ڤوليتيكنيك بروني

POLITEKNIK BRUNEI

NS4307
Network Programming

Java Persistence API

Requirements

- For this topic you would need to install a database management system server.
  - https://dev.mysql.com/downloads/mysql/
- I will be personally be using MySQL Community Server, but feel free to use other database management system server.
- For the practical session, I will be setting up our Spring Application to connect to MySQL Community Server.

Configuring Spring Application

- Please make sure your web application is already set up with all the dependencies.
  - From Lecture 7 – Java Web Programming I.

- Then we need to add two more dependency to the package:
  - Spring Data JPA
  - MySQL Connector Java

Spring Data JPA

- Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This will significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed.

- Implement in our Web Application:
  - Open up your pom.xml and add the following in between the <dependencies></dependencies> tag.

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

MySQL Connector Java

- This will add JDBC driver for MySQL to your application. JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases.

- Implement in our Web Application:
  - Open up your pom.xml and add the following in between the <dependencies></dependencies> tag.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Configuring Spring Application (cont.)

- Now open up application.properties and we need to configure our web application to connect to our database.

- We need to set up the following properties:
  - spring.datasource.driver-class-name
    - The driver that will be used to connect to a database.
  - spring.datasource.url
    - URL location the database is hosted and connect to that database.
  - spring.datasource.username
    - Username used to connect to the database server
  - spring.datasource.password
    - Password used to connect to the database server.

Configuring Spring Application (cont.)

- spring.jpa.hibernate.ddl-auto
  - What you want to do to the tables in the database, everytime you start the web application.
  - The value can be:
    - create - creates the database tables, destroying previous data.
    - create-drop - drop the database tables when the application is stopped.
    - update - update the database tables.
    - validate - validate the database tables, makes no changes to the database.
- spring.jpa.show-sql
  - To show or not the sql statement in the console.
- spring.jpa.properties.hibernate.dialect
  - To tell what kind of SQL statement Hibernate should use to access and manipulate data in the database.

Example: Configuring Spring Application

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/learning
spring.datasource.username=jailanihar
spring.datasource.password=Mypassword123-

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

Updating Our POJO class

- If you're using JPA, you may use POJO class to structure how you want to store your data into your database.

- Add these two annotations to the POJO class on top of the class declaration:
  - @Entity – This is to state that the POJO class is a Database entity.
  - @Table(name = "tableName") – This is to specify the table name for the Database entity.

```
@Entity
@Table(name = "products")
public class Product {
```

Updating Our POJO class (cont.)

- The following are the constraints you can set to your POJO class attributes.
  - @Id – To specify the primary key of an entity.
  - @GeneratedValue(strategy = GenerationType.IDENTITY) – To make that attribute to be automatically generated by the database. (In this case the strategy will tell the database to auto increment the attribute).
  - @NotNull – To specify the attribute will not allow null value to be stored into the database.

- Refer to the link below under Annotation Types Summary to find out more annotation you can implement.

  http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

Updating Our POJO class (cont.)

- Lets add constraint to our POJO class.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;

@NotNull
private String name;

@NotNull
private double price;

@NotNull
private String picFile;

@NotNull
private boolean inStock;
```

Updating Our POJO class (cont.)

- We might as well implement the default constructor which is needed by Hibernate to out POJO class.

```
public Product() {}
```

Example: Product.java

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @NotNull
    private String name;

    @NotNull
    private double price;

    @NotNull
    private String picFile;

    @NotNull
    private boolean inStock;
```

Example: Product.java (cont.)

```
public Product() {}

public Product(String name, double price, String picFile, boolean inStock) {
    this.name = name;
    this.price = price;
    this.picFile = picFile;
    this.inStock = inStock;
}

public long getId() {
    return id;
}

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

Example: Product.java (cont.)

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}
```

```
public String getPicFile() {
    return picFile;
}

public void setPicFile(String picFile) {
    this.picFile = picFile;
}

public boolean isInStock() {
    return inStock;
}

public void setInStock(boolean inStock) {
    this.inStock = inStock;
}
```

Data Access Object

- Now we want to implement Data Access Object to be able to perform Create, Read, Update and Delete (CRUD) operation to the database.

- Spring framework have provided an interface called CrudRepository<T, ID> .

http://docs.spring.io/autorepo/docs/spring-data-commons/1.9.1.RELEASE/api/org/springframework/data/repository/CrudRepository.html

- So you don't need to implement the CRUD operation yourselvf.

Data Access Object (cont.)

- So lets create an interface in our data package and call it ProductDAO.java

```
@Transactional
public interface ProductDAO extends CrudRepository<Product, Long> {


}
```

This is the POJO class we want to manipulate.

This is data type of the id in our POJO class.

- ProductDAO will inherite all the methods from CrudRepository.

Data Access Object (cont.)

- The following are the methods that we will learn in the practical.
  - delete(ID id) – Deletes the entity with the given id.
  - exists(ID id) – Returns whether an entity with the given id exists.
  - findAll() – Returns all entities.
  - findOne(ID id) – Retrieves an entity by its id.
  - save(S entity) – Saves a given entity.

http://docs.spring.io/autorepo/docs/spring-data-commons/1.9.1.RELEASE/api/org/springframework/data/repository/CrudRepository.html

Replace ProductRepository with ProductDAO

- First thing that you need to do. In your ControllerClass.java, replace ProductRepository with ProductDAO.

```
@Controller
public class ControllerClass {
    // @Autowired
    // private ProductRepository productRepository;

    @Autowired
    private ProductDAO productDAO;
```

- Now you can use productDAO to access and manipulate the database.

Retrieving All Entities

- To retrieve all entities, we need to use the method provided in CrudRepository called findAll().
  - So for now lets edit the implementation that we made to retrieve all products from ProductRepository

```
@RequestMapping(value="/")
public String index(ModelMap modelMap) {
    //List<Product> products = productRepository.getAllProducts();
    List<Product> products =
            (List<Product>) productDAO.findAll();
    modelMap.put("products", products);
    return "index";
}
```

- It will not have any product since we haven't add data into database.

Retrieve One Entity

- To retrieve one entity, we need to use the method provided in CrudRepository called findOne(ID id).
  - So for now lets edit the implementation that we made to a single product using PathVariable.

```
@RequestMapping(value="/product/{id}")
public String product(@PathVariable String id, ModelMap modelMap) {
    try {
        long idInLong = Long.parseLong(id);
        if(productDAO.exists(idInLong)) {
            Product product = productDAO.findOne(idInLong);
            modelMap.put("product", product);
        }
    } catch (NumberFormatException e) {

    }
    //Product product = productRepository.findProductByName(name);
    //modelMap.put("product", product);
    return "product";
}
```

Retrieve One Entity (cont.)

- The changes we made are
  - We changed the PathVariable from name to id.
  - Since the id is Long data type we need to parse from String to Long.
  - Then we need to use the method exist(ID id) to check if the product with the id exists in our database or not.
  - If exists show the details of the product.

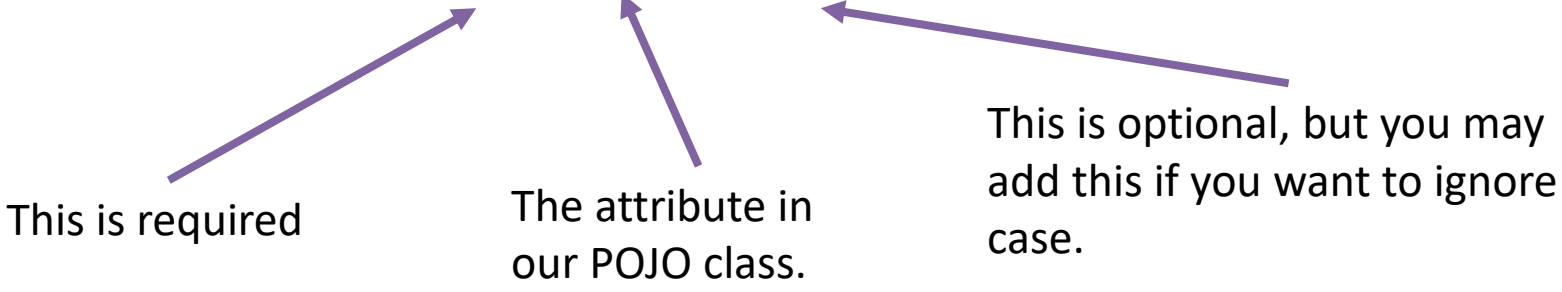
- Please make sure you change the hyperlink in index.html from ${product.name} to ${product.id} .

Find Product using Name

- To retrieve entity with a specific name, we need to implement the method ourselves.

- In our ProductDAO.java, we just add the following method.

```
List<Product> findByNameIgnoreCase(String name);
```



- Note: We don't need to implement any code for the method.

Find Product using Name (cont.)

- After you add findByNameIgnoreCase(String name) to ProductDAO.java, lets use that method in our controller class.
  - Last time we implemented a search feature to our web application. Lets change that method.

```
@RequestMapping(value="/search")
public String search(@RequestParam(required=false) String productName, ModelMap modelMap) {
    List<Product> products;
    if(productName == null) {
        //products = productRepository.getAllProducts();
        products = (List<Product>) productDAO.findAll();
    } else {
        //products = productRepository.findProductWithName(productName);
        products = productDAO.findByNameIgnoreCase(productName);
    }
    modelMap.put("products", products);
    return "search";
}
```

Find Product contain Name

- To retrieve entity containing a name, we need to implement the method ourselves.

- In our ProductDAO.java, we just add the following method.

```
List<Product> findByNameLikeIgnoreCase(String name);
```

This will allow you to find product with wild cards.

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for a single character

Find Product contain Name (cont.)

- We can use it in our search feature. We need to implement the wildcard as well.

```
@RequestMapping(value="/search")
public String search(@RequestParam(required=false) String productName, ModelMap modelMap) {
    List<Product> products;
    if(productName == null) {
        //products = productRepository.getAllProducts();
        products = (List<Product>) productDAO.findAll();
    } else {
        //products = productRepository.findProductWithName(productName);
        products = productDAO.findByNameLikeIgnoreCase("%" + productName + "%");
    }
    modelMap.put("products", products);
    return "search";
}
```

Add Data to Database

- To add data to database, we need to use the method provided in CrudRepository called save(S entity).
  - First thing that you need to implement is the html file for user to fill in.
  - Then we need to handle the HTTP request before and after the user click a button to add into the database in our ControllerClass.

- Adding data to the database is simple:
  - You need to create an object of POJO class that we created previously.
  - Then we call the method save(pojoObject)

Add Data to Database (cont.)

```
<!DOCTYPE html>
<html xmlns:th="http://thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Add New Product</title>
</head>
<body>
<form action="/addproduct" method="get">
    <h2>Add New Product</h2>
    <label>Name:</label>
    <input type="text" name="name" /><br><br>
    <label>Price:</label>
    <input type="text" name="price" /><br><br>
    <label>File:</label>
    <input type="text" name="file" /><br><br>
    <label>In Stock:</label>
    <input type="radio" name="instock" value="true"> True
    <input type="radio" name="instock" value="false"> False<br><br>
    <button type="submit">Add</button>
</form>
<label th:if="${successful == true}">Add new product successful!</label>
<label th:if="${failed == true}">Add new product failed!</label>
</body>
</html>
```

Add Data to Database (cont.)

```
@RequestMapping(value="/addproduct")
public String addProduct(
        @RequestParam(required=false) String name,
        @RequestParam(required=false) String price,
        @RequestParam(required=false) String file,
        @RequestParam(required=false) String instock,
        ModelMap modelMap
        ) {
    boolean addSuccessful;
    boolean addFailed;
    if(name==null && price==null && file==null && instock==null) {
        addSuccessful = false;
        addFailed = false;
    } else {
```

This is where you implement when user go to localhost:8080/addproduct

Add Data to Database (cont.)

```
} else {
    if(name==null || price==null || file==null || instock==null) {
        addSuccessful = false;
        addFailed = true;
    } else {
        try {
            double priceInDouble = Double.parseDouble(price);
            boolean instockInBoolean = Boolean.parseBoolean(instock);
            Product product = new Product(name, priceInDouble,
                    file, instockInBoolean);
            productDAO.save(product);
            addSuccessful = true;
            addFailed = false;
        } catch (NumberFormatException e) {
            addSuccessful = false;
            addFailed = true;
        }
    }
}
modelMap.put("successful", addSuccessful);
modelMap.put("failed", addFailed);
return "addproduct";
```

This is where you implement when you want to handle if the user omit one of the @RequestParam.

You need to make sure the price is in correct format. If not the add is not successful. If yes, it will add the product to the database.

Add Data to Database (cont.)

- The following is the code snippet from the previous method. It creates the Product object and then we call the save method and put the product object in its parameter.

```
Product product = new Product(name, priceInDouble,
        file, instockInBoolean);
productDAO.save(product);
```

Remove Data from Database

- To remove data from database, we need to use the method provided in CrudRepository called delete(ID id).
  - First thing that you need to implement is the html file for user to fill in the ID we want to delete.
  - Then we need to handle the HTTP request before and after the user click a button to delete from the database in our ControllerClass.

Remove Data from Database (cont.)

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Remove Existing Product</title>
</head>
<body>
    <form action="/removeproduct" method="post">
        <h2>Remove Existing Product</h2>
        <label>Id:</label>
        <input type="text" name="id"/>
        <button type="submit">Remove</button>
    </form>
</body>
</html>
```

Remove Data from Database (cont.)

```
@RequestMapping(value="/removeproduct")
public String removeProduct(
        @RequestParam(required=false) String id,
        ModelMap modelMap
        ) {
    boolean removeSuccessful = false;
    boolean removeFailed = false;
    if(id==null) {
        removeSuccessful = false;
        removeFailed = false;
    } else {
```

This is where you implement when user go to localhost:8080/removeproduct

Remove Data from Database (cont.)

```
    } else {
        try {
            long idInLong = Long.parseLong(id);
            if(productDAO.exists(idInLong)) {
                productDAO.delete(idInLong);
                removeSuccessful = true;
                removeFailed = false;
            } else {
                removeSuccessful = false;
                removeFailed = true;
            }
        } catch (NumberFormatException e) {
            removeSuccessful = false;
            removeFailed = true;
        }
    }
    return "removeproduct";
}
```

You need to make sure the id is in correct format. If not the remove is not successful. If yes, it will add the product to the database.

Then we check if the product exists or not. If not the remove is not successful. If exists, remove is successful.

Update Data from Database

- To update data from database, we need to use the method provided in CrudRepository called save(S entity).

- There are multiple ways to do this. One of them is to allow user to search for the id of the data first then update.
  - Implement two html file for finding data and then for updating data.
  - Then we need to handle the HTTP request before and after the user click a button to find and update data from the database in our ControllerClass.

Update Data from Database (cont.)

- This is html file to find product.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Find Existing Product</title>
</head>
<body>
    <form action="/findproduct" method="get">
        <h2>Find Existing Product</h2>
        <label>Id:</label>
        <input type="text" name="id"/>
        <button type="submit">Find</button>
    </form>
</body>
</html>
```

Update Data from Database (cont.)

- This is html file to update product.

```
<!DOCTYPE html>
<html xmlns:th="http://thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Update Existing Product</title>
</head>
<body>
<form th:action="@{'/updateproduct/' + ${product.id}}" method="get">
    <h2>Update Existing Product</h2>
    <label>Name:</label>
    <input type="text" name="name" th:value="${product.name}"/><br><br>
    <label>Price:</label>
    <input type="text" name="price" th:value="${product.price}"/><br><br>
    <label>File:</label>
    <input type="text" name="file" th:value="${product.picFile}"/><br><br>
    <label>In Stock:</label>
    <input type="radio" name="instock" value="true" th:checked="${product.inStock == true}"> True
    <input type="radio" name="instock" value="false" th:checked="${product.inStock == false}"> False<br><br>
    <button type="submit">Update</button>
</form>
</body>
</html>
```

Update Data from Database (cont.)

```
@RequestMapping(value="/findproduct")
public String findProduct(
        @RequestParam(required=false) String id
        ) {
    if(id == null) {
        return "findproduct";
    } else {
        try {
            long idInLong = Long.parseLong(id);
            if(productDAO.exists(idInLong)) {
                return "redirect:/updateproduct/" + idInLong;
            } else {
                return "findproduct";
            }
        } catch (NumberFormatException e) {
            return "findproduct";
        }
    }
}
```

If the product exists, it will redirect the user to /updateproduct/{id}

Update Data from Database (cont.)

- After finding the product, it will go to this controller.

```
@RequestMapping(value="/updateproduct/{id}")
public String updateProduct(
        @PathVariable String id,
        @RequestParam(required = false) String name,
        @RequestParam(required = false) String price,
        @RequestParam(required = false) String file,
        @RequestParam(required = false) String instock,
        ModelMap modelMap
        ) {
    try {
        long idInLong = Long.parseLong(id);
        Product product = productDAO.findOne(idInLong);
        if(product == null) {
            return "redirect:/findproduct";
        } else {
            if(name==null || price==null
                    || file==null || instock==null) {

            } else {
```

We retrieve the product object.

Update Data from Database (cont.)

```
        } else {
            try {
                double priceInDouble = Double.parseDouble(price);
                boolean instockInBoolean = Boolean.parseBoolean(instock);

                product.setName(name);
                product.setPrice(priceInDouble);
                product.setPicFile(file);
                product.setInStock(instockInBoolean);
                productDAO.save(product);
            } catch (NumberFormatException e) {

            }
        }
        modelMap.put("product", product);
        return "updateproduct";
    }
} catch (NumberFormatException e) {
    return "redirect:/findproduct";
}
}
```

Then we use the setter method to change the data in the retrieved object.

Finally we use the save method and save the changes made to the retrieved object into the database.