EECS 2311 - TEAM 7 - PlatePlan

Iteration 3 Wiki

Team Members

Name | Student ID
Farah Madkour | 219913219
Meem Morshed | 219476142
Ricky Nguyen | 219461201
Andrew Nong | 219661537
Pouya Sameni | 216491623

ITR3 User Stories

(A description of each user story)
Story 1 (Farah Madkour) - Working on the UI and objects such as Order.java and other service functions

Story 2 (Ricky Nguyen) - Changes to order service implementation and adding to the database

Story 3 (Andrew Nong) - This story will be able to produce a receipt with the tables orders. Ultimately, taking off the work load from the waitress through an implementation which can effortlessly finalize a customers experience.

Story 4 (Pouya Sameni) - This story focuses on providing the business side a unique feature which allows them to analytically track spending and projected earnings per day based on previously collected data. This feature will allow a business to quickly respond to projections accordingly which ultimately leads to a successful business.

Story 5 (Meem Morshed) - In a similar way to story 4, this one provides a unique feature to customers by suggesting popular menu items based on previous data on orders. Not only is this helpful for the customers side but it also assists with the business side in knowing popular items.

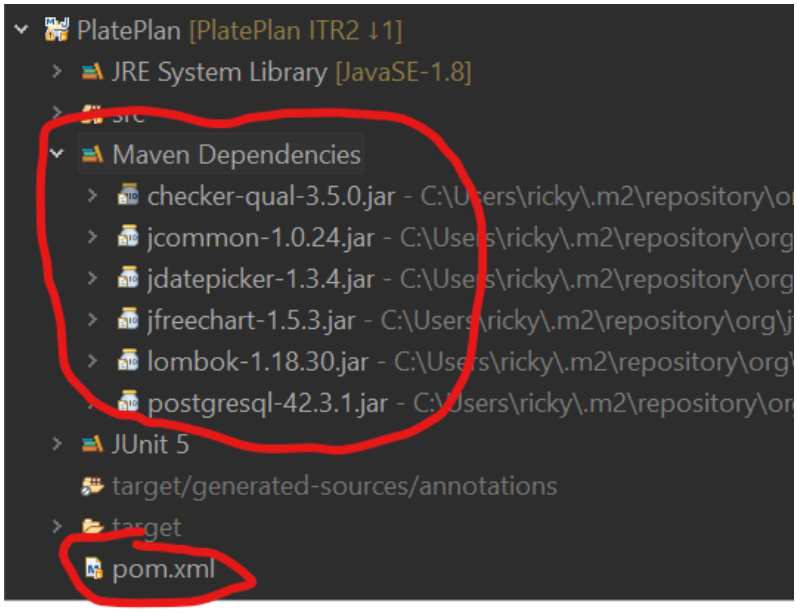
Technical Setup

Setup Video

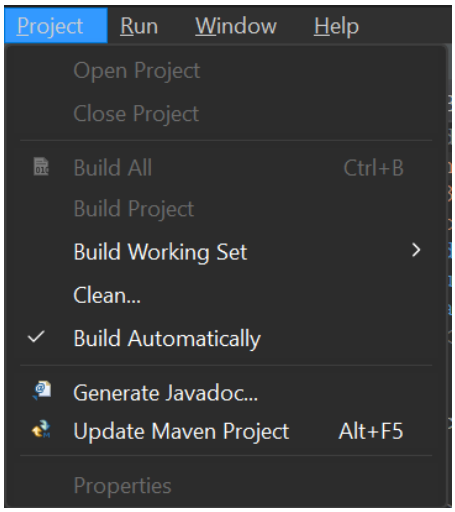
PlatePlan - Project Setup.mkv

Project Setup

To set up the project, we first want to import the maven project “PlatePlan”. Then, you want to make sure all dependencies are available



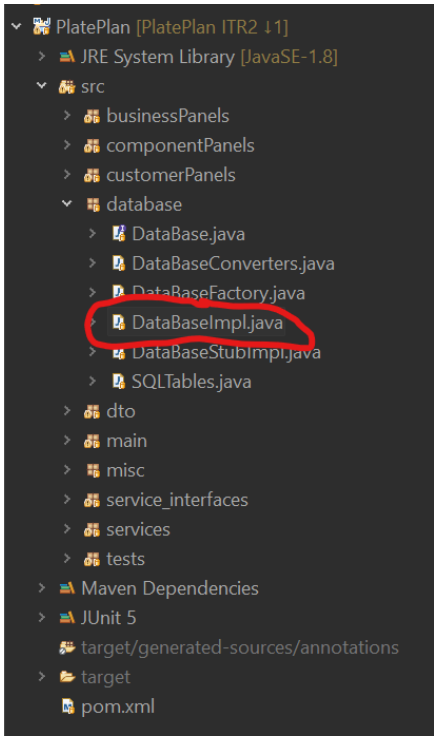
This can be done by simply adding the .jar files manually. An alternative would be to build the project through Maven and do the following:
First, clone the repository. Then, once the files are in your developing tool clean the project.



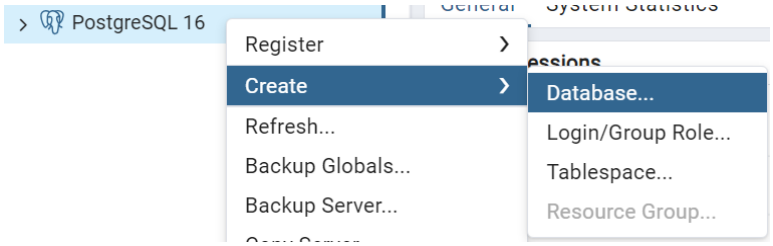
Database Setup

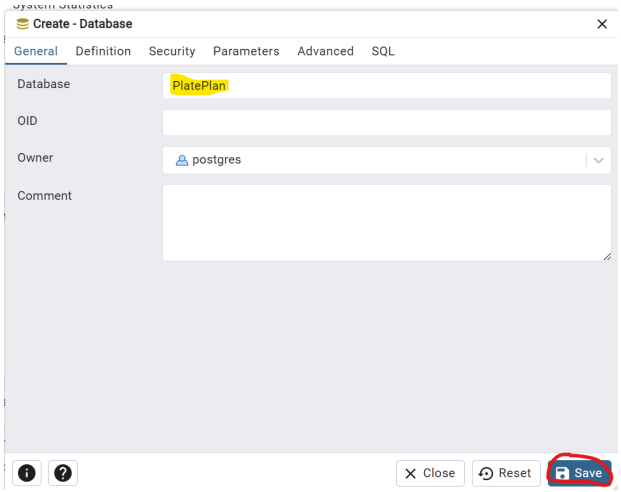
When setting up the database, please note that the username and password of the database can be found in DataBaseImpl.java (/PlatePlan/src/database/DataBaseImpl.java).

```
// Private constructor for singleton pattern
private DataBaseImpl() {
    // Set up the database connection here
    try {
        // Example connection setup
        String url = "jdbc:postgresql://localhost:5432/PlatePlan";
        String user = "postgres";
        String password = "admin";
```

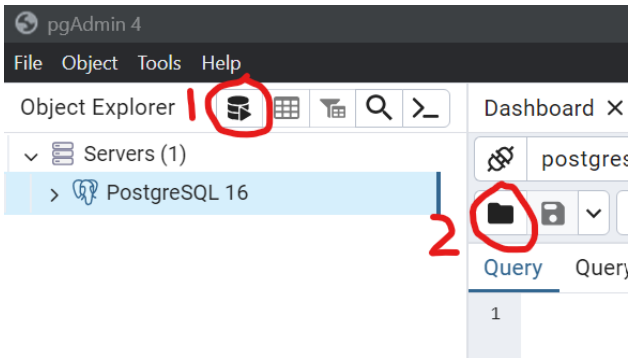


When adding the database, right-click on PostgreSQL and create a new database. Please note that the database should be named “PlatePlan”.

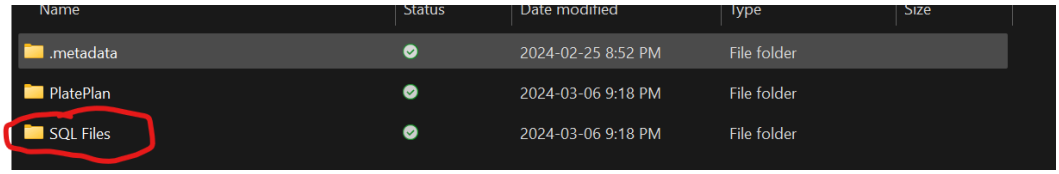




Click the Query Tool button. This will open up a window for you to upload the PlatePlanBackup.sql file.



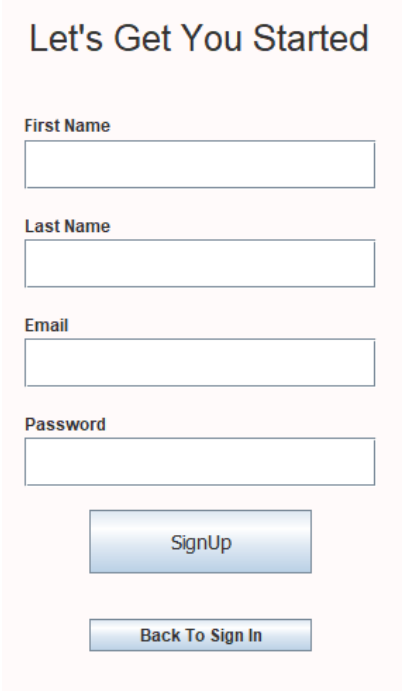
Can't find the PlatePlanBackup.sql file? It will be located under the address (/PlatePlan/SQL Files/PlatePlanBackup.sql).

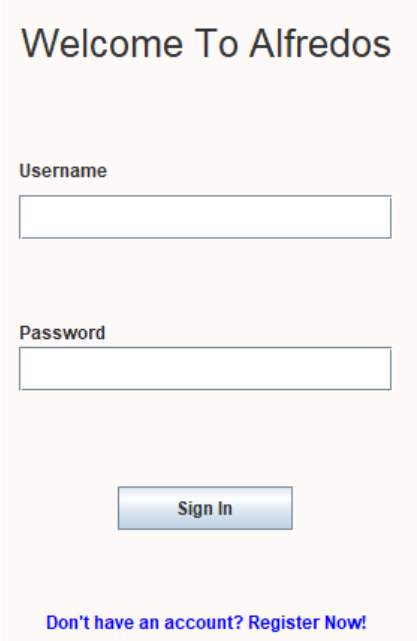


Finally, if required, update the username and password for Postgres as needed in the DataBaseImpl.java previously mentioned.

Accounts

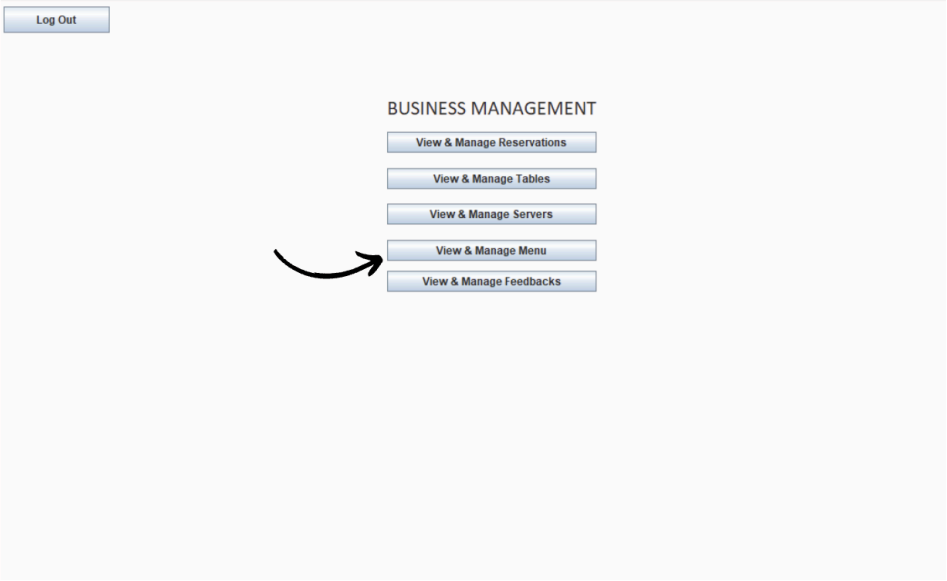
There are two types of accounts one can log in with. On the business side, you can log in utilizing the business username and password. **They are *alfredo* and *password* respectively.**
For customers, they can register through the registering panel. Once finished, they can enter their username and password to sign in.



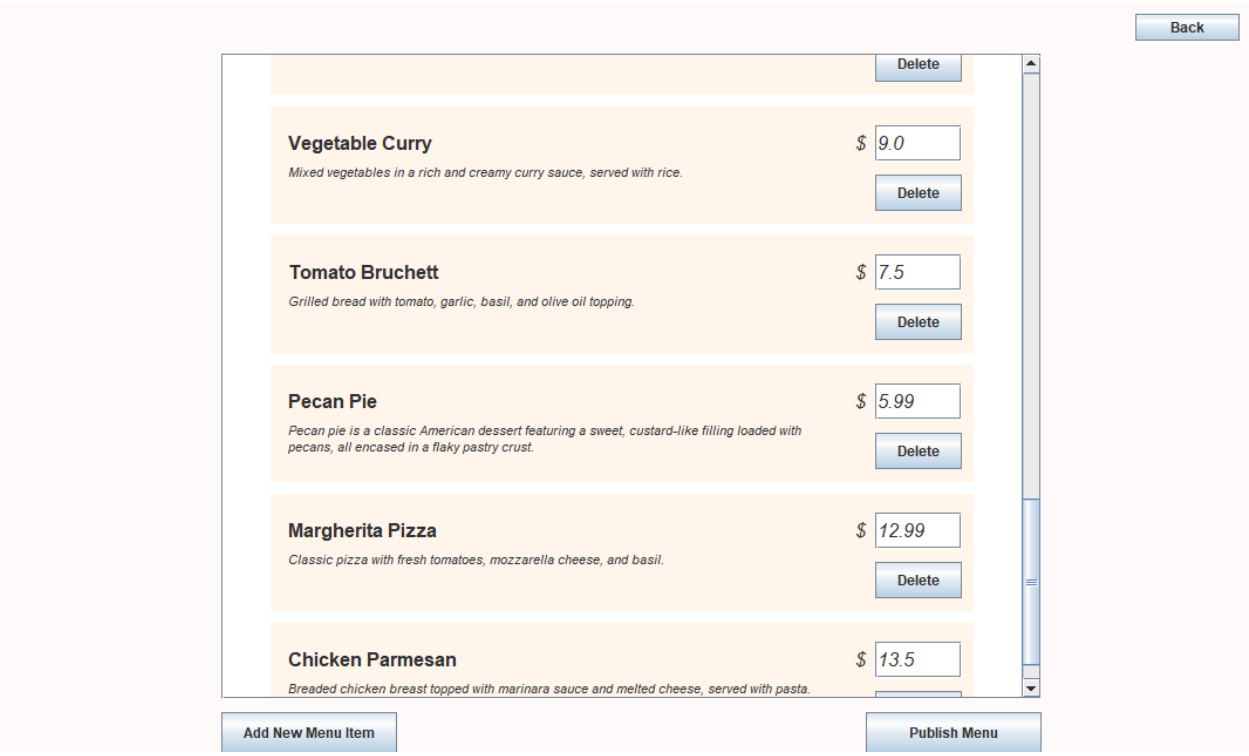


Use Cases (Add on to this with new use cases)

Using the menu as business

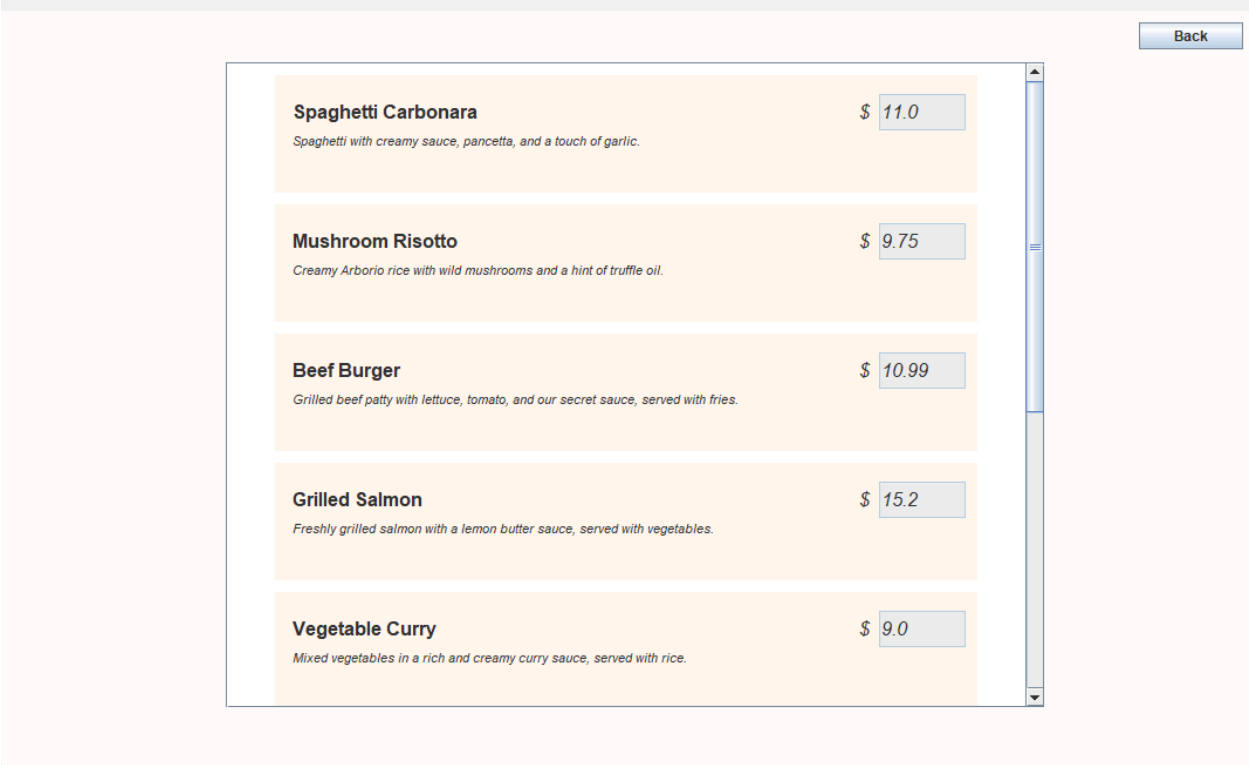


After logging in with the business account details, you have the option to use the menu from the business side.



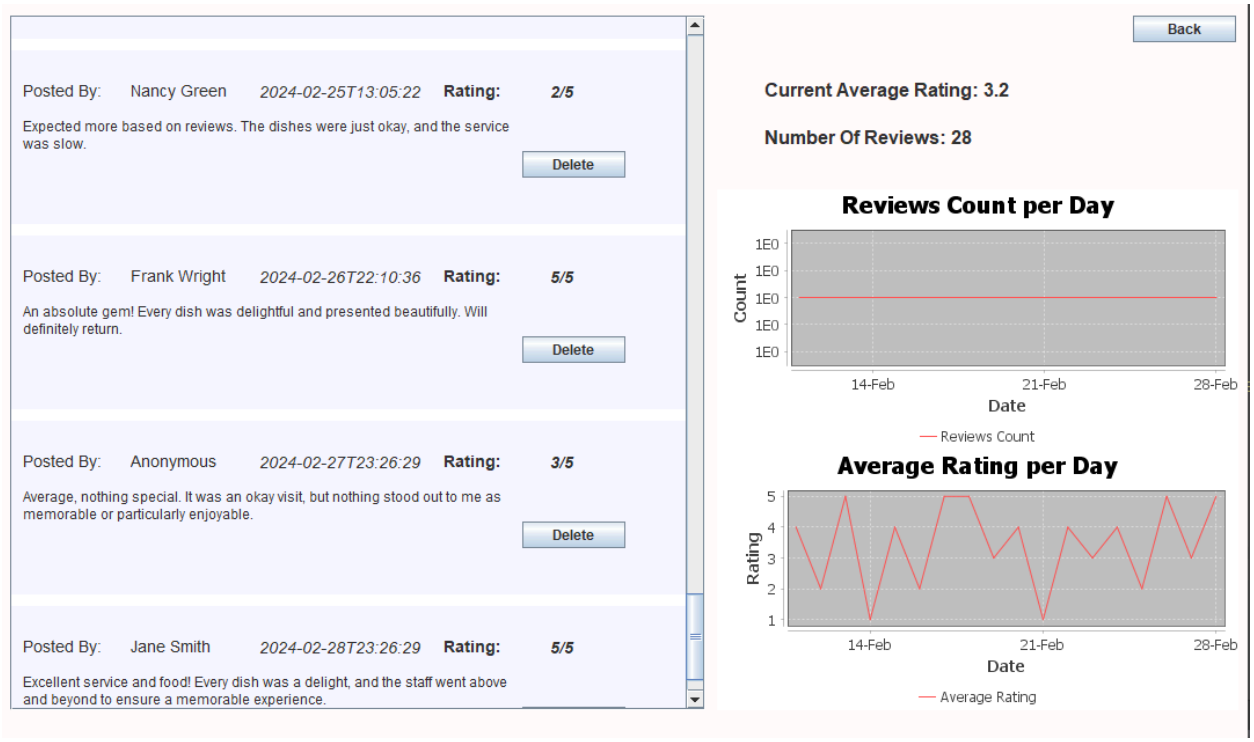
From here, you have the option to delete current menu items, or change the price of them. There are also buttons that allow the business to add new menu items. Once you're done, publish the menu.

Using the menu as customer



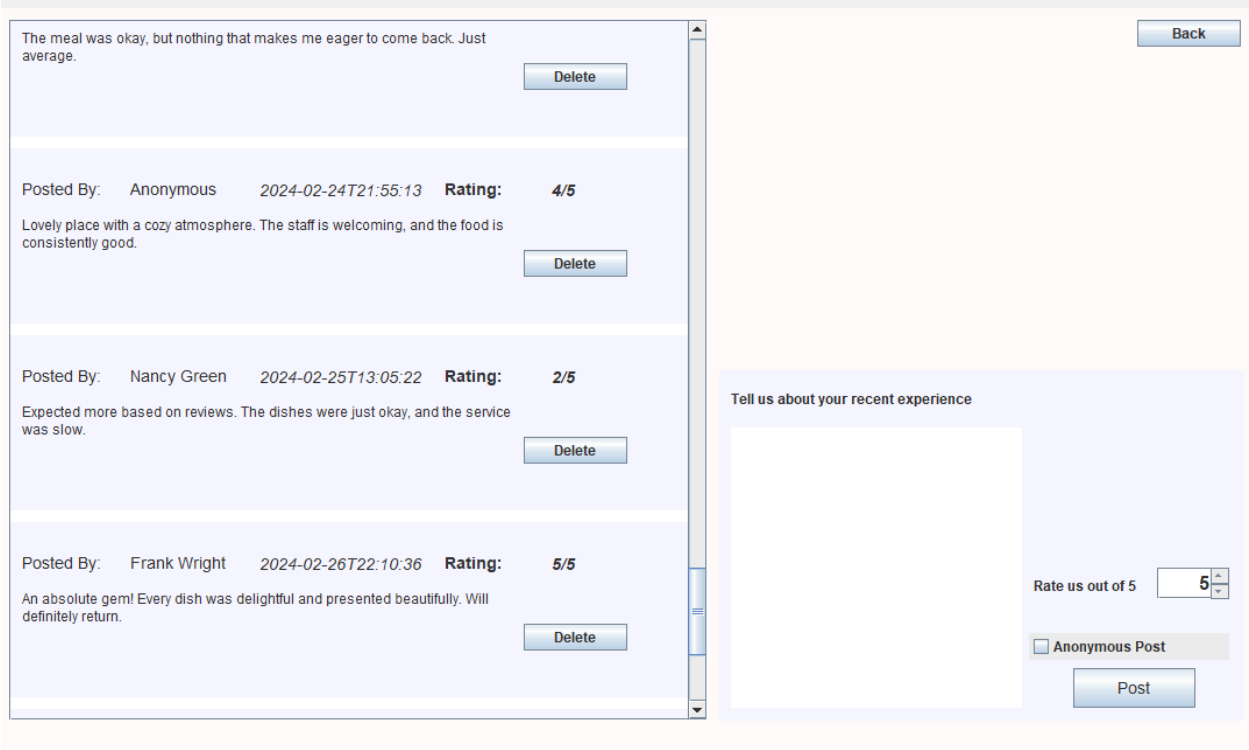
When using the menu as a customer, the user can view all of the available options, their descriptions, and their prices.

Using feedback as business



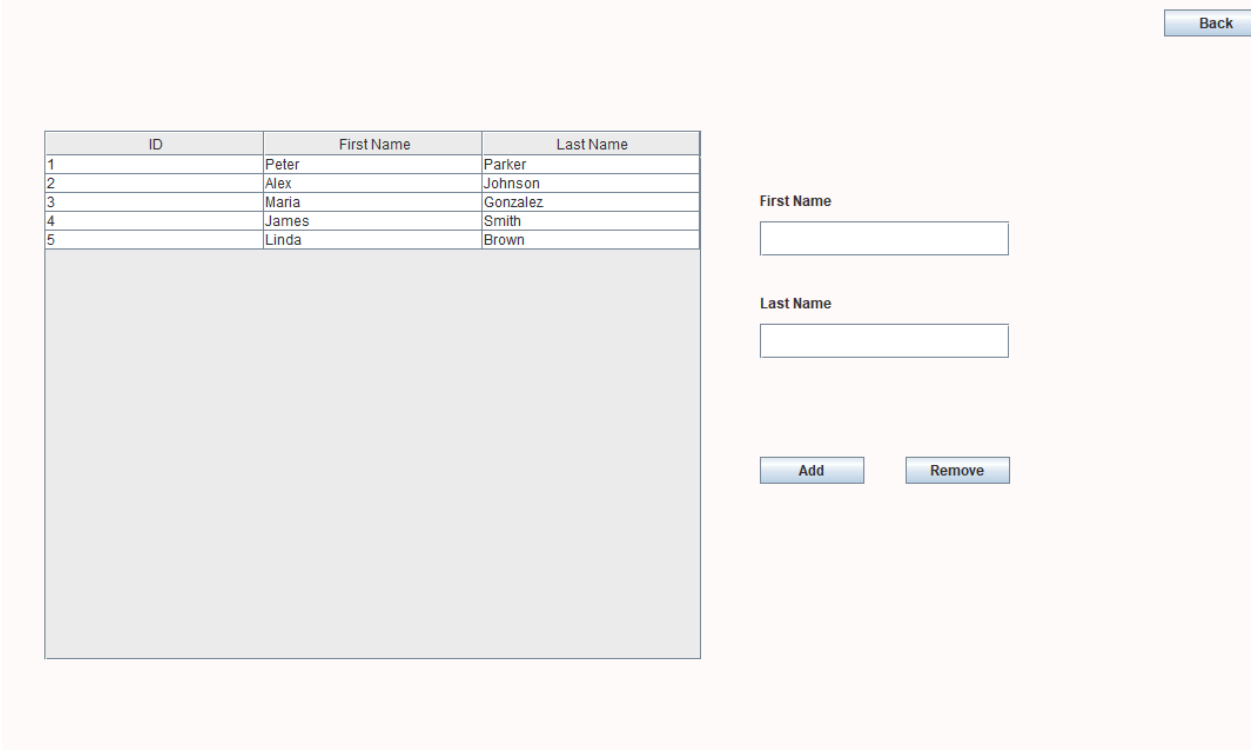
When using feedback as the business, you can view all of the customer reviews and ratings of their experience at the restaurant, with the option to delete any comments. On the right of the screen, there are two charts, one showing the review count per day, and the other showing the average rating per day.

Using feedback as customer



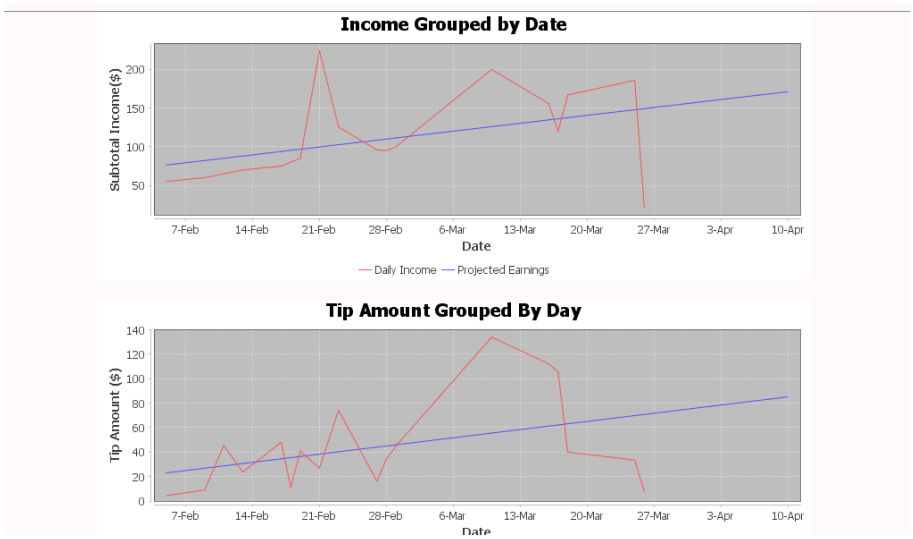
When using the feedback feature as a user, you have the option to view other peoples comments and ratings about the restaurant, as well as post your own review with a rating out of 5, with the option to post anonymously as well.

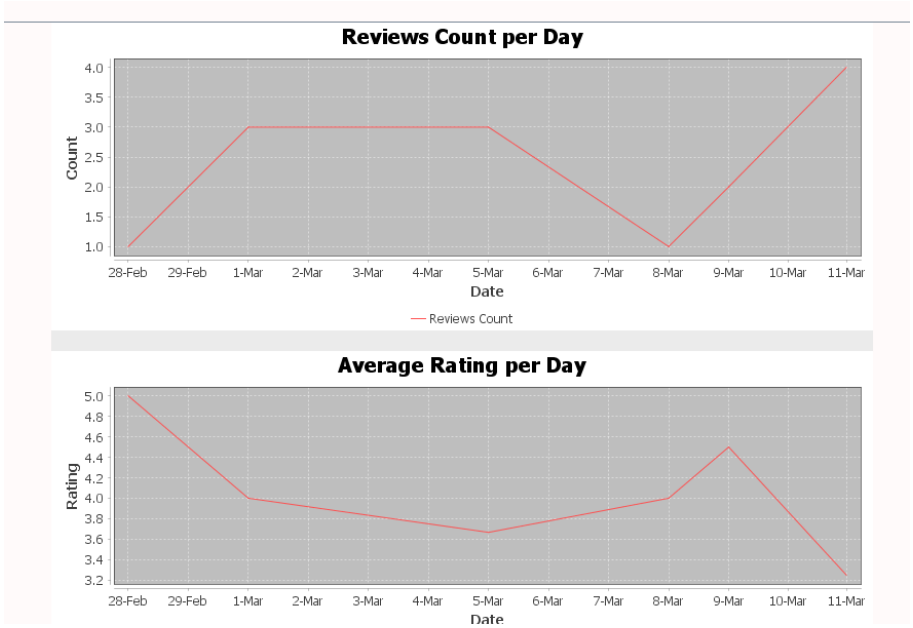
Using Server screen as business



When using the server screen from the business POV, you can view all of the current servers along with their IDs. Apart from that you can also add new servers, and remove current servers.

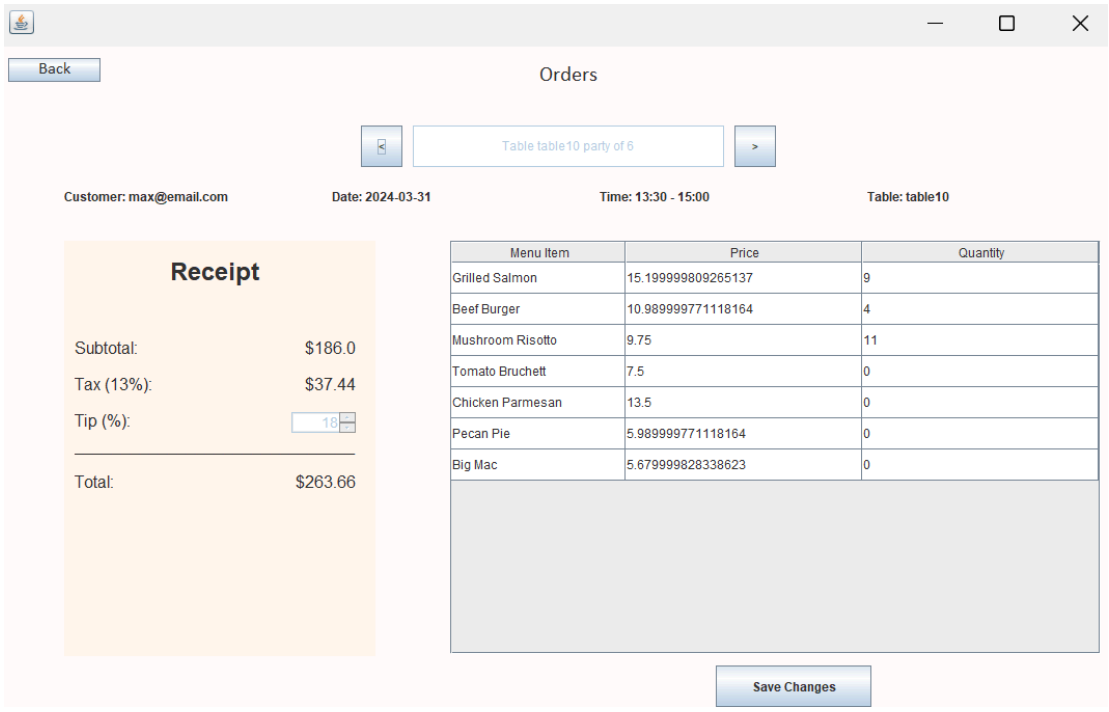
Using analytics on business





With this page, the business side can view a variety of statistics such as income, tips, review counts, and average ratings. In addition to this, this panel also provides projections of income and tips in the future.

Using the orders page on business



This page allows the business side to keep track of orders from each reservation and generate a receipt of the total including tip

Unit Testing

In our quality assurance process, we implemented unit tests to scrutinize both the services and business logic layers of our application. This testing approach utilized stub databases, facilitated by a toggle within our database factory settings. By adjusting this toggle to a "development" mode, our tests bypassed direct connections to PostgreSQL, instead leveraging these stub databases to simulate database interactions. This method enabled us to achieve an impressive 95% average coverage across all pertinent files, thoroughly examining both standard functionalities and potential edge cases to confirm the reliability and correctness of our codebase.

For integration testing, the database factory setting was switched to "production" mode. This adjustment allowed for focused testing on the actual database implementation methods. The primary objective of these integration tests was to verify seamless interaction between our application and the PostgreSQL database, ensuring no discrepancies in database behavior or issues with data integrity. Particular attention was paid to the functionality of converters responsible for transforming PostgreSQL query results into Java objects, validating their accuracy and effectiveness in a production-like environment.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
PlatePlan	35.1 %	7,035	13,018	20,053
src	35.1 %	7,035	13,018	20,053
main	50.6 %	79	77	156
services	89.9 %	1,276	144	1,420
MenuServiceImpl.java	90.8 %	59	6	65
ServerServiceImpl.java	97.6 %	83	2	85
GraphGenerators.java	0.0 %	0	102	102
FeedbackServiceImpl.java	96.9 %	127	4	131
AccountsServiceImpl.java	100.0 %	142	0	142
ReservationServiceImpl.java	90.4 %	160	17	177
TablesServiceImpl.java	98.8 %	322	4	326
OrdersServiceImpl.java	97.7 %	383	9	392
tests	99.0 %	2,043	20	2,063
ServerServiceTest.java	100.0 %	82	0	82
AccountServiceTest.java	100.0 %	140	0	140
MenuServiceTest.java	100.0 %	141	0	141
FeedbackServiceTest.java	100.0 %	173	0	173
TableServiceTest.java	100.0 %	323	0	323
OrdersServiceTest.java	100.0 %	366	0	366
ReservationServiceTest.java	98.8 %	412	5	417
DataBaseIntegrationTests.java	96.4 %	406	15	421
database	88.7 %	2,120	269	2,389
SQLTables.java	0.0 %	0	3	3
DataBaseFactory.java	87.0 %	20	3	23
StubDataBaseRecords.java	100.0 %	311	0	311
DataBaseConverters.java	87.1 %	434	64	498
DataBaseStubImpl.java	92.0 %	621	54	675
DataBaseImpl.java	83.5 %	734	145	879
componentPanels	0.0 %	0	2,471	2,471
dto	53.2 %	1,517	1,335	2,852
customerPanels	0.0 %	0	2,888	2,888
businessPanels	0.0 %	0	5,814	5,814

Core components and all features currently available

Login and Registration Account System

The program PlatePlan has an account system that can store account information including passwords, emails, names, and usernames. The system will then only allow users with the right credentials to login to access the rest of the user features. If the user does not have an account setup then they can easily register by inputting a first name, last name, email, and password.

Reservation System

The system allows the customer to set a reservation, to reserve a table the following information is needed. The Date of reservation, number of people, the available timeslot and any special notes and/or allergies. On the business side the business can see any upcoming reservations which will show the date and time slot of the day, but also automatically assign a server and table to the customer.

Rating and Analytics

The customer receives the average rating of the restaurant, and can provide ratings as a message and stars and can post the rating anonymously as a public rating, or displaying their name. On the business side of the reviews and rating system, the business can see all the ratings that have been posted and read the messages. There are also graphs to display information such as average rating per day as a graph and a value to see past and current trends, reviews per day to see how many reviews are posted in a graph to see past and current trends, and displays the total amount of reviews of all time. In addition to this, the business side can view data such as income and tip projections which can be useful.

Menu

For the customer the menu displays the items the restaurant serves, alongside with a description of the food item, and the pricing of said item. On the business side the menu can be edited such as removing food items, changing the price of the items and adding new items.

Table and Server System

The table and server system is used by the business side and allows the business to define tables that will be taken by a server. The table is defined by the capacity, ID and the waitstaff that will be attending the table. Tables can be added, removed and combined to allow for different seating arrangements. Servers are also defined in the system with first name and last name. Servers can also be removed and added as seen fit.

Orders

The business side view allows the user to check the order history of each table. In addition to this, the user can produce a receipt with all ordered items. This will reduce the workload for servers.

Design Level Refactoring

Refactor #1

In the third iteration of our project, a significant enhancement was made by introducing an interface for our Data Transfer Objects (DTOs). These DTOs, which include various system objects such as orders, reservations, customer accounts, and receipts, now follow a newly created interface. This improvement started with the creation of an interface named "QueryGenerator." This interface has two essential methods: "generateInsertQuery" and "generateUpdateQuery." All DTOs are now required to implement this interface, ensuring a standardized implementation of these methods. These methods are important for our database interaction, enabling the generation of insert and update queries.

This approach enhances scalability by eliminating the need for conditional statements to determine the relevant database table and class, as previously encountered. The implementation simplifies code maintenance and scalability by replacing extensive conditional logic with a straightforward method call to "generateInsertQuery" or "generateUpdateQuery," depending on the operation. This method dynamically identifies and executes the appropriate query for the given object, thereby streamlining database interactions.

The "QueryGenerator" interface includes methods for creating insert and update statements, as demonstrated in the "QueryGenerator.java" class with the following signatures:

```
public PreparedStatement generateInsertStatement(Connection conn,
List<String> columns);
public PreparedStatement generateUpdateStatement(Connection conn,
List<String> columns);
```

These changes are integrated into our database implementation, as shown in the "DataBaseImpl.java" class, illustrating a clear before-and-after comparison of our approach to database interactions, significantly contributing to the project's scalability and maintainability.

Before

```java
    @Override
    public boolean insertRecord(String tableName, Object object) {
        String sql = "INSERT INTO %s %s VALUES ";
        sql = String.format(sql, tableName, getColumnNamesString(tableName));

        PreparedStatement pstmt = null;

        if (tableName.equals(SQLTables.RESERVATION_TABLE)) {
            Reservation reservation = (Reservation) object;
            pstmt = reservation.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.TABLES_TABLE)) {
            Table table = (Table) object;
            pstmt = table.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.ACCOUNTS_TABLE)) {
            Customer customer = (Customer) object;
            pstmt = customer.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.SERVERS_TABLE)) {
            Server server = (Server) object;
            pstmt = server.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.MENU_TABLE)) {
            MenuItem menuItem = (MenuItem) object;
            pstmt = menuItem.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.FEEDBACKS_TABLE)) {
            Feedback feedback = (Feedback) object;
            pstmt = feedback.getSQLString(connection, sql);
        }

        System.out.println("Executing Command: " + pstmt.toString());
        try {
            if (pstmt != null && pstmt.executeUpdate() > 0) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

After

```java
    @Override
    public boolean insertRecord(String tableName, QueryGenerator object) {

        PreparedStatement pstmt = object.generateInsertStatement(connection, getColumnNamesList(tableName));

        System.out.println("Executing Command: " + pstmt.toString());
        try {
            if (pstmt != null && pstmt.executeUpdate() > 0) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

Refactor #2

The second major change we implemented was breaking down a large class into several smaller ones, specifically the ServiceUtil.java class. Initially, in our project, this class served multiple purposes, handling both server and table management tasks. At first, we thought it wasn't necessary to have separate classes for these tasks. However, by the second phase of our project, we realized that ServiceUtil was becoming too big and difficult to manage.

As a result, we decided to remove ServiceUtil completely and introduce two new classes: ServersService and TableService. The goal was to make the structure more organized by dividing responsibilities into more specific classes. This change made things much simpler to manage, almost like sorting items into their own compartments.

Another significant advantage of this approach was the improvement it brought to testing. Instead of dealing with a single, large JUnit test file, we were able to create individual test files for each class. This made the testing process much easier, as it eliminated the need to sift through a lot of code to locate the specific functions we needed to test.

As seen on the link below

https://github.com/Pouya-Sameni/PlatePlan/blob/ITR1/PlatePlan/src/misc/ServiceUtils.java

Our first iteration had a larger ServiceUtil class without a clear goal. This is now changed to the following two service interfaces.

Tables Service:
https://github.com/Pouya-Sameni/PlatePlan/blob/ITR3/PlatePlan/src/service_interfaces/TablesService.java

Servers Service:
https://github.com/Pouya-Sameni/PlatePlan/blob/ITR3/PlatePlan/src/service_interfaces/ServerService.java

Refactor #3

The third significant improvement we made was during iteration 2. We noticed that as our collection of objects and tables grew, we constantly had to create methods for deleting entries from these tables. However, all our tables had a similar structure with a unique ID column. So, we combined these delete methods into one. Now, in future iterations, we only need to pass the ID and table name to delete an entry. This streamlined our database interface and simplified our code, reducing it to a single method. The images below show the before and after of our interface and the code changes.

```java
@Override
public boolean deleteDataBaseEntry(String table, String id) {
    int affectedRows = 0;
    String sql = "DELETE FROM " + table + " WHERE id = ?;";

    // SQL command to delete rows with the specific ID
    if (table.equals(SQLTables.ACCOUNTS_TABLE)) {
        sql = "DELETE FROM " + table + " WHERE email = ?;";

    }

    try {
        // Set the ID in the prepared statement to avoid SQL injection
        PreparedStatement pstmt = connection.prepareStatement(sql);
        pstmt.setString(1, id);
        System.out.println("Delete Query Executed: " + pstmt.toString());
        // Execute the delete command
        affectedRows = pstmt.executeUpdate();
        System.out.println("Deleted " + affectedRows + " rows.");

    } catch (SQLException e) {
        System.out.println("Error occurred during delete operation: " + e.getMessage());
    }
    return affectedRows <= 0 ? false : true;

}
```

Additionally, we observed that when fetching data from the database, we repeatedly had to set certain fields for each object type. For instance, for objects like reservations. To streamline this process, we created a database converters class. This class has methods for converting database responses into Java objects. One method converts a single database response into a Java object, while the other converts query responses into lists of Java objects. This approach made it easier to convert PostgreSQL database responses into Java objects. See the implementation of our database converters below.

```java
19 public class DataBaseConverters {
20
21     public static List<Reservation> convertReservationList(ResultSet rs, Business business) {
22         List<Reservation> reservations = new ArrayList<>();
23         try {
24             while (rs.next()) {
25                 Reservation reservation = convertReservation(rs, business);
26                 if (reservation != null) {
27                     reservations.add(reservation);
28                 }
29             }
30         } catch (SQLException e) {
31             // TODO Auto-generated catch block
32             e.printStackTrace();
33         }
34         return reservations;
35     }
36
37     public static Reservation convertReservation(ResultSet rs, Business business) {
38         try {
39             Reservation reservation = new Reservation();
40
41             reservation.setId(rs.getString("id"));
42             reservation.setCustomerId(rs.getString("customer_id"));
43             reservation.setDate(rs.getDate("date").toLocalDate());
44             reservation.setTime(new TimeSlot(rs.getTime("time").toLocalTime(),
45                     rs.getTime("time").toLocalTime().plusMinutes(business.getReservationSlots())));
46             reservation.setSpecialNotes(rs.getString("special_notes"));
47             reservation.setTableId(rs.getString("table_id"));
48             reservation.setPartySize(rs.getInt("party_size"));
49             reservation.setServerId(rs.getString("server"));
50             return reservation;
51
52         } catch (Exception e) {
53             e.printStackTrace();
54         }
55         return null;
```

Proposed Designs

Sign Up For Business

This feature allows the business user to sign in

Customize Restaurant Settings

This feature allows the business to set up opening/closing times and seating options

Accessing Reservations

This feature allows the business to view reservations set by customers

Managing Reservation

This feature allows the business to manage reservations by deleting them on behalf of the customer

Customer Sign Up

This feature allows the customer to create and log in to an account

Dining Availability

This feature allows the customer to check open reservations

Manage Reservations

This feature allows the customer to set up, choose, and cancel reservations with the restaurant

Completed Designs

Sign Up For Business

This feature allows the business user to sign in

Customize Restaurant Settings

This feature allows the business to set up opening/closing times and seating options

Accessing Reservations

This feature allows the business to view reservations set by customers

Managing Reservation

This feature allows the business to manage reservations by deleting them on behalf of the customer

Customer Sign Up

This feature allows the customer to create and log in to an account

Dining Availability

This feature allows the customer to check open reservations

Manage Reservations

This feature allows the customer to set up, choose, and cancel reservations with the restaurant

Automatic Receipt Generator

This features allows for automated generation of receipts

Feedback System

This feature allows for customers to give feedback on a scale of 1-5 and options to post anonymously.

Analytics of Business

This feature allows the business to see many analytics regarding the business this includes. Income grouped by date, tip amount grouped by day, reviews per day, and average rating per day.

Menu

Allows customers to view the menu of the restaurant, allows the business to edit prices, change menu items.