

2.11 Activité pratique : Hibernate

Introduction

L'objectif de ce TP est de configurer une application Java avec Hibernate et MySQL pour gérer la persistance des données. Chaque étape du TP est détaillée avec des explications pour faciliter la compréhension. La structure finale de ce projet est présenté dans la figure 2.2.

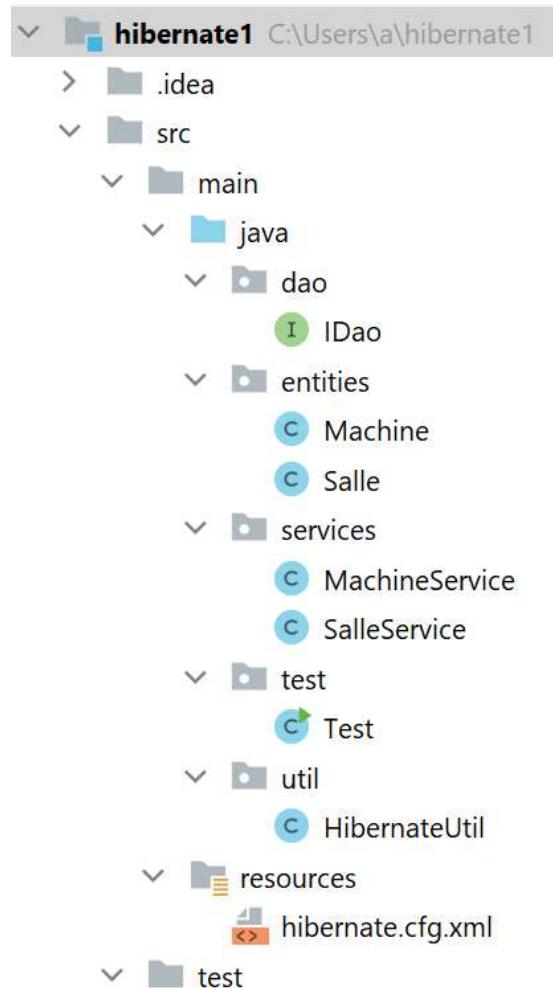


Figure 2.2: Diagramme de classe.

Étape 1: Configuration du fichier pom.xml

La première tâche consiste à configurer les dépendances Maven nécessaires pour le projet.

Listing 2.19: Configuration du fichier pom.xml

```

<dependencies>
    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
    
```

```

<version>5.6.9.Final</version>
</dependency>
<!-- MySQL Connector -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
<!-- JPA API -->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>
<!-- JUnit pour les tests -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>
</dependencies>

```



Les dépendances spécifiées dans ce fichier incluent :

- hibernate-core : Permet l'intégration d'Hibernate pour la gestion de la persistance.
- mysql-connector-java : Permet la connexion à une base de données MySQL.
- javax.persistence-api : Fournit l'API standard JPA pour les annotations.
- junit : Nécessaire pour effectuer des tests unitaires, facultatif.

Étape 2: Création de l'interface DAO

La création d'une interface DAO permet de définir les opérations CRUD de base.

Listing 2.20: Création de l'interface IDao

```

package dao;

import java.util.List;

public interface IDao<T> {
    boolean create(T o);
    boolean delete(T o);
}

```

```

boolean update(T o);
T findById(int id);
List<T> findAll();
}

```

R Cette interface générique définit cinq méthodes principales :

- **create** : Permet de créer et de persister un objet.
- **delete** : Supprime un objet de la base de données.
- **update** : Met à jour un objet existant.
- **findById** : Récupère un objet à partir de son identifiant.
- **findAll** : Récupère tous les objets.

Étape 3: Création des entités JPA

La création des entités permet de définir les classes persistantes à mapper dans la base de données. Le diagramme de classe présenté dans la figure 2.3 illustre les entités Salle et Machine.

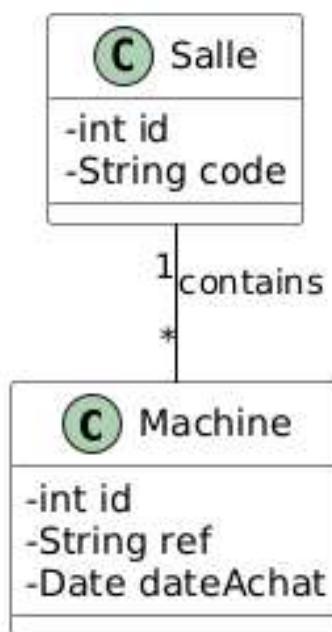


Figure 2.3: Diagramme de classe.

Classe Machine

Listing 2.21: Crédation de la classe Machine

```

package entities;

import java.util.Date;

```

```

import javax.persistence.*;

@Entity
@NamedNativeQuery(name = "findBetweenDateNative", query = "select *
    from machine where dateAchat between :d1 and :d2", resultClass
    = Machine.class)
@NamedQuery(name = "findBetweenDate", query = "from Machine where
    dateAchat between :d1 and :d2")
public class Machine {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String ref;

    @Temporal(TemporalType.DATE)
    private Date dateAchat;

    @ManyToOne
    private Salle salle;

    // Constructeurs, getters et setters
}

```



La classe Machine est une entité persistante configurée avec plusieurs annotations JPA :

- `@Entity` : Indique que cette classe est une entité persistante.
- `@Id` : Spécifie la clé primaire de l'entité.
- `@GeneratedValue` : Permet l'auto-génération de l'identifiant.
- `@Temporal` : Spécifie le type de la date (`TemporalType.DATE`).
- `@ManyToOne` : Définit une relation avec une autre entité, ici Salle.
- `@NamedNativeQuery` et `@NamedQuery` : Permettent de définir des requêtes nommées pour effectuer des recherches spécifiques dans la base de données.

Classe Salle

Listing 2.22: Crédation de la classe Salle

```

package entities;

import java.util.List;
import javax.persistence.*;

```

```

@Entity
@Table(name = "salles")
public class Salle {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String code;

    @OneToMany(mappedBy = "salle", fetch = FetchType.EAGER)
    private List<Machine> machines;

    // Constructeurs, getters et setters
}

```



La classe Salle définit une relation avec plusieurs machines :

- `@OneToMany` : Indique qu'une salle peut contenir plusieurs machines.
- Le `fetch` est défini à `FetchType.EAGER`, ce qui signifie que les machines sont récupérées immédiatement avec la salle.

Étape 4: Configuration de Hibernate

Le fichier de configuration Hibernate permet de spécifier les paramètres de connexion à la base de données MySQL.

Listing 2.23: Configuration de Hibernate dans 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.dialect">
            org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/base?zeroDateTimeBehavior=convertToNull
        </property>
        <property name="hibernate.connection.username">root</property>
        <property
            name="hibernate.connection.password">votre_mot_de_passe
        </property>
        <property name="hibernate.show_sql">true</property>
    </session-factory>
</hibernate-configuration>

```

```

<property name="hibernate.format_sql">true</property>
<property name="hibernate.hbm2ddl.auto">update</property>
<mapping class="entities.Salle"/>
<mapping class="entities.Machine"/>
</session-factory>
</hibernate-configuration>

```

R Le fichier hibernate.cfg.xml configure la session Hibernate avec les propriétés suivantes :

- hibernate.dialect : Spécifie le dialecte SQL utilisé, ici MySQL.
- hibernate.connection.driver_class : Détermine le driver JDBC à utiliser.
- hibernate.connection.url, hibernate.connection.username, hibernate.connection.password : Spécifient l'URL, l'utilisateur et le mot de passe pour la connexion à la base de données.
- hibernate.show_sql : Active l'affichage des requêtes SQL générées.
- hibernate.hbm2ddl.auto : Défini à update pour mettre à jour automatiquement la structure de la base de données.
- mapping : Indique les classes d'entités à mapper.

Étape 5: Création d'une classe utilitaire pour Hibernate

La classe utilitaire suivante permet de gérer la création de la SessionFactory pour interagir avec la base de données.

Listing 2.24: Classe HibernateUtil

```

package util;

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

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Crée la SessionFactory à partir du fichier de configuration
            // standard (hibernate.cfg.xml)
            sessionFactory = new
                Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log l'exception pour le débogage

```

```

        System.err.println("Échec de la création de SessionFactory."
            + ex);
        throw new ExceptionInInitializerError(ex);
    }
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}

```

- R** La classe `HibernateUtil` gère la création et l'obtention de la `SessionFactory` qui est essentielle pour interagir avec la base de données.

- Le bloc statique initialise la `SessionFactory` lors du chargement de la classe.
- En cas d'échec lors de la création de la `SessionFactory`, une erreur est loguée et une exception est lancée pour arrêter l'application.
- La méthode `getSessionFactory` permet d'obtenir l'instance de `SessionFactory` partout dans l'application.

Étape 6: Création de la couche Service

La couche service implémente les opérations définies dans l'interface DAO pour chaque entité. Commençons par `SalleService`.

Classe `SalleService`

Listing 2.25: Classe SalleService

```

package services;

import dao.IDao;
import entities.Salle;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import util.HibernateUtil;

import java.util.List;

public class SalleService implements IDao<Salle> {

    @Override
    public boolean create(Salle o) {
        Session session = null;

```

```
Transaction tx = null;
boolean etat = false;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    tx = session.beginTransaction();
    session.save(o);
    tx.commit();
    etat = true;
} catch (HibernateException e) {
    if(tx != null)
        tx.rollback();
    e.printStackTrace();
} finally {
    if(session != null)
        session.close();
}
return etat;
}

@Override
public boolean delete(Salle o) {
    Session session = null;
    Transaction tx = null;
    boolean etat = false;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.delete(o);
        tx.commit();
        etat = true;
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return etat;
}

@Override
public boolean update(Salle o) {
    Session session = null;
    Transaction tx = null;
```

```
boolean etat = false;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    tx = session.beginTransaction();
    session.update(o);
    tx.commit();
    etat = true;
} catch (HibernateException e) {
    if(tx != null)
        tx.rollback();
    e.printStackTrace();
} finally {
    if(session != null)
        session.close();
}
return etat;
}

@Override
public Salle findById(int id) {
    Session session = null;
    Transaction tx = null;
    Salle salle = null;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        salle = session.get(Salle.class, id);
        tx.commit();
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return salle;
}

@Override
public List<Salle> findAll() {
    Session session = null;
    Transaction tx = null;
    List<Salle> salles = null;
    try {
```

```
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        salles = session.createQuery("from Salle",
            Salle.class).list();
        tx.commit();
    } catch (HibernateException e) {
        if(tx != null)
        tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
        session.close();
    }
    return salles;
}
```

 La classe SalleService implémente l’interface IDao pour gérer les opérations CRUD sur l’entité Salle :

- Chaque méthode ouvre une session Hibernate, commence une transaction, effectue l’opération souhaitée, puis valide la transaction.
- En cas d’exception HibernateException, la transaction est annulée (rollback) et l’exception est loguée.
- La méthode findAll utilise une requête HQL pour récupérer toutes les instances de Salle.

Classe MachineService

Listing 2.26: Classe MachineService

```
package services;

import dao.IDao;
import entities.Machine;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import util.HibernateUtil;

import java.util.Date;
import java.util.List;

public class MachineService implements IDao<Machine> {
```

```
@Override
public boolean create(Machine o) {
    Session session = null;
    Transaction tx = null;
    boolean etat = false;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.save(o);
        tx.commit();
        etat = true;
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return etat;
}

@Override
public boolean delete(Machine o) {
    Session session = null;
    Transaction tx = null;
    boolean etat = false;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.delete(o);
        tx.commit();
        etat = true;
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return etat;
}

@Override
```

```
public boolean update(Machine o) {
    Session session = null;
    Transaction tx = null;
    boolean etat = false;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.update(o);
        tx.commit();
        etat = true;
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return etat;
}

@Override
public Machine findById(int id) {
    Session session = null;
    Transaction tx = null;
    Machine machine = null;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        machine = session.get(Machine.class, id);
        tx.commit();
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return machine;
}

@Override
public List<Machine> findAll() {
    Session session = null;
```

```

Transaction tx = null;
List<Machine> machines = null;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    tx = session.beginTransaction();
    machines = session.createQuery("from Machine",
        Machine.class).list();
    tx.commit();
} catch (HibernateException e) {
    if(tx != null)
        tx.rollback();
    e.printStackTrace();
} finally {
    if(session != null)
        session.close();
}
return machines;
}

public List<Machine> findBetweenDate(Date d1, Date d2) {
    Session session = null;
    Transaction tx = null;
    List<Machine> machines = null;
    try {
        session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        machines = session.getNamedQuery("findBetweenDate")
            .setParameter("d1", d1)
            .setParameter("d2", d2)
            .list();
        tx.commit();
    } catch (HibernateException e) {
        if(tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        if(session != null)
            session.close();
    }
    return machines;
}

```



La classe MachineService suit une structure similaire à SalleService pour

gérer les opérations CRUD sur l'entité Machine :

- En plus des méthodes CRUD, une méthode supplémentaire `findBetweenDate` est définie pour récupérer les machines dont la date d'achat est comprise entre deux dates spécifiques.
- Cette méthode utilise une requête nommée (NamedQuery) définie dans l'entité Machine.

Étape 7: Classe de test pour générer la base de données

La classe de test permet de vérifier le bon fonctionnement des services et de la persistance des entités dans la base de données.

Listing 2.27: Classe de test

```
package test;

import entities.Machine;
import entities.Salle;
import services.MachineService;
import services.SalleService;

import java.util.Date;

public class Test {

    public static void main(String[] args) {
        SalleService salleService = new SalleService();
        MachineService machineService = new MachineService();

        // Création et insertion de salles
        Salle salle1 = new Salle("A1");
        Salle salle2 = new Salle("B2");
        salleService.create(salle1);
        salleService.create(salle2);

        // Création et insertion de machines
        Machine machine1 = new Machine("M123", new Date(),
            salleService.findById(1));
        Machine machine2 = new Machine("M124", new Date(),
            salleService.findById(2));
        machineService.create(machine1);
        machineService.create(machine2);

        // Affichage des salles et leurs machines
        for(Salle salle :salleService.findAll()) {
            System.out.println("Salle: " + salle.getCode());
        }
    }
}
```

```

        for(Machine machine :salle.getMachines()) {
            System.out.println(" Machine: " + machine.getRef());
        }
    }

    // Utilisation de la méthode findBetweenDate
    Date d1 = new Date(110, 0, 1); // 1er janvier 2010
    Date d2 = new Date(); // Date actuelle
    System.out.println("Machines achetées entre " + d1 + " et " +
        d2 + ":");
    for(Machine m :machineService.findBetweenDate(d1, d2)) {
        System.out.println(m.getRef() + " achetée le " +
            m.getDateAchat());
    }
}
}

```



La classe Test effectue les opérations suivantes :

- Crée deux instances de Salle et les persiste dans la base de données.
- Crée deux instances de Machine, les associe aux salles précédemment créées et les persiste.
- Récupère toutes les salles et affiche leurs codes ainsi que les références des machines associées.
- Utilise la méthode `findBetweenDate` pour récupérer et afficher les machines achetées entre deux dates spécifiques.

Cette classe permet de vérifier que la configuration d’Hibernate est correcte et que les entités sont correctement mappées et persistées.

Étape 8: Mise en place des tests unitaires avec JUnit

Dans cette étape, l’objectif est de valider les opérations CRUD (Create, Read, Update, Delete) des services associés aux entités Salle et Machine. Pour cela, JUnit sera utilisé pour tester les méthodes des services.

Tests pour l’entité Salle

Voici un exemple de tests unitaires pour la classe `SalleService`. Ce test couvre les méthodes `create`, `findById`, `update`, `delete`, et `findAll`.

Listing 2.28: Test pour la classe `SalleService`

```

import entities.Salle;
import org.junit.After;
import org.junit.Before;

```

```
import org.junit.Test;
import services.SalleService;

import java.util.List;

import static org.junit.Assert.*;

public class SalleServiceTest {

    private SalleService salleService;
    private Salle salle;

    @Before
    public void setUp() {
        salleService = new SalleService();
        salle = new Salle();
        salle.setCode("A101"); // Exemple de code pour la salle

        // Créer et persister la salle avant chaque test
        salleService.create(salle);
    }

    @After
    public void tearDown() {
        // Supprimer la salle après chaque test si elle existe
        Salle foundSalle = salleService.findById(salle.getId());
        if (foundSalle != null) {
            salleService.delete(foundSalle);
        }
    }

    @Test
    public void testCreate() {
        assertNotNull("Salle should have been created with an ID",
                     salle.getId());
    }

    @Test
    public void testFindById() {
        Salle foundSalle = salleService.findById(salle.getId());
        assertNotNull("Salle should be found", foundSalle);
        assertEquals("Found salle should match", salle.getCode(),
                     foundSalle.getCode());
    }
}
```

```

@Test
public void testUpdate() {
    salle.setCode("B202"); // Modifiez le code pour tester la mise à
                           // jour
    boolean result = salleService.update(salle);
    assertTrue("Salle should be updated successfully", result);

    Salle updatedSalle = salleService.findById(salle.getId());
    assertEquals("Updated salle code should match", "B202",
                 updatedSalle.getCode());
}

@Test
public void testDelete() {
    boolean result = salleService.delete(salle);
    assertTrue("Salle should be deleted successfully", result);

    Salle foundSalle = salleService.findById(salle.getId());
    assertNull("Salle should not be found after deletion",
               foundSalle);
}

@Test
public void testFindAll() {
    List<Salle> salles = salleService.findAll();
    assertNotNull("Salles list should not be null", salles);
    assertTrue("Salles list should contain at least one salle",
               salles.size() > 0);
}
}

```



Ce test couvre plusieurs aspects importants :

- **setUp()**: Cette méthode est exécutée avant chaque test. Elle initialise une instance de SalleService et persiste un objet Salle dans la base de données pour que les tests puissent l'utiliser.
- **tearDown()**: Cette méthode est exécutée après chaque test. Elle supprime l'objet Salle de la base pour éviter les conflits avec les autres tests.
- **testCreate()**: Vérifie que l'objet Salle a été créé et que son identifiant est non nul.
- **testFindById()**: Vérifie que l'objet Salle peut être récupéré par son identifiant et que ses attributs sont corrects.
- **testUpdate()**: Modifie un attribut de l'objet Salle et vérifie que la mise à jour a réussi.
- **testDelete()**: Supprime l'objet Salle et vérifie qu'il a été supprimé.

- **testFindAll()**: Vérifie que la liste des Salle n'est pas vide après l'ajout d'une salle.

Tests pour l'entité Machine

Les tests suivants couvrent les mêmes opérations que pour l'entité Salle, mais appliquées à MachineService.

Listing 2.29: Test pour la classe MachineService

```
import entities.Machine;
import entities.Salle;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import services.MachineService;
import services.SalleService;

import java.util.Date;
import java.util.List;

import static org.junit.Assert.*;

public class MachineServiceTest {

    private MachineService machineService;
    private Machine machine;
    private Salle salle;
    private SalleService salleService;

    @Before
    public void setUp() {
        machineService = new MachineService();
        salleService = new SalleService();
        salle = new Salle("A101"); // Exemple de salle

        // Persister la salle avant d'utiliser
        salleService.create(salle);

        machine = new Machine();
        machine.setRef("MACH-001");
        machine.setDateAchat(new Date());
        machine.setSalle(salle);

        // Créer et persister la machine avant chaque test
        machineService.create(machine);
```

```

}

@After
public void tearDown() {
    // Supprimer la machine après chaque test si elle existe
    Machine foundMachine = machineService.findById(machine.getId());
    if (foundMachine != null) {
        machineService.delete(foundMachine);
    }

    // Supprimer la salle après chaque test si elle existe
    Salle foundSalle = salleService.findById(salle.getId());
    if (foundSalle != null) {
        salleService.delete(foundSalle);
    }
}

@Test
public void testCreate() {
    assertNotNull("Machine should have been created with an ID",
        machine.getId());
}

@Test
public void testFindById() {
    Machine foundMachine = machineService.findById(machine.getId());
    assertNotNull("Machine should be found", foundMachine);
    assertEquals("Found machine should match", machine.getRef(),
        foundMachine.getRef());
}

@Test
public void testUpdate() {
    machine.setRef("MACH-002"); // Modifiez la référence pour
    // tester la mise à jour
    boolean result = machineService.update(machine);
    assertTrue("Machine should be updated successfully", result);

    Machine updatedMachine =
        machineService.findById(machine.getId());
    assertEquals("Updated machine ref should match", "MACH-002",
        updatedMachine.getRef());
}

@Test

```

```
public void testDelete() {
    boolean result = machineService.delete(machine);
    assertTrue("Machine should be deleted successfully", result);

    Machine foundMachine = machineService.findById(machine.getId());
    assertNull("Machine should not be found after deletion",
               foundMachine);
}

@Test
public void testFindBetweenDate() {
    List<Machine> machines = machineService.findBetweenDate(
        new Date(System.currentTimeMillis() - 86400000), // Hier
        new Date() // Aujourd'hui
    );
    assertNotNull("Machines list should not be null", machines);
    assertTrue("Machines list should contain at least one machine",
               machines.size() > 0);
}
}
```



Comme pour SalleService, ces tests couvrent :

- **setUp()**: Initialise les instances de MachineService, SalleService et persiste une salle et une machine avant chaque test.
- **tearDown()**: Supprime la machine et la salle après chaque test.
- **testCreate()**: Vérifie la création de l'objet Machine.
- **testFindById()**: Vérifie la récupération de la machine par son identifiant.
- **testUpdate()**: Modifie la référence de la machine et vérifie que la mise à jour est correcte.
- **testDelete()**: Supprime la machine et vérifie qu'elle n'existe plus dans la base de données.
- **testFindBetweenDate()**: Vérifie que la méthode findBetweenDate récupère les machines achetées dans une période de temps donnée. Ce test utilise une date d'achat fixée pour la machine et vérifie que la machine est présente dans les résultats renvoyés par la requête.



À la fin de ce TP, une application Java utilisant Hibernate pour gérer des entités JPA avec une base de données MySQL est mise en place. Les étapes ont couvert la configuration des dépendances Maven, la création des entités et des services, ainsi que la réalisation de tests pour valider le bon fonctionnement de la persistance des données.