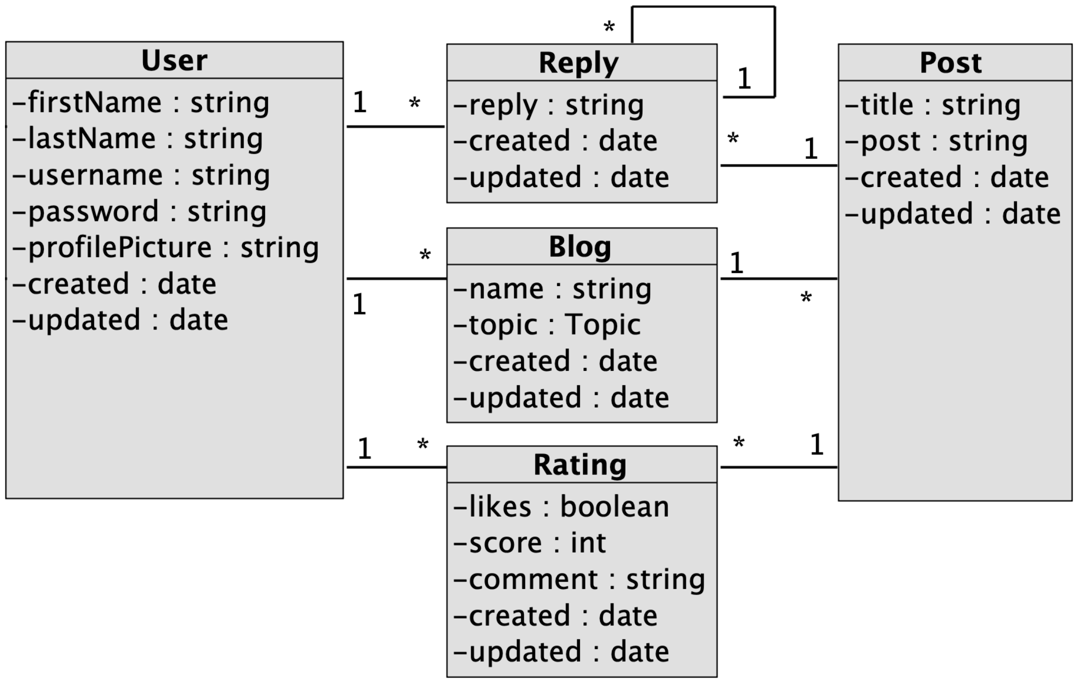
Object Relational Mapping

Introduction

In this assignment we're going to practice programming the database using a technique called Object Relational Mapping (ORM). ORMs address the challenge of bridging two distinct worlds: the relational world and the object oriented world. In the relational world data is represented in terms of tables, records, rows, and columns. Relationships between records is implemented as foreign key fields from one record referencing primary key fields of other records in some other table. Data in a relational database is stored long term, permanently, and generally contain thousands, even millions of records. In the object oriented world data is represented in terms of classes, object instances, and state variables. Relationships between objects is implemented as variables or arrays holding pointers or handles to other objects in memory. Unlike records that are stored long term, object instances are transient, often existing for just milliseconds, and generally contains a fraction of the data stored in a database. In the object oriented world we have concepts such as inheritance and interfaces, which have no equivalence in the relational world. One to many relationships in relational models are implemented with references from the child records to their common parent, e.g., child foreign keys referencing a parent primary key. This is in contrast with the object oriented world where it is parent object instances that hold a references to their children as state variables or arrays of objects. Applications are often written in high level object oriented languages such as C#, Python, or Java, but the data is stored in long term storage databases such as MySQL, Oracle, or MongoDB. The challenges of bridging the gap between object oriented data models and relational models is often referred to as the ***object–relational impedance mismatch***.



In this assignment we will address the object-relational impedance mismatch using an ORM solution called Java Persistance API (JPA) based on the popular Hibernate ORM object-relational mapping tool.

Importing a Database

In a previous assignment you created a schema called ***db_design***. Double click the schema ***db_design*** so that it is the default schema and all commands will be based on that schema. Download the data from my GitHub and import it into your ***db_design*** schema. Unzip the downloaded file into a folder. To import the data, from ***MySQL Workbench***, from the main menu, select ***Server***, ***Data Import***. In the ***Data Import*** screen, click the ***Import from Disk*** tab, and then click the ***Import from Dump Project Folder***. Click the browse button all the way on the right to select the directory you unzipped earlier. Select the folder and then click on ***Start Import***. Close the screen when done. Double click the ***db_design*** schema on the left hand side to make it the default schema.

Clone or download the sample code

We have prepared a Java project that has already been configured to connect to a MySQL database. You'll need to clone it or download the project, configure it and run it from your local computer. Download the project from https://github.com/jannunzi/db-design-orm-assignment. If you have git installed you can clone the project as follows from a terminal or console window:

```
git clone https://github.com/jannunzi/db-design-orm-assignment.git
```

Alternatively you can download the code by clicking on ***Code*** and then ***Download ZIP*** and then unzipping the file. In either case you'll end up with a new folder called ***db-design-orm-assignment***. We'll refer to this new folder as the ***root of the project***. Do all your work in the new folder.

Download and install an IDE

Although you can do your work using any text editor you prefer, we recommend you use one of the following IDEs to make your life easier. These IDEs are either free or offer a free version if you register with your university email:

- IntelliJ Ultimate (preferred)
- Eclipse Java EE
- VS Code

Users Table

In the database you imported earlier there's a table called users. The users table has the schema described below. We'll create an object oriented class that will represent object instances that correspond to records stored in the users table.

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `first_name` varchar(255) DEFAULT NULL,
  `last_name` varchar(255) DEFAULT NULL,
  `profile_picture` varchar(255) DEFAULT NULL,
  `created` datetime DEFAULT CURRENT_TIMESTAMP,
  `updated` datetime DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
);
```

User Class

We've created a class called ***User*** that mirrors the definition of the ***users*** table. The intention is that each object instance of the ***User*** class will correspond to a record in the ***users*** table. From the root of the project folder, in the directory ***src/main/java/com/example/springtemplate/models***, open the file called ***User.java*** and confirm that it declares the following class variables, as well as corresponding setters, getters and constructors.

```
package com.example.springtemplate.models;
import java.sql.Timestamp;

public class User {
    private Integer id;
    private String firstName;
    private String lastName;
    private String username;
    private String password;
    private String profilePicture;
    private String handle;
    // setters and getters here
}
```

The default constructor, the one with no arguments, is required. We've also provided for convenience a constructor that takes 5 of the class variables.

User JDBC DAO

Before learning about ORMs, let's take a look at interacting with the database using JDBC, Java's database connectivity interface implemented by most commercial databases. In ***src/main/java/com/example/springtemplate/daos***, open ***UserJdbcDao.java*** and let's review its implementation.

```
package com.example.springtemplate.daos;

import com.example.springtemplate.models.User;            // DAO will create instances of User

import java.sql.*;                                        // JDBC library
import java.util.*;                                       // Java util library

public class UserJdbcDao {                                // DAO Class
    static final String DRIVER = "com.mysql.cj.jdbc.Driver";    // MySQL Java Connector implements JDBC
    static final String HOST = "localhost:3306";                // your local database
    static final String SCHEMA = "YOUR_SCHEMA";                 // replace with your schema, e.g., db_design
    static final String CONFIG = "serverTimezone=UTC";
    static final String URL =                                   // connection string pointing to your database
          "jdbc:mysql://"+HOST+"/"+SCHEMA+"?"+CONFIG;           // server
    static final String USERNAME = "YOUR_USERNAME";             // replace with your username, e.g., cs3200
    static final String PASSWORD = "YOUR_PASSWORD";             // replace with your password, e.g., cs3200
    static Connection connection = null;                        // connection to the database
    static PreparedStatement statement = null;                  // statement to execute queries and updates

    public static void main(String[] args) throws ... {         // entry point to execute DAO standalone
       System.out.println("JDBC DAO");
    }
}
```

Connection and PreparedStatements

Method ***getConnection()*** logs into the database and returns a connection your app can use to interact with the database. When done, you should close the connection with ***closeConnection()*** so that others can use the connection and you don't run out of connections.

```
    static Connection connection = null;                             // connection to the database
    static PreparedStatement statement = null;                       // statement to execute queries
                                                                     // and updates
    private Connection getConnection() throws ... {                  // opens connection to
       Class.forName(DRIVER);                                        // database server loading driver
       return DriverManager                                          // dynamically and loging in with
                   .getConnection(URL, USERNAME, PASSWORD);          // your credentials
    }

    private void closeConnection(Connection connection) throws ... { // closes connection when you're
       connection.close();                                           // done using it
    }
```

CRUD SQL Statements

CRUD (Create Read Update Delete) operations are common and the SQL strings below provide templates we can parameterize.

```
    String CREATE_USER = "INSERT INTO users VALUES (null, ?, ?, ?, ?, ?, null)";
    String FIND_ALL_USERS = "SELECT * FROM users";
    String FIND_USER_BY_ID = "SELECT * FROM users WHERE id=?";
    String DELETE_USER = "DELETE FROM users WHERE id=?";
    String UPDATE_USER_PASSWORD = "UPDATE users SET password=? WHERE id=?";
```

CRUD Methods

The following CRUD methods parameterize the CRUD strings to implement the CRUD operations. We'll walk you through implementing these methods in the following sections to give you a chance to practice building your own DAO.

```
public Integer createUser(User user) { … }                       // inserts a user into the users table
public List<User> findAllUsers() { … }                           // selects all users from the users table
public User findUserById() { … }                                 // selects a user from users table by id
public Integer deleteUser(Integer userId) { … }                  // deletes a user from users table
public Integer updateUser(Integer userId, User newUser) { … }    // updates a user in the users table
```

Insert a user into the users table

Lets first implement the ***createUser()*** CRUD method. It'll use the ***CREATE_USER*** SQL template to insert a user instance into the database as shown below.

```
String CREATE_USER =                                           // SQL statement to insert user. Note
  "INSERT INTO users VALUES (null, ?, ?, ?, ?, ?, null)";     // the 5 place holders denoted with "?".
                                                               // We'll refer to them by index 1, 2, 3, 4, 5
public Integer createUser(User user)                           // method to insert object instance user
    throws ClassNotFoundException, SQLException {
  Integer rowsInserted = 0;
  connection = getConnection();                                // get the connection
  statement = connection.prepareStatement(CREATE_USER);        // prepare the statement
  statement.setString(1, user.getUsername());                  // set 1st placeholder with username
  statement.setString(2, user.getPassword());                  // set 2nd placeholder with password
  statement.setString(3, user.getFirstName());                 // set 3rd placeholder with first name
  statement.setString(4, user.getLastName());                  // set 4th placeholder with last name
  statement.setString(5, user.getProfilePicture());            // set 5th placeholder with picture
  rowsInserted = statement.executeUpdate();                    // execute the update, e.g., insert
  closeConnection(connection);                                 // close the connection
  return rowsInserted;                                         // return number of records inserted
}
```

To test, let's insert a couple of users. In the ***main()*** method, create an instance of ***UserJdbcDao*** and use it to insert the following users.

```
public static void main(String[] args) throws ... {
  UserJdbcDao dao = new UserJdbcDao();          // create an instance of UserJdbcDao
  User adam =                                   // create a user instance
    new User("Adam", "Smith", "adams",
      "invisiblehand", "http://bbc.in/30gXhI4");
  User thomas =                                 // create another user instance
    new User("Thomas", "Sowell", "thomas",
      "polymath", "http://www.tsowell.com/");
  User catherine =                              // create another user instance
    new User("Catherine", "Wood", "cathie",
      "bitcoinisbig", "https://ark-invest.com/");
  dao.createUser(adam);                        // insert each
  dao.createUser(thomas);                      // of the user
  dao.createUser(catherine);                   // instances above
}
```

Run ***UserJdbcDao*** and then verify the three users were inserted into the database. In IntelliJ you execute a Java class by right clicking it and then selecting Run.

Retrieve all users

Now that we've inserted data into the database, let's now try to retrieve it. Let's create ***findAllUsers()*** as shown below.

```
   String FIND_ALL_USERS = "SELECT * FROM users";                // SQL to retrieve all users

   public List<User> findAllUsers() throws ... {
      List<User> users = new ArrayList<User>();                  // initialize empty list of users
      connection = getConnection();                              // connect to the database
      statement = connection.prepareStatement(FIND_ALL_USERS);   // prepare the statement SELECT *
      ResultSet resultSet = statement.executeQuery();            // retrieve records from query
      while (resultSet.next()) {                                 // iterate over the records
         User user = new User(                                   // create user instance
               resultSet.getString("first_name"),                // with all the columns, e.g., first name
               resultSet.getString("last_name"),                 // last name
               resultSet.getString("username"),                  // username
               resultSet.getString("password"),                  // password
               resultSet.getString("profile_picture")            // profile picture
         );
         users.add(user);                                        // add user instance to list of users
      }
      closeConnection(connection);                               // close connection
      return users;                                              // return list of users representingg
   }                                                             // records in database
```

To test, let's use the same DAO from before to retrieve the records from the database and then print their usernames to the console.

```
   public static void main(String[] args) throws ... {
      UserJdbcDao dao = new UserJdbcDao();         // use the same DAO
      // User adam = ...                           // comment out the users
      // User thomas = ...
      // User catherine = ...
      // dao.createUser(adam);                     // comment out inserts so you don't
      // dao.createUser(thomas);                   // insert the same users twice
      // dao.createUser(catherine);
      List<User> users = dao.findAllUsers();       // get the list of users from the database
      for(User user: users) {                      // iterate over the list of users
         System.out.println(user.getUsername());   // print out each user's username
      }
```

```
}
```

Retrieve user by ID

Another common CRUD operation is to retrieve a single record by their primary key. Implement ***findUserById()*** as shown below.

```
String FIND_USER_BY_ID = "SELECT * FROM users WHERE id=?";     // SQL to retrieve a single user

public User findUserById(Integer id) throws ... {              // accept user's id, return found user
   User user = null;                                           // initialize found user to null
   connection = getConnection();                               // connect to the database
   statement = connection.prepareStatement(FIND_USER_BY_ID);   // create a statement from connection
   statement.setInt(1, id);                                    // set the ID of the SQL statement
   ResultSet resultSet = statement.executeQuery();             // execute query
   if(resultSet.next()) {                                      // check if there were any results
      user = new User(                                         // if there were, create user instance
            resultSet.getString("first_name"),                 // copy columns
            resultSet.getString("last_name"),                  // from record
            resultSet.getString("username"),                   // to local user
            resultSet.getString("password"),                   // instance
            resultSet.getString("profile_picture")
      );
   }
   closeConnection(connection);                                // close the connection
   return user;                                                // return found user
}
```

To test user the same DAO as before, comment out the previous test, then use the new ***findUserById()*** method to retrieve a user from the database and print out their username as shown below.

```
public static void main(String[] args) throws ... {
   UserJdbcDao dao = new UserJdbcDao();             // use the same DAO
   …
   // List<User> users = dao.findAllUsers();        // comment
   // for(User user: users) {                       // previous
   //    System.out.println(user.getUsername());    // test
   // }
   User user = dao.findUserById(1);                 // retrieve one of the users by their ID. If 1 does not
   System.out.println(user.getUsername());          // work, use another ID. Then print their username
}
```

Delete user by ID

Let's now implement deleting users from the database. We'll parameterize the DELETE_USER SQL statement with the ID of the user we want to remove as shown below.

```
String DELETE_USER = "DELETE FROM users WHERE id=?";     // SQL to delete a single user
```

```
public Integer deleteUser(Integer userId) throws ... {                // accept id of user to remove
   Integer rowsDeleted = 0;
   connection = getConnection();                                      // connect to the database
   statement = connection.prepareStatement(DELETE_USER);              // prepare the statement
   statement.setInt(1, userId);                                       // set the user ID to remove
   rowsDeleted = statement.executeUpdate();                           // execute the delete
   closeConnection(connection);                                       // close the connection
   return rowsDeleted;                                                // return now many records were deleted
}
```

To test use, use the new deleteUser() method to remove one of the users in the database, and then retrieve them all to confirm the user is no longer there:

```
public static void main(String[] args) throws ... {
   UserJdbcDao dao = new UserJdbcDao();            // use the same DAO
   ...
   dao.deleteUser(1);                              // delete one of the users by their ID. If 1 does not
   List<User> users = dao.findAllUsers();          // work, use another ID. Then retrieve all users
   for(User user: users) {                         // and print their username to
      System.out.println(user.getUsername());      // confirm user is no longer there
   }
}
```

Update a user

Finally let's try updating a user as follows

```
String UPDATE_USER = "UPDATE users SET first_name=?,                      // SQL to update a single user
                      last_name=?, username=?,                            // set new column values and
                      password=? WHERE id=?";                             // ID of user to update

public Integer updateUser(Integer userId, User newUser) throws ... {     // accept user ID and new values
   Integer rowsUpdated = 0;
   connection = getConnection();                                          // connect to database
   statement = connection.prepareStatement(UPDATE_USER);                  // prepare the statement
   statement.setString(1, newUser.getFirstName());                       // set new column
   statement.setString(2, newUser.getLastName());                        // values from the
   statement.setString(3, newUser.getUsername());                        // user object passed
   statement.setString(4, newUser.getPassword());                        // as parameter
   statement.setInt(5, userId);                                           // set user ID to update
   rowsUpdated = statement.executeUpdate();                               // execute update
   closeConnection(connection);                                           // close connection
   return rowsUpdated;                                                    // return how many users updated
}
```

To test, update one of the users already in the database as follows

```
public static void main(String[] args) throws ... {
      UserJdbcDao dao = new UserJdbcDao();            // use the same DAO
      ...
      User thomas = new User(...);                    // uncomment thomas
      User newTom = new User(                         // create a new tom
            "Tom",                                    // new first name
            "Sowell",                                 // same last name
            "tom",                                    // same username
            "knowitall",                              // new password
            thomas.getProfilePicture());              // same profile picture
      dao.updateUser(2, newTom);                      // update record
      User tom = dao.findUserById(2);                 // retrieve record to confirm
      System.out.println(tom.getUsername());          // print username to confirm

   }
```

ORM (Object Relational Mapping)

JDBC is all and good and works well when you need direct interaction with the database using specific SQL commands. JDBC allows detailed control over the mapping of a relational model and a corresponding object model. One of the downsides of JDBC is having to deal with the details of database connections, configuration of SQL statement, iterating and parsing result sets, populating the object data model from the relational model. ORMs tools such as JPA (Java Persistence API) allow developers escape much of the plumbing and boilerplate code associated with JDBC, while retaining the flexibility of mapping a relational model to an equivalent object data model. In this section we're going to use JPA to implement the same DAO as before, but using JPA's ORM implementation. The takeaway should be that it is much more simpler and convenient to use.

Configure database connection

First we'll deal with the database connection. Using JDBC we connected to the database through the DriverManager by explicitly configuring the URL, username, and password, as well as loading the driver by it's class name. Using JPA we can instead configure all that in a separate configuration file and let the server deal with all that so that we can forget about all those details. Open **/src/main/resources/application.properties** and take a look at the URL, username, and password configuration shown below.

```
spring.datasource.url=jdbc:mysql://localhost:3306/YOUR_SCHEMA?...     // replace, e.g., db_design
spring.datasource.username=YOUR_USERNAME                              // replace, e.g., cs3200
spring.datasource.password=YOUR_PASSWORD                              // replace, e.g., cs3200
```

Configuring this in a separate file is so much more convenient since it makes our code cleaner and independent of the underlying infrastructure. We can easily change the configuration without touching our code.

Annotate User as an Entity

Next we would like to automate the mapping of the records in the users table to equivalent Java object instances. Using JDBC we had to process the result set containing the records, retrieve each of the fields from the records and then populate an object instance. That's a lot of work! Using ORMs we can instead automate all that by annotating a few things in the target object model as shown below

```
import javax.persistence.*;                                 // JPA's ORM package

@Entity                                                     // configure class as mapped to a table
@Table(name="users")                                        // configure name of source table
public class User {
   @Id                                                      // configure primary key field
   @GeneratedValue(strategy = GenerationType.IDENTITY)      // configure auto_increment
   private Integer id;
   ...
}
```

That's it! The ORM will do the rest of the work for us. It will automagically parse columns of the same name and populate matching class variables. No more iterating over rows, retrieving each column with the correct data type, and creating and populating object instances ourselves. The ORM will take care of all that under the covers. How cool is that! Actually there's a little bit more to do to make the magic happen.

Create UserRepository

In JPA, ***repositories*** are responsible for actually implementing the mapping between the relational model and the object model. We just have to configure the source relational model, the target object model, and the primary key data type. Below we configure a ***CrudRepository*** with the ***User*** as the target object model, the source relational model is configured by ***@Table*** in the ***User*** class, and the primary key data type is configured below as Integer. Ok, now we are really ready to try out the magic.

```
package com.example.springtemplate.repositories;

import com.example.springtemplate.models.User;
import org.springframework.data.repository.CrudRepository;

public interface UserRepository
   extends CrudRepository<User, Integer> {
}
```

Create UserOrmDao

To illustrate how much simpler it is to use an ORM, we'll re-implement the Use DAO using the ORM and contrast with the JDBC implementation. We'll use slightly different signatures for our methods just because there's more magic to come. Open ***src/main/java/com/example/springtemplate/daos/UserOrmDao.java*** and let's implement each of the CRUD methods again, bur using JPA's ORM.

```
package com.example.springtemplate.daos;

import com.example.springtemplate.models.User;                       // we'll CRUD User instances
import com.example.springtemplate.repositories.UserRepository;       // using UserRepository
import org.springframework.beans.factory.annotation.Autowired;              // autowired into the DAO
import org.springframework.web.bind.annotation.*;
import java.util.List;

public class UserOrmDao {
   @Autowired                                                               // create an instance of
   UserRepository userRepository;                                           // the user repository
   public User createUser() { return null; }                                // and use
   public List<User> findAllUser() { return null; }                         // it to implement
   public User findUserById(Integer id) { return null; }                    // all these
   public Integer deleteUser(Integer id) { return null; }                   // CRUD
   public Integer updateUser() { return null; }                             // methods
}
```

Create User

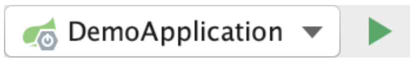
Let's implement ***createUser*** as shown below

```
   public User createUser(                               // return the user inserted into the database
         String first,                                   // use parameters
         String last,                                    // to create
         String uname,                                   // new user
         String pass) {                                  // instance
      User user = new User(first, last, uname, pass, null);   // using the constructor
      return userRepository.save(user);                  // insert into database and return resulting record
   }
```

to test whether this is working let's make the class available through HTTP and map the ***createUser*** method to a URL we can access with our browser:

```
@RestController                                          // access this resource through HTTP
public class UserOrmDao {
   ...
   @GetMapping("/orm/users/create/{fn}/{ln}/{un}/{pw}")  // pass parameters in path
   public User createUser(
         @PathVariable("fn") String first,               // parse path
         @PathVariable("ln") String last,                // variables
         @PathVariable("un") String uname,               // and pass
         @PathVariable("pw") String pass) {              // as parameters
      User user = new User(first, last, uname, pass, null);   // create instance
      return userRepository.save(user);                  // insert into database, and return new record
   }
   ...
}
```

To test, start the server and point your browser to the URL below. To start the server in IntelliJ, select ***DemoApplication*** from the ***Configurations*** dropdown and click on the triangular green run button (shown right)



```
http://localhost:8080/orm/users/create/Walter/Williams/walter/serving your fellow man
```

Confirm a new user record is created in the database. How cool is that? No need to load the driver, get a connection, create/configure a SQL statement, close the connection. None of that. We just used save() on the user repository and the ORM issued the correct INSERT statement to the database. Very nice!

Retrieve all users

Now that we have created a record using the ORM, let's now retrieve all records. Let's configure the SQL statement in the repository as shown below. SQL queries can be associated to a method by using the ***@Query*** annotation. We didn't really have to configure the SQL. The ORM has a default implementation to retrieve all the records, but it entails some type casting that makes the code look ugly. Instead, we'll tell the ORM the exact SQL we want it to use and then wrap the SQL statement with a nice high level function called findAllUsers() that hides those pesky details, i.e., applying the *information hiding principle*.

```
public interface UserRepository extends CrudRepository<User, Integer> {
  @Query(value = "SELECT * FROM users", nativeQuery = true)          // configure the SQL statement
  public List<User> findAllUsers();                                  // wrap SQL with Java interface
}
```

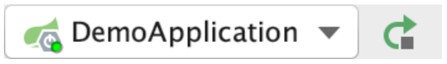
Now we can use the SQL statement as if it was a simple Java method in the repository. Implement method ***findAllUsers()*** using the new ***findallUser()*** interface in the repository as shown below:

```
public class UserOrmDao {
  ...
  public List<User> findAllUsers() {
    return userRepository.findAllUsers();
  }
  ...
}
```

To test let's map the ***findAllUser()*** method to a URL we can access from our browser as shown below:

```
  @GetMapping("/orm/users/find")
  public List<User> findAllUsers() {
    return userRepository.findAllUsers();
  }
```

Restart the server by clicking the restart button next to the ***DemoApplication*** dropdown selection, then point your browser to the URL shown below. Verify that all users are returned from the database



```
http://localhost:8080/orm/users/find                // retrieve all users
```

Find user by id

Now let's implement retrieving a single user by their primary key. In the repository implement the SQL query to retrieve the single user and then wrap it with a function called ***findUserById()*** as shown below:

```
public interface UserRepository extends CrudRepository<User, Integer> {
   @Query(value = "SELECT * FROM users", nativeQuery = true)
   public List<User> findAllUsers();
   @Query(value = "SELECT * FROM users WHERE id=:userId", nativeQuery = true)
   public User findUserById(@Param("userId") Integer id);
}
```

Note that the query needs the user ID as a parameter. We declare the parameter using a colon and a variable name, as in ":userId". We can then provide the value for the parameter by associating the parameter variable to the method parameter using the ***@Param("userId")*** annotation. Now we can execute the SQL query as if it was a simple Java method ***findUserById()***. The method takes a parameter that will eventually be part of the SQL statement.

```
   public User findUserById(Integer id) {
      return userRepository.findUserById(id);
   }
```

To test, map the new method to a URL that we can invoke from our browser. Encode the id of the user at the end of the URL and then parse it to use it in the repository to retrieve the record.

```
   @GetMapping("/orm/users/find/id/{userId}")
   public User findUserById(
         @PathVariable("userId") Integer id) {
      return userRepository.findUserById(id);
   }
```

From your browser, use the following URL to retrieve a user with a specific ID

```
http://localhost:8080/orm/users/find/id/3        // retrieve user 3 to test
                                                 // if user ID 3 doesn't work, try another
```

Delete User

Next let's remove a user by their ID. Use the repository's builtin deleteById() method to delete a record by their primary key as shown below:

```
   public void deleteUser(Integer id) {            // accept user's ID to be deleted
      userRepository.deleteById(id);               // use builtin deleteById method to remove record
   }
```

To test, map the deleteUser() method to a URL that has the user's ID at the end of the path as shown below.

```
@GetMapping("/orm/users/delete/{userId}")     // encode user's ID at the end of the path
public void deleteUser(
      @PathVariable("userId") Integer id) {          // parse the user's ID from the path and pass as parameter
   userRepository.deleteById(id);
}
```

Use your browser to delete a user by their ID. Point your browser to the URLs below to remove a specific user and then retrieve all the users to confirm that the user was removed. If the user ID shown below does not work, try using a different ID you know exists. You can always take a look at the actual records using a database client such as MySQL Workbench

```
http://localhost:8080/orm/users/delete/3    // delete user whose ID is 3. If 3 doesn't work, user another

http://localhost:8080/orm/users/find          // retrieve all users to confirm user 3 is no longer there
```

Update a user

Finally let's try updating an existing record. The way JPA updates records is by first retrieving the record you want to update from the database. Then you update the fields you want to change, and finally save the record back to the database. The ORM will issue an SQL UPDATE command that will save the changes permanently.

```
public User updateUser(
      Integer id,
      String newPass) {
   User user = userRepository.findUserById(id);
   user.setPassword(newPass);
   return userRepository.save(user);
}
```

Note that the ***save()*** method used above is the same ***save()*** method used earlier to insert the record. The method is smart enough to use either an ***INSERT*** or an ***UPDATE*** statement. The way it can tell is that if the object is a brand new, not tied to any existing record, the ***save()*** will issue an ***INSERT***. If instead the object is tied to an existing record, then ***save()*** will issue an ***UPDATE***. Smart hugh!

To test the ***updateUser()***, let's try using it to change a user's password. Let's encode the user ID and new password in the URL as shown below, and then use it to retrieve the existing user, change their password, and then save the updates back to the database.

```
@GetMapping("/orm/users/update/{userId}/{password}")
public User updateUser(
      @PathVariable("userId") Integer id,
      @PathVariable("password") String newPass) {
   User user = userRepository.findUserById(id);
   user.setPassword(newPass);
   return userRepository.save(user);
}
```

Use your browser to update a user's password using the URL below and then retrieve that user to confirm that the user was updated. If the user ID shown below does not work, try using a different ID you know exists. You can always take a look at the actual records using a database client such as MySQL Workbench

```
http://localhost:8080/orm/users/update/3/my super secure password     // change password for user 3

http://localhost:8080/orm/users/find/id/3                             // retrieve user 3 to confirm
```

Although we used the updateUser() method to change the password only, we could have written it to change any other field or set of fields.

Deliverables

As a deliverable, zip up your project and upload it to Canvas. Before you zip the whole directory, remove temporary directories that contain unnecessary files. Make sure to remove the target directory or any other build directories created by your IDE. Your code should contain the implementation for the following Java classes:

1. User.java
    a. all properties
    b. all setters and getters
    c. default constructor
    d. any other constructor you might need
    e. any other method you might need
    f. annotations as described in this document
2. UserJdbcDao.java
    a. createUser()
    b. findAllUsers()
    c. findUserById()
    d. deleteUser()
    e. updateUser()
3. UserOrmDao.java
    a. createUser()
    b. findAllUsers()
    c. findUserById()
    d. deleteUser()
    e. updateUser()
    f. annotations as described in this document
4. UserRepository.java
    a. findAllUsers()
    b. findUserById()
    c. annotations as described in this document