TECHNOLOGY

Caltech | Center for Technology & Management Education

Develop Java Backend for Admin Dashboard

Powerd by simplilearn

TECHNOLOGY

Caltech | Center for Technology & Management Education

DAO Design Pattern

Powerd by simplilearn

You Already Know

Before we begin, let's recall what we have covered till now:

MongoDB

Recap

Maven Project for Backend

- Created a Maven project with an archetype as a web app in Eclipse EE

Developed POJO Classes

- Created various classes for the Admin and End User Projects
- Developed POJO with constructors, getters, setters and toString

Project Configuration

- Configured MySQL, Servlet, JSP, and Apache Tomcat Web Server

Build and Execute

- Built and executed the Maven Web App Project
- Packaged the Web Project as a war file

A Day in the Life of a Full Stack Developer

As a full stack web developer, our key role is to develop both client and server software.

- Angular and Node can be used to build the front end of the web page.

- Spring Boot, Java, and MySQL or MongoDB can be used to build at the back end.

A Day in the Life of a Full Stack Developer

Now, Bob needs to develop the design Pattern DAO in a generic manner. So, Bob brainstorms and finds a solution.

Let me use Java, OOPS, and JDBC to develop the design Pattern DAO in a generic manner.

In this lesson, we will learn the Java, OOPS, and JDBC skills for created Maven Project and develop the design Pattern DAO in a generic manner. Further, we will implement CRUD Operations with JDBC for various Models and help Bob to complete his task effectively and quickly.

Learning Objectives

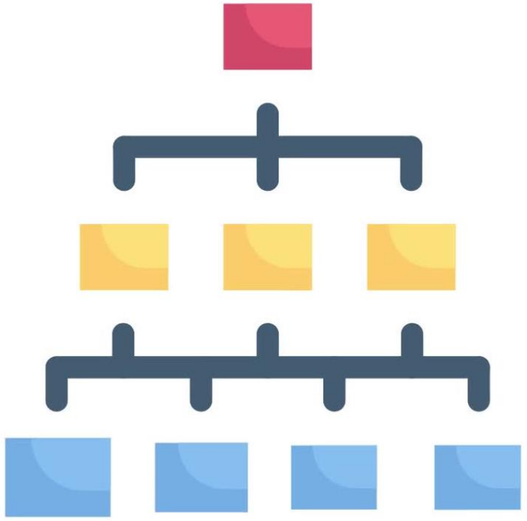
By the end of this lesson, you will be able to:

- Implement DAO design pattern in a generic manner
- Connect with MySQL using JDBC
- Create CRUD operations for the admin models
- Decouple the model from persistence layer

Develop Generic DAO Design Pattern for the Admin Backend

Data Access Object (DAO)

DAO is a structural design pattern. The main role is to decouple the application layer from the persistence layer using abstraction.



The DAO APIs will hide all the complexity for CRUD operations and hence both layers work in isolation.

Create DAO Interface: DAO.java

Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

Interface uses generic concept of Java i.e. <T> for Type

Create DAO Interface: DAO.java

Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

get method return the object based on id as input

Create DAO Interface: DAO.java

Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

getAll method will return list of all the objects and can serve as cache

Create DAO Interface: DAO.java

Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

save method shall save the object passed as input

Create DAO Interface: DAO.java

Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

update method shall update the object passed as input

Create DAO Interface: DAO.java

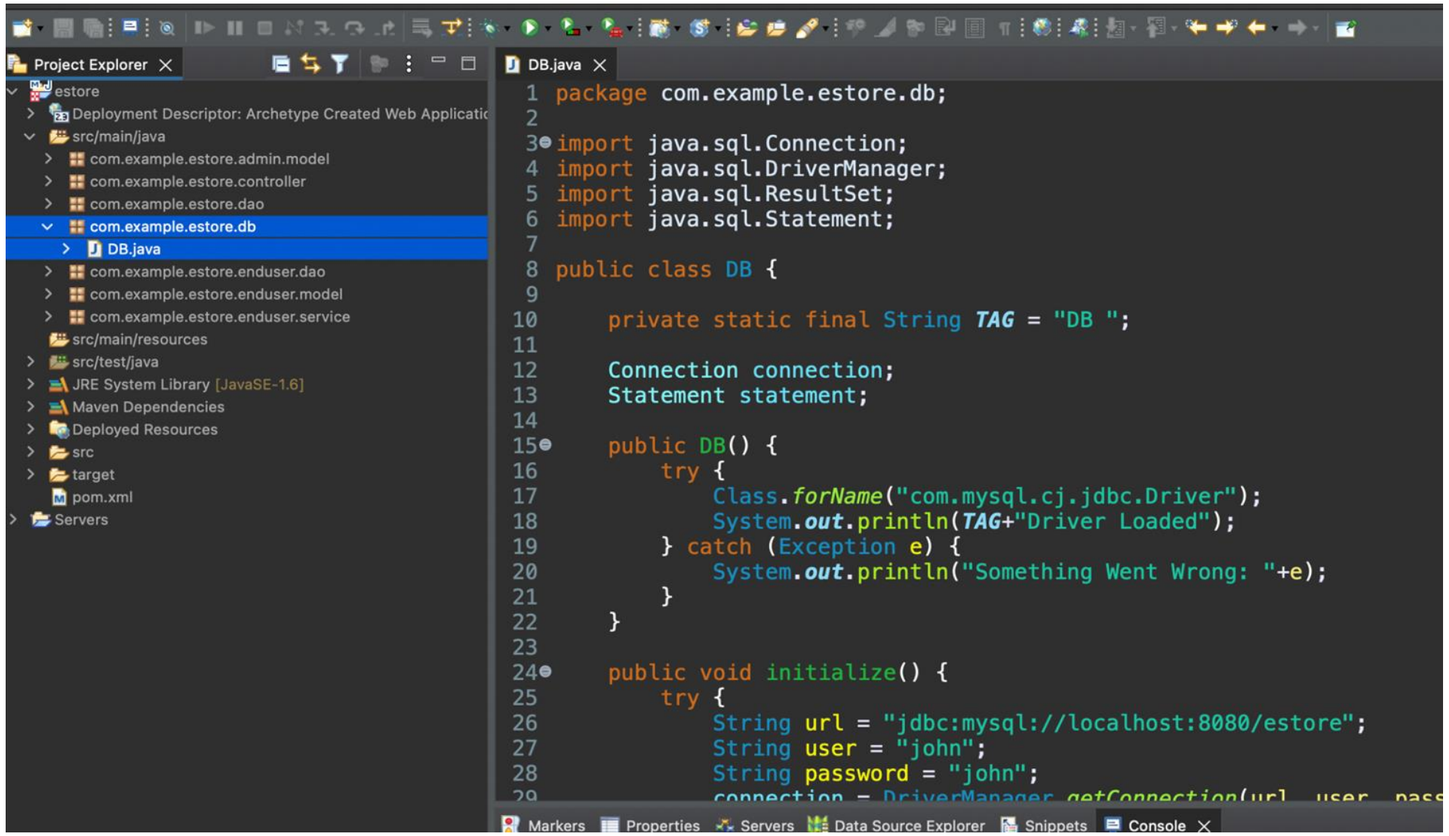
Create a new interface named DAO under the package com.example.estore.dao

```
package com.example.estore.dao;
import java.util.List;

// Generic Interface for CRUD Operations
public interface DAO<T> {

        T get(long id);
        List<T> getAll();
        void save(T object);
        void update(T object);
        void delete(long id);
}
```

delete method shall delete the object based on id as input

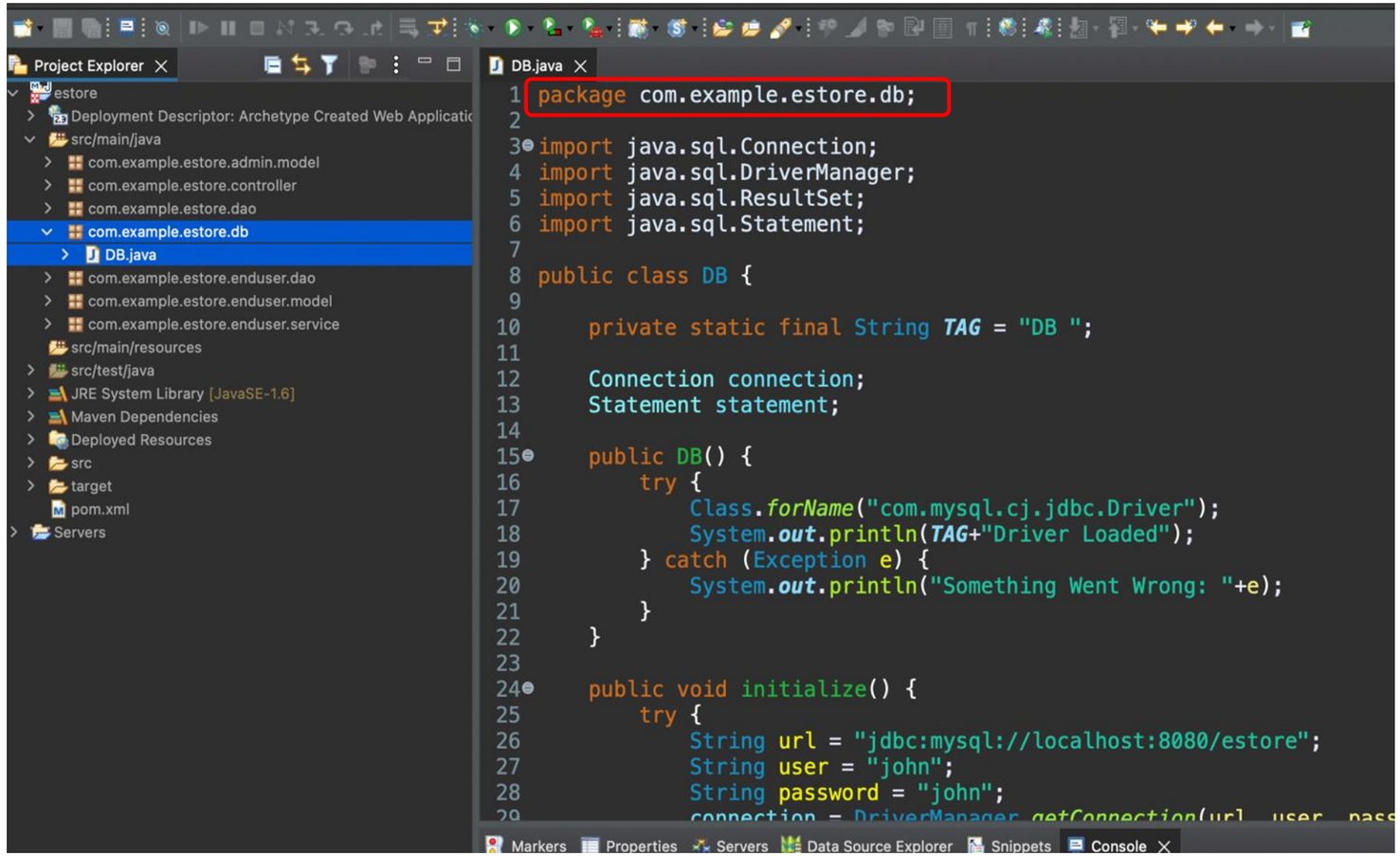
DB.java

Create DB class which will have the DB Connectivity and other operations for the database operations



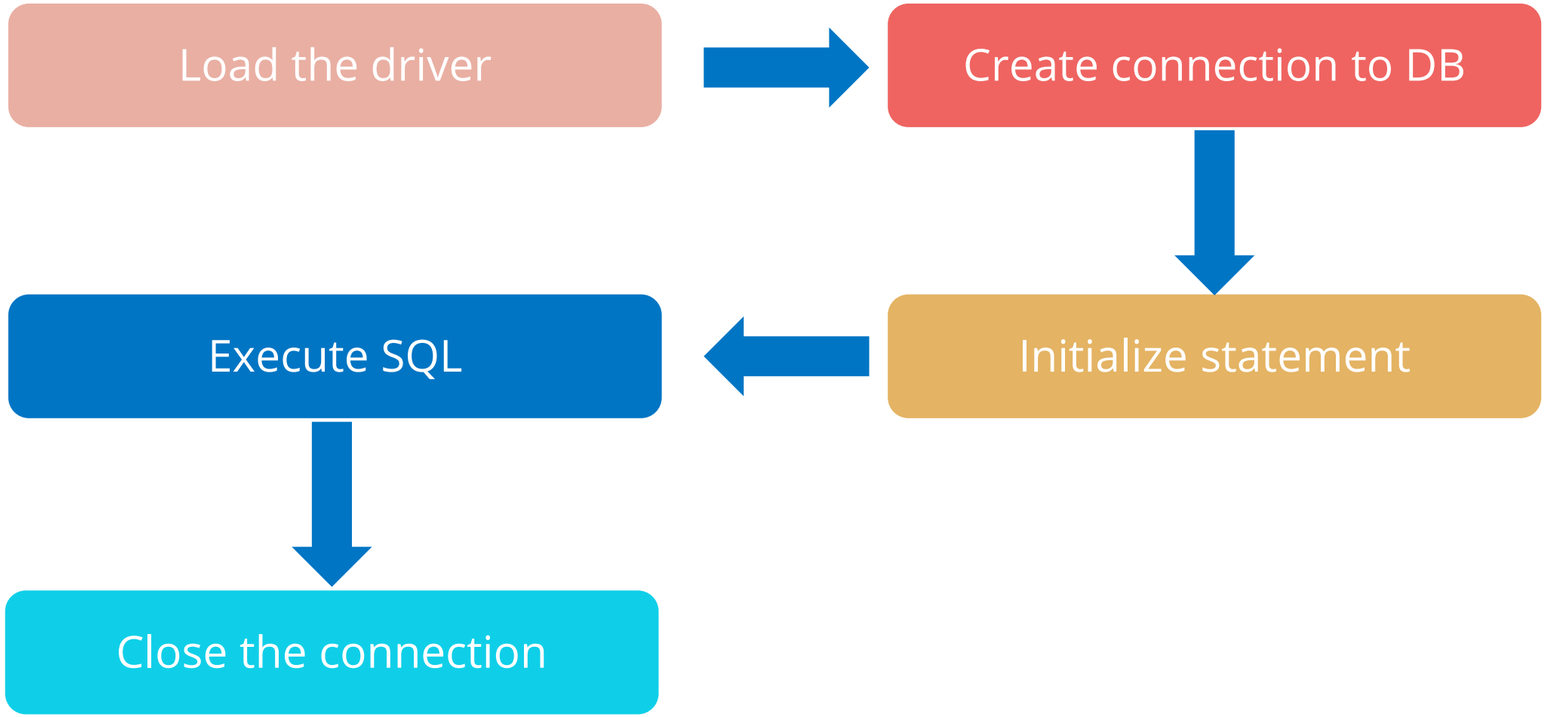
DB.java

Under the package com.example.estore.db, create the class DB



DB.java

Implementations to be included are:



DB.java: With Singleton Design Pattern

DB objects will be used in various DAO Objects. They are used by Singleton Design Pattern for better memory management.

DB.java

```java

public class DB {

    private static final String TAG = "DB ";
    private static DB db = new DB();


    public static DB getDB() {
        return db;
    }


    Connection connection;
    Statement statement;

    private DB() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println(TAG+"Driver Loaded");
            initialize();
        } catch (Exception e) {
            System.out.println("Something Went Wrong: "+e);
        }
    }

    public void initialize() {
        try {
            String url = "jdbc:mysql://localhost:8080/estore";
            String user = "john";
            String password = "john";
            connection = DriverManager.getConnection(url, user, password);
            System.out.println(TAG+"Connection Created");
            statement = connection.createStatement();
            System.out.println(TAG+"Statement Created");
        } catch (Exception e) {
            System.out.println("Something Went Wrong: "+e);
        }
    }
```

DB.java: With Singleton Design Pattern

Reference Code for DB.java after singleton:

```
public class DB {

        private static final String TAG = "DB ";
        private static DB db = new DB();


        public static DB getDB() {
                return db;
        }


        Connection connection;
        Statement statement;
```

DB.java: With Singleton Design Pattern

Reference Code for DB.java after singleton:

```
private DB() {
                try {
                        Class.forName("com.mysql.cj.jdbc.Driver");
                        System.out.println(TAG+"Driver Loaded");
                        initialize();
                } catch (Exception e) {
                        System.out.println("Something Went Wrong: "+e);
                }
        }
```

DB.java: With Singleton Design Pattern

Reference Code for DB.java after singleton:

```
public void initialize() {
                try {
                        String url = "jdbc:mysql://localhost:8080/estore";
                        String user = "john";
                        String password = "john";
                        connection = DriverManager.getConnection(url, user, password);
                        System.out.println(TAG+"Connection Created");
                        statement = connection.createStatement();
                        System.out.println(TAG+"Statement Created");
                } catch (Exception e) {
                        System.out.println("Something Went Wrong: "+e);
                }
        }
```

DB.java: With Singleton Design Pattern

Reference Code for DB.java after singleton:

```
public int executeUpdate(String sql) {
                int result = 0;
                try {
                        System.out.println(TAG+"Executing SQL "+sql+" ...");
                        result = statement.executeUpdate(sql);
                        System.out.println(TAG+"Statement Executed Successfully");
                } catch (Exception e) {
                        System.out.println("Something Went Wrong: "+e);
                }
                return result;
        }
```

DB.java: With Singleton Design Pattern

Reference Code for DB.java after singleton:

```
public ResultSet executeQuery(String sql) {
                ResultSet set = null;
                try {
                        System.out.println(TAG+"Executing SQL "+sql+" ...");
                        set = statement.executeQuery(sql);
                        System.out.println(TAG+"Statement Executed Successfully");
                } catch (Exception e) {
                        System.out.println("Something Went Wrong: "+e);
                }
                return set;
        }
```

DB.java: With Singleton Design Pattern

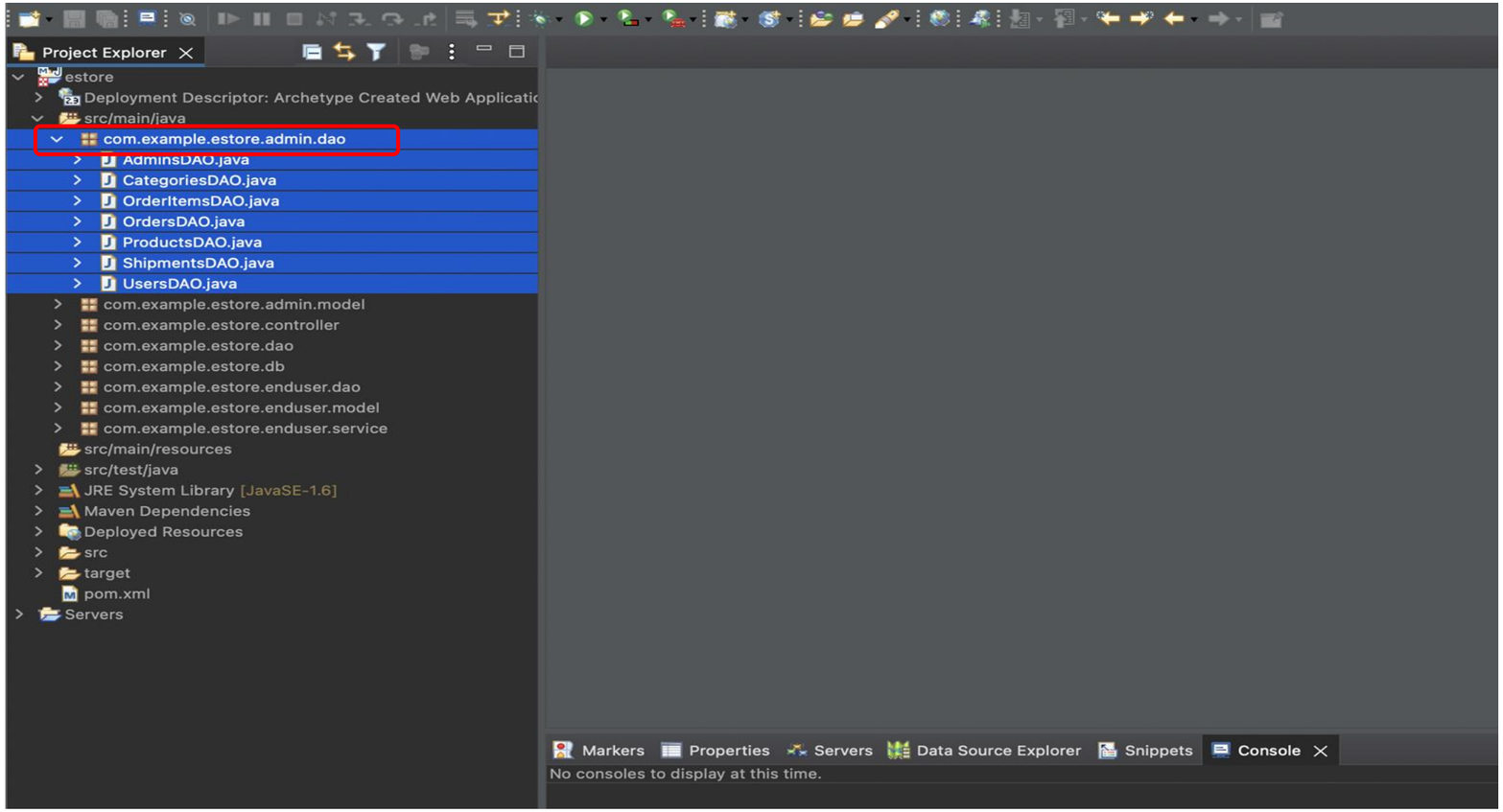
Reference Code for DB.java after singleton:

```
public void close() {
                try {
                        connection.close();
                        System.out.println(TAG+"Connection Closed");
                } catch (Exception e) {
                        System.out.println("Something Went Wrong: "+e);
                }
        }

}
```

Implement CRUD Operations

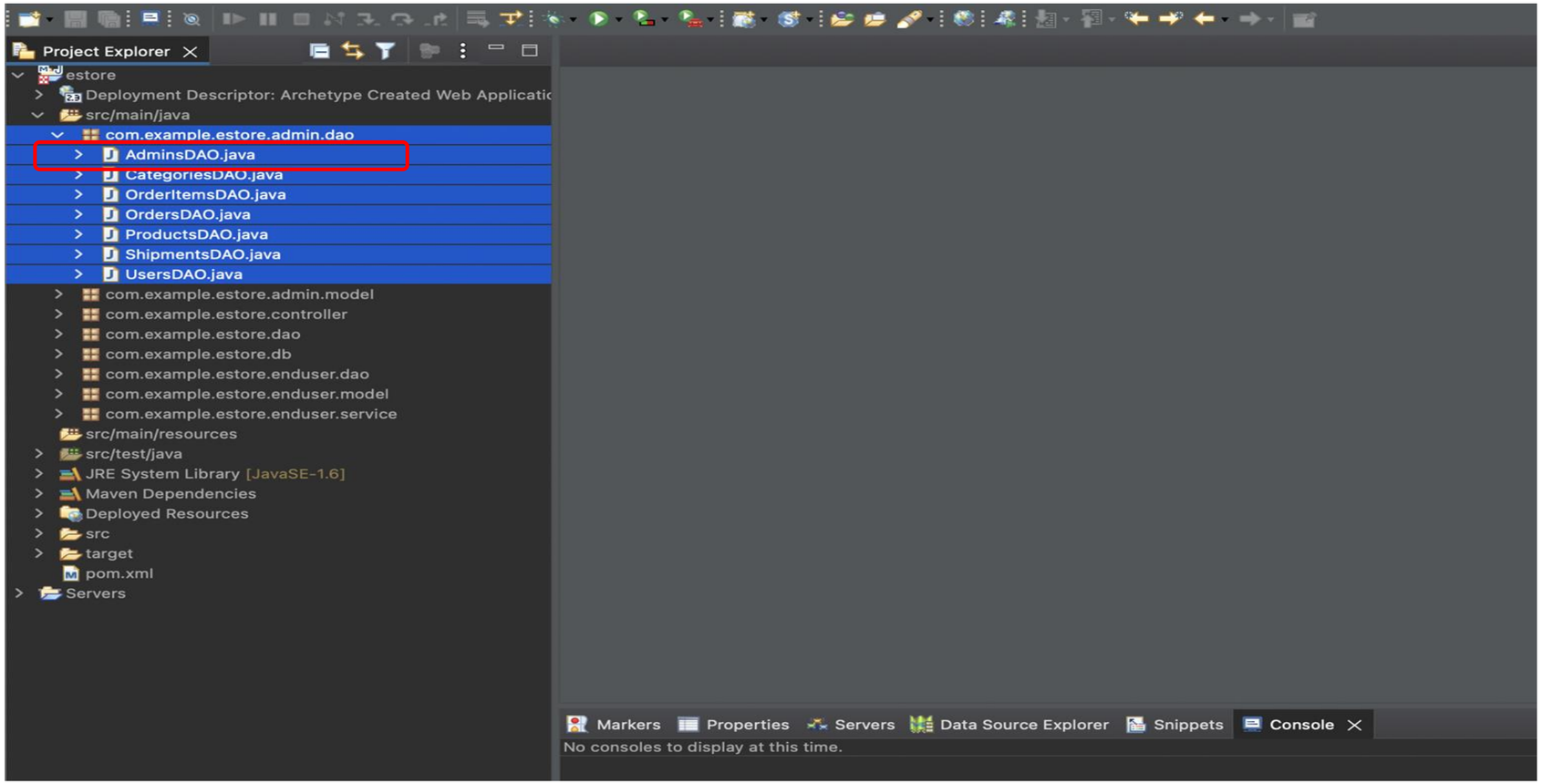
Create Classes Implementing DAO for the Admin Dashboard

Create various classes implementing the DAO interface to perform CRUD Operations for the Admin Dashboard



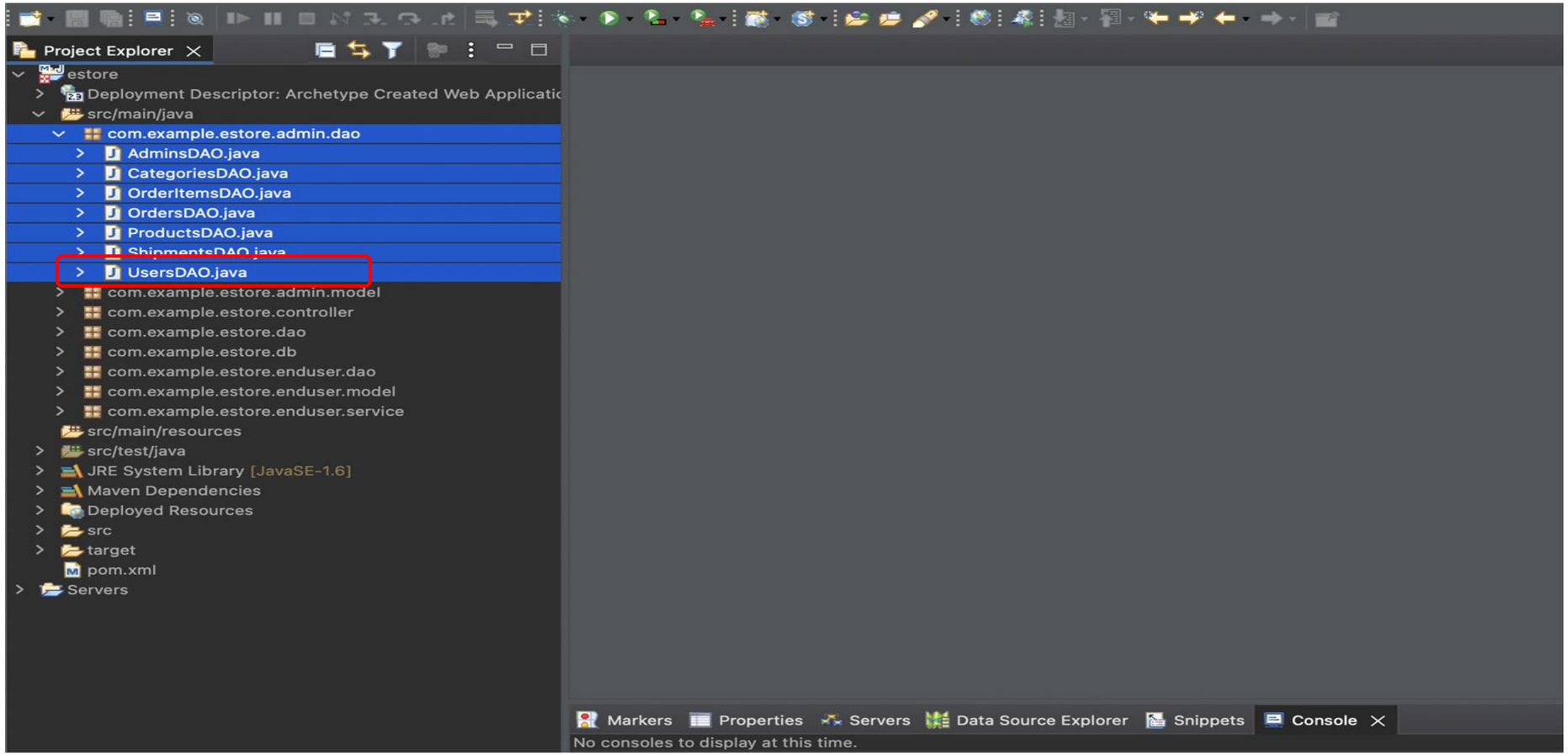
Create Classes Implementing DAO for the Admin Dashboard

AdminsDAO.java: CRUD operations for the admin users and authentication of the admin's users



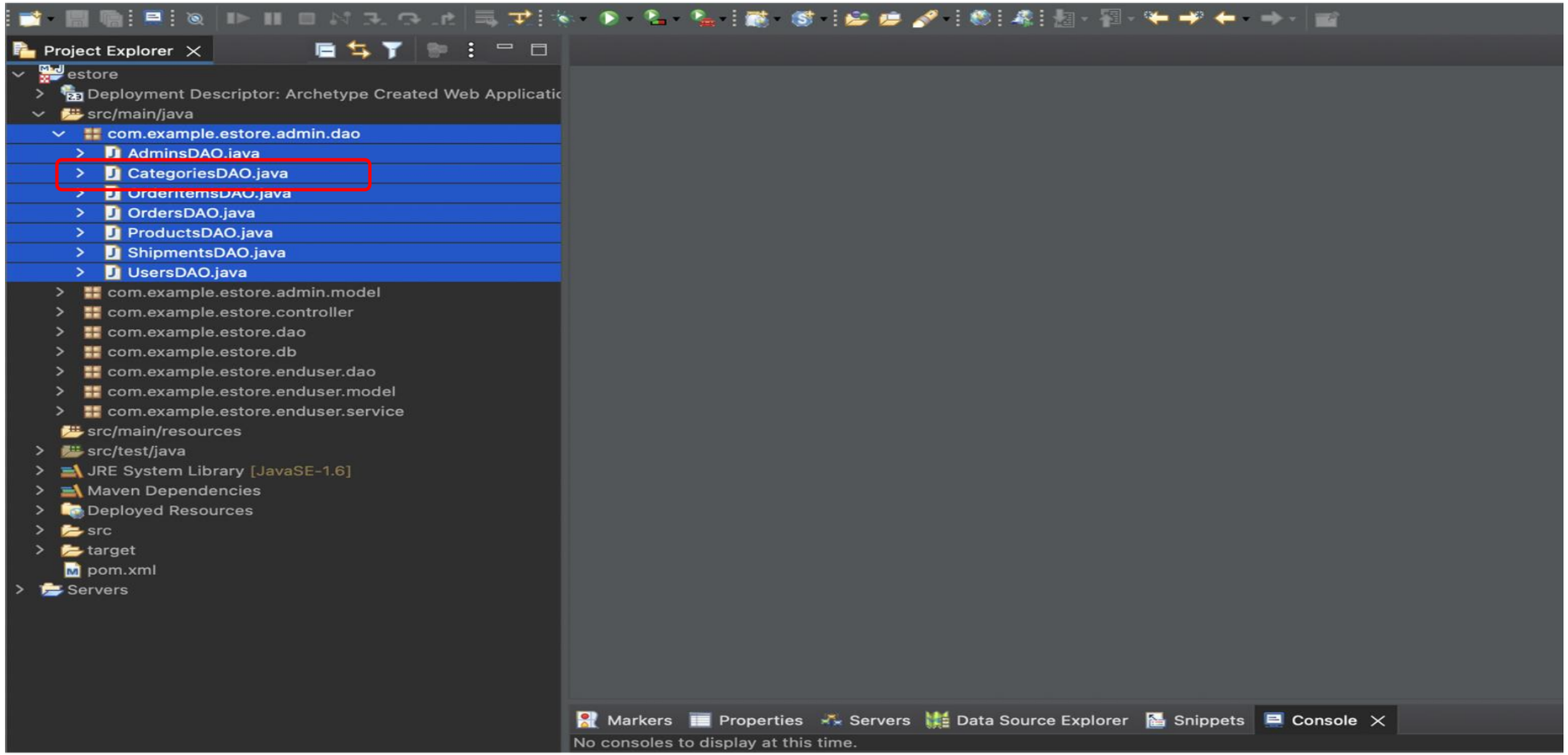
Create Classes Implementing DAO for the Admin Dashboard

UsersDAO.java: CRUD operations registered on the web backend



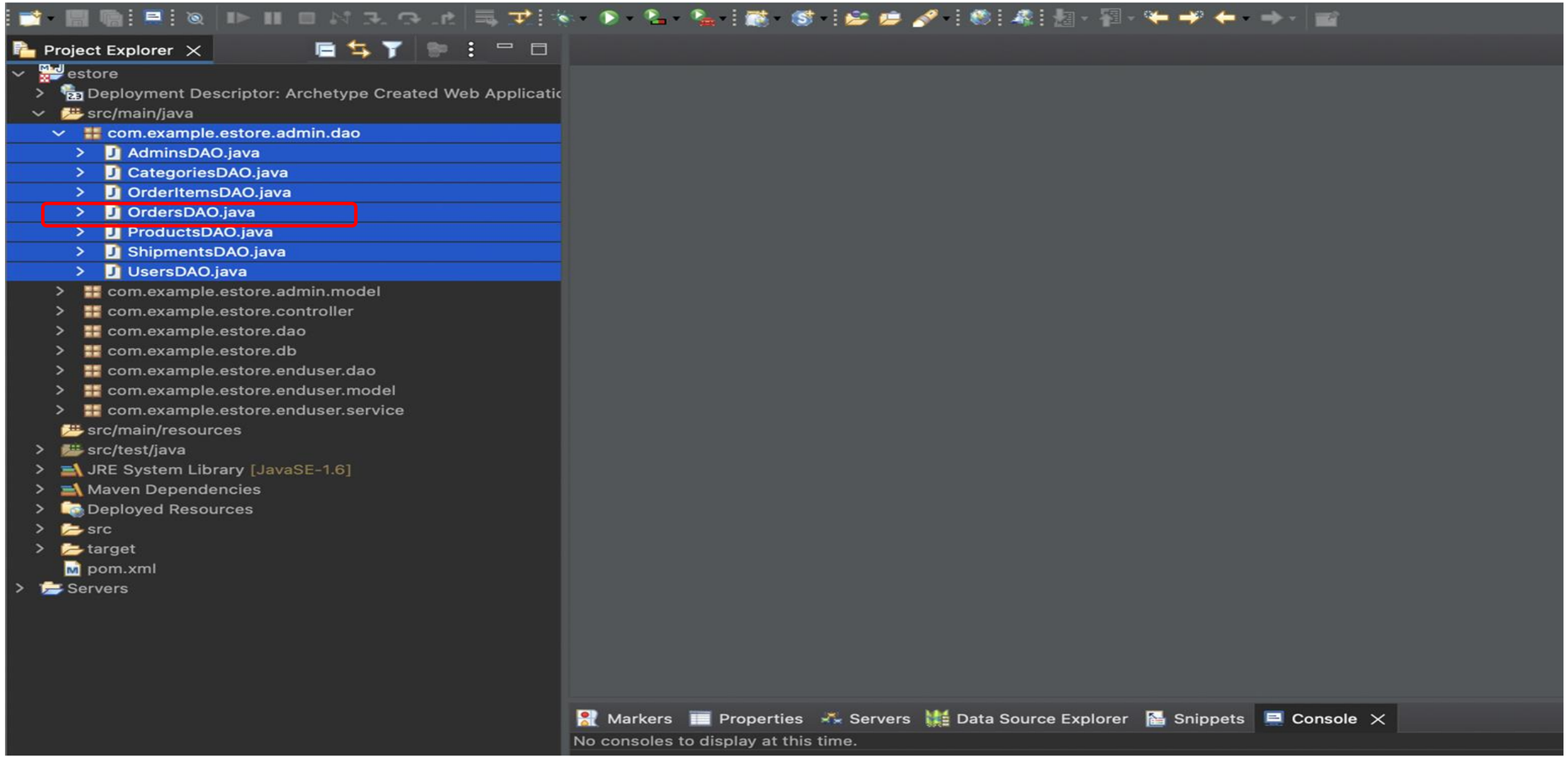
Create Classes Implementing DAO for the Admin Dashboard

CategoriesDAO.java: CRUD operations for the product categories



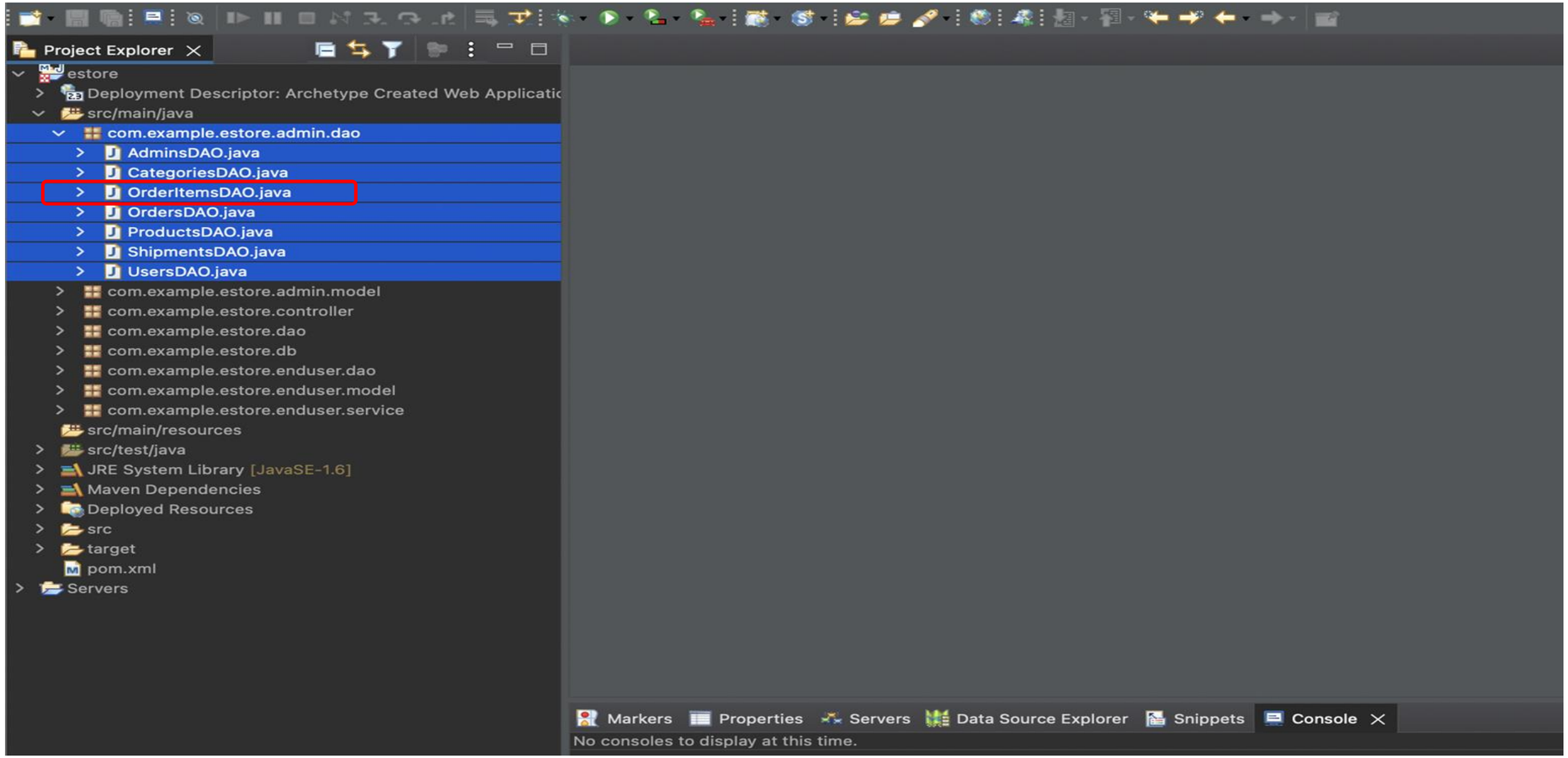
Create Classes Implementing DAO for the Admin Dashboard

OrdersDAO.java: CRUD operations for the orders placed by the users



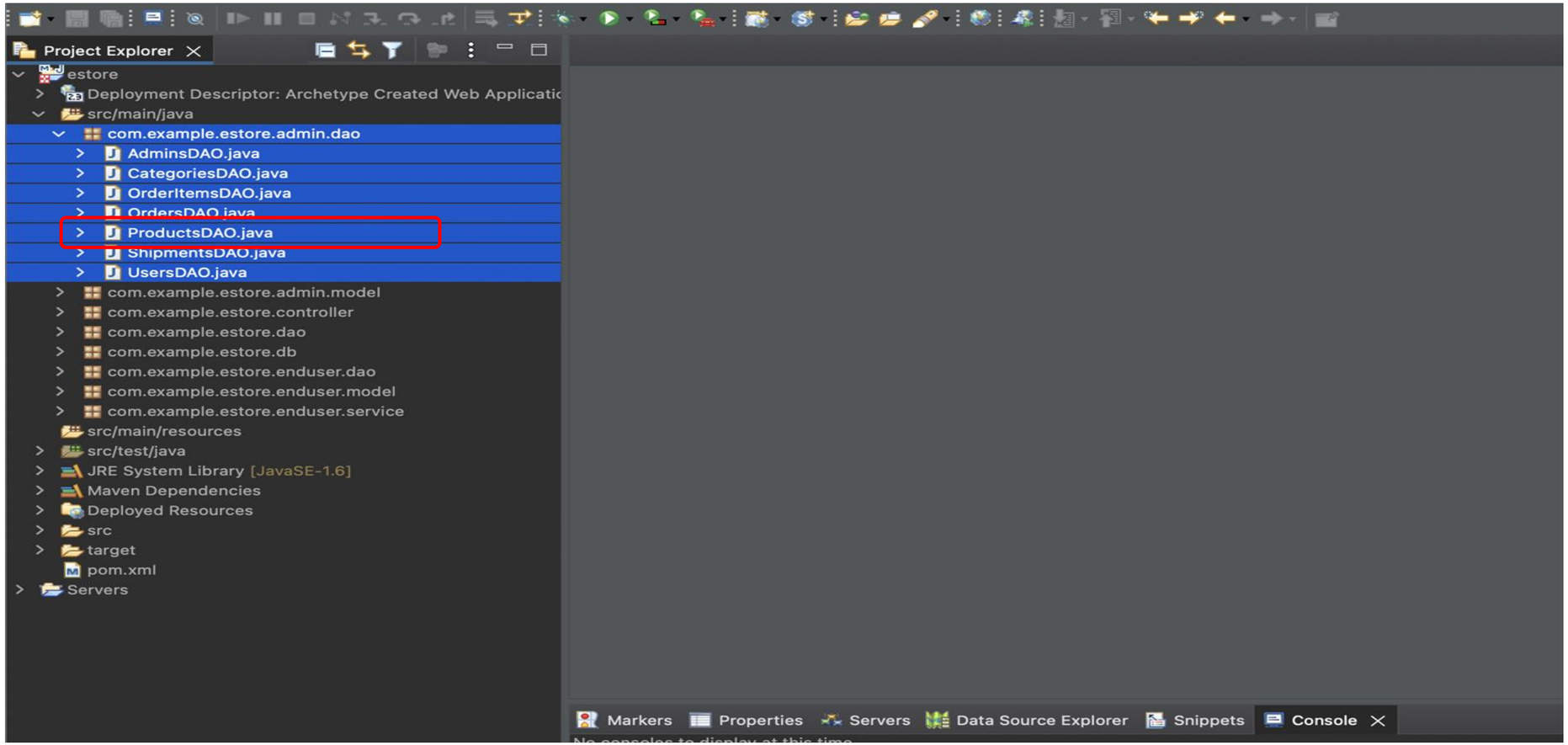
Create Classes Implementing DAO for the Admin Dashboard

OrderItemsDAO.java: CRUD operations for the products available in an order placed by users.



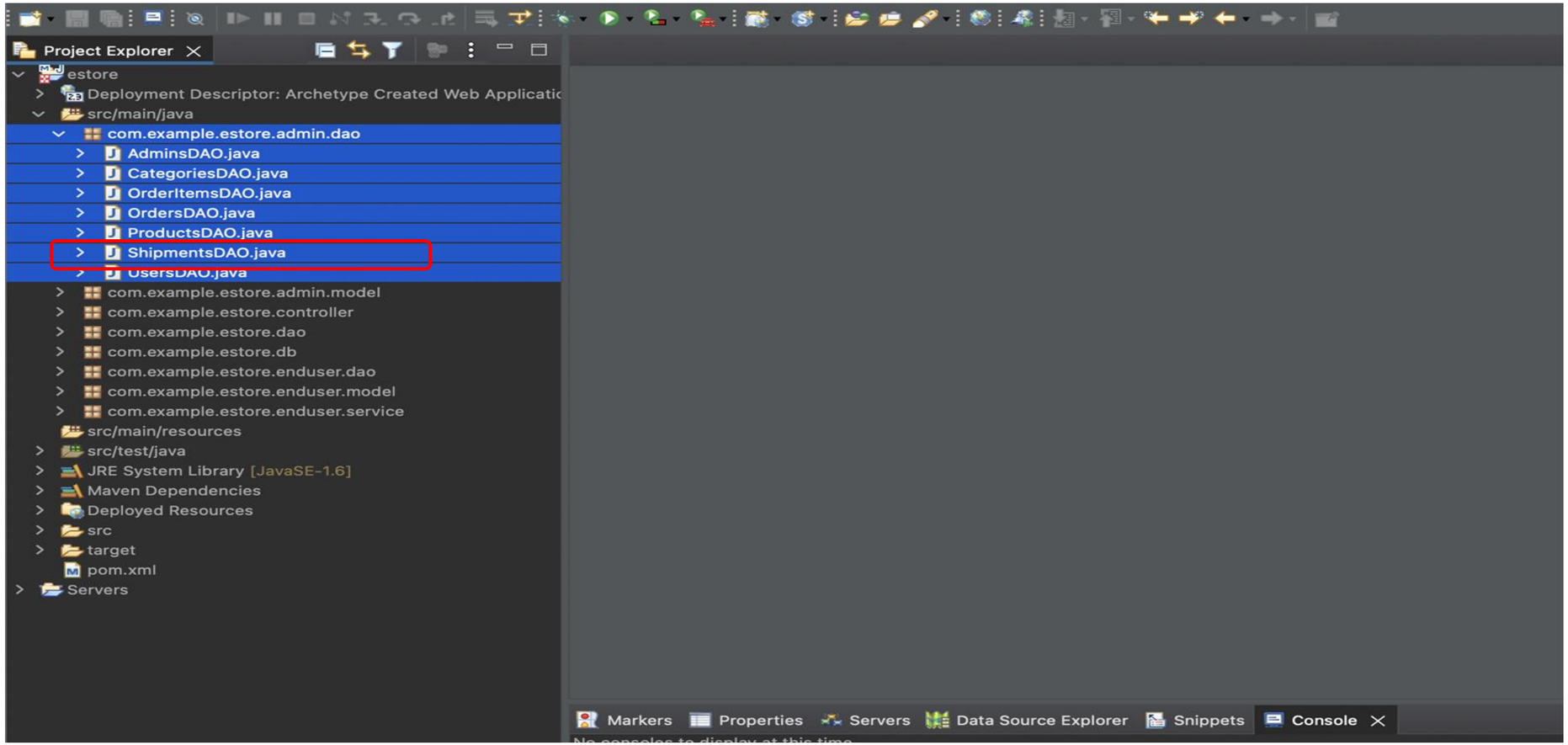
Create Classes Implementing DAO for the Admin Dashboard

ProductsDAO.java: CRUD operations the products along with the category to be added by admin for end users



Create Classes Implementing DAO for the Admin Dashboard

ShipmentsDAO.java: CRUD Operations the placed order shipped for a users



AdminsDAO.java

In order to perform CRUD operations for the admin authentication, the user needs the admin's table and admins model.

In the admins DAO class, DB is initialized to make connections and execute SQL statements.

```
package com.example.estore.admin.dao;

import com.example.estore.admin.model.Admins;
import com.example.estore.dao.DAO;
import com.example.estore.db.DB;


public class AdminsDAO implements DAO<Admins>{

        DB db = DB.getDB();
        //.....
}
```

UsersDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the admin record
- execute Query to fetch the details of an admin based on the ID or fetch the details of all the admins as a list

AdminDAO.java : Implementation of DAO Methods

In admins DAO class implements the DAO methods.

```
public class AdminsDAO implements DAO<Admins>{
@Override
        public Admins get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
        @Override
        public List<Admins> getAll() {
                // TODO Auto:generated method stub
                return null;
        }
        @Override
        public void save(Admins object) {
                // TODO Auto:generated method stub
        }
        @Override
        public void update(Admins object) {
                // TODO Auto:generated method stub
        }
        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub
        }
}
```

AdminsDAO.java : Login Method for Authentication

To login the admin to the dashboard, authenticate the admin with email and password.

```
public void login(Admins object) {

                try {
                        String sql = "select * from Admins where email =
'"+object.getEmail()+"' and password = '"+object.getPassword()+"'";
                        ResultSet set = db.executeQuery(sql);
```

AdminsDAO.java : Login Method for Authentication

Create an additional method other than CRUD operations

```
if(set.next()) {
                                object.setAdminId(set.getInt("adminId"));
                                object.setFullName(set.getString("fullName"));
                                object.setEmail(set.getString("email"));
                                object.setLoginType(set.getInt("loginType"));
                                object.setPassword(set.getString("password"));
                                String date = set.getString("addedOn");
                                SimpleDateFormat format = new SimpleDateFormat("YYYY:MM:DD");
                                Date addedOn = format.parse(date);
                                object.setAddedOn(addedOn);
                        }
                }catch(Exception e) {
                        System.out.println("Something went wrong: "+e);
                }
        }
```

UsersDAO.java : Implementation of DAO Methods

In order to perform CRUD operations for the user and details, the user must already have the user table and user model created.

In the UsersDAO class, DB is initialized to make connections and execute SQL statements.

```
package com.example.estore.admin.dao;

public class UsersDAO implements DAO<Users>{

        DB db = DB.getDB();

        @Override
        public Users get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

UsersDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the user record
- execute Query to fetch the details of a user based on id or fetch the details of all the users as list

UsersDAO.java : Implementation of DAO Methods

In Users DAO class implements the DAO methods.

```
        public List<Users> getAll() {
                // TODO Auto:generated method stub
                return null;
        }

        @Override
        public void save(Users object) {
                // TODO Auto:generated method stub
        }

        @Override
        public void update(Users object) {
                // TODO Auto:generated method stub
        }

        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub

        }

}
```

CategoriesDAO.java : Implementation of DAO Methods

In Order to perform CRUD Operations for the Product Categories and details of the Product Category we got Categories Table and Categories Model created already.

In the CategoriesDAO class, DB is initialized to make connections and execute SQL statements.

```
package com.example.estore.admin.dao;

public class CategoriesDAO implements DAO<Categories>{

        DB db = DB.getDB();

        @Override
        public Categories get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

CategoriesDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the product category record
- execute Query to fetch the details of a user based on ID or fetch the details of all the product categories as a list

CategoriesDAO.java : Implementation of DAO Methods

In Categories DAO class implements the DAO methods.

```
        @Override
        public List<Categories> getAll() {
                // TODO Auto:generated method stub
                return null;
        }

        @Override
        public void save(Categories object) {
                // TODO Auto:generated method stub

        }

        @Override
        public void update(Categories object) {
                // TODO Auto:generated method stub

        }

        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub
        }
}
```

ProductsDAO.java : Implementation of DAO Methods

In Order to perform CRUD Operations for the Product and details of the Product we got Products Table and Products Model created already.

In the ProductsDAO class, DB is initialized to make connections and execute SQL statements.

```
package com.example.estore.admin.dao;


public class ProductsDAO implements DAO<Products>{

        DB db = DB.getDB();

        @Override
        public Products get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

ProductsDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the products record
- execute Query to fetch the details of a user based on ID or fetch the details of all the products as a list

ProductsDAO.java : Implementation of DAO Methods

In Products DAO class implements the DAO methods.

```
        @Override
        public List<Products> getAll() {
                // TODO Auto:generated method stub
                return null;
        }

        @Override
        public void save(Products object) {
                // TODO Auto:generated method stub

        }
        @Override
        public void update(Products object) {
                // TODO Auto:generated method stub

        }
        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub
        }
}
```

OrdersDAO.java : Implementation of DAO Methods

In Order to perform CRUD Operations for the Orders and details of the User we got Orders Table and Orders Model created already.

In the OrdersDAO class, we will initialize the DB to make connections and execute SQL statements.

```
package com.example.estore.admin.dao;

public class OrdersDAO implements DAO<Orders>{

        DB db = DB.getDB();

        @Override
        public Orders get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

OrdersDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the order record.
- execute Query to fetch the details of a user based on ID or fetch the details of all the Orders as a list

OrdersDAO.java : Implementation of DAO Methods

In Orders DAO class implements the DAO methods.

```
@Override
        public List<Orders> getAll() {
                // TODO Auto:generated method stub
                return null;
        }
        @Override
        public void save(Orders object) {
                // TODO Auto:generated method stub

        }
        @Override
        public void update(Orders object) {
                // TODO Auto:generated method stub

        }

        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub

        }
}
```

OrderItemsDAO.java : Implementation of DAO Methods

In Order to perform CRUD Operations for the OrderItems and details of the OrderItems we got OrderItems Table and OrderItems Model created already.

In the OrderItemsDAO class, we will initialize the DB so as to make the connection and execute SQL statements.

```
package com.example.estore.admin.dao;

public class OrderItemsDAO implements DAO<OrderItems>{

        DB db = DB.getDB();

        @Override
        public OrderItems get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

OrderItemsDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the OrderItems record
- execute Query to fetch the details of a user based on ID or fetch the details of all the OrderItemsas list

OrderItemsDAO.java : Implementation of DAO Methods

In OrderItems DAO class implements the DAO methods.

```
@Override
        public List<OrderItems> getAll() {
                // TODO Auto:generated method stub
                return null;
        }

        @Override
        public void save(OrderItems object) {
                // TODO Auto:generated method stub

        }

        @Override
        public void update(OrderItems object) {
                // TODO Auto:generated method stub
        }

        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub

        }
}
```

ShipmentsDAO.java : Implementation of DAO Methods

In Order to perform CRUD Operations for the Shipments and details of the Shipments we got Shipments Table and Shipments Model created already.

In the ShipmentsDAO class, we will initialize the DB to make a connection and execute SQL statements.

```
package com.example.estore.admin.dao;

public class OrderItemsDAO implements DAO<OrderItems>{

        DB db = DB.getDB();

        @Override
        public OrderItems get(long id) {
                // TODO Auto:generated method stub
                return null;
        }
```

ShipmentsDAO.java : Implementation of DAO Methods

Implement the methods from the DAO to perform CRUD operations:

- Execute Update for insert, update, and delete of the Shipments record
- execute Query to fetch the details of a user based on ID or fetch the details of all the Shipments as a list

ShipmentsDAO.java : Implementation of DAO Methods

In Shipments DAO class implements the DAO methods.

```
@Override
        public List<Shipments> getAll() {
                // TODO Auto:generated method stub
                return null;
        }

        @Override
        public void save(Shipments object) {
                // TODO Auto:generated method stub

        }

        @Override
        public void update(Shipments object) {
                // TODO Auto:generated method stub
        }

        @Override
        public void delete(long id) {
                // TODO Auto:generated method stub

        }
}
```

Key Takeaways

- DAO design patterns are implemented generically.
- Singleton Design Patterns are implemented for DB.
- CRUD operations are implemented for various models in the admin.
- CRUD operations are tested for various models in the admin.

Before the Next Class

Since you have successfully completed this session, before the next discussion you should go through:

- Spring

What’s Next?

Now, user have finished developing Java Backend for Admin Dashboard. In our next live session, the user will:

- Perform CRUD operations with DB
- Work with Design Patterns
- Implement the Backend DAO module for end user web app
- Work with Cart and Wishlist for end user