Tècniques d'accés a dades

Raúl Velaz Mayo

Índex

Introducció

Tot i que la finalitat i la diversitat d'aplicacions que existeixen és difícilment classificable, sí que es pot dir que totes tenen en comú uns components bàsics. Una lògica de negoci que determinarà la funció del programari, les connexions amb sistemes externs, una sèrie d'interfícies d'usuari i les dades que han de ser guardades en bases de dades amb la capacitat de ser modificades. La diversitat d'aplicacions està relacionada amb la diversitat de bases de dades (BD) que existeixen, i en funció de l'aplicació que esteu construint haureu d'escollir una base de dades o una altra. Això podria presentar un problema si per a cada tipus de base de dades haguéssiu d'escriure un codi diferent. Imagineu, per exemple, que dissenyeu una aplicació que utilitza MySQL com a base de dades però que uns mesos després canvien els requeriments i decidiu emprar Postgresql. Idealment voldríeu fer la transició a la nova base de dades amb els mínims canvis possibles, és a dir, el que voleu és que la base de dades estigui desacoblada de l'aplicació. Per aconseguir aquesta independència existeix JDBC (Java DataBase Connectivity), que és una API (Application Programming Interface) que defineix com els clients (aplicacions Java) accedeixen a les bases de dades. Proporciona una interfície de programació comú que permet que les aplicacions Java accedeixin a les bases dades sense preocupar-se dels detalls particulars de cada BD.

Explicarem com connectar les nostres aplicacions a bases de dades mitjançant JDBC i farem tests per comprovar la connexió i les diferents consultes i actualitzacions de les dades.

Continuarem amb la utilització de Java Persistence API (JPA), una interfície de Java que ens permetrà tenir un lleguatge de consulta independent de la base de dades. Primer farem servir la implementació de JPA EclipseLink directament amb Java Enterprise Edition, configurant la connexió sobre el servidor d'aplicacions Glassfish.

Finalment, farem servir la implementació de JPA Hibernate amb el *framework* Spring. Per una banda, Hibernate ens facilitarà la connexió a la base de dades, i Spring ens ajudarà en el disseny de l'aplicació.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions d'accés a magatzems de dades, aplicant mesures per mantenir la seguretat i la integritat de la informació.

- Analitza les tecnologies que permeten l'accés mitjançant programació a la informació disponible en magatzems de dades.
- Crea aplicacions que estableixin connexions amb bases de dades.
- Recupera informació emmagatzemada en bases de dades.
- Publica en aplicacions web la informació recuperada.
- Utilitza conjunts de dades per emmagatzemar la informació.
- Crea aplicacions web que permetin l'actualització i l'eliminació d'informació disponible en una base de dades.
- Utilitza transaccions per mantenir la consistència de la informació.
- Prova i documenta les aplicacions.

1. Accés a dades amb JDBC

Comencem a desenvolupar un projecte que anomenarem "SocIoc" (SOCial IOC) i que consisteix en una xarxa social de preguntes i respostes per a estudiants de l'IOC. La idea està basada en stackoverflow.com, una xarxa social on els desenvolupadors de programari poden fer preguntes de desenvolupament que són contestades per altres desenvolupadors. La funcionalitat és simple: els estudiants de l'IOC poden fer i contestar preguntes i votar positiva o negativament les respostes a cada pregunta. Els estudiants que rebin vots positius a les respostes aniran guanyant punts.

En aquesta unitat posem les bases de l'aprenentatge configurant una aplicació Java per establir connexions amb la BD, executar consultes SQL (Structured Query Language) i tancar la connexió. Explicarem:

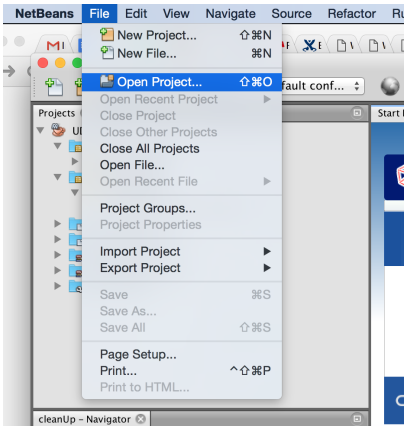
- Com fer connexions a la base de dades H2 utilitzant *drivers* JDBC.
- Com escriure tests unitaris per comprovar que el codi funciona correctament.
- Com fer operacions de lectura, escriptura, actualització i esborrat de base de dades.
- Com preveure injeccions malicioses de codi SQL quan fem consultes a les bases de dades.

1.1 Importar un projecte de Maven a Netbeans

El projecte amb el qual treballareu el podeu trobar a l'apartat d'annexos de la unitat. Un cop descarregat el fitxer i descomprimit ja el podeu importar.

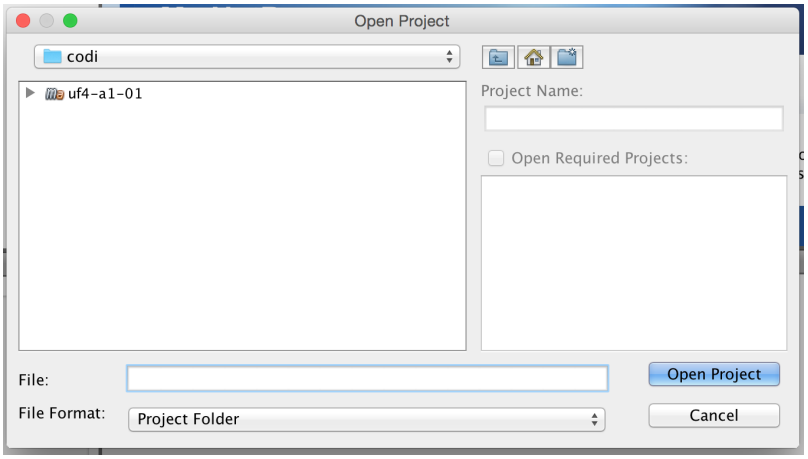
En la figura 1.1 es mostra on és l'opció per obrir un projecte existent.

FIGURA 1.1. Obrir un projecte



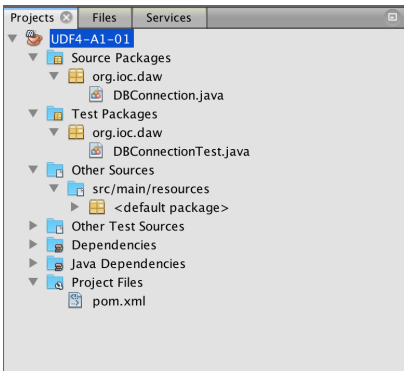
Navegueu per les carpetes fins on teniu la carpeta amb el projecte. Com podeu veure en la figura 1.2, Netbeans reconeixerà que és un projecte Maven.

FIGURA 1.2. Seleccionar el projecte



Finalment, un cop obert el projecte, Maven començarà a descarregar les dependències de JUnit i el *driver* d'H2. Hauríeu de veure un projecte amb l'estructura que es mostra en la figura 1.3.

FIGURA 1.3. Estructura del projecte



1.2 La primera connexió a una base de dades

Apache Maven permet obtenir les llibreries necessàries per al projecte que veureu a continuació, on simplement fareu una connexió a la BD en memòria H2.

En el fitxer pom.xml de Maven, les úniques dependències que us fan falta són la del *driver* de la BD H2 i de JUnit, que és un conjunt de llibreries que utilitzareu per escriure els tests unitaris.

```
<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.190</version>
    </dependency>
    <!-- testing -->
    <dependency>
```

```
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

Un cop Maven descarrega les dependències i les afegeix al *classpath*, ja podeu escriure la primera classe, que simplement establirà una connexió amb una base de dades.

```
public Connection getConnection() {
    Connection con = null;
    try {
        Class.forName("org.h2.Driver");
        con = DriverManager.getConnection("jdbc:h2:mem:socioc_db", "usuari" , "
            passwd");
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
    return con;
}
```

Cada *driver* JDBC té una classe que s'encarrega d'inicialitzar el *driver* quan es carrega a memòria. En el cas d'una BD H2:

```
Class.forName("org.h2.Driver");
```

Carregar el *driver* en memòria no és necessari, ja que des de la versió 6 de Java es fa automàticament. Això implica que com que estem utilitzant una versió de Java superioir a la 6, no fa falta que carreguem el *driver* explícitament. Quan, més tard, refactoritzarem el codi, podrem eliminar *Class.forName("org.h2.Driver");*.

Un cop el *driver* està carregat en la memòria es pot procedir a connectar amb la BD. Per obrir una connexió amb la BD s'utilitza la classe `java.sql.DriverManager`.

```
con = DriverManager.getConnection("jdbc:h2:mem:socioc_db");
```

Fixeu-vos en el paràmetre que s'ha utilitzat per establir la connexió:

```
jdbc:h2:mem:socioc_db
```

Intenteu contestar a les següents preguntes:

- Quina és l'estructura d'aquest paràmetre de connexió?
- Què representa el paràmetre `socioc_db`?

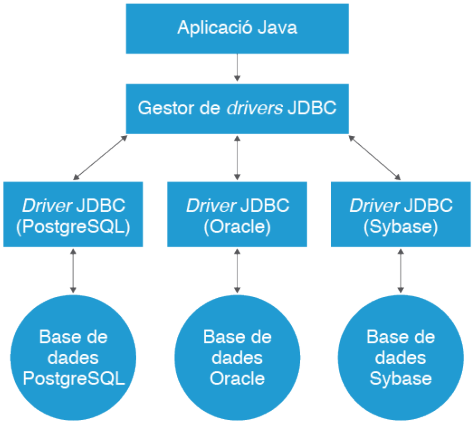
La resposta està relacionada amb el disseny de JDBC i amb la seva arquitectura, que té tres components principals que podeu veure en la figura 1.4:

- Gestor de *drivers* (`DriverManager`): és el responsable de trobar el *driver* que l'aplicació necessita. Quan es sol·licita una connexió amb la BD es fa mitjançant un URL (Uniform Resource Locator), que descriu com ha de ser la connexió.

Algunes de les bases de dades compatibles amb JDBC són: DB2, H2, Informix, JavaDB/Derby, Microsoft SQL Server, Mimer SQL, MySQL, NuoDB, Oracle, PostgreSQL, SQLite, Sybase i Vertica.

- Driver JDBC: cada *driver* present al sistema es carrega en la màquina virtual de Java (JVM Java Virtual Machine) i es registra amb el `DriverManager`. Quan una aplicació sol·licita una connexió amb una BD, el `DriverManager` s'encarrega de preguntar a cada *driver* present si pot connectar amb la BD amb l'URL de connexió especificat.
- Base de dades compatibles amb JDBC.

FIGURA 1.4. Arquitectura JDBC



L'URL de connexió amb una BD JDBC té el següent format:

```
1 jdbc:Tipus_de_base_de_dades://<Host>:<Port>/<Base_de_dades>
```

Per tant, la URL de connexió *jdbc:h2:mem:socioc_db* indica el següent:

- jdbc: indica que s'utilitzarà JDBC per fer la connexió.
- h2: és el tipus de servidor de BD amb el qual volem connectar.
- mem: en aquest cas, com que és una BD en memòria no s'ha d'especificar ni la IP ni el port on està corrent la BD.
- socioc_db: és el nom de l'esquema que s'utilitzarà.

1.3 Millorant el codi: fitxers de propietats i tests unitaris

Hi ha dues coses al codi de l'apartat 1.2 que podeu millorar. La primera és que no és necessari carregar el *driver* explícitament. La segona és que mai és una bona idea posar l'URL de connexió amb la BD directament a una classe. La gràcia d'utilitzar JDBC és que l'aplicació serà independent de la BD que utilitzem; si posem el valor de l'URL de connexió estarem trencant això, ja que en canviar de BD haurem de modificar la classe. Això es pot evitar creant un fitxer de propietats on es configurin els paràmetres de connexió amb la BD. El format del fitxer és PROPIETAT=VALOR, com podeu veure:

```
DB_DRIVER_CLASS=org.h2.Driver
DB_URL=jdbc:h2:mem:socioc_db
DB_USERNAME=usuari
DB_PASSWORD=passwd
```

El fitxer de propietats s'anomena db.properties i es troba a *src/main/resources*. Un cop el fitxer de propietats està definit ja el podeu utilitzar per llegir les propietats de configuració.

```
public Connection getConnection() throws SQLException, IOException {
    Properties props = new Properties();
    InputStream resourceAsStream = null;
    Connection con = null;
    try {
        ClassLoader classLoader = getClass().getClassLoader();
        URL urlResource = classLoader.getResource("db.properties");
        if(urlResource != null){
            resourceAsStream  = urlResource.openStream();
            props.load(resourceAsStream);
            con = DriverManager.getConnection(props.getProperty("DB_URL"),
            props.getProperty("DB_USERNAME"),
            props.getProperty("DB_PASSWORD"));
        }
    } catch (IOException | ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    } finally {
        if (resourceAsStream != null) {
            resourceAsStream.close();
        }
    }
    return con;
}
```

Si volguéssiu canviar la BD, l'únic que s'hauria de fer és canviar el contingut del fitxer de propietats. D'aquesta manera aconseguireu que el codi estigui feblement acoblat (*loosely coupled*) amb la configuració.

Un cop s'ha llegit el fitxer que conté les propietats amb el codi:

```
URL urlResource = classLoader.getResource("db.properties");
```

La classe `java.util.Properties` permet recuperar el valor de les propietats:

```
props.getProperty(NOM_DE_LA_PROPIETAT)
```

Un cop tenim la classe, escriurem el test unitari.

```
public class DBConnectionTest {
    DBConnection dBConnection;
    Connection connection;

    @Before
    public void setUp(){
        dBConnection = new DBConnection();
    }

    @After
    public void cleanUp() throws SQLException {
        connection.close();
    }

    @Test
```

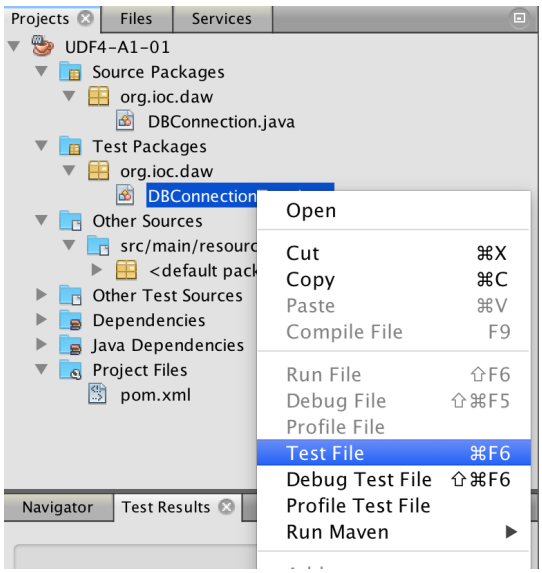
```
16      public void connectarAmbLaBaseDeDades() throws IOException, SQLException {
17          connection = dBConnection.getConnection();
18          Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().
                getDriverName());
19          Assert.assertEquals("SOCIOC_DB", connection.getCatalog());
20      }
21  }
```

En aquest primer test només esteu comprovant dues coses: que el nom de l'esquema que hem definit a l'URL de connexió és SOCIOC_DB i que el *driver* que esteu utilitzant és el *driver* JDBC d'H2. El mètode `assertEquals` de JUnit permet comprovar que l'objecte esperat (primer argument) és igual que el que s'obté quan s'executa el nostre codi.

```
1  Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().getDriverName())
       ;
2      Assert.assertEquals("SOCIOC_DB", connection.getCatalog());
```

A Netbeans, els tests es poden executar de forma individual, és a dir, classe per classe o tots els del projecte. Tal com podeu veure en la figura 1.5, si feu clic amb el botó dret sobre una classe dins de la carpeta *Test Packages* podreu executar els tests per a la classe seleccionada.

Figura 1.5. Execució dels tests d'una classe



Per executar tots els tests d'un projecte, primer heu de designar com a projecte principal el projecte UDF4-A1-01, tal com podeu veure en la figura 1.6. A continuació podreu executar tots els tests del projecte, tal com s'indica en la figura 1.7.

FIGURA 1.6. Selecció del projecte principal

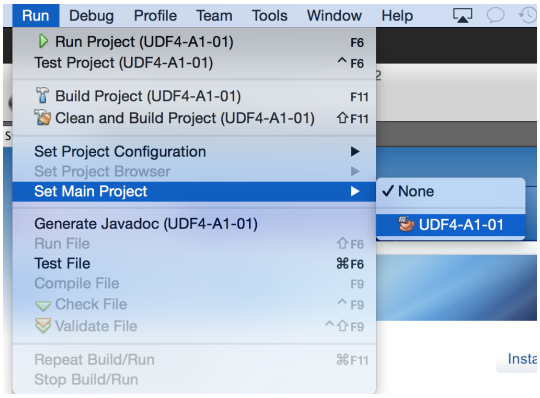
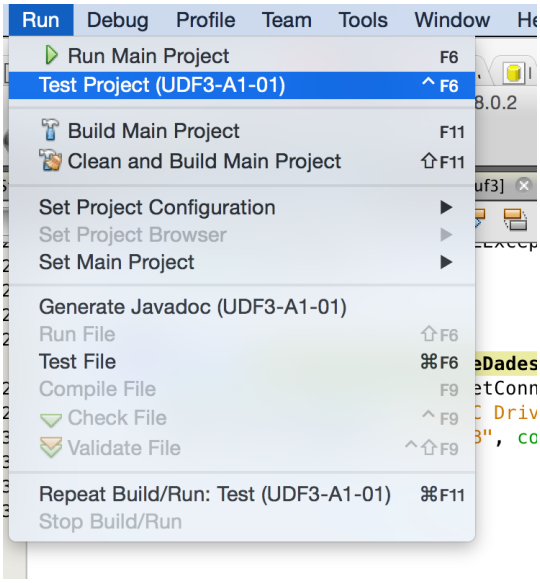


FIGURA 1.7. Selecció del projecte principal



Quan executeu els tests de JUnit a Netbeans, els resultats es mostren a la finestra de resultats de la figura 1.8. Per mostrar aquesta finestra seguiu els passos mostrats en figura 1.9.

FIGURA 1.8. Finestra de resultat de tests

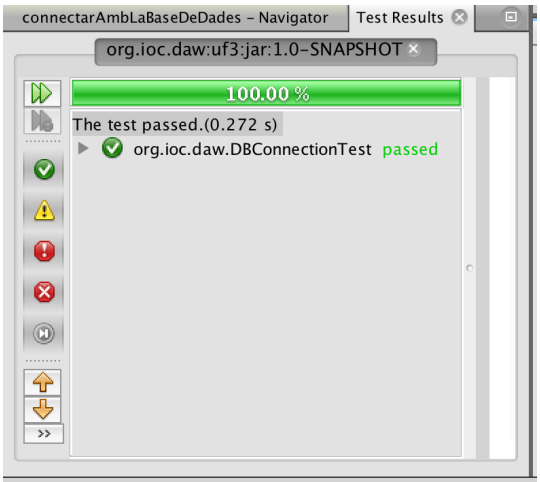
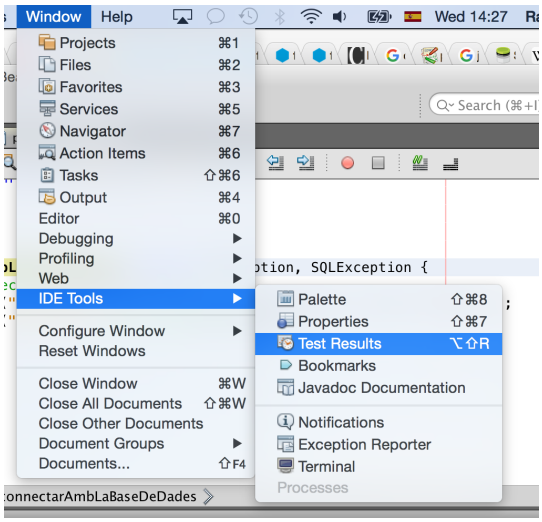


FIGURA 1.9. Mostrar finestra de resultat de tests



1.4 Tests unitaris amb JUnit

Els tests unitaris o proves de components són un tipus de proves de programari que consisteixen a fer proves sobre els components o unitats més petits del codi font d'una aplicació o d'un sistema. Això dóna la capacitat de verificar que les vostres funcions funcionen com s'esperava. És a dir, que per a qualsevol funció, i donat un conjunt d'entrades, podeu determinar si la funció retorna els valors adients i tracta correctament els errors.

Això ajuda a identificar les falles en els nostres algoritmes i/o lògica i ajuda a millorar la qualitat del codi que comprèn una funció determinada. Absolutament tots els components d'una aplicació han de tenir tests unitaris, fet que permet, durant qualsevol moment durant el desenvolupament, verificar la qualitat del treball.

Un segon avantatge de desenvolupar codi pensant sempre que s'ha de poder testejar és que el codi resultant és més fàcil de testejar. Com a resultat s'acaba estructurant el codi millor i es creen un major nombre de funcions més petites i especialitzades.

Un tercer avantatge de tenir un conjunt de tests unitaris sòlids és que preveuen que futurs canvis al codi trenquin la funcionalitat. Si en fer un canvi i executar els tests hi ha un error és clar que els canvis han introduït un error a la part específica que el test unitari està comprovant. Finalment, els tests unitaris proporcionen la millor documentació del sistema, ja que reflecteix exactament què s'espera que faci el codi. Els desenvolupadors que vulguin aprendre quina funcionalitat proporciona cada un dels components del sistema només han de llegir els tests unitaris.

1.4.1 El 'framework' de test JUnit

JUnit en la versió 4.x és un *framework* de test que utilitza anotacions per identificar els mètodes que especifiquen un test. Un test unitari és un mètode que s'especifica en una classe que només s'utilitza per al test. Això s'anomena *classe de test*. Per definir un mètode de test amb el *framework* JUnit 4.x es fa amb l'anotació `@org.junit.Test`. En aquest mètode s'utilitza un mètode d'asserció en el qual es comprova el resultat esperat de l'execució de codi en comparació del resultat real.

Vegeu amb detall el test unitari del punt 1.3:

```
1  public class DBConnectionTest {
2      DBConnection dBConnection;
3      Connection connection;
4
5      @Before
6      public void setUp(){
7          dBConnection = new DBConnection();
8      }
9
10     @After
11     public void cleanUp() throws SQLException {
12         connection.close();
13     }
14
15     @Test
16     public void connectarAmbLaBaseDeDades() throws IOException, SQLException {
17         connection = dBConnection.getConnection();
18         Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().
               getDriverName());
19         Assert.assertEquals("SOCIOC_DB", connection.getCatalog());
20     }
21 }
```

A la línies 5-9 hi ha el següent codi:

```
1  @Before
2    public void setUp(){
3        dBConnection = new DBConnection();
4    }
```

L'anotació `@Before` serveix per marcar una funció que s'executarà abans de l'execució de cada test i serveix per assegurar que tots els tests sempre parteixen de les mateixes condicions per ser executats. De manera similar, les línies 10-13:

```
1  @After
2    public void cleanUp() throws SQLException {
3      connection.close();
4    }
```

Sempre s'executaran en acabar cada un dels tests unitaris. En el vostre cas us assegurareu que en iniciar un test s'establirà connexió amb la base de dades, i en acabar es tancarà la connexió.

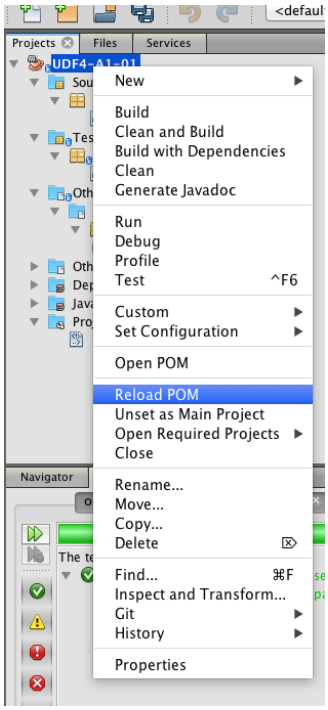
1.4.2 Cobertura dels tests unitaris

Una cosa fonamental és saber quina part del vostre codi està cobert pels tests unitaris. Idealment s'ha d'intentar aconseguir un 100%, però normalment es considera com a acceptable una cobertura de 80-90%. La versió 7 de Netbeans ja té incorporat un *plugin* que permet veure la cobertura de projectes Maven (`MavenCodeCoverage`). Afegiu el següent al final del fitxer pom.xml.

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.7.5.201505241946</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

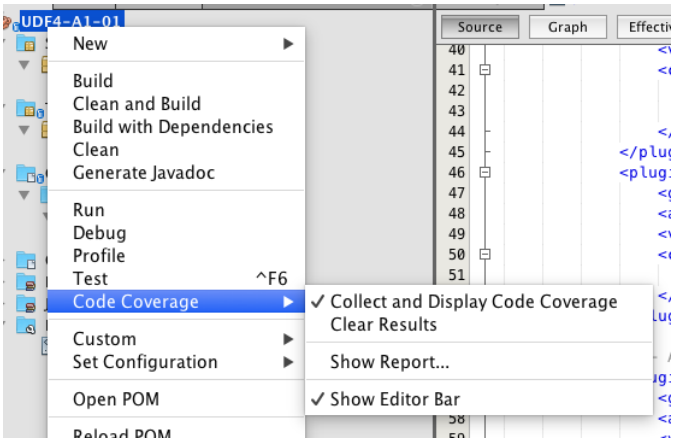
A continuació només fa falta dir a Netbeans que torni a carregar el pom.xml, com podeu veure en la figura 1.10.

FIGURA 1.10. Recarregar pom.xml



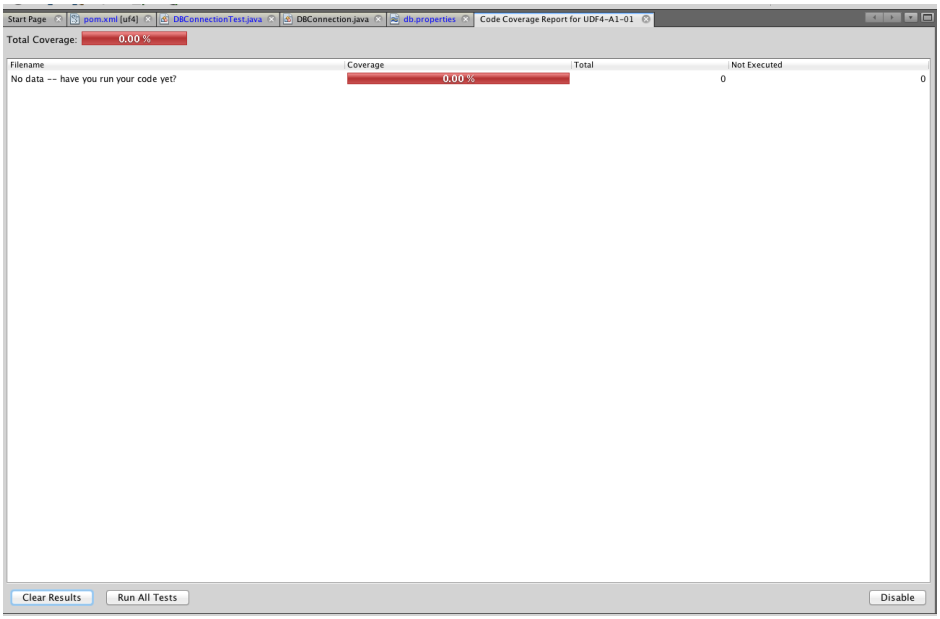
Un cop fet això ja podeu accedir a la cobertura dels tests unitaris que teniu al projecte, tal com podeu veure en la figura 1.11 i seleccionant *Show report*.

Figura 1.11. Cobertura



Inicialment no hi ha informació, així que haureu d'executar tots els tests prenent el botó *Run All Tests* (vegeu la figura 1.12).

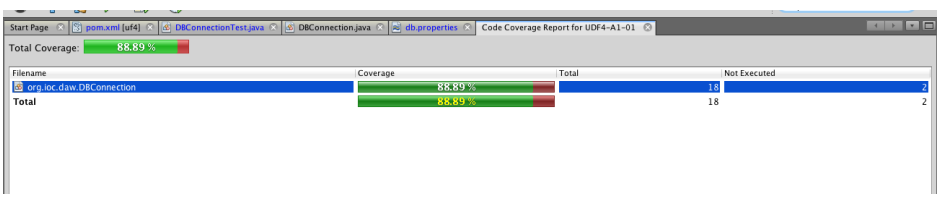
Figura 1.12. Report de cobertura inicial



Per veure el resultat heu de tancar i tornar a obrir la pestanya on es mostren els resultats (*Code Coverage report*).

Un cop el vostre test s'ha executat podeu veure que tenim una cobertura d'un 88.89% (vegeu la figura 1.13).

Figura 1.13. Report de cobertura inicial



Si feu clic a la classe que hem testejat (`DBConnection`) podeu veure les parts del codi que estan testejades i les que no (vegeu la figura 1.14).

Figura 1.14. Report de cobertura inicial

```
package org.ioc.daw;

import org.h2.jdbc.JdbcConnection;

import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DBConnection {
    public Connection getConnection() throws SQLException, IOException {
        Properties props = new Properties();
        InputStream resourceAsStream = null;
        Connection con = null;
        try {
            ClassLoader classLoader = getClass().getClassLoader();
            URL urlResource = classLoader.getResource("db.properties");
            if (urlResource != null) {
                resourceAsStream = urlResource.openStream();
                props.load(resourceAsStream);
                Class.forName(props.getProperty("DB_DRIVER_CLASS"));
                con = DriverManager.getConnection(props.getProperty("DB_URL"),
                        props.getProperty("DB_USERNAME"),
                        props.getProperty("DB_PASSWORD"));
            }
        } catch (IOException | ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        } finally {
            if (resourceAsStream != null) {
                resourceAsStream.close();
            }
        }
        return con;
    }
}
```

Podeu veure que la part de codi sense cobertura dels nostres tests és la que fa referència a com tractar les excepcions.

1.4.3 Augmentant la cobertura del codi

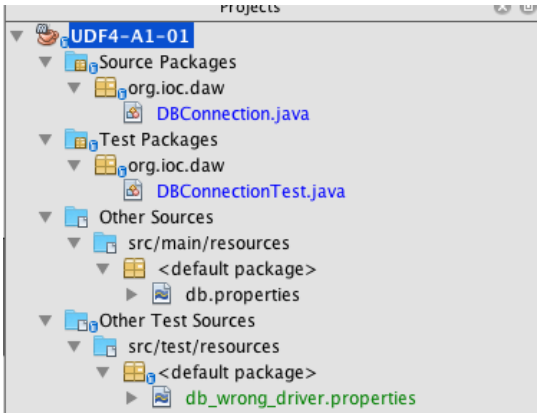
El codi del qual partireu el teniu disponible a l'apartat d'annexos de la unitat.

El vostre següent objectiu és aconseguir un 100% de cobertura als tests unitaris. Per aconseguir això heu de canviar el codi, ja que heu de ser capaços de poder crear de manera controlada errors i assegurar-vos que es tracten correctament.

En el vostre cas, una forma fàcil de testejar les excepcions és, per exemple, crear un fitxer de propietats que contingui un nom de *driver* de base de dades incorrecte. Creeu el següent fitxer als recursos de test (figura 1.15) amb el contingut següent i amb el nom db_wrong_driver.properties:

```
#BD en memoria H2
DB_DRIVER_CLASS=org.h2.WrongDriver
DB_URL=jdbc:h2:mem:socioc_db
DB_USERNAME=usuari
DB_PASSWORD=passws
```

FIGURA 1.15. Fitxer de connexió amb la BD



En aquest exemple, la classe `org.h2.WrongDriver` no existeix, fet que provocarà un error quan s'intenti carregar el *driver* en memòria amb `Class.forName(props.getProperty(‘‘DB_DRIVER_CLASS’’));`.

Això presenta un problema: a la classe `DBConnection` no hi ha manera de seleccionar quin fitxer de propietats s'ha d'utilitzar. El primer que haureu de fer, doncs, és refactoritzar el mètode `getConnection` perquè accepti un paràmetre que us serveixi per passar a quin fitxer de propietats utilitzar.

> **Refactorització** és el procés de reestructurar el codi d'una aplicació sense canviar la seva funcionalitat per tal de millorar la seva eficiència, estructura, llegibilitat o reutilització.

```
1  public Connection getConnection(String dbProperties) throws SQLException,
       IOException {
2      Properties props = new Properties();
3      InputStream resourceAsStream = null;
4      Connection con = null;
5      try {
6        ClassLoader classLoader = getClass().getClassLoader();
7        URL urlResource = classLoader.getResource(dbProperties);
8        if (urlResource != null) {
9          resourceAsStream = urlResource.openStream();
10         props.load(resourceAsStream);
11         Class.forName(props.getProperty("DB_DRIVER_CLASS"));
12         con = DriverManager.getConnection(props.getProperty("DB_URL"),
13               props.getProperty("DB_USERNAME"),
14               props.getProperty("DB_PASSWORD"));
15       }
16     } catch (IOException | ClassNotFoundException | SQLException e) {
17       e.printStackTrace();
18     } finally {
19       if (resourceAsStream != null) {
20         resourceAsStream.close();
21       }
22     }
23   return con;
24 }
```

Vegeu aquí un dels avantatges de fer tests unitaris, el fet de pensar com testejar el codi millora el diseny de la nostra API. En aquest cas, afegir aquest paràmetre us permetrà utilitzar el mateix codi amb diferents bases de dades, ja que l'únic

que farà falta serà passar el fitxer de propietats adient. Ara afegireu un test que carregui el fitxer amb la configuració errònia:

```
@Test
public void dbConnectionWrongDriver() throws IOException, SQLException {
    connection = dBConnection.getConnection("db_wrong_driver.properties");
    Assert.assertNull(connection);
}
```

Com que l'objecte `Connection` pot ser ara *null*, s'ha de canviar el que es fa després d'executar el test.

```
@After
public void cleanUp() throws SQLException {
  if(connection != null){
    connection.close();
  }
}
```

Comproveu vosaltres mateixos que ara tenim una cobertura dels tests unitaris del 100%.

1.5 Fent consultes a la base de dades

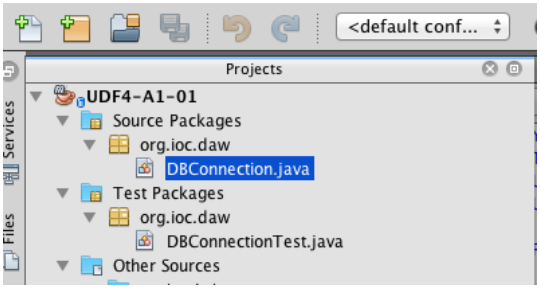
A continuació començareu a treballar amb la base de dades que utilitzareu per construir l'aplicació que anomenareu "SocIoc". La idea general és desenvolupar una xarxa social on els estudiants pugueu preguntar i resoldre preguntes relacionades amb les assignatures que esteu cursant. Les millors respostes es votaran i anireu guanyant punts i creant-vos una reputació. Començareu per obtenir un llistat dels alumnes que hi ha al sistema.

Intenteu contestar a les següents preguntes:

- Quines classes haurem d'afegir al sistema?
- Com obtindrem les dades de la base de dades?

Les bases de dades s'estructuren en taules que tenen una sèrie de registres amb informació. Quan vulgueu extraure una part d'aquesta informació fareu una consulta que us retornarà els registres adients. La vostra aplicació no entén de registres de bases de dades, i necessitareu expressar la informació que voleu guardar a la base de dades com una entitat que pugueu utilitzar a l'aplicació. Llavors, el primer que fareu serà definir una classe que representi un usuari del sistema. Com podeu veure en la figura 1.16, teniu un paquet `org.iow.daw` que és molt general, així que ara que afegireu més classes és necessari que comenceu a estructurar les classes en diferents paquets.

FIGURA 1.16. Paquet org.ioc.daw



Hi ha moltes formes de fer aquesta organització, i no n'hi cap de correcta o incorrecta mentre siguin consistents i ben organitzades. La millor estructura és organitzar els paquets en funció de les entitats de domini que hi hagi a la nostra aplicació. Per exemple, el paquet `org.ioc.daw.user` contindrà totes les classes amb funcionalitat relacionada amb els usuaris. Creeu aquest paquet i el paquet `org.ioc.daw.db` tal com es mostra en la figura 1.17 i en la figura 1.18.

FIGURA 1.17. Paquet org.ioc.daw

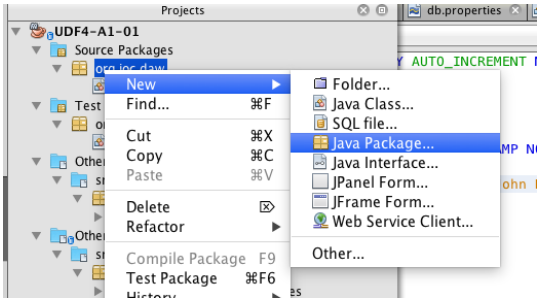
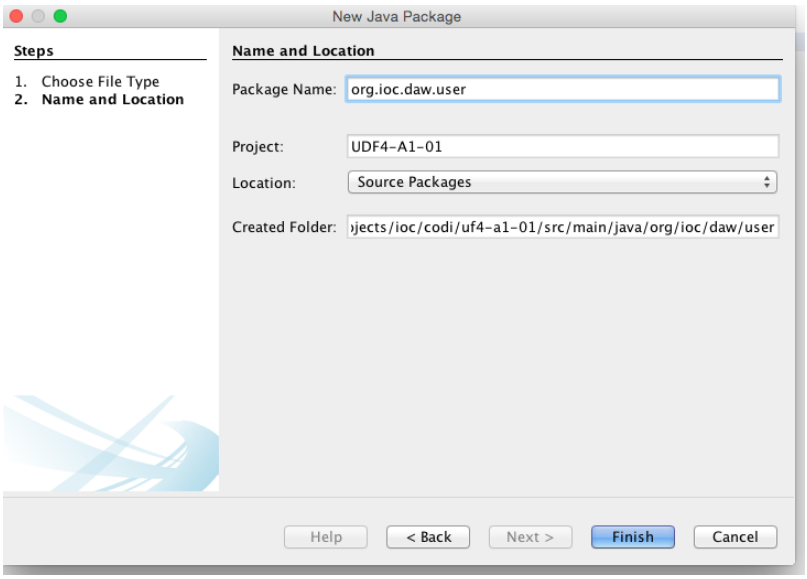


FIGURA 1.18. Paquet org.ioc.daw



A partir del codi de partida del present apartat, creeu la classe `User` amb el contingut que es mostra a continuació i moveu la classe `DBConnection` al paquet `org.ioc.daw.db`.

Trobareu el codi de partida per al present apartat als annexos de la unitat.

```java
public class User {
    private int userId;
    private String username;
    private String name;
    private String email;
    private int rank;
    private Timestamp createdOn;
    private boolean active;

    public User(int userId, String username, String name, String email, int
         rank, Timestamp createdOn, boolean active) {
        this.userId = userId;
        this.username = username;
        this.name = name;
        this.email = email;
        this.rank = rank;
        this.createdOn = createdOn;
        this.active = active;
    }

    public int getUserId() {
        return userId;
    }

    public String getUsername() {
        return username;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public int getRank() {
        return rank;
    }

    public Timestamp getCreatedOn() {
        return createdOn;
    }

    public boolean isActive() {
        return active;
    }
}
```

Com que aquesta classe no té cap lògica, no fa falta escriure un test unitari. Seguidament definireu una classe que s'encarregui de fer les operacions amb la base de dades. Normalment, a aquest tipus de classes se les anomena *Data Access Objects* (DAO, objectes d'accés a dades), per la qual cosa anomenareu la vostra classe `UserDAO`.

```java
public class UserDAO {
    public List<User> findAllUsers() {
        String qry = "select user_id, username, name, email, rank, active,
             created_on from users";
        DBConnection dBConnection = new DBConnection();
        List<User> users = new ArrayList<>();
        try (
                Connection conn = dBConnection.getConnection("db.properties");
                Statement stmt = conn.createStatement();
                ResultSet rs = stmt.executeQuery(qry);) {
            while (rs.next()) {
```

```
11                  int userId = rs.getInt("user_id");
12                  String username = rs.getString("username");
13                  String name = rs.getString("name");
14                  String email = rs.getString("email");
15                  int rank = rs.getInt("rank");
16                  boolean active = rs.getBoolean("active");
17                  Timestamp timestamp = rs.getTimestamp("created_on");
18                  User user = new User(userId, username, name, email, rank,
                        timestamp, active);
19                  users.add(user);
20              }
21          } catch (SQLException | IOException e) {
22              e.printStackTrace();
23          }
24          return users;
25      }
26  }
```

Una de les operacions fetes més comunament contra una base de dades és una consulta. Fer consultes de bases de dades utilitzant JDBC és bastant fàcil, encara que hi ha una mica de codi repetitiu que s'ha d'utilitzar cada vegada que s'executa una consulta. En primer lloc, es necessita obtenir un objecte de connexió amb la BD. Després es crea una una consulta i es guarda a una variable de tipus *string*. La línia 3 defineix la consulta que es farà a la base de dades. En aquesta estem demanant a la BD que retorni tots els registres de la taula "Users".

```
1  select user_id, username, name, email, rank, active, created_on from users;
```

A continuació s'utilitza una clàusula `try-with-resources` per crear els objectes que són necessaris per fer la consulta de la base de dades. Això farà que si hi ha qualsevol problema i el programa llença una excepció, tots els recursos es tancaran automàticament.

```
1  try (
2          Connection conn = dBConnection.getConnection("db.properties");
3          Statement stmt = conn.createStatement();
4          ResultSet rs = stmt.executeQuery(qry);)
```

En aquest cas, si hi ha cap problema creant la connexió amb la BD, creant o executant la consulta, tancarem automàticament la connexió amb la BD.

Com es pot veure, el mètode `executeQuery` accepta un *string* i retorna un objecte `ResultSet`. L'objecte `ResultSet` fa que sigui fàcil treballar amb els resultats de la consulta. Si observeu la següent línia de codi (línia 9), un bucle *while* es crea amb `rs.next()`. Aquest bucle recorrerà els continguts de l'objecte `ResultSet`, obtenint la següent fila que es retorna des de la consulta amb cada iteració. Una vegada totes les files retornades han estat processades, `rs.next()` retornarà *false* per indicar que no hi ha més resultats que processar.

Dins del bucle *while*, l'objecte `ResultSet` s'utilitza per obtenir els valors dels noms de les columnes indicades amb cada passada. Observeu que si s'espera que la columna retorni un *string* heu d'utilitzar el mètode `ResultSet.getString`, passant el nom de la columna en format *string*. De la mateixa manera, si s'espera que la columna retorni un *int* haureu d'utilitzar el mètode `ResultSet.getInt`. Un cop tenim tota la informació necessària, es creen els objectes de tipus `User` i s'afegeixen a la llista de tots els usuaris (línies 18-19).

A continuació necessiteu fer un test unitari per assegurar-vos que la classe `UserDAO` té el funcionament desitjat. Teniu un problema, però: el fitxer db.properties conté les dades d'accés a la BD principal o de producció. Per fer els tests voleu poder configurar la BD que utilitzareu, ja que això permetrà usar una BD específica per executar-los. Refactoritzareu el codi per tal que l'objecte `DBConnection` no es creï dintre de la classe `UserDAO`, sinó que es passi com a paràmetre. D'aquesta forma aconseguireu el desacoblament de la classe que fa consultes a la BD amb la qual s'encarrega de fer la connexió.

```
1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
6
7      public List<User> findAllUsers() {
8          String qry = "select user_id, username, name, email, rank, active,
               created_on from users";
9          List<User> users = new ArrayList<>();
10         try (
11                 Connection conn = getDBConnection();
12                 Statement stmt = conn.createStatement();
13                 ResultSet rs = stmt.executeQuery(qry);) {
14             while (rs.next()) {
15                 int userId = rs.getInt("user_id");
16                 String username = rs.getString("username");
17                 String name = rs.getString("name");
18                 String email = rs.getString("email");
19                 int rank = rs.getInt("rank");
20                 boolean active = rs.getBoolean("active");
21                 Timestamp timestamp = rs.getTimestamp("created_on");
22                 User user = new User(userId, username, name, email, rank,
                       timestamp, active);
23                 users.add(user);
24             }
25         } catch (SQLException | IOException e) {
26             e.printStackTrace();
27         }
28         return users;
29     }
30
31     public getDBConnection(){
32         this.dBConnection = dBConnection;
33     }
34 }
```

Una forma de fer això és declarar `DBConnection dBConnection` com un atribut de la classe `UserDAO` i fer que sigui necessari per a la creació de l'objecte.

```
1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
```

A la classe `DBConnection`, el fitxer amb els paràmetres de connexió amb la base de dades és imprescindible, així que també refactoritzareu aquesta classe.

```
1  public class DBConnection {
2      private String connectionFile;
3      public DBConnection(String connectionFile) {
4          this.connectionFile = connectionFile;
5      }
```

Aquest últim canvi trencarà el test unitari, així que haureu de canviar la classe `DBConnectionTest`.

Ara ja podeu crear el test unitari per testejar la classe `UserDAO`.

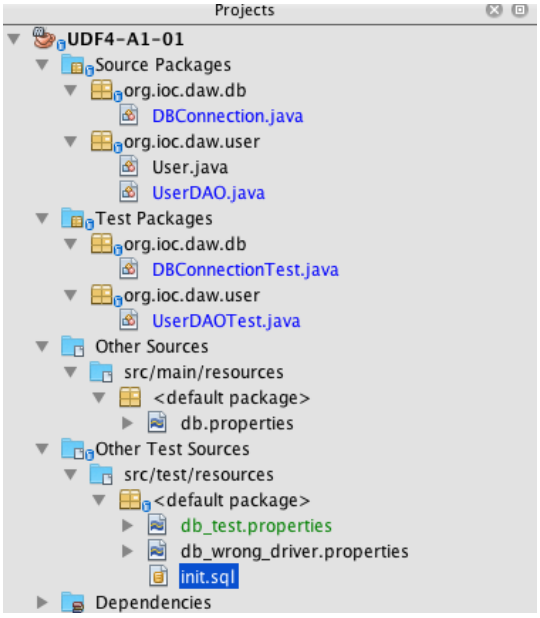
```
public class UserDAOTest {
    private DBConnection dBConnection;
    private String connectionProperties = "db_test.properties";
    @Before
    public void setUp(){
        dBConnection = new DBConnection(connectionProperties);
    }

    @Test
    public void findAllUsers(){
        UserDAO userDAO = new UserDAO(dBConnection);
        List<User> users  = userDAO.findAllUsers();
        Assert.assertEquals("Hauriem de tenir 2 usuaris a la base de dades", 2,
             users.size());
    }
}
```

Abans de poder executar el test, creareu el fitxer de connexió amb la BD de test.

```
DB_DRIVER_CLASS=org.h2.Driver
DB_URL=jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql';
DB_USERNAME=usuari
DB_PASSWORD=passwd
```

Com que en aquest cas és una BD en memòria, no necessiteu canviar el nom de la BD. La part més important d'aquest fitxer és la segona línia, que executarà una sèrie d'instruccions SQL que asseguraran que la BD de test sempre estarà en el mateix estat. En la figura 1.19 es mostren els fitxers que heu de tenir abans d'executar els tests unitaris.

FIGURA 1.19. Fitxers pel test unitari de UserDAO




```
CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
                   username VARCHAR(10) NOT NULL,
                   name  VARCHAR(20) NOT NULL,
```

```
4                   email VARCHAR(50) NOT NULL,
5                   rank INT DEFAULT 0,
6                   active BOOLEAN DEFAULT true,
7                   created_on TIMESTAMP AS CURRENT_TIMESTAMP NOT NULL);
8
9   INSERT INTO users (username, name, email) VALUES ('user1', 'John Test', '
        john@email.com');
10  INSERT INTO users (username, name, email) VALUES ('user2', 'Paul Test', '
        paul@email.com');
```

1.6 Operacions CRUD

CRUD correspon a l'acrònim *Create Read Update Delete*: crear, recuperar , actualitzar i eliminar, i es refereix a les quatre funcions principals que implementareu quan desenvolupeu aplicacions de bases de dades. Les funcions CRUD són les que donen la capacitat a les aplicacions de comportar-se d'una manera dinàmica, ja que les dades podran ser canviades pels usuaris. Aquests podran crear, visualitzar, modificar i alterar les dades. Les operacions CRUD permeten accedir i manipular les entitats definides a les bases de dades.

Per exemple, a la taula d'usuaris definida a l'apartat 1.5, *Create* implicarà afegir un nou usuari; *Read*, accedir a les dades d'un o diversos estudiants; *Update* modificarà les dades dels usuaris i *Delete* eliminarà un o diversos registres de la base de dades.

El primer que farem serà modificar la classe `UserDAO` per permetre recuperar les dades de la base de dades. Recordeu que l'objectiu és construir una aplicació que permeti preguntar i resoldre preguntes relacionades amb les assignatures que esteu cursant. Les millors respostes es votaran i anireu guanyant punts i creant-vos una reputació. Quines creieu, doncs, que seran el tipus d'informació que necessitareu extreure de la base de dades? Intenteu contestar a les següents preguntes:

- En quines situacions necessitareu obtenir la informació d'un usuari?
- Per mostrar un llistat amb els alumnes que mostri un *ranking* dels alumnes més ben classificats, quina informació necessitareu?

Les operacions de lectura que haureu de dissenyar per a la vostra aplicació estaran totalment determinades per la seva funcionalitat. Tot i així, veureu amb la vostra experiència que hi haurà unes operacions que es repetiran i seran comuns per a moltes aplicacions. Per exemple, en el cas que us ocupa, una aplicació amb usuaris que es poden registrar i donar-se de baixa, moltes de les operacions que definim serviran per a la majoria d'aplicacions que us trobareu, des d'una botiga amb comerç electrònic, un joc *online* o una xarxa social.

1.6.1 Operacions de lectura

Primer de tot, quan un usuari es registri en el sistema haurà d'introduir el seu nom, el nom d'usuari que desitja i el seu correu electrònic. Què passa si el nom d'usuari ja està en ús per part d'un altre usuari? I el correu electrònic? En aquests casos hauríem d'avisar l'usuari que ha d'escollir un altre nom d'usuari o correu electrònic. Ja tenim dues operacions de lectura que haurem d'implementar: trobar usuaris a partir del correu electrònic i a partir del nom d'usuari. Implementem-les:

```
public User findUserByEmail(String userEmail){
  String qry = "select * from users where email ='" + userEmail +"'";

  User user = null;
  try (
    Connection conn = dBConnection.getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(qry);) {
    while (rs.next()) {
      int userId = rs.getInt("user_id");
      String username = rs.getString("username");
      String name = rs.getString("name");
      String email = rs.getString("email");
      int rank = rs.getInt("rank");
      boolean active = rs.getBoolean("active");
      Timestamp timestamp = rs.getTimestamp("created_on");
      user = new User(userId, username, name, email, rank, timestamp, active);
    }
  } catch (SQLException | IOException e) {
    e.printStackTrace();
  }
  return user;
```

La part més important és on es defineix la consulta que es farà a la base de dades:

```
String qry = "select * from users where email ='" + userEmail +"'";
```

Aquesta és la consulta SQL que retornarà un usuari a partir del seu correu electrònic. Si no hi ha cap usuari registrat amb el correu electrònic retornarà *null*. Això es pot expressar amb el següent test unitari:

```
@Test
  public void findUserByEmail(){
      String existingEmail = "john@email.com";
      String unknownEmail = "does.not@exist.com";

      UserDAO userDAO = new UserDAO(dBConnection);
      User user = userDAO.findUserByEmail(existingEmail);
      Assert.assertNotNull(user);
      user = userDAO.findUserByEmail(unknownEmail);
      Assert.assertNull(user);
  }
```

`Assert.assertNotNull` verifica que un objecte no és *null*, és a dir, s'ha trobat un resultat a la base de dades. Al contrari, `Assert.assertNull` verifica que un objecte és *null*; en el vostre cas, que no s'ha trobat cap usuari a la base de dades amb un cert correu electrònic.

A continuació implementareu un mètode que us permeti trobar usuaris a partir del nom d'usuari.

```
public User findUserByUsername(String username){
  String qry = "select * from users where username ='" + username +"'";
  User user = null;
  try (
          Connection conn = dBConnection.getConnection();
          Statement stmt = conn.createStatement();
          ResultSet rs = stmt.executeQuery(qry);) {
      while (rs.next()) {
          int userId = rs.getInt("user_id");
          username = rs.getString("username");
          String name = rs.getString("name");
          String email = rs.getString("email");
          int rank = rs.getInt("rank");
          boolean active = rs.getBoolean("active");
          Timestamp timestamp = rs.getTimestamp("created_on");
          user = new User(userId, username, name, email, rank, timestamp,
              active);
      }
  } catch (SQLException | IOException e) {
      e.printStackTrace();
  }
  return user;
}
```

I el test unitari:

```
@Test
  public void findUserByUsername(){
      String existingUsername = "user1";
      String unknownUsername = "unknown";

      UserDAO userDAO = new UserDAO(dBConnection);
      User user = userDAO.findUserByUsername(existingUsername);
      Assert.assertNotNull(user);
      user = userDAO.findUserByUsername(unknownUsername);
      Assert.assertNull(user);
  }
```

Doneu una ullada a la classe sencera, ja que hi ha codi repetit; símptoma inequívoc que és hora de refactoritzar el codi.

```
public class UserDAO {
  private DBConnection dBConnection;
  public UserDAO(DBConnection dBConnection){
      this.dBConnection = dBConnection;
  }

  public List<User> findAllUsers() {
      String qry = "select user_id, username, name, email, rank, active,
          created_on from users";
      List<User> users = new ArrayList<>();
      try (
          Connection conn = dBConnection.getConnection();
          Statement stmt = conn.createStatement();
          ResultSet rs = stmt.executeQuery(qry);) {
        while (rs.next()) {
          int userId = rs.getInt("user_id");
          String username = rs.getString("username");
          String name = rs.getString("name");
          String email = rs.getString("email");
          int rank = rs.getInt("rank");
          boolean active = rs.getBoolean("active");
          Timestamp timestamp = rs.getTimestamp("created_on");
```

```
22          User user = new User(userId, username, name, email, rank, timestamp,
                active);
23          users.add(user);
24        }
25      } catch (SQLException | IOException e) {
26          e.printStackTrace();
27      }
28      return users;
29  }
30
31  public User findUserByEmail(String userEmail){
32      String qry = "select * from users where email ='" + userEmail +"'";
33      User user = null;
34      try (
35          Connection conn = dBConnection.getConnection();
36          Statement stmt = conn.createStatement();
37          ResultSet rs = stmt.executeQuery(qry);) {
38        while (rs.next()) {
39          int userId = rs.getInt("user_id");
40          String username = rs.getString("username");
41          String name = rs.getString("name");
42          String email = rs.getString("email");
43          int rank = rs.getInt("rank");
44          boolean active = rs.getBoolean("active");
45          Timestamp timestamp = rs.getTimestamp("created_on");
46          user = new User(userId, username, name, email, rank, timestamp,
                active);
47          }
48      } catch (SQLException | IOException e) {
49          e.printStackTrace();
50      }
51      return user;
52  }
53
54
55  public User findUserByUsername(String username){
56      String qry = "select * from users where username ='" + username +"'";
57      User user = null;
58      try (
59            Connection conn = dBConnection.getConnection();
60            Statement stmt = conn.createStatement();
61            ResultSet rs = stmt.executeQuery(qry);) {
62          while (rs.next()) {
63            int userId = rs.getInt("user_id");
64            username = rs.getString("username");
65            String name = rs.getString("name");
66            String email = rs.getString("email");
67            int rank = rs.getInt("rank");
68            boolean active = rs.getBoolean("active");
69            Timestamp timestamp = rs.getTimestamp("created_on");
70            user = new User(userId, username, name, email, rank, timestamp,
                  active);
71          }
72      } catch (SQLException | IOException e) {
73          e.printStackTrace();
74      }
75      return user;
76  }
77 }
```

Si us hi fixeu, la part del codi que s'encarrega de fer la connexió amb la base de dades i extreure els resultats es repeteix, millor posar-la a un mètode. Per què? Imagineu la següent situació: més endavant, durant el desenvolupament de l'aplicació, volem afegir un altre atribut a la classe `User`, com per exemple la data de naixement. Tal com tenim el codi ara mateix, fer aquest canvi implicarà canviar tres mètodes: `findAllUsers`, `findUserByEmail` i `findUserByUsername`. Si

aquest codi comú el poseu en un mètode separat, el canvi només l'haureu de fer en un lloc.

```
public class UserDAO {
  private DBConnection dBConnection;
  public UserDAO(DBConnection dBConnection){
      this.dBConnection = dBConnection;
  }

  public List<User> findAllUsers() {
      String qry = "select user_id, username, name, email, rank, active,
           created_on from users";
      List<User> users = new ArrayList<>();
      try (
              Connection conn = dBConnection.getConnection();
              Statement stmt = conn.createStatement();
              ResultSet rs = stmt.executeQuery(qry);) {
          while (rs.next()) {
              User user = buildUserFromResultSet(rs);
              users.add(user);
          }
      } catch (SQLException | IOException e) {
          e.printStackTrace();
      }
      return users;
  }

  public User findUserByEmail(String userEmail){
      String qry = "select * from users where email ='" + userEmail +"'";
      User user = null;
      try (
              Connection conn = dBConnection.getConnection();
              Statement stmt = conn.createStatement();
              ResultSet rs = stmt.executeQuery(qry);) {
          while (rs.next()) {
              user = buildUserFromResultSet(rs);
          }
      } catch (SQLException | IOException e) {
          e.printStackTrace();
      }
      return user;
  }

  public User findUserByUsername(String username){
      String qry = "select * from users where username ='" + username +"'";
      User user = null;
      try (
              Connection conn = dBConnection.getConnection();
              Statement stmt = conn.createStatement();
              ResultSet rs = stmt.executeQuery(qry);) {
          while (rs.next()) {
              user = buildUserFromResultSet(rs);
          }
      } catch (SQLException | IOException e) {
          e.printStackTrace();
      }
      return user;
  }

  private User buildUserFromResultSet(ResultSet rs) throws SQLException{
      int userId = rs.getInt("user_id");
      String username = rs.getString("username");
      String name = rs.getString("name");
      String email = rs.getString("email");
      int rank = rs.getInt("rank");
      boolean active = rs.getBoolean("active");
      Timestamp timestamp = rs.getTimestamp("created_on");
      User user = new User(userId, username, name, email, rank, timestamp,
           active);
      return user;
```

```
  }
}
```

Hem creat la funció `buildUserFromResultSet`, que s'encarrega de construir un objecte usuari a partir del resultat d'executar la consulta SQL. Un dels avantatges de tenir tests unitaris és que podeu assegurar-vos que la refactorització no ha trencat la funcionalitat del vostre codi. Si correu els tests a la classe `UserDAOTest` ha de continuar passant.

Encara es pot refactoritzar més, ja que totes les funcions tenen la mateixa estructura:

- Establir connexió amb la base de dades
- Executar una consulta
- Iterar sobre resultats i construir un o diversos objectes `User`

L'única diferència és que hi haurà funcions que retornaran només un usuari o diversos. Això ho podem solucionar creant les següents funcions:

```
private User findUniqueResult(String query) throws Exception{
    List<User> users = executeQuery(query);
    if(users.isEmpty()){
        return null;
    }
    if(users.size() > 1){
        throw new Exception("Only one result expected");
    }
    return users.get(0);
}

private List<User> executeQuery(String query){
    List<User> users = new ArrayList<>();
    try (
            Connection conn = dBConnection.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(query);) {
        while (rs.next()) {
            User user = buildUserFromResultSet(rs);
            if(user != null){
                users.add(user);
            }
        }
    } catch (SQLException | IOException e) {
        e.printStackTrace();
    }
    return users;
}
```

La funció `executeQuery` s'encarrega d'establir la connexió amb la base de dades i retornar tots els usuaris que resultin d'executar la consulta. En aquest cas no sabem si retornarà un, diversos o cap resultat. Això es pot expressar amb el tipus de dades *List*; si la llista està buida voldrà dir que no hi ha cap resultat, i si n'hi ha, n'hi haurà tants com elements contingui la llista. Per altra part, la funció `findUniqueResult` examinarà la llista retornada per `executeQuery` i retornarà un objecte *null* si no hi ha cap resultat per a la consulta; un objecte `User`, si hi és, troba un resultat i es llençarà una excepció si hi ha un error. Per avisar la vostra aplicació que alguna cosa ha anat malament llençareu una excepció.

```
if(users.size() > 1){
  throw new Exception("Only one result expected");
}
```

Aquesta excepció la propagareu a capes superiors de l'aplicació, on decidireu què fer. Les funcions que permeten obtenir informació sobre els usuaris quedaran molt més simplificades.

```
public List<User> findAllUsers() {
    String qry = "select user_id, username, name, email, rank, active,
        created_on from users";
    List<User> users = executeQuery(qry);
    return users;
}

public User findUserByEmail(String userEmail) throws Exception{
    String qry = "select * from users where email ='" + userEmail +"'";
    return findUniqueResult(qry);
}

public User findUserByUsername(String username) throws Exception{
    String qry = "select * from users where username ='" + username +"'";
    return findUniqueResult(qry);
}
```

1.6.2 Operacions d'escriptura

Comenceareu a treballar a partir del codi que teniu disponible a l'apartat d'annexos de la unitat.

Hi ha dues formes diferents d'escriure dades: crear una nova entrada a la base de dades o actualitzar un registre existent.

Contesteu a les següents preguntes:

- Quines operacions necessitareu de creació de dades?
- Quines faran falta d'actualització?

Com podeu veure al codi, en aquest moment només heu definit els usuaris, llavors només podreu crear usuaris. Per altra part, respecte a l'actualització hi ha diverses operacions que caldrà fer:

- Crear un usuari.
- Canviar el correu electrònic.
- Actualitzar el *ranking*.
- Actualitzar el nom.

La creació es fa mitjançant l'operació SQL *INSERT* i l'actualització amb *UPDATE*. Voleu crear un usuari a partir del nom, nom d'usuari i correu electrònic. Què passarà amb la resta d'atributs de l'objecte `User` (`userId`, `rank`, `createdOn` i `active`)? Les principals diferències amb el codi per llegir dades seran dues: la

sentència SQL, que ara contindrà la paraula clau *INSERT*, i l'operació que cridarem en l'objecte `Statement`, que serà `executeUpdate` en lloc de `executeQuery`.

```
1  public User createUser(String username, String name, String email) throws
       Exception {
2    String qry = "INSERT INTO users (username, name, email) VALUES ('"
3            + username + "', '"
4            + name + "', '"
5            + email + "')";";
6    try (
7          Connection conn = dBConnection.getConnection();
8          Statement stmt = conn.createStatement()) {
9
10     int result = stmt.executeUpdate(qry);
11     if (result == 0) {
12         throw new Exception("Error creating user");
13     }
14     return findUserByUsername(username);
15
16   } catch (SQLException | IOException e) {
17       e.printStackTrace();
18       return null;
19   }
20 }
21
```

El mètode `executeUpdate` (línia 11) retornarà la quantitat de registres que s'han inserit en la base de dades. Per aquest motiu, si el nombre de registres inserits és 0 voldrà dir que hi ha hagut un error afegint l'usuari a la base de dades (línies 12-14). Finalment (línia 15), un cop creat el nou usuari voldreu retornar l'objecte `User` amb tots els seus atributs. Aprofiteu la funció `findUserByUsername` per recuperar l'usuari recentment creat de la base de dades. A continuació podeu veure el test corresponent:

```
1  @Test
2  public void createUser() throws Exception {
3      String username = "testUser";
4      String name = "Pete Test";
5      String email = "pete@email.com";
6      UserDAO userDAO = new UserDAO(dBConnection);
7      User createdUser = userDAO.createUser(username, name, email);
8      Assert.assertNotNull(createdUser);
9      Assert.assertEquals(username, createdUser.getUsername());
10     Assert.assertNotEquals(0, createdUser.getUserId());
11 }
```

Primer comproveu que l'usuari creat no és *null*, que el nom d'usuari correspon al nom que hem passat a la funció `createUser` i que l'atribut `userId` del nou usuari no és 0.

> Quan es va definir la taula, el camp "user_id" es va definir com *user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL*; llavors, si el nou usuari s'ha afegit correctament a la base de dades, es pot afirmar que el seu valor no serà *null*.

Quan executeu aquest test us trobareu que hi ha un error:

```
1  org.h2.jdbc.JdbcSQLException: Table "USERS" already exists; SQL statement:
2  CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
```

```
                    username VARCHAR(10) NOT NULL,
                    name  VARCHAR(20) NOT NULL,
                    email VARCHAR(50) NOT NULL,
                    rank INT DEFAULT 0,
                    active BOOLEAN DEFAULT true,
                    created_on TIMESTAMP AS CURRENT_TIMESTAMP NOT NULL)
                        [42101-190]
  at org.h2.message.DbException.getJdbcSQLException(DbException.java:345)
```

Què ha passat? A la funció `createUser`, després de definir la consulta a la base de dades, el que feu és establir una connexió amb la base de dades:

```
Connection conn = dBConnection.getConnection();
```

Quan després crideu la funció `findUserByUsername`, s'executarà la mateixa funció. Recordeu que aquest mètode crea una connexió amb la base de dades de la següent manera:

```
con = DriverManager.getConnection(props.getProperty("DB_URL"),
        props.getProperty("DB_USERNAME"),
        props.getProperty("DB_PASSWORD"));
```

Aquestes propietats estan definides al fitxer de propietats de la base dades, en el cas dels tests (db_test.properties):

```
DB_URL=jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql';
DB_USERNAME=usuari
DB_PASSWORD=passwd
```

La primera vegada que s'executa `dBConnection.getConnection()` s'executarà el fitxer init.sql, que crearà la taula "Users". Afegireu l'usuari a la base de dades, i en executar `findUserByUsername` s'intentarà executar de nou init.sq, però com que ja està creada l'aplicació llençarà aquest error.

Això posa de rellevància un problema del vostre codi: no s'estan reutilitzant les connexions amb la base de dades. Fer una connexió amb la bases de dades és una operació molt cara en termes de recursos: primer s'ha de fer la connexió a través de la xarxa amb la base de dades; s'ha d'inicialitzar una sessió de connexió, que sovint requereix molt de temps de processament per fer l'autenticació d'usuari, establir contextos transaccionals i definir altres aspectes de la sessió que es requereixen per a l'ús de bases de dades subsegüent. En properes seccions veurem com solucionar aquest problema utilitzant *pools* de connexió. La idea és crear una sèrie de connexions amb la base de dades a l'inici de l'aplicació que es compartiran per a totes les operacions que es facin. De moment, però, refactoritzarem el nostre codi de `UserDAO` per permetre reutilitzar les connexions establertes amb la base de dades. Això implica afegir una nova propietat i modificar el mètode `executeQuery`.

```
public class UserDAO {
  private DBConnection dBConnection;
  private Connection connection;

  private List<User> executeQuery(String query) {
    List<User> users = new ArrayList<>();

```

```
    if (getConnection() == null) {
        try {
            setConnection(dBConnection.getConnection());
        } catch (SQLException | IOException e) {
            e.printStackTrace();
        }
    }
    try (
            Statement stmt = getConnection().createStatement();
            ResultSet rs = stmt.executeQuery(query)) {
        while (rs.next()) {
            User user = buildUserFromResultSet(rs);
            users.add(user);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return users;
  }

  public Connection getConnection() {
      return connection;
  }

  public void setConnection(Connection connection) {
      this.connection = connection;
  }

}
```

El que heu fet per solucionar el problema és afegir un atribut a la classe `UserDAO` i, abans de crear-ne un de nou, comprovar si hi ha objecte `Connection` vàlid; si és el cas, el reutilitzeu. Abans d'executar el test heu d'assegurar-vos que tanquem la connexió amb la base de dades per tal de a cada test hi hagi només les dades definides al fitxer init.sql. Un cop executat, us assegurareu que la connexió amb la base de dades es tanca de manera que cada vegada la base de dades es torni a crear de nou.

```
@Before
public void setUp() {
    dBConnection = new DBConnection(connectionProperties);
    userDAO = new UserDAO(dBConnection);
}
 @After
  public void tearDown() throws IOException, SQLException {
      userDAO.getConnection().close();
  }
```

Afegiu ara un altre test per comprovar què passaria en cas que hi hagi un error creant l'usuari. En aquest cas provoquem l'error introduint un nom d'usuari incorrecte:

```
@Test(expected = Exception.class)
public void createUserWithError() throws Exception {
    String username = "sl',’sls";
    String name = "Pete Test";
    String email = "pete@email.com";
    User createdUser = userDAO.createUser(username, name, email);
    Assert.assertNull(createdUser);
}
```

Per actualitzar un registre, l'únic que canvia respecte a l'escriptura és que s'utilitza la clàusula SQL *UPDATE* enlloc d'*INSERT*. Vegeu com podeu modificar el correu

electrònic d'un usuari: afegireu la funció `executeUpdateQuery` a `UserDAO` i escriureu el corresponent test unitari.

```
private int executeUpdateQuery(String query) {
    int result = 0;
    if (getConnection() == null) {
        try {
            setConnection(dBConnection.getConnection());
        } catch (SQLException | IOException e) {
            e.printStackTrace();
        }
    }
    try (
            Statement stmt = getConnection().createStatement()) {
        result = stmt.executeUpdate(query);
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return result;
}
</java>

<code java>
 public User updateUserEmail(String username, String newEmail) throws Exception
       {
  String qry = "UPDATE users "
          + "SET email = '" + newEmail + "' "
          + "WHERE username = '" + username + "' "
          + ";";
  int result = executeUpdateQuery(qry);
  if (result == 0) {
      throw new Exception("Error udpating user");
  }
  return findUserByUsername(username);
}
```

I el test: primer creareu un usuari, modificareu el correu electrònic i comprovareu que és l'única informació que ha canviat:

```
@Test
public void updateUserEmail() throws Exception {
    String username = "testUser";
    String name = "Pete Test";
    String email = "pete@email.com";
    User createdUser = userDAO.createUser(username, name, email);
    Assert.assertNotNull(createdUser);
    Assert.assertEquals(email, createdUser.getEmail());
    User updatedUser = userDAO.updateUserEmail(createdUser.getUsername(), "
        new@email.com");
    Assert.assertEquals(createdUser.getUserId(), updatedUser.getUserId());
    Assert.assertEquals("new@email.com", updatedUser.getEmail());
}
```

Si doneu una ullada a les funcions `updateUserEmail` i `createUser` veureu que són molt similars; l'únic que canvia és la consulta a la base de dades. Això vol dir que aquestes dues funcions són candidates per a la refactorització.

```
public User createUser(String username, String name, String email) throws
    Exception {
  String qry = "INSERT INTO users (username, name, email) VALUES ('"
          + username + "', '"
          + name + "', '"
          + email + "')";
  return createOrUpdateUser(username, qry);
}
```

```
9
10 public User updateUserEmail(String username, String newEmail) throws Exception
       {
11   String qry = "UPDATE users "
12           + "SET email = '" + newEmail + "' "
13           + "WHERE username = '" + username + "' "
14           + ";";
15   return createOrUpdateUser(username, qry);
16 }
17
18 private User createOrUpdateUser(String username, String query) throws Exception
       {
19   int result = executeUpdateQuery(query);
20   if (result == 0) {
21       throw new Exception("Error creating user");
22   }
23   return findUserByUsername(username);
24 }
```

1.6.3 Eliminació de dades

L'eliminació de dades també es considera una actualització de la base de dades; en aquest cas, modificar-la per eliminar informació.

La modificació de l'actualització de la base de dades per eliminar la informació es farà utilitzant el mètode `executeUpdate` de l'objecte `Statement`, per a la qual cosa es pot reutilitzar la funció `createOrUpdateUser` que heu creat en l'apartat "Operacions d'escriptura" i que podeu trobar a l'apartat d'annexos de la unitat.

```
1 public void deleteUser(User user) throws Exception {
2     String query = "DELETE FROM users WHERE user_id = '" + user.getUserId() + "
          ' ";
3     createOrUpdateUser(user.getUsername(), query);
4 }
```

A continuació creeu el test:

```
1 @Test
2   public void deleteUser() throws Exception {
3       String username = "testUser";
4       String name = "Pete Test";
5       String email = "pete@email.com";
6       User createdUser = userDAO.createUser(username, name, email);
7       Assert.assertNotNull(createdUser);
8       userDAO.deleteUser(createdUser);
9       User deletedUser = userDAO.findUserByUsername(username);
10      Assert.assertNull(deletedUser);
11  }
```

1.7 Injecció SQL

Un atac d'injecció SQL és exactament el que el seu nom indica: és quan algú intenta "injectar" el seu codi SQL maliciós a la base de dades d'una altra persona, obligant aquesta base de dades a executar SQL que no estava previst. Això podria arruïnar les seves taules de bases de dades i fins i tot extreure informació valuosa o privada. Vegem-ne un exemple: aquesta és la consulta que hem utilitzat per trobar un usuari a partir del correu electrònic:

```
String qry = "select * from users where email ='" + userEmail + "'";
```

És a dir, que per al correu electrònic test@email.com utilitzaríeu la crida a la funció `findUserByEmail(‘‘test@email.com’’)`, amb la qual cosa la consulta SQL quedaria:

```
select * from users where email ='test@email.com';
```

Un atac d'injecció SQL consistiria a modificar aquest SQL. Com es pot fer? N'hi hauria prou amb modificar l'argument que s'utilitza en cridar la funció. Si executem `findUserByEmail(‘‘test@email.com ' OR 1=1’’)`, la consulta SQL que s'executaria seria:

```
select * from users where email ='test@email.com' OR '1'='1;
```

La segona condició sempre es complirà; per tant, en lloc d'efectuar una consulta que retorni els usuaris amb el correu, retornarà un llistat de tots els usuaris. En aquest cas estaríeu exposant dades sense voler, però el codi SQL inserit podria ser fàcilment modificat per esborrar totes les dades.

Seguint l'exemple de trobar un usuari a partir del correu electrònic:

```
PreparedStatement stmt = connection.prepareStatement("select * from users where
     email =?");
  stmt.setString(1, userEmail);
```

Per poder utilitzar `PreparedStatements` refactoritzem el codi per tal de que `executeQuery`, `executeupdateQuery` i `findUniqueResult` no usin més una cadena de caràcters com a paràmetre. Fixeu-vos també que he introduït un nou mètode `getPreparedStatement` que s'encarrega d'obtenir un objecte `PreparedStatement` ja sigui reutilitzant una connexió existent o creant-ne una de nova:

```
public class UserDAO {
    private DBConnection dBConnection;
    private Connection connection;

    public UserDAO(DBConnection dBConnection) {
        this.dBConnection = dBConnection;
    }

    public List<User> findAllUsers() throws SQLException {
        String qry = "select user_id, username, name, email, rank, active,
             created_on from users";
        PreparedStatement preparedStatement = getPreparedStatement(qry);
        List<User> users = executeQuery(preparedStatement);
        return users;
    }

    public User findUserByEmail(String userEmail) throws Exception {
        String qry = "select * from users where email = ?";
        PreparedStatement preparedStatement = getPreparedStatement(qry);
        preparedStatement.setString(1, userEmail);
        return findUniqueResult(preparedStatement);
    }

    public User findUserByUsername(String username) throws Exception {
```

```
24          String qry = "select * from users where username =?";
25          PreparedStatement preparedStatement = getPreparedStatement(qry);
26          preparedStatement.setString(1, username);
27          return findUniqueResult(preparedStatement);
28      }
29
30      public User createUser(String username, String name, String email) throws
            Exception {
31          String qry = "INSERT INTO users (username, name, email) VALUES (?, ?,
                ?)";
32          PreparedStatement preparedStatement = getPreparedStatement(qry);
33          preparedStatement.setString(1, username);
34          preparedStatement.setString(2, name);
35          preparedStatement.setString(3, email);
36          return createOrUpdateUser(username, preparedStatement);
37      }
38
39      public User updateUserEmail(User user, String newEmail) throws Exception {
40          String qry = "UPDATE users SET email = ?  WHERE user_id = ? ";
41          PreparedStatement preparedStatement = getPreparedStatement(qry);
42          preparedStatement.setString(1, newEmail);
43          preparedStatement.setInt(2, user.getUserId());
44          return createOrUpdateUser(user.getUsername(), preparedStatement);
45      }
46
47
48      private User createOrUpdateUser(String username, PreparedStatement
            preparedStatement) throws Exception {
49          int result = executeUpdateQuery(preparedStatement);
50          if (result == 0) {
51              throw new Exception("Error creating user");
52          }
53          return findUserByUsername(username);
54      }
55
56      public void deleteUser(User user) throws Exception {
57          String qry = "DELETE FROM users WHERE user_id = ?";
58          PreparedStatement preparedStatement = getPreparedStatement(qry);
59          preparedStatement.setInt(1, user.getUserId());
60          createOrUpdateUser(user.getUsername(), preparedStatement);
61      }
62
63      private User findUniqueResult(PreparedStatement preparedStatement) throws
            Exception {
64          List<User> users = executeQuery(preparedStatement);
65          if (users.isEmpty()) {
66              return null;
67          }
68          if (users.size() > 1) {
69              throw new Exception("Only one result expected");
70          }
71          return users.get(0);
72      }
73
74      private List<User> executeQuery(PreparedStatement preparedStatement) {
75          List<User> users = new ArrayList<>();
76
77          try (
78                  ResultSet rs = preparedStatement.executeQuery()) {
79              while (rs.next()) {
80                  User user = buildUserFromResultSet(rs);
81                  users.add(user);
82              }
83          } catch (SQLException e) {
84              e.printStackTrace();
85          }
86          return users;
87      }
88
```

```
89      private PreparedStatement getPreparedStatement(String query) throws
            SQLException {
90          if (getConnection() == null) {
91              try {
92                  setConnection(dBConnection.getConnection());
93              } catch (SQLException | IOException e) {
94                  e.printStackTrace();
95              }
96          }
97          return getConnection().prepareStatement(query);
98      }
99
100
101     private int executeUpdateQuery(PreparedStatement preparedStatement) {
102         int result = 0;
103         if (getConnection() == null) {
104             try {
105                 setConnection(dBConnection.getConnection());
106             } catch (SQLException | IOException e) {
107                 e.printStackTrace();
108             }
109         }
110         try {
111             result = preparedStatement.executeUpdate();
112         } catch (SQLException e) {
113             e.printStackTrace();
114         }
115         return result;
116     }
117
118     private User buildUserFromResultSet(ResultSet rs) throws SQLException {
119         int userId = rs.getInt("user_id");
120         String username = rs.getString("username");
121         String name = rs.getString("name");
122         String email = rs.getString("email");
123         int rank = rs.getInt("rank");
124         boolean active = rs.getBoolean("active");
125         Timestamp timestamp = rs.getTimestamp("created_on");
126         User user = new User(userId, username, name, email, rank, timestamp,
                active);
127         return user;
128     }
129
130     public Connection getConnection() {
131         return connection;
132     }
133
134     public void setConnection(Connection connection) {
135         this.connection = connection;
136     }
137 }
```

A l'apartat d'annexos de la unitat podeu trobar el codi després de la refactorització.

1.8 Què s'ha après?

Heu après que JDBC és la tecnologia que permet a una aplicació Java connectar-se a diferents bases de dades utilitzant una única interfície de programació.

Resumint, heu après a:

- Conèixer la utilitat i l'arquitectura de JDBC
- Usar una base de dades en memòria per fer tests unitaris

- Escriure el codi Java per poder fer operacions CRUD amb la base de dades

Ja esteu preparats per començar les activitats proposades en aquest apartat per tal de poder endinsar-vos en el món de la programació amb Java i bases de dades.

2. Accés a dades amb Java Enterprise Edition

Quan programeu en Java, la forma més simple per accedir a una base és mitjançant JDBC. Per desgràcia, amb JDBC es necessita una gran quantitat de treball manual per convertir els resultats d'una consulta a la base de dades en classes Java.

En el següent fragment de codi es mostra com transformar un resultat de consulta JDBC en un objecte `User`:

```
ResultSet rs = stmt.executeQuery(qry);) {
  while (rs.next()) {
      int userId = rs.getInt("user_id");
      String username = rs.getString("username");
      String name = rs.getString("name");
      String email = rs.getString("email");
      int rank = rs.getInt("rank");
      boolean active = rs.getBoolean("active");
      Timestamp timestamp = rs.getTimestamp("created_on");
      User user = new User(userId, username, name, email, rank, timestamp,
          active);
      users.add(user);
  }
```

Com podeu veure, hi ha molt codi repetitiu per obtenir els resultats dels diferents camps de la base de dades i transformar-los en variables de Java. Penseu que si aquesta classe tingués 30 atributs la quantitat de codi necessària augmentaria considerablement. Aquesta situació seria encara pitjor si a més a més de tenir 30 atributs estigués relacionada amb una altra classe. Per exemple, a l'aplicació "SocIoc" que construireu, un alumne (*User*) pot crear preguntes. Si la classe que representa les preguntes té 10 atributs, cada cop que fem una consulta on es retornin alumnes com a resultat hauríeu de repetir el codi anterior 40 vegades.

Un altre desavantatge de JDBC és la seva portabilitat. La sintaxi de la consulta canviarà d'una base de dades a una altra. Per exemple, amb la base de dades Oracle la instrucció ROWNUM s'utilitza per limitar la quantitat de resultats retornats, mentre que SqlServer és TOP. El fet d'haver d'utilitzar SQL natiu específic per a cada BD fa que el nostre codi estigui acoblat amb el tipus de BD utilitzat. Això és un problema, ja que si en el futur es vol canviar de base de dades tindrem problemes. Hi ha diverses solucions a aquest tipus de problema, però totes passen per mantenir el codi SQL necessari per a cada tipus de BD; això costarà molt de mantenir i és molt propici a errors.

JPA (Java Persistence API) es va crear com una solució als problemes esmentats anteriorment. JPA permet treballar amb les classes de Java, ja que proporciona una capa transparent que s'encarrega de gestionar els detalls específics per a cada BD i permet focalitzar els esforços de desenvolupament en el codi Java. JPA representa una sèrie d'interfícies Java, així com una sèrie d'estàndards i especificacions que defineixen com han de ser les implementacions. JPA per si sol no farà res, necessitarà d'una implementació per poder ser utilitzada. Hi ha

moltes implementacions de JPA disponibles, tant gratuïtes com de pagament; per exemple, Hibernate, OpenJPA i EclipseLink.

La principal característica de JPA és la capacitat d'establir i gestionar les relacions entre les taules de la BD i les classes del codi Java. En Java, l'aplicació es modela través d'objectes, però les bases de dades relacionals només poden emmagatzemar valors escalars, com cadenes de caràcters o enters, i organitzar-los en taules. Sense JPA, tal com hem vist a l'exemple de JDBC, és el programador qui ha de convertir els valors representats en forma d'objectes en valors simples o agrupats per poder-los emmagatzemar a la BD, així com implementar el procés invers per poder extreure les dades. JPA defineix com s'ha de resoldre aquest problema i defineix com s'ha de fer el mapatge d'objectes relacionals (ORM Object-Relational Mapping).

Posarem les bases de l'aprenentatge configurant l'aplicació Java "SocIoc". Explicarem:

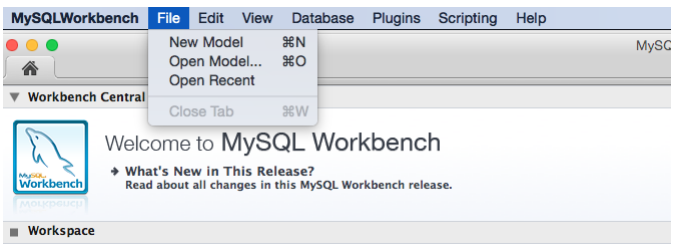
- Creació de la base de dades "SocIoc" a partir d'un model de MySQL Workbench
- Notacions JPA
- Tipus de dades a la BD i correspondència amb Java
- *Persistence units*
- Entorn Glassfish configurant el *datasource* i *connection pool*
- Validació
- Tests unitaris
- Utilitzar la base de dades en memòria H2 per escriure tests unitaris

2.1 "Socloc". Dialogant amb clients amb JPA

Podeu trobar la base de dades "Socloc" al fitxer de MySQL Workbench que teniu disponible als annexos de la unitat.

El primer que haureu de fer serà importar la base de dades "SocIoc", tal com es mostra en la figura 2.1.

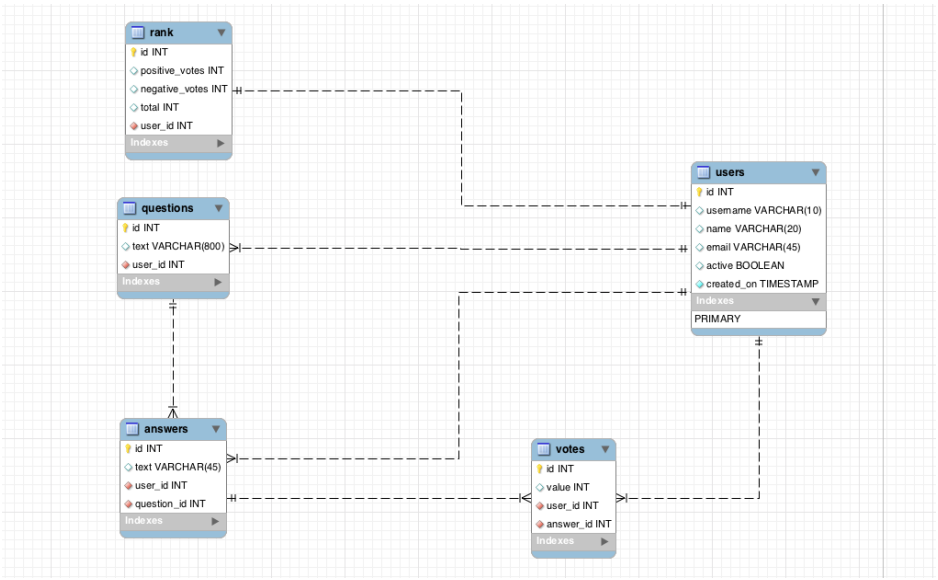
FIGURA 2.1. Importar model de MySQL Workbench



Un cop importat, s'haurà d'haver importat el model entitat relació de la figura 2.2. El model ER de la base de dades "SocIoc" es compon de les següents taules:

- **rank**: serveix per guardar la puntuació que aconsegueix un alumne en base a les votacions que reben les respostes que dóna.
- **questions**: conté les preguntes dels alumnes.
- **answers**: guarda les respostes.
- **votes**: els alumnes que llegeixen una resposta la podran votar de forma positiva o negativa.
- **users**: guarda informació dels alumnes.

FIGURA 2.2. Model ER "SocIoc"



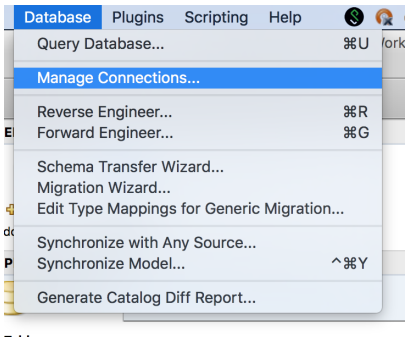
A continuació haureu de connectar-vos al vostre servidor MySQL per tal de crear les taules que hem definit al model ER "SocIoc". El primer pas és crear la base de dades al servidor MySQL. Hi ha diferents formes de fer-ho, però en aquest exemple utilitzarem el client de la línia de comandes. En el cas del servidor que s'està fent servir, aquestes són les dades de connexió:

- usuari: root
- contrasenya: root
- IP del servidor: 192.168.99.100
- port: 32769

```
1 mysql  – –user=root – –password=root – –host=192.168.99.100  – –port=32769
```

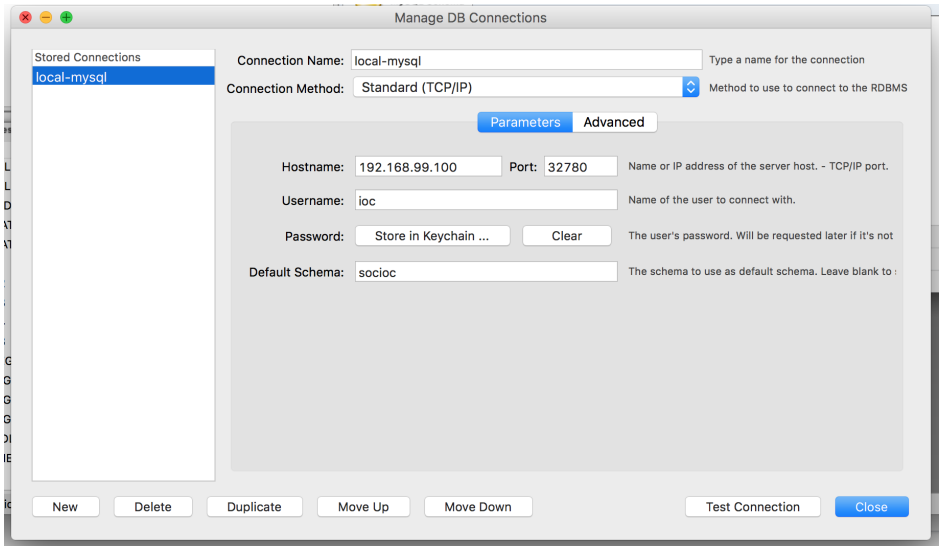
Utilitzant Mysql Workbench, creeu una connexió amb el vostre servidor MySQL. Per fer-ho, seleccioneu *Manage connections*, tal com podeu veure en la figura 2.3.

Figura 2.3. Manage connections



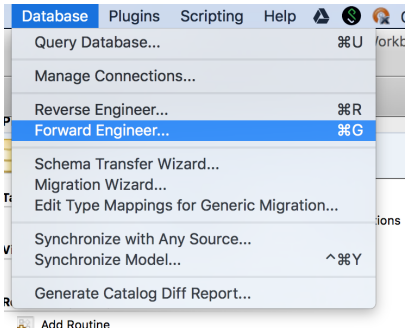
A continuació afegiu les dades de connexió. En la figura 2.4 podeu veure les dades de connexió al servidor local de MySQL.

Figura 2.4. Crear connexió amb DB



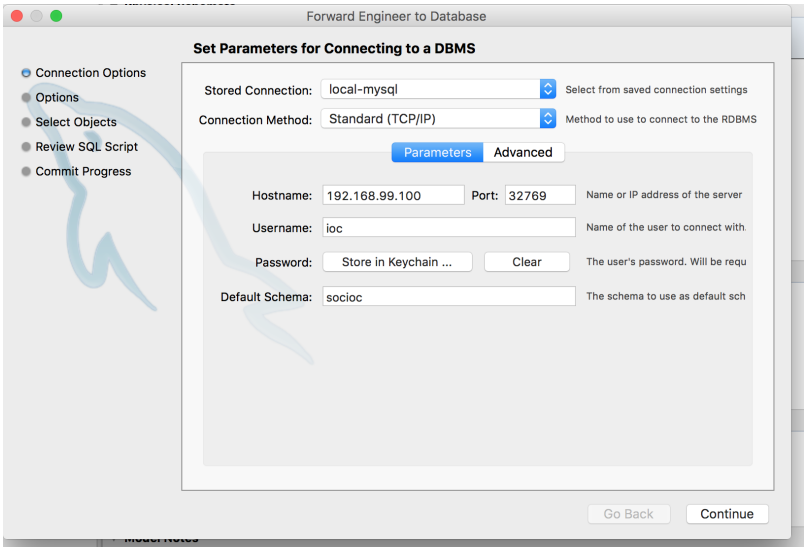
Un cop està configurada la connexió ja es pot passar a exportar el model al servidor. Seleccioneu l'opció *Forward Engineering* del menú *Database* (vegeu la figura 2.5).

Figura 2.5. Exportar taules al servidor



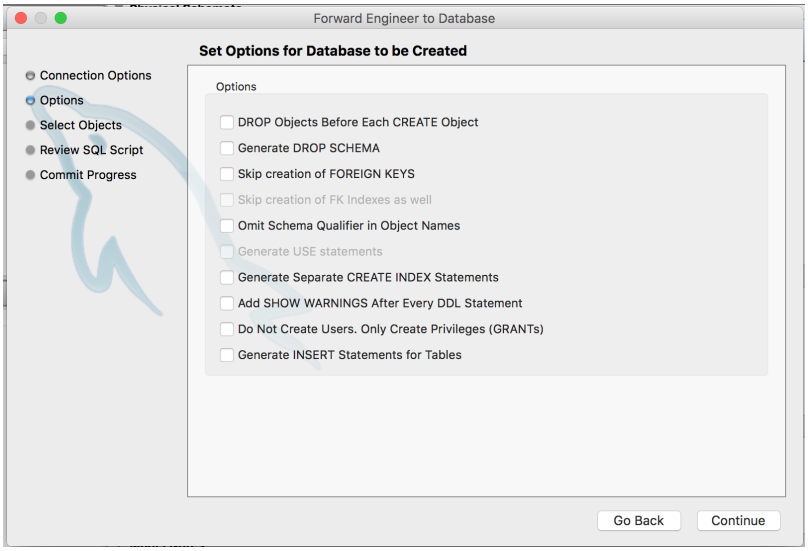
Seleccioneu la connexió que heu creat (vegeu la figura 2.6); en fer clic a *Continue* us demanarà la contrasenya per accedir a la BD.

Figura 2.6. Seleccioneu la connexió amb la BD



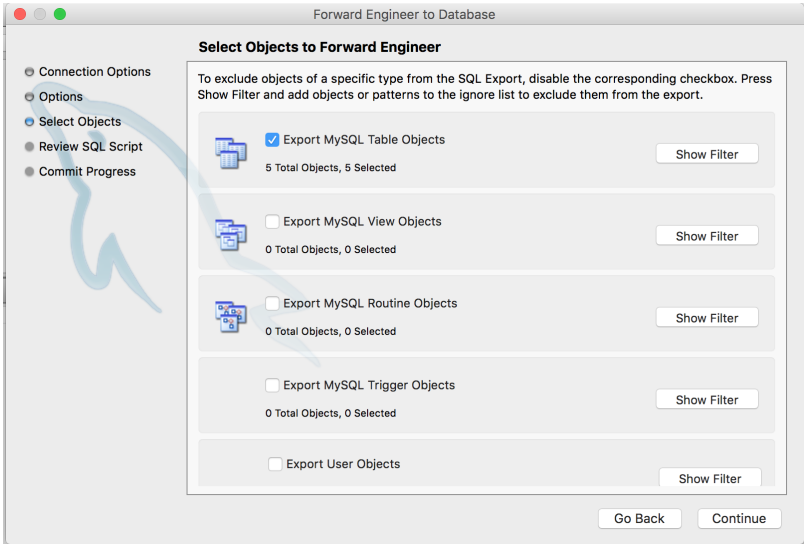
Quant a les opcions, podeu deixar les que hi ha per defecte (vegeu la figura 2.7).

Figura 2.7. Opcions d'exportació



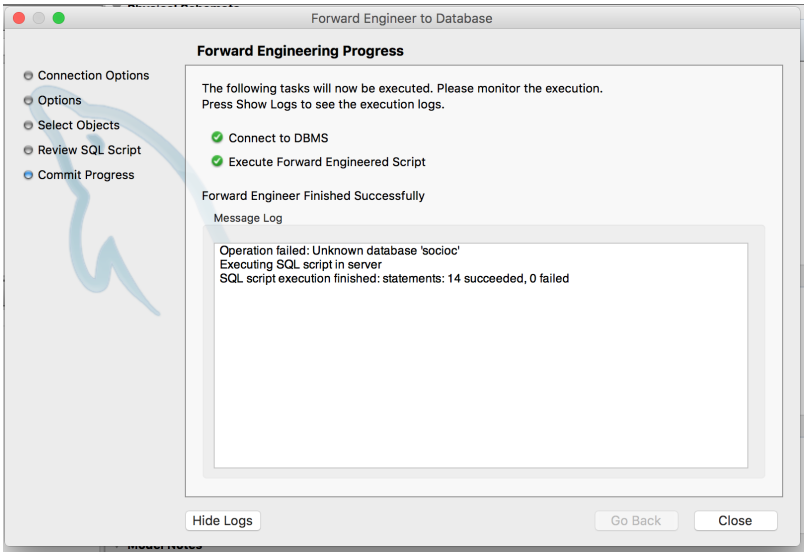
Com que només heu creat taules (vegeu la figura 2.8), només fa falta seleccionar la primera de les opcions, que exportarà les taules.

FIGURA 2.8. Opcions d'exportació



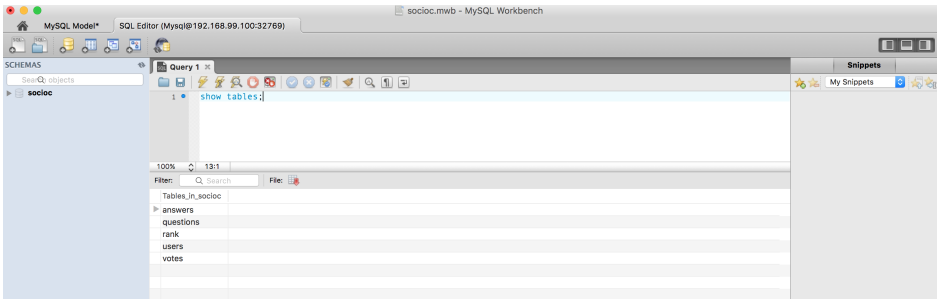
Els dos últims passos són una revisió dels *scripts* de creació dels objectes seleccionats (en el nostre cas, les taules). Si el procés de creació és correcte haureu de veure un missatge de confirmació com el de la figura 2.9.

FIGURA 2.9. Confirmació de la correcta exportació de les taules



Per comprovar que les taules s'han creat adequadament podeu utilitzar l'editor SQL de Workbench. Seleccioneu del menú *Database* l'opció *Query database*; després d'escollir la connexió amb la vostra BD accedireu a l'editor (vegeu la figura 2.10) i podreu executar una consulta perquè es mostrin totes les taules de la BD.

FIGURA 2.10. Editor SQL



2.1.1 Annotacions JPA

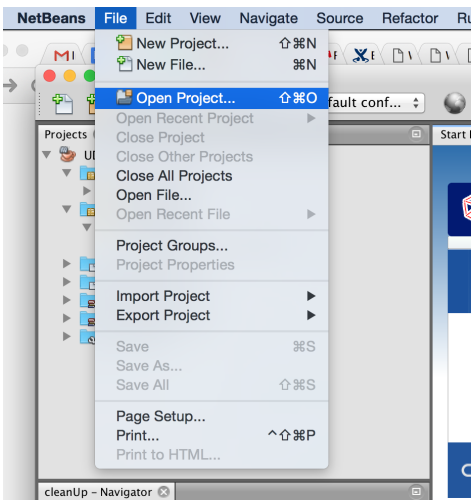
Recordeu que JPA defineix com s'ha de fer el mapatge d'objectes relacionals (ORM Object-Relational Mapping) amb la base de dades. D'aquesta forma, quan desenvolupeu una aplicació, com a programadors us podeu centrar en el codi Java i deixar que JPA s'encarregui dels detalls específics de la base de dades. Començareu creant una classe que representi un usuari de la nostra aplicació i que es pugui utilitzar per emmagatzemar dades.

Per fer això creareu una classe i l'anotareu amb l'anotació `@Entity`. Això transformarà una classe Java simple (POJO, Plain Old Java Object) en una EJB (Enterprise Java Bean). L'ús d'EJB permet gaudir dels serveis prestats pels servidors Java Enterprise Edition (Java EE). Hi ha diferents serveis que proporcionen el seu ús, com *clustering*, transaccionalitat a través de JTA, seguretat i connexions amb base de dades. A més de l'anotació `@Entity` hi ha altres anotacions que permeten definir quins atributs de la classe seran persistents i a quina columna de la taula seran emmagatzemats.

El fitxer de partida per fer annotacions JPA el trobareu als annexos de la unitat.

Un cop descarregat i descomprimit el fitxer de partida, ja el podeu importar; el nom del projecte és "UDF4-02". En la figura 2.11 es mostra on és l'opció per obrir un projecte existent.

FIGURA 2.11. Editor SQL



Netbeans, en detectar el fitxer pom.xml, reconeixerà que és un projecte Maven i començarà a descarregar les dependències.

En el fitxer pom.xml de Maven, les úniques dependències que us fan falta són la del *driver* de la BD H2, de JUnit, `javax.persistence` i `javax.validation`. `javax.persistence` conté les llibreries amb totes les classes que necessitareu per treballar amb JPA. Encara que amb les llibreries de `javax.persistence` no necessitaríeu res més, `javax.validation` us aporta una sèrie d'anotacions que seran molt útils a l'hora de validar les dades que emmagatzemareu a la base de dades.

```
<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.190</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>persistence-api</artifactId>
        <version>1.0.2</version>
    </dependency>
    <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>1.1.0.Final</version>
    </dependency>
</dependencies>
```

Com a programadors d'aplicacions orientades a objectes, el que us interessa és treballar amb objectes que representin les dades, més que haver de treballar directament amb SQL. JPA permet això, utilitza ORM (Object Relational Mapping) per emmagatzemar i recuperar dades de la base de dades a través de l'ús de classes de tipus entitat (*entity classes*). Mitjançant l'anotació `@Entity` es transforma una classe Java simple (POJO, Plain Old Java Object) en una EJB (Enterprise Java Bean). L'ús d'EJB permet gaudir dels serveis prestats pels servidors Java Enterprise Edition (Java EE). Hi ha diferents serveis que proporciona el seu ús, com *clustering*, transaccionalitat a través JTA, seguretat i connexions amb base de dades. A més de l'anotació `@Entity` hi ha altres anotacions que permeten definir quins atributs de la classe seran persistents i a quina columna de la taula seran emmagatzemats. En el següent codi podeu veure les diferents anotacions que farem servir.

```
@Entity
  @Table(name = "users")
  public class User implements Serializable{
    private static final long serialVersionUID = 1L;
    @Id
    @NotNull
    @Column(name = "user_id")
    private Long userId;

    @NotNull
    @Size(max = 30)
    @Column(name = "username")
```

```
13      private String username;
14
15      @NotNull
16      @Size(max = 30)
17      @Column(name = "name")
18      private String name;
19
20      @NotNull
21      @Size(max = 30)
22      @Column(name = "email")
23      private String email;
24
25      @NotNull
26      @Size(max = 30, min = 5)
27      @Column(name = "password")
28      private String password;
29
30      @NotNull
31      @Column(name = "rank")
32      private Integer rank;
33
34      @NotNull
35      @Column(name = "active")
36      private Boolean active;
37
38      @NotNull
39      @Column(name = "created_on")
40      private Timestamp createdOn;
41
42      public User(){}
43
44      public Long getUserId() {
45          return userId;
46      }
47
48      public void setUserId(Long userId) {
49          this.userId = userId;
50      }
51
52      public String getUsername() {
53          return username;
54      }
55
56      public void setUsername(String username) {
57          this.username = username;
58      }
59
60      public String getName() {
61          return name;
62      }
63
64      public void setName(String name) {
65          this.name = name;
66      }
67
68      public String getEmail() {
69          return email;
70      }
71
72      public void setEmail(String email) {
73          this.email = email;
74      }
75
76      public String getPassword() {
77          return password;
78      }
79
80      public void setPassword(String password) {
81          this.password = password;
82      }
```

```
83
84      public Integer getRank() {
85          return rank;
86      }
87
88      public void setRank(Integer rank) {
89          this.rank = rank;
90      }
91
92      public Boolean getActive() {
93          return active;
94      }
95
96      public void setActive(Boolean active) {
97          this.active = active;
98      }
99
100     public Timestamp getCreatedOn() {
101         return createdOn;
102     }
103
104     public void setCreatedOn(Timestamp createdOn) {
105         this.createdOn = createdOn;
106     }
107 }
```

L'anotació `@Entity` indica que serà una classe de tipus entitat i l'anotació `@Table` permet indicar quin és el nom de la taula que estarà lligada a aquesta classe. Un altre aspecte que és important destacar és que és sempre una bona idea fer que una classe de tipus entitat sigui serialitzable, per tal que la classe pugui ser passada per valor i no per referència. Això s'aconsegueix fent que la classe implementi la interfície `java.io.Serializable` i afegint l'atribut estàtic `serialVersionUID`. Sense entrar en molts detalls, `serialVersionUID` s'utilitza per comprovar que els objectes serialitzats i desserialitzats són compatibles.

Una classe del tipus entitat necessita tenir un constructor sense arguments i atributs per a cada una de les columnes de la taula. Les anotacions JPA que hem utilitzat les podeu veure en la taula 2.1.

TAULA 2.1. Annotacions JPA

Anotació	Descripció
`@Entity`	Transforma un POJO en una classe de tipus entitat per tal de poder utilitzar-la amb els serveis JPA.
`@Table` (opcional)	Especifica el nom de la taula de la base de dades associada amb l'entitat.
`@Id`	Designa un o més dels atributs de l'entitat com a la clau principal de la taula.
`@Column`	Associa un atribut que es vol emmagatzemar amb el nom d'una columna de la taula.

Hi ha altres anotacions que hem utilitzat i que no apareixen en la llista. Intenteu contestar a les següents preguntes abans de continuar:

- Quines són les anotacions que hem utilitzat i que no pertanyen a l'API de JPA?
- A quina API pertanyen?

- Per a què creieu que serveixen?

Les anotacions que no són part de JPA i que hem utilitzat (taula 2.2) són part de l'API `javax.validation` i serveixen per validar les dades que guardarem a la base de dades abans d'intentar guardar-les.

TAULA 2.2. Annotacions de validació

Anotació	Descripció
`@Size`	Permet especificar la longitud màxima i/o mínima d'una variable de tipus *string*.
`@NotNull`	Estableix que l'atribut not pot tenir un valor *null*.

Com ja hem vist, JPA és una especificació que indica com es transformen classes en entitats i com aquestes es poden fer servir per llegir i escriure en una base de dades. És una especificació però no una implementació, és a dir, defineix com han de ser les operacions però no les implementa. Això vol dir que amb l'exemple de la classe `User` que hem vist no podem fer cap operació amb la base de dades. Per poder fer-ho hi ha dues opcions: utilitzar Persistence Units (unitats de persistència) o emprar Enterprise Java Beans, que requerirà de l'ús d'un servidor d'aplicacions com pot ser Glassfish.

2.1.2 Unitats de persistència

Les unitats de persistència són les peces que enllacen l'entitat que hem definit amb la base de dades. Poden usar un *pool* de connexions amb la BD definides al servidor d'aplicacions o, com en l'exemple següent, una connexió JDBC. El que es aconseguir és crear una unitat de persistència que pugueu utilitzar en els tests unitaris. Obriu el codi que crearà el projecte UDF4-02. Al fitxer src/test/resources/META-INF/persistence.xml trobareu la configuració de la unitat de persistència.

Als annexos de la unitat trobareu l'arxiu amb el codi per crear el projecte UDF4-02.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="InMemoryH2PersistenceUnit" transaction-type="
        RESOURCE_LOCAL">
        <class>org.ioc.daw.user.User</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
                "/>
            <property name="javax.persistence.jdbc.user" value="username"/>
            <property name="javax.persistence.jdbc.password" value="password"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:
                socioc_db"/>
        </properties>
    </persistence-unit>
</persistence>
```

El fitxer XML que defineix la unitat de persistència és on resideix informació que necessita JPA per saber com connectar-se a la BD. Aquest fitxer pot contenir la configuració de com connectar-se a múltiples BD. Cadascuna d'aquestes configuracions indicarà el tipus de transacció que s'utilitzarà a l'hora de fer les operacions amb la BD que serà JTA o RESOURCE_LOCAL.

A *persistence-unit name*= s'indica el nom de la unitat de persistència, que serà utilitzat a l'aplicació per obtenir una referència a la configuració que defineix com ens connectem a la BD. El tipus de transacció indica si s'utilitzarà Java Transaction API (JTA) *entity managers* (gestors d'entitats) per ser utilitzats en un servidor d'aplicacions o bé *entity managers* locals que no necessiten de cap servidor d'aplicacions, i que és el tipus que us interessa per als vostres tests unitaris.

`<class>` permet definir quines classes (en el vostre cas només n'hi ha una, de moment) seran mapejades amb la BD. Després es defineixen els elements que permeten la connexió amb la BD. En aquest cas, nom d'usuari i contrasenya per accedir a la BD i l'*string* de connexió que defineix com accedir a la base de dades.

Un cop heu establert com es farà la connexió amb la base de dades, ja podeu escriure un test unitari.

```
1  public class UserTest {
2      private EntityManager entityManager;
3      private EntityTransaction entityTransaction;
4
5      @Before
6      public void setUp() {
7          entityManager  = Persistence.createEntityManagerFactory("
               InMemoryH2PersistenceUnit").createEntityManager();
8          entityTransaction = entityManager.getTransaction();
9      }
10
11     @After
12     public void cleanUp() {
13         entityManager.close();
14     }
15
16     @Test
17     public void findAllUsers(){
18
19         User user = new User();
20         user.setActive(true);
21         user.setCreatedOn(new Timestamp(new Date().getTime()));
22         user.setEmail("test@test.com");
23         user.setName("Jane");
24         user.setPassword("password");
25         user.setRank(100);
26         user.setUsername("jdoe");
27         user.setUserId(23L);
28
29         entityTransaction.begin();
30         entityManager.persist(user);
31         entityTransaction.commit();
32
33         Query query = entityManager.createNativeQuery("select * from users",
               User.class);
34         List<User> userList = query.getResultList();
35         Assert.assertEquals(1, userList.size());
36         Assert.assertEquals("jdoe", userList.get(0).getUsername());
37     }
38
```

```
}
```

Amb `Persistence.createEntityManagerFactory` definiu quina serà la unitat de persistència que necessitareu per crear l'`EntityManager`. Després creeu un objecte per fer transaccions amb la BD; és part del mètode `setUp`, perquè aquest objecte l'utilitzareu en tots els tests. Al test `findAllUsers` primer creeu un objecte `User` (que, recordeu, serà una entitat), inicieu la transacció amb la BD, salveu l'objecte amb `entityManager.persist(user);` i finalment dieu a l'objecte que gestiona les transaccions que apliqui tots els canvis a la BD. Fixeu-vos que no serà fins a `entityTransaction.commit()` que els canvis s'escriuran a la base de dades. Finalment, el test comprova que a la taula "Users" de la BD només hi ha una entrada i que correspon a l'usuari que acabeu de crear. Executeu el test i obtindreu el següent error:

```
javax.persistence.PersistenceException: No resource files named META-INF/
    services/javax.persistence.spi.PersistenceProvider were found. Please make
     sure that the persistence provider jar file is in your classpath.

  at javax.persistence.Persistence.findAllProviders(Persistence.java:167)
```

Quin és el motiu pel qual teniu aquest error?

Com ja heu vist, JPA només és una especificació, però no una implementació. L'error vol dir exactament això: quan s'intenta executar el codi no es troba cap llibreria (`PersistenceProvider`) que implementi JPA, per la qual cosa resulta impossible fer cap operació amb la base de dades. Ho solucionareu utilitzant la implementació EclipseLink. No entrarem en detalls sobre EclipseLink, ja que més endavant utilitzareu Hibernate com a implementació.

Per incloure EclipseLink al nostre projecte, el primer serà modificar el fitxer de dependències pom.xml.

```
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.5.0</version>
  </dependency>
```

A continuació haureu de modificar la unitat de persitència per indicar quina implementació utilitzareu:

```
<persistence-unit name="InMemoryH2PersistenceUnit" transaction-type="
    RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>org.ioc.daw.user.User</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
                "/>
            <property name="javax.persistence.jdbc.user" value="username"/>
            <property name="javax.persistence.jdbc.password" value="password"/>
            <property name="javax.persistence.jdbc.url" value="
                jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql'
                "/>
            <property name="javax.persistence.schema-generation.database.action
                " value="drop-and-create"/>
            <property name="javax.persistence.schema-generation.create-source"
                value="metadata"/>
```

```
11              <property name="javax.persistence.schema-generation.drop-source"
                    value="metadata"/>
12          </properties>
13      </persistence-unit>
```

A la línia 2 s'ha afegit quina serà la implementació del `PersistenceProvider`. A la línia 8, on s'indica l'*string* de connexió, afegim també que s'executi un *script* on creareu la taula "Users", que utilitzareu per emmagatzemar els usuaris del nostre sistema. El fitxer src/test/resources/init.sql té el següent contingut:

```
1  CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
2                     username VARCHAR(30) NOT NULL,
3                     name  VARCHAR(20) NOT NULL,
4                     email VARCHAR(50) NOT NULL,
5                     password VARCHAR(50) NOT NULL,
6                     rank INT DEFAULT 0,
7                     active BOOLEAN DEFAULT true,
8                     created_on TIMESTAMP AS CURRENT_TIME)
```

Torneu a executar el test `findAllUsers`, que aquest cop hauria de passar. Un problema d'aquest codi és que la lògica per emmagatzemar i recuperar usuaris està al test. És important que aquest codi estigui a la seva pròpia classe per tal que es pugui utilitzar en tota l'aplicació. Creareu una classe, `UserService`, que s'encarregarà d'aquesta funció i que tindrà la lògica necessària per guardar un usuari a la base de dades, modificar-lo i buscar usuaris pel seu nom. Començareu creant una interfície que definirà les operacions relacionades amb la BD. Programar utilitzant interfícies té avantatges, com heu pogut veure amb JPA. Es defineix una interfície i després es pot canviar la implementació; així, si canviéssiu de BD i per exemple escollíssiu una BD NoSQL on no es pugui utilitzar JPA només hauríeu de preocupar-vos de canviar la implementació. Definiu la interfície `org.ioc.daw.user.UserDAO` que us permeti guardar usuaris a la BD, esborrar-los, modificar-los i trobar un usuari pel seu nom.

```
1  public interface UserService {
2      public void create(User user);
3      public void edit(User user);
4      public void remove(User user);
5      public User findUserByUsername(String username);
6    }
```

A continuació creeu la seva implementació:

```
1  public class UserServiceImpl implements UserService {
2      private EntityManager entityManager;
3
4      public UserServiceImpl(EntityManager entityManager) {
5          this.entityManager = entityManager;
6      }
7
8      @Override
9      public void create(User user) {
10         entityManager.persist(user);
11     }
12
13     @Override
14     public void edit(User user) {
15         entityManager.merge(user);
16     }
17
```

```
18      @Override
19      public void remove(User user) {
20          entityManager.remove(user);
21      }
22
23      @Override
24      public User findUserByUsername(String username) {
25        return (User) entityManager.createQuery("select object(o) from User o " +
26            "where o.username = :username")
27            .setParameter("username", username)
28            .getSingleResult();
29      }
30  }
```

Els mètodes `create`, `remove` i `edit` defineixen les operacions JPA per guardar, actualitzar i esborrar entitats de la BD. La part més interessant és la del mètode `findUserByUsername`. Utilitzeu `createQuery`, i en aquest cas feu servir una consulta parametritzada on indiqueu la classe de l'objecte que retornarà la consulta, el paràmetre que passareu (`username`) i que només retornarà un resultat, ja que només podeu tenir un usuari amb un nom determinat. El test el podeu implementar de la següent manera (en aquest cas emmagatzemeu dos usuaris):

```
1   public class UserServiceTest {
2       private EntityManager entityManager;
3       private EntityTransaction entityTransaction;
4       private UserService userService;
5
6       @Before
7       public void setUp() {
8           entityManager  = Persistence.createEntityManagerFactory("
                InMemoryH2PersistenceUnit").createEntityManager();
9           userService = new UserServiceImpl(entityManager);
10          entityTransaction = entityManager.getTransaction();
11      }
12
13      @After
14      public void cleanUp() {
15          entityManager.close();
16      }
17
18      @Test
19      public void findAllUsers(){
20          String username = "jdoe";
21          User user = new User();
22          user.setActive(true);
23          user.setCreatedOn(new Timestamp(new Date().getTime()));
24          user.setEmail("test@test.com");
25          user.setName("Jane");
26          user.setPassword("password");
27          user.setRank(100);
28          user.setUsername(username);
29          User user1 = new User();
30          user1.setActive(true);
31          user1.setCreatedOn(new Timestamp(new Date().getTime()));
32          user1.setEmail("test1@test.com");
33          user1.setName("Joe");
34          user1.setPassword("password");
35          user1.setRank(100);
36          user1.setUsername("joeTest");
37
38          entityTransaction.begin();
39          userService.create(user);
40          userService.create(user1);
41          entityTransaction.commit();
42
43          User userFromDB = userService.findUserByUsername(username);
```

```
44          Assert.assertNotNull(userFromDB);
45          Assert.assertEquals("jdoe", userFromDB.getUsername());
46          Assert.assertEquals("test@test.com", userFromDB.getEmail());
47          Assert.assertNotNull(userFromDB.getUserId());
48      }
49  }
```

Fixeu-vos que en aquest test no heu donat valor a `userId`, però tal com testeja `Assert.assertNotNull(userFromDB.getUserId());` sí que té un valor. Quina anotació JPA s'ha d'afegir a l'atribut `userId` de la classe `User` que generarà automàticament un valor? La resposta és `GeneratedValue`.

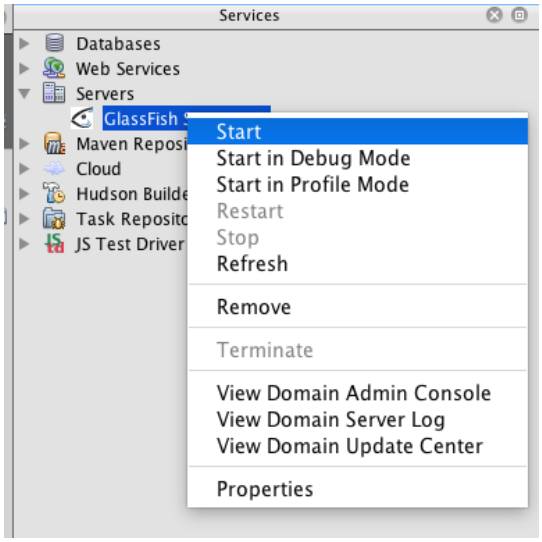
```
1  @Id
2    @NotNull
3    @GeneratedValue
4    @Column(name = "user_id")
5    private Long userId;
```

2.2 Servidor d'aplicacions Glassfish

Quan vulgueu desplegar la vostra aplicació, utilitzar EclipseJPA com a implementació de JPA no serà suficient. Necessiteu un servidor d'aplicacions que permeti desplegar aplicacions Java EE. Glassfish és el servidor d'aplicacions de referència per a aplicacions Java EE i inclou les tecnologies EJB, JPA, JSF (Java Server Faces) i JMS (Java Messaging System). Netbeans inclou per defecte un servidor Glassfish. A la pestanya *Serveis*, tal com podeu veure en la figura 2.11, ja hi ha un servidor Glassfish configurat.

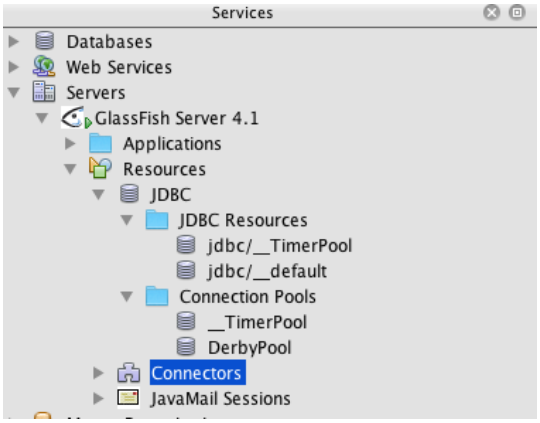
En algunes versions del servidor Glassfish hi ha un *bug* no solucionat en afegir un *pool* de connexions. Utilitzeu la versió 4.0 o superior, que podeu descarregar de: glassfish.java.net/download-archive.html.

FIGURA 2.12. Glassfish



Arranqueu el servidor; després d'uns segons estarà funcionant, i si feu clic a `Resources` veureu que hi ha dues carpetes, *JDBC Resources* i *Connection Pools* (vegeu la figura 2.13).

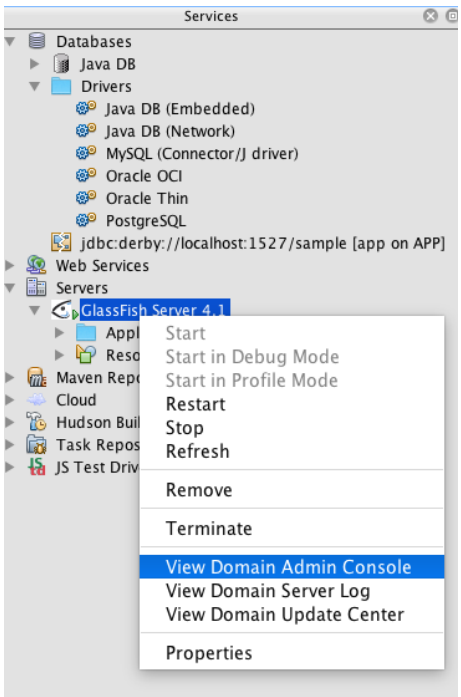
FIGURA 2.13. 'Resources and connection pools'



2.2.1 'Pool' de connexions

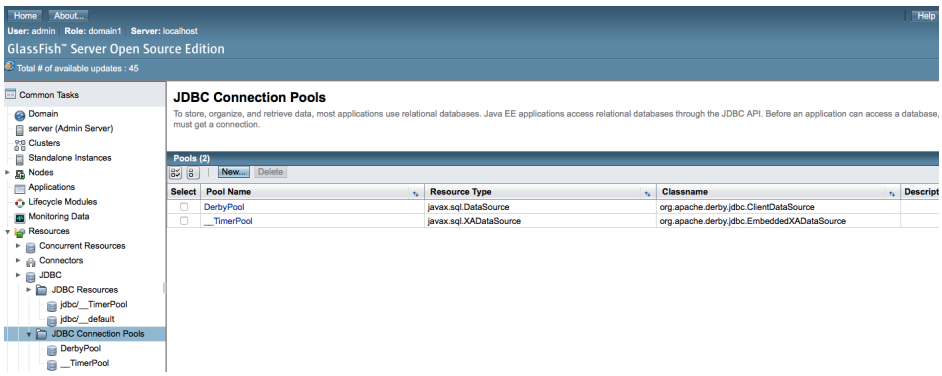
Un *pool* de connexions és un sèrie de connexions amb la BD que es guarden en *cache* i que poden ser reutilitzades per diferents consultes que es facin a la BD. El motiu d'utilitzar-les és que milloren notablement el rendiment de l'aplicació i la fan molt més ràpida. La gestió d'obrir i tancar connexions amb al BD cada cop que es necessiti guardar o recuperar dades és molt costosa, especialment en aplicacions dinàmiques on el contingut es genera dinàmicament a partir de les dades de la BD. La forma de funcionar és que quan es crea una connexió es posa al *pool*, de manera que quan es torni a necessitar la connexió no s'ha de tornar a establir. El que volem fer és establir un *pool* de connexions amb la base de dades MySQL. Per fer-ho, un cop el servidor Glassfish ha arrencat, accediu a la consola d'administració (vegeu la figura 2.14).

FIGURA 2.14. 'Resources and connection pools'



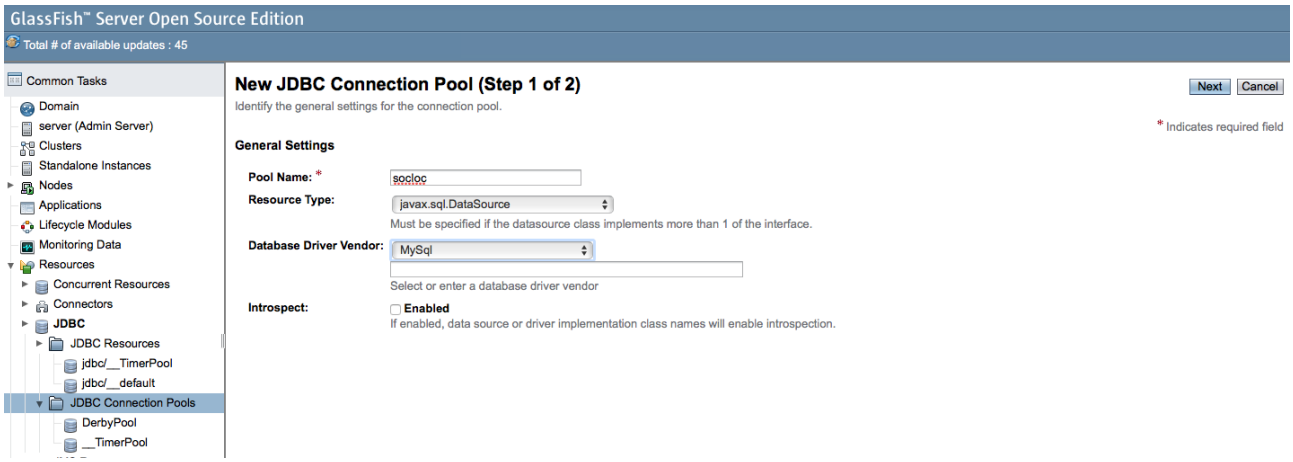
La consola d'administració és una aplicació web, així que s'obrirà una finestra al vostre navegador. Navegueu a `Resources/JDBC/JDBC Connection Pools` i veureu els *pools* de connexions creats per defecte (vegeu la figura 2.15).

FIGURA 2.15. 'Resources and connection pools'



Feu clic a *New* i podreu crear el *pool* de connexions. Es fa en dues parts. En la primera part es dóna nom al *pool* de connexions i se selecciona el tipus de recurs i el *driver* que s'utilitzarà per connectar-se amb la base de dades (vegeu la figura 2.16).

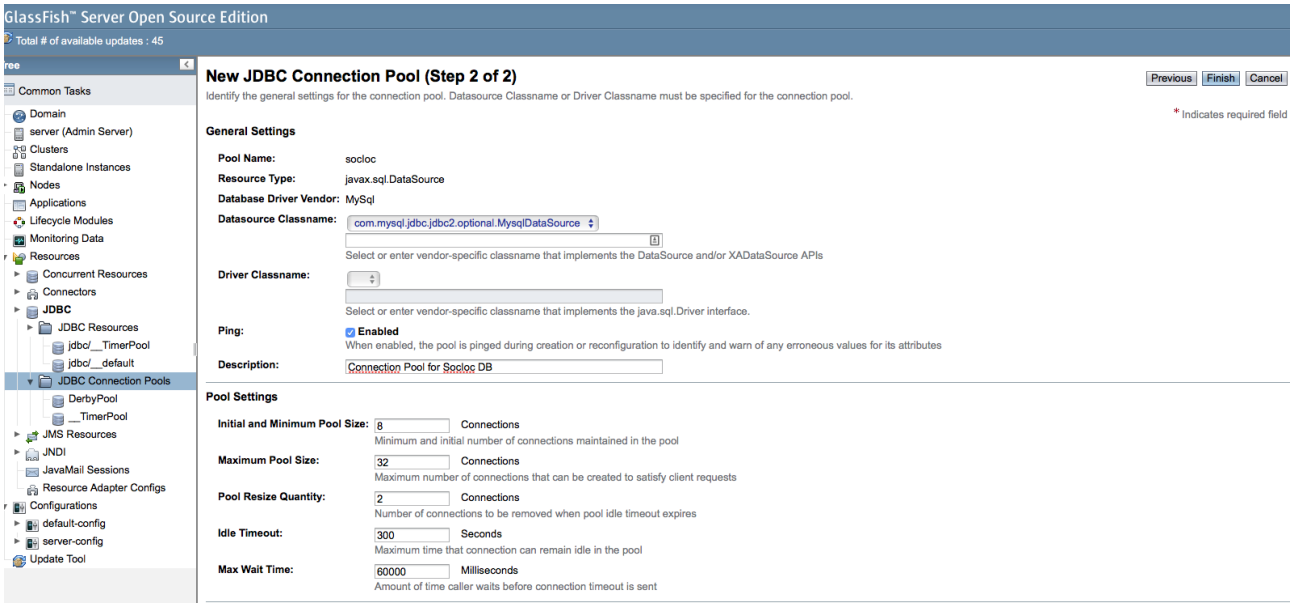
FIGURA 2.16. Configuració del 'pool' de connexions I



A continuació es configuren els paràmetres del *pool* de connexions (vegeu la figura 2.17). Els que hi ha per defecte ja aniran bé, així que no els canviareu. Cal destacar els següents:

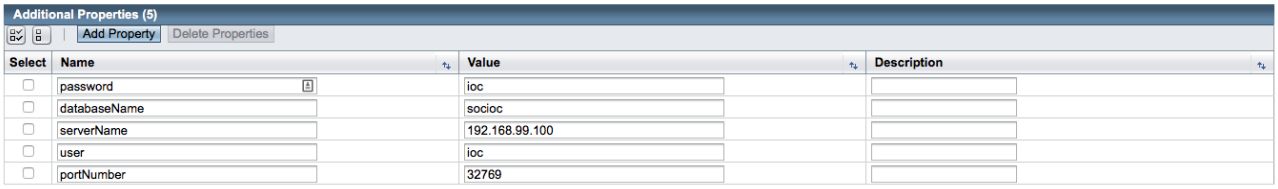
- Initial and Minimum Pool Size: nombre mínim de connexions amb la BD que mantindrem al *pool*.
- Maximum Pool Size: nombre màxim de connexions simultànies que mantindrem al *pool*.
- Pool Resize Quantity: nombre de connexions que es trauran del *pool* quan no hi hagi activitat durant més del temps establert pel paràmetre `Idle Timeout`.
- Idle Timeout: temps màxim que una connexió pot estar inactiva.

FIGURA 2.17. Configuració del 'pool' de connexions II



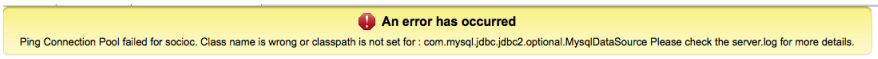
Els únics valors que heu de canviar són els relatius al nom de la base de dades, l'usuari, la contrasenya, la IP i el port que utilitza el servidor MySQL, que trobareu al final de la pàgina web (vegeu la figura 2.18).

FIGURA 2.18. Configuració del 'pool' de connexions III



Un cop configurat, assegureu-vos que la connexió amb la BD funciona utilitzant el botó *Ping*. Si obteniu l'error mostrat en la figura 2.19 és perquè el servidor Glassfish no pot accedir al *driver* de connexió amb MySQL.

FIGURA 2.19. Connector MySQL no trobat



En aquest cas feu el següent:

1. Descarregueu-vos el connector de dev.mysql.com/downloads/connector/j/5.1.html.

2. Descomprimiu-lo i copieu l'arxiu mysql-connector-java-5.1.40-bin.jar a */instalacio-del-servidor-glassfish/glassfish-4.1/glassfish/domains/domain1/lib/* . Glassfish estarà instal·lat en el directori on hagueu instal·lat Netbeans. Per veure on està instal·lat exactament mireu les propietats del servidor Glassfish (*Installation Location*) (vegeu la figura 2.20 i la figura 2.21).

FIGURA 2.20. Directori d'instal·lació de Glassfish

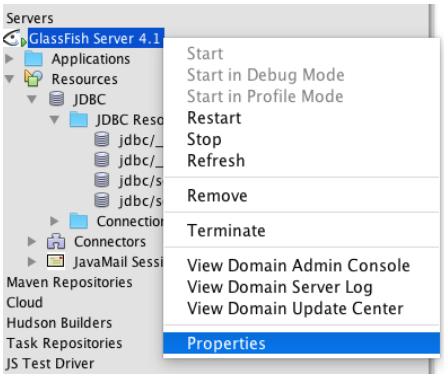
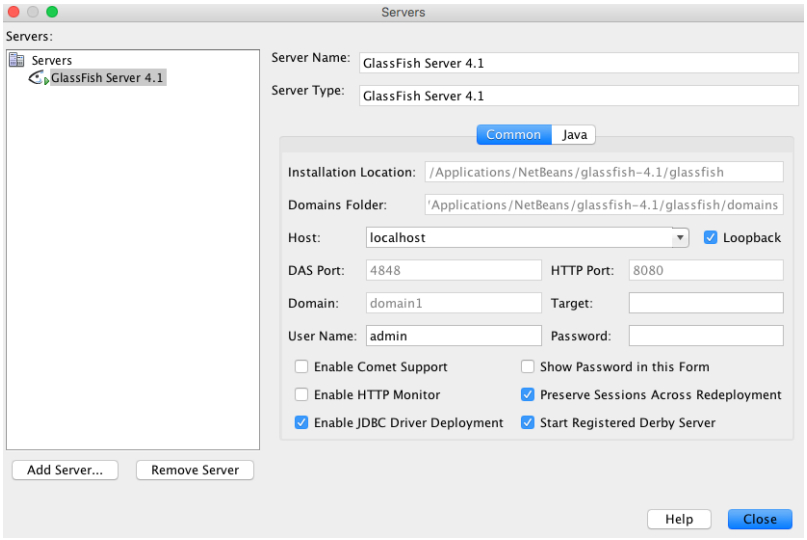


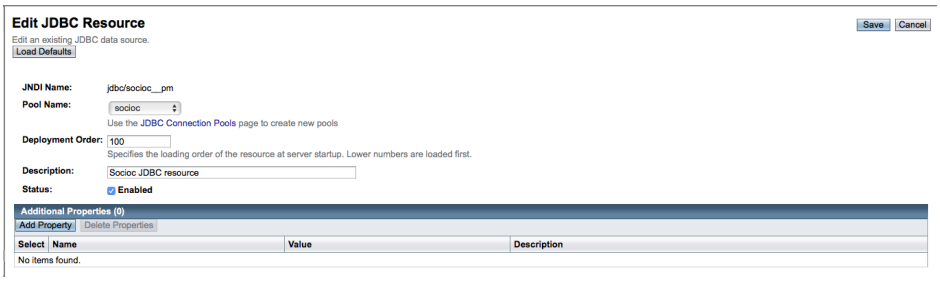
FIGURA 2.21. Directori d'instal·lació de Glassfish



2.2.2 Recursos JDBC

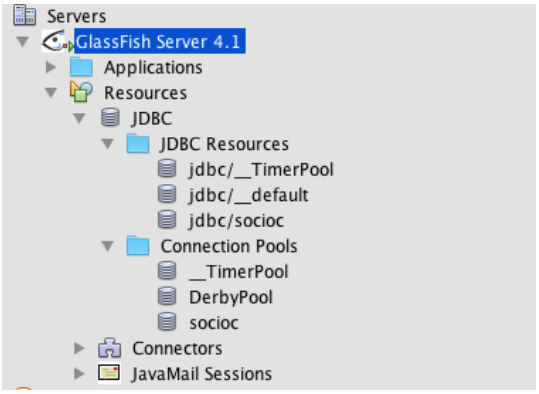
Un recurs JDBC proporciona a una aplicació la forma de connectar-se a una base de dades. Típicament, l'administrador del servidor Glassfish crearà un recurs JDBC que establirà quin dels *pools* de connexions utilitzarà l'aplicació. Cada recurs JDBC creat al servidor d'aplicacions tindrà un únic identificador JNDI (Java Naming and Directory Interface). JNDI és un servei ofert pels servidors d'aplicacions Java EE que permet als clients (en aquest cas, l'aplicació que estem desenvolupant) descobrir serveis i objectes utilitzant un nom. L'objecte que la vostra aplicació descobrirà serà l'objecte que representa el *pool* de connexions. A l'hora d'escollir el nom per al recurs JDBC es pot triar qualsevol, però per convenció s'utilitza jdbc/nom_del_recurs_pm. Per crear un recurs JDBC aneu a la consola d'administració de Glassfish i navegueu a *Resources/JDBC/JDBC Resources*. Amb el botó *New*, creeu un nou recurs i afegiu-hi les dades, tal com podeu veure en la figura 2.22.

FIGURA 2.22. Creació d'un recurs JDBC



Si refresqueu la informació del servidor Glassfish a Netbeans hauríeu de veure tant el recurs JDBC creat com el *pool* de connexions (vegeu la figura 2.23).

FIGURA 2.23. Recurs JDBC i 'pool' de connexions



2.2.3 Connectar l'aplicació "Socloc" amb el recurs JDBC

Un cop ja teniu el *pool* de connexions i el recurs JDBC creat, ja podeu fer que la vostra aplicació els utilitzi. Per fer-ho haureu de definir una unitat de persistència que especifiqui el recurs JDBC que utilitzareu. A partir del codi , importeu el projecte UDF4-02 i editeu el fitxer persistence.xml i amb el següent contingut:

Trobareu el codi per connectar l'aplicació "Socloc" amb el recurd JDBC als annexos de la unitat.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
3               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6      <persistence-unit name="InMemoryH2PersistenceUnit" transaction-type="
           RESOURCE_LOCAL">
7          <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
8          <class>org.ioc.daw.user.User</class>
9          <properties>
10             <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
                   "/>
11             <property name="javax.persistence.jdbc.user" value="username"/>
12             <property name="javax.persistence.jdbc.password" value="password"/>
13             <property name="javax.persistence.jdbc.url" value="
                   jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql'
                   "/>
14             <property name="javax.persistence.schema-generation.database.action
                   " value="drop-and-create"/>
15             <property name="javax.persistence.schema-generation.create-source"
                   value="metadata"/>
16             <property name="javax.persistence.schema-generation.drop-source"
                   value="metadata"/>
```

```
        </properties>
    </persistence-unit>
    <persistence-unit name="MysqlResourceSocIocPersistence" transaction-type="
        JTA">
        <jta-data-source>jdbc/socioc__pm</jta-data-source>
        <class>org.ioc.daw.user.User</class>
        <properties>
            <property name="eclipselink.logging.level" value="FINEST"/>
        </properties>
    </persistence-unit>
</persistence>
```

Hi ha diversos canvis. El primer és que la definició de la nova unitat de persistència `MysqlResourceSocIocPersistence` és molt més simple. Penseu que ara tots els detalls de la connexió estan gestionats per Glassfish, així que vosaltres només us heu de preocupar d'indicar quin serà el recurs JDBC que utilitzareu. L'única propietat que s'ha afegit a la línia 23 és `eclipselink.logging.level`, que indica el nivell de *logging* de l'aplicació i que serà útil per solucionar problemes. Al test unitari anterior, la gestió de les transaccions era part del test.

```
entityTransaction.begin();
    userService.create(user);
    userService.create(user1);
    entityTransaction.commit();
```

Això no hauria de formar part del test, ja que el que us interessa és que sigui la classe que fa les consultes a la BD la que s'encarregui de gestionar les transaccions. A més a més, en utilitzar un servidor d'aplicacions les vostres classes seran ara entitats de ple dret, per la qual cosa es pot aprofitar per simplificar el codi utilitzant anotacions. Per fer-ho haureu d'afegir la següent dependència al fitxer pom.xml.

```
<dependency>
      <groupId>javax.ejb</groupId>
      <artifactId>javax.ejb-api</artifactId>
      <version>3.2</version>
  </dependency>
```

Vegeu com s'ha de modificar la classe `UserServiceImpl`:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class UserServiceImpl implements UserService {
    @PersistenceContext(unitName = "MysqlResourceSocIocPersistence")
    private EntityManager entityManager;

    @Resource
    private EJBContext context;

    @Override
    public void create(User user) {
        UserTransaction utx = context.getUserTransaction();
        try {
            utx.begin();
            entityManager.persist(user);
            utx.commit();
        } catch (Exception e) {
            e.printStackTrace();
            try {
                utx.rollback();
            } catch (Exception e1) {
                e1.printStackTrace();
            }
        }
```

```
25      }
26
27      @Override
28      public void edit(User user) {
29          UserTransaction utx = context.getUserTransaction();
30          try {
31              utx.begin();
32              entityManager.merge(user);
33              utx.commit();
34          } catch (Exception e) {
35              try {
36                  utx.rollback();
37              } catch (Exception e1) {
38                  e1.printStackTrace();
39              }
40              e.printStackTrace();
41          }
42      }
43
44      @Override
45      public void remove(User user) {
46          UserTransaction utx = context.getUserTransaction();
47          try {
48              utx.begin();
49              entityManager.remove(user);
50              utx.commit();
51          } catch (Exception e) {
52              try {
53                  utx.rollback();
54              } catch (Exception e1) {
55                  e1.printStackTrace();
56              }
57              e.printStackTrace();
58          }
59      }
60
61      @Override
62      public User findUserByUsername(String username) {
63        return (User) entityManager.createQuery("select object(o) from User o " +
64          "where o.username = :username")
65          .setParameter("username", username)
66          .getSingleResult();
67      }
```

L'anotació `Stateless` transforma un POJO en una EJB de sessió que adquireix la capacitat d'executar operacions en un servidor d'aplicacions (Glassfish, en el vostre cas). `TransactionManagement` és necessària per permetre fer operacions amb la BD a través del servidor d'aplicacions. En aquest cas, farà que hi hagi disponible al context de EJB (`EJBContext`) la classe `UserTransaction`, que utilitzareu per indicar quan comencen i acaben les transaccions amb la BD. L'anotació `PersistenceContext` permet declarar quina serà la unitat de persistència que utilitzareu per indicar com l'EJB `UserServiceImpl` s'ha de connectar amb la BD. Finalment, a cadascun dels mètodes que s'encarreguen de modificar dades a la BD utilitzeu `UserTransaction` per establir quan comencen i acaben les transaccions. Hi ha un mètode, `utx.rollback()`, que desfarà les dades modificades a la BD en cas que hi hagi un error.

A continuació cal modificar el vostre test unitari. Hi ha un problema: volem que el test utilitzi el recurs JDBC que heu creat al servidor Glassfish. Això vol dir que el vostre test ha de poder accedir a les EJB a Glassfish. Per aconseguir-ho utilitzareu un servidor Glassfish-embedded, que s'iniciarà durant els tests i permetrà accedir als recursos JDBC i EJB del servidor real Glassfish. El primer pas és afegir unes dependències al fitxer pom.xml. Fixeu-vos que heu de substituir

DIRECTORI_INSTALACIO_NETBEANS pel directori on tingueu instal·lat el servidor Glassfish (vegeu la figura 2.20 i la figura 2.21). També heu de copiar el fitxer mysql-connector-java-5.1.40-bin.jar al directori */NetBeans/glassfish-4.1/glassfish/lib/embedded*.

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <glassfish.embedded-static-shell.jar>/DIRECTORI_INSTALACIO_NETBEANS/
        glassfish-4.1//glassfish/lib/embedded/glassfish-embedded-static-shell.
        jar</glassfish.embedded-static-shell.jar>
    <glassfish.mysql.jar>/DIRECTORI_INSTALACIO_NETBEANS/glassfish-4.1//
        glassfish/lib/embedded/mysql-connector-java-5.1.40-bin.jar</glassfish.
        mysql.jar>
</properties>

<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.190</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>1.1.0.Final</version>
    </dependency>
    <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>eclipselink</artifactId>
        <version>2.5.0</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.main.extras</groupId>
        <artifactId>glassfish-embedded-all</artifactId>
        <version>4.1.1</version>
        <scope>system</scope>
        <systemPath>${glassfish.embedded-static-shell.jar}</systemPath>
    </dependency>
    <dependency>
        <groupId>javax.ejb</groupId>
        <artifactId>javax.ejb-api</artifactId>
        <version>3.2</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.40</version>
        <scope>system</scope>
        <systemPath>${glassfish.mysql.jar}</systemPath>
    </dependency>
</dependencies>
```

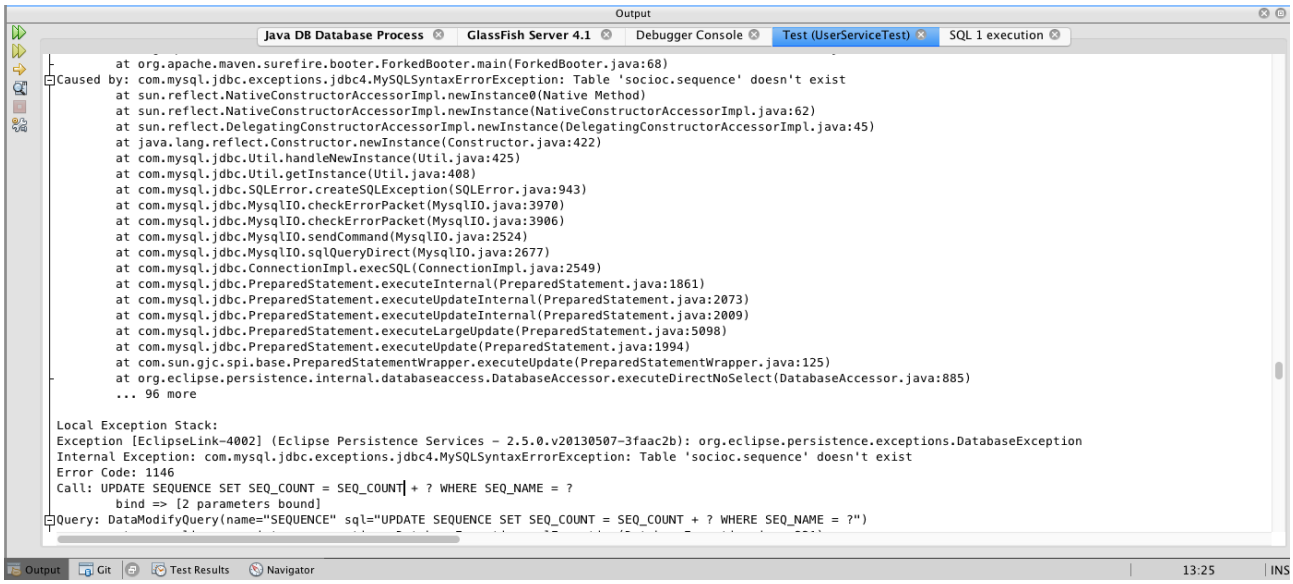
El directori glassfish-4.1 correspon a la versió 4.1. Heu de fer servir el que correspon a la vostra versió.

A continuació creeu un test que utilitzarà el recurs JDBC definit a `UserServiceImpl` i que, per tant, escriurà dades a la BD MySQL que heu creat.

```java
@Before
  public void setUp() throws NamingException {
      Context context = EJBContainer.createEJBContainer().getContext();
      userService = (UserService) context.lookup("java:global/classes/
          UserServiceImpl");
  }

  @Test
  public void findUserByUsername() {
      String username = "jdoe";
      User user = new User();
      user.setActive(true);
      user.setCreatedOn(new Timestamp(new Date().getTime()));
      user.setEmail("test@test.com");
      user.setName("Jane");
      user.setPassword("password");
      user.setRank(100);
      user.setUsername(username);
      User user1 = new User();
      user1.setActive(true);
      user1.setCreatedOn(new Timestamp(new Date().getTime()));
      user1.setEmail("test1@test.com");
      user1.setName("Joe");
      user1.setPassword("password");
      user1.setRank(100);
      user1.setUsername("joeTest");

      userService.create(user);
      userService.create(user1);

      User userFromDB = userService.findUserByUsername(username);
      Assert.assertNotNull(userFromDB);
      Assert.assertEquals("jdoe", userFromDB.getUsername());
      Assert.assertEquals("test@test.com", userFromDB.getEmail());
      Assert.assertNotNull(userFromDB.getUserId());
  }
```

Primer inicialitzeu el servidor Glassfish-embedded i recupereu l'EJB del contenidor d'EJB del servidor Glassfish real. Un cop teniu l'EJB `UserServiceImpl` ja podeu fer les operacions amb la base de dades. Executeu el test, i hauríeu d'obtenir un error similar al de la figura 2.24.

FIGURA 2.24. Error SQL



Per què creieu que teniu aquest error?

- Perquè no es pot connectar amb la BD?
- Perquè la taula "Users" no està disponible?
- Perquè la taula "Sequence" no existeix?

L'error és perquè la taula "Sequence" no existeix, no l'heu definit i no hi feu referència enlloc; llavors, quan s'està intentant crear? Per què quan vau fer el test amb la BD en memòria no va fallar? La resposta està a la classe `User`.

```
@GeneratedValue
  @Column(name = "user_id")
  private Long userId;
```

L'anotació `GeneratedValue` té com a estratègia per defecte `GenerationType.AUTO`, que intentarà crear la taula "Sequence" per guardar el valor autogenerat per al camp "user_id" de la taula. Com que la taula no existeix, l'aplicació acaba amb un error, ja que no hi pot escriure. A la vostra taula vau definir que el cap que formaria la clau principal s'autoincrementaria, llavors l'estratègia que heu d'utilitzar és `GenerationType.IDENTITY`. Modifiqueu la classe `User`.

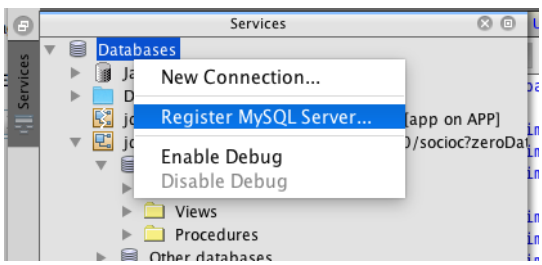
```
@GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "user_id")
  private Long userId;
```

Executeu el test un altre cop, aquesta vegada tindreu el següent error:

```
Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown
    column 'user_id' in 'field list'
```

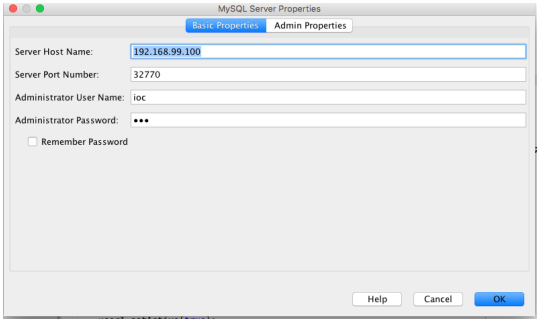
L'error diu ara que no existeix un camp anomenat "user_id" a la taula "Users". Netbeans deixa crear una connexió amb la BD que us permetrà examinar l'estructura de la taula. Seleccioneu la pestanya "Services", feu clic amb el botó dret a *Database* i registreu una connexió amb un servidor MySQL, tal com podeu veure en la figura 2.25.

Figura 2.25. Registrar un servidor MySQL a Netbeans



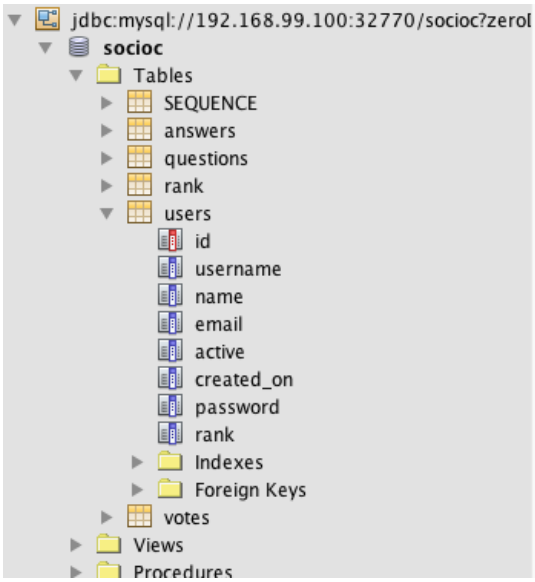
A continuació, afegiu les dades del vostre servidor MySQL (vegeu la figura 2.26).

FIGURA 2.26. Registrar el servidor MySQL a Netbeans



Un cop definida la connexió ja podeu examinar el contingut de la taula "Users" (vegeu la figura 2.27) a través de la pestanya "Services" de Netbeans.

FIGURA 2.27. Contingut de la taula "Users"



El problema és que la columna de la taula "Users" s'anomena "id" i no "user_id". Canvieu, doncs, la classe `User`.

```
@Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "id")
  private Long userId;
```

Executeu de nou el test, i ara acabarà sense cap error. Com que heu utilitzat la BD MySQL, podeu veure que els dos usuaris creats al tests estan a la taula "Users". Amb l'opció *Execute command* (vegeu la figura 2.28) podeu executar consultes i veure que la taula conté els registres esperats (vegeu la figura 2.29).

FIGURA 2.28. Executar una consulta SQL

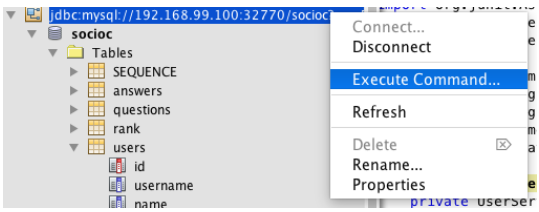
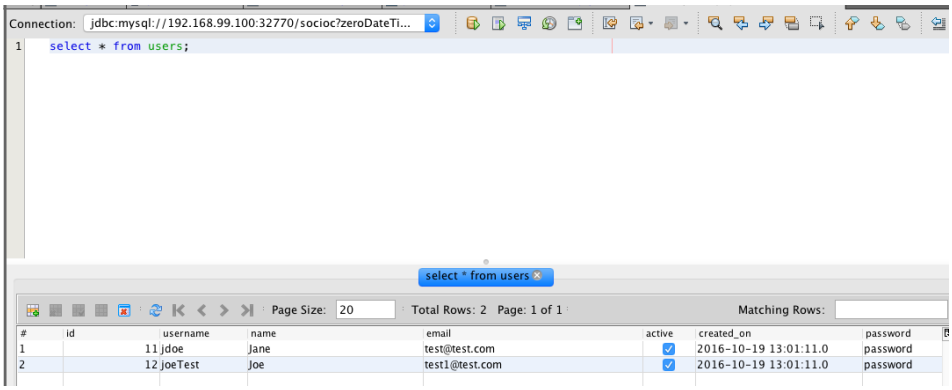


FIGURA 2.29. Registres creats durant el test



2.2.4 Refactoritzant el codi per simplificar el codi i millorar el test

Escriure test unitaris que modifiquin el contingut de la base de dades MySQL és un problema. Penseu que aquesta serà la BD que utilitzareu per a la vostra aplicació; tenir un test que escriu dades a aquesta BD és, doncs, un error. Us ha servit per veure com utilitzar el servidor Glassfish per emprar un recurs JDBC i escriure a la BD, però ho heu de canviar.

Un altre problema està a la classe `UserServiceImpl`. El primer problema és que esteu determinant quina unitat de persistència utilitzarà aquesta EJB, i això impossibilita escollir quina unitat de persistència emprar quan s'executen els tests. Heu de refactoritzar el codi per tal de poder escollir la unitat de persistència en funció que executeu tests unitaris o desplegueu l'aplicació al servidor Glassfish.

```
@PersistenceContext(unitName = “MysqlResourceSocIocPersistence")
```

Un altra cosa que podeu millorar es la gestió de les transaccions. Aquest és el codi que heu utilitzat:

```
@Override
 public void create(User user) {
     UserTransaction utx = context.getUserTransaction();
     try {
         utx.begin();
         entityManager.persist(user);
         utx.commit();
     } catch (Exception e) {
         e.printStackTrace();
         try {
             utx.rollback();
         } catch (Exception e1) {
             e1.printStackTrace();
         }
     }
 }
```

El que passa és que quan s'utilitza l'anotació `@Stateless`, automàticament EJB proporciona la següent anotació a cadascun dels mètodes `@TransactionAttribute(TransactionAttributeType.REQUIRED)`. Això fa que s'encarregui automàticament de gestionar les transaccions; per tant, podeu simplificar la classe `UserServiceImpl` notablement.

```java
@Stateless
public class UserServiceImpl implements UserService {
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public void create(User user) {
        entityManager.persist(user);
    }

    @Override
    public void edit(User user) {
        entityManager.merge(user);
    }

    @Override
    public void remove(User user) {
        entityManager.remove(user);
    }

    @Override
    public User findUserByUsername(String username) {
      return (User) entityManager.createQuery("select object(o) from User o " +
        "where o.username = :username")
        .setParameter("username", username)
        .getSingleResult();
    }
}
```

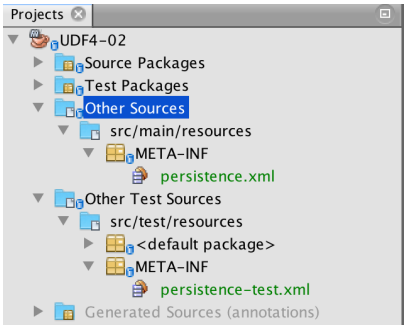
De la forma que heu fet el test anterior era necessari que el servidor Glassfish estigués corrent. Això és un problema, ja que necessiteu executar els tests de manera independent del servidor d'aplicacions, i el més important: heu de ser capaços de poder especificar la unitat de persistència que voleu utilitzar en cada moment. Per poder aconseguir això fareu servir Arquillian (red.ht/2ls9x1Q), que és un *framework* de testeig que facilita escriure tests del components EJB. Utilitzant Arquillian podreu llançar una versió d'un servidor Glassfish en memòria que us permetrà accedir a les EJB tal com si estiguessin corrent al servidor real. Per aconseguir això, el primer que heu de fer és afegir unes dependències. Hi ha uns quants canvis, per la qual cosa aquí teniu tot el fitxer pom.xml.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>
            <artifactId>arquillian-bom</artifactId>
            <version>1.1.11.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.190</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
```

```
25          <groupId>javax.validation</groupId>
26          <artifactId>validation-api</artifactId>
27          <version>1.1.0.Final</version>
28      </dependency>
29      <dependency>
30          <groupId>org.eclipse.persistence</groupId>
31          <artifactId>eclipselink</artifactId>
32          <version>2.6.4</version>
33      </dependency>
34      <dependency>
35          <groupId>javax.ejb</groupId>
36          <artifactId>javax.ejb-api</artifactId>
37          <version>3.2</version>
38      </dependency>
39      <dependency>
40          <groupId>mysql</groupId>
41          <artifactId>mysql-connector-java</artifactId>
42          <version>5.1.40</version>
43      </dependency>
44
45      <dependency>
46          <groupId>javax.el</groupId>
47          <artifactId>javax.el-api</artifactId>
48          <version>3.0.1-b04</version>
49      </dependency>
50
51      <dependency>
52          <groupId>org.jboss.arquillian.container</groupId>
53          <artifactId>arquillian-glassfish-embedded-3.1</artifactId>
54          <version>1.0.0.Final</version>
55          <scope>test</scope>
56      </dependency>
57      <dependency>
58          <groupId>org.glassfish.main.extras</groupId>
59          <artifactId>glassfish-embedded-all</artifactId>
60          <version>4.1.1</version>
61          <scope>provided</scope>
62      </dependency>
63      <dependency>
64          <groupId>javax.inject</groupId>
65          <artifactId>javax.inject</artifactId>
66          <version>1</version>
67      </dependency>
68
69      <dependency>
70          <groupId>org.jboss.arquillian.junit</groupId>
71          <artifactId>arquillian-junit-container</artifactId>
72          <scope>test</scope>
73      </dependency>
74  </dependencies>
```

El primer que fareu serà separar el fitxer on definiu les unitats de persistència en dos fitxers. Un contindrà la unitat de persistència per escriure a la BD MySQL i l'altre la unitat de persistència per a testeig (vegeu la figura 2.30).

FIGURA 2.30. Registres creats durant el test



Aquest és la unitat de persistència per escriure a MySQL:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="SocIocPersistenceUnit" transaction-type="JTA">
        <jta-data-source>jdbc/socioc__pm</jta-data-source>
        <class>org.ioc.daw.user.User</class>
        <properties>
            <property name="eclipselink.logging.level" value="FINEST"/>
        </properties>
    </persistence-unit>
</persistence>
```

I aquest és la unitat de persistència que utilitzareu per al testeig:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="SocIocPersistenceUnit-TEST" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>org.ioc.daw.user.User</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
                "/>
            <property name="javax.persistence.jdbc.user" value="username"/>
            <property name="javax.persistence.jdbc.password" value="password"/>
            <property name="javax.persistence.jdbc.url" value="
                jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql'
                "/>
            <property name="javax.persistence.schema-generation.database.action
                " value="drop-and-create"/>
            <property name="javax.persistence.schema-generation.create-source"
                value="metadata"/>
            <property name="javax.persistence.schema-generation.drop-source"
                value="metadata"/>
        </properties>
    </persistence-unit>
</persistence>
```

Ara ja podeu reescriure el test unitari. Si teniu el servidor Glassfish en execució pareu-lo, o tindreu conflictes de port.

```
@RunWith(Arquillian.class)
public class UserServiceTest {
    @Inject
```

```
4       private UserService userService;
5
6       @Deployment(testable = true)
7       public static JavaArchive createTestableDeployment() {
8           final JavaArchive jar = ShrinkWrap.create(JavaArchive.class, "example.
                jar")
9                   .addClasses(UserService.class, UserServiceImpl.class)
10                  .addAsManifestResource("META-INF/persistence-test.xml", "
                        persistence.xml")
11                  .addAsManifestResource(EmptyAsset.INSTANCE, ArchivePaths.create
                        ("beans.xml"));
12          return jar;
13      }
14
15      @Test
16      public void findUserByUsername() {
17          String username = "jdoe";
18          User user = new User();
19          user.setActive(true);
20          user.setCreatedOn(new Timestamp(new Date().getTime()));
21          user.setEmail("test@test.com");
22          user.setName("Jane");
23          user.setPassword("password");
24          user.setRank(100);
25          user.setUsername(username);
26          User user1 = new User();
27          user1.setActive(true);
28          user1.setCreatedOn(new Timestamp(new Date().getTime()));
29          user1.setEmail("test1@test.com");
30          user1.setName("Joe");
31          user1.setPassword("password");
32          user1.setRank(100);
33          user1.setUsername("joeTest");
34
35          userService.create(user);
36          userService.create(user1);
37
38          User userFromDB = userService.findUserByUsername(username);
39          Assert.assertNotNull(userFromDB);
40          Assert.assertEquals("jdoe", userFromDB.getUsername());
41          Assert.assertEquals("test@test.com", userFromDB.getEmail());
42          Assert.assertNotNull(userFromDB.getUserId());
43      }
44  }
```

`@RunWith(Arquillian.class)` indica que executareu el test utilitzant Arquillian, i a continuació injecteu l'EJB UserService. Això és possible perquè Arquillian crea un servidor Glassfish en memòria amb un contenidor amb totes les EJB de la vostra aplicació, la qual cosa us permet afegir-les a una determinada classe quan sigui necessari. A `createTestableDeployment` és on es configura el servidor d'aplicacions utilitzant Arquillian. La part més important és on indiqueu quines classes són les EJB que voleu afegir al contenidor EJB del servidor i quin és el fitxer amb la definició de la unitat de persistència que voleu fer servir.

Podeu trobar tot el codi resultant de la refactorització als annexos de la unitat.

2.3 Què s'ha après?

Heu après que utilitzant JDBC directament al codi té una sèrie d'inconvenients, ja que fa que us hagueu d'encarregar de programar operacions de baix nivell amb la base de dades, com establir i tancar les connexions, encarregar-se del mapatge de les dades de la BD amb els objectes del codi, etc.

Per solucionar aquests problemes heu vist JPA, que és una especificació que determina i fa estàndards les operacions amb la BD. És una especificació però no hi ha una implementació, així que per fer els exemples hem utilitzat EclipseLink, una de les implementacions disponibles. També heu vist que, encara que pugueu utilitzar implementacions de JPA per accedir directament a les bases de dades, utilitzar un servidor EJB com Glassfish proporciona diversos avantatges.

Pel que fa a JPA, permet establir un *pool* de connexions per fer més eficients les operacions i reutilitzar les connexions establertes amb la BD, que són operacions molt costoses. Finalment, heu vist diferents formes de testejar aplicacions que treballen amb bases de dades i heu creat una estructura, separant les unitats de persistència i utilitzant el *framework* Arquillian, que pot servir de base per testejar qualsevol aplicació que treballi amb JPA.

3. Accés a dades amb Spring i Hibernate

Java té una API que ens facilita la feina d'escriure codi que interactuï amb una base de dades. Aquesta API és la Java Persistance API (JPA), i és simplement una especificació que defineix les interfícies per fer operacions amb la BD. JPA no consisteix en cap implementació de les dades i per si sola no ens servirà de res. Necessitem unes llibreries que implementin aquesta especificació i que siguin capaces de transformar objectes Java en registres a la BD (ORM, Object-Relational Mapping) a més d'implementar totes les operacions amb la BD. Hi ha moltes implementacions, com ara Eclipselink (www.eclipse.org/eclipselink), Open JPA (openjpa.apache.org) o TopLink (bit.ly/2lAMG8o), però una de les més populars és Hibernate (hibernate.org).

Hibernate és, doncs, un *framework* ORM per a Java que té un gran rendiment i velocitat i facilita considerablement la feina de programació amb BD. Amb una senzilla configuració, permet establir una relació directa entre classes Java amb taules i tipus de dades SQL i facilita fins i tot la creació automàtica de les taules. Hibernate s'encarregarà aproximadament del 90% de la feina que s'ha de fer per treballar amb una BD automatitzant tasques repetitives. Una altra característica que fa de Hibernate un *framework* molt popular és el fet que té un llenguatge propi de consulta amb la BD que es diu Hibernate Query Language (HQL). Això permet utilitzar qualsevol de les 10 BD suportades per Hibernate sense haver de canviar ni una sola línia de codi, ja que Hibernate s'encarregarà de traduir les consultes HQL que fem a un llenguatge SQL específic per a cadascuna de les BD. Altres característiques són:

- És un projecte Opensource amb llicència LGPL.
- Alt rendiment: utilitza una memòria *cache* interna que fa que les operacions de lectura de la BD (normalment sempre hi ha moltes més operacions de lectura que d'escriptura) siguin molt ràpides.
- Creació automàtica de les taules.
- Simplifica l'obtenció de dades de múltiples taules.

Hibernate ens ajudarà pel que fa a les bases de dades, i Spring, per la seva banda, ens ajudarà en el disseny de la nostra aplicació. Spring és un *framework* que utilitza injecció de dependències (en anglès, DI, Dependency Injection) o la inversió del control (en anglès, IoC, Inversion of Control). IoC es refereix al fet que una classe no s'encarregarà de crear instàncies de les seves dependències, sinó que el contenidor DI s'encarregarà de crear els objectes amb la configuració necessària i injectar-los on faci falta. Aquest fet, que sembla tan simple, té grans implicacions a l'hora de desenvolupar una aplicació. Afavoreix la composició sobre l'herència de classes, la qual cosa fa que hi hagi menys dependències entre les classes. I

menys dependències implica que les classes faran menys coses i més específiques, per la qual cosa el codi serà més fàcilment reutilitzable i més fàcilment testejable.

La combinació de Spring i Hibernate ens permetrà dissenyar i desenvolupar un codi que pugui treballar fàcilment amb diferents tipus de BD (Hibernate) i on Spring ens donarà flexibilitat per canviar si és necessari Hibernate per qualsevol altre *framework* ORM.

Continuarem amb el desenvolupament de l'aplicació Java "SocIoc". Explicarem:

- Hibernate
- Spring i IoC: inversió del control o injecció de dependències
- HQL
- Com configurar Spring i Hibernate
- Anotacions
- Relacions 1..M i N..M
- Validació
- Tests unitaris

3.1 "Socloc". Dialogant amb usuaris amb Spring i Hibernate

Les classes Java haurien de ser al més independents possible d'altres classes. Això augmenta la possibilitat de reutilitzar aquestes classes i simplifica els tests unitaris. Per aconseguir aquesta separació o desacoblament, la dependència que una classe tingui amb d'altres s'ha d'injectar, més que fer que la classe creï o busqui la dependència. Això representa el principi d'inversió de control o principi de Hollywood: "no ens truquis" (crear/buscar objectes), "nosaltres et trucarem" (injectarem els objectes). Per exemple, la classe A dependrà de B si usa la classe B com a variable. Si usem injecció de dependències, la classe B serà passada a la A via el constructor (injecció de construcció) o a través d'un mètode *setter* (injecció *setter*). Vegem un exemple:

```
1  public interface UserDAO {
2      public void create(User user);
3      public User edit(User user);
4      public void remove(User user);
5      public User findUserByUsername(String username);
6      public User findUserWithHighestRank();
7      public List<User> findActiveUsers();
8  }
9
10 @Stateless
11 public class UserDAOJPA implements UserDAO {
12     @PersistenceContext
13     private EntityManager entityManager;
14
15     @Override
```

```
    public void create(User user) {
        entityManager.persist(user);
    }

    @Override
    public User edit(User user) {
        return entityManager.merge(user);
    }

    @Override
    public void remove(User user) {
        user = entityManager.merge(user);
        entityManager.remove(user);
    }

    @Override
    public User findUserByUsername(String username) {
        try {
            return (User) entityManager.createQuery("select object(o) from User
                  o " +
                     "where o.username = :username")
                     .setParameter("username", username)
                     .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }

    }

    @Override
    public List<User> findActiveUsers() {
        try {
            return (List<User>) entityManager.createQuery("select object(o)
                from User o " +
                    "where o.active= true")
                    .getResultList();
        } catch (NoResultException e) {
            return null;
        }
    }

    @Override
    public User findUserWithHighestRank() {
        try {
            return (User) entityManager.createQuery("select object(o) from User
                  o order by o.rank DESC")
                     .setMaxResults(1)
                     .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
}

public class UserController {
    private UserDAO userDAO;

    public User getUser(String username){
        return userDAO.findUserByUsername(username);
    }
}
```

Podeu veure que la classe `UserController` utilitza la interfície `UserDAO`. Si us hi fixeu, el codi no fa referència a cap implementació de la interfície. Hem de modificar el codi de `UserController` per fer referència a una implementació.

```
public class UserService {
    private UserDAO userDAO;
```

```
3
4       public UserService() {
5           this.userDAO = new UserDAOJPA();
6       }
7
8       public User getUser(String username) {
9           return userDAO.findUserByUsername(username);
10      }
11  }
```

En aquest cas, fem que la implementació de la interfície sigui la de la classe `UserDAOJPA`, que ens proporciona la funcionalitat per accedir a la BD utilitzant JPA. Aquest codi té diversos problemes. La classe `UserService` té la configuració de la implementació de la interfície `UserDAO`. Això farà que per testejar la classe `UserService` utilitzem un objecte real `UserDAOJPA`, que establirà una connexió amb una BD. Un altre problema és que si canviem la implementació de `UserDAO` i fem una implementació que usa Hibernate, haurem de venir a la classe `UserController` i canviar el codi. El que ens permet la injecció de dependències (DI) és que la classe `UserController` no conegui els detalls de la configuració de `UserDAO`, sinó que aquests es passin. Si anéssim a utilitzar DI podríem refactoritzar el codi:

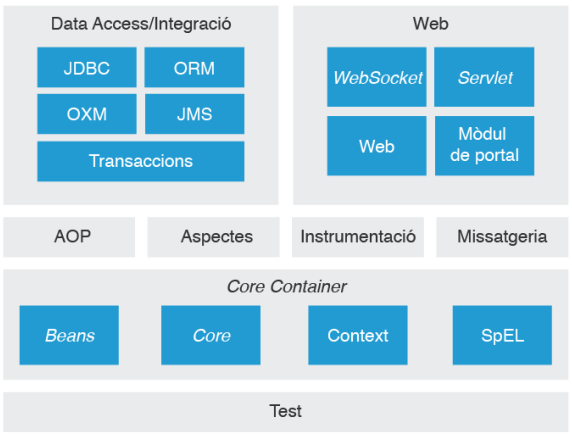
```
1   public class UserController {
2       private UserDAO userDAO;
3
4       public UserController(UserDAO userDAO) {
5           this.userDAO = userDAO;
6       }
7
8       public User getUser(String username){
9           return userDAO.findUserByUsername(username);
10      }
11  }
```

D'aquesta manera, la classe `UserController` està totalment desacoblada de la implementació de `UserDAO`. El *framework* de DI s'encarregarà de passar la versió correcta de `UserDAO` a la classe `UserController`. La injecció de dependències es pot aconseguir amb Java Standard. Spring, però, simplifica el procés mitjançant una forma estàndard de configurar les dependències entre els objectes.

3.1.1 Integrant l'aplicació "Socloc" amb Spring

Spring és un *framework* d'inversió del control (IoC) format d'una sèrie de mòduls que faciliten la feina de desenvolupar aplicacions Java. En ser modular, ens permet escollir quins mòduls utilitzar segons les necessitats de l'aplicació que estem desenvolupant. Hi ha uns 20 mòduls, i en la figura 3.1 podeu veure la seva arquitectura.

FIGURA 3.1. Mòduls Spring



Core container: aquest és el mòdul fonamental de Spring, i permet fer IoC i DI. També defineix el context de l'aplicació que indicarà quins *beans* (objectes gestionats pel *container* IoC) hi ha i com es relacionen entre si. Està format per quatre mòduls:

- Core proporciona les parts fonamentals del *framework*, incloent IoC i injecció de dependències.
- Beans proporciona `BeanFactory`, que és una classe que segueix el patró de disseny de *software* anomenat *factory pattern* i que serveix per crear objectes a partir d'una classe.
- Context proporciona la funcionalitat per accedir als objectes gestionats per Spring (*beans*). La interfície `ApplicationContext interface` és la part central d'aquest mòdul.
- SpEL proporciona un llenguatge potent i flexible per manipular un *bean* i les seves dependències mentre l'aplicació s'està executant.

Data Access/Integration: aquest és el mòdul que proporciona accés a dades i sistemes d'integració. Està format pels següents mòduls:

- JDBC proporciona la funcionalitat JDBC per accedir a bases de dades.
- ORM proporciona capes d'integració per a API de mapeig d'objectes Java amb taules de les BD. Entre les API suportades hi ha JPA, JDO, Hibernate i iBatis.
- OXM proporciona la funcionalitat per a la transformació d'objectes a XML, i viceversa.
- JMS (Java Messaging Service) és un mòdul que conté la funcionalitat per consumir i produir missatges JMS.
- Transaction és un mòdul que facilita la gestió de transaccions.

Web: la capa web està formada pels següents mòduls:

- Web proporciona funcionalitats típiques que s'utilitzen a les aplicacions web, com pujada de fitxers o la inicialització del contenidor IoC utilitzant *servlets*.
- Web-MVC: conté una implementació de MVC (Model View Controller) per a aplicacions web.
- Web-Socket proporciona suport per a comunicacions client-servidor en aplicacions web.
- Web-Portlet proporciona una implementació de MVC per ser utilitzada en entorns *portlet*.

Hi ha altres mòduls importants:

- AOP (Aspect-Oriented Programming): consisteix en una implementació de programació orientada a aspectes. AOP permet interceptar crides a funcions i injectar codi que s'executarà abans o després d'executar el codi de la funció.
- Aspects: aquest mòdul permet la integració d'AspectJ, que és un *framework* d'AOP.
- Instrumentation: proporciona instrumentació de classes i funcionalitat per carregar classes que són necessàries a certs servidors d'aplicacions.
- Test: permet testejar aplicacions Spring utilitzant *frameworks* de testeig, com JUnit o TestNG.

No tots aquests mòduls seran necessaris quan desenvolupem una aplicació. El que serà fonamental serà afegir el *core*, que inclou el contenidor IoC, que serà l'encarregat de crear instàncies dels objectes, configurar-los i crear les dependències necessàries. El contenidor IoC obté aquesta informació de la configuració de Spring que es pot definir amb fitxers XML o amb classes de configuració Java.

L'arxiu de partida per treballar el descrit en aquest apartat el teniu disponble als annexos de la unitat.

El primer que haurem de fer és afegir les dependències. A partir del fitxer proporcionat als annexos afegirem les següents dependències:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <springframework.version>4.3.4.RELEASE</springframework.version>
    <mysql.connector.version>5.1.40</mysql.connector.version>
    <junit.version>4.12</junit.version>
    <mockito.version>1.10.19</mockito.version>
    <h2.version>1.4.190</h2.version>
</properties>
<dependencies>
...
 <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${springframework.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
```

```
    <version>${springframework.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${springframework.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${springframework.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${springframework.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${springframework.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
...
</dependencies>
```

Fixeu-vos que hem definit una sèrie de propietats al pom.xml que permeten centralitzar la versió utilitzada en una variable. D'aquesta manera, quan hi hagi disponible una nova versió de Spring, enlloc de canviar el valor en sis dependències, només ho haurem de fer en un lloc.

```
<properties>
     <springframework.version>4.3.4.RELEASE</springframework.version>
  </properties>
```

A la implementació que tenim de `UserService`, la classe crea una instància de `UserDAO`. El que volem ara és que Spring s'encarregui d'aquesta configuració. Refactoritzarem `UserService` de manera que no creï una instància de `UserDAO`.

```
public class UserService {
    private UserDAO userDAO;

    public UserService(UserDAO userDAO) {
        this.userDAO = userDAO;
    }

    public User getUser(String username) {
        return userDAO.findUserByUsername(username);
    }
}
```

El que farem serà que Spring s'encarregui d'injectar la configuració de `UserDAO` necessària. Això vol dir que podrem testejar la classe `UserService` sense necessitar tenir cap informació de com serà la implementació de `UserDAO`. Per poder fer-ho necessitem configurar la nostra aplicació per tal que utilitzi Spring. Spring permet escollir quin serà el context a utilitzar en els tests. El que nosaltres

volem testejar ara és que la classe `UserService` i les seves dependències estan gestionades per Spring. El que faci la implementació de la classe `UserDAO` realment no ens interessa en aquest moment, per la qual cosa utilitzarem un *mock* de la classe. Definim llavors la classe que tindrà la configuració del context de Spring a *src/test/java* al paquet `package org.ioc.daw.config;`.

```
1  package org.ioc.daw.config;
2
3  import org.ioc.daw.user.UserDAO;
4  import org.ioc.daw.user.UserService;
5  import org.mockito.Mockito;
6  import org.springframework.context.annotation.Bean;
7  import org.springframework.context.annotation.Configuration
8
9  @Configuration
10 public class SpringTestConfig {
11   @Bean
12   public UserDAO userDAO() {
13       return Mockito.mock(UserDAO.class);
14   }
15
16   @Bean
17   public UserService userService(UserDAO userDAO) {
18       return new UserService(userDAO);
19   }
20 }
```

`@Configuration` indica que la classe conté un o més mètodes anotats amb `@Bean` i que produeixen *beans* gestionats pel contenidor de Spring. Aquesta configuració defineix dos *beans* que estaran gestionats pel contenidor IoC de Spring. Això permetrà injectar aquests dos *beans* quan ens faci falta.

Fixeu-vos que l'objecte que retorna `userDAO()` no és cap implementació real de la interfície. Utilitzant `Mockito.mock(UserDAO.class)` retornem un *mock*, és a dir, un objecte que fa de *proxy* amb la interfície però que no té cap codi. El que això permet és oblidar-se completament del funcionament de les classes que implementen `UserDAO` i focalitzar el test en `UserService`. Si l'objecte *mock* no té cap implementació, com es pot usar en el codi? Al test veureu que podeu definir què retornen els diferents mètodes quan són invocats. Creeu la classe de test `UserServiceTest` al paquet `package org.ioc.daw.user;`.

Mockito és un *framework* de testeig que facilita la creació d'objectes de test (*mock*) que amaguen la implementació real i que faciliten els tests unitaris.

```
1  import org.ioc.daw.config.SpringTestConfig;
2  import org.junit.Test;
3  import org.junit.runner.RunWith;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.test.context.ContextConfiguration;
6  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
7
8  import static org.junit.Assert.assertEquals;
9  import static org.mockito.Mockito.times;
10 import static org.mockito.Mockito.verify;
11 import static org.mockito.Mockito.when;
12
13 @RunWith(SpringJUnit4ClassRunner.class)
14 @ContextConfiguration(classes = {SpringTestConfig.class})
15 public class UserServiceTest {
16     @Autowired
17     private UserDAO userDAO;
18
19     @Autowired
20     private UserService userService;
21
```

```
22      @Test
23      public void getUserByUsername() {
24          String username = "test";
25          User user = new User();
26          user.setUsername(username);
27          user.setUserId(1L);
28
29          when(userDAO.findUserByUsername(username)).thenReturn(user);
30
31          User userResult = userService.getUser(username);
32          assertEquals(username, userResult.getUsername());
33          assertEquals(new Long(1), userResult.getUserId());
34          verify(userDAO, times(1)).findUserByUsername(username);
35      }
36  }
```

`@RunWith(SpringJUnit4ClassRunner.class)` especifica que el test carregarà un context de Spring per ser utilitzat en un test `JUnit`, i `@ContextConfiguration(classes = {SpringTestConfig.class})` especifica quines classes tenen la configuració del context. Com que hem definit que `SpringTestConfig` serà la configuració a utilitzar, podem injectar (amb la notació `@Autowired`) els *beans* definits. Fixeu-vos que com que `UserDAO` és un *mock*, podem dir el que retornaran els seus mètodes:

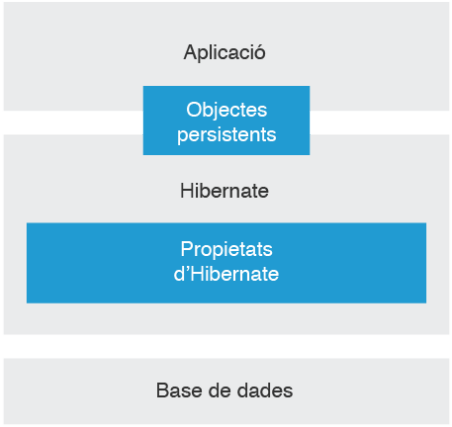
```
1  when(userService.findUserByUsername(username)).thenReturn(user);
```

Estem dient que quan es cridi al mètode `findUserByUsername` retornarem l'objecte `user`. Les línies que utilitzen `assertEquals` fan les comprovacions del test, comproven que l'objecte retornat pel mètode `findUserByUsername` de `UserService` és el mateix que el que retorna `UserDAO`. Finalment, `verify(userDAO, times(1)).findUserByUsername(username);` comprova que el mètode `findUserByUsername` només és invocat un cop. Podeu trobar el codi en el fitxer .

3.1.2 Integrant l'aplicació "Socloc" amb Hibernate

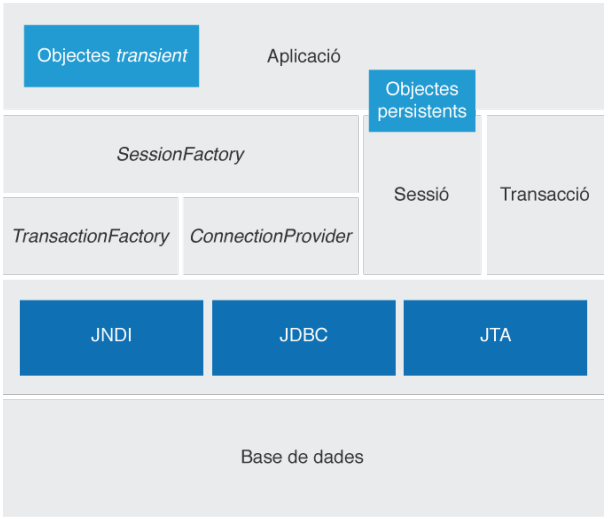
Hibernate és un *framework* ORM (Object-Relational Mapping). És a dir, Hibernate s'encarrega de relacionar taules d'una base de dades amb objectes Java. Com es pot veure en la figura 3.2, Hibernate crea una capa entre la BD i l'aplicació. S'encarregarà de gestionar la configuració de com accedir a la BD, el tipus de BD, com fer el mapeig entre les classes i taules, i establir les relacions entre diferents taules.

Figura 3.2. Arquitectura d'Hibernate



En la figura 3.3 es representa amb més detall com funciona Hibernate. A l'hora de guardar les dades a la BD, Hibernate crea una instància de la classe de tipus entitat (una classe Java mapejada amb una taula). Aquest objecte s'anomena objecte *transient*, ja que no està associat amb cap sessió i no està guardat a la BD. Per guardar un objecte a la BD s'utilitza una instància de la interfície `SessionFactory`, un objecte de tipus *singleton* (només hi ha una instància de l'objecte a l'aplicació) que implementa el patró de disseny *factory*. `SessionFactory` carrega la configuració d'Hibernate i s'encarrega de gestionar la configuració de la connexió amb la BD.

Figura 3.3. Arquitectura detallada d'Hibernate



Cada connexió amb la BD a Hibernate es fa creant una instància d'una implementació de la interfície `Session`. Hibernate també disposa d'una API per gestionar les transaccions i que permet utilitzar transaccions JDBC o JTA. Una transacció representa una única unitat de treball amb la base de dades. Vegem amb una mica més de detall els diferents blocs de la figura 3.3:

- SessionFactory: és una classe encarregada de produir objectes de tipus `Session`. Opcionalment, manté una memòria *cache* de segon nivell que guarda dades de la connexió amb la BD perquè siguin reutilitzades entre diferents transaccions.

- Session: s'encarreguen de la conversa entre les aplicacions i la BD. Manté una *cache* de primer nivell dels objectes de l'aplicació. Aquesta *cache* s'utilitza a l'hora de recuperar objectes utilitzant el seu identificador o a l'hora de navegar a través de les dependències de l'objecte.
- Objectes persistents: són objectes que contenen la funcionalitat de l'aplicació. Cada objecte està associat amb una única sessió d'Hibernate. Un cop la sessió associada a un objecte es tanca, els objectes passen a estar a l'estat *detached* (separat).
- Objectes tipus *transient* i *detached*: són les instàncies de classes de tipus `Entity` que no estan associades a cap sessió d'Hibernate. Poden haver estat creades per l'aplicació i no haver estat guardades, o poden ser el resultat que s'hagi tancat una sessió d'Hibernate.
- Proveïdor de connexions (`ConnectionProvider`): és opcional i permet crear un *pool* de connexions JDBC.
- `TransactionFactory`: permet crear instàncies d'objectes de tipus `Transaction`.

Per poder treballar amb Hibernate i que formi part de la vostra aplicació, el primer que fareu serà afegir les dependències necessàries. Importeu a Netbeans el codi descarregat dels annexos. Modificareu el fitxer pom.xml per afegir les dependències d'Hibernate.

Als annexos de la unitat trobareu un arxiu amb el codi per importar a Netbeans i treballar amb Hibernate.

```
<properties>
  ..
    <hibernate.version>5.2.5.Final</hibernate.version>
  ..
  </properties>
 <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>5.3.4.Final</version>
  </dependency>
  <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
  </dependency>
  <dependency>
      <groupId>org.jadira.usertype</groupId>
      <artifactId>usertype.core</artifactId>
      <version>6.0.1.GA</version>
  </dependency>
```

Hibernate a la llibreria *hibernate-core* porta l'especificació JPA 2.1 i la seva implementació. Per tant, no és necessari incloure les següents dependències.

```
<dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>persistence-api</artifactId>
        <version>1.0.2</version>
    </dependency>
```

Hibernate utilitza el concepte de sessions per gestionar les connexions amb la base de dades. A cada sessió s'obre una única connexió amb la BD i s'utilitza

fins que la sessió es tanca. Cada objecte que es carrega en memòria per Hibernate estarà associat amb la sessió. Això permet a Hibernate persistir automàticament els objectes que s'han modificat. Quan la sessió persisteix canvis a la BD es diu *flushing*. Cada objecte associat amb la sessió es comprova per veure si ha canviat d'estat. Qualsevol objecte amb canvi d'estat es conservarà a la base de dades, amb independència que els objectes modificats es guardin o no explícitament. Aquesta característica es pot configurar, però per defecte podeu configurar el comportament arran d'Hibernate, es farà automàticament. Hibernate fa *flushing* en les següents situacions:

- Quan s'executa directament el mètode `flush()`.
- Abans que Hibernate faci una consulta, si creu que és necessari per obtenir un resultat precís.
- Quan es confirma una transacció.
- Quan es tanca la sessió.

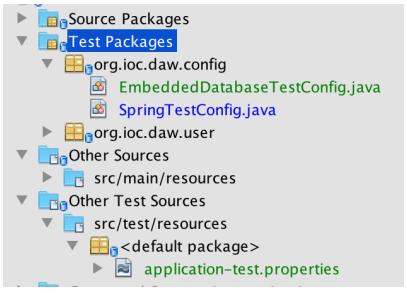
El que volem fer a continuació és començar a utilitzar Hibernate. Per fer-ho definireu una classe que implementi la interfície `UserDAO` i que utilitzi la funcionalitat d'Hibernate. Creareu la classe `org.ioc.daw.user.UserHibernateDAO`.

```
1  import org.hibernate.Criteria;
2  import org.hibernate.Session;
3  import org.hibernate.SessionFactory;
4  import org.hibernate.criterion.Order;
5  import org.hibernate.criterion.Restrictions;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.stereotype.Repository;
8  import javax.transaction.Transactional;
9  import java.util.List;
10
11 @Transactional
12 @Repository("userHibernateDAO")
13 public class UserHibernateDAO implements UserDAO {
14
15     @Autowired
16     private SessionFactory sessionFactory;
17
18     @Override
19     public void create(User user) {
20         getSession().saveOrUpdate(user);
21     }
22
23     @Override
24     public User edit(User user) {
25         return (User) getSession().merge(user);
26     }
27
28     @Override
29     public void remove(User user) {
30         getSession().delete(user);
31     }
32
33     @Override
34     public User findUserByUsername(String username) {
35         Criteria criteria = createEntityCriteria();
36         criteria.add(Restrictions.eq("username", username));
37         return (User) criteria.uniqueResult();
38     }
39
```

```
40      @Override
41      public User findUserWithHighestRank() {
42          Criteria criteria = createEntityCriteria();
43          criteria.addOrder(Order.desc("rank"));
44          return (User) criteria.uniqueResult();
45      }
46
47      @Override
48      public List<User> findActiveUsers() {
49          Criteria criteria = createEntityCriteria();
50          criteria.add(Restrictions.eq("active", true));
51          return (List<User>) criteria.list();
52      }
53
54      protected Session getSession() {
55          return sessionFactory.getCurrentSession();
56      }
57
58      private Criteria createEntityCriteria() {
59          return getSession().createCriteria(User.class);
60      }
61  }
```

`@Repository(“userHibernateDAO”)` indica que quan la classe sigui escanejada per Spring es crearà un *bean* anomenat `userHibernateDAO`. La notació `@Transactional` indica que els mètodes definits a la classe utilitzaran transaccions, és a dir, que quan el mètode es comença a executar s'obre una transacció i abans d'acabar es tanca. A les línies 4-5 injecteu l'objecte que permetrà fer totes les operacions utilitzant Hibernate `SessionFactory` i obtenir una sessió d'Hibernate. A la classe `UserHibernateDAO` heu vist com injectar les dependències necessàries per utilitzar Hibernate i com fer que els mètodes siguin transaccionals. El que heu de fer ara és crear la configuració que creï el *bean* `SessionFactory`, un altre *bean* que defineixi la classe que gestioni les transaccions, i finalment necessitareu definir un *bean* de tipus `DataSource`, que establirà com es farà la connexió amb la base de dades. El que interessa és testejar que Hibernate treballi amb la BD correctament, no tant la BD en si, i per això utilitzareu una BD en memòria. En aquest cas, H2. Al directori *test/java* creareu la classe `org.ioc.daw.config.EmbeddedDatabaseTestConfig`, i el fitxer de propietats al directori `test/resources application-test.properties`, tal com podeu veure en la figura 3.4.

Figura 3.4. Hibernate




```
1  import org.hibernate.SessionFactory;
2  import org.springframework.beans.factory.annotation.Autowired;
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.context.annotation.PropertySource;
6  import org.springframework.core.env.Environment;
```

```
7  import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
8  import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
9  import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
10 import org.springframework.orm.hibernate5.HibernateTransactionManager;
11 import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
12 import org.springframework.transaction.annotation.EnableTransactionManagement;
13 import javax.sql.DataSource;
14 import java.util.Properties;
15
16 @Configuration
17 @EnableTransactionManagement
18 @PropertySource(value = {"application-test.properties"})
19 public class EmbeddedDatabaseTestConfig {
20
21     @Autowired
22     private Environment environment;
23
24     @Bean
25     public UserDAO userDAO() {
26         return  new UserHibernateDAO();
27     }
28
29     @Bean
30     public DataSource dataSource() {
31         EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
32         EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
33         return db;
34     }
35
36     @Bean
37     @Autowired
38     public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
39         LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
40         sessionFactory.setDataSource(dataSource);
41         sessionFactory.setPackagesToScan("org.ioc.daw");
42         sessionFactory.setHibernateProperties(hibernateProperties());
43         return sessionFactory;
44     }
45
46     private Properties hibernateProperties() {
47         Properties properties = new Properties();
48         properties.put("hibernate.dialect", environment.getRequiredProperty("
               hibernate.dialect"));
49         properties.put("hibernate.hbm2ddl.auto", environment.
               getRequiredProperty("hibernate.hbm2ddl"));
50         properties.put("hibernate.show_sql", environment.getRequiredProperty("
               hibernate.show_sql"));
51         properties.put("hibernate.format_sql", environment.getRequiredProperty(
               "hibernate.format_sql"));
52         return properties;
53     }
54
55     @Bean
56     @Autowired
57     public HibernateTransactionManager transactionManager(SessionFactory s) {
58         HibernateTransactionManager txManager = new HibernateTransactionManager
               ();
59         txManager.setSessionFactory(s);
60         return txManager;
61     }
62 }
```

`@EnableTransactionManagement` habilita la capacitat de gestionar les transaccions amb la BD. `@PropertySource(value = {‘‘classpath:application-test.properties’’})` permet definir propietats a un fitxer de propietats que serà accessible a través del component injectat *Environment*.

```
@Bean
    public UserDAO userDAO() {
        return  new UserHibernateDAO();
    }
```

Defineix que com el Bean de tipus `UserDAO` que utilitzarem serà del tipus `UserHibernateDAO`.

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
    return db;
}
```

Aquest codi crea un *bean* de tipus `DataSource`, que és el que establirà amb quina base de dades es connectarà Hibernate. En el vostre cas, indiqueu que serà una BD en memòria de tipus H2. Això serà l'únic que haureu de canviar en el codi si voleu utilitzar una BD diferent.

```
@Bean
  @Autowired
  public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
      LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
      sessionFactory.setDataSource(dataSource);
      sessionFactory.setPackagesToScan("org.ioc.daw");
      sessionFactory.setHibernateProperties(hibernateProperties());
      return sessionFactory;
  }
```

El mètode `sessionFactory()` crea un *bean* de tipus `LocalSessionFactoryBean` que té informació de com connectar amb la base de dades a través de l'objecte `DataSource`. Fixeu-vos que el bean `DataSource` s'injecta utilitzant la notació `@Autowired`. A més de `DataSource`, es necessita definir les propietats d'Hibernate (`hibernateProperties()`). Gràcies a la notació `@PropertySource` es poden externalitzar les propietats a fitxers, fet que permet carregar diferents fitxers de propietats a diferents contextos. Un cop l'objecte `SessionFactory` s'ha creat s'injectarà al mètode `transactionManager`, que proporcionarà suport per a les transaccions amb la base de dades.

```
@Bean
  @Autowired
  public HibernateTransactionManager transactionManager(SessionFactory s) {
      HibernateTransactionManager txManager = new HibernateTransactionManager()
          ;
      txManager.setSessionFactory(s);
      return txManager;
  }
```

És a dir, Spring crearà tres *beans*: un que té informació de la BD a utilitzar, un que usa aquest *bean* per crear una sessió que gestiona la connexió amb la BD i un altre que gestiona les transaccions. El fitxer de propietats application-test.properties d'Hibernate té el contingut mostrat a continuació. A més de definir que el dialecte SQL que usarà Hibernate és H2Dialect, estem indicant també amb la propietat `hibernate.hbm2ddl` que es creïn les taules automàticament a partir dels objectes.

```
hibernate.dialect = org.hibernate.dialect.H2Dialect
hibernate.hbm2ddl = create
hibernate.show_sql = true
hibernate.format_sql = true
```

Un cop Hibernate i Spring estan configurats podem passar a crear el test `UserDAOTest` al paquet `org.ioc.daw.user`. De moment comprovareu que podeu escriure i llegir informació de la BD.

```
import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import java.sql.Timestamp;
import java.util.Date;


@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {EmbeddedDatabaseTestConfig.class})
public class UserDAOTest {
    @Autowired
    private UserDAO userDAO;

    @Test
    public void saveUser() {
        User user = new User();
        user.setUsername("test");
        user.setActive(true);
        user.setEmail("email@test.com");
        user.setPassword("password");
        user.setName("name");
        user.setRank(10);
        user.setCreatedOn(new Timestamp(new Date().getTime()));
        assertNull(user.getUserId());
        userDAO.create(user);
        assertNotNull(user.getUserId());

        User userFromDb = userDAO.findUserByUsername("test");
        assertEquals(user.getUserId(), userFromDb.getUserId());
    }
}
```

Amb `@ContextConfiguration(classes = {EmbeddedDatabaseTestConfig .class})` definiu quin serà el fitxer de propietats que utilitzareu al test; en el vostre cas és el fitxer de configuració que defineix un `DataSource` per connectar a la BD H2. Fixeu-vos que abans de guardar l'usuari el valor del seu `userId` és *null*, però que un cop Hibernate persisteix l'objecte, tal com es va definir a la classe `User` (`@GeneratedValue(strategy = GenerationType.IDENTITY)`), es genera un valor automàticament. El test consisteix a guardar l'usuari a la BD, recuperar-lo usant el nom d'usuari i comprovar que el `userId` dels dos objectes és el mateix.

Una de les característiques d'Hibernate és que un cop un objecte forma part d'una sessió si es fan canvis sobre alguna de les seves propietats, els canvis es persisteixen a la BD. Modificareu el test anterior per comprovar-ho.

```
@Test
public void saveUser(){
```

```
    User user = new User();
    user.setUsername("test");
    user.setActive(true);
    user.setEmail("email@test.com");
    user.setPassword("password");
    user.setName("name");
    user.setRank(10);
    user.setCreatedOn(new Timestamp(new Date().getTime()));

    assertNull(user.getUserId());
    userDAO.create(user);
    assertNotNull(user.getUserId());
    user.setEmail("new-email@test.com");

    User userFromDb = userDAO.findUserByUsername("test");
    assertEquals(user.getUserId(), userFromDb.getUserId());
    assertEquals("new-email@test.com", userFromDb.getEmail());
}
```

Un cop s'ha guardat l'objecte `user`, modifiqueu el correu-e i comproveu que s'ha guardat correctament. Si executeu el test veureu que hi ha un error:

```
Failed tests:   saveUser(org.ioc.daw.user.UserDAOTest): expected:<[new-]
    email@test.com> but was:<[]email@test.com>
```

El problema és que necessiteu establir el context per a la transacció, és a dir, indicar on comença i on acaba la transacció. Si afegiu la notació `@Transactional` al mètode del test i torneu a executar el test veureu que aquest cop s'executa correctament.

Podeu accedir al codi al fitxer que trobareu als annexos de la unitat.

Com podríeu utilitzar el test que heu creat per comprovar que podeu escriure dades a la BD MySQL "SocIoc"?

La configuració està definida a la classe `EmbeddedDatabaseTestConfig`, així que tot el que haureu de canviar estarà aquí:

- `@PropertySource(value = {''classpath:application-test.properties''})` indica que s'han d'agafar els valors d'aquest fitxer de propietats per fer la connexió amb la BD.
- El bean `DataSource` indica que utilitzareu una BD en memòria.

Llavors, per connectar-vos a la BD MySQL només haureu de canviar el fitxer de propietats i el tipus de `DataSource`. Si feu el canvi a `EmbeddedDatabaseTestConfig`, si voleu tornar a utilitzar la BD en memòria, com serà el cas, haureu de tornar a canviar un altre cop el fitxer. Creareu una altra classe de configuració al paquet `org.ioc.daw.config` que establirà com treballar amb la BD MySQL que anomenarem `HibernateMysqlConfiguration`.

```
package org.ioc.daw.config;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
```

```
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
@ComponentScan({"org.ioc.daw.user", "org.ioc.daw.question",
        "org.ioc.daw.answer", "org.ioc.daw.vote", "org.ioc.daw.rank"})
@PropertySource(value = {"jdbc.properties", "hibernate.properties"})
@Import(value = {HibernateConfiguration.class})
public class HibernateMysqlConfiguration {
    @Autowired
    private Environment environment;

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.
            driverClassName"));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        dataSource.setUsername(environment.getRequiredProperty("jdbc.username")
            );
        dataSource.setPassword(environment.getRequiredProperty("jdbc.password")
            );
        return dataSource;
    }
}
```

Afegiu també un fitxer de propietats src/main/resources/application.properties. Fixeu-vos que haureu de canviar jdbc.url per la IP:PORT on estigui funcionant la vostra base de dades.

```
jdbc.driverClassName = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://192.168.99.100:32768/socioc?useUnicode=true&
    useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&
    serverTimezone=UTC
jdbc.username = root
jdbc.password = root
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.show_sql = true
hibernate.format_sql = true
hibernate.hbm2ddl = validate
```

A causa de possibles problemes amb la configuració de MySQL i la seva configuració horària, afegiu una sèrie de paràmetres (`?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC/`) a la cadena de caracters de la connexió amb el servidor MySQL.

Si us hi fixeu, la classe de configuració `HibernateMysqlConfiguration` té repetit gairebé tot el codi respecte a `EmbeddedDatabaseTestConfig`. Per tant, si voleu fer algun canvi al `DataSource` o introduir una nova propietat d'Hibernate us haureu de recordar de canviar les dues classes. Això no és una bona pràctica de desenvolupament i és susceptible a errors. El que fareu serà separar les classes en tres: una classe de configuració que contindrà el codi comú i dues amb les configuracions específiques per a MySQL i H2, i afegir els DAO al seu propi fitxer de configuració. Creareu `DAOConfig` al paquet `org.ioc.daw.config`.

```
import org.ioc.daw.user.UserDAO;
import org.ioc.daw.user.UserHibernateDAO;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DAOConfig {

    @Bean
    public UserDAO userDAO() {
        return new UserHibernateDAO();
    }
}
```

HibernateConfiguration conté la configuració comú:

```
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.util.Properties;
@Configuration
@EnableTransactionManagement
@Import(value = {DAOConfig.class})
public class HibernateConfiguration {
    @Autowired
    private Environment environment;

    @Bean
    @Autowired
    public LocalSessionFactoryBean sessionFactory(DataSource dataSource) throws
         NamingException {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource);
        sessionFactory.setPackagesToScan("org.ioc.daw.user");
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.getRequiredProperty("
            hibernate.dialect"));
        properties.put("hibernate.show_sql", environment.getRequiredProperty("
            hibernate.show_sql"));
        properties.put("hibernate.format_sql", environment.getRequiredProperty(
            "hibernate.format_sql"));
        properties.put("hibernate.hbm2ddl.auto", environment.
            getRequiredProperty("hibernate.hbm2ddl"));
        return properties;
    }

    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory s) {
        HibernateTransactionManager txManager = new HibernateTransactionManager
            ();
        txManager.setSessionFactory(s);
        return txManager;
    }
}
```

`HibernateMysqlConfiguration` tindrà la configuració específica per connectar amb la BD MySQL.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;
@Configuration
@EnableTransactionManagement
@PropertySource(value = {"application.properties"})
@Import(value = {HibernateConfiguration.class})
public class HibernateMysqlConfiguration {
    @Autowired
    private Environment environment;

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.
            driverClassName"));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        dataSource.setUsername(environment.getRequiredProperty("jdbc.username")
            );
        dataSource.setPassword(environment.getRequiredProperty("jdbc.password")
            );
        return dataSource;
    }
}
```

I finalment, `EmbeddedDatabaseTestConfig` tindrà la configuració per connectar-se a la BD en memòria H2.

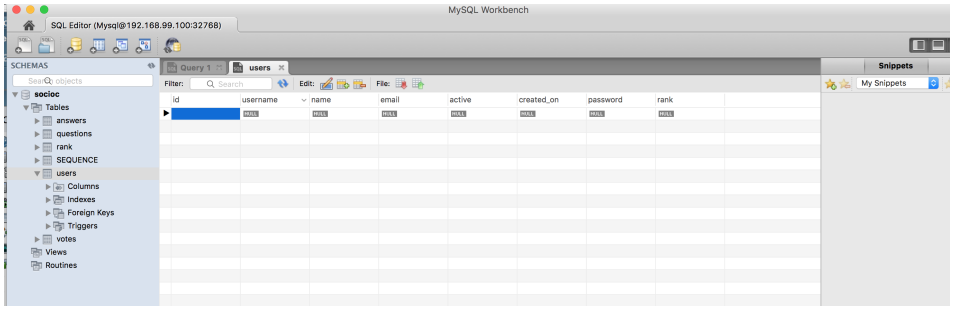
```
@Configuration
@EnableTransactionManagement
@PropertySource(value = {"application-test.properties"})
@Import(value = {HibernateConfiguration.class})
public class EmbeddedDatabaseTestConfig {

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
        return db;
    }
}
```

Als annexos de la unitat trobareu un arxiu amb el codi refactoritzat, així com un altre arxiu de MySQL Workbench amb la base de dades "SocIoc" per importar-la.

Comproveu que amb el codi refactoritzat podeu executar tots els tests. A continuació modificareu `EmbeddedDatabaseTestConfig` per tal d'executar els tests contra la BD MySQL. Podeu importar la base de dades "SocIoc" del fitxer de MySQL Workbench descarregat dels annexos.

Per comprovar que el vostre codi i configuració pot escriure a la BD MySQL que vau definir a Workbench, podeu utilitzar l'editor de *queries* de Workbench per fer la comprovació. En la figura 3.3 podeu veure que el contingut de la taula està buit.

FIGURA 3.5. Contingut de la taula "Users"



A continuació canvieu el context del test `UserDAOTest` perquè utilitzi la classe de configuració de MySQL i executeu el test `UserDAOTest.saveUser`.

```
@ContextConfiguration(classes = {HibernateMysqlConfiguration.class})
```

Si executeu el test hi haurà un altre error:

```
.HibernateConfiguration: Invocation of init method failed; nested exception is
    org.hibernate.tool.schema.spi.SchemaManagementException: Schema-validation
    : wrong column type encountered in column [id] in table [users]; found [
    int (Types#INTEGER)], but expecting [bigint (Types#BIGINT)]
```

El problema és que vau definir la taula "Users" com a tipus *int*, i Hibernate el que espera és un tipus *bigint*. Això passa perquè heu definit `userId` com a *Long*, que Hibernate mapeja amb el tipus *bigint* per tal d'acomodar a tots els possibles valors que pot guardar una variable de tipus *Long*. Per solucionar el problema modifiqueu la classe `User` canviant el tipus de `userId` de *Long* a *Integer*.

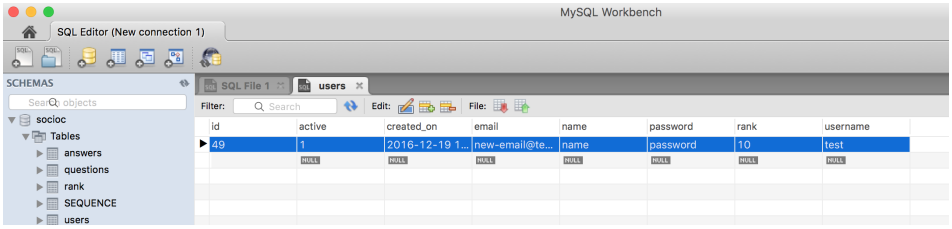
```
@Id
    @NotNull
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer userId;
    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }
```

Ara sí, executeu el test i mireu el contingut de la taula "Users"; veureu està buida. Què ha passat? Doncs que com que és un test, Hibernate fa un *rollback* (desfer els canvis d'una transacció) abans de tancar el test i esborra les dades guardades a la BD. Per tal que es guardin les dades a la BD és necessari indicar al test que no faci *rollback* amb la notació `@Rollback(false)`. El mètode amb el test tindrà llavors les següents anotacions:

```
@Test
  @Transactional
  @Rollback(false)
  public void saveUser() {
```

Ara sí que veureu que hi ha l'usuari que el test ha creat (vegeu la figura 3.4).

FIGURA 3.6. Contingut de la taula "Users"



Aquests canvis que heu fet són només per veure que podeu guardar dades a la BD només canviant un fitxer de configuració. Desfeu els canvis (treure la notació `@Rollback` i utilitzar la configuració `EmbeddedDatabaseTestConfig`) abans de continuar endavant.

A continuació veurem com podreu configurar Spring i Hibernate per treballar amb el servidor d'aplicacions Glassfish. Primer afegireu al fitxer pom.xml una dependència que us permetrà cercar recursos (en el vostre cas, un recurs JDBC) utilitzant JNDI. Java Naming and Directory Interface és una API que permet trobar recursos, serveis i components EJB que estan distribuïts. Afegiu el següent al pom.xml:

```
<dependency>
      <groupId>org.glassfish.main.common</groupId>
      <artifactId>glassfish-naming</artifactId>
      <version>4.1.1</version>
  </dependency>
```

A continuació heu de crear una classe de configuració amb els detalls específics de Glassfish al paquet `org.ioc.daw.config`. Fixeu-vos que l'únic que heu d'especificar és el tipus de `DataSource`. En aquest cas utilitzeu `JndiDataSourceLookup` per buscar el recurs JDBC especificat per a la propietat `jndi.socioc` del fitxer de propietats glassfish-application.properties.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.lookup.JndiDataSourceLookup;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;

@Configuration
@EnableTransactionManagement
@PropertySource(value = {"glassfish-application.properties"})
@Import(value = {HibernateConfiguration.class})

public class GlassfishPoolConfiguration {

    @Autowired
    private Environment environment;

    @Bean(name = "jndiDataSource")
    public DataSource dataSource() {
        String jndiName = environment.getRequiredProperty("jndi.socioc");
        JndiDataSourceLookup lookup = new JndiDataSourceLookup();
        lookup.setResourceRef(true);
        return lookup.getDataSource(jndiName);
    }
```

```
}
```

El fitxer de propietats només ha de contenir les propietats que configuren Hibernate i la propietat que indica el nom del recurs JNDI.

```
jndi.socioc = jdbc/socioc
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.show_sql = true
hibernate.format_sql = true
hibernate.hbm2ddl = validate
```

Si us hi fixeu, les propietats d'Hibernate són les mateixes que les que vau utilitzar per a la connexió MySQL. Com que no voleu tenir codi ni configuració repetides, refactoritzareu els fitxers de propietats. Creareu el fitxer hibernate.properties amb les propietats d'Hibernate, jdbc.properties amb les propietats específiques per a JDBC i glassfish.properties amb les propietats de Glassfish.

hibernate.properties:

```
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.show_sql = true
hibernate.format_sql = true
hibernate.hbm2ddl = validate
```

jdbc.properties:

```
jdbc.driverClassName = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://192.168.99.100:32768/socioc
jdbc.username = root
jdbc.password = root
```

glassfish.properties:

```
jndi.socioc = jdbc/socioc
```

A continuació us heu d'assegurar que les classes de configuració inclouen els fitxers de propietats adients.

```
@PropertySource(value = {"glassfish.properties","hibernate.properties" })
@Import(value = {HibernateConfiguration.class})
public class GlassfishPoolConfiguration {

@PropertySource(value = {"jdbc.properties", "hibernate.properties"})
@Import(value = {HibernateConfiguration.class})
public class HibernateMysqlConfiguration {
```

Finalment, voleu testejar que la nova configuració funciona i que podeu escriure a la BD MySQL. El problema és que el *bean* de tipus `DataSource` configurat per utilitzar el recurs JDBC a Glassfish, si l'aplicació no està desplegada al servidor Glassfish, no el trobarà. Hi diverses alternatives: una podria ser utilitzar un servidor Glassfish que s'executi com a part del test utilitzant Glassfish-embedded. Un altra aproximació, que és molt més flexible, ràpida i que ens permet testejar la funcionalitat de Spring, Hibernate i la nostra aplicació, és ampliar la funcionalitat de la classe `Spring` que executa els tests JUnit perquè sigui ella la que s'encarregui que el test tingui disponible recursos JNDI. La idea és molt senzilla: crear un

context JNDI on registrareu un *bean* que després utilitzareu al test. Això ho podeu fer amb la classe `org.ioc.daw.SpringJNDIRunner`, que formarà part dels tests.

```
1  package org.ioc.daw;
2
3  import javax.naming.NamingException;
4  import org.ioc.daw.config.HibernateMysqlConfiguration;
5  import org.junit.runners.model.InitializationError;
6  import org.springframework.context.ApplicationContext;
7  import org.springframework.context.annotation.
       AnnotationConfigApplicationContext;
8  import org.springframework.mock.jndi.SimpleNamingContextBuilder;
9  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10
11 public class SpringJNDIRunner extends SpringJUnit4ClassRunner {
12     public static boolean isJNDIactive;
13     public SpringJNDIRunner(Class<?> klass) throws InitializationError,
           IllegalStateException, NamingException {
14         super(klass);
15
16         synchronized (SpringJNDIRunner.class) {
17             if (!isJNDIactive) {
18
19                 ApplicationContext applicationContext = new
                       AnnotationConfigApplicationContext(
                       HibernateMysqlConfiguration.class);
20                 SimpleNamingContextBuilder builder = new
                       SimpleNamingContextBuilder();
21                 builder.bind("jdbc/socioc", applicationContext.getBean("
                       dataSource"));
22                 builder.activate();
23
24                 isJNDIactive = true;
25             }
26         }
27     }
28 }
```

Amb `new AnnotationConfigApplicationContext` definiu un context de Spring que carrega el *bean* definit a la classe de configuració `HibernateMysqlConfiguration`. A continuació registreu el *bean* anomenat `dataSource` amb el nom JNDI jdbc/socioc. El *bean* `dataSource` és el definit a `HibernateMysqlConfiguration`, que permet la connexió amb la BD MySQL. Per executar el test només heu d'especificar la nova classe que utilitzareu per definir el context de Spring en el qual correrà el test i el fitxer on heu configurat el `DataSource` que busca el recurs JDBC utilitzant JNDI.

Assegureu-vos de buidar el contingut de la taula "Users" abans d'executar el test. Aquest és el problema de treballar amb BD reals amb els tests, que mai es pot saber en quin es troba la BD i quines dades conté abans d'executar els tests. Això pot fer que els tests fallin, no per un error en el codi, sinó per un error en les dades.

```
1 @RunWith(SpringJNDIRunner.class)
2 @ContextConfiguration(classes = {GlassfishPoolConfiguration.class})
```

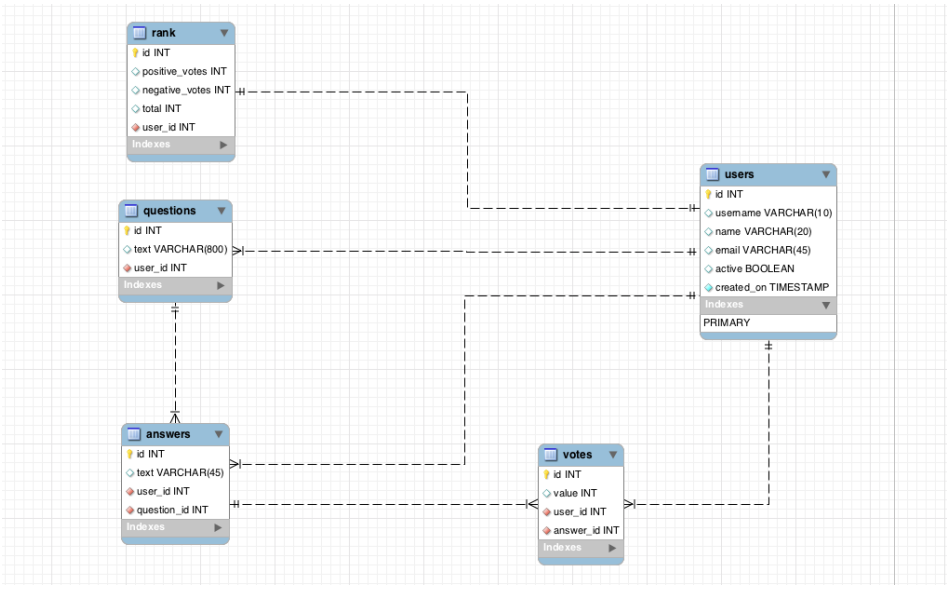
Podeu trobar el codi al fitxer que teniu disponible als annexos de la unitat.

Heu vist els avantatges d'utilitzar Spring i Hibernate, així com una breu introducció al seu funcionament. Després heu vist com configurar i integrar l'aplicació "SocIoc" amb aquests dos *frameworks* i com treballar amb diferents recursos JDBC, ja sigui en memòria, connectant directament amb MySQL o utilitzant Glassfish i l'accés als seus recursos utilitzant JNDI.

3.2 "Socloc". Dialogant amb preguntes i respostes

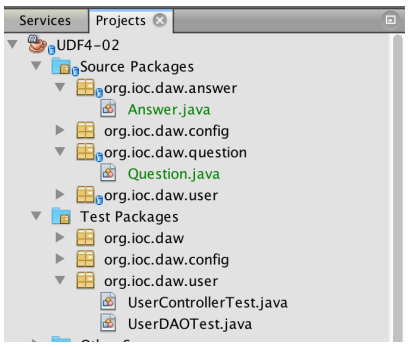
A contnuació aprofundirem més a fons en Hibernate i explorar les relacions entre els objectes i com es guardaran en la BD. Per fer-ho treballarem amb les preguntes, respostes i vots de l'aplicació SocIoc. Recordeu les taules de la base de dades "SocIoc" i com es relacionen. En la figura 3.7 podeu veure que les preguntes (taula "Questions") poden tenir més d'una resposta (taula "Answers"), i que un usuari pot fer més d'una pregunta i contestar moltes altres.

Figura 3.7. Contingut de la taula "Users"



Començareu creant les classes que representen les preguntes i respostes (vegeu la figura 3.8).

Figura 3.8. Classes Question i Answer




```
1 import javax.persistence.Column;
2 import javax.persistence.Entity;
3 import javax.persistence.GeneratedValue;
4 import javax.persistence.GenerationType;
5 import javax.persistence.Id;
6 import javax.persistence.Table;
7 import javax.validation.constraints.NotNull;
8 import javax.validation.constraints.Size;
9 import java.io.Serializable;
```

```
@Entity
@Table(name = "answers")
public class Answer implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @NotNull
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer answerId;

    @NotNull
    @Size(max = 45)
    @Column(name = "text")
    private String text;

    public Integer getAnswerId() {
        return answerId;
    }

    public void setAnswerId(Integer answerId) {
        this.answerId = answerId;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}

import org.ioc.daw.answer.Answer;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.io.Serializable;
import java.util.Set;
@Entity
@Table(name = "questions")
public class Question implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @NotNull
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer questionId;

    @NotNull
    @Size(max = 800)
    @Column(name = "text")
    private String text;

    @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
    private Set<Answer> answers;

    public Integer getQuestionId() {
        return questionId;
    }

    public void setQuestionId(Integer questionId) {
        this.questionId = questionId;
```

```
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public Set<Answer> getAnswers() {
        return answers;
    }

    public void setAnswers(Set<Answer> answers) {
        this.answers = answers;
    }
}
```

Com podeu veure, a la classe `Question` s'ha definit la relació amb les respostes amb la notació `@OneToMany`, que indica que una pregunta podrà tenir moltes respostes. `CascadeType.ALL` indica que si volem persistir un objecte que té com a atribut algun objecte que no està persistit, i per tant gestionat per Hibernate, el persistirem. Per exemple, si voleu guardar un objecte `User` que té una pregunta que no està guardada, la guardarà. De la mateixa manera, si esborreu un usuari que té preguntes, totes les preguntes s'esborraran. `FetchType` indica l'estratègia que s'utilitzarà per carregar les dades, `FetchType.EAGER` recuperarà les dades immediatament i `FetchType.LAZY` ho farà quan faci falta. S'ha de tenir present que per poder utilitzar `FetchType.LAZY` la sessió ha d'estar encara oberta quan s'intenti accedir a les dades. En el vostre cas `FetchType.EAGER` és suficient, però s'ha d'estudiar cada cas per veure quina estratègia de càrrega de dades s'utilitza.

```
@OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
    private Set<Answer> answers;
```

Fixeu-vos que a la classe `Question` no hi ha cap referència als usuaris; com es fa llavors per indicar la relació? Les preguntes no tenen usuaris, una pregunta estarà formulada per un usuari, però serà l'entitat de tipus usuari la que pot tenir múltiples preguntes. Definiu aquesta relació a la classe `User`. De la mateixa manera, afegiu la relació amb `Answer`.

```
@OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
    private Set<Question> questions;

  @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
  private Set<Answer> answers;

  public Set<Question> getQuestions() {
      return questions;
  }

  public void setQuestions(Set<Question> questions) {
      this.questions = questions;
  }

  public Set<Answer> getAnswers() {
      return answers;
  }

  public void setAnswers(Set<Answer> answers) {
      this.answers = answers;
```

```
  }
```

Un cop definides les noves entitats, heu de canviar la configuració de `HibernateConfiguration` per tal que Hibernate escanegi els nous paquets.

```
public class HibernateConfiguration {
....
    @Bean
    @Autowired
    public LocalSessionFactoryBean sessionFactory(DataSource dataSource) throws
         NamingException {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource);
        sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.
            question", "org.ioc.daw.answer");
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }
  ....
```

Un cop teniu definides les relacions, definireu els objectes per accedir a les dades (DAO). Creeu `org.ioc.daw.question.QuestionDAO` i la seva implementació `org.ioc.daw.question.QuestionHibernateDAO`.

```
public interface QuestionDAO {
    Question getById(Integer questionId);
    void save(Question question);
    Question update(Question question);
}

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import javax.transaction.Transactional;

@Transactional
@Repository("questionHibernateDAO")
public class QuestionHibernateDAO implements QuestionDAO {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public Question getById(Integer questionId) {
        return getSession().get(Question.class, questionId);
    }

    @Override
    public void save(Question question) {
        getSession().saveOrUpdate(question);
    }

   @Override
    public Question update(Question question) {
       return (Question) getSession().merge(question);
    }

    protected Session getSession() {
        return sessionFactory.getCurrentSession();
    }

}
```

I declareu el nou bean a `org.ioc.daw.config.DAOConfig`.

```
@Bean
  public QuestionDAO questionDAO(){
      return new QuestionHibernateDAO();
  }
```

Amb aquest DAO que només permet guardar i trobar preguntes pel seu identificador, com podreu trobar per exemple totes les preguntes d'un usuari o afegir una resposta a una pregunta? Com que els objectes estan relacionats, per fer alguna d'aquestes operacions utilitzareu una combinació de diferents DAO. Creareu, al paquet `org.ioc.daw.question`, la interfície `QuestionService` i la seva implementació `QuestionServiceImpl`, però primer afegireu un mètode a `UserDAO` que permeti obtenir un usuari utilitzant el seu *id*.

```
public interface UserDAO {
  ....

    public User getById(Integer id);
  ...
}
public class UserDAOJPA implements UserDAO {
  .....

    @Override
    public User getById(Integer id) {
        try {
            return (User) entityManager.createQuery("select object(o) from User
                  o " +
                     "where o.id = :id")
                     .setParameter("id", id)
                     .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
 .....
}

@Repository("userHibernateDAO")
public class UserHibernateDAO implements UserDAO {
 .....
    @Override
    public User getById(Integer userId) {
        return getSession().get(User.class, userId);
    }
 .....
}
```

```
public interface QuestionService {
    public Set<Question> getAllQuestions(Integer userId);
    public void addAnswer(Answer answer, Integer questionId, Integer userId);
    public void create(Question question, Integer userId);
}

import org.ioc.daw.answer.Answer;
import org.ioc.daw.user.User;
import org.ioc.daw.user.UserDAO;
import org.springframework.beans.factory.annotation.Autowired;
import javax.transaction.Transactional;
import java.util.HashSet;
import java.util.Set;

@Transactional
public class QuestionServiceImpl implements QuestionService {
    @Autowired
    private UserDAO userDAO;
```

```

    @Autowired
    private QuestionDAO questionDAO;


    @Override
    public Set<Question> getAllQuestions(Integer userId) {
        User user = userDAO.getById(userId);
        return user.getQuestions();
    }

    @Override
    public Question addAnswer(Answer answer, Integer questionId, Integer userId
        ) {
        User user = userDAO.getById(userId);
        Set<Answer> userAnswers = user.getAnswers();
        addAnswerToCollection(answer, userAnswers);
        user.setAnswers(userAnswers);
        userDAO.create(user);

        Question question = questionDAO.getById(questionId);
        Set<Answer> answers = question.getAnswers();
        answers = addAnswerToCollection(answer, answers);
        question.setAnswers(answers);
        return questionDAO.update(question);
    }

    private Set<Answer> addAnswerToCollection(Answer answer, Set<Answer>
         answers) {
        if (answers != null) {
            answers.add(answer);
        } else {
            answers = new HashSet<Answer>();
            answers.add(answer);
        }
        return answers;
    }

    @Override
    public void create(Question question, Integer userId) {
        User user = userDAO.getById(userId);
        Set<Question> questions = user.getQuestions();
        if (questions != null) {
            questions.add(question);
        } else {
            questions = new HashSet<>();
            questions.add(question);
            user.setQuestions(questions);
        }
        userDAO.create(user);
    }
}
```

Creeu un nou fitxer de configuracions org.ioc.daw.config.ServicesConfig i declareu el *now bean*.

```
@Configuration
  public class ServicesConfig {
    @Bean
    public QuestionService questionService(){
          return new QuestionServiceImpl();
      }
    @Bean
    public UserService userService(UserDAO userDAO) {
        return new UserService(userDAO);
    }
  }
```

Comproveu que estem injectant dependències de forma diferent als *beans* `QuestionService` i `UserController`. A `UserController` esteu injectant `UserDAO` al constructor, mentre que a `QuestionService` ho feu amb la notació `@Autowired`, que injectarà els *beans* utilitzant els *setters*. Què és millor, utilitzar injecció per a *setters* o per a constructors? No és qüestió de millor ni pitjor, i realment depèn de cada cas. De forma general, si les dependències que esteu injectant són imprescindibles per al funcionament de la classe es recomana utilitzar injecció mitjançant constructors; per tant, refactoritzarem la classe `QuestionServiceImpl`. La classe que configura els *beans* de servei quedarà de la següent forma:

```
@Configuration
public class ServicesConfig {
    @Bean
    public QuestionService questionService(UserDAO userDAO, QuestionDAO
        questionDAO) {
        return new QuestionServiceImpl(userDAO, questionDAO);
    }

    @Bean
    public UserService userService(UserDAO userDAO) {
        return new UserService(userDAO);
    }
}
```

```
public class QuestionServiceImpl implements QuestionService {
    private UserDAO userDAO;
    private QuestionDAO questionDAO;

    public QuestionServiceImpl(UserDAO userDAO, QuestionDAO questionDAO) {
        this.userDAO = userDAO;
        this.questionDAO = questionDAO;
    }
```

A continuació creareu el test `QuestionDAOTest` per comprovar que les preguntes es guarden correctament a la base de dades.

```
import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
import org.ioc.daw.config.ServicesConfig;
import org.ioc.daw.user.User;
import org.ioc.daw.user.UserDAO;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import java.sql.Timestamp;
import java.util.Date;
import java.util.Set;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ServicesConfig.class,
    EmbeddedDatabaseTestConfig.class})
public class QuestionDAOTest {
    @Autowired
    private QuestionService questionService;

    @Autowired
    private UserDAO userDAO;

@Test
```

```
26      public void createQuestion() {
27          User user = new User();
28          user.setUsername("test");
29          user.setActive(true);
30          user.setEmail("email@test.com");
31          user.setPassword("password");
32          user.setName("name");
33          user.setRank(10);
34          user.setCreatedOn(new Timestamp(new Date().getTime()));
35          userDAO.create(user);
36          Question question = new Question();
37          question.setText("This is a question");
38
39          questionService.create(question, user.getUserId());
40          assertNotNull(question.getQuestionId());
41
42          Set<Question> questions = questionService.getAllQuestions(user.
                getUserId());
43          assertEquals(1, questions.size());
44      }
45  }
```

Podeu trobar aquest codi al fitxer que teniu disponible als annexos de la unitat.

A continuació comprovareu si la vostra aplicació seria capaç de guardar les dades a la base de dades "SocIoc" que vau crear a MySQL. Per fer-ho només fa falta canviar el `ContextConfiguration` del test.

```
1  @ContextConfiguration(classes = {ServicesConfig.class,
       HibernateMysqlConfiguration.class})
```

Si executeu el test veureu el següent error on s'indica que la taula "questions_answers" no existeix.

```
1  Caused by: org.springframework.beans.factory.BeanCreationException: Error
       creating bean with name 'sessionFactory' defined in org.ioc.daw.config.
       HibernateConfiguration: Invocation of init method failed; nested exception
        is org.hibernate.tool.schema.spi.SchemaManagementException: Schema-
       validation: missing table [users_answers]
```

D'on surt aquesta taula? Per defecte, Hibernate utilitza taules intermèdies per desnormalitzar la base de dades. Això no té gaire a veure amb Hibernate *per se*, però sí amb el correcte disseny de la BD. En la taula 3.1 podeu veure com quedarien les dades de la taula "Questions" amb l'exemple d'un usuari que hagués fet diverses preguntes.

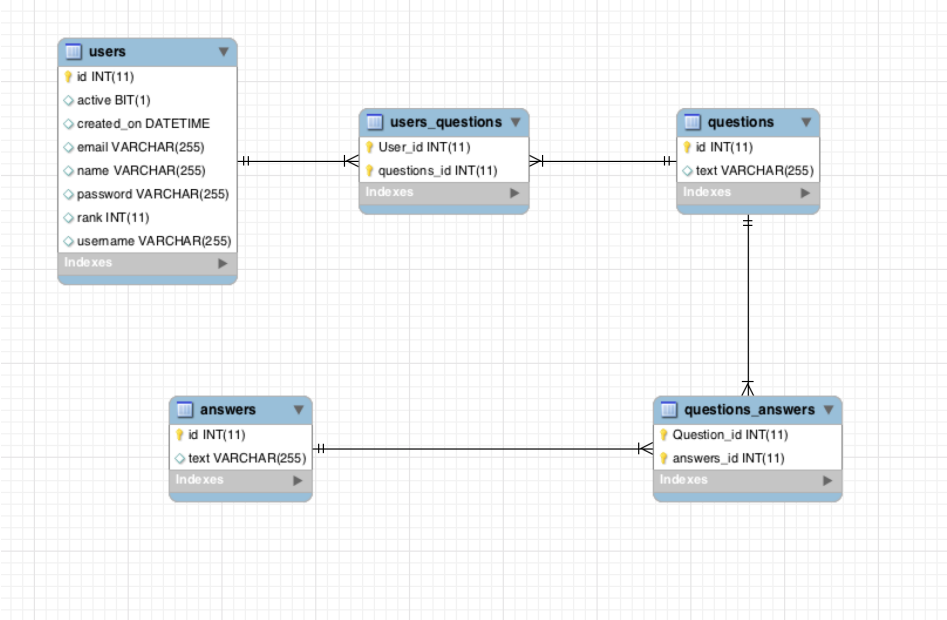
Taula 3.1. Taula "Questions"

id	text	user_id
1	pregunta 1	2
2	pregunta 2	2
3	pregunta 3	4
4	pregunta 4	2

Podeu veure que el fet que la taula tingui el camp "user_id" fa que no representi només les dades d'una pregunta, sinó que hi ha també informació dels usuaris. Això trenca la segona norma de normalització de les base de dades, que diu que totes les columnes han de ser dependents de la clau principal. És a dir, si a la pregunta "aquesta columna descriu el que la clau principal identifica?" la resposta és no, llavors vol dir que la columna no és dependent de la clau principal. En el vostre cas, "la columna 'user_id' descriu el que la clau principal 'id' identifica (una

pregunta)?", clarament no. Això implica que aquesta taula no està normalitzada. En la figura 3.9 podeu veure com es podria aconseguir la normalització per a les taules "Questions", "Users" i "Answers".

Figura 3.9. Normalització de les taules



Amb les noves taules, les dades que relacionen preguntes i usuaris quedarien tal com es pot veure en la taula 3.2 i la taula 3.3.

Taula 3.2. Taula "Users questions"

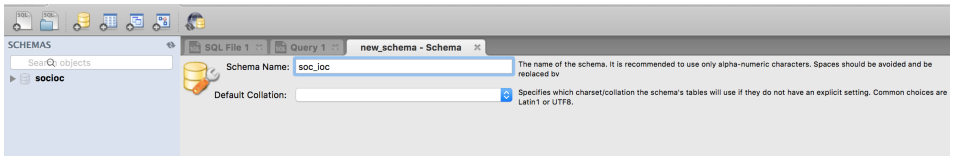
user_id	question_id
2	1
2	2
4	3
2	4

Taula 3.3. Taula "Questions"

id	text
1	pregunta 1
2	pregunta 2
3	pregunta 3
4	pregunta 4

Un dels beneficis d'utilitzar Hibernate és que es pot encarregar d'això. El que fareu serà crear un nou esquema a MySQL que utilitzareu per fer que Hibernate s'encarregui de la generació de les taules necessàries. Creeu un nou esquema utilitzant MySQL Workbench anomenat "soc_ioc", tal com es mostra en la figura 3.10.

Figura 3.10. Schema soc_ioc



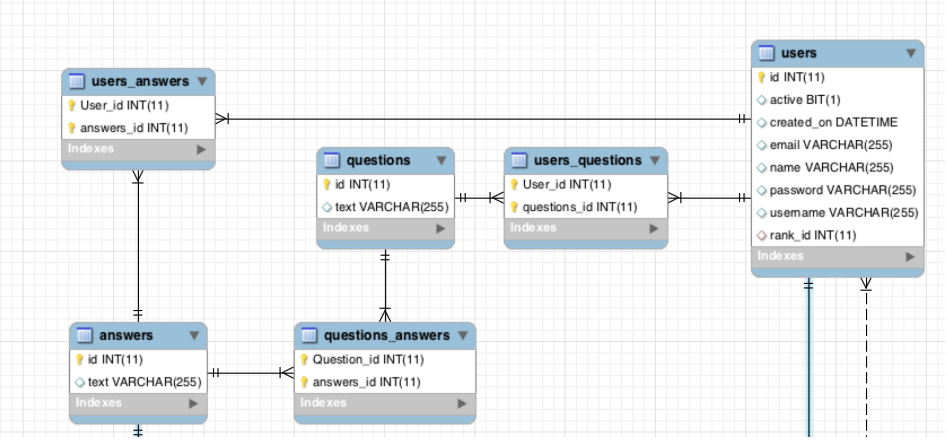
Canvieu l'URL de connexió JDBC per utilitzar el nou esquema modificant el paràmetre `jdbc.url` del fitxer jdbc.properties i modifiqueu la propietat `hibernate.hbm2ddl` de hibernate.properties perquè creï les taules automàticament, però que no modifiqui les taules si hi ha canvis a les entitats.

```
jdbc.url = jdbc:mysql://192.168.99.100:32768/soc_ioc
```

```
hibernate.hbm2ddl = update
```

Si ara executeu el test `org.ioc.daw.question.QuestionDAOTest#createQuestion` comprovareu que no hi ha cap error. Si comproveu a MySQL Worbench veureu que Hibernate haurà creat les taules a partir de les entitats que hem definit, tal com podeu veure en la figura 3.11.

FIGURA 3.11. Taules creades per Hibernate



Ara que sabeu que el codi és capaç d'escriure i llegir dades correctament a la BD MySQL, tornareu a canviar la BD utilitzada als tests per a la BD en memòria H2 i hi afegireu més tests.

```
@Test
  public void addAnswer() {
      User user = new User();
      user.setUsername("test");
      user.setActive(true);
      user.setEmail("email@test.com");
      user.setPassword("password");
      user.setName("name");
      user.setCreatedOn(new Timestamp(new Date().getTime()));
      userDAO.create(user);
      Question question = new Question();
      question.setText("This is a question");

      questionService.create(question, user.getUserId());

      Answer answer = new Answer();
      answer.setText("This is an answer");
      question = questionService.addAnswer(answer, question.getQuestionId(),
          user.getUserId());

      assertEquals(1, question.getAnswers().size());
  }
```

Fixeu-vos en un detall: no heu definit cap objecte `AnswersDAO` i heu pogut persistir una resposta. Recordeu que això és possible gràcies al fet que hem utilitzat

`CascadeType.ALL`, que automàticament s'encarregarà de persistir els objectes que no estiguin associats a una sessió d'Hibernate.

A continuació escriureu el test per a la classe `QuestionService`. En aquest cas el que voldrem fer serà injectar objectes *mock* per a `UserDAO` i `QuestionDAO` i que el test comprovi que la lògica és correcta. La configuració dels objectes *mocks* és a la classe `SpringTestConfig`.

```
@Configuration
@EnableTransactionManagement
@Import(value = {ServiceConfig.class})
public class SpringTestConfig {
    @Bean
    public UserDAO userDAO() {
        return Mockito.mock(UserDAO.class);
    }

    @Bean
    public QuestionDAO questionDAO() {
        return Mockito.mock(QuestionDAO.class);
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return Mockito.mock(PlatformTransactionManager.class);
    }
}
```

Un cop la configuració dels *beans* està preparada, ja podeu escriure el test.

```
import org.ioc.daw.answer.Answer;
import org.ioc.daw.config.SpringTestConfig;
import org.ioc.daw.user.User;
import org.ioc.daw.user.UserDAO;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.*;


@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringTestConfig.class})
public class QuestionServiceTest {
    @Autowired
    private UserDAO userDAOMock;

    @Autowired
    private QuestionDAO questionDAOMock;

    @Autowired
    private QuestionService questionService;

    @Test
    public void getAllQUestions() {
        Question question1 = getDummyQuestion(1);
        Set<Question> questions = new HashSet<>();
```

```
38          questions.add(question1);
39
40          int userId = 1;
41          User userMock = Mockito.mock(User.class);
42          when(userMock.getQuestions()).thenReturn(questions);
43
44          when(userDAOMock.getById(userId)).thenReturn(userMock);
45
46          Set<Question> questionsResponse = questionService.getAllQuestions(
                userId);
47          assertEquals(1, questionsResponse.size());
48      }
49
50      @Test
51      public void addTheFirstAnswerToAQuestion() {
52          int userID = 1;
53          int questionId = 1;
54          int answerId = 1;
55          Answer answer = getDummyAnswer(answerId);
56          Question question1 = getDummyQuestion(questionId);
57          User user1 = getDummyUser(userID);
58
59          when(userDAOMock.getById(userID)).thenReturn(user1);
60          when(questionDAOMock.getById(questionId)).thenReturn(question1);
61          when(questionDAOMock.update(question1)).thenReturn(question1);
62
63          Question questionResponse = questionService.addAnswer(answer,
                questionId, userID);
64          assertEquals(1, questionResponse.getAnswers().size());
65          verify(userDAOMock, Mockito.times(1)).create(user1);
66      }
67
68
69      @Test
70      public void addTheAnswerToAQuestionOnceItHasSomeAnswers() {
71          int questionId = 1;
72          int userID = 1;
73
74          Answer answer1 = getDummyAnswer(1);
75          Answer answer2 = getDummyAnswer(2);
76          Question question1 = getDummyQuestion(questionId);
77          Set<Answer> answers = new HashSet<>();
78          answers.add(answer1);
79          question1.setAnswers(answers);
80
81          List<Question> questions = new ArrayList<>();
82          questions.add(question1);
83
84          User user1 = getDummyUser(userID);
85          when(userDAOMock.getById(userID)).thenReturn(user1);
86          when(questionDAOMock.getById(questionId)).thenReturn(question1);
87          when(questionDAOMock.update(question1)).thenReturn(question1);
88
89          questionService.addAnswer(answer2, questionId, userID);
90          verify(userDAOMock, Mockito.times(1)).create(user1);
91          assertEquals(2, answers.size());
92      }
93
94      @Test
95      public void createFirstUserQuestion() {
96          int userId = 1;
97          Question question = getDummyQuestion(1);
98          User user = getDummyUser(1);
99          when(userDAOMock.getById(userId)).thenReturn(user);
100
101         questionService.create(question, userId);
102
103         verify(userDAOMock, timeout(1)).create(user);
104         assertEquals(1, user.getQuestions().size());
105     }
```

```
106
107
108     @Test
109     public void createUserQuestionWhenItsNotTheFirstOne() {
110         int userId = 1;
111         Question question1 = getDummyQuestion(1);
112         Question question2 = getDummyQuestion(2);
113         User user = getDummyUser(1);
114         Set<Question> questions = new HashSet<>();
115         questions.add(question1);
116         user.setQuestions(questions);
117         when(userDAOMock.getById(userId)).thenReturn(user);
118
119         questionService.create(question2, userId);
120
121         verify(userDAOMock, timeout(1)).create(user);
122         assertEquals(2, questions.size());
123     }
124
125     private Question getDummyQuestion(int questionId) {
126         Question question1 = new Question();
127         question1.setQuestionId(questionId);
128         question1.setText("Some question");
129         Set<Question> questions = new HashSet<>();
130         questions.add(question1);
131         return question1;
132     }
133
134     private Answer getDummyAnswer(int answerId) {
135         Answer answer = new Answer();
136         answer.setAnswerId(answerId);
137         answer.setText("This is an answer");
138         return answer;
139     }
140
141     private User getDummyUser(int userId) {
142         String username = "test";
143         User user = new User();
144         user.setUsername(username);
145         user.setUserId(userId);
146         return user;
147     }
148 }
```

Una cosa que heu de comprovar és la diferència de què passa quan creem preguntes o respostes per primer cop. Si és el primer cop, la col·lecció s'ha de crear. Agafem el test `createUserQuestionWhenItsNotTheFirstOne` com a exemple. A la línia 86 afegim una llista de preguntes a l'objecte `usuari`. Com que la llista existirà, el que farà `QuestionService` serà afegir un element nou, però l'objecte `llista` serà el mateix que tenia l'usuari. Per això podem comprovar que la llista creada té un objecte més. Al test `createFirstUserQuestion`, l'usuari no té cap pregunta, llavors un nou objecte de tipus llista es crearà i s'associarà a l'objecte `user`, que és el que comprovem a la línia 74.

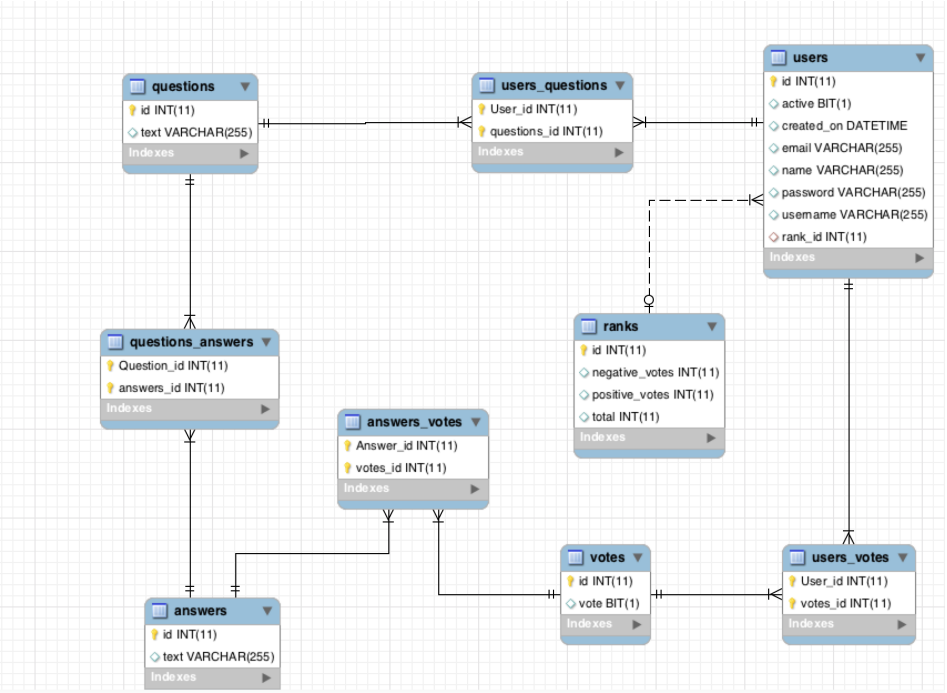
Al fitxer que teniu disponible als annexos de la unitat podeu trobar el codi d'aquest apartat.

3.3 "Socloc". Dialogant amb usuaris, rangs i vots

Un usuari tindrà associat un rang, que serà el resultat d'un càlcul dels vots rebuts per les respostes d'un usuari. Els vots poden ser positius o negatius. Així,

un usuari tindrà associada una sèrie de vots, que estaran associats a diferents preguntes. D'aquesta manera, un usuari tindrà múltiples vots i una resposta també pot tenir múltiples vots. Vegeu en la figura 3.12 com serà la relació entre les taules.

FIGURA 3.12. Taules normalitzades



La taula "Votes" està relacionada amb "Answers" i "Users" no directament, però amb unes taules intermèdies . Fixeu-vos també en com la taula "Users" es relaciona amb *rank*, hi ha un camp "user_id" i no hi ha cap taula intermèdia. El motiu és que la relació d'un usuari amb el seu *rank* és 1 a 1, és a dir, cada usuari tindrà només un *rank* i cada *rank* pertany només a un usuari. Vegeu com es representa això en el codi Java de la vostra aplicació. Creeu la classe `Rank` al paquet `org.ioc.daw.rank`. Com podeu veure, l'atribut `total` el calcularem a partir dels vots positius i negatius. Cada cop que s'actualitzi el valor dels vots, s'actualitzarà el total.

```
1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.GenerationType;
5  import javax.persistence.Id;
6  import javax.persistence.Table;
7  import javax.validation.constraints.NotNull;
8  import java.io.Serializable;
9
10 @Entity
11 @Table(name = "ranks")
12 public class Rank implements Serializable {
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @NotNull
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     @Column(name = "id")
19     private Integer rankId;
20
21     @Column(name = "positive_votes")
```

```
22      private int positive;
23
24      @Column(name = "negative_votes")
25      private int negative;
26
27      @Column(name = "total")
28      private int total;
29
30      public int getPositive() {
31          return positive;
32      }
33
34      public void setPositive(int positive) {
35          this.positive = positive;
36          this.total = positive - getNegative();
37      }
38
39      public int getNegative() {
40          return negative;
41      }
42
43      public void setNegative(int negative) {
44          this.negative = negative;
45          this.total = getPositive() - negative;
46      }
47
48      public Integer getRankId() {
49          return rankId;
50      }
51
52      public void setRankId(Integer rankId) {
53          this.rankId = rankId;
54      }
55
56      public int getTotal() {
57          return total;
58      }
59
60      public void setTotal(int total) {
61          this.total = total;
62      }
63  }
```

A la classe "User" hi afegiu l'atribut `rank` amb el seu *getter* i *setter*, i també el nou paquet a escanejar per Hibernate a `HibernateConfiguration`. En la definició de la interfície `UserDAO` hi ha un mètode relacionat amb el rang. L'implementareu i hi afegireu un test.

```
@OneToOne(cascade = {CascadeType.ALL})
  private Rank rank;
  public Rank getRank() {
      return rank;
  }
  public void setRank(Rank rank) {
      this.rank = rank;
  }
```

```
sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.question",
                "org.ioc.daw.answer", "org.ioc.daw.rank");
```

La implementació que tenim de `UserHibernateDAO#findActiveUsers` està basada en quan la classe `User` tenia un atribut de tipus *enter* que tenia el rang de l'usuari. Ara aquesta implementació no funcionarà, ja que l'atribut que conté el rang d'un usuari és una classe, i a la BD, una altra taula. Per obtenir quin és

l'usuari que té un major rang ho podríeu fer mitjançant el llenguatge de consultes d'Hibernate (HQL); el problema és que si canvieu d'implementació de JPA haureu de tornar a escriure totes les consultes de nou. Una solució és utilitzar consultes JPA.

```
@Override
    public User findUserWithHighestRank() {
        Criteria criteria = createEntityCriteria();
        criteria.addOrder(Order.desc("rank"));
        return (User) criteria.uniqueResult();
    }
```

JPA permet crear consultes amb la classe `CriteriaBuilder`. El que primer indicareu serà de quina classe voldreu fer la consulta (línia 5) i què és el que retornarà la consulta (línia 4) i com s'ordenaran els resultats retornats (línia 7). En aquest cas, l'ordre serà descendent *(cb.desc*) i estarà ordenat per l'atribut total de la classe `Rank`, que és l'atribut `rank` a la classe `User`. Finalment, indiqueu que només voleu retornar el primer resultat (`setMaxResults(1)`) i que per tant aquesta consulta només ha de retornar un únic resultat (`getSingleResult()`).

```
@Override
    public User findUserWithHighestRank() {
        CriteriaBuilder cb = createCriteriaBuilder();
        CriteriaQuery<User> criteriaQuery = cb.createQuery(User.class);
        Root<User> root = criteriaQuery.from(User.class);
        criteriaQuery.select(root);
        criteriaQuery.orderBy(cb.desc(root.join("rank").get("total")));
        EntityManager em = createEntityManager();
        return em.createQuery(criteriaQuery).setMaxResults(1).getSingleResult()
            ;
    }
   private CriteriaBuilder createCriteriaBuilder() {
        return getSession().getEntityManagerFactory().getCriteriaBuilder();
    }

    private EntityManager createEntityManager() {
        return getSession().getEntityManagerFactory().createEntityManager();
    }
```

A continuació creeu la classe `org.ioc.daw.vote.Vote`.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import java.io.Serializable;

@Entity
@Table(name = "votes")
public class Vote implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @NotNull
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer voteId;

    @Column(name = "vote")
    private Boolean vote;
```

```
23
24      public Integer getVoteId() {
25          return voteId;
26      }
27
28      public void setVoteId(Integer voteId) {
29          this.voteId = voteId;
30      }
31
32      public Boolean getVote() {
33          return vote;
34      }
35
36      public void setVote(Boolean vote) {
37          this.vote = vote;
38      }
39  }
```

Afegiu el paquet a la configuració d'Hibernate (`HibernateConfiguration`) per tal que l'escanegi.

```
1  sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.question",
2                  "org.ioc.daw.answer", "org.ioc.daw.rank", "org.ioc.daw.vote");
```

Abans de crear el les classes relacionades amb els vots creareu un objecte DAO per a "Answers" i afegireu la relació entre respostes i vots a la classe `Answer`. Creareu la interfície `AnswersDAO` i la seva implementació `AnswerHibernateDAO`, i afegireu el nou DAO a `DAOConfig` i el *mock* a `SpringTestConfig`.

```
1  public class Answer implements Serializable {
2  ....
3      @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
4      private Set<Vote> votes;
5
6      public Set<Vote> getVotes() {
7          return votes;
8      }
9
10     public void setVotes(Set<Vote> votes) {
11         this.votes = votes;
12     }
13 ....
14 }
15
16
17 public interface AnswerDAO {
18     Answer getById(Integer questionId);
19     void save(Answer question);
20 }
21
22 @Transactional
23 @Repository("answerHibernateDAO")
24 public class AnswerHibernateDAO implements AnswerDAO {
25     @Autowired
26     private SessionFactory sessionFactory;
27
28     @Override
29     public Answer getById(Integer questionId) {
30         return getSession().get(Answer.class, questionId);
31     }
32
33    @Override
34     public void save(Answer answer) {
35         getSession().saveOrUpdate(answer);
36     }
37
```

```
    protected Session getSession() {
        return sessionFactory.getCurrentSession();
    }

}

@Configuration
public class DAOConfig {
   .......
    @Bean
    public AnswerDAO answerDAO(){
        return new AnswerHibernateDAO();
    }
    .......
}
```

Un usuari votarà, així que heu d'afegir la relació dels usuaris amb els vots.

```
public class User implements Serializable {
....
    @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
    private Set<Vote> votes;
    public Set<Vote> getVotes() {
        return votes;
    }
    public void setVotes(Set<Vote> votes) {
        this.votes = votes;
    }
....
```

Com sempre, ara heu de testejar que podem treballar amb l'entitat `Votes`. Però què és el que realment volem testejar? Realment no volem testejar que podem guardar un vot a la base de dades, ni mai guardarem un vot de forma aïllada. Els vots sempre estaran relacionats amb les respostes i els usuaris. Llavors, el que volem testejar és que un usuari té la capacitat de votar una pregunta i que, si ho fa, aquesta informació es guardarà a la base de dades. Per fer-ho creareu el test `VoteDAOTest`, encara que no hem creat la classe `VoteDAO`; el que volem testejar és que els vots es guarden a la base de dades correctament. Abans implementarem la funcionalitat per votar negativament i positivament. Creem la interfície `org.ioc.daw.vote.VoteService` i la seva implementació. A partir del l'identificador d'una pregunta, recupereu les dades de la BD i creeu el nou objecte de tipus `Vote`, i en guardar l'objecte `pregunta` es guardarà la informació del vot.

```
public interface VoteService {
    void votePositive(Integer answerId, Integer userId);
    void voteNegative(Integer answerId, Integer userId);
}

import org.ioc.daw.answer.Answer;
import org.ioc.daw.answer.AnswerDAO;
import javax.transaction.Transactional;
import java.util.HashSet;
import java.util.Set;

@Transactional
public class VoteServiceImpl implements VoteService {
    private AnswerDAO answerDAO;

    public VoteServiceImpl(AnswerDAO answerDAO, UserDAO userDAO){
        this.answerDAO = answerDAO;
        this.userDAO = userDAO;
```

```
    }

    @Override
    public void votePositive(Integer answerId, Integer userId) {
        vote(answerId, userId, true);
    }

    @Override
    public void voteNegative(Integer answerId, Integer userId) {
        vote(answerId, userId, false);
    }

    private Vote vote(Integer userId, Integer answerId, Boolean value) {
        User user = userDAO.getById(userId);
        Set<Vote> userVotes = user.getVotes();
        Vote vote = new Vote();
        vote.setVote(value);
        userVotes = getVotes(vote, userVotes);
        user.setVotes(userVotes);
        userDAO.create(user);

        Answer answer = answerDAO.getById(answerId);
        Set<Vote> votes = answer.getVotes();
        votes = getVotes(vote, votes);
        answer.setVotes(votes);
        answerDAO.save(answer);
        return vote;
    }

    private Set<Vote> getVotes(Vote vote, Set<Vote> votes) {
        if (votes != null) {
            votes.add(vote);
        } else {
            votes = new HashSet<Vote>();
            votes.add(vote);

        }
        return votes;
    }
```

No us heu d'oblidar d'afegir `VoteService` a `ServiceConfig` per tal que Spring creï el *bean*.

```
@Bean
  public VoteService voteService(AnswerDAO answerDAO, UserDAO userDAO){
      return new VoteServiceImpl(answerDAO, userDAO);
  }
```

Com sempre, ara heu de testejar que podeu treballar amb l'entitat `Votes`. Però què és el que realment volem testejar? Realment no volem testejar que podem guardar un vot a la base de dades, ni mai guardarem un vot de forma aïllada. Els vots sempre estaran relacionats amb les respostes i els usuaris. Llavors, el que volem testejar és que un usuari té la capacitat de votar una pregunta i que, si ho fa, aquesta informació es guardarà a la base de dades. Per fer-ho creareu el test `VoteDAOTest`, encara que no hem creat la classe `VoteDAO`; el que volem testejar és que els vots es guarden a la base de dades correctament.

En el test fareu diverses coses. Primer creeu i persistiu tres usuaris, després l'usuari "user1" crea una pregunta, l'usuari "user2" crea una resposta per a la pregunta i feu que l'usuari "user3" voti de forma positiva la pregunta. Finalment, comproveu que la resposta i l'usuari "user3" tenen un vot i que l'identificador del vot és el mateix en els dos casos.

```java
import org.ioc.daw.answer.Answer;
import org.ioc.daw.answer.AnswerDAO;
import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
import org.ioc.daw.config.ServicesConfig;
import org.ioc.daw.question.Question;
import org.ioc.daw.question.QuestionService;
import org.ioc.daw.user.User;
import org.ioc.daw.user.UserDAO;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import java.sql.Timestamp;
import java.util.Date;
import static org.junit.Assert.assertEquals;


@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ServicesConfig.class,
    EmbeddedDatabaseTestConfig.class})
public class VoteDAOTest {
    @Autowired
    private QuestionService questionService;

    @Autowired
    private UserDAO userDAO;

    @Autowired
    private AnswerDAO answerDAO;

    @Autowired
    private VoteService voteService;


    @Test
    public void votePositive() {
        User user1 = getUser("test", "test@email.com");
        User user2 = getUser("test1", "test1@email.com");
        User user3 = getUser("test2", "test2@email.com");
        userDAO.create(user1);
        userDAO.create(user2);
        userDAO.create(user3);

        Question question = new Question();
        question.setText("This is a question");
        questionService.create(question, user1.getUserId());

        Answer answer = new Answer();
        answer.setText("This is an answer");
        question = questionService.addAnswer(answer, question.getQuestionId(),
            user2.getUserId());

        int answerId = question.getAnswers().iterator().next().getAnswerId();
        voteService.votePositive(user3.getUserId(), answerId);

        User userDB = userDAO.getById(user3.getUserId());
        Answer answerDB = answerDAO.getById(answerId);
        assertEquals(1, userDB.getVotes().size());
        assertEquals(1, answerDB.getVotes().size());
        assertEquals(userDB.getVotes().iterator().next().getVoteId(), answerDB.
            getVotes().iterator().next().getVoteId());

    }

    private User getUser(String username, String email) {
        User user = new User();
        user.setUsername(username);
        user.setActive(true);
```

```
67          user.setEmail(email);
68          user.setPassword("password");
69          user.setName("name");
70          user.setCreatedOn(new Timestamp(new Date().getTime()));
71          return user;
72      }
73  }
```

Podeu trobar el fitxer amb aquest codi als annexos de la unitat.

3.4 Què s'ha après?

Heu après que hi ha diferents *frameworks* que ens poden ajudar a l'hora de desenvolupar aplicacions. Spring ens ajuda a crear un codi més modular, reutilitzable i fàcil de testejar. Hem vist com podem canviar fàcilment quina base de dades utilitzar o els *beans* a injectar a una classe. Per una altra banda, Hibernate ens dóna les eines per treballar amb bases de dades focalitzant els esforços en el desenvolupament del codi i no en el disseny de la BD. Això no vol dir que el disseny de la BD no tingui importància; al contrari, serà fonamental per al correcte comportament de l'aplicació quan estigui a producció, però aquesta tasca serà responsabilitat de l'administrador de la BD. Hibernate ens farà més fàcil la tasca de relacionar els objectes de l'aplicació amb les taules de la base dades. També heu après a fer tests unitaris i a testejar l'aplicació utilitzant una BD en memòria emprant els *frameworks* JUnit i Mockito.