

Advanced DataBase Systems- Project

Adella Neeharika(nadella@kent.edu)

This is the Postgres Admin application(pgadmin). Here I have used Users and Orders tables.

The screenshot shows the pgAdmin 4 interface. The left sidebar is the Object Explorer, displaying various database objects like Schemas, Tables, and Functions. The main area is a query editor window titled 'postgres/postgres@PostgreSQL 15*'. It contains the following SQL code:

```
1 CREATE TABLE orders (
2     order_id INT PRIMARY KEY,
3     order_description VARCHAR(255) NOT NULL,
4     user_id INT,
5     FOREIGN KEY (user_id) REFERENCES Users(id)
6 );
7
8 Insert into orders(order_id,order_description,user_id)
9 values (1234,'Note books',2);
10
11 select * from orders;
12
13 SELECT o FROM Orders o JOIN FETCH o.users;
14
15 Select * from Orders;
```

Below the query editor, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is selected. At the bottom of the interface, there are status bars for 'Recorded time', 'Event', 'Process ID', and 'Payload'. The status bar also indicates 'Total rows: 2 of 2' and 'Query complete 00:00:00.370'. The bottom right corner shows 'Ln 12, Col 1'.

Below attached are Users tables's model class, controller class, repository class

The screenshot shows a Java-based Spring Boot application structure in an IDE. The project is named "CRUD-Demo". The file "Users.java" is open in the code editor, which contains the following code:

```
1 package com.Neeharika.CRUDDemo.Model;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.GeneratedValue;
5 import jakarta.persistence.GenerationType;
6 import jakarta.persistence.Id;
7
8 @Entity
9 public class Users {
10     @Id
11     private int id;
12     private String name;
13     private String email;
14
15     public Users() {
16     }
17
18     public Users(int id, String name, String email) {
19         this.id = id;
20         this.name = name;
21         this.email = email;
22     }
23
24     public int getId() {
25         return id;
26     }
27
28     public void setId(int id) {
29         this.id = id;
30     }
31
32     public String getName() {
33         return name;
34     }
35
36     public void setName(String name) {
37         this.name = name;
38     }
39
40     public String getEmail() {
41         return email;
42     }
43
44     public void setEmail(String email) {
45         this.email = email;
46     }
47 }
```

The project structure on the left includes:

- Project (Idea)
- .mvn
- src
 - main
 - java
 - com.Neeharika.CRUDDemo
 - Controller
 - OrdersController
 - UsersController
 - Model
 - Orders
 - Users
 - Repository
 - OrdersRepo
 - UserRepo
 - Service
 - OrdersService
 - UserService
 - resources
 - static
 - templates
 - application.properties
 - test
 - target
 - gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

CRUD-Demo Version control

UserRepo.java

```
package com.Neeharika.CRUDDemo.Repository;

import com.Neeharika.CRUDDemo.Model.Users;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepo extends JpaRepository<Users, Integer> { }
```

Project

- .idea
- .mvn
- src
 - main
 - java
 - com.Neeharika.CRUDDemo.Controller
 - OrdersController
 - UsersController
 - Model
 - Orders
 - Users
 - Repository
 - OrdersRepo
 - UserRepo
 - Service
 - OrdersService
 - UserService
 - resources
 - static
 - templates
 - test
 - target
 - .gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

Run Unnamed

CRUD-Demo > src > main > java > com > Neeharika > CRUDDemo > Repository > UserRepo

7:12 LF UTF-8 4 spaces

CRUD-Demo Version control

UserController.java

```
package com.Neeharika.CRUDDemo.Controller;

import com.Neeharika.CRUDDemo.Model.Orders;
import com.Neeharika.CRUDDemo.Model.Users;
import com.Neeharika.CRUDDemo.Repository.OrdersRepo;
import com.Neeharika.CRUDDemo.Repository.UserRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
public class UsersController {
    @Autowired
    private UserRepo userRepo;
    //get

    @GetMapping("/")
    public String welcomeMessage(){
        return "Welcome";
    }

    @GetMapping("/getUsers")
    public List<Users> getUsers(){
        return this.userRepo.findAll();
    }

    //get user by id
    @GetMapping("/getUsers/{id}")
    public Optional<Users> getUserId(@PathVariable(value = "id") int id ){
        Optional<Users> user;
        return user;
    }
}
```

Project

- .idea
- .mvn
- src
 - main
 - java
 - com.Neeharika.CRUDDemo.Controller
 - OrdersController
 - UsersController
 - Model
 - Orders
 - Users
 - Repository
 - OrdersRepo
 - UserRepo
 - Service
 - OrdersService
 - UserService
 - resources
 - static
 - templates
 - test
 - target
 - .gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

Run Unnamed

CRUD-Demo > src > main > java > com > Neeharika > CRUDDemo > Service > OrdersService

40:41 LF UTF-8 4 spaces

Below attached are Orders table's model,repository and controller classes

The screenshot shows the IntelliJ IDEA interface with the file `OrdersRepo.java` open in the editor. The code defines a repository interface for managing orders. It imports various Java and Spring framework packages, including `com.Neeharika.CRUDDemo.Model.Orders`, `jakarta.persistence.criteria.Order`, `org.hibernate.query.Query`, `org.springframework.data.jpa.repository.JpaRepository`, `org.springframework.data.jpa.repository.Query`, `org.springframework.stereotype.Repository`, and `java.util.List`. The interface itself extends `JpaRepository<Orders, Integer>` and contains two methods: `findAllOrdersWithUsers()` and `findAllOrdersWithUsers()`.

```
1 package com.Neeharika.CRUDDemo.Repository;
2
3 import com.Neeharika.CRUDDemo.Model.Orders;
4 import jakarta.persistence.criteria.Order;
5 //import org.hibernate.query.Query;
6 import org.springframework.data.jpa.repository.JpaRepository;
7 //import org.springframework.data.jpa.repository.Query;
8 import org.springframework.data.jpa.repository.Query;
9 import org.springframework.stereotype.Repository;
10
11 import java.util.List;
12
13 @Repository
14 public interface OrdersRepo extends JpaRepository<Orders, Integer> {
15
16     //Query("SELECT o FROM Order o JOIN FETCH o.user")
17     @Query("SELECT o FROM Orders o JOIN FETCH o.user")
18     List<Orders> findAllOrdersWithUsers();
19 }
```

The screenshot shows the IntelliJ IDEA interface with the file `OrdersController.java` open in the editor. This controller handles requests for orders. It imports `com.Neeharika.CRUDDemo.Model.Orders`, `com.Neeharika.CRUDDemo.Model.Users`, `com.Neeharika.CRUDDemo.Repository.OrdersRepo`, `jakarta.persistence.criteria.Order`, `org.springframework.beans.factory.annotation.Autowired`, `org.springframework.http.ResponseEntity`, `org.springframework.web.bind.annotation.*`, `java.util.List`, and `java.util.Optional`. The controller is annotated with `@RestController` and contains two methods: `getOrders()` and `getOrderById()`.

```
1 package com.Neeharika.CRUDDemo.Controller;
2
3 import com.Neeharika.CRUDDemo.Model.Orders;
4 import com.Neeharika.CRUDDemo.Model.Users;
5 import com.Neeharika.CRUDDemo.Repository.OrdersRepo;
6 import jakarta.persistence.criteria.Order;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.http.ResponseEntity;
9 import org.springframework.web.bind.annotation.*;
10
11 import java.util.List;
12 import java.util.Optional;
13
14 @RestController
15 public class OrdersController {
16
17     @Autowired
18     private OrdersRepo ordersRepo;
19
20     @GetMapping("/getOrders")
21     public List<Orders> getOrders(){
22         return this.ordersRepo.findAll();
23         //return this.ordersRepo.findAllOrdersWithUsers();
24     }
25
26     @GetMapping("/getOrders/{id}")
27     public Optional<Orders> getOrderById(@PathVariable(value = "id") int id){
28         Optional<Orders> order;
29         order = ordersRepo.findById(id);
30         return (order);
31     }
32 }
```

The screenshot shows the IntelliJ IDEA interface with the project 'CRUD-Demo' open. The left sidebar displays the project structure, including the 'src' directory with 'main' and 'java' sub-directories containing 'com.Neeharika.CRUDDemo' package. Inside 'java', there are 'Controller', 'Model', 'Repository', 'Service', and 'CrudDemoApplication' packages. The 'Orders.java' file is currently selected and displayed in the main editor window. The code implements a many-to-one relationship between Orders and Users, with methods for getting and setting order details and users.

```

15     @ManyToOne
16     @JoinColumn(name = "user_id", referencedColumnName = "id")
17     private Users user;
18
19     public int getOrder_id() {
20         return order_id;
21     }
22
23     public void setOrder_id(int order_id) {
24         this.order_id = order_id;
25     }
26
27     public String getOrder_description() {
28         return order_description;
29     }
30
31     public void setOrder_description(String order_description) {
32         this.order_description = order_description;
33     }
34
35     public Users getUser() {
36         return user;
37     }
38
39     public void setUser(Users user) {
40         this.user = user;
41     }
42
43     public Orders() {
44
45
46

```

Below attached is the properties of database connection

The screenshot shows the IntelliJ IDEA interface with the 'application.properties' file selected in the resources directory. A warning icon is visible in the status bar, indicating an 'Unused property'. A tooltip for the line 'spring.http.converters.preferred-json-mapper=jackson' provides options to remove the property or view more actions. The file contains standard Spring Boot configuration for a PostgreSQL database and JSON converters.

```

1 spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
2 spring.datasource.username=vs
3 spring.datasource.password=vs
4 spring.jpa.hibernate.ddl-auto=update
5 spring.h2.console.enabled=true
6 spring.jpa.show-sql=true
7 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
8 spring.sql.init.mode=always
9 spring.sql.init.platform=postgres
10 spring.jpa.defer-datasource-initialization=true
11
12
13 spring.http.converters.preferred-json-mapper=jackson
14

```

Unused property
Remove property ⌘+Delete More actions... ⌘+Delete
spring.http.converters.preferred-json-mapper="jackson"
[application.properties]

Read All Users

The screenshot shows the Postman interface with a collection named "expense_tracker". A new request is being made to the URL `http://localhost:8080/getUsers`. The method is set to GET. The response body is displayed in JSON format, showing a list of users:

```
1 [ {  
2     "id": 2,  
3     "name": "Nikku",  
4     "email": "Nikku@gmail"  
5 },  
6 {  
7     "id": 1,  
8     "name": "Nani",  
9     "email": "Nani@gmail"  
10 },  
11 {  
12     "id": 4,  
13     "name": "Neeharika",  
14     "email": "Neeharika@gmail"  
15 }  
16 ]
```

The status bar at the bottom indicates a 200 OK response with a time of 52 ms.

Read particular User

The screenshot shows the Postman interface with the same collection. A new request is being made to the URL `http://localhost:8080/getUsers/2`. The method is set to GET. The response body is displayed in JSON format, showing a single user:

```
1 [ {  
2     "id": 2,  
3     "name": "Nikku",  
4     "email": "Nikku@gmail"  
5 }]
```

The status bar at the bottom indicates a 200 OK response with a time of 124 ms.

Insert one user

The screenshot shows the Postman application interface. In the top navigation bar, 'Explore' is selected. Below it, 'My Workspace' is shown with a collection named 'expense_tracker'. A search bar at the top right contains the text 'Search Postman'.

In the main workspace, a POST request is being prepared to the URL `http://localhost:8080/getUsers`. The method dropdown shows 'POST'. The 'Body' tab is selected, showing a JSON payload:

```
1 ...
2 ... "id": 4,
3 ... "name": "Neeharika",
4 ... "email": "Neeharika@gmail"
5 ...
```

Below the request details, the response is displayed. It shows a status of `200 OK`, a time of `122 ms`, and a size of `217 B`. The response body is identical to the request body, indicating a successful insertion of a new user.

Update user

Delete User

The screenshot shows the Postman interface with a DELETE request to `http://localhost:8080/getUsers/3`. The request body contains the user ID to be deleted:

```
1 id: 4,
2   "name": "Neeharika",
3   "email": "Neeharika@gmail"
4 }
```

The response status is 200 OK with the message "Deleted: chinni".

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/getUsers`. The response is a JSON array of users:

```
1 [
2   {
3     "id": 2,
4     "name": "Nikku",
5     "email": "Nikku@gmail"
6   },
7   {
8     "id": 1,
9     "name": "Nani",
10    "email": "Nani@gmail"
11  },
12  {
13    "id": 3,
14    "name": "chinni",
15    "email": "chinni@gmail"
16  }
17 ]
```

Delete order

The screenshot shows the Postman interface with a DELETE request to `http://localhost:8080/getOrders/1234`. The request body is a JSON object representing an order. The response status is 200 OK, and the response body contains the message "Deleted: 1234 Note books".

```
1
2   ...
3   ...
4   ...
5   ...
6   ...
7   ...
8   ...
9   ...
```

Body: Deleted: 1234 Note books

The screenshot shows the Postman interface with a PUT request to `http://localhost:8080/getOrders/12`. The request body is a JSON object representing an order. The response status is 200 OK, and the response body contains the same JSON object as the request body.

```
1
2   ...
3   ...
4   ...
5   ...
6   ...
7   ...
8   ...
9   ...
```

Above is Update particular order

Read particular order

The screenshot shows the Postman interface with a collection named "expense_tracker". A GET request is made to `http://localhost:8080/getOrders/1234`. The response status is 200 OK, time is 72 ms, and size is 267 B. The response body is a JSON object:

```
1  {
2      "order_id": 1234,
3      "order_description": "Note books",
4      "user": {
5          "id": 2,
6          "name": "Nikku",
7          "email": "Nikku@gmail"
8      }
9 }
```

The screenshot shows the Postman interface with a collection named "expense_tracker". A GET request is made to `http://localhost:8080/getOrders`. The response status is 200 OK, time is 85 ms, and size is 383 B. The response body is a JSON array:

```
1 [
2     {
3         "order_id": 1234,
4         "order_description": "Note books",
5         "user": {
6             "id": 2,
7             "name": "Nikku",
8             "email": "Nikku@gmail"
9         }
10    },
11    [
12        {
13            "order_id": 123456,
14            "order_description": "third Perfume bottle",
15            "user": {
16                "id": 1,
17                "name": "Nani",
18                "email": "Nani@gmail"
19            }
20        }
21    ]
22 ]
```

Above is read all orders

- 1.I have used SpringBoot with Postgres and Postman tool**
 - 2.Used two tables- Users and Orders**
 - 3.Users(id,name,email)**
Orders(order_id,order_description,User_id) user_id references to id in Users table acts as foreign key
 - 4. This is the simple web api, we can integrate it to any service who are in need of it.(I am working on it to add Simple UI to it), I will push my remaining work to GIT.**
 - 5. The reason why I used two tables is: I don't want to make ambiguity by putting orders and users in the same table, another one is I want to create many to one relations from orders to user because users may have many orders.**
- The last important one is, i wanted to show relations between them maintain scalability and achieve Normalisation.
- 6. I have implemented controllers for CRUD operations on Users and Orders separately. You can create a user separately but when you create an Order, you have to associate its user as well(Thats is how I have implemented these operations).**
 - 7.The reasons why i choose this database is that PostgreSQL is reliable and powerful dbms, spring boot is known for its enterprise grade features, scalability, it provides comprehensive ecosystem that includes Spring Data JPA for easy database integrations which simplifies CRUD operations.**