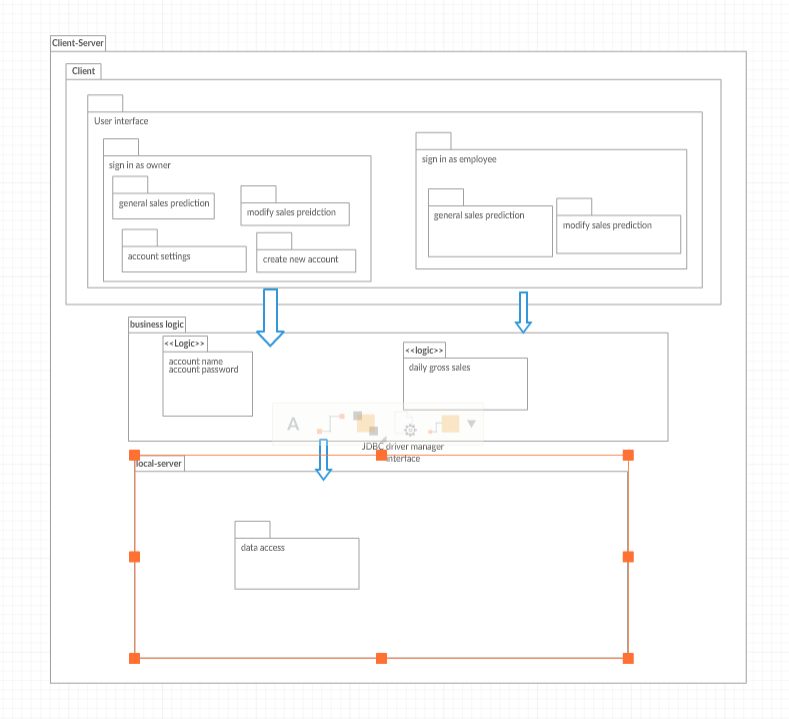
**Software architecture**



The architecture pattern we chose to implement is a Client-Server design. As a client, you will have the ability to login as an account with either manager or employee authorization. Due to the different authorization levels, a user may have restricted access to certain data. A user with an employee authorization level will be brought to a subtly different home screen with fewer options.

As a user with manager authorization, there will be two more options on the home screen, *Account Settings* and *Create New Account*.

When any user selects the *Generate Sales Prediction* option, a new window will pop up allowing the user to input a certain date range and, according to the sales data previously inputted in the database, predict the average gross sales for that future date.

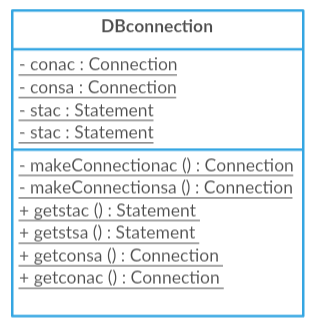
When any user selects the *Input Sales Data* option, a window pops up prompting the user to select a date and is given the option to input a value for sales data. The user can then submit this data by pressing the *Upload* button.

A user with manager authorization can choose *Account Settings*. After the user selects the *Load* button, the usernames, passwords and associated positions of those accounts from the database are then loaded into the table on this screen. The user can then edit an account’s credentials and position after inputting new data and then clicking *Update*.

A user with manager authorization can click *Create New Account* and will then be prompted to enter an account’s username, password and choose a role. This new user’s information will then be submitted into the account database.

**Design pattern:**

**Creational pattern:** Inabstract factory, builder design pattern, and factory method, these provide an interface for creating families of related dependent objects, but we do not have any interface to use. In prototype pattern, we need to have an abstract class, but we do not have any abstract class so we do not use that pattern either. In **Singleton Pattern**, we want to ensure that a class has one globally accessible instance and exactly one object is needed to coordinate actions across the system. The instance of st and con are private and provide a public static method that returns a reference to the instance **public static String getNumDay.** The following is the reference**.**



public class DbManager {

**static private Statement st = DBconnection.getstsa();**

**static private Connection con = DBconnection.getconsa();**

**private static boolean UpdateOneDa**y(String month, float highorder, String day, float loworder) {

try {

String query1 = "INSERT INTO " + month

+ " SET DayOfMonth=?,AvgGrossSales=? ON DUPLICATE KEY UPDATE AvgGrossSales = VALUES(AvgGrossSales)";

PreparedStatement ps = con.prepareStatement(query1);

ps.setString(1, day);

ps.setFloat(2, ((highorder) \* 3 + loworder) / 4);

//System.out.println(month + " " + day + " " + ((highorder) \* 3 + loworder) / 4);

ps.executeUpdate();

return true;

} catch (SQLException e) {

System.out.println(e);

return false;

}

}

**public static String getNumDay**(String date, String day1) {

String day = formatday(day1);

String[] dayof = date.split("/");

day = dayofmonth(Integer.parseInt(dayof[1])) + day;

return day;

}