listOperation(codeCompte, PageRequest.of(page,sizePage,
    Sort.by(("dateOperation")).descending()));
}

}

```

Compléter toutes les méthodes de la classe pour faire appel à la couche DAO

La méthode `listOperationsCompte` de la couche métier fait appel à la méthode `listOperation` dans le repository `IOperationRepository`, cette méthode est définie moyennant le langage HQL(pour définir la requête personnalisée) :

```

//importer org.springframework.data.domain.Pageable;
//importer org.springframework.data.domain.Page;

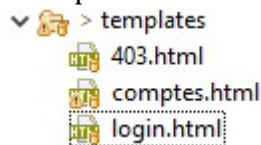
@Query("select o from Operation o where o.compte.codeCompte=:x order by o.dateOperation desc") //
public Page<Operation> listOperation(@Param("x")String codeCompte,Pageable pageable);

```

7. La couche web:

Dans la couche web nous allons utiliser le template `thymeleaf`, pour ce faire nous devons appliquer ces recommandations dans les vues correspondantes :

- L'emplacement de toutes les vues est le dossier templates sous ressources



- Utilisation du langage XHTML (fermer toutes les balises)

a. Tester le lien entre contrôleur et vue:

Créer votre page html nommée comptes:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<!-- Puisque thymeleaf utilise XHTML , il faudra fermer les balises et changer "ISO-8859-1" par "utf-8"/ -->
<title>Test Banque</title>
</head>
<body>
    Test
</body>
</html>

```

Dans le contrôleur ajouter la méthode

```

@Controller
public class BanqueController {
    //la couche web a besoin de la couche metier --> d'où
    @Autowired
    private IBanqueMetier iBanqueMetier;

    @RequestMapping("/comptes") //localhost :8080/comptes affiche la page comptes.html

```



```

    public String index() { //cette méthode retourne une vue tous simplement
        return "comptes"; //càd : le nom de la vue est : comptes.html
    }
}

```

b. Opérations de consultation et de mise à jour du compte:

Compléter la page comptes.html pour afficher le résultat suivant :

- Ajouter `<form th:action="@{/consultercompte}" methode="get">` pour la consultation du compte
- Ajouter `<form th:action="@{/saveOperation}" method="post">` pour les opérations de versement de retrait et de virement

Pour utiliser le template correctement modifier la balise html comme suit

```

<html xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">

```

Consultation d'un compte

Code Compte:

Informations sur le compte

Client: Najla

Code: compte1

Solde : 201111.0

Date Création : 2021-04-29 09:14:45.0

Type de compte: CompteCourant

Découvert: 6000.0

Opérations sur le compte

Compte : compte1

☒ Versement ☐ Retrait ☐ Virement

Montant :

Pour afficher les opérations sur le compte :

```

<table class="table table-striped">
    <tr>
        <th>Numéro</th><th>Type</th><th>Date</th><th>Montant</th>
    </tr>
    <tr th:each="o:${listOperations}">
        <td th:text="${o.numeroOperation}"></td>
    <!--à compléter -->

```

```
</tr>
</table>
```

Liste des opérations

Numéro	Type	Date	Montant
1	Versement	2021-04-29 09:14:45.0	9000.0
2	Versement	2021-04-29 09:14:45.0	6000.0
3	Versement	2021-04-29 09:14:45.0	2300.0
4	Retrait	2021-04-29 09:14:45.0	9000.0
9	Versement	2021-04-29 09:14:45.0	111111.0

Dans le contrôleur ajouter les méthodes

```
@RequestMapping("/consultercompte")
public String consulterCompte(Model model, String
codeCompte,@RequestParam(name="page",defaultValue="0")int
page,@RequestParam(name="size",defaultValue="5")int size) {
//compléter le code
Page<Operation> pageOperations = iBanqueMetier.listOperationsCompte(codeCompte ,page, size);
model.addAttribute("listOperations",pageOperations.getContent());
int[] pages=new int[pageOperations.getTotalPages()];
//ajouter les attributs page et compte.....

return "comptes";
}

@RequestMapping(value="/saveOperation",method=RequestMethod.POST)
public String saveOperation(Model model, String typeOperation, String codeCompte,
double montant, String codeCompte2) {
//compléter le code

return "redirect:/consultercompte?codeCompte="+codeCompte;
}
```