Lecture Notes

Database & ORMs

JDBC

In this module, you have learnt to integrate the database to the technical blog application using the concept of object-relational mapping. Object-relational mapping is a technique in which object-oriented programming languages, like Java, are used to convert data between ‘incompatible type systems’. Firstly, you have learnt to connect the database to the application using JDBC API.

Introduction to JDBC

- **JDBC API -** JDBC (Java Database Connectivity) is an Application Program Interface (API) provided by Java. JDBC API represents the classes and interfaces that enable communication between the application and the database. JDBC API requires drivers, which are basically Java programs or files, to connect with the database.
- **JDBC driver** - JDBC drivers are JAR files which translate the instructions and information for both the application and the database. JDBC drivers implement the connection to the database and also implement a protocol to transfer the query in the application and the result given by the database between the JDBC API and the database. Each database comes with its own JDBC driver to connect with the JDBC API.

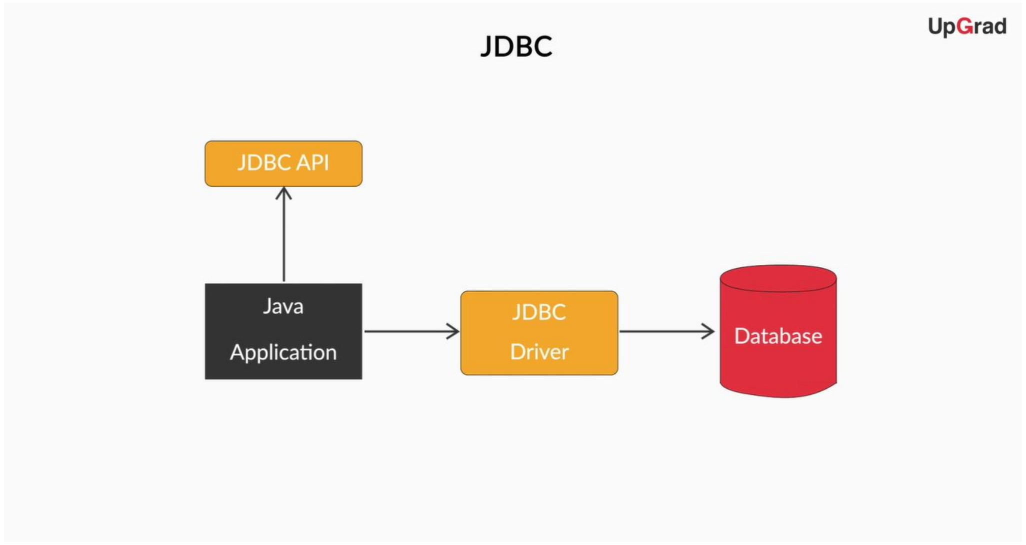


Figure 1 - JDBC

JDBC driver helps to basically manage the following three operations:

1. Connect to the database
2. Send queries to the database
3. Retrieve and process the results received from the database in answer to the client's query.

Establishing a connection with the database using JDBC API

The 'posts' table is created manually in a PostgreSQL database using PgAdmin III. The table contains three columns id, title, and body.

- In order to establish a connection between the technical blog application and the database 'technicalblog' using JDBC API, the following PostgreSQL dependency is added in the pom.xml file:

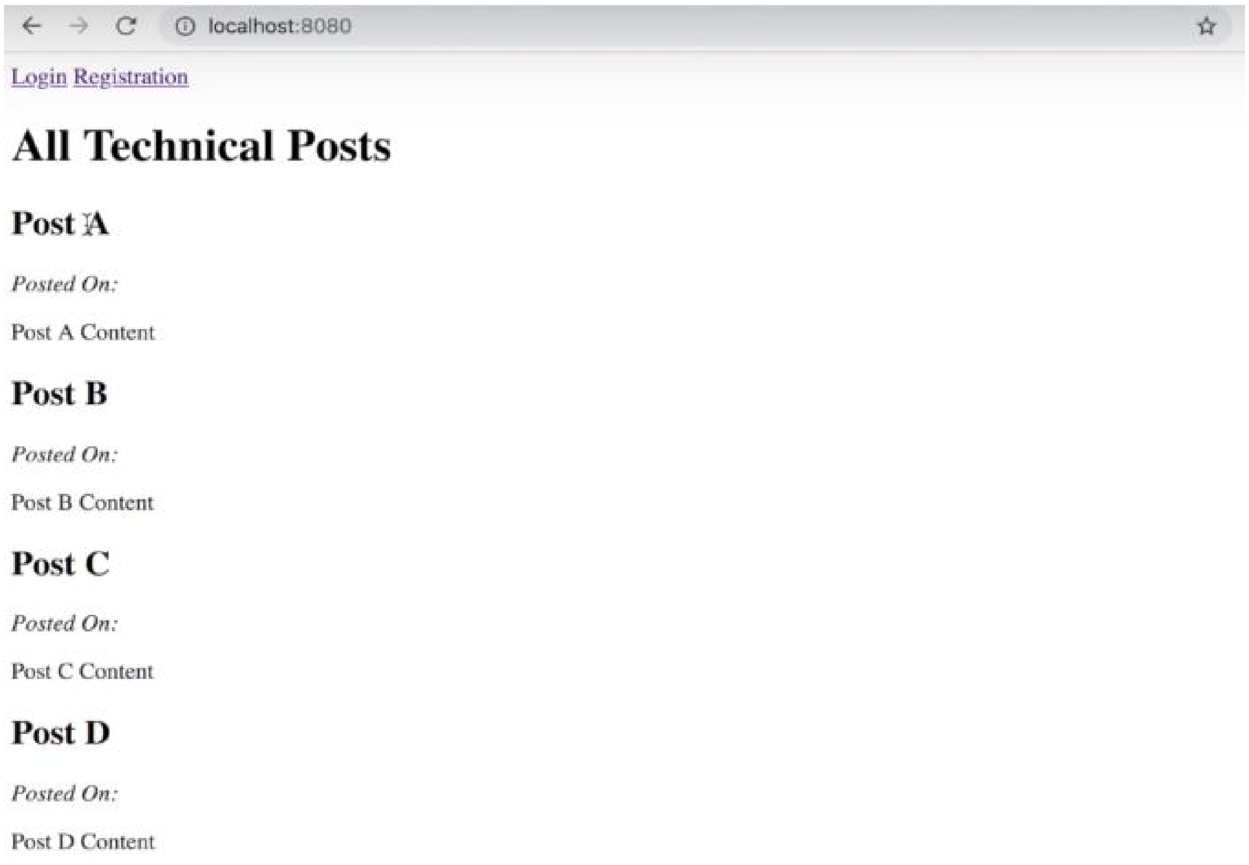
```
<dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<version>42.2.2</version>
<scope>runtime</scope>
</dependency>
```

- The code implemented to connect the database using JDBC driver is as follows:

```
ArrayList<Post> posts = new ArrayList<>();
    try{
            Class.forName("org.postgresql.Driver");

            Connection
connection=DriverManager.getConnection("jdbc:postgresql://localhost:5432/technicalb
log","postgres","1234");
            Statement statement=connection.createStatement();
            ResultSet rs=statement.executeQuery("SELECT * FROM posts");
            while(rs.next()){
            Post post=new Post();
            post.setTitle(rs.getString("title"));
            post.setBody(rs.getString("body"));
            posts.add(post);
            }
            }catch(ClassNotFoundException|SQLException e){
            e.printStackTrace();
            }
            return posts;
```

- Load the driver
- Establish the connection using URL, username, and password.
- Create a Statement object which is used to run the SQL statements against the database.
- Execute the query using executeQuery() method.
- Process ResultSet type object.

- Observe all the posts on the UI as shown below:



All Technical Posts

Post A

Posted On:

Post A Content

Post B

Posted On:

Post B Content

Post C

Posted On:

Post C Content

Post D

Posted On:

Post D Content

Figure 2 - All posts on UI

- Similarly, the code to establish a connection with the database and fetch the post with id equal to 4 from the database is modified as shown below:

```
ArrayList<Post> posts = new ArrayList<>();
try{
   Class.forName("org.postgresql.Driver");

   Connection connection =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/technicalblog","postgres",
"1234");
```

```
   Statement statement = connection.createStatement();
   ResultSet rs = statement.executeQuery("SELECT * FROM posts WHERE id = 4");
   while(rs.next()){
       Post post = new Post();
       post.setTitle(rs.getString("title"));
       post.setBody(rs.getString("body"));
       posts.add(post);
   }
} catch (ClassNotFoundException | SQLException e) {
   e.printStackTrace();
}
return posts;
```

- When you establish the connection with the database using JDBC in the application, then for each connection, certain database resources are allocated and also each connection occupies certain memory space in the JVM (Java virtual machine). Hence, it is important to close the connection once the required task is accomplished. The code to close a connection is as follows:

```
finally{
   try{
       connection.close();
       }catch(SQLException e){
      e.printStackTrace();
       }
       }
```

ORM

The object-oriented Java program and the relational database are incompatible types, and there is a mismatch in how the data is represented as objects in the program and as tables (values) in the relational database. In JDBC, for any operation against the database, you need to manually map the object's model attributes with the columns of the database. To overcome the disadvantages of only using JDBC, you have learnt the concept of ORM - object-relational mapping.

Introduction to ORM

Object-relational mapping (ORM) is a technique that helps to translate the object-oriented data to a relational database. ORM allows you to map the object model with the corresponding tables in the database. This helps the developer to avoid native SQL queries and implement the queries using objects which make the Java program independent of the database. ORMs automatically map your classes to tables, class variables to columns, and instances variables to rows.

Advantages of using ORMs

ORM automatically maps the object model of the application with the corresponding relational data model of the database.

1. The abstraction of mapping between the object model and the database using ORMs helps to migrate from one database to another database.
2. It avoids the requirement of writing native SQL.

JPA

Java Persistence API (JPA) is a specification provided by Java which facilitates the object-relational mapping between the databases and Java applications. It helps to store, access, update and retrieve data from databases to Java objects and vice versa.

Entity and its lifecycle

What is an entity?

An entity is a Java class which is mapped to the corresponding table in the database. In the technical blog application, the Post model class can be considered to be an entity that needs to be added with certain annotations for the JPA implementation to map the class with corresponding table and attributes with the columns.

Entity lifecycle

The object instance of an entity can be in one of the following states, and the changes made to an object will reflect in the database based on the state in which the instance exists.

- **Transient** - The entity is said to be in the transient state when it has just been instantiated and has no representation or has not yet been stored in the database.

- **Persistent** - The entity is said to be in the persistent state when the instance of the entity is in the persistence context and the object instance has a corresponding unique row saved or persisted in the database. This state is either called as managed or persisted and during which any change/update with the object is reflected in the database. When the instance is in the persistent state, the specific row is associated with a unique identifier (primary key) in the database.

- **Detached** - The entity is said to be in the detached state when the instance is detached or no longer exists in the persistence context (Java cache memory), which means the data is already stored in the database with a unique identifier and there is no active object instance representing the specific row. Any change/update done with the object in this state is not reflected in the database.

- **Removed** - The entity is said to be in the removed state when the instance is removed from the persistence context and also the corresponding row is removed/deleted from the database.

Who manages the Entity?
Entity Manager manages the entity. An Entity Manager is a JPA interface which represents the connection with the database. The Entity Manager is responsible for managing the state of an entity in a database, such as creating a row for an entity in the database, deleting a row that represents the entity in the database, updating the row that represents the entity in the database and so forth. The Entity Manager accomplishes the above functionalities through the methods persist(save), merge(update), remove(delete), and createQuery(read) to implement the data operations between the database and Java application.

- **persist()** - This method takes the argument of an object and creates a corresponding row in the database. It changes the state of the object instance from transient to persistent or managed.
- **merge()** - This method is used to update any modified data in the database and changes the state of an object from detached to persistent.
- **remove()** - This method removes/deletes the specific record from the database.

JPA Configuration

Add the required configurations to implement the JPA specification and integrate the database with the technical blog application.

Dependency
The application framework 'Spring Boot' provides a dependency which consists of certain support libraries or jar files to work with the JPA specification. The dependency is as follows:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

persistence.xml
persistence.xml contains the details of the JPA provider such as Hibernate and configuration details of the data source such as database driver, database URL, user, and password.

The code for persistence.xml is as follows:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
            http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
            version="2.1">
```

```
   <persistence-unit name="techblog">

       <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

       <properties>

   <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
           <property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/technicalblog" />
           <property name="javax.persistence.jdbc.user" value="postgres" />
           <property name="javax.persistence.jdbc.password" value="1234" />

           <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQL82Dialect"
/>

       </properties>

   </persistence-unit>

</persistence>
```

- The persistence unit name is the unique name given as a reference identifier for a specific data source.
- The provider element represents the JPA reference implementation, which is Hibernate.
- You can specify the database configuration such as JDBC driver class, url address, user and password information of the database in the persistence.xml file using the javax.persistence package.
- The dialect property helps to generate the database-specific SQL queries. In order to make the application database independent, this property helps to generate the database specific native SQL queries based on the queries written using objects.

JpaConfig

Implement the JPA configuration class to instantiate EntityManagerFactory. The code for JpaConfig class is as shown below:

```
package technicalblog.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;

@Configuration
public class JpaConfig {

   @Bean
   public EntityManagerFactory entityManagerFactory() {
       LocalContainerEntityManagerFactoryBean emfb  = new
LocalContainerEntityManagerFactoryBean();
       emfb.setPersistenceXmlLocation("classpath:META-INF/persistence.xml");
       emfb.afterPropertiesSet();

       return emfb.getObject();
   }

   @Bean
   public DataSource dataSource() {
       DriverManagerDataSource ds = new DriverManagerDataSource();
       ds.setDriverClassName("org.postgresql.Driver");
       ds.setUrl("jdbc:postgresql://localhost:5432/technicalblog");
       ds.setUsername("postgres");
       ds.setPassword("1234");
       return ds;
   }
```

Annotations used

1. **@Configuration** - This annotation helps to refer the classes with any configuration and helps to generate the required instances as spring beans in the spring container.
2. **@Bean** - This annotation refers to the method in the configuration classes which returns a bean or object instance to be managed by the Spring IOC container.

entityManagerFactory() method

- The class 'LocalContainerEntityManagerFactoryBean' provided by spring will generate the EntityManagerFactory object instance by taking the input information of the persistence.xml file. The method 'setPersistenceXmlLocation' takes the location of the persistence.xml as an argument.
- Call the method afterpropertiesSet(), this allows to validate the overall configuration and initialize the instance of EntityManagerFactory with all the properties set.
- Finally, the getObject() method will return the instance of EntityManagerFactory.

dataSource() method

- In order to establish the connection with the database, you need to provide the database configuration details of drivername, URL, user, password in the dataSource() method in the configuration class.
- DriverManagerDataSource class acts as the primary mediator between the application and the drivers of the application you want to connect with.
- Since we want the application to connect with 'org.postgresql.Driver', we need DataManagerDataSource class to load this driver and establish a connection with the database.

JPQL

Introduction

JPQL is a query language that allows writing SQL queries using the objects or entity models, which means the queries are defined based on the name of the entity models rather than the table names of the database. Hibernate helps to convert the JPQL query into the SQL query specific to the database.

JPQL to fetch all the posts from database

The JPQL query to retrieve all the posts from the database is ‘SELECT p from Post p’ which is similar to the SQL query ‘SELECT * FROM posts’.

Retrieve Post

Post class

For creating a post, you need to create a Post model class as shown below:

```
@Entity
@Table(name = "posts")
public class Post {

   @Id
   @Column(name = "id")
   private Integer id;

   @Column(name = "title")
   private String title;

   @Column(name = "body")
   private String body;

@Column(name = "date")
   private Date date;


   public Integer getId() {
       return id;
   }

   public void setId(Integer id) {
       this.id = id;
   }

   public String getTitle() {
       return title;
   }

   public void setTitle(String title) {
       this.title = title;
   }

   public String getBody() {
       return body;
   }

   public void setBody(String body) {
       this.body = body;
   }

   public Date getDate() {
       return date;
   }

   public void setDate(Date date) {
       this.date = date;
   }
}
```

Annotations used:

1. **@Entity:** This is class level annotation and marks the class as an entity to map with the corresponding table in the database.
2. **@Id:** This annotation marks the specific variable as the primary key of the table.
3. **@Table:** The @Table annotation is used to specify the 'posts' table to persist the data. The name attribute decides the name of the table created in the database. Hibernate will automatically create a table named 'posts' as is specified in the name attribute of the @Table annotation and map this class to the table.
4. **@Column:** The @Column annotation specifies that the attribute will be mapped to a corresponding column in the database and explicitly specifies the details of the column to which the attribute will be mapped in the database. The name attribute in the @Column annotation specifies the name of the column in the database.

The attributes of Post class are id, title to save the title of the post, body to save the content of the post, and date to save the upload time of the post.

Repository layer

A service layer cannot implement the business logic as well as interact with the database. It is only responsible for implementing the business logic and you have learnt about a separate repository layer for interaction with the database. The getAllPosts() method in the service logic calls the getAllPosts() method in the repository. Similarly, the getOnePost() method in the service logic calls the getLatestPost() method in the repository class instead of directly interacting with the database. The code for the repository class fetching the posts from the database is shown below:

```
@Repository
public class PostRepository {

   @PersistenceUnit(unitName = "techblog")
   private EntityManagerFactory emf;

   public List<Post> getAllPosts() {
       EntityManager em = emf.createEntityManager();
       TypedQuery<Post> query = em.createQuery("SELECT p from Post p", Post.class);
       List<Post> resultList = query.getResultList();
       return resultList;
   }

   public Post getLatestPost() {
   EntityManager em = emf.createEntityManager();
   return em.find(Post.class, 3);
   }
}
```

Annotation used:

@Repository - The annotation is a specialization of the @Component annotation. The class annotated with the @Repository annotation provides a mechanism to interact with the database. Therefore, it is mandatory to annotate a class with the @Repository annotation, performing all the retrieval, search, update, and delete operations in the database. In addition, the Spring container auto detects the class annotated with the @Repository annotation and instantiates the class bean when the application is run.

Create Post

The user must be able to upload a post and the UI to upload a post is as shown in the below image:

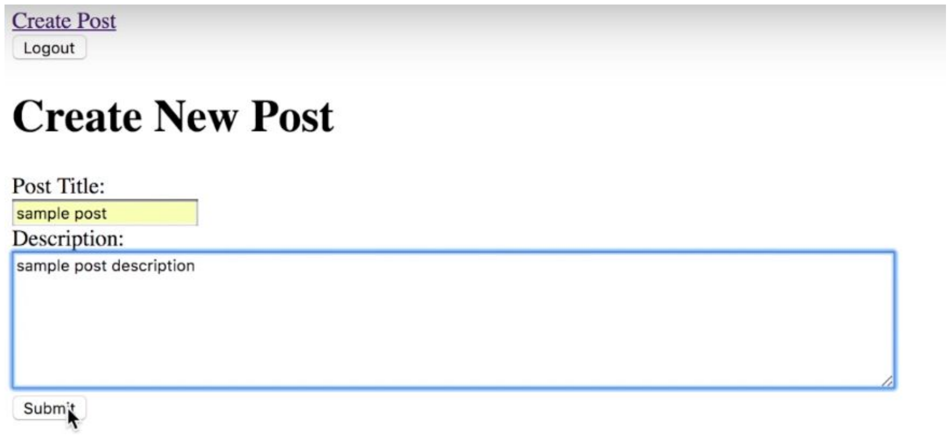


Figure 3 - UI to upload the image

Code Implementation

After a user is logged in the application, he/she gets the option to create a post in the application. When you click on ‘Create Post’ button, Spring scans for controller logic and looks for a controller method with request mapping pattern ‘/posts/newpost’ and handling the request of type GET. The code for the method is as shown below:

```
@RequestMapping("/posts/newpost")
    public String newPost() {
        return "posts/create";
    }
```

The method returns the ‘posts/create.html’ file as shown below:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

<head th:replace="layout :: site-head">
    <title>Spring MVC Application - Create Post</title>
</head>

<header th:replace="layout :: logged-in"></header>

<body>
    <h1>Create New Post</h1>

    <form method="post" th:action="@{/posts/create}">
        <div>Post Title:</div>
        <div><input type="text" id="title" name="title" /></div>

        <div>Description:</div>

        <div><textarea rows="7" cols="100" name="body"></textarea></div>
        <div>
            <input type="submit" value="Submit"/>
        </div>
    </form>

</body>
</html>
```

Enter the title and body of the post and click ‘Submit’. After you click ‘Submit’, Post object is declared and Spring maps the Post object attributes with the values entered in divisions of the same name as the attribute name. The value entered in the division named ‘title’ is mapped to the title attribute of Post object, and the value entered in the division named ‘body’ is mapped to the body attribute of Post object. Spring again scans the controller logic and looks for a controller method with request mapping pattern ‘/posts/create’ and handling the request of type POST. The Post object is also passed to the method. The code for the method is as shown below:

```
@RequestMapping(value = "/posts/create", method = RequestMethod.POST)
    public String createPost(Post newPost {

        postService.createPost(newPost);
        return "redirect:/posts";
    }
```

The method calls the createPost() method in the service layer, which is responsible for persisting the post in the database. After the post is persisted in the database, the method calls the controller method with request mapping pattern ‘/posts’, i.e. getUserPosts() method which again displays all the posts in the database. The code for the service layer is as shown below:

```
public void createPost(Post newPost) {
   repository.createPost(newPost);
   System.out.println("New Post: "+newPost);
}
```

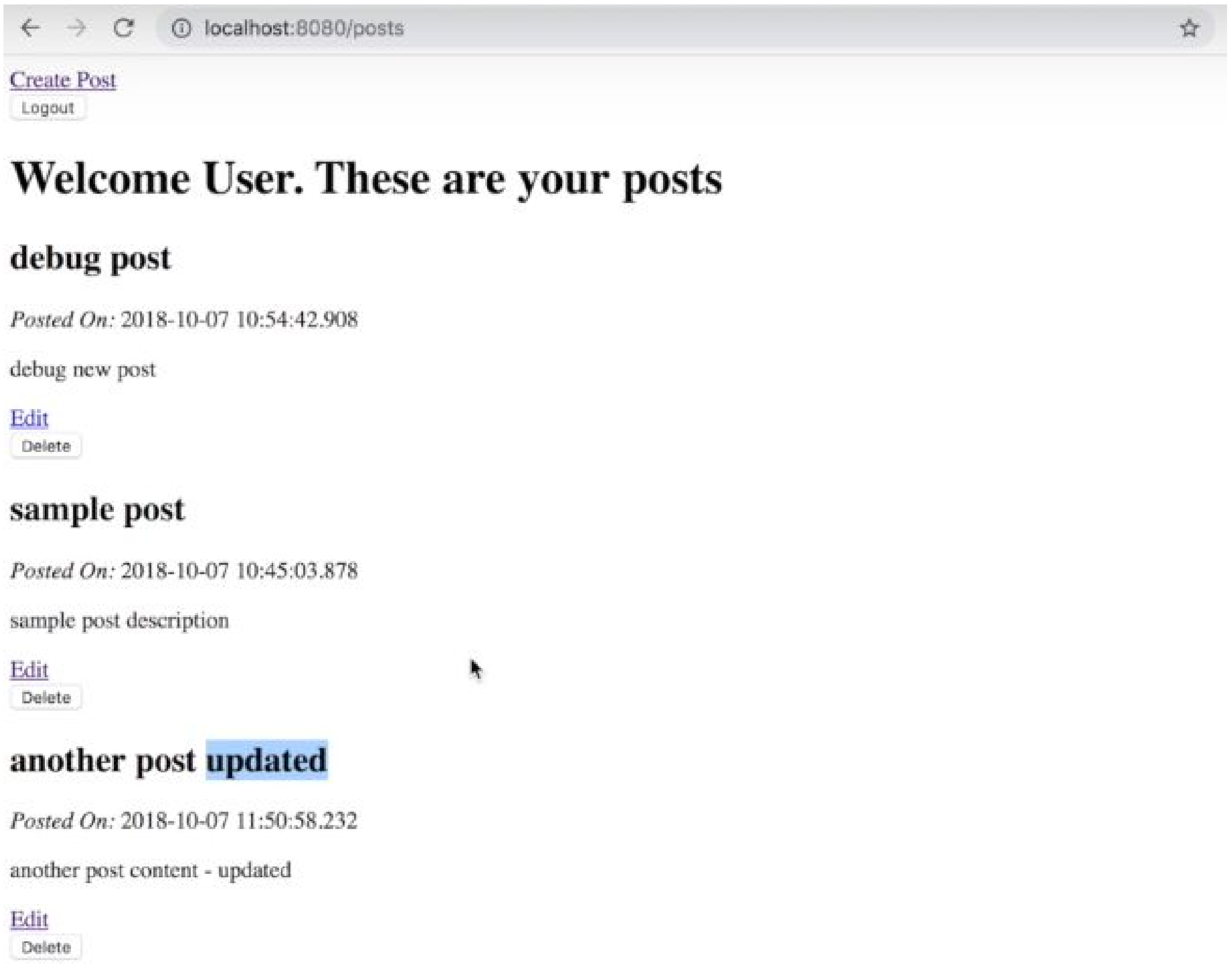
The method calls the createPost() method in the repository layer. The code for the repository layer is as shown below:

```
public Post createPost(Post newPost) {

   EntityManager em = emf.createEntityManager();
   EntityTransaction transaction = em.getTransaction();

   try {
       transaction.begin();
       em.persist(newPost);
       transaction.commit();
   }catch(Exception e) {
       transaction.rollback();
   }

   return newPost;
}
```

The createPost() method simply receives the Post type object to be persisted in the database. It then changes the state of the Post type object from the transient state to the managed/persistent state using the persist() method, so the object has its corresponding representation in the database. The transaction is committed if the operation is successful; else, it is rolled back. By rollback we mean to say that if there are multiple operations in a transaction and any one of the operations fails, then none of the operations will be executed or the transaction is rolled back.

Edit Post

Once the user logs in to the application, the home page should have an option to edit the blog post as shown in the below image:



localhost:8080/posts

Create Post

Logout

Welcome User. These are your posts

debug post

Posted On: 2018-10-07 10:54:42.908

debug new post

Edit

Delete

sample post

Posted On: 2018-10-07 10:45:03.878

sample post description

Edit

Delete

another post updated

Posted On: 2018-10-07 11:50:58.232

another post content - updated

Edit

Delete

Figure 4 - Edit option on UI

The posts.html file displays all the posts in the database on the user home page. This file must consist of the instructions to provide an option to edit the image. When the user clicks on the 'Edit' option, the server receives a GET request for the URL "/editPost/{postId}. The controller logic must have the corresponding request handling method to handle the GET request to retrieve the details of the specific post from the database.

Thymeleaf template
The code for 'posts.html' along with the 'Edit' option is shown below:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

   <head th:replace="layout :: site-head">
       <title>Spring MVC Application - User Posts</title>
   </head>

   <header th:replace="layout :: logged-in"></header>

   <body>
       <h1>Welcome User. These are your posts</h1>
       <main id="posts">
           <post th:each="p : ${posts}">
               <h2 th:text="${p.title}"></h2>
               <i>Posted On: </i> <span th:text="${p.date}"></span>
               <div>
                   <p th:text="${p.body}"></p>
               </div>
               <div>
                   <a th:href="@{/editPost(postId=${p.id})}">Edit</a>
               </div>
           </post>
       </main>
   </body>
</html>
```

Note that the code above iterates over all the blog posts stored in the posts variable in the Model object and assigns each post to the variable ‘p’. When you click on the Edit option, Spring scans the controller logic and looks for a controller method with request mapping of type ‘/editPost’ and GET request method. Also, note that the piece of code adds the id of the post as the request parameter (postId) in the request, and the request parameter, postId, can be accessed in the controller method using the @RequestParam annotation.

Controller code

The code for the controller method with request mapping of type ‘/editPost’ and GET request method, and handling the user request to edit the form is as follows:

```
@RequestMapping(value = "/editPost", method = RequestMethod.GET)
public String editPost(@RequestParam(name="postId") Integer postId, Model model) {
   Post post = postService.getPost(postId);
   model.addAttribute("post",post);
   return "posts/edit";
}
```

This method accepts the GET request to edit the post, retrieves the request parameter postId using the @RequestParam annotation, and assigns it to an integer variable postId. It then fetches the post from the database with its corresponding post id by calling the getPost() method in the business logic.

Service code
The code for the getPost() method in the service layer is as follows:

```
public Post getPost(Integer postId) {
   return repository.getPost(postId);
}
```

The method above in the service layer simply calls the getPost() method in the repository class and returns the Post object returned by repository class.

Repository code
The code for the getPost() method in the repository class is as follows:

```
public Post getPost(Integer postId) {
   EntityManager em = emf.createEntityManager();
   return em.find(Post.class, postId);
}
```

The getPost() method in the repository class retrieves the details of the post from the database and returns the Post object to the service class method. Then, the service class method returns the same object to the controller class; the controller logic gets the post object and adds the same to the Model object and returns to the 'posts/edit.html' thymeleaf template file.

Thymeleaf template
The code for the 'posts/edit.html' file returned by the controller logic is shown below:

```
<html xmlns:th="http://thymeleaf.org">

<head th:replace="layout :: site-head">
   <title>Spring MVC Technical Blog</title>
</head>

<header th:replace="layout :: logged-out"></header>

<h1>Create New Post</h1>

<form th:method="put" th:action="@{/editPost(postId=${post.id})}">
   <div>Post Title :</div>
   <div>
       <input type="text" name="title" size="70" th:value="${post.title}"/>
   </div>

   <div> Description :</div>
   <div>
       <textarea rows="7" cols="100" name="body" th:text="${post.body}"></textarea>
   </div>
```

```
    <input type="submit" value="Submit"/>
    </div>
</form>

</html>
```

Note that the post data is passed using the Model object, and all the attributes in the Post object can be accessed using the dot operator. The code displays the current title of the post and provides an option to edit the title in an input field. The code also displays the current description of the post and provide a text box to edit the description in the text bar. After the edit form is submitted, Spring scans the controller logic and looks for the request handling method with url ‘/editPost’ and handling the HTTP request PUT.

Controller code
The code for the controller method with request mapping of type ‘/editPost’ and request method type PUT is as follows:

```
@RequestMapping(value = "/editPost", method = RequestMethod.PUT)
public String editPostSubmit(@RequestParam(name="postId") Integer postId, Post updatedPost) {
    updatedPost.setId(postId);
    postService.updatePost(updatedPost);
    return "redirect:/posts";
}
```

The method receives the postId request parameter using the @RequestParam annotation and assigns it to an integer variable postId. When you submit the form after editing the post details, an object of the Post type is instantiated, the title and the description attributes are mapped to the title and the description edited by the user. The method then sets the post id and calls the business logic to update the post in the database.

Service code
The code for updatePost() method in service layer is as follows:

```
public void updatePost(Post updatedPost) {
    updatedPost.setDate(new Date());
    repository.updatePost(updatedPost);
}
```

The above method sets the date and time at which the post is edited and calls the repository class to update the post details in the database.

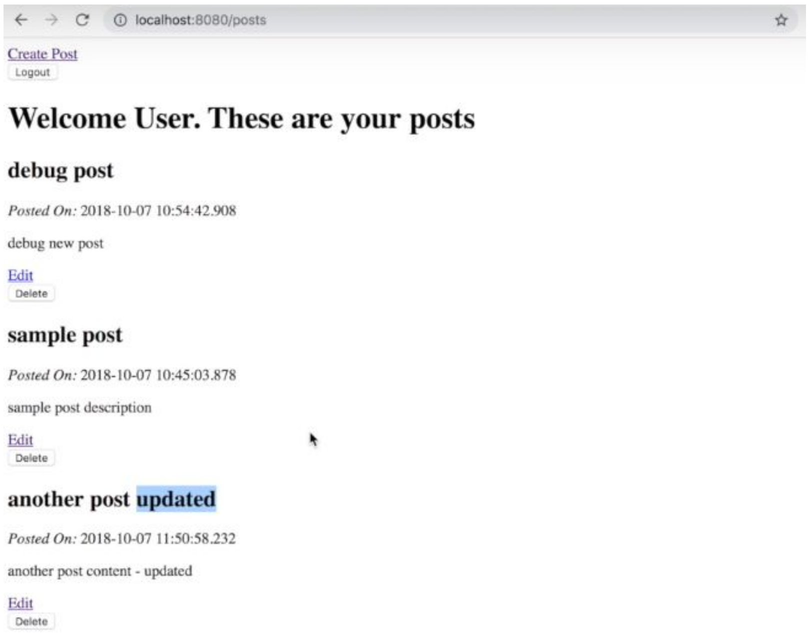
Repository code
The code for updatePost() method in repository class is as shown below:

```
public void updatePost(Post updatedPost) {
   EntityManager em = emf.createEntityManager();
   EntityTransaction transaction = em.getTransaction();

   try {
       transaction.begin();
       em.merge(updatedPost);
       transaction.commit();
   }catch(Exception e) {
       transaction.rollback();
   }
}
```

The above method helps to receive the Post object to be updated in the database. The method declares an EntityManager object using the instance of EntityManagerFactory and starts the transaction. Change the state of the post object from detached to persistent using the merge() method. Once the entity state is changed to persistent, any changes made to the post object are directly reflected in the database. Hence, the post is updated in the database.

Delete Post

Now, we are left with the delete operation on blog post in technical blog application. Once the user logs in to the application, the home page should offer the option to delete the blog post as you may observe in the below image:



Welcome User. These are your posts

debug post

Posted On: 2018-10-07 10:54:42.908

debug new post

sample post

Posted On: 2018-10-07 10:45:03.878

sample post description

another post updated

Posted On: 2018-10-07 11:50:58.232

another post content - updated

Figure 5 - Delete option on UI

The posts.html file displays all the posts in the database on the user home page. This file must consist of the instructions to provide an option to delete the blog post. When the user clicks on the ‘Delete’ option, the server receives a DELETE request for a particular post. In order to handle the DELETE request, the controller logic must have a request handling method which would call the service layer to implement the business logic and repository layer to interact with the database.

Thymeleaf template
The code for ‘posts.html’ is as shown below:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

   <head th:replace="layout :: site-head">
       <title>Spring MVC Application - User Posts</title>
   </head>

   <header th:replace="layout :: logged-in"></header>

   <body>
       <h1>Welcome User. These are your posts</h1>
       <main id="posts">
           <post th:each="p : ${posts}">
               <h2 th:text="${p.title}"></h2>
               <i>Posted On: </i> <span th:text="${p.date}"></span>
               <div>
                   <p th:text="${p.body}"></p>
               </div>
               <div>
                   <a th:href="@{/editPost(postId=${p.id})}">Edit</a>
                   <form th:action="@{/deletePost(postId=${p.id})}" th:method="delete">
                       <input type="submit" value="Delete" />
                   </form>
               </div>
           </post>
       </main>
   </body>
</html>
```

The code adds a ‘Delete’ button to delete a post when the post is displayed. And when you click on the ‘Delete’ button, Spring scans the controller logic and looks for a controller method with request mapping url ‘/deletePost’ and DELETE request method. Also, note that this piece of code adds the id of the post as the request parameter (postId) in the request. This id is used to identify the post that we want to delete, and this request parameter, postId, can be accessed in the controller method using the @RequestParam annotation and stored in a variable.

Controller code

The code for the controller method with request mapping of type '/deletePost' and accepting the request of type DELETE is as follows:

```
@RequestMapping(value = "/deletePost", method = RequestMethod.DELETE)
public String deletePostSubmit(@RequestParam(name="postId") Integer postId) {
   postService.deletePost(postId);
   return "redirect:/posts";
}
```

This method accepts the DELETE request to delete the post, retrieves the request parameter postId using @RequestParam annotation, and assigns it to an integer variable postId. The method then calls the deletePost() method in the PostService class by passing the id of the post that is to be deleted. After the business logic deletes the post from the database, the method redirects to the user to the homepage, displaying all the posts by calling the getUserPosts() method. The code for deletePost() method in the service layer is as follows:

```
public void deletePost(Integer postId) {
   repository.deletePost(postId);
}
```

The method accepts the id of the post which is to be deleted and calls the repository class by passing the id of the post to be deleted. The code for the deletePost() method in the repository class is as follows:

```
public void deletePost(Integer postId) {
   EntityManager em = emf.createEntityManager();
   EntityTransaction transaction = em.getTransaction();

   try {
       transaction.begin();
       Post post = em.find(Post.class, postId);
       em.remove(post);
       transaction.commit();
   }catch(Exception e) {
       transaction.rollback();
   }
}
```

The method described above receives the id of the post to be deleted. It then declares an EntityManager object using an instance of EntityManagerFactory. The method then starts the transaction using an instance of EntityManager. We first fetch the post from the database with the corresponding id using the find() method. This changes the state of the entity object to persistent. Then use the remove() method on the Post object, which is in the persistent state, and the corresponding post is deleted from the database. If the operation is successful, the transaction is committed. Else, the transaction is rolled back.

User Registration

Let’s now move a step ahead in the technical blog application and store the details of the user in the database. In this context, you have come across the relationships that can be established between different tables in the database to implement the functionality of user registration.

One To One mapping

In order to store user details in the database, the database schema consists of two tables: the ‘users’ table to store username and password and the user_profile table to store full name, email address, and mobile number of the user as shown below:

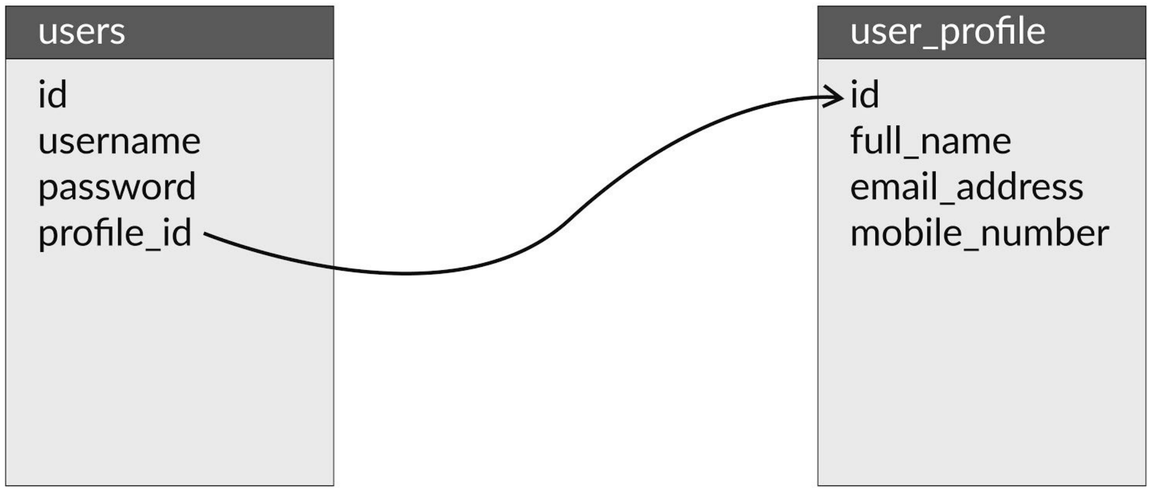


Figure 6 - One To One mapping

Since a user can have only one user profile and one user profile can only belong to a single user, there exists One To One relationship between the ‘users’ table and the ‘user_profile’ table. The ‘profile_id’ column in the ‘users’ table in the database is the foreign key and references the ‘id’ column in the ‘user_profile’ table for its validation.

Entity classes

The first step is to define the two entity classes for each table. These classes will be mapped to the ‘users’ and ‘user_profile’ table in the database. The attributes in these classes will be mapped to the corresponding columns in the ‘users’ and ‘user_profile’ tables in the database.

The code for the ‘User’ class is as shown below:

```java
@Entity
@Table(name = "users")
public class User {

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

   @Column(name = "username")
   private String username;

   @Column(name = "password")
   private String password;

   @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
   @JoinColumn(name = "profile_id")
   private technicalblog.model.UserProfile profile;

   @OneToMany(mappedBy = "user", cascade = CascadeType.REMOVE, fetch = FetchType.LAZY)
   private List<Post> posts = new ArrayList<>();

   public Integer getId() {
       return id;
   }

   public void setId(Integer id) {
       this.id = id;
   }

   public String getUsername() {
       return username;
   }

   public void setUsername(String username) {
       this.username = username;
   }

   public String getPassword() {
       return password;
   }

   public void setPassword(String password) {
       this.password = password;
   }

   public UserProfile getProfile() {
       return profile;
   }

   public void setProfile(UserProfile profile) {
       this.profile = profile;
   }
   public List<Post> getPosts() {
       return posts;
   }

   public void setPosts(List<Post> posts) {
       this.posts = posts;
   }

}
```

The code for the ‘UserProfile’ class is as shown below:

```
@Entity
@Table(name = "user_profile")
public class UserProfile {

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

   @Column(name = "full_name")
   private String fullName;

   @Column(name = "email_address")
   private String emailAddress;

   @Column(name = "mobile_number")
   private String mobileNumber;

   public String getFullName() {
       return fullName;
   }

   public void setFullName(String fullName) {
       this.fullName = fullName;
   }

   public String getEmailAddress() {
       return emailAddress;
   }

   public void setEmailAddress(String emailAddress) {
       this.emailAddress = emailAddress;
   }

   public String getMobileNumber() {
       return mobileNumber;
   }

   public void setMobileNumber(String mobileNumber) {
       this.mobileNumber = mobileNumber;
   }
}
```

Annotations used

1. **@Entity:** This is class level annotation and marks the class as an entity to map with the corresponding table in the database.
2. **@Id:** This annotation marks the specific variable as the primary key of the table.
3. **@Table**: The @Table annotation is used to specify the 'posts' table to persist the data. The name attribute decides the name of the table created in the database. Hibernate will automatically create a table named 'users' as is specified in the name attribute of the @Table annotation and map this class to the table.
4. **@Column**: The @Column annotation specifies that the attribute will be mapped to a corresponding column in the database and explicitly specifies the details of the column to which the attribute will be mapped in the database. The name attribute in the @Column annotation specifies the name of the column in the database.

Relationship between User class and UserProfile class

The @OneToOne annotation is used to define a One to One relationship between User and UserProfile entities. If a User object propagates through PERSIST, REMOVE, REFRESH, MERGE, or DETACH operations, the UserProfile object being referenced by that particular User object must also propagate through the same operations. cascade = CascadeType.ALL is used to implement this functionality and Fetch type is EAGER. The @JoinColumn annotation is used to specify that the 'profile_id' column created in the user table references the 'id' column in the 'user_profile' table.

EAGER vs LAZY fetching

LAZY: This is the default FetchType. So, if you don't specify any FetchType, it will be automatically be specified for you. This means that when you load the User object, Hibernate will not load this user profile information unless you explicitly ask it to do so using the getProfile() method. This fetch type may throw NullPoinerException when you directly try to access user profile information without using the getProfile() method. Then why do we use it? Suppose the User class has 10 different relationships to other objects and you just want to access the username and password of the user. In that case, loading all the 10 different relationships into Java objects and keeping them in memory would be very expensive. So, in that case, the LAZY fetch type is preferred.

EAGER: This is the opposite of LAZY. This means that if this FetchType is used instead of LAZY, Hibernate will load the user profile attributes into the user object by default. Using this fetch type avoids NullPointerException, but loading all the different related objects into memory is very expensive.

Controller code

When you click on ‘Registration’, Spring scans for controller class and looks for a controller method with request mapping of pattern ‘/users/registration’ and accepting the request of type GET. The request is handled by the registration() method in the UserController class with request mapping url ‘/users/registration’. The code is as shown below:

```
@RequestMapping("users/registration")
public String registration(Model model) {
   User user = new User();
   UserProfile profile = new UserProfile();
   user.setProfile(profile);

   model.addAttribute("User", user);

   return "users/registration";
}
```

The method creates an instance of the User class and an instance of the UserProfile class and sets the profile attribute in the User object to the newly created UserProfile instance. The User object is then added in the Model object with ‘User’ as the key. The method then returns ‘users/registration.html’ file. When a user fills the form and submits details, all the details are mapped to the corresponding attributes in the User and UserProfile objects that we instantiated above. After this, Spring scans for controller class and looks for a controller method with request mapping of pattern ‘/users/registration’ and accepting the request of type POST. The request is handled by the registerUser() method in the UserController class with request mapping of pattern ‘/users/registration’ and accepting the request of type POST. The code is as shown below:

```
@RequestMapping(value = "users/registration", method = RequestMethod.POST)
public String registerUser(User user) {
   userService.registerUser(user);
   return "users/login";
}
```

The method receives the User object with all the attributes mapped to the details provided by the user in the UI. Now, this User object is to be persisted in the database.

Service code

The code for the service layer to register the user in the application is as shown below:

```
public void registerUser(User newUser) {
   repository.registerUser(newUser);
}
```

The method simply passes the User object to be persisted in the database to the repository class.

Repository code

The code for the repository class to register the user in the application is as shown below:

```
public void registerUser(User newUser) {
   EntityManager em = emf.createEntityManager();
   EntityTransaction transaction = em.getTransaction();

   try {
       transaction.begin();
       em.persist(newUser);
       transaction.commit();
   }catch(Exception e) {
       transaction.rollback();
   }
}
```

The method detailed above receives the User object to be persisted in the database. The method declares an EntityManager object using an instance of EntityManagerFactory. The method then starts the transaction using an instance of EntityManager. persist() method is used to change the state of the object from transient to persistent. Once the user object is persisted in the database, it is mapped to the corresponding row in the database.

User login

You have learnt to modify the login functionality such that only a registered user can login to the application.

Controller code

When the application is made to run on a localhost, the page displays all the posts in the application. It is actually the output of the 'index.html' file, which contains a 'logged-out' fragment that has an option to log in to the application. When you click on 'Login', the Spring scans for the controller class and looks for a controller method with the request mapping of pattern '/users/login' and accepting the request of type GET. The request is handled by the login() method in the UserController class with the request mapping the pattern '/users/login' and accepting the request of type GET. The code is as shown below:

```
@RequestMapping("users/login")
public String login() {
   return "users/login";
}
```

This method simply returns the 'users/login.html' thymeleaf template file, where you need to enter the username and password.

Thymeleaf template

The code for ‘users/login.html’ is given below:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

<head th:replace="layout :: site-head">
   <title>Spring MVC Application - Login</title>
</head>

<header th:replace="layout :: logged-out"></header>

<h2>Please Login:</h2>

<form th:action="@{/users/login}" th:method="POST" th:object="${User}">
   <label for="username">Username:</label>
   <input type="text" id="username" name="username"/>
   <label for="password">Password:</label>
   <input type="password" id="password" name="password"/>
   <input type="submit" value="Login"/>
</form>

</html>
```

After you enter the username and password and click on the ‘Login’ button, Spring scans for the controller class and looks for a controller method with the request mapping of pattern ‘/users/login’ and accepting the request of type POST.

Controller code

The request is handled by the loginUser() method in the UserController class with the request mapping of pattern ‘/users/login’ and accepting the request of type POST. The code is given below:

```
@RequestMapping(value = "users/login", method = RequestMethod.POST)
public String loginUser(User user) {
   User existingUser = userService.login(user);
   if (existingUser != null) {
       return "redirect:/posts";
   } else {
       return "users/login";
   }
}
```

The method receives the User object with the username and password attributes mapped to the username and password entered in the UI. The method calls the login() method in the UserService class. If the username and password exist in the database, then, the business logic will return a User object representing the logged in user. If the username and password do not exist in the database, the business logic returns a null value. This tells us that the username and password are incorrect, and we should direct the user to the same login page for entering the valid credentials again.

Service code
The code for the service layer in the UserService class is given below:

```
public User login(User user) {

   User existingUser = repository.checkUser(user.getUsername(), user.getPassword());
   if(existingUser != null) {
       return existingUser;
   }
   else {
       return null;
   }
}
```

The method receives the User object containing the username and password. The username and password are sent to the UserRepository class for validation by calling the checkUser() method. If the username and password exist in the database, the service layer returns the User object corresponding to that username and password returned by the repository class. Else, the service layer returns null.

Repository code
The code for the repository class is given below:

```
public User checkUser(String username, String password) {

   try {
       EntityManager em = emf.createEntityManager();
       TypedQuery<User> typedQuery = em.createQuery("SELECT u FROM User u WHERE u.username =
:username AND u.password = :password", User.class);
       typedQuery.setParameter("username", username);
       typedQuery.setParameter("password", password);

       return typedQuery.getSingleResult();
   }catch(NoResultException nre) {
       return null;
   }
}
```

The above method receives the username and password and declares an EntityManager object using an instance of the EntityManagerFactory. Now, we will use the createQuery() method of the EntityManager class to create a dynamic query. The JPQL query fetches the record from the 'users' table where the username is equal to the username parameter specified with the colon (the parameter will be set to the username entered on the UI) and the password is equal to the password parameter specified with the colon (the parameter will be set to the password entered on the UI). u is an alias made for the User class. getSingleResult() method

executes the query that returns a single result. This result is a User object fetched from the database with the corresponding username and password and is returned to the service layer.

User Logout

Http Session

Web Servers have a short-term memory. As soon as they send any response to the clients, they lose the clients details. In other words, web servers do not remember what the clients had requested in the past. Sometimes, that is fine, but in many scenarios, you need to maintain the conversational state between the client and the server. One common example could be of a shopping cart. Most users select multiple things to buy before proceeding to checkout. In such scenarios, the web server should be smart enough to remember the products that the user selected. Surprisingly, Spring provides a very simple solution to handle such scenarios: the HttpSession. An HttpSession object can hold a conversational state across multiple requests from the same client (the client is a web browser in this example). In other words, the Http session persists for an entire session between the server and a specific client, i.e., web browser. We can use an Http session to store all the information about a client in all the requests made by the client during a session. Such that you keep the track of the information and the state about the client.
The user details are added in the Http session when the user sends the POST type request to the server to login to the application. The modified code for the controller method is given below:

```
@RequestMapping(value = "users/login", method = RequestMethod.POST)
public String loginUser(User user, HttpSession session) {
   User existingUser = userService.login(user);
   if (existingUser != null) {
       session.setAttribute("loggeduser", existingUser);
       return "redirect:/posts";
   } else {
       return "users/login";
   }
}
```

The method receives an HttpSession object provided by Spring. If the user has provided valid login credentials, add the User object in the HttpSession object with ‘loggeduser’ as the key and the user is successfully logged in to the application. Now, the application can access the author of the user by accessing the ‘loggeduser’ value from the HttpSession object. The application will display the full name of the user after he/she is logged in by accessing the user from the Http session.

Logging out

After the user is logged in to the application, a unique session id is assigned to him/her. Now the login functionality would simply require to call the invalidate() method, and the current session is invalidated. First, you need to add a logout button in the ‘logged-in’ fragment in the ‘layout.html’ file such that user has the option to log out of the application when he/she is logged in. When you click on the ‘Logout’ button, Spring scans for the controller class and looks for a controller method with request mapping of the pattern ‘/users/logout’ and accepting the request of the type POST. The request is handled by the logout() method in the UserController class with the request mapping of the pattern ‘/users/logout’ and accepting the request of the type POST. The code is given below:

```
@RequestMapping(value = "users/logout", method = RequestMethod.POST)
public String logout(Model model, HttpSession session) {
   session.invalidate();

   List<Post> posts = postService.getAllPosts();

   model.addAttribute("posts", posts);
   return "index";
}
```

The session.invalidate() dynamically invalidates or destroys the current session. Now, after the user is logged out, the user will be directed to the index web page displaying all the posts in the application, which is the output returned by the ‘index.html’ file. Therefore, we will return the ‘index.html’ file.

Map post to a user

Then you have learnt about the one to many relationship between a user to blog posts,

Many To One / One To Many mapping
A user can upload multiple posts in the application. But, there is only one user corresponding to a particular post. Therefore, there exists One to Many mapping between the users table and the posts table in the database. In One to Many mapping or Many to One mapping between two tables, a new column is added on the ‘Many’ side of the relationship, which acts as a foreign key and references the primary key of the other table which is on the ‘One’ side. Let’s add a column ‘user_id’ in the ‘user_profile’ table, which acts as a primary key and references the id (primary key) column of the ‘users’ table for its validation.

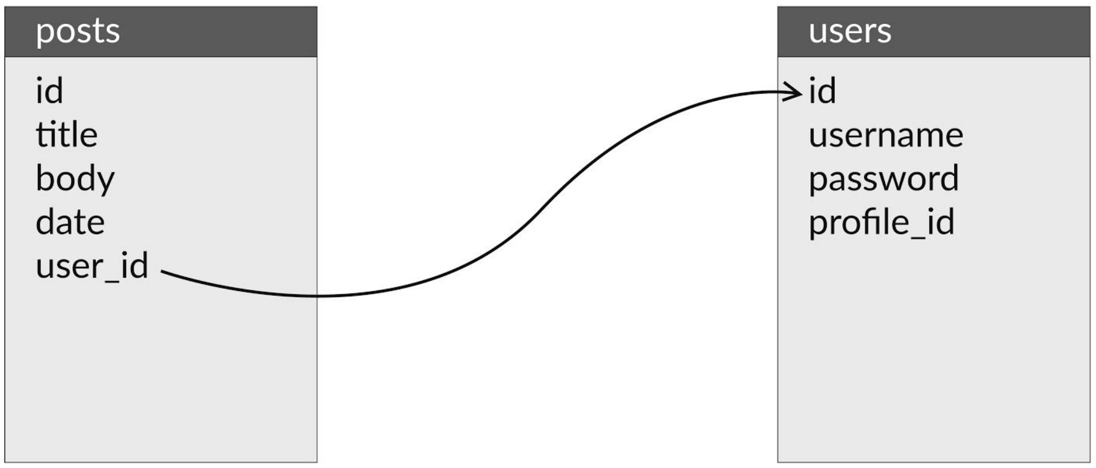


Figure 7 - Many To One / One To Many mapping

Entity classes

The code to add an attribute of List<Post> type in the User model class to specify the list of all the posts that a particular user has uploaded is as shown below:

```
@OneToMany(mappedBy = "user", cascade = CascadeType.REMOVE, fetch = FetchType.LAZY)
private List<Post> posts = new ArrayList<>();
```

@OneToMany annotation is used to specify One to Many mapping between the User and Post entities. The mappedBy attribute specifies the attribute in the Post class that is responsible for One to Many relationships between the User and Post model classes. If a User object propagates through the REMOVE operation, the Post object referencing that particular User object must also propagate through the same operation. Fetch type is LAZY, as we do not want to load all the posts related to that user immediately when other user details are loaded. An attribute of the User type in the Post model class which maps to the 'user_id' column in the 'posts' table in the database is as shown below:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "user_id")
private User user;
```

@ManyToOne annotation is used to specify Many to One mapping between the Post and User entities. Fetch type is EAGER since we want to load all the user details corresponding to a particular post when the other details of post are loaded. The @JoinColumn annotation is used to specify that the 'user_id' column created in the 'posts' table in the database references the 'id' column in the 'users' table.

Controller code
The user details (User object) of the current logged in user are fetched from the current session using getAttribute() method. Once the User object is fetched, you need to map it to the 'user' attribute of the Post object. In this way, the user attribute of a post is set to the current logged in user when a post is uploaded. The code for createPost() method is as shown below:

```
@RequestMapping(value = "/posts/create", method = RequestMethod.POST)
public String createPost(Post newPost, HttpSession session) {
   User user = (User)session.getAttribute("loggeduser");
   newPost.setUser(user);
   postService.createPost(newPost);
   return "redirect:/posts";
}
```

The same step should be followed when the post is edited.

Map category to a post

Many To Many mapping
Each blog post can be classified into different categories based on the content. A post can have multiple categories and there can be multiple posts under one category. Therefore, there exists Many to Many mapping between the posts table and the categories table in the database. Note that in Many to Many mapping, a new join table is created which consists of two columns, and both the columns are primary keys of both the tables. This is how a Many to Many relationship can be specified between two tables.

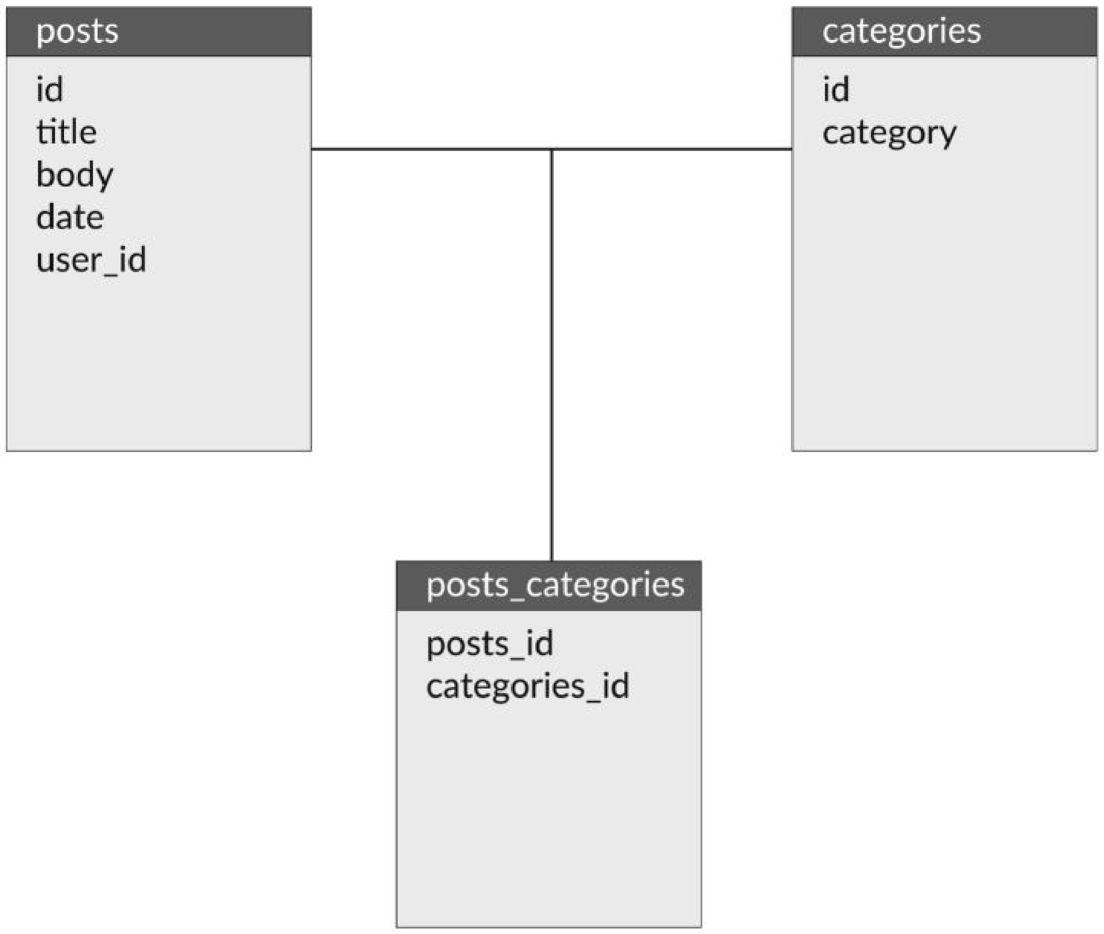


Figure 8 - Many To Many mapping

Also, to simplify the functionality, we will restrict the number of categories to two. You might write a blog on Java or Spring. Therefore, there are only two categories, namely ‘Java Blog’ and ‘Spring Blog’ in the technical blog application.

Entity classes

The code for the Category class is as shown below:

```
@Entity
@Table(name = "categories")
public class Category {

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

```
    @Column(name = "category")
    private String category;

    @ManyToMany(mappedBy = "categories", fetch = FetchType.LAZY)
    private List<Post> posts = new ArrayList<>();

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

}
```

Annotations used

1. **@Entity:** This is class level annotation and marks the class as an entity to map with the corresponding table in the database.
2. **@Id:** This annotation marks the specific variable as the primary key of the table.
3. **@Table**: The @Table annotation is used to specify the 'categories'' table to persist the data. The name attribute decides the name of the table created in the database. Hibernate will automatically create a table named 'users' as is specified in the name attribute of the @Table annotation and map this class to the table.
4. **@Column**: The @Column annotation specifies that the attribute will be mapped to a corresponding column in the database and explicitly specifies the details of the column to which the attribute will be mapped in the database. The name attribute in the @Column annotation specifies the name of the column in the database.

@ManyToMany annotation is used to specify that there exists Many to Many mapping between the Post and Category entities. We want to load all the categories of a post when the other details of the post are loaded. Therefore, the fetch type is EAGER. If a Post object propagates through PERSIST, REMOVE, REFRESH, MERGE, or DETACH operations, the Category object mapped to that Post object must also propagate through these operations. Also, add two more attributes in the Post class as shown below:

```
@Transient
private String springBlog;
@Transient
private String javaBlog;
```

The @Transient annotation is used to specify that these attributes are not mapped to any columns in the database. Hence, class parameters marked by the @Transient annotation will not be saved in the database.

Thymeleaf templete

Let’s now add the checkboxes in the ‘posts/create.html’ file as shown below:

```
<div>
<input type="checkbox" name="SpringBlog" value="Spring Blog"/>Spring Blog
<input type="checkbox" name="JavaBlog" value="Java Blog"/>Java Blog
</div>
```

The name field refers to the name of the checkbox. The value field refers to the value assigned to the mapping variable. When you create a post and submit it, an object of the Post type is instantiated and all the details of the post are mapped to the corresponding attributes of the Post object. Similarly, the values of the springBlog attribute and the javablog attribute are mapped according to the selected checkbox. Spring then scans the controller logic and looks for the controller method which can handle this request. The createPost() method in the PostController class handles the request. This object is then passed on to the controller method. The controller method also receives the HttpSession object to identify the user.

Controller code

As you already know, the controller method gets the user from the current session and maps it to the user attribute. Now, the controller logic is modified to:

```
@RequestMapping(value = "/posts/create", method = RequestMethod.POST)
public String createPost(Post newPost, HttpSession session) {
   User user = (User) session.getAttribute("loggeduser");
   newPost.setUser(user);

   if (newPost.getSpringBlog() != null) {
       Category springBlogCategory = new Category();
       springBlogCategory.setCategory(newPost.getSpringBlog());
       newPost.getCategories().add(springBlogCategory);
   }

   if (newPost.getJavaBlog() != null) {
       Category javaBlogCategory = new Category();
       javaBlogCategory.setCategory(newPost.getJavaBlog());
       newPost.getCategories().add(javaBlogCategory);
   }

   postService.createPost(newPost);
   return "redirect:/posts";
}
```

If the user had selected the ‘Java Blog’ checkbox, the value of the javaBlog attribute is Java Blog. newPost.getJavaBlog() and it gets the value of the javaBlog attribute; if it is not null (or equal to the Java Blog), we need to add this category to a list of categories for this post. So, we declare a Category object first and set the category name as ‘Java Blog’. This object is then added to the categories attribute of the Post object, which is a list of all the categories related to that post. Similar is the case for the ‘Spring Blog’ category. The Post object is then sent to the business logic and gets persisted in the database.

Summary

In this module you first learnt about JDBC API which helps to connect to the database through JDBC drivers. Then the ORM framework was implemented, with JPA specification using Hibernate framework. Then you learnt about all the CRUD operations on post. You learnt how to create post, retrieve post, update post, and delete a post. Then all types of mappings were introduced. You implemented One To One mapping through User and UserProfile. Many To One / One To Many mapping was implemented between Post and User. And Many To Many mapping was implemented in Post and Category.