

I - TP – JPA (Java Persistence Api)

1. TP1 / préparation de l'environnement de Tp

Sur un ordinateur personnel (ou bien sur une machine virtuelle préparée par l'organisme de formation), installer si besoin :

- un **jdk** (ex : 17)
- un **IDE java** (exemple : **Eclipse-Jee** en version 2025-09 ou plus récent ou bien **IntelliJ**).
- le SGBDR "**mariadb**" (compatible MySql et facile à installer sur PC-Windows) en mémorisant bien le mot de passe principal (ex : root/root) .

- récupérer (via *git clone* ou via "**code / download zip**") tout le contenu du référentiel
https://github.com/didier-tp/jpa_2025

- charger le projet exemple "**appliJpa**" (basé sur java/maven) dans l'IDE favori (eclipse ou intelliJ)
- lancer le script **appliJpa2025/src/script/mysql_mariadb/init-db.bat** (après éventuels ajustements) de manière à préparer la base de données pour les Tps.
- vérifier le contenu du fichier de configuration
appliJpa2025/src/main/resources/META-INF/persistence.xml
(*jdbc:mysql://localhost:3306/BaseQuiVaBien , root/root*)
- lancer l'exécution de **appliJpa2025/src/main/java/tp/appliJpa/TestSansSpringApp** et résoudre les éventuels problèmes avec l'aide du formateur .

2. TP2 : JPA seul (sans spring)

- En se guidant sur le mode opératoire de l'annexe "**Configuration JPA pour TP**" du **support de cours**, créer (à partir de zéro) un nouveau projet java/maven/jpa/spring_boot intitulé "**myJpa**"
- configurer le fichier **src/main/resources/META-INF/persistence.xml** selon le paragraphe "**Premiers Tps sans SpringBoot (JPA seul)**" de l'annexe "**Configuration JPA pour TP**" du *support de cours*
- en s'inspirant toujours du paragraphe "**Premiers Tps sans SpringBoot (JPA seul)**" de l'annexe "**Configuration JPA pour TP**" du *support de cours*, **coder les éléments complémentaires suivants** :

tp....entity.Employe.java (avec **@Entity** , **@Id** , ...)
tp....dao.DaoEmploye (interface) et **tp....dao.DaoEmployeJpa** (classe) avec **@PersistenceContext** et **EntityManager**
tp.....TestSansSpringApp (avec *main()* et **EntityManagerFactory**)

NB :

- Le code java.JPA devra utiliser partiellement la table **employe** de la base de données (script *appJpa/src/script/init-db.sql*)
- On pourra se contenter de mapper les colonnes essentielles "emp_id" , "firstname" et "lastname"
- On testera au minimum un ajout d'employé et une relecture (dans le *main()* et/ou ...)

3. TP3 : JPA dans un cadre JEE (ex : Spring)

En se guidant sur le mode opératoire du paragraphe "**Config JPA avec SpringBoot et transactions**" de l'annexe "Configuration JPA pour TP" du support de cours :

- configurer correctement le fichier "**src/main/resources/application.properties**"
- effectuer un petit test (en environnement spring + jpa) en utilisant un code de de genre dans *MyJpaApplication.main()*

```
/* ATTENTION: avec Spring moderne (SpringBoot) , la configuration JPA
n'est pas recherchée dans META-INF/persistence.xml
mais plutôt dans application.properties ou application.yml */
ApplicationContext contexteSpring = SpringApplication.run(MyJpaApplication.class, args);

DaoEmploye daoEmploye = contexteSpring.getBean(DaoEmploye.class);

Employe emp1 = new Employe(null,"prenomQuiVabien","NomQuiVaBien");
daoEmploye.insertNew(emp1);
List<Employe> employes = daoEmploye.findAll();
for(Employe emp : employes) {
    System.out.println(emp);
}
```

Coder un test unitaire équivalent (ex : **TestDaoEmploye**) en s'appuyant sur **@SpringBootTest**, **@Autowired** , **@Test**,

4. TP4: avec Dao generic

- **Dupliquer** les fichiers **tp....dao.DaoEmploye** et **tp....dao.DaoEmployeJpa** en **tp....dao.DaoEmployeSansGeneric** et **tp....dao.DaoEmployeJpaSansGeneric**
- Récupérer depuis le projet AppliJpa une copie adaptée des fichiers **tp...repository.RepositoryGeneric** et **tp...repository.RepositoryGenericJpa** (à placer par exemple dans **tp....dao**)
- Recoder les fichiers **tp....dao.DaoEmploye** et **tp....dao.DaoEmployeJpa** en héritant des classes génériques
- Relancer le test unitaire **TestDaoEmploye** pour s'assurer du bon fonctionnement .

5. TP5 : avec requête JPQL simple

- Coder la méthode **findByFirstname()** sur la classe **dao.DaoEmployeJpa** en s'appuyant sur une requête **JPQL**
- Tester cela via une méthode **testFindByFirstName()** à ajouter sur **TestDaoEmploye**
Après avoir insérer 3 employés en base , l'appel à **daoEmploye.findByFirstname("alain")**; ne doit retourner que les 2 employés qui ont le prénom "alain" .

6. TP6 : relation 1-n (@OneToMany)

De manière à partiellement mapper les tables **compte** et **operation** de la base de données (script appJpa/src/script/init-db.sql), coder et tester les différentes classes complémentaires suivantes :

tp....entity.Compte

tp.....entity.Operation (avec @ManyToOne)

tp.....dao.DaoCompteJpa, **tp.....dao.DaoOperationJpa** (*plus interfaces*)

tp.....dao.TestDaoCompte

NB :

- On pourra se baser sur le diagramme UML **appliJpa/src/main/resources/classDiag_p1.png**
- On pourra utiliser **temporairement** @OneToMany(fetch = FetchType.EAGER, mappedBy = "...") au dessus de private List<Operation> operations dans Compte.java
- On pourra ensuite basculer sur FetchType.LAZY et coder/appeler une méthode de recherche spécifique (ex : daoCompte.findWithOperations) .

```
@Test
void testCompteAvecOperations() {
    Compte c1 = new Compte(null,"compteA",50.0);
    doaCompte.insertNew(c1);
    System.out.println("c1.getNumero()=" + c1.getNumero());

    Operation op1C1 = new Operation(null,"achat xxx",-5.6);
    op1C1.setCompte(c1);
    daoOperation.insertNew(op1C1);

    Operation op2C1 = new Operation(null,"achat yyy",-45.6);
    op2C1.setCompte(c1);
    daoOperation.insertNew(op2C1);

    //relire et afficher le compte 1
    //afficher les operations associées au compte 1
    //Compte c1Relu = daoCompte.findById(c1.getNumero());
    Compte c1Relu = daoCompte.findWithOperations(c1.getNumero());
    System.out.println("c1Relu="+c1Relu);
    for(Operation op : c1Relu.getOperations()){
        System.out.println("\t op="+op);
    }

    System.out.println("via query sur Operation:");
    for(Operation op : daoOperation.findByCompteNumero(c1.getNumero())){
        System.out.println("\t op="+op);
    }
}
```

Conseils :

- utiliser le mot clef **fetch** dans une requête introduite via l'annotation @NamedQuery() placée sur le haut de la classe Compte .

- une solution du Tp se trouve dans les projets AppJpa et AppliJpa du référentiel

https://github.com/didier-tp/m2i_jpa_dec2022.git

7. Tp ou demo "avec transaction"

Dans un nouveau package **tp.....service** coder une interface **CompteService** et une classe **CompteServiceImpl** comportant la méthode **effectuerVirement** suivante :

```
@Transactional //ou equivalent EJB
@Service //ou bien @Stateless sur EJB
public class CompteServiceImpl implements CompteService {

    @Autowired //ou bien @Inject sur EJB
    private DaoCompte daoCompte;

    @Autowired //ou bien @Inject sur EJB
    private DaoOperation daoOperation;

    @Override
    public void effectuerVirement(double montant, long numCptDeb, long numCptCred)
        throws RuntimeException {

        Compte compteDeb = daoCompte.findById(numCptDeb);
        compteDeb.setSolde(compteDeb.getSolde()-montant);
        daoCompte.update(compteDeb);//quelquefois automatique si transaction
        Operation opDebit = new Operation(null,"debit suite virement",-montant);
        opDebit.setCompte(compteDeb);
        daoOperation.insertNew(opDebit);

        Compte compteCred = daoCompte.findById(numCptCred);
        compteCred.setSolde(compteCred.getSolde()+montant);
        daoCompte.update(compteCred);//quelquefois automatique si transaction
        Operation opCredit = new Operation(null,"credit suite virement",+montant);
        opCredit.setCompte(compteCred);
        daoOperation.insertNew(opCredit);

    }
}
```

Dans la partie srs/test/java , ajouter la nouvelle classe suivante :

```
@SpringBootTest //à lancer avec Run as ... / JUnit Test
class TestCompteService {

    @Autowired //equivalent de @Inject
    private DaoCompte repositoryCompte;

    @Autowired //equivalent de @Inject
    private CompteService compteService;

    @Test
    void testBonVirement() {
        Compte compteC1 = daoCompte.insertNew(new Compte(null,"compteC1" , 101.0));
    }
}
```

```

        Compte compteC2 = daoCompte.insertNew(new Compte(null,"compteC2" , 202.0));
        System.out.println("avant bon virement: solde c1="+compteC1.getSolde());
        System.out.println("avant bon virement: solde c2="+compteC2.getSolde());
        compteService.effectuerVirement(50.0, compteC1.getNumero(), compteC2.getNumero());
        Compte compteC1ReluApresVirement = daoCompte.findById(compteC1.getNumero());
        Compte compteC2ReluApresVirement = daoCompte.findById(compteC2.getNumero());
        System.out.println("apres bon virement: solde c1="+compteC1ReluApresVirement.getSolde());
        System.out.println("apres bon virement: solde c2="+compteC2ReluApresVirement.getSolde());
        Assertions.assertEquals(compteC1.getSolde() - 50, compteC1ReluApresVirement.getSolde());
        Assertions.assertEquals(compteC2.getSolde() + 50, compteC2ReluApresVirement.getSolde());
    }

    @Test
    void testMauvaisVirement() {
        Compte compteC1 = daoCompte.insertNew(new Compte(null,"compteC1" , 101.0));
        Compte compteC2 = daoCompte.insertNew(new Compte(null,"compteC2" , 202.0));
        System.out.println("avant mauvais virement: solde c1="+compteC1.getSolde());
        System.out.println("avant mauvais virement: solde c2="+compteC2.getSolde());
        try {
            compteService.effectuerVirement(50.0, compteC1.getNumero(), -78);
            //le compte à créditer -78 n'existe pas
        } catch (RuntimeException e) {
            e.printStackTrace();
        }
        Compte compteC1ReluApresVirement = daoCompte.findById(compteC1.getNumero());
        Compte compteC2ReluApresVirement = daoCompte.findById(compteC2.getNumero());
        System.out.println("apres mauvais virement: solde c1="+compteC1ReluApresVirement.getSolde());
        System.out.println("apres mauvais virement: solde c2="+compteC2ReluApresVirement.getSolde());
        Assertions.assertEquals(compteC1.getSolde(), compteC1ReluApresVirement.getSolde());
        Assertions.assertEquals(compteC2.getSolde(), compteC2ReluApresVirement.getSolde());
    }
}

```

Lancer plusieurs fois ce test en retirant et en ajoutant **@Transactional** au dessus de la classe **CompteServiceImpl**

Analyser les comportements différents de `effectuerVirement()` selon l'absence ou la présence de **@Transactional** :

- états "persistant" ou bien "détaché" des entités
- propagation des transactions entre service appelant et sous services (dao/repository)

8. TP8 : Many-to-many entre Client et Compte

De manière à partiellement mapper les tables **compte** , **client** et **client_compte** de la base de données (script `appJpa/src/script/init-db.sql`) , coder et tester les différentes classes complémentaires suivantes :

tp.....entity.Compte (avec **@ManyToMany** et **@JoinTable**)

tp.....entity.Client (avec **@ManyToMany(mappedBy="...")**)

tp.....dao.DaoClientJpa (*plus interfaces*)

NB :

- On pourra se baser sur le diagramme UML `appliJpa/src/main/resources/classDiag_p1.png`

Test à coder/ajuster/adapter et à lancer :

```

@Test
void testClientAvecComptes() {
    Client cli1 = new Client(null,"Condor" , "Olie"); //mappedBy coté client
    daoClient.insertNew(cli1);

    Compte cc1 = new Compte(null,"comptecA",50.0);
    cc1.getClients().add(cli1); //JoinTable coté compte
    daoCompte.insertNew(cc1);

    Compte cc2 = new CompteEpargne(null,"comptecB",70.0 , 2.5);
    cc2.getClients().add(cli1); //JoinTable coté compte
    daoCompte.insertNew(cc2);

    //afficher valeurs relues pour vérifier
    Client cli1Relu=daoClient.findByIdWithComptes(cli1.getId());
    System.out.println("cli1Relu"+cli1Relu);
    for(Compte c : cli1Relu.getComptes()){
        System.out.println("\t" + c.toString());
    }

    //Solution2:
    System.out.println("via repositoryCompte.findById(idClient):");
    for(Compte c : daoCompte.findById(cli1.getId())){
        System.out.println("\t" + c.toString());
    }
}

```

Conseils :

- utiliser si besoin le mot clef **fetch** dans une requête introduite via l'annotation **@NamedQuery()** placée sur le haut de la classe Compte ou Client.
- une solution du Tp se trouve dans les projets AppJpa et AppliJpa du référentiel
https://github.com/didier-tp/m2i_jpa_dec2022.git

9. TP9 : héritage JPA

Ajuster la classe existante **tp.....entity.Compte** et coder la nouvelle classe **tp.....entity.CompteEpargne** héritant de **Compte** en tenant compte de :

- la structure de la table **compte** (script appJpa/src/script/*init-db.sql*)
- utilisant la stratégie "SINGLE_TABLE"

Retoucher la méthode de test **testClientAvecComptes()** de manière à utiliser des instances de **Compte** et **CompteEpargne**.

10. TP10: autres aspects divers de JPA

Selon le temps disponible, on pourra coder et tester l'un des aspects suivants :

- **relation one-to-one** entre **Client** (ou **ClientAvecAdresse**) et **AdresseClient**
- **@Version** dans **ResaAvecVersion**
- **@Lob** sur `private byte[]` image dans **BigData**
- ...

NB :

- On pourra se baser sur les diagrammes UML
appliJpa/src/main/resources/classDiag_p1.png et **classDiag_p2.png**

- une solution du Tp se trouve dans le projet appliJpa2025 du référentiel https://github.com/didier-tp/jpa_2025.git