

Review Eat

HND Graded Unit Project

Implementation

Yogesh Parajuli

# Development Report

## Changes to the Original Plan

1. User class has fewer attributes.

* During the development, I realized that the user class is holding the information that might not be useful for us. Hence, to reduce the data clutter, the user class was simplified to have fewer attributes.

1. Similarly, the address class has not been implemented. Instead, address is implemented as one of the attributes of Restaurant class and input validation has been used.
2. Likewise, the id for restaurant is not assigned by the application layer. Instead, the database assigns the id for the restaurants.
3. Login class and Review Eat class were eliminated.
4. Since Login class and Review Eat were eliminated, the function decided for those class were handled by either the database class (Example: User validation) and by the controller classes (Example checking if the restaurant has already been added).
5. The User Interface is slightly different.

## General Overview of the Requirements

Review Eat is a software that would allow users to add reviews on the restaurants that have been added by the admin. Users can register to the system, log into the system, view all restaurants that have been added by the admin and add a review onto them. Here are the listed initially functional requirements of the software and brief explanation on if they have been achieved:

* Allow user to create an account with username and password. User can choose their Username, but it must be unique, and the password will have specific criteria.
* The software allows user to register to the system. The registration process includes several validations. For example, passwords must be 6-20 characters long, and they can only contain letters and numbers.
* Allow user to reset the password if necessary.
* The software does not feature password reset functionality, instead, it allows user to update their password if they can verify with their username and password.
* The software should store the user’s information in a database.
* The software uses the SQLite database to store user’s information.
* User can update their information and delete their account if they want.
* Users can update their password if necessary, and they can also delete their account from the system if they can verify their username and password.
* Allow access to the software with valid id and password.
* User or Admin can log in to the system using a valid username and password.
* Admin will have predefined Username and password.
* Admin has been given a fixed username and password in the application layer. Also, a regular user is prohibited of using “admin” as their username which is reserved for the admin of the system.
* Admin will be able to add restaurant details in the system.
* Admin is logged into an admin view page where they can add a restaurant to the system.
* Admin can update restaurant details or delete them.
* Admin can update restaurant or delete them if necessary.
* Admin also has the privilege to delete a user’s account.
* This feature has also been met.
* User logged in the system will see all the restaurant added by the admin. The restaurant can be ranked according to their review rating or by alphabetical order.
* User can see all the restaurants added by the admin. They have their average rating displayed on the side. The users have a choice of sorting restaurant based on rating or alphabetical order.
* Allow user to add a review for a restaurant. The review is displayed to everyone, but it can be edited or deleted by the person who added the review. There will be a five-star rating system that a user can use to rate the restaurant.
* Users can add review onto the restaurant. The average rating of the restaurants is based on the reviews added by the users. The reviews can be seen by the users, but it cannot be edited or deleted by the users. The reviews are deleted when a user is deleted, or a restaurant is deleted. The rating system has been implemented slightly different. User can rate the restaurant based on the scale of 1 to 10.
* A review can only be added if the user has been to the restaurant in the past thirty days.
* Not implemented
* User should be able to search for a restaurant using a search bar. User can filter results based on categories of restaurants.
* This feature has been implemented.
* User should be able to log out of the system.
* Users and admin can log out of the system.

## Use of Unfamilier Libraries/Concepts

The software is developed in java using Object Oriented Programming practices. There are several classes and objects that are used for the program. As mentioned in the planning stage, I am using Multi-Tier architecture for development. The multi- layer is made up of Presentation layer, which is the user interface of the project, Application layer, which is part which consists of inner workings of the program and Data layer, which is basically a database.

Here is the list of all the libraries and concepts that is used for Review Eat software.

* JavaFX Library
* Java FX Library is used to create the Graphical User Interface in our program.
* Lambda Expressions
* Lambda Expression is used to handle events. This is a very new concept for me. I have made use of lambda expression for handling events such as when letters are typed in a search box and others.
* SQL Library
* SQL Library has been used for CRUD operations in our program.
* Design Patterns
  + Singleton Design Pattern
  + Strategy Design Pattern
  + Iterator Design Pattern

## Coding the domain

This section of the documentation contains the code list for the software.

### Application Layer

#### Admin.java

1. /\*\*
2. \*
3. \*/
4. package application;
6. /\*\*
7. \* This is a sub class of SoftwareUsers.
8. \*
9. \* Uses static attributes for admin Username and admin Password
10. \* so that they are not changable.
11. \* Represents user with admin privilege
12. \*
13. \* @author Yogesh Parajuli
14. \*
15. \*/


19. public class Admin extends SoftwareUsers{
20. private static String adminUsername = "admin";
21. private static String adminPassword = "administrator";

24. /\*\*
25. \* Constructor for admin.
26. \* We set the privilege to Privileged as admin has admin privileges.
27. \* @throws InputValidationException
28. \*
29. \*/
30. public Admin() throws InputValidationException {
31. super(adminUsername, adminPassword);
32. privilege = new Privileged();
33. }

36. /\*\*
37. \* Getter method for admin username
38. \* @return adminUsername
39. \*/
40. public static String getAdminUsername() {
41. return adminUsername;
42. }

45. /\*\*
46. \* Getter method for admin password.
47. \* @return admin Password
48. \*/
49. public static String getAdminPassword() {
50. return adminPassword;
51. }

54. }

#### InputValidationException.java

1. /\*\*
2. \*
3. \*/
4. package application;
6. /\*\*
7. \* Used by SoftwareUsers class, Restaurant class and Review class to validate input
8. \*
9. \* @author Yogesh Parajuli
10. \*
11. \*/
12. public class InputValidationException extends Exception{
14. }

#### Main.java

1. package application;
3. import javafx.application.Application;
4. import javafx.stage.Stage;
5. import javafx.scene.Scene;
6. import javafx.scene.layout.AnchorPane;
7. import javafx.fxml.FXMLLoader;

10. /\*\*
11. \* @author Yogesh Parajuli
12. \* Graded Unit Project
13. \* Class containing main method.
14. \* It loads loginFXML file
15. \*
16. \*/
17. public class Main extends Application {
18. @Override
19. public void start(Stage primaryStage) {
20. try {
21. AnchorPane root = (AnchorPane)FXMLLoader.load(getClass().getResource("/GUI/loginFXML.fxml"));
22. Scene scene = new Scene(root,312,413);
23. primaryStage.setTitle("Review Eat");
24. primaryStage.setScene(scene);
25. primaryStage.show();
26. } catch(Exception e) {
27. e.printStackTrace();
28. }
29. }

32. /\*\*
33. \* @param args
34. \* Public Main method to launch the application
35. \*/
36. public static void main(String[] args) {
37. launch(args);
38. }
39. }

#### Restaurant.java

1. /\*\*
2. \*
3. \*/
4. package application;

7. import java.util.HashMap;
8. import java.util.Iterator;

11. /\*\*
12. \*
13. \* This class represents the Restaurant
14. \* Implements Comparable Interface to compare restaurants based on average ratings
15. \* @author Yogesh Parajuli
16. \*
17. \*/
18. public class Restaurant implements Comparable<Restaurant>, Iterable<Review>{
20. private int id;
21. private String name;
22. private String restaurantDescription;
23. private String cuisine;
24. private String contact;
25. private String address;
26. private double averageRating = 0;
27. private HashMap<String, Review> reviews = new HashMap<String, Review>();


31. /\*\*
32. \* Constructor for restaurant.
33. \* This is used when an restaurant object is created
34. \* @param name
35. \* @param restaurantDescription
36. \* @param cuisine
37. \* @param contact
38. \* @param address
39. \* @throws InputValidationException
40. \*/
41. public Restaurant(String name, String restaurantDescription, String cuisine, String contact, String address) throws InputValidationException {
42. setAddress(address);
43. setContact(contact);
44. setCuisine(cuisine);
45. setName(name);
46. setRestaurantDescription(restaurantDescription);
48. }
49. /\*\*
50. \* Empty Constructor
51. \*
52. \*/
53. public Restaurant() {}
55. /\*\*
56. \* Another constructor that takes in restID
57. \* This will be used during data retireval from the database
58. \*
59. \* @param restID
60. \* @param name
61. \* @param restaurantDescription
62. \* @param cuisine
63. \* @param contact
64. \* @param address
65. \* @throws InputValidationException
66. \*/
67. public Restaurant(int restID, String name, String restaurantDescription, String cuisine, String contact, String address) {
68. this.id = restID;
69. // setAddress(address);
70. // setContact(contact);
71. // setCuisine(cuisine);
72. // setName(name);
73. // setRestaurantDescription(restaurantDescription);
74. this.address = address;
75. this.name = name;
76. this.contact = contact;
77. this.cuisine = cuisine;
78. this.restaurantDescription = restaurantDescription;
80. }
82. /\*\*
83. \* Getter for reviews
84. \* @return reviews
85. \*/
86. public HashMap<String, Review> getReviews() {
87. return reviews;
88. }
90. /\*\*
91. \* Setter for reviews
92. \* Setter also calls setAverageRating method
93. \* because we want the average rating to change when review is added
94. \* in this case when a collection of reviews are input
95. \* @param reviews
96. \*/
97. public void setReviews(HashMap<String, Review>reviews) {
98. this.reviews = reviews;
99. setAverageRating();
100. }



105. /\*\*
106. \* Gettter method for average rating
107. \* @return averageRating
108. \*/
109. public double getAverageRating() {
110. return averageRating;
111. }
113. /\*\*
114. \* Setter for averageRating
115. \*
116. \*/
117. public void setAverageRating() {
118. int totalRating= 0;
119. int reviewCount = reviews.values().size();
120. Iterator<Review> iter = iterator();
121. if(reviewCount > 0) {
122. while(iter.hasNext()) {
124. totalRating = totalRating + iter.next().getRating();
125. }
126. this.averageRating = totalRating/reviewCount;
127. }
128. else {
129. this.averageRating = 0;
130. }
131. }

134. /\*\*
135. \* Getter method for ID
136. \* @return id
137. \*/
138. public int getId() {
139. return id;
140. }

143. /\*\*
144. \* Setter method for ID
145. \* The id is stored as integer in the database that automatically increases
146. \* @param id
147. \*/
148. public void setId(int id) {
150. this.id = id;
151. }

154. /\*\*
155. \* Getter method for name
156. \* @return name
157. \*/
158. public String getName() {
159. return name;
160. }

163. /\*\*
164. \* Setter method for name
165. \* @param name
166. \* @throws InputValidationException
167. \*/
168. public void setName(String name) throws InputValidationException {
169. if(name.matches("^[a-zA-Z ]{4,30}$")) {
170. this.name = name;}
171. else {
172. throw new InputValidationException();
173. }
174. }

177. /\*\*
178. \* Getter method for restaurantDescription
179. \* @return restaruant Description
180. \*/
181. public String getRestaurantDescription() {
182. return restaurantDescription;
183. }

186. /\*\*
187. \* Setter method for restaurant Description
188. \* @param restaurantDescription
189. \* @throws InputValidationException
190. \*/
191. public void setRestaurantDescription(String restaurantDescription) throws InputValidationException {
192. if(restaurantDescription.matches("\\p{ASCII}{4,100}")) {
193. this.restaurantDescription = restaurantDescription;
194. }
195. else {
196. throw new InputValidationException();
197. }
198. }

201. /\*\*
202. \* Getter method for Cuisine
203. \* @return cuisine
204. \*/
205. public String getCuisine() {
206. return cuisine;
207. }

210. /\*\*
211. \* Setter method for cuisine
212. \* @param cuisine
213. \* @throws InputValidationException
214. \*/
215. public void setCuisine(String cuisine) throws InputValidationException {
216. if(cuisine.matches("^[a-zA-Z]{3,15}$")) {
217. this.cuisine = cuisine;
218. }
219. else {
220. throw new InputValidationException();
221. }
223. }

226. /\*\*
227. \* Getter method for contacts
228. \* @return contact
229. \*/
230. public String getContact() {
231. return contact;
232. }

235. /\*\*
236. \* Setter method for contacts
237. \* @param contact
238. \* @throws InputValidationException
239. \*/
240. public void setContact(String contact) throws InputValidationException {
241. if(contact.matches("\\d{11}")) {
242. this.contact = contact;
243. }
244. else {
245. throw new InputValidationException();
246. }
247. }

250. /\*\*
251. \* Getter method for address
252. \* @return address
253. \*/
254. public String getAddress() {
255. return address;
256. }

259. /\*\*
260. \* Setter method for address
261. \* The address should have digits followed by a space and string characters
262. \* @param address
263. \* @throws InputValidationException
264. \*/
265. public void setAddress(String address) throws InputValidationException {
266. if(address.matches("^(\\d+)\\s(\\w+)\\s(\\w+)$")) {
267. this.address = address;
268. }
269. else {
270. throw new InputValidationException();
271. }
273. }


277. /\*\*
278. \* Add review when username and review are provided
279. \* @param username
280. \* @param r
281. \* set average rating is called so that the average rating would change dynamically as the review is added.
282. \*/
283. public void addReview( String username, Review r) {
284. reviews.put(username,r);
285. //updateAverageRating(getAverageRating());;
286. setAverageRating();
287. }


291. /\*\*
292. \*
293. \* Add review when user is provided and description and rating is provided for the review.
294. \* @param username
295. \* @param description
296. \* @param rating
297. \* @throws InputValidationException
298. \*
299. \* set average rating is called so that the average rating would change dynamically as the review is added.
300. \*/
301. public void addReview( String username, String description, int rating) throws InputValidationException {
302. reviews.put(username, new Review(description, rating));
303. setAverageRating();
304. }
306. /\*\*
307. \* method to remove reivew from a restuarnt
308. \* @param user
309. \* @param review
310. \*/
311. public void removeReview(User user, Review review) {
312. // reviews.remove(user.getUsername(), review);
313. // setAverageRating();
315. reviews.remove(user.getUsername());
316. setAverageRating();
317. }


321. //CompareTo method that we should implement as the class implements comparable
322. //Here we are defining comparision constrains
324. @Override
325. public int compareTo(Restaurant o) {
326. if (this.averageRating > o.getAverageRating()) {
327. return -1;
328. }
329. else {
330. return 1;
331. }
333. }

336. @Override
337. public String toString() {
339. return
340. name + "(" + String.format("%.2f", getAverageRating()) + ")" + "\n" + restaurantDescription +"\nCuisine: " +cuisine + "\nAddress: " + address + "\nContact: " +contact;
341. }
342. @Override
343. public Iterator<Review> iterator() {
344. return reviews.values().iterator();
345. }
347. }

#### Review.java

1. /\*\*
2. \*
3. \*/
4. package application;
6. /\*\*
7. \* This class represents the review.
8. \*
9. \* @author Yogesh Parajuli
10. \*
11. \*/
12. public class Review {
13. private int reviewId;
14. private String description;
15. private int rating;

18. /\*\*
19. \* Constructor for Review class
20. \* @param description
21. \* @param rating
22. \* @throws InputValidationException
23. \*/
24. public Review(String description, int rating) throws InputValidationException {
25. setDescription(description);
26. setRating(rating);
28. }
29. /\*\*
30. \* Constructor used during data retrival from database
31. \* @param reviewId
32. \* @param rating
33. \* @param description
34. \* @throws InputValidationException
35. \*/
36. public Review(int reviewId, int rating, String description) {
38. this.description = description;
39. this.rating = rating;
40. setReviewId(reviewId);
41. }

44. /\*\*
45. \* Getter for review id
46. \* @return reviewID
47. \*/
48. public int getReviewId() {
49. return reviewId;
50. }
52. /\*\*
53. \* Setter for reviewId
54. \* @param reviewId
55. \*/
56. public void setReviewId(int reviewId) {
57. this.reviewId = reviewId;
58. }
59. /\*\*
60. \* Getter method for description
61. \* @return description
62. \*/
63. public String getDescription() {
64. return description;
65. }

68. /\*\*
69. \* Setter method for description
70. \* It has a regex to validate the input.
71. \* @param description
72. \* @throws InputValidationException
73. \*/
74. public void setDescription(String description) throws InputValidationException {
75. if(description.matches("(\\p{ASCII}){4,100}")) {
77. this.description = description;
78. }
79. else {
80. throw new InputValidationException();
81. }
82. }
84. /\*\*
85. \* Getter method for rating
86. \* @return rating
87. \*/
88. public int getRating() {
89. return rating;
90. }

93. /\*\*
94. \* Setter method for rating
95. \* @param rating
96. \* @throws InputValidationException
97. \*/
98. public void setRating(int rating) throws InputValidationException {
99. if(rating<=10 && rating >0) {
100. this.rating = rating;
101. }
102. else {
103. throw new InputValidationException();
105. }
106. }
108. @Override
109. public String toString() {
110. return getDescription() + "\n Rating: " + getRating();
111. }
113. }

#### SoftwareUsers.java

1. package application;
3. /\*\*
4. \* Represents user of the software
5. \* @author Yogesh Parajuli
6. \*
7. \*/
8. public class SoftwareUsers{

11. private String username;
12. private String password;
14. /\*\*
15. \* We use instance variable that is subclass of UsabilityPrivilege Interface
16. \*
17. \*/
18. public UsabilityPrivilege privilege;
20. /\*\*
21. \* Constructor for SoftwareUsers class
22. \* @param username
23. \* @param password
24. \* @throws InputValidationException
25. \*
26. \*/
27. public SoftwareUsers(String username, String password) throws InputValidationException {
28. setUsername(username);
29. setPassword(password);
31. }
32. /\*\*
33. \* Second Constructor for Software User class
34. \* This is used when retrieving data from the database
35. \* @param username
36. \*/
37. public SoftwareUsers(String username) {
38. this.username = username;
39. }
41. /\*\*
42. \* Getter method for username
43. \* @return username
44. \*/
45. public String getUsername() {
46. return username;
47. }
49. /\*\*
50. \* Setter method for username
51. \* The user name should can have any alphabet and numbers.
52. \* The length of username should be 3 or more and less than 10.
53. \* @param username
54. \* @throws InputValidationException
55. \*/
56. public void setUsername(String username) throws InputValidationException {
57. if(username.matches("^[a-zA-Z0-9]{3,10}$")) {
58. this.username = username;
59. }
60. else {
61. throw new InputValidationException();
62. }
64. }
66. /\*\*
67. \* Getter method for password
68. \* @return password
69. \*/
70. public String getPassword() {
71. return password;
72. }
74. /\*\*
75. \* Setter method for password
76. \* The password can be any alphanumeric value.
77. \* The length of password should be 6 letters or more and less than 20
78. \* @param password
79. \* @throws InputValidationException
80. \*/
81. public void setPassword(String password) throws InputValidationException {
82. if(password.matches("^[a-zA-Z0-9]{6,20}$")) {
83. this.password = password;
84. }
85. else {
86. throw new InputValidationException();
87. }
88. }
90. /\*\*
91. \* This method allows the subclasses to have their own adminPrivilege
92. \* @return privilege
93. \*
94. \*/
95. public boolean areYouAdmin() {
96. return privilege.adminPrivilege();
97. }
99. @Override
100. public int hashCode() {
101. return getUsername().hashCode();
102. }

105. }

#### testScript.java

1. package application;
3. import java.util.ArrayList;
4. import java.util.TreeSet;
6. import Database.dataSQLite;
8. /\*\*
9. \* This is a test script used for white box testing
10. \*
11. \* Please delete the previous sqlite file before running this test script
12. \*
13. \* @author Yogesh Parajuli
14. \*
15. \*/
16. public class testScript {
18. /\*\*
19. \* Main Method
20. \* @param args
21. \*/
22. public static void main(String[] args) {
24. //This is test stage one of whitebox testing
25. //This includes testing of individual classes for valid inputs and invalid inputs

28. // Software Users tests
30. SoftwareUsers testSoftwareUsers = null;
31. System.out.println("SoftwareUsers Class Checks");
33. try {
34. //valid inputs
35. testSoftwareUsers = new SoftwareUsers("yogesh12", "password123");
36. System.out.println("Pass");
38. System.out.println(testSoftwareUsers.getUsername());
39. System.out.println(testSoftwareUsers.getPassword());
41. //Invalid Usename Check
42. try {
43. testSoftwareUsers.setUsername("y");
44. System.out.println("Fail");
45. }
46. catch(InputValidationException e) {
47. System.out.println("Pass");
48. }
50. try {
51. testSoftwareUsers.setUsername("yog3shp@");
52. System.out.println("Fail");
53. }
54. catch(InputValidationException e) {
55. System.out.println("Pass");
56. }

59. //Invalid Password Check
60. try {
61. testSoftwareUsers.setPassword("pass");
62. System.out.println("Fail");
63. }
64. catch(InputValidationException e) {
65. System.out.println("Pass");
66. }
68. try {
69. testSoftwareUsers.setPassword("p@ssword1");
70. System.out.println("Fail");
71. }
72. catch(InputValidationException e) {
73. System.out.println("Pass");
74. }
76. }
77. catch (InputValidationException e) {
78. System.out.println("Fail");
79. }



84. //Users test
85. User testUser = null;
86. System.out.println("User class Checks");
87. try {
88. //For username and password, user calls its super class
89. //i.e. softwareUsers class which has already been tested above
91. //valid inputs
92. testUser = new User("yogesh12", "password123","Yogesh", "Parajuli","yogesh@gmail.com");
93. System.out.println("Pass");
95. if(!testUser.areYouAdmin()) {
96. //User does not have admin priviledge
97. System.out.println("Pass");
98. }
99. else {
100. System.out.println("Fail");
101. }
103. System.out.println(testUser.getfName());
104. System.out.println(testUser.getsName());
105. System.out.println(testUser.getEmail());
106. System.out.println(testUser.getUsername());
107. System.out.println(testUser.getPassword());
109. //Invalid first name checks
110. try {
111. //invalid length
112. testUser.setfName("y");
113. System.out.println("Fail");
114. }
115. catch(InputValidationException e) {
116. System.out.println("Pass");
117. }
119. try {
120. //invalid characters
121. testUser.setfName("Yog3sh");
122. System.out.println("Fail");
123. }
124. catch(InputValidationException e) {
125. System.out.println("Pass");
126. }
128. //Invalid Second name checks
129. try {
130. //invalid length
131. testUser.setsName("p");
132. System.out.println("Fail");
133. }
134. catch(InputValidationException e) {
135. System.out.println("Pass");
136. }
138. try {
139. //invalid characters
140. testUser.setsName("p@r@juli");
141. System.out.println("Fail");
142. }
143. catch(InputValidationException e) {
144. System.out.println("Pass");
145. }
147. }
148. catch(InputValidationException e) {
149. System.out.println("Fail");
150. }


154. //Restaurant Test
155. Restaurant testRestaurant = null;
156. System.out.println("Restaurant class Checks");
158. try {
159. // all valid inputs
160. testRestaurant = new Restaurant("Alpha Restaurant", "British Cuisine at its finest.", "British","01416666666", "100 Hello Road" );
161. System.out.println("Pass");
163. System.out.println(testRestaurant.getName());
164. System.out.println(testRestaurant.getRestaurantDescription());
165. System.out.println(testRestaurant.getCuisine());
166. System.out.println(testRestaurant.getContact());
167. System.out.println(testRestaurant.getAddress());
169. //Invalid Name Checks
170. try {
171. //invalid length for restaurant name and special characters are not allowed
172. testRestaurant.setName("Hi@");
173. System.out.println("Fail");
174. }
175. catch (InputValidationException e) {
176. System.out.println("Pass");
177. }
179. try {
180. //invalid length for restaurant name
181. testRestaurant.setName("The best Restaurant that you have ever been to.");
182. System.out.println("Fail");
183. }
184. catch (InputValidationException e) {
185. System.out.println("Pass");
186. }
188. //Invalid Restaurant Description Check
189. try {
190. //invalid length for a description
191. testRestaurant.setRestaurantDescription("Hi");
192. System.out.println("Fail");
193. }
194. catch(InputValidationException e) {
195. System.out.println("Pass");
196. }
198. try {
199. //invalid length for a description
200. testRestaurant.setRestaurantDescription("This is the text that contains more than 100 characters. The restaurant is the best one that you have ever been to.");
201. System.out.println("Fail");
202. }
203. catch(InputValidationException e) {
204. System.out.println("Pass");
205. }

208. //Invalid Cuisine check
209. try {
210. testRestaurant.setCuisine("Br");
211. System.out.println("Fail");
212. }
213. catch(InputValidationException e) {
214. System.out.println("Pass");
215. }
217. try {
218. testRestaurant.setCuisine("Best British Ever");
219. System.out.println("Fail");
220. }
221. catch(InputValidationException e) {
222. System.out.println("Pass");
223. }
225. //Invalid Contact Check
226. try {
227. //the contact number has to be 11 digits
228. testRestaurant.setContact("123456789");
229. System.out.println("Fail");
230. }
231. catch(InputValidationException e) {
232. System.out.println("Pass");
233. }
235. try {
236. //only digits are allowed
237. testRestaurant.setContact("Hello");
238. System.out.println("Fail");
239. }
240. catch(InputValidationException e) {
241. System.out.println("Pass");
242. }
244. //Invalid Address Check
245. try {
246. //Adrees should have a number followed by a space followed by a word followed by another space and a word
247. testRestaurant.setAddress("Random Road");
248. System.out.println("Fail");
249. }
250. catch (InputValidationException e){
251. System.out.println("Pass");
252. }

255. try {
256. testRestaurant.setAddress("100");
257. System.out.println("Fail");
258. }
259. catch(InputValidationException e) {
260. System.out.println("Pass");
261. }
263. }
264. catch(InputValidationException e) {
265. System.out.println("Fail");
266. }
268. //Review Class Check
269. Review testReview = null;
270. System.out.println("Review Class checks");
272. try {
273. testReview = new Review("I absolutely loved this place", 8);
274. System.out.println("Pass");
276. System.out.println(testReview.getRating());
277. System.out.println(testReview.getDescription());
279. //Invalid description test
280. try {
281. testReview.setDescription("Nah");
282. System.out.println("Fail");
283. }
284. catch(InputValidationException e) {
285. System.out.println("Pass");
286. }
288. try {
289. testReview.setDescription("This is a review description that contains more than one hundred characters. It was just awesome. Just awesome.");
290. System.out.println("Fail");
291. }
292. catch(InputValidationException e) {
293. System.out.println("Pass");
294. }
296. //Invalid Rating test
297. try {
298. testReview.setRating(-1);
299. System.out.println("Fail");
300. }
301. catch(InputValidationException e) {
302. System.out.println("Pass");
303. }
304. try {
305. testReview.setRating(11);
306. System.out.println("Fail");
307. }
308. catch(InputValidationException e) {
309. System.out.println("Pass");
310. }
312. }
313. catch(InputValidationException e) {
314. System.out.println("Fail");
315. }
317. //This is the second stage of testing
318. //In this stage, we check if the classes work together.
320. try {
321. //Created a couple of restaurant classes
322. Restaurant restaurant1 = new Restaurant("Alpha Restaurant", "British Cuisine at its finest.", "British","01416666666", "100 Hello Road");
323. Restaurant restaurant2 = new Restaurant("Beta Restaurant", "Indian Cuisine at its finest.", "Indian","01416666667", "5 Glasgow Cresent");

326. User user1 = new User("yogesh12", "password123","Yogesh", "Parajuli","yogesh@gmail.com");
327. User user2 = new User("random12", "password321","Random", "Person","random@gmail.com");
329. Review review1 = new Review("I had an awesome experience", 10);
330. Review review2 = new Review("It was really lovely.", 9);
332. restaurant1.addReview(user1.getUsername(), review1);
333. restaurant2.addReview(user2.getUsername(), review2);
334. restaurant1.addReview(user2.getUsername(),"Test Description", 6);
336. System.out.println(restaurant1.getReviews());
337. System.out.println("--");
338. System.out.println(restaurant2.getReviews());
339. System.out.println("--");
340. System.out.println("First Restaurant Average Rating:" + restaurant1.getAverageRating());
341. System.out.println("Second Restaurant Average Rating:" + restaurant2.getAverageRating());
342. System.out.println("--");
344. System.out.println("Comparision Check");
345. System.out.println("1 expected: " + restaurant1.compareTo(restaurant2));
346. System.out.println("-1 expected: " + restaurant2.compareTo(restaurant1));
347. System.out.println("--");
348. System.out.println("Remove Review from restaurant check");
349. restaurant1.removeReview(user2, review1);
350. System.out.println(restaurant1.getReviews());
351. System.out.println("-1 expected: " + restaurant1.compareTo(restaurant2));


355. }
356. catch(InputValidationException e) {
357. System.out.println("Fail");
358. }

361. //The third stage is when we test the database
362. //the database is controlled by dataSQLite class
363. //In this stage we test all the methods in the dataSQLite class
364. dataSQLite data = dataSQLite.getInstance();
365. System.out.println("---");
366. System.out.println("Database Check\n");

369. //Here we create some more restaurant object and some more review objects.
370. try {
372. User user1 = new User("yogesh12", "password123","Yogesh", "Parajuli","yogesh@gmail.com");
373. User user2 = new User("random12", "password321","Random", "Person","random@gmail.com");
375. Restaurant restaurant1 = new Restaurant("Alpha Restaurant", "British Cuisine at its finest.", "British","01416666666", "100 Hello Road");
376. Restaurant restaurant2 = new Restaurant("Beta Restaurant", "Indian Cuisine at its finest.", "Indian","01416666667", "5 Glasgow Cresent");
377. Restaurant restaurant3 = new Restaurant("Charlie Restaurant", "French Cuisine at its finest.", "French","01416666677", "12 Glasgow Cresent");
378. Restaurant restaurant4 = new Restaurant("Zuli Restaurant", "Italian Cuisine at its finest.", "Italian","01416666767", "5 Heaven Road");
380. Review review1 = new Review("I had an awesome experince", 10);
381. Review review2 = new Review("It was really lovely.", 9);
382. Review review3 = new Review("Worst Place ever", 1);
383. Review review4 = new Review("It was okay", 5);
385. // Now we add restaurants on the database
386. data.saveRestaurant(restaurant1);
387. data.saveRestaurant(restaurant2);
388. data.saveRestaurant(restaurant3);
389. data.saveRestaurant(restaurant4);
391. //check this by opening all the restaurants
392. TreeSet<Restaurant> restaurants = data.openRestaurants();
394. System.out.println("\n Here are a list of restaurants that have been added to the database\n" + restaurants);
396. //database sets the id as auto increment instead of the restaurant class itselg
397. //we dont set the id when an object is created
398. //but here we need an id to add review on a restaurant
399. //This works well with the GUI because we will be selectin a restaurant that has been given an id by the database
400. restaurant1.setId(1);
401. restaurant2.setId(2);
402. restaurant3.setId(3);
403. restaurant4.setId(4);
405. //Now we save reviews on the restaurants
406. data.saveReview(user1, restaurant1, review1);
407. data.saveReview(user2, restaurant1, review2);
409. data.saveReview(user1, restaurant2, review3);
410. data.saveReview(user2, restaurant2, review4);
412. data.saveReview(user1, restaurant4, review2);
413. data.saveReview(user2, restaurant4, review1);

416. System.out.println("\n Here the restaurants should be printed with their average rating on the side.");
417. //check this by opening all the restaurants along with the reviews
419. TreeSet<Restaurant> restWithReview = data.openRestaurants();
420. for(Restaurant r: restWithReview) {
421. r.setReviews(data.openReview(r.getId()));
422. r.getAverageRating();
424. System.out.println(r);
425. }

428. //add users to the database
429. data.saveUser(user1);
430. data.saveUser(user2);

433. System.out.println("\n List of all users");
434. //Check this by opening all the users
435. ArrayList<User> users = data.openAllUsers();
436. System.out.println(users);
438. //open a single user
439. System.out.println("----");
440. System.out.println("Retrieve single user");
441. System.out.println(data.openUser(user1.getUsername()));
443. System.out.println("\nCheck for username");
444. //CheckUsername
445. System.out.println("Should be true: " + data.checkUsername(user1.getUsername()));
446. System.out.println("Should be false: " + data.checkUsername("Random1234"));

449. System.out.println("\n validate username and password");
450. //Validate Username
451. System.out.println("\nShould be true: " + data.validate(user1.getUsername(), user1.getPassword()));
453. System.out.println("\nShould be false: " + data.validate("Completely Random", "Hello World"));
455. //removeUser
456. System.out.println("\n Remove a user test");
457. data.removeUser(user1.getUsername());
459. System.out.println("Should be false:" + data.checkUsername(user1.getUsername()));

462. //remove Restaurant
463. System.out.println("\nRemove a restaurant");


467. data.removeRestaurant(restaurant3);
468. System.out.println(data.openRestaurants());
469. System.out.println(restaurant3.getName() + "should not be in the list");
471. //Update password test
472. System.out.println("\nUpdate Password");
473. data.updatePassword(user2.getUsername(), "TestPassword1");
475. System.out.println("Should be true:" + data.validate(user2.getUsername(), "TestPassword1"));

478. //Update Restaurant test
479. System.out.println("\nUpdate Restaurant");
480. data.updateRestaurant(2, "01419090900", "1 ONe Road", "Updated Description");
482. System.out.println(data.openRestaurants());
483. System.out.println(restaurant2.getName() + "Should have updated details.");

486. } catch (InputValidationException e) {
487. // TODO Auto-generated catch block
488. System.out.println("Fail");
489. }
491. }
493. }

#### UsabilityPrivilege.java

1. package application;
3. /\*\*
4. \*
5. \* Using Strategy Design Pattern
6. \* Has methods that indicates if the software user has admin privilege or not.
7. \* @author Yogesh Parajuli
8. \*
9. \*/
10. public interface UsabilityPrivilege {
11. /\*\*
12. \* @return true or false i.e. if they have privilege or not.
13. \*/
14. boolean adminPrivilege();
15. }
17. class Privileged implements UsabilityPrivilege{
18. public boolean adminPrivilege() {
19. return true;
20. }
21. }
23. class notPrivileged implements UsabilityPrivilege{
24. public boolean adminPrivilege() {
25. return false;
26. }
27. }

#### User.java

1. /\*\*
2. \*
3. \*/
4. package application;
6. /\*\*
7. \* This is sub class of SoftwareUser class.
8. \* It represents just regular user
9. \* @author Yogesh Parajuli
10. \*
11. \*/
12. public class User extends SoftwareUsers{
14. private String fName,sName,email;
16. /\*\*
17. \* Constructor for User class
18. \* We also set the privilege to Not Privileged as regular user
19. \* doesnot have admin privilege.
20. \*
21. \* @param username
22. \* @param password
23. \* @param fName
24. \* @param sName
25. \* @param email
26. \* @throws InputValidationException
27. \*/
29. public User(String username, String password, String fName, String sName, String email) throws InputValidationException {
30. super(username, password);
31. setfName(fName);
32. setEmail(email);
33. setsName(sName);
34. privilege = new notPrivileged();
36. }
38. /\*\*
39. \* Constructor for User class
40. \* We also set the privilege to Not Privileged as regular user
41. \* doesnot have admin privilege.
42. \*
43. \* @param username
44. \* @param password
45. \* @param fName
46. \* @param sName
47. \* @param email
48. \* @throws InputValidationException
50. \*/
51. public User(String username, String fName, String sName, String email) throws InputValidationException{
52. super(username);
53. setfName(fName);
54. setEmail(email);
55. setsName(sName);
56. // this.fName = fName;
57. // this.email = email;
58. // this.sName = sName;
59. privilege = new notPrivileged();
61. }



66. /\*\*
67. \* Getter method for fName
68. \* @return fName
69. \*/
70. public String getfName() {
71. return fName;
72. }

75. /\*\*
76. \* Setter method for fName
77. \* The name has to be all alphabetic characters.
78. \*
79. \* @param fName
80. \* @throws InputValidationException
81. \*/
82. public void setfName(String fName) throws InputValidationException {
83. if(fName.matches("^[a-zA-Z]{3,15}$")) {
84. this.fName = fName;
85. }
86. else {
87. throw new InputValidationException();
88. }
89. }
91. /\*\*
92. \* Getter method for sName
93. \* @return sName
94. \*/
95. public String getsName() {
96. return sName;
97. }
99. /\*\*
100. \* Setter method for sName
101. \* Again, similar Regex pattern to the fName
102. \* @param sName
103. \* @throws InputValidationException
104. \*/
105. public void setsName(String sName) throws InputValidationException {
106. if(sName.matches("^[a-zA-Z]{3,15}$")) {
107. this.sName = sName;
108. }
109. else {
110. throw new InputValidationException();
111. }
112. }
114. /\*\*
115. \* Getter method for email
116. \* @return email
117. \*/
118. public String getEmail() {
119. return email;
120. }


124. /\*\*
125. \* Setter method for email
126. \* Appropriate regex has been used.
127. \* @param email
128. \* @throws InputValidationException
129. \*/
130. public void setEmail(String email) throws InputValidationException {
131. if(email.matches("^(.+)@(.+)$")) {
132. this.email = email;
133. }
134. else {
135. throw new InputValidationException();
136. }
138. }
140. @Override
141. public String toString() {
142. //return "User [fName=" + getfName() + ", sName=" + getsName() + ", email=" + getEmail() + ", Username=" + getUsername()
143. // + "]";
144. return "Username: " + getUsername() + "\nEmail: " + getEmail();
145. }

148. }

### Database Layer

#### dataSQLite.java

1. package Database;
2. import java.sql.\*;
3. import java.util.ArrayList;
4. import java.util.HashMap;
5. import java.util.TreeSet;
7. import application.InputValidationException;
8. import application.Restaurant;
9. import application.Review;
11. import application.User;
13. /\*\*
14. \*
15. \* This is the data layer of the program
16. \* @author Yogesh Parajuli
17. \*
18. \*/
19. public final class dataSQLite {
20. private static final dataSQLite instance = new dataSQLite();
22. /\*\*
23. \* Using Singleton pattern.
24. \* Returns the dataSQLite's instance
25. \* @return instance
26. \*/
27. public static dataSQLite getInstance() {
28. return instance;
29. }
31. //Private constructor so that the class cannot be instantiated
32. // The constructor also creates the necessary tables on the database
33. @SuppressWarnings("deprecation")
34. private dataSQLite() {
35. try {
36. //Loads SQLite Driver
37. Class.forName("org.sqlite.JDBC").newInstance();
38. }
39. catch (IllegalAccessException e) {
40. e.printStackTrace();
41. } catch (ClassNotFoundException e) {
42. e.printStackTrace();
43. } catch (InstantiationException e) {
44. e.printStackTrace();
45. }
47. try {
49. // Make connection with the database
50. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
52. //Create a User Table if it hasnt already been created
53. PreparedStatement statement =connection.prepareStatement("CREATE TABLE IF NOT EXISTS user (username VARCHAR(256) PRIMARY KEY NOT NULL, password VARCHAR(256) NOT NULL,fName VARCHAR(256) NOT NULL, sName VARCHAR(256) NOT NULL, email VARCHAR(256) NOT NULL)");
55. try {
56. statement.executeUpdate();
57. statement.close();

60. // Create Restaurant Table if not created already
61. statement = connection.prepareStatement("CREATE TABLE IF NOT EXISTS restaurant (restId INTEGER PRIMARY KEY AUTOiNCREMENT, name VARCHAR(256) NOT NULL,description VARCHAR(256) NOT NULL, cuisine VARCHAR(256) NOT NULL,contact VARCHAR(256) NOT NULL,address VARCHAR(256) NOT NULL)");
63. statement.executeUpdate();
64. statement.close();
66. //Create Review Table if not created already
68. // Foreign keys have been initialised
69. //If data in parent table are deleted, the clild class data are also deleted
70. //i.e if restaurant is deleted, reviews related to the restaurant are also deleted
71. // same with the user
72. statement = connection.prepareStatement("CREATE TABLE IF NOT EXISTS review (reviewId INTEGER PRIMARY KEY AUTOINCREMENT, restId INTEGER NOT NULL, username VARCHAR(256) NOT NULL, rating INTEGER NOT NULL, description VARCHAR(256) NOT NULL, FOREIGN KEY (restId) REFERENCES restaurant (restId) ON DELETE CASCADE ON UPDATE NO ACTION, FOREIGN KEY (username) REFERENCES user(username) ON UPDATE NO ACTION ON DELETE CASCADE)");
74. statement.executeUpdate();
75. statement.close();
77. }
78. catch (SQLException e) {
80. }
81. finally {
82. statement.close();
83. connection.close();
84. }
85. }
86. catch (SQLException e){
87. e.printStackTrace();
89. }
91. }// closing parenthesis for constructor
93. /\*\*
94. \* Method to save restaurants onto the database
95. \* @param restaurant
96. \*/
97. //method to save restarant
98. public void saveRestaurant(Restaurant restaurant) {
99. try {
100. //Establish connection
101. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
103. //Write a statement
104. PreparedStatement statement = connection.prepareStatement("INSERT INTO restaurant( name,description, cuisine, contact, address) VALUES (?,?,?,?,?)");
106. try{
107. //SQL statements to assign value to the "?"
108. //statement.setString(1, restaurant.getId());
109. statement.setString(1, restaurant.getName());
110. statement.setString(2, restaurant.getRestaurantDescription());
111. statement.setString(3, restaurant.getCuisine());
112. statement.setString(4, restaurant.getContact());
113. statement.setString(5, restaurant.getAddress());
115. statement.executeUpdate();
116. statement.close();
118. }
119. catch (SQLException e) {
120. e.printStackTrace();
121. }
122. finally {
123. statement.close();
124. connection.close();
125. }
127. }
128. catch (SQLException e) {
129. e.printStackTrace();
130. }
132. }




138. /\*\*
139. \* Method to save review onto the database
140. \* @param user
141. \* @param restaurant
142. \* @param review
143. \*
144. \*/
145. public void saveReview(User user,Restaurant restaurant, Review review) {
147. try {
148. //Establish a connection
149. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
151. //Prepare a statement
152. PreparedStatement statement = connection.prepareStatement("INSERT INTO review ( restId, rating, description,username) VALUES (?,?,?,?)");
154. try {
156. statement.setInt(1, restaurant.getId());
157. statement.setInt(2, review.getRating());
158. statement.setString(3, review.getDescription());
159. statement.setString(4, user.getUsername());
161. statement.executeUpdate();
162. statement.close();
164. }
165. catch(SQLException e) {
166. e.printStackTrace();
167. }
168. finally {
169. statement.close();
170. connection.close();
171. }
172. }
173. catch (SQLException e) {
174. e.printStackTrace();
175. }

178. }
179. /\*\*
180. \* Method to save user into the database
181. \* @param user
182. \*/
183. //method to save users
184. public void saveUser(User user) {
186. try {
187. //Establish a connection
188. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
190. //prepare a statement
191. PreparedStatement statement = connection.prepareStatement("INSERT INTO user (username,password,fName,sName,email) VALUES (?,?,?,?,?)");
193. try {
194. statement.setString(1, user.getUsername());
195. statement.setString(2, user.getPassword());
196. statement.setString(3, user.getfName());
197. statement.setString(4, user.getsName());
198. statement.setString(5, user.getEmail());
200. statement.executeUpdate();
201. statement.close();
202. }
203. catch(SQLException e) {
204. e.printStackTrace();
205. }
206. finally {
207. statement.close();
208. connection.close();
209. }
210. }
211. catch(SQLException e) {
212. e.printStackTrace();
213. }
215. }
217. /\*\*
218. \* Since we are letting the user to choose their own username, we need to check if the username is already taken
219. \* @param username
220. \* @return whether there is a match in username i.e expect true if username is already in the database
221. \*/
222. public boolean checkUsername(String username) {
223. try {
224. //Establish a connection
225. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
227. PreparedStatement statement = connection.prepareStatement("SELECT username FROM user");
229. try {
230. ResultSet results = statement.executeQuery();
231. while(results.next()) {
232. if (username.equals(results.getString("username"))) {
233. return true;
234. }
235. }
236. statement.close();
237. }
238. catch(SQLException e) {
239. e.printStackTrace();
240. }
241. finally {
242. statement.close();
243. connection.close();
244. }
246. }
247. catch(SQLException e) {
248. e.printStackTrace();
249. }
250. return false;

253. }

256. /\*\*
257. \* Since we are letting the user to choose their own username, we need to check if the username is already taken
258. \* @param username
259. \* @param restId
260. \* @return whether there is a match in username i.e expect true if username is already in the database
261. \*/
262. public boolean checkReview(String username, int restId) {
263. try {
264. //Establish a connection
265. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
267. PreparedStatement statement = connection.prepareStatement("SELECT username, restId FROM review");
269. try {
270. ResultSet results = statement.executeQuery();
271. while(results.next()) {
272. if (username.equals(results.getString("username")) && restId == results.getInt("restId")) {
273. return true;
274. }
275. }
276. statement.close();
277. }
278. catch(SQLException e) {
279. e.printStackTrace();
280. }
281. finally {
282. statement.close();
283. connection.close();
284. }
286. }
287. catch(SQLException e) {
288. e.printStackTrace();
289. }
290. return false;

293. }


297. /\*\*
298. \* Method to validate username and password for login
299. \* @param username
300. \* @param password
301. \* @return if the username and password are valid i.e expects true if username and password match and are stored in the database
302. \*/
303. public boolean validate (String username, String password) {
304. try {
305. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
307. PreparedStatement statement = connection.prepareStatement("SELECT username, password FROM user");
308. try {
309. ResultSet results = statement.executeQuery();
310. while(results.next()) {
311. if (username.equals(results.getString("username")) && password.equals(results.getString("password"))) {
312. return true;
313. }
314. }
315. statement.close();
316. }
317. catch(SQLException e) {
318. e.printStackTrace();
319. }
320. finally {
321. connection.close();
322. statement.close();
323. }
324. }
325. catch(SQLException e) {
326. e.printStackTrace();
327. }
328. return false;
329. }
331. /\*\*
332. \* A method to retrieve a User object
333. \* @param username
334. \* @param password
335. \* @return User object
336. \* @throws InputValidationException
337. \*/
338. public User openUser(String username) throws InputValidationException {
339. User user;
340. try {
341. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
343. PreparedStatement statement1 = connection.prepareStatement("Select DISTINCT \* FROM user WHERE username = ?");
344. try {
345. statement1.setString(1, username);
347. // there are no if clause used here because this method will be called after validate method
348. //therefore we know that the result exists and there are no null values
349. ResultSet results = statement1.executeQuery();
350. //System.out.println(results.isClosed());
352. user = new User(results.getString("username"), results.getString("fName"), results.getString("sName"), results.getString("email"));
354. return user;


358. }
359. catch(SQLException e) {
360. e.printStackTrace();
361. }
362. finally {
363. connection.close();
364. statement1.close();
365. }
367. }
368. catch(SQLException e) {
369. e.printStackTrace();
370. }
371. return null;
372. }
374. /\*\*
375. \* Method to retrieve all the users that are registered to the system
376. \* @return collection of users
377. \* @throws InputValidationException
378. \*/
379. public ArrayList<User> openAllUsers() throws InputValidationException{
380. ArrayList<User> users = new ArrayList<User>();
382. try {
383. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
384. PreparedStatement statement = connection.prepareStatement("Select \* FROM user");
386. try {
387. ResultSet allUsers = statement.executeQuery();
388. while(allUsers.next()) {
389. users.add(new User(allUsers.getString("username"), allUsers.getString("fName"), allUsers.getString("sName"), allUsers.getString("email")));
390. }
392. }
393. catch(SQLException e) {
394. e.printStackTrace();
395. }
396. finally {
397. connection.close();
398. statement.close();
399. }
400. }
401. catch(SQLException e) {
402. e.printStackTrace();
403. }
404. return users;
406. }

409. /\*\*
410. \*
411. \* Method to retrieve Restaurant data held by the database
412. \* @return Review Eat object (collection of restaurants)
414. \*/
415. public TreeSet<Restaurant> openRestaurants() {
416. //we create a review eat object to populate it with details
417. //ReviewEat re = new ReviewEat();
419. TreeSet<Restaurant> re = new TreeSet<>();

422. try {
423. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");

426. //create a statement to get restaurants
427. PreparedStatement statement = connection.prepareStatement("SELECT \* FROM restaurant ORDER BY name");
428. try {
429. ResultSet results = statement.executeQuery();
430. //Loop to populate Review Eat object with restaurants
431. while (results.next()) {
432. Restaurant restaurant = new Restaurant (results.getInt("restId"), results.getString("name"), results.getString("description"), results.getString("cuisine"), results.getString("contact"), results.getString("address"));
434. //restaurant.setReview(openReview(restaurant.getId()));
435. re.add(restaurant);
437. }

440. }
441. catch (SQLException e) {
442. e.printStackTrace();
443. }
444. finally {
445. connection.close();
446. statement.close();
448. }
450. }
451. catch (SQLException e) {
452. e.printStackTrace();
453. }
454. return re;
456. }


460. /\*\*
461. \*Method to open reviews from the database
462. \* @param restId
463. \* @return arraylist
464. \* @throws InputValidationException
465. \*/
466. public HashMap<String,Review> openReview(int restId) {
467. HashMap<String,Review> reviews= new HashMap<String, Review>();
468. try {
469. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
471. PreparedStatement statement = connection.prepareStatement("SELECT \* FROM review WHERE restId = ?");
472. try {
473. statement.setInt(1, restId);
475. ResultSet allReviews = statement.executeQuery();
476. while(allReviews.next()) {
478. // Review r = new Review(allReviews.getInt("rating"),allReviews.getString("description"));
479. //User u = openUser(allReviews.getString("username"));
480. reviews.put ( allReviews.getString("username"),new Review(allReviews.getInt("reviewId"),allReviews.getInt("rating"), allReviews.getString("description")));
482. }
483. }
484. catch(SQLException e) {
485. e.printStackTrace();
486. }
487. finally {
488. connection.close();
489. statement.close();
490. }
491. }
492. catch (SQLException e) {
493. e.printStackTrace();
494. }
495. return reviews;
497. }

500. /\*\*
501. \* Method to remove user from the database
502. \* @param username
503. \* @param password
504. \*/
505. public void removeUser(String username) {
506. try {
507. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");

510. //create a statement to remove the user from the database
511. PreparedStatement statement = connection.prepareStatement("DELETE FROM user WHERE username = ?");
513. try {
514. statement.setString(1, username);
515. statement.executeUpdate();
517. statement.close();
519. removeReview(username);
520. }
521. catch (SQLException e) {
522. e.printStackTrace();
523. }
524. finally {
525. connection.close();
526. statement.close();
528. }
529. }
530. catch(SQLException e) {
531. e.printStackTrace();
532. }

535. }
537. /\*\*
538. \* Method to remove restaurant from the database
539. \* @param restaurant
540. \*/
541. public void removeRestaurant(Restaurant restaurant) {
542. try {
543. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
545. PreparedStatement statement = connection.prepareStatement("DELETE FROM restaurant WHERE restId = ?");
546. try {
547. statement.setInt(1, restaurant.getId());
548. statement.executeUpdate();
550. statement.close();
551. removeReview(restaurant);
553. }
554. catch(SQLException e){
555. e.printStackTrace();
556. }
557. finally {
558. statement.close();
559. connection.close();
560. }
561. }
562. catch(SQLException e) {
563. e.printStackTrace();
564. }

567. }


571. /\*\*
572. \* I have added "ON CASCADE" constrain in reveiw class however during testing, review is not being deleted from the database when user is deleted
573. \* hence i have created a method to force delete the review associated with the user
574. \* @param username
575. \*/
576. public void removeReview(String username) {
577. try {
578. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
580. PreparedStatement statement = connection.prepareStatement("DELETE FROM review WHERE username = ?");
581. try {
582. statement.setString(1, username);
583. statement.executeUpdate();
585. statement.close();
587. }
588. catch(SQLException e){
589. e.printStackTrace();
590. }
591. finally {
592. statement.close();
593. connection.close();
594. }
595. }
596. catch(SQLException e) {
597. e.printStackTrace();
598. }
600. }

603. /\*\*
604. \* Same as above
605. \* But here review is deleted when restaurant is deleted
606. \* @param restaurant
607. \*/
608. public void removeReview(Restaurant restaurant) {
609. try {
610. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
612. PreparedStatement statement = connection.prepareStatement("DELETE FROM review WHERE restId = ?");
613. try {
614. statement.setInt(1, restaurant.getId());
615. statement.executeUpdate();
617. statement.close();
619. }
620. catch(SQLException e){
621. e.printStackTrace();
622. }
623. finally {
624. statement.close();
625. connection.close();
626. }
627. }
628. catch(SQLException e) {
629. e.printStackTrace();
630. }
632. }
634. /\*\*
635. \* Method to update users password
636. \* @param username
637. \* @param newPassword
638. \*/
639. public void updatePassword(String username, String newPassword) {
640. try {
641. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
643. PreparedStatement statement = connection.prepareStatement("UPDATE user SET password = ? WHERE username = ?");
645. try {
646. statement.setString(1, newPassword);
647. statement.setString(2, username);
648. statement.executeUpdate();
650. statement.close();
652. }
653. catch(SQLException e) {
654. e.printStackTrace();
655. }
656. finally {
657. statement.close();
658. connection.close();
659. }
660. }
661. catch(SQLException e) {
662. e.printStackTrace();
663. }
664. }


668. /\*\*
669. \* Method to update restaurant
670. \* @param restId
671. \* @param contact
672. \* @param address
673. \* @param description
674. \*/
675. public void updateRestaurant (int restId, String contact, String address, String description) {
676. try {
677. Connection connection = DriverManager.getConnection("jdbc:sqlite:reviewEat.sqlite");
679. PreparedStatement statement = connection.prepareStatement("UPDATE restaurant SET contact = ?, address = ? , description =? WHERE restId = ?");
681. try {
682. statement.setString(1, contact );
683. statement.setString(2, address);
684. statement.setString(3, description);
685. statement.setInt(4, restId);
686. statement.executeUpdate();
688. statement.close();
689. }
690. catch(SQLException e) {
691. e.printStackTrace();
692. }
693. finally {
694. statement.close();
695. connection.close();
696. }
697. }
698. catch(SQLException e) {
699. e.printStackTrace();
700. }
702. }
704. }

### GUI

The GUI section is further divided into FXML files and FXML controller files.

#### FXML Files

##### addRestaurantFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.ChoiceBox?>
5. <?import javafx.scene.control.Label?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.control.TextArea?>
8. <?import javafx.scene.control.TextField?>
9. <?import javafx.scene.layout.AnchorPane?>
10. <?import javafx.scene.text.Font?>
12. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.addRestaurantFXMLController">
13. <children>
14. <AnchorPane **layoutY**="4.0" **prefHeight**="492.0" **prefWidth**="407.0">
15. <children>
16. <Label **layoutX**="33.0" **layoutY**="24.0" **prefHeight**="42.0" **prefWidth**="341.0" **text**="Welcome to Review Eat">
17. <font>
18. <Font **name**="18thCentury" **size**="40.0" />
19. </font>
20. </Label>
21. <Separator **layoutX**="56.0" **layoutY**="64.0" **prefHeight**="2.0" **prefWidth**="294.0" />
22. <Label **layoutX**="150.0" **layoutY**="66.0" **prefHeight**="26.0" **prefWidth**="106.0" **text**="Add Restaurant">
23. <font>
24. <Font **name**="18thCentury" **size**="19.0" />
25. </font>
26. </Label>
27. <Label **layoutX**="40.0" **layoutY**="125.0" **text**="Name:">
28. <font>
29. <Font **name**="18thCentury" **size**="19.0" />
30. </font>
31. </Label>
32. <Separator **layoutX**="114.0" **layoutY**="92.0" **prefHeight**="3.0" **prefWidth**="177.0" />
33. <Label **layoutX**="40.0" **layoutY**="160.0" **text**="Cuisine:">
34. <font>
35. <Font **name**="18thCentury" **size**="19.0" />
36. </font>
37. </Label>
38. <Label **layoutX**="40.0" **layoutY**="195.0" **text**="Address:">
39. <font>
40. <Font **name**="18thCentury" **size**="19.0" />
41. </font>
42. </Label>
43. <Label **layoutX**="40.0" **layoutY**="230.0" **text**="Contact:">
44. <font>
45. <Font **name**="18thCentury" **size**="19.0" />
46. </font>
47. </Label>
48. <Label **layoutX**="40.0" **layoutY**="265.0" **text**="Description:">
49. <font>
50. <Font **name**="18thCentury" **size**="19.0" />
51. </font>
52. </Label>
53. <TextField **fx:id**="nameTextField" **layoutX**="173.0" **layoutY**="122.0" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="Name" />
54. <TextField **fx:id**="addressTextField" **layoutX**="173.0" **layoutY**="192.0" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="Street Number and Street Name" />
55. <TextField **fx:id**="contactTextField" **layoutX**="173.0" **layoutY**="227.0" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="11 Digits" />
56. <Button **fx:id**="addButton" **layoutX**="86.0" **layoutY**="420.0" **mnemonicParsing**="false" **onAction**="#addRestaurant" **prefHeight**="32.0" **prefWidth**="84.0" **text**="Add">
57. <font>
58. <Font **name**="18thCentury" **size**="18.0" />
59. </font>
60. </Button>
61. <Button **fx:id**="cancelButton" **layoutX**="246.0" **layoutY**="420.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="26.0" **prefWidth**="84.0" **text**="Cancel">
62. <font>
63. <Font **name**="18thCentury" **size**="18.0" />
64. </font>
65. </Button>
66. <TextArea **fx:id**="descriptionTextArea" **layoutX**="173.0" **layoutY**="265.0" **maxHeight**="-Infinity" **minHeight**="-Infinity" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="100 Characters Maximum" />
67. <ChoiceBox **fx:id**="cuisineDropBox" **layoutX**="171.0" **layoutY**="158.0" **prefHeight**="26.0" **prefWidth**="187.0" />
68. </children>
69. </AnchorPane>
70. </children>
71. </AnchorPane>

##### addReviewFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.ChoiceBox?>
5. <?import javafx.scene.control.Label?>
6. <?import javafx.scene.control.TextArea?>
7. <?import javafx.scene.layout.AnchorPane?>
8. <?import javafx.scene.text.Font?>
10. <AnchorPane **prefHeight**="334.0" **prefWidth**="310.0" **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.addReviewFXMLController">
11. <children>
12. <AnchorPane **prefHeight**="366.0" **prefWidth**="371.0">
13. <children>
14. <Label **layoutX**="47.0" **layoutY**="14.0" **prefHeight**="32.0" **prefWidth**="278.0" **text**="Welcome to Review Eat">
15. <font>
16. <Font **name**="18thCentury" **size**="33.0" />
17. </font>
18. </Label>
19. <Label **layoutX**="74.0" **layoutY**="104.0" **text**="Rating:">
20. <font>
21. <Font **name**="18thCentury" **size**="19.0" />
22. </font>
23. </Label>
24. <Label **layoutX**="39.0" **layoutY**="169.0" **text**="Description:">
25. <font>
26. <Font **name**="18thCentury" **size**="19.0" />
27. </font>
28. </Label>
29. <TextArea **fx:id**="descriptionTextArea" **layoutX**="129.0" **layoutY**="169.0" **prefHeight**="114.0" **prefWidth**="200.0" **promptText**="Detailed Review" />
30. <Button **fx:id**="addReveiwButton" **layoutX**="74.0" **layoutY**="314.0" **mnemonicParsing**="false" **onAction**="#addReview" **prefHeight**="26.0" **prefWidth**="94.0" **text**="Add Review">
31. <font>
32. <Font **name**="18thCentury" **size**="16.0" />
33. </font></Button>
34. <Button **fx:id**="cancelButton" **layoutX**="238.0" **layoutY**="314.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="26.0" **prefWidth**="94.0" **text**="Cancel">
35. <font>
36. <Font **name**="18thCentury" **size**="16.0" />
37. </font></Button>
38. <ChoiceBox **fx:id**="ratingChoiceBox" **layoutX**="131.0" **layoutY**="101.0" **prefWidth**="150.0" />
39. <Label **layoutX**="129.0" **layoutY**="54.0" **text**="Add Review">
40. <font>
41. <Font **name**="18thCentury" **size**="19.0" />
42. </font>
43. </Label>
44. </children>
45. </AnchorPane>
46. </children>
47. </AnchorPane>

##### adminViewReviewEat.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.ListView?>
6. <?import javafx.scene.layout.AnchorPane?>
7. <?import javafx.scene.text.Font?>
9. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.adminViewReviewEatFXMLController">
10. <children>
11. <AnchorPane **prefHeight**="500.0" **prefWidth**="504.0">
12. <children>
13. <Label **layoutX**="80.0" **layoutY**="26.0" **text**="Welcome to Review Eat">
14. <font>
15. <Font **name**="18thCentury" **size**="42.0" />
16. </font>
17. </Label>
18. <Label **layoutX**="62.0" **layoutY**="92.0" **text**="Restaurants">
19. <font>
20. <Font **name**="18thCentury" **size**="23.0" />
21. </font>
22. </Label>
23. <ListView **fx:id**="listRestaurants" **fixedCellSize**="100.0" **layoutX**="38.0" **layoutY**="125.0" **prefHeight**="310.0" **prefWidth**="428.0" />
24. <Button **fx:id**="addRestaurantButton" **layoutX**="35.0" **layoutY**="448.0" **mnemonicParsing**="false" **onAction**="#addRestaurant" **prefHeight**="35.0" **prefWidth**="148.0" **text**="Add Restaurant">
25. <font>
26. <Font **name**="18thCentury" **size**="20.0" />
27. </font>
28. </Button>
29. <Button **fx:id**="updateRestaurantButton" **layoutX**="191.0" **layoutY**="448.0" **mnemonicParsing**="false" **onAction**="#updateRestaurant" **prefHeight**="35.0" **prefWidth**="148.0" **text**="Update Restaurant">
30. <font>
31. <Font **name**="18thCentury" **size**="17.0" />
32. </font>
33. </Button>
34. <Button **fx:id**="deleteRestaurantButton" **layoutX**="346.0" **layoutY**="448.0" **mnemonicParsing**="false" **onAction**="#deleteRestaurant" **prefHeight**="35.0" **prefWidth**="148.0" **text**="Delete Restaurant">
35. <font>
36. <Font **name**="18thCentury" **size**="18.0" />
37. </font>
38. </Button>
39. <Button **fx:id**="logOutButton" **layoutX**="420.0" **layoutY**="92.0" **mnemonicParsing**="false" **onAction**="#logOut" **text**="Log Out">
40. <font>
41. <Font **name**="18thCentury" **size**="12.0" />
42. </font>
43. </Button>
44. <Button **fx:id**="deleteUsersButton" **layoutX**="333.0" **layoutY**="94.0" **mnemonicParsing**="false" **onAction**="#removeUsers" **text**="Remove Users">
45. <font>
46. <Font **name**="18thCentury" **size**="12.0" />
47. </font>
48. </Button>
49. </children>
50. </AnchorPane>
51. </children>
52. </AnchorPane>

##### deleteAccountFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.PasswordField?>
6. <?import javafx.scene.control.TextField?>
7. <?import javafx.scene.layout.AnchorPane?>
8. <?import javafx.scene.text.Font?>
10. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.deleteAccountFXMLController">
11. <children>
12. <AnchorPane **layoutX**="1.0" **layoutY**="1.0" **prefHeight**="413.0" **prefWidth**="312.0">
13. <children>
14. <Label **layoutX**="18.0" **layoutY**="32.0" **prefHeight**="34.0" **prefWidth**="275.0" **text**="Welcome to ReviewEat" **underline**="true">
15. <font>
16. <Font **name**="18thCentury" **size**="33.0" />
17. </font>
18. </Label>
19. <TextField **fx:id**="usernameTextField" **layoutX**="123.0" **layoutY**="148.0" **promptText**="Username" />
20. <Label **layoutX**="42.0" **layoutY**="150.0" **text**="Username:">
21. <font>
22. <Font **name**="18thCentury" **size**="20.0" />
23. </font>
24. </Label>
25. <Label **layoutX**="42.0" **layoutY**="205.0" **text**="Password:">
26. <font>
27. <Font **name**="18thCentury" **size**="20.0" />
28. </font>
29. </Label>
30. <PasswordField **fx:id**="passwordTextField" **layoutX**="123.0" **layoutY**="203.0" **promptText**="Password" />
31. <Button **fx:id**="deleteButton" **layoutX**="100.0" **layoutY**="268.0" **mnemonicParsing**="false" **onAction**="#deleteUser" **prefHeight**="30.0" **prefWidth**="112.0" **text**="Delete Account">
32. <font>
33. <Font **name**="18thCentury" **size**="16.0" />
34. </font>
35. </Button>
36. <Button **fx:id**="cancelButton" **layoutX**="100.0" **layoutY**="320.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="30.0" **prefWidth**="112.0" **text**="Cancel">
37. <font>
38. <Font **name**="18thCentury" **size**="16.0" />
39. </font>
40. </Button>
41. <Label **layoutX**="111.0" **layoutY**="81.0" **text**="Delete Account">
42. <font>
43. <Font **name**="18thCentury" **size**="17.0" />
44. </font>
45. </Label>
46. <Label **layoutX**="83.0" **layoutY**="114.0" **text**="Please enter your details.">
47. <font>
48. <Font **name**="18thCentury" **size**="17.0" />
49. </font>
50. </Label>
51. </children>
52. </AnchorPane>
53. </children>
54. </AnchorPane>

##### loginFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.PasswordField?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.control.TextField?>
8. <?import javafx.scene.layout.AnchorPane?>
9. <?import javafx.scene.text.Font?>
11. <AnchorPane **prefHeight**="413.0" **prefWidth**="312.0" **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.loginFXMLController">
12. <children>
13. <AnchorPane **layoutX**="-7.0" **prefHeight**="414.0" **prefWidth**="326.0">
14. <children>
15. <Label **layoutX**="91.0" **layoutY**="97.0" **text**="Please Login to Continue.">
16. <font>
17. <Font **name**="18thCentury" **size**="17.0" />
18. </font>
19. </Label>
20. <Label **layoutX**="25.0" **layoutY**="27.0" **prefHeight**="34.0" **prefWidth**="275.0" **text**="Welcome to ReviewEat" **underline**="true">
21. <font>
22. <Font **name**="18thCentury" **size**="33.0" />
23. </font>
24. </Label>
25. <Label **layoutX**="42.0" **layoutY**="135.0" **text**="Username:">
26. <font>
27. <Font **name**="18thCentury" **size**="20.0" />
28. </font>
29. </Label>
30. <Label **layoutX**="42.0" **layoutY**="192.0" **text**="Password:">
31. <font>
32. <Font **name**="18thCentury" **size**="20.0" />
33. </font>
34. </Label>
35. <TextField **fx:id**="usernameField" **layoutX**="125.0" **layoutY**="131.0" **promptText**="Username" />
36. <PasswordField **fx:id**="passwordField" **layoutX**="125.0" **layoutY**="188.0" **promptText**="Password" />
37. <Button **fx:id**="registerButton" **alignment**="CENTER" **layoutX**="42.0" **layoutY**="325.0" **mnemonicParsing**="false" **onAction**="#register" **text**="Register Here">
38. <font>
39. <Font **name**="18thCentury" **size**="13.0" />
40. </font>
41. </Button>
42. <Button **fx:id**="updatePasswordButton" **layoutX**="178.0" **layoutY**="327.0" **mnemonicParsing**="false" **onAction**="#updatePassword" **text**="Update Password">
43. <font>
44. <Font **name**="18thCentury" **size**="12.0" />
45. </font>
46. </Button>
47. <Button **fx:id**="loginButton" **layoutX**="126.0" **layoutY**="236.0" **mnemonicParsing**="false" **onAction**="#login" **prefHeight**="30.0" **prefWidth**="72.0" **text**="Login">
48. <font>
49. <Font **name**="18thCentury" **size**="16.0" />
50. </font>
51. </Button>
52. <Button **fx:id**="deleteAccountButton" **layoutX**="178.0" **layoutY**="359.0" **mnemonicParsing**="false" **onAction**="#deleteAccount" **text**="Delete Account">
53. <font>
54. <Font **name**="18thCentury" **size**="12.0" />
55. </font>
56. </Button>
57. <Label **layoutX**="42.0" **layoutY**="302.0" **text**="New to the system? ">
58. <font>
59. <Font **name**="18thCentury" **size**="14.0" />
60. </font>
61. </Label>
62. <Separator **layoutX**="32.0" **layoutY**="290.0" **prefHeight**="2.0" **prefWidth**="261.0" />
63. <Label **layoutX**="163.0" **layoutY**="302.0" **text**="Looking for something else?">
64. <font>
65. <Font **name**="18thCentury" **size**="14.0" />
66. </font>
67. </Label>
68. <Separator **layoutX**="148.0" **layoutY**="296.0" **orientation**="VERTICAL" **prefHeight**="81.0" **prefWidth**="5.0" />
69. </children>
70. </AnchorPane>
71. </children>
72. </AnchorPane>

##### registerUserFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.PasswordField?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.control.TextField?>
8. <?import javafx.scene.layout.AnchorPane?>
9. <?import javafx.scene.text.Font?>
11. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.registerUserFXMLController">
12. <children>
13. <AnchorPane **prefHeight**="413.0" **prefWidth**="312.0">
14. <children>
15. <Label **layoutX**="34.0" **layoutY**="26.0" **text**="Welcome to Review Eat">
16. <font>
17. <Font **name**="18thCentury" **size**="30.0" />
18. </font>
19. </Label>
20. <Label **layoutX**="101.0" **layoutY**="68.0" **text**="Registration Form">
21. <font>
22. <Font **name**="18thCentury" **size**="18.0" />
23. </font>
24. </Label>
25. <Label **layoutX**="48.0" **layoutY**="140.0" **text**="First Name:">
26. <font>
27. <Font **name**="18thCentury" **size**="16.0" />
28. </font>
29. </Label>
30. <Separator **layoutX**="56.0" **layoutY**="94.0" **prefWidth**="200.0" />
31. <Label **layoutX**="38.0" **layoutY**="170.0" **text**="Second Name:">
32. <font>
33. <Font **name**="18thCentury" **size**="16.0" />
34. </font>
35. </Label>
36. <Label **layoutX**="79.0" **layoutY**="200.0" **text**="Email:">
37. <font>
38. <Font **name**="18thCentury" **size**="16.0" />
39. </font>
40. </Label>
41. <Label **layoutX**="59.0" **layoutY**="230.0" **text**="Username:">
42. <font>
43. <Font **name**="18thCentury" **size**="16.0" />
44. </font>
45. </Label>
46. <Label **layoutX**="22.0" **layoutY**="290.0" **text**="Retype Password:">
47. <font>
48. <Font **name**="18thCentury" **size**="16.0" />
49. </font>
50. </Label>
51. <Label **layoutX**="60.0" **layoutY**="260.0" **text**="Password:">
52. <font>
53. <Font **name**="18thCentury" **size**="16.0" />
54. </font>
55. </Label>
56. <Label **layoutX**="86.0" **layoutY**="97.0" **text**="Please Enter Your Details">
57. <font>
58. <Font **name**="18thCentury" **size**="16.0" />
59. </font>
60. </Label>
61. <Button **fx:id**="registerButton" **layoutX**="70.0" **layoutY**="337.0" **mnemonicParsing**="false" **onAction**="#registerUser" **text**="Register">
62. <font>
63. <Font **name**="18thCentury" **size**="15.0" />
64. </font>
65. </Button>
66. <Button **fx:id**="cancelButton" **layoutX**="183.0" **layoutY**="337.0" **mnemonicParsing**="false" **onAction**="#cancel" **text**="Cancel">
67. <font>
68. <Font **name**="18thCentury" **size**="15.0" />
69. </font>
70. </Button>
71. <TextField **fx:id**="fNameTextfield" **layoutX**="128.0" **layoutY**="135.0" **promptText**="First Name " />
72. <TextField **fx:id**="sNameTextfield" **layoutX**="128.0" **layoutY**="165.0" **promptText**="Second Name" />
73. <TextField **fx:id**="emailTextfield" **layoutX**="128.0" **layoutY**="195.0" **promptText**="Valid Email Address" />
74. <TextField **fx:id**="usernameTextfield" **layoutX**="128.0" **layoutY**="225.0" **promptText**="Username" />
75. <PasswordField **fx:id**="passwordTextfield" **layoutX**="128.0" **layoutY**="255.0" **promptText**="6- 20 Characters " />
76. <PasswordField **fx:id**="rePasswordTextfield" **layoutX**="128.0" **layoutY**="285.0" **promptText**="Retype Password" />
77. </children>
78. </AnchorPane>
79. </children>
80. </AnchorPane>

##### removeUsersFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.ListView?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.layout.AnchorPane?>
8. <?import javafx.scene.layout.StackPane?>
9. <?import javafx.scene.text.Font?>
11. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.removeUsersFXMLController">
12. <children>
13. <AnchorPane **prefHeight**="385.0" **prefWidth**="372.0">
14. <children>
15. <Label **layoutX**="35.0" **layoutY**="14.0" **text**="Welcome to ReviewEat">
16. <font>
17. <Font **name**="18thCentury" **size**="40.0" />
18. </font>
19. </Label>
20. <Label **layoutX**="134.0" **layoutY**="56.0" **text**="Remove Users">
21. <font>
22. <Font **name**="18thCentury" **size**="21.0" />
23. </font>
24. </Label>
25. <StackPane **layoutX**="29.0" **layoutY**="91.0" **prefHeight**="227.0" **prefWidth**="316.0">
26. <children>
27. <ListView **fx:id**="usersListView" **fixedCellSize**="50.0" **prefHeight**="228.0" **prefWidth**="279.0" />
28. </children>
29. </StackPane>
30. <Separator **layoutX**="86.0" **layoutY**="54.0" **prefWidth**="200.0" />
31. <Button **fx:id**="removeUsersButton" **layoutX**="41.0" **layoutY**="332.0" **mnemonicParsing**="false" **onAction**="#removeUser" **prefHeight**="26.0" **prefWidth**="130.0" **text**="Remove User">
32. <font>
33. <Font **name**="18thCentury" **size**="15.0" />
34. </font>
35. </Button>
36. <Button **fx:id**="cancelButton" **layoutX**="216.0" **layoutY**="332.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="26.0" **prefWidth**="112.0" **text**="Cancel">
37. <font>
38. <Font **name**="18thCentury" **size**="15.0" />
39. </font>
40. </Button>
41. </children>
42. </AnchorPane>
43. </children>
44. </AnchorPane>

##### updatePasswordFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.PasswordField?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.control.TextField?>
8. <?import javafx.scene.layout.AnchorPane?>
9. <?import javafx.scene.text.Font?>
11. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.updatePasswordFXMLController">
12. <children>
13. <AnchorPane **prefHeight**="413.0" **prefWidth**="312.0">
14. <children>
15. <Label **layoutX**="18.0" **layoutY**="32.0" **prefHeight**="34.0" **prefWidth**="275.0" **text**="Welcome to ReviewEat" **underline**="true">
16. <font>
17. <Font **name**="18thCentury" **size**="33.0" />
18. </font>
19. </Label>
20. <TextField **fx:id**="usernameTextField" **layoutX**="144.0" **layoutY**="148.0" **promptText**="Username" />
21. <Label **layoutX**="62.0" **layoutY**="150.0" **text**="Username:">
22. <font>
23. <Font **name**="18thCentury" **size**="20.0" />
24. </font>
25. </Label>
26. <Label **layoutX**="10.0" **layoutY**="195.0" **prefHeight**="22.0" **prefWidth**="145.0" **text**="Current Password:">
27. <font>
28. <Font **name**="18thCentury" **size**="20.0" />
29. </font>
30. </Label>
31. <PasswordField **fx:id**="currentPasswordTextField" **layoutX**="144.0" **layoutY**="194.0" **promptText**="Current Password" />
32. <Button **fx:id**="updateButton" **layoutX**="27.0" **layoutY**="343.0" **mnemonicParsing**="false" **onAction**="#updatePassword" **prefHeight**="30.0" **prefWidth**="121.0" **text**="Update Password">
33. <font>
34. <Font **name**="18thCentury" **size**="16.0" />
35. </font>
36. </Button>
37. <Button **fx:id**="cancelButton" **layoutX**="172.0" **layoutY**="343.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="30.0" **prefWidth**="112.0" **text**="Cancel">
38. <font>
39. <Font **name**="18thCentury" **size**="16.0" />
40. </font>
41. </Button>
42. <Label **layoutX**="99.0" **layoutY**="66.0" **text**="Update Password">
43. <font>
44. <Font **name**="18thCentury" **size**="17.0" />
45. </font>
46. </Label>
47. <Label **layoutX**="76.0" **layoutY**="115.0" **text**="Please enter your details.">
48. <font>
49. <Font **name**="18thCentury" **size**="17.0" />
50. </font>
51. </Label>
52. <Label **layoutX**="31.0" **layoutY**="242.0" **text**="New Password:">
53. <font>
54. <Font **name**="18thCentury" **size**="20.0" />
55. </font>
56. </Label>
57. <Label **layoutX**="31.0" **layoutY**="293.0" **text**="New Password:">
58. <font>
59. <Font **name**="18thCentury" **size**="20.0" />
60. </font>
61. </Label>
62. <PasswordField **fx:id**="newPasswordTextField" **layoutX**="144.0" **layoutY**="240.0" **promptText**="New Password" />
63. <PasswordField **fx:id**="reNewPasswordTextField" **layoutX**="144.0" **layoutY**="291.0" **promptText**="Retype New Password" />
64. <Separator **layoutX**="44.0" **layoutY**="90.0" **prefWidth**="200.0" />
65. </children>
66. </AnchorPane>
67. </children>
68. </AnchorPane>

##### updateRestaurantFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.Separator?>
6. <?import javafx.scene.control.TextArea?>
7. <?import javafx.scene.control.TextField?>
8. <?import javafx.scene.layout.AnchorPane?>
9. <?import javafx.scene.layout.StackPane?>
10. <?import javafx.scene.text.Font?>
12. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.updateRestaurantFXMLController">
13. <children>
14. <AnchorPane **layoutX**="10.0" **layoutY**="10.0">
15. <children>
16. <AnchorPane **layoutX**="-14.0" **layoutY**="-10.0" **prefHeight**="492.0" **prefWidth**="407.0">
17. <children>
18. <Label **layoutX**="33.0" **layoutY**="24.0" **prefHeight**="42.0" **prefWidth**="341.0" **text**="Welcome to Review Eat">
19. <font>
20. <Font **name**="18thCentury" **size**="40.0" />
21. </font>
22. </Label>
23. <Separator **layoutX**="56.0" **layoutY**="64.0" **prefHeight**="2.0" **prefWidth**="294.0" />
24. <Label **layoutX**="140.0" **layoutY**="66.0" **prefHeight**="26.0" **prefWidth**="126.0" **text**="Update Restaurant">
25. <font>
26. <Font **name**="18thCentury" **size**="19.0" />
27. </font>
28. </Label>
29. <Separator **layoutX**="114.0" **layoutY**="92.0" **prefHeight**="3.0" **prefWidth**="177.0" />
30. <Label **layoutX**="40.0" **layoutY**="305.0" **text**="Address:">
31. <font>
32. <Font **name**="18thCentury" **size**="19.0" />
33. </font>
34. </Label>
35. <Label **layoutX**="40.0" **layoutY**="340.0" **text**="Contact:">
36. <font>
37. <Font **name**="18thCentury" **size**="19.0" />
38. </font>
39. </Label>
40. <Label **layoutX**="41.0" **layoutY**="375.0" **text**="Description:">
41. <font>
42. <Font **name**="18thCentury" **size**="19.0" />
43. </font>
44. </Label>
45. <TextField **fx:id**="addressTextField" **layoutX**="163.0" **layoutY**="302.0" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="Street Number and Street Name" />
46. <TextField **fx:id**="contactTextField" **layoutX**="163.0" **layoutY**="337.0" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="11 Digits" />
47. <Button **fx:id**="updateButton" **layoutX**="86.0" **layoutY**="420.0" **mnemonicParsing**="false" **onAction**="#updateRestaurant" **prefHeight**="32.0" **prefWidth**="84.0" **text**="Update">
48. <font>
49. <Font **name**="18thCentury" **size**="18.0" />
50. </font>
51. </Button>
52. <Button **fx:id**="cancelButton" **layoutX**="246.0" **layoutY**="420.0" **mnemonicParsing**="false" **onAction**="#cancel" **prefHeight**="26.0" **prefWidth**="84.0" **text**="Cancel">
53. <font>
54. <Font **name**="18thCentury" **size**="18.0" />
55. </font>
56. </Button>
57. <TextArea **fx:id**="descriptionTextArea" **layoutX**="163.0" **layoutY**="372.0" **maxHeight**="-Infinity" **minHeight**="-Infinity" **prefHeight**="26.0" **prefWidth**="187.0" **promptText**="100 Characters Maximum" />
58. <StackPane **layoutX**="57.0" **layoutY**="106.0" **prefHeight**="95.0" **prefWidth**="294.0">
59. <children>
60. <StackPane **prefHeight**="150.0" **prefWidth**="200.0">
61. <children>
62. <Label **fx:id**="restaurantLabel" />
63. </children>
64. </StackPane>
65. </children></StackPane>
66. </children>
67. </AnchorPane>
68. </children>
69. </AnchorPane>
70. </children>
71. </AnchorPane>

##### userViewReviewEatFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.ListView?>
6. <?import javafx.scene.control.MenuButton?>
7. <?import javafx.scene.control.MenuItem?>
8. <?import javafx.scene.control.Separator?>
9. <?import javafx.scene.control.TextField?>
10. <?import javafx.scene.layout.AnchorPane?>
11. <?import javafx.scene.layout.HBox?>
12. <?import javafx.scene.layout.Pane?>
13. <?import javafx.scene.layout.StackPane?>
14. <?import javafx.scene.text.Font?>
16. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.userViewReviewEatFXMLController">
17. <children>
18. <AnchorPane **maxHeight**="-Infinity" **maxWidth**="-Infinity" **minHeight**="-Infinity" **minWidth**="-Infinity" **prefHeight**="700.0" **prefWidth**="700.0" **snapToPixel**="false">
19. <children>
20. <TextField **fx:id**="searchTextField" **layoutX**="274.0" **layoutY**="141.0" **minWidth**="-Infinity" **prefHeight**="25.0" **prefWidth**="191.0" **promptText**="Search" **snapToPixel**="false">
21. <font>
22. <Font **name**="18thCentury" **size**="24.0" />
23. </font>
24. </TextField>
25. <Label **layoutX**="180.0" **layoutY**="26.0" **text**="Welcome to Review Eat">
26. <font>
27. <Font **name**="18thCentury" **size**="42.0" />
28. </font>
29. </Label>
30. <Separator **layoutX**="179.0" **layoutY**="79.0" **prefHeight**="3.0" **prefWidth**="347.0" />
31. <Label **layoutX**="77.0" **layoutY**="197.0" **text**="Cuisine">
32. <font>
33. <Font **name**="18thCentury" **size**="23.0" />
34. </font>
35. </Label>
36. <Label **layoutX**="307.0" **layoutY**="197.0" **text**="Restaurant">
37. <font>
38. <Font **name**="18thCentury" **size**="23.0" />
39. </font>
40. </Label>
41. <Separator **layoutX**="5.0" **layoutY**="182.0" **prefHeight**="3.0" **prefWidth**="688.0" />
42. <Separator **layoutX**="10.0" **layoutY**="221.0" **prefHeight**="9.0" **prefWidth**="176.0" />
43. <StackPane **layoutX**="216.0" **layoutY**="236.0" **prefHeight**="379.0" **prefWidth**="474.0">
44. <children>
45. <ListView **fx:id**="restaurantListView" **fixedCellSize**="125.0" **prefHeight**="308.0" **prefWidth**="374.0" />
46. </children>
47. </StackPane>
48. <StackPane **layoutX**="216.0" **layoutY**="627.0" **prefHeight**="59.0" **prefWidth**="383.0">
49. <children>
50. <HBox **prefHeight**="100.0" **prefWidth**="200.0">
51. <children>
52. <Button **fx:id**="viewReviewButton" **mnemonicParsing**="false" **onAction**="#viewReview" **prefHeight**="32.0" **prefWidth**="121.0" **text**="View Reviews">
53. <font>
54. <Font **name**="18thCentury" **size**="15.0" />
55. </font>
56. </Button>
57. <Pane **prefHeight**="38.0" **prefWidth**="124.0" />
58. <Button **fx:id**="addReveiwButton" **mnemonicParsing**="false" **onAction**="#addButton" **prefHeight**="32.0" **prefWidth**="128.0" **text**="Add Review">
59. <font>
60. <Font **name**="18thCentury" **size**="15.0" />
61. </font>
62. </Button>
63. </children>
64. </HBox>
65. </children>
66. </StackPane>
67. <AnchorPane **layoutX**="14.0" **layoutY**="236.0" **prefHeight**="59.0" **prefWidth**="176.0">
68. <children>
69. <MenuButton **fx:id**="menuButton" **layoutX**="6.0" **layoutY**="14.0" **mnemonicParsing**="false" **prefHeight**="28.0" **prefWidth**="164.0" **text**="Filter By Cuisine">
70. <items>
71. <MenuItem **mnemonicParsing**="false" **onAction**="#defaultFilter" **text**="All" />
72. <MenuItem **mnemonicParsing**="false" **onAction**="#britishFilter" **text**="British" />
73. <MenuItem **mnemonicParsing**="false" **onAction**="#chineseFilter" **text**="Chinese" />
74. <MenuItem **mnemonicParsing**="false" **onAction**="#frenchFilter" **text**="French" />
75. <MenuItem **mnemonicParsing**="false" **onAction**="#indianFilter" **text**="Indian" />
76. <MenuItem **mnemonicParsing**="false" **onAction**="#italianFilter" **text**="Italian" />
77. <MenuItem **mnemonicParsing**="false" **onAction**="#jamaicanFilter" **text**="Jamaican" />
78. </items>
79. <font>
80. <Font **name**="18thCentury" **size**="17.0" />
81. </font>
82. </MenuButton>
83. </children>
84. </AnchorPane>
85. <Button **fx:id**="logOutButton" **layoutX**="589.0" **layoutY**="142.0" **minHeight**="17.0" **mnemonicParsing**="false" **onAction**="#logOut" **prefHeight**="22.0" **prefWidth**="85.0" **text**="Log Out">
86. <font>
87. <Font **name**="18thCentury" **size**="13.0" />
88. </font>
89. </Button>
90. <AnchorPane **layoutX**="14.0" **layoutY**="295.0" **prefHeight**="59.0" **prefWidth**="176.0">
91. <children>
92. <MenuButton **fx:id**="menuButton1" **layoutX**="7.0" **layoutY**="-1.0" **mnemonicParsing**="false" **prefHeight**="28.0" **prefWidth**="164.0" **text**="Sort By:">
93. <items>
94. <MenuItem **mnemonicParsing**="false" **onAction**="#defaultSort" **text**="Name" />
95. <MenuItem **mnemonicParsing**="false" **onAction**="#sortByRating" **text**="Rating" />
96. </items>
97. <font>
98. <Font **name**="18thCentury" **size**="17.0" />
99. </font>
100. </MenuButton>
101. </children>
102. </AnchorPane>
103. </children>
104. </AnchorPane>
105. </children>
106. </AnchorPane>

##### viewReviewsFXML.fxml

1. <?xml version="1.0" encoding="UTF-8"?>
3. <?import javafx.scene.control.Button?>
4. <?import javafx.scene.control.Label?>
5. <?import javafx.scene.control.ListView?>
6. <?import javafx.scene.control.Separator?>
7. <?import javafx.scene.layout.AnchorPane?>
8. <?import javafx.scene.layout.StackPane?>
9. <?import javafx.scene.text.Font?>
11. <AnchorPane **xmlns**="http://javafx.com/javafx/11.0.1" **xmlns:fx**="http://javafx.com/fxml/1" **fx:controller**="GUI.viewReviewsFXMLController">
12. <children>
13. <AnchorPane **prefHeight**="434.0" **prefWidth**="326.0">
14. <children>
15. <Separator **layoutX**="62.0" **layoutY**="45.0" **prefWidth**="200.0" />
16. <Label **layoutX**="37.0" **layoutY**="14.0" **text**="Welcome to Review Eat">
17. <font>
18. <Font **name**="18thCentury" **size**="31.0" />
19. </font>
20. </Label>
21. <Label **layoutX**="122.0" **layoutY**="57.0" **text**="View Review">
22. <font>
23. <Font **name**="18thCentury" **size**="18.0" />
24. </font>
25. </Label>
26. <StackPane **layoutX**="31.0" **layoutY**="183.0" **prefHeight**="185.0" **prefWidth**="278.0">
27. <children>
28. <ListView **fx:id**="reviewListView" **fixedCellSize**="100.0" **prefHeight**="200.0" **prefWidth**="200.0" />
29. </children>
30. </StackPane>
31. <Button **fx:id**="backButton" **layoutX**="121.0" **layoutY**="384.0" **mnemonicParsing**="false" **onAction**="#back" **prefHeight**="26.0" **prefWidth**="82.0" **text**="Back">
32. <font>
33. <Font **name**="18thCentury" **size**="14.0" />
34. </font>
35. </Button>
36. <StackPane **layoutX**="66.0" **layoutY**="77.0" **prefHeight**="92.0" **prefWidth**="194.0">
37. <children>
38. <Label **fx:id**="restaurantLabel">
39. <font>
40. <Font **name**="18thCentury" **size**="15.0" />
41. </font>
42. </Label>
43. </children>
44. </StackPane>
45. </children>
46. </AnchorPane>
47. </children>
48. </AnchorPane>

#### FXML Controllers

##### addRestaurantFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
5. import java.util.ResourceBundle;
6. import java.util.TreeSet;
8. import Database.dataSQLite;
9. import application.InputValidationException;
10. import application.Restaurant;
12. import javafx.event.ActionEvent;
13. import javafx.fxml.FXML;
14. import javafx.fxml.FXMLLoader;
15. import javafx.fxml.Initializable;
16. import javafx.scene.Node;
17. import javafx.scene.Parent;
18. import javafx.scene.Scene;
19. import javafx.scene.control.Alert;
20. import javafx.scene.control.Button;
21. import javafx.scene.control.ChoiceBox;
22. import javafx.scene.control.TextArea;
23. import javafx.scene.control.TextField;
24. import javafx.scene.layout.Region;
25. import javafx.stage.Stage;
26. import javafx.scene.control.Alert.AlertType;
28. /\*\*
29. \* Controller Class for Add Restaurant fxml
30. \* This class is responsible to provide an interface for adding retaurant to the database
31. \* @author Yogesh Parajuli
32. \*
33. \*/
34. public class addRestaurantFXMLController implements Initializable {
36. private dataSQLite data = dataSQLite.getInstance();
37. Alert a = new Alert(AlertType.NONE);
39. @FXML
40. private TextField nameTextField;
42. @FXML
43. private ChoiceBox<String> cuisineDropBox = new ChoiceBox<String>();
45. @FXML
46. private TextField addressTextField;
48. @FXML
49. private TextField contactTextField;
51. @FXML
52. private TextArea descriptionTextArea;
54. @FXML
55. private Button addButton;
57. @FXML
58. private Button cancelButton;
60. @FXML
61. void addRestaurant(ActionEvent event) throws IOException, NullPointerException {
62. String name = nameTextField.getText();
63. String cuisine = cuisineDropBox.getValue();
64. String address = addressTextField.getText();
65. String contact = contactTextField.getText();
66. String restaurantDescription = descriptionTextArea.getText();

69. // we need to perform some sort of check so that same restaurant is not added twice
70. TreeSet<Restaurant> restaurants = data.openRestaurants();
71. boolean check = false;
72. for (Restaurant r: restaurants) {
73. if (r.getAddress().equals(address) && r.getName().equals(name)) {
74. check = true;
75. }
76. }

79. if (!check) {
80. //save the restaurant to the database
81. try {
82. data.saveRestaurant(new Restaurant(name, restaurantDescription, cuisine, contact, address));
84. //display an alert saying that the restaurant has been added
85. a.setAlertType(AlertType.INFORMATION);
86. a.setHeaderText("Restaurant Added");
87. a.setContentText("Restaurant has been successfully added");
88. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
89. a.show();
91. // go back to the adminview fxml
92. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));
94. Scene scene = new Scene(root);
96. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
98. stage.setScene(scene);
99. stage.show();
101. } catch (InputValidationException e) {
103. // General Error Alert
104. e.printStackTrace();
105. a.setAlertType(AlertType.ERROR);
106. a.setHeaderText("Input Invalid");
107. a.setContentText("Please check the information you have entered.");
108. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
109. a.show();
110. }
111. }
112. else {
113. a.setAlertType(AlertType.ERROR);
114. a.setContentText("Restaurant has already been added.");
115. a.show();
116. }


120. }
122. // Takes you back to previous fxml
123. @FXML
124. void cancel(ActionEvent event) throws IOException {
125. // go back to the adminview fxml
126. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));
128. Scene scene = new Scene(root);
130. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
132. stage.setScene(scene);
133. stage.show();
135. }
137. @Override
138. public void initialize(URL arg0, ResourceBundle arg1) {
139. // Since we are using a dropbox, initialising the choice box with items and a default value
140. cuisineDropBox.getItems().addAll("British","Chinese", "French","Indian", "Italian", "Jamaican");
141. cuisineDropBox.setValue("British");
143. }
145. }

##### addReviewFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
6. import java.util.ResourceBundle;
8. import Database.dataSQLite;
9. import application.InputValidationException;
10. import application.Restaurant;
11. import application.Review;
13. import application.User;
14. import javafx.event.ActionEvent;
15. import javafx.fxml.FXML;
16. import javafx.fxml.FXMLLoader;
17. import javafx.fxml.Initializable;
18. import javafx.scene.control.Button;
19. import javafx.scene.control.ChoiceBox;
20. import javafx.scene.control.TextArea;
21. import javafx.scene.Node;
22. import javafx.scene.Parent;
23. import javafx.scene.Scene;
24. import javafx.scene.control.Alert;
25. import javafx.scene.control.Alert.AlertType;
26. import javafx.stage.Stage;
28. /\*\*
29. \* Controller class for add review fxml
30. \* This class is responsible for providing an interface to the user to add review on a restaurant
31. \* @author Yogesh parajuli
32. \*
33. \*/
34. public class addReviewFXMLController implements Initializable{
35. dataSQLite data = dataSQLite.getInstance();
37. // we need a user and restaurant object to add a review
38. // User is being passed on from previous fxml
39. // Restaurant object is being selected here
40. private User currentUser;
41. private Restaurant selectedRestaurant;
42. Alert a = new Alert(AlertType.NONE);


46. /\*\*
47. \* Setter for current user
48. \* @param currentUser
49. \*/
50. public void setCurrentUser(User currentUser) {
51. this.currentUser = currentUser;
52. }


56. /\*\*
57. \* Setter for the restaurant
58. \* @param selectedRestaurant
59. \*/
60. public void setSelectedRestaurant(Restaurant selectedRestaurant) {
61. this.selectedRestaurant = selectedRestaurant;
62. }

65. @FXML
66. private ChoiceBox<Integer> ratingChoiceBox = new ChoiceBox<Integer>();
68. @FXML
69. private TextArea descriptionTextArea;
71. @FXML
72. private Button addReveiwButton;
74. @FXML
75. private Button cancelButton;
77. @FXML
78. void addReview(ActionEvent event) throws IOException, NullPointerException{
79. //read in data from the fields
80. int rating = ratingChoiceBox.getValue();
81. String description = descriptionTextArea.getText();

84. try {
86. Review review = new Review(description, rating);
87. if(data.checkReview(currentUser.getUsername(), selectedRestaurant.getId())) {
88. //General error alert
89. a.setAlertType(AlertType.ERROR);
90. a.setContentText("User has already reviewed this restaurant.");
91. a.show();
93. }
94. else {
96. data.saveReview(currentUser, selectedRestaurant, review);
98. a.setAlertType(AlertType.CONFIRMATION);
99. a.setContentText("Review Saved");
100. a.show();


104. FXMLLoader loader = new FXMLLoader(getClass().getResource("userViewReviewEatFXML.fxml"));
105. Parent root = loader.load();
107. userViewReviewEatFXMLController nextController = loader.getController();
108. nextController.setUser(currentUser);

111. //Set the scene to the new root i.e new fxml file
112. Scene scene = new Scene(root);
114. //Set the stage information
115. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
117. stage.setScene(scene);
118. stage.show();
119. }
120. } catch (InputValidationException e) {
121. // TODO Auto-generated catch block
122. e.printStackTrace();
124. // General error alert
125. a.setAlertType(AlertType.ERROR);
126. a.setHeaderText("Invalid Input");
127. a.setContentText("Please Check the information you have entered");
128. a.show();
130. }
131. }



136. // Takes you back to previous fxml
137. @FXML
138. void cancel(ActionEvent event) throws IOException {

141. FXMLLoader loader = new FXMLLoader(getClass().getResource("userViewReviewEatFXML.fxml"));
142. Parent root = loader.load();
144. userViewReviewEatFXMLController nextController = loader.getController();
145. nextController.setUser(currentUser);

148. //Set the scene to the new root i.e new fxml file
149. Scene scene = new Scene(root);
151. //Set the stage information
152. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
154. stage.setScene(scene);
155. stage.show();
157. }


161. @Override
162. public void initialize(URL arg0, ResourceBundle arg1) {
163. // Populating the choice box with items and giving it a default value
164. ratingChoiceBox.getItems().addAll(1,2,3,4,5,6,7,8,9,10);
165. ratingChoiceBox.setValue(5);
167. }
169. }

##### adminViewReviewEatFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
5. import java.util.ResourceBundle;
6. import java.util.TreeSet;
8. import Database.dataSQLite;
9. import application.Restaurant;
10. import javafx.event.ActionEvent;
11. import javafx.fxml.FXML;
12. import javafx.fxml.FXMLLoader;
13. import javafx.fxml.Initializable;
14. import javafx.scene.Node;
15. import javafx.scene.Parent;
16. import javafx.scene.Scene;
17. import javafx.scene.control.Alert;
18. import javafx.scene.control.Button;
19. import javafx.scene.control.ListView;
20. import javafx.scene.control.Alert.AlertType;
21. import javafx.stage.Stage;

24. /\*\*
25. \* Controller for Admin View review eat
26. \* This class is the defauld view for the admin providing access to different actions
27. \* @author Yogesh Parajuli
28. \*
29. \*/
30. public class adminViewReviewEatFXMLController implements Initializable{

33. @FXML
34. private ListView<Restaurant> listRestaurants;
35. Alert a = new Alert(AlertType.NONE);
36. private dataSQLite data = dataSQLite.getInstance();

39. @Override
40. public void initialize(URL arg0, ResourceBundle arg1) {

43. // Populate the list view with restaurant
44. // Likewise, open the reviews that are associated with that restaurant
45. TreeSet <Restaurant> restaurants = data.openRestaurants();
46. for(Restaurant r: restaurants) {
47. r.setReviews(data.openReview(r.getId()));
49. r.getAverageRating();
50. }
51. listRestaurants.getItems().addAll(restaurants);

54. }


58. @FXML
59. private Button addRestaurantButton;
61. @FXML
62. private Button updateRestaurantButton;
64. @FXML
65. private Button deleteRestaurantButton;
67. @FXML
68. private Button deleteUsersButton;
70. @FXML
71. private Button logOutButton;

74. @FXML
75. void addRestaurant(ActionEvent event) throws IOException {
76. Parent root = FXMLLoader.load(getClass().getResource("addRestaurantFXML.fxml"));
78. Scene scene = new Scene(root);
80. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
82. stage.setScene(scene);
83. stage.show();
84. }

87. @FXML
88. void deleteRestaurant(ActionEvent event) {
89. Restaurant restaurant=listRestaurants.getSelectionModel().getSelectedItem();
90. if(restaurant != null) {
91. listRestaurants.getItems().remove(restaurant);
92. data.removeRestaurant(restaurant); }
93. else {
94. a.setAlertType(AlertType.ERROR);
95. a.setHeaderText("Invalid Choice");
96. a.setContentText("Please select an Item");
97. a.show();
98. }
99. }
101. @FXML
102. void updateRestaurant(ActionEvent event) throws IOException {
103. Restaurant restaurant=listRestaurants.getSelectionModel().getSelectedItem();
104. if(restaurant == null) {
105. a.setAlertType(AlertType.ERROR);
106. a.setHeaderText("Invalid Choice");
107. a.setContentText("Please select an Item");
108. a.show();
109. }
110. else {


114. FXMLLoader loader = new FXMLLoader(getClass().getResource("updateRestaurantFXML.fxml"));
115. Parent root = loader.load();
117. // Here we are trying to pass restaurant object to the next controller
119. updateRestaurantFXMLController nextController = loader.getController();
121. nextController.setRestaurant(restaurant);
122. nextController.setLabel(restaurant.toString());

125. Scene scene = new Scene(root);
127. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
129. stage.setScene(scene);
130. stage.show();
131. }
132. }
134. // Logs you out of the system
135. @FXML
136. void logOut(ActionEvent event) throws IOException {
137. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
139. Scene scene = new Scene(root);
141. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
143. stage.setScene(scene);
144. stage.show();
145. }
147. @FXML
148. void removeUsers(ActionEvent event) throws IOException {
149. Parent root = FXMLLoader.load(getClass().getResource("removeUsersFXML.fxml"));
151. Scene scene = new Scene(root);
153. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
155. stage.setScene(scene);
156. stage.show();
157. }
158. }

##### deleteAccountFXMLController.java

1. package GUI;
3. import java.io.IOException;
5. import Database.dataSQLite;
6. import javafx.event.ActionEvent;
7. import javafx.fxml.FXML;
8. import javafx.fxml.FXMLLoader;
9. import javafx.scene.Node;
10. import javafx.scene.Parent;
11. import javafx.scene.Scene;
12. import javafx.scene.control.Alert;
13. import javafx.scene.control.Button;
14. import javafx.scene.control.ButtonType;
15. import javafx.scene.control.PasswordField;
16. import javafx.scene.control.TextField;
17. import javafx.scene.layout.Region;
18. import javafx.scene.control.Alert.AlertType;
19. import javafx.stage.Stage;
21. /\*\*
22. \* Controller to delete Account fxml
23. \* This class is responsible for proving an interface to the user and allow them to delete thier account
24. \* @author Yogesh Parajuli
25. \*
26. \*
27. \*/
28. public class deleteAccountFXMLController {
29. private dataSQLite data = dataSQLite.getInstance();
30. Alert a = new Alert(AlertType.NONE);
32. @FXML
33. private TextField usernameTextField;
35. @FXML
36. private PasswordField passwordTextField;
38. @FXML
39. private Button deleteButton;
41. @FXML
42. private Button cancelButton;
44. //Takes user back to previous fxml
45. @FXML
46. void cancel(ActionEvent event) throws IOException {
47. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
49. Scene scene = new Scene(root);
51. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
53. stage.setScene(scene);
54. stage.show();
56. }
58. @FXML
59. void deleteUser(ActionEvent event) throws IOException {
60. String username = usernameTextField.getText();
61. String password = passwordTextField.getText();
63. // first we validate the username and password entered
64. if(data.validate(username, password)) {
65. a.setAlertType(AlertType.CONFIRMATION);
66. a.setHeaderText("Delete Account");
67. a.setContentText("Are you sure you want to delete your account?");
68. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
70. // Here we confirm if the user really wants to delete their account
71. if (a.showAndWait().get() == ButtonType.OK) {
72. data.removeUser(username);
74. a.setAlertType(AlertType.INFORMATION);
75. a.setHeaderText("Delete Account");
76. a.setContentText("Account Successfully Deleted");
77. a.show();

80. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
82. Scene scene = new Scene(root);
84. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
86. stage.setScene(scene);
87. stage.show();
89. }
90. else {
91. a.close();
92. }
94. }
95. else {
96. a.setAlertType(AlertType.ERROR);
97. a.setContentText("Account Not found");
98. a.setHeaderText("Delete Account");
99. a.show();
101. }
102. }
104. }

##### loginFXMLController.java

1. package GUI;
2. import java.io.IOException;
4. import Database.dataSQLite;
5. import application.Admin;
6. import application.InputValidationException;
7. import application.User;
8. import javafx.event.ActionEvent;
9. import javafx.fxml.FXML;
10. import javafx.fxml.FXMLLoader;
11. import javafx.scene.Node;
12. import javafx.scene.Parent;
13. import javafx.scene.Scene;
14. import javafx.scene.control.Alert;
15. import javafx.scene.control.Alert.AlertType;
16. import javafx.scene.control.Button;
17. import javafx.scene.control.PasswordField;
18. import javafx.scene.control.TextField;
19. import javafx.stage.Stage;
21. /\*\*
22. \*
23. \* Controller for loginFXML
24. \* This is the first fxml that can be viewed
25. \* This fxml provides access to vaious other fxmls
26. \* @author Yogesh Parajuli
27. \*
28. \*/
29. public class loginFXMLController {
30. User currentUser;
32. private dataSQLite data = dataSQLite.getInstance();
33. Alert a = new Alert(AlertType.NONE);
35. @FXML
36. private TextField usernameField;
38. @FXML
39. private PasswordField passwordField;
41. @FXML
42. private Button registerButton;
44. @FXML
45. private Button updatePasswordButton;
47. @FXML
48. private Button loginButton;
50. @FXML
51. private Button deleteAccountButton;
53. // takes you to delete accoutn page
54. @FXML
55. void deleteAccount(ActionEvent event) throws IOException {
56. Parent root = FXMLLoader.load(getClass().getResource("deleteAccountFXML.fxml"));
58. Scene scene = new Scene(root);
60. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
62. stage.setScene(scene);
63. stage.show();
65. }
67. //Logs you in
68. @FXML
69. void login(ActionEvent event) throws InputValidationException, IOException {
70. String username = usernameField.getText();
71. String password = passwordField.getText();
73. if(data.validate(username, password)) {
74. currentUser = data.openUser(username);
75. if (!currentUser.areYouAdmin()) {
76. // Change the parent i.e base class using fxml loader
78. FXMLLoader loader = new FXMLLoader(getClass().getResource("userViewReviewEatFXML.fxml"));
79. Parent root = loader.load();
81. // Trying to pass on a User object to next controller
82. userViewReviewEatFXMLController nextController = loader.getController();
83. nextController.setUser(currentUser);

86. //Set the scene to the new root i.e new fxml file
87. Scene scene = new Scene(root);
89. //Set the stage information
90. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
92. stage.setScene(scene);
93. stage.show();
94. }
96. }
98. //Admin details are not stored in the database
99. //instead they are set as private static attribute in admin class
101. else if(username.equals(Admin.getAdminUsername()) && password.equals(Admin.getAdminPassword())) {
102. // Change the parent i.e base class using fxml loader
103. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));

106. //Set the scene to the new root i.e new fxml file
107. Scene scene = new Scene(root);
109. //Set the stage information
110. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();



115. stage.setScene(scene);
117. stage.show();

120. }
121. else {
122. a.setAlertType(AlertType.ERROR);
123. a.setContentText("Login Unsuccessful. Please check your detail and try again");
124. a.show();
125. }

128. }
130. // Takes you to register page
131. @FXML
132. void register(ActionEvent event) throws IOException {
133. Parent root = FXMLLoader.load(getClass().getResource("registerUserFXML.fxml"));
135. Scene scene = new Scene(root);
137. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
139. stage.setScene(scene);
140. stage.show();
142. }
144. // Takes you to update password page
145. @FXML
146. void updatePassword(ActionEvent event) throws IOException {
147. Parent root = FXMLLoader.load(getClass().getResource("updatePasswordFXML.fxml"));
149. Scene scene = new Scene(root);
151. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
153. stage.setScene(scene);
154. stage.show();
156. }
158. }

##### registerUserFXMLController.java

1. package GUI;
3. import java.io.IOException;
5. import Database.dataSQLite;
6. import application.Admin;
7. import application.InputValidationException;
8. import application.User;
9. import javafx.event.ActionEvent;
10. import javafx.fxml.FXML;
11. import javafx.fxml.FXMLLoader;
12. import javafx.scene.Node;
13. import javafx.scene.Parent;
14. import javafx.scene.Scene;
15. import javafx.scene.control.Alert;
16. import javafx.scene.control.PasswordField;
17. import javafx.scene.control.TextField;
18. import javafx.scene.layout.Region;
19. import javafx.scene.control.Alert.AlertType;
20. import javafx.scene.control.Button;
21. import javafx.stage.Stage;
23. /\*\*
24. \* Register User fxml controller
25. \*
26. \* This class is responsible for registering a user
27. \* @author Yogesh Parajuli
28. \*
29. \*/
30. public class registerUserFXMLController {
31. User user;
32. private dataSQLite data = dataSQLite.getInstance();
33. Alert a = new Alert(AlertType.NONE);
35. @FXML
36. private Button registerButton;
38. @FXML
39. private Button cancelButton;
41. @FXML
42. private TextField fNameTextfield;
44. @FXML
45. private TextField sNameTextfield;
47. @FXML
48. private TextField emailTextfield;
50. @FXML
51. private TextField usernameTextfield;
53. @FXML
54. private PasswordField passwordTextfield;
56. @FXML
57. private PasswordField rePasswordTextfield;
59. //Takes you back to preious page
60. @FXML
61. void cancel(ActionEvent event) throws IOException {
62. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
64. Scene scene = new Scene(root);
66. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
68. stage.setScene(scene);
69. stage.show();

72. }

75. //regiseters the user
76. @FXML
77. void registerUser(ActionEvent event) throws IOException, NullPointerException{
78. String fName = fNameTextfield.getText();
79. String sName = sNameTextfield.getText();
80. String email = emailTextfield.getText();
81. String username = usernameTextfield.getText();
82. String password = passwordTextfield.getText();
83. String rePassword = rePasswordTextfield.getText();
85. //check the user name is available
86. //i.e. if the user with same name is already registered
87. if (data.checkUsername(username)) {
88. a.setAlertType(AlertType.ERROR);
89. a.setHeaderText("Username not available");
90. a.setContentText("Please choose a different username because current username has already been taken.");
91. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
92. a.show();
93. }
95. // Prevents using admin as username
96. else if (username.equals(Admin.getAdminUsername())) {
97. a.setAlertType(AlertType.ERROR);
98. a.setHeaderText("Reserved Username");
99. a.setContentText("You cannot use reserved username. Please choose a different username");
100. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
101. a.show();
102. }
103. //check the passwords match
104. else if(!password.equals(rePassword)) {
105. a.setAlertType(AlertType.ERROR);
106. a.setHeaderText("Passwords do not match.");
107. a.setContentText("Please type the passwords again");
108. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
109. a.show();
110. }
111. else {
112. //add to the database
113. try {
114. user = new User(username, password, fName, sName, email);
115. } catch (InputValidationException e) {
116. e.printStackTrace();
117. a.setAlertType(AlertType.ERROR);
118. a.setHeaderText("Input Invalid");
119. a.setContentText("Please check the details you have entered.");
120. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
121. a.show();
122. }
123. data.saveUser(user);
124. a.setAlertType(AlertType.CONFIRMATION);
125. a.setHeaderText("Registration successful");
126. a.setContentText("Now you can login to Revew Eat");
127. a.show();
129. // send the user back to login page
130. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
132. Scene scene = new Scene(root);
134. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
136. stage.setScene(scene);
137. stage.show();
138. }
140. }
142. }

##### removeUsersFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
5. import java.util.ResourceBundle;
7. import Database.dataSQLite;
8. import application.InputValidationException;
9. import application.User;
10. import javafx.event.ActionEvent;
11. import javafx.fxml.FXML;
12. import javafx.fxml.FXMLLoader;
13. import javafx.fxml.Initializable;
14. import javafx.scene.control.Button;
15. import javafx.scene.control.ButtonType;
16. import javafx.scene.control.ListView;
17. import javafx.scene.Node;
18. import javafx.scene.Parent;
19. import javafx.scene.Scene;
20. import javafx.scene.control.Alert;
21. import javafx.scene.control.Alert.AlertType;
22. import javafx.scene.layout.Region;
23. import javafx.stage.Stage;
25. /\*\*
26. \* Controller Class for remove users fxml
27. \* @author Yogesh Parajuli
28. \*
29. \*/
30. public class removeUsersFXMLController implements Initializable{
31. private dataSQLite data= dataSQLite.getInstance();
32. Alert a = new Alert(AlertType.NONE);

35. @FXML
36. private ListView<User> usersListView;
38. @FXML
39. private Button removeUsersButton;
41. @FXML
42. private Button cancelButton;
44. // Takes you back to previous window
45. @FXML
46. void cancel(ActionEvent event) throws IOException {
47. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));
49. Scene scene = new Scene(root);
51. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
53. stage.setScene(scene);
54. stage.show();
56. }
58. //Removes a user from the system
59. @FXML
60. void removeUser(ActionEvent event) {
61. User selectedUser = usersListView.getSelectionModel().getSelectedItem();
63. if(!(selectedUser == null)) {
65. a.setAlertType(AlertType.CONFIRMATION);
66. a.setHeaderText("Delete Account");
67. a.setContentText("Are you sure you want to remove this user?");
68. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);

71. // Waits for confirmation before deleting the account
72. if(a.showAndWait().get() == ButtonType.OK) {
73. usersListView.getItems().remove(selectedUser);
74. data.removeUser(selectedUser.getUsername());
76. a.setAlertType(AlertType.INFORMATION);
77. a.setHeaderText("Delete Account");
78. a.setContentText("Account Successfully Deleted");
79. a.show();
80. }
81. else {
82. a.close();
83. }
85. }
86. else {
87. a.setAlertType(AlertType.ERROR);
88. a.setContentText("Please select a user.");
89. a.setTitle("Invalid Input");
90. a.show();
91. }
93. }
95. @Override
96. public void initialize(URL arg0, ResourceBundle arg1) {
97. // Fetching all the users data
98. try {
99. usersListView.getItems().addAll(data.openAllUsers());
100. } catch (InputValidationException e) {
101. // TODO Auto-generated catch block
102. e.printStackTrace();
103. }
105. }
107. }

##### updatePasswordFXMLController.java

1. package GUI;
3. import java.io.IOException;
5. import Database.dataSQLite;
6. import javafx.event.ActionEvent;
7. import javafx.fxml.FXML;
8. import javafx.fxml.FXMLLoader;
9. import javafx.scene.Node;
10. import javafx.scene.Parent;
11. import javafx.scene.Scene;
12. import javafx.scene.control.Alert;
13. import javafx.scene.control.Button;
14. import javafx.scene.control.PasswordField;
15. import javafx.scene.control.TextField;
16. import javafx.scene.layout.Region;
17. import javafx.scene.control.Alert.AlertType;
18. import javafx.stage.Stage;
20. /\*\*
21. \*
22. \* Controller class for update password
23. \* This class is responsible for updating password if valid credentials are provided
24. \* @author Yogesh Parajuli
25. \*
26. \*/
27. public class updatePasswordFXMLController {
29. private dataSQLite data = dataSQLite.getInstance();
30. Alert a = new Alert(AlertType.NONE);


34. @FXML
35. private TextField usernameTextField;
37. @FXML
38. private PasswordField currentPasswordTextField;
40. @FXML
41. private Button updateButton;
43. @FXML
44. private Button cancelButton;
46. @FXML
47. private PasswordField newPasswordTextField;
49. @FXML
50. private PasswordField reNewPasswordTextField;
52. // clicking cancel button would take you back to login page
53. @FXML
54. void cancel(ActionEvent event) throws IOException {
55. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
57. Scene scene = new Scene(root);
59. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
61. stage.setScene(scene);
62. stage.show();
63. }
65. //Updates the password
66. @FXML
67. void updatePassword(ActionEvent event) throws IOException {
68. String username = usernameTextField.getText();
69. String curPassword = currentPasswordTextField.getText();
70. String newPassword = newPasswordTextField.getText();
71. String reNewPassword = reNewPasswordTextField.getText();
73. //Validates the details first
74. if(data.validate(username, curPassword)) {
76. //checks if the passwords matches
77. if(newPassword.equals(reNewPassword)) {
79. // checks for validation
80. if(newPassword.matches("^[a-zA-Z0-9]{6,20}$")) {

83. data.updatePassword(username, newPassword);
85. a.setAlertType(AlertType.CONFIRMATION);
86. a.setHeaderText("Successful");
87. a.setContentText("Password successfully updated");
88. a.show();

91. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
93. Scene scene = new Scene(root);
95. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
97. stage.setScene(scene);
98. stage.show();
99. }
100. else {

103. a.setAlertType(AlertType.ERROR);
104. a.setHeaderText("Invalid password");
105. a.setContentText("Please check the details and try again. This is caused due to password constrains. Refer to 'How to register?' section of the manual.");
106. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
107. a.show();
108. }

111. }
112. else {
113. a.setAlertType(AlertType.ERROR);
114. a.setHeaderText("Passwords do not match");
115. a.setContentText("Please try again");
116. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
117. a.show();
118. }
120. }
121. else {
122. a.setAlertType(AlertType.ERROR);
123. a.setHeaderText("Invalid username or password");
124. a.setContentText("Please check your credential and try again.");
125. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
126. a.show();
127. }
129. }
131. }

##### updateRestauratnFXMLController.java

1. package GUI;
3. import javafx.scene.control.Label;
4. import java.io.IOException;
5. //import java.net.URL;
6. //import java.util.ResourceBundle;
7. import java.net.URL;
8. import java.util.ResourceBundle;
10. import Database.dataSQLite;
11. import application.Restaurant;
12. import javafx.event.ActionEvent;
13. import javafx.fxml.FXML;
14. import javafx.fxml.FXMLLoader;
15. import javafx.fxml.Initializable;
16. //import javafx.fxml.Initializable;
17. import javafx.scene.Node;
18. import javafx.scene.Parent;
19. import javafx.scene.Scene;
20. import javafx.scene.control.Alert;
21. import javafx.scene.control.Alert.AlertType;
22. import javafx.scene.control.Button;
24. import javafx.scene.control.TextArea;
25. import javafx.scene.control.TextField;
26. import javafx.scene.layout.Region;
27. import javafx.stage.Stage;
29. /\*\*
30. \* Controller class for Update Restaurant
31. \* @author Yogesh Parajuli
32. \*
33. \*/
34. public class updateRestaurantFXMLController implements Initializable{
35. private dataSQLite data = dataSQLite.getInstance();
36. Alert a = new Alert(AlertType.NONE);
37. private Restaurant restaurant;

40. /\*\*
41. \* Getter for restaurant
42. \* @return restaurnat
43. \*/
44. public Restaurant getRestaurant() {
45. return restaurant;
46. }
48. /\*\*
49. \* Setter for restaurant
50. \* @param restaurant
51. \*/
52. public void setRestaurant(Restaurant restaurant){
53. this.restaurant = restaurant;
54. }

57. @FXML
58. private Label restaurantLabel ; // = new Label(getRestaurant().toString())

61. @FXML
62. private TextField addressTextField;
64. @FXML
65. private TextField contactTextField;
67. @FXML
68. private Button updateButton;
70. @FXML
71. private Button cancelButton;
73. @FXML
74. private TextArea descriptionTextArea;


78. //Takes you back to previous window
79. @FXML
80. void cancel(ActionEvent event) throws IOException {
81. // go back to the adminview thing
82. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));
84. Scene scene = new Scene(root);
86. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
88. stage.setScene(scene);
89. stage.show();
90. }
92. //Updates the details of the selected restaurant
93. @FXML
94. void updateRestaurant(ActionEvent event) throws IOException, NullPointerException{
95. String address = addressTextField.getText();
96. String contact = contactTextField.getText();
97. String restaurantDescription = descriptionTextArea.getText();
98. System.out.println(getRestaurant());
100. // checks that the the textfields are not null
101. if(!address.equals(null) && (!contact.equals(null) && (!restaurantDescription.equals(null)))) {
103. //checks for input validation
104. if(contact.matches("\\d{11}") && address.matches("^(\\d+)\\s(\\w+)\\s(\\w+)$") && restaurantDescription.matches("\\p{ASCII}{4,100}")) {


108. data.updateRestaurant(restaurant.getId(), contact, address, restaurantDescription);
110. a.setAlertType(AlertType.CONFIRMATION);
111. a.setContentText("Restaurant details successfully updated");
112. a.show();
114. // go back to the adminview fxml
115. Parent root = FXMLLoader.load(getClass().getResource("adminViewReviewEat.fxml"));
117. Scene scene = new Scene(root);
119. Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
121. stage.setScene(scene);
122. stage.show();
123. }
124. else {
125. a.setAlertType(AlertType.ERROR);
126. a.setHeaderText("Invalid Input");
127. a.setContentText("Please Check your input and try again. This is caused due to input validation error.");
128. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
129. a.show();
130. }
132. }
133. else {
134. a.setAlertType(AlertType.ERROR);
135. a.setHeaderText("Invalid Input");
136. a.setContentText("Empty values not accepted");
137. a.getDialogPane().setMinHeight(Region.USE\_PREF\_SIZE);
138. a.show();
139. }
141. }
143. @Override
144. public void initialize(URL arg0, ResourceBundle arg1) {
145. // TODO Auto-generated method stub
147. }

150. /\*\*
151. \* Method to set the label
152. \* @param currentRestaurant
153. \*/
154. public void setLabel(String currentRestaurant) {
155. restaurantLabel.setText(currentRestaurant);
156. }


160. }

##### userViewReveiwEatFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
5. import java.util.Collections;
6. import java.util.ResourceBundle;
7. import java.util.TreeSet;
9. import Database.dataSQLite;
10. import application.Restaurant;
12. import application.User;
13. import javafx.collections.FXCollections;
14. import javafx.collections.ObservableList;
15. import javafx.collections.transformation.FilteredList;
16. import javafx.event.ActionEvent;
17. import javafx.fxml.FXML;
18. import javafx.fxml.FXMLLoader;
19. import javafx.fxml.Initializable;
20. import javafx.scene.Node;
21. import javafx.scene.Parent;
22. import javafx.scene.Scene;
23. import javafx.scene.control.Button;
24. import javafx.scene.control.ListView;
25. import javafx.scene.control.MenuButton;
26. import javafx.scene.control.TextField;
27. import javafx.scene.control.Alert;
28. import javafx.scene.control.Alert.AlertType;
29. import javafx.stage.Stage;

32. /\*\*
33. \* Controller Class for user View Review Eat
34. \* @author Yogesh Parajuli
35. \*
36. \*/
37. public class userViewReviewEatFXMLController implements Initializable{
38. //private dataSQLite data = dataSQLite.getInstance();
39. private dataSQLite data =dataSQLite.getInstance();
40. private static User currentUser;
41. Restaurant selectedRestaurant;
42. Alert a = new Alert(AlertType.NONE);


46. /\*\*
47. \* Getter for current user
48. \* @return current user
49. \*/
50. public User getCurrentUser() {
51. return currentUser;
52. }
54. /\*\*
55. \* Setter for user
56. \* @param user
57. \*/
58. public void setUser(User user) {
59. userViewReviewEatFXMLController.currentUser = user;
60. }
62. @FXML
63. private TextField searchTextField;
65. @FXML
66. private Button viewReviewButton;
68. @FXML
69. private ListView<Restaurant> restaurantListView;
70. private ObservableList <Restaurant> restData = FXCollections.observableArrayList();


74. @FXML
75. private MenuButton menuButton;
77. @FXML
78. private MenuButton menuButton1;
80. @FXML
81. private Button addReveiwButton;
83. @FXML
84. private Button logOutButton;


88. @FXML
89. void addButton(ActionEvent event) throws IOException {
90. selectedRestaurant = restaurantListView.getSelectionModel().getSelectedItem();
92. if (selectedRestaurant == null) {
93. a.setAlertType(AlertType.ERROR);
94. a.setContentText("Restaurant not selected");
95. a.setHeaderText("Please select a restaurnt");
96. a.show();
97. }
98. else {
100. // Change the parent i.e base class using fxml loader

103. FXMLLoader loader = new FXMLLoader(getClass().getResource("addReviewFXML.fxml"));
104. Parent root = loader.load();
106. addReviewFXMLController nextController = loader.getController();
108. // here i am trying to pass user object to another controller
109. nextController.setCurrentUser(getCurrentUser());
110. nextController.setSelectedRestaurant(selectedRestaurant);
112. //Set the scene to the new root i.e new fxml file
113. Scene scene = new Scene(root);
115. //Set the stage information
116. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
118. stage.setScene(scene);
119. stage.show();
120. }
121. }


125. @Override
126. public void initialize(URL arg0, ResourceBundle arg1) {
127. TreeSet <Restaurant> restaurants = data.openRestaurants();
128. for(Restaurant r: restaurants) {
129. r.setReviews(data.openReview(r.getId()));
131. r.getAverageRating();
132. }
134. restData.addAll(restaurants);

137. FilteredList <Restaurant> filteredData = new FilteredList<>(restData, b-> true);
138. // 2. Set the filter Predicate whenever the filter changes.
139. searchTextField.textProperty().addListener((observable, oldValue, newValue) -> {
140. filteredData.setPredicate(restaurant -> {
141. // If filter text is empty, display all persons.
143. if (newValue == null || newValue.isEmpty()) {
144. return true;
145. }
147. // Compare restaurant's name with filter text.
148. String lowerCaseFilter = newValue.toLowerCase();
150. if (restaurant.getName().toLowerCase().indexOf(lowerCaseFilter) != -1 ) {
151. return true; // Filter matches
152. }
153. else
154. return false; // Does not match.
155. });
156. });
158. restaurantListView.setItems(filteredData);


162. }
164. @FXML
165. void defaultFilter(ActionEvent event) {
166. filter("all");

169. }
171. @FXML
172. void britishFilter(ActionEvent event) {
173. filter("British");
174. }
176. @FXML
177. void chineseFilter(ActionEvent event) {
178. filter("Chinese");
179. }
181. @FXML
182. void frenchFilter(ActionEvent event) {
183. filter("French");
184. }
186. @FXML
187. void indianFilter(ActionEvent event) {
188. filter("Indian");
189. }
191. @FXML
192. void italianFilter(ActionEvent event) {
193. filter("Italian");
194. }
196. @FXML
197. void jamaicanFilter(ActionEvent event) {
198. filter("Jamaican");
199. }
201. /\*\*
202. \* Method to filer the restaurant based on cuisine
203. \* We use lamda expression to get all the restaurants based on selected cuisine
204. \* @param cuisine
205. \*/
206. public void filter(String cuisine) {
207. restaurantListView.getItems().removeAll();
208. FilteredList <Restaurant> filteredData = new FilteredList<>(restData, b-> true);
210. //Lamda expression to match to the cuisines of the restaurants
211. filteredData.setPredicate(restaurant ->{
212. if (restaurant.getCuisine().equals(cuisine)) {
213. return true;
214. }
215. else if(cuisine.equals("all")) {
216. return true;
217. }
218. else return false;
220. });
222. //lamda expression for the search box
223. searchTextField.textProperty().addListener((observable, oldValue, newValue) -> {
224. filteredData.setPredicate(restaurant -> {
225. // If filter text is empty, display all values
227. if (newValue == null || newValue.isEmpty()) {
228. return true;
229. }
230. String lowerCaseFilter = newValue.toLowerCase();
232. if (restaurant.getName().toLowerCase().indexOf(lowerCaseFilter) != -1 ) {
233. return true; // Filter matches
234. }
235. else
236. return false; // Does not match.
237. });
238. });
240. restaurantListView.setItems(filteredData);
241. }
243. //Logs you out
244. @FXML
245. void logOut(ActionEvent event) throws IOException {

248. Parent root = FXMLLoader.load(getClass().getResource("loginFXML.fxml"));
250. //Set the scene to the new root i.e new fxml file
251. Scene scene = new Scene(root);
253. //Set the stage information
254. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
256. stage.setScene(scene);
257. stage.show();
258. }


262. @FXML
263. void sortByRating(ActionEvent event) {
264. sort("rating");
266. }
268. @FXML
269. void defaultSort(ActionEvent event) {
270. sort("default");
272. }
274. /\*\*
275. \* Method to sort the restaurants
276. \* Here we are using the lamda expression to manage the comparision of restaurant bases on given parameter
277. \* also making sure that we can still use the search box as well
278. \* searching uses lambda expression as well
279. \* @param str
280. \*/
281. public void sort(String str) {
282. restaurantListView.getItems().removeAll();
283. //restData.removeAll();


287. if (str.equals("rating")){
288. restaurantListView.setItems(null);

291. Collections.sort(restData);
293. FilteredList <Restaurant> filteredData = new FilteredList<>(restData, b-> true);
295. // Uses lamda expression to compare
296. searchTextField.textProperty().addListener((observable, oldValue, newValue) -> {
297. filteredData.setPredicate(restaurant -> {
298. // If filter text is empty, display all data
300. if (newValue == null || newValue.isEmpty()) {
301. return true;
302. }
304. // matches the characters input in the search box
305. String lowerCaseFilter = newValue.toLowerCase();
307. if (restaurant.getName().toLowerCase().indexOf(lowerCaseFilter) != -1 ) {
308. return true; // Filter matches first name.
309. }
310. else
311. return false; // Does not match.
312. });
313. });
315. restaurantListView.setItems(filteredData);
316. }
317. else {
318. restaurantListView.setItems(null);
320. ObservableList <Restaurant> newRestData = FXCollections.observableArrayList();
322. TreeSet <Restaurant> restaurants = data.openRestaurants();
323. for(Restaurant r: restaurants) {
324. r.setReviews(data.openReview(r.getId()));
326. r.getAverageRating();
327. }
329. //restData.addAll(restaurants);
330. newRestData.addAll(restaurants);

333. FilteredList <Restaurant> filteredData = new FilteredList<>(newRestData, b-> true);

336. searchTextField.textProperty().addListener((observable, oldValue, newValue) -> {
337. filteredData.setPredicate(restaurant -> {

340. if (newValue == null || newValue.isEmpty()) {
341. return true;
342. }

345. String lowerCaseFilter = newValue.toLowerCase();
347. if (restaurant.getName().toLowerCase().indexOf(lowerCaseFilter) != -1 ) {
348. return true; // Filter matches .
349. }
350. else
351. return false; // Does not match.
352. });
353. });
355. restaurantListView.setItems(filteredData);
356. }
357. }
359. //Method to allow users to view all reviews of the selected Restaurant
360. @FXML
361. void viewReview(ActionEvent event) throws IOException {
362. selectedRestaurant = restaurantListView.getSelectionModel().getSelectedItem();
364. if (selectedRestaurant == null) {
365. a.setAlertType(AlertType.ERROR);
366. a.setContentText("Restaurant not selected");
367. a.setHeaderText("Please select a restaurnt");
368. a.show();
369. }
370. else {
372. // Change the parent i.e base class using fxml loader
373. FXMLLoader loader= new FXMLLoader(getClass().getResource("viewReviewsFXML.fxml"));
374. Parent root = loader.load();

377. viewReviewsFXMLController nextController = loader.getController();
379. // here i am trying to pass user object to another controller
381. nextController.setLabel(selectedRestaurant.toString());
382. nextController.setListView(selectedRestaurant.getReviews().values());
384. //Set the scene to the new root i.e new fxml file
385. Scene scene = new Scene(root);
387. //Set the stage information
388. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
390. stage.setScene(scene);
391. stage.show();


395. }


399. }
400. }

##### viewReviewsFXMLController.java

1. package GUI;
3. import java.io.IOException;
4. import java.net.URL;
5. import java.util.Collection;
7. import java.util.ResourceBundle;

10. import application.Review;
11. import javafx.event.ActionEvent;
12. import javafx.fxml.FXML;
13. import javafx.fxml.FXMLLoader;
14. import javafx.fxml.Initializable;
15. import javafx.scene.Node;
16. import javafx.scene.Parent;
17. import javafx.scene.Scene;
18. import javafx.scene.control.Button;
19. import javafx.scene.control.Label;
20. import javafx.scene.control.ListView;
21. import javafx.stage.Stage;
23. /\*\*
24. \* Controller class to allow users to view reviews.
25. \* @author Yogesh Parajuli
26. \*
27. \*/
28. public class viewReviewsFXMLController implements Initializable{

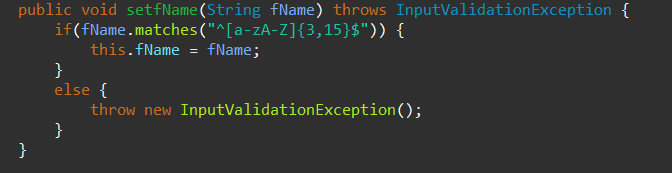

32. @FXML
33. private Label restaurantLabel;
35. @FXML
36. private ListView<Review> reviewListView;
38. @FXML
39. private Button backButton;
41. //Takes you back to previous page
42. @FXML
43. void back(ActionEvent event) throws IOException {
44. Parent root = FXMLLoader.load(getClass().getResource("userViewReviewEatFXML.fxml"));
46. //Set the scene to the new root i.e new fxml file
47. Scene scene = new Scene(root);
49. //Set the stage information
50. Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
52. stage.setScene(scene);
53. stage.show();
54. }
56. @Override
57. public void initialize(URL arg0, ResourceBundle arg1) {
58. // TODO Auto-generated method stub
60. // reviewListView.getItems().setAll(data.openReview(restaurant.getId()).values());
62. }
64. /\*\*
65. \* Method to set the label
66. \* @param restaurant
67. \*/
68. public void setLabel(String restaurant) {
69. restaurantLabel.setText(restaurant);
70. }
72. /\*\*
73. \* Method to initialize the List View
74. \* @param collection
75. \*/
76. public void setListView(Collection<Review> collection) {
77. reviewListView.getItems().setAll(collection);
79. }
81. }

## Error Handling

Errors have been handled appropriately in the program to prevent any errors and to make the software more robust. Here is a list of potential errors that could occur in our program and some example of how they are handled.

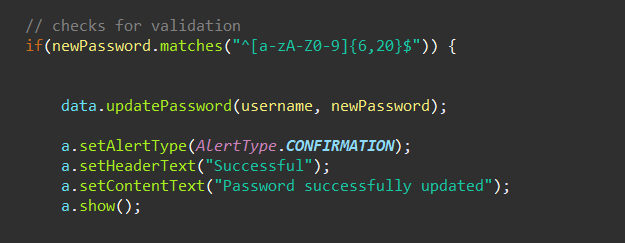
* **Use of Regex**

Regex or Regular Expressions can be used in java to match input patterns. I have used regex to match pattern for all the inputs that will be entered by a user or an admin. For example:



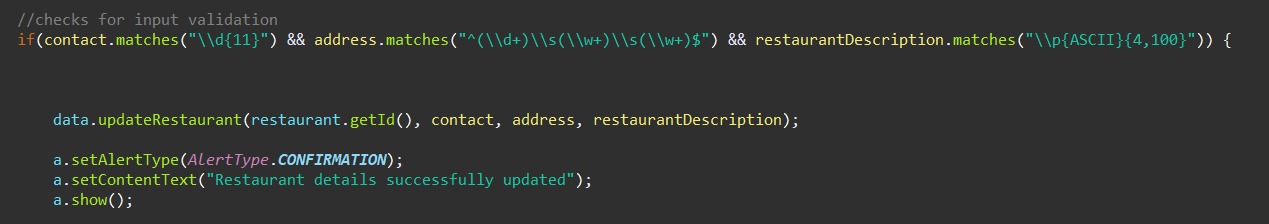
Here, the regex checks if fName contains only letters and is 3 to 15 characters long. If the pattern matches, then the code runs well but if it does not then it throws an exception error.

Furthermore, I have made use of regex for updating password and updating restaurant details as well. The reason for this is that we not updating information of an object rather it is being done in a database. So, to prevent errors here, I have made use of regex used in SoftwareUser class and Restaurant class. For example:

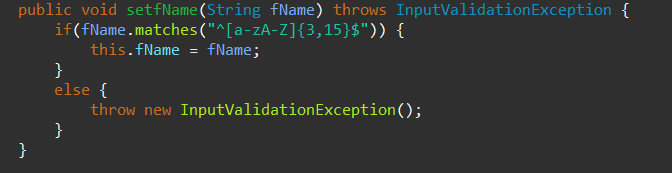


Here, the regex is same as the one that is used on SoftwareUsers class to match a pattern for a password.

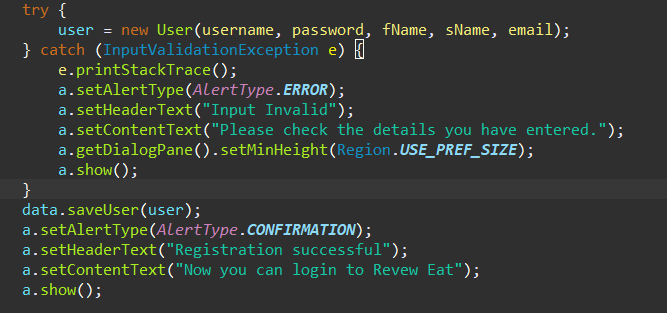
Likewise, for updating restaurant details following regex has been used:



* **Input Validation Exception**
* In order to validate the inputs from the user, a dedicated class was created called InputValidatationException. For every stage where input from the user had to be validated, this validation was used. Here is an example of when Input validation exception was used to validate the input:



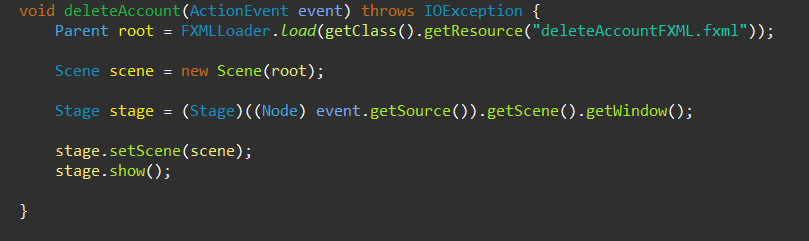
In the above-mentioned example, I have used an if clause to match the pattern for an input and if the input is not met, an exception error is thrown. Similar concept in the controller class has been implemented in this way:



Here, a try-catch block is used instead of throw statement. The code in try section is run and if there is an inputValidationException error then the code in catch section is run.

* **IO Exception**

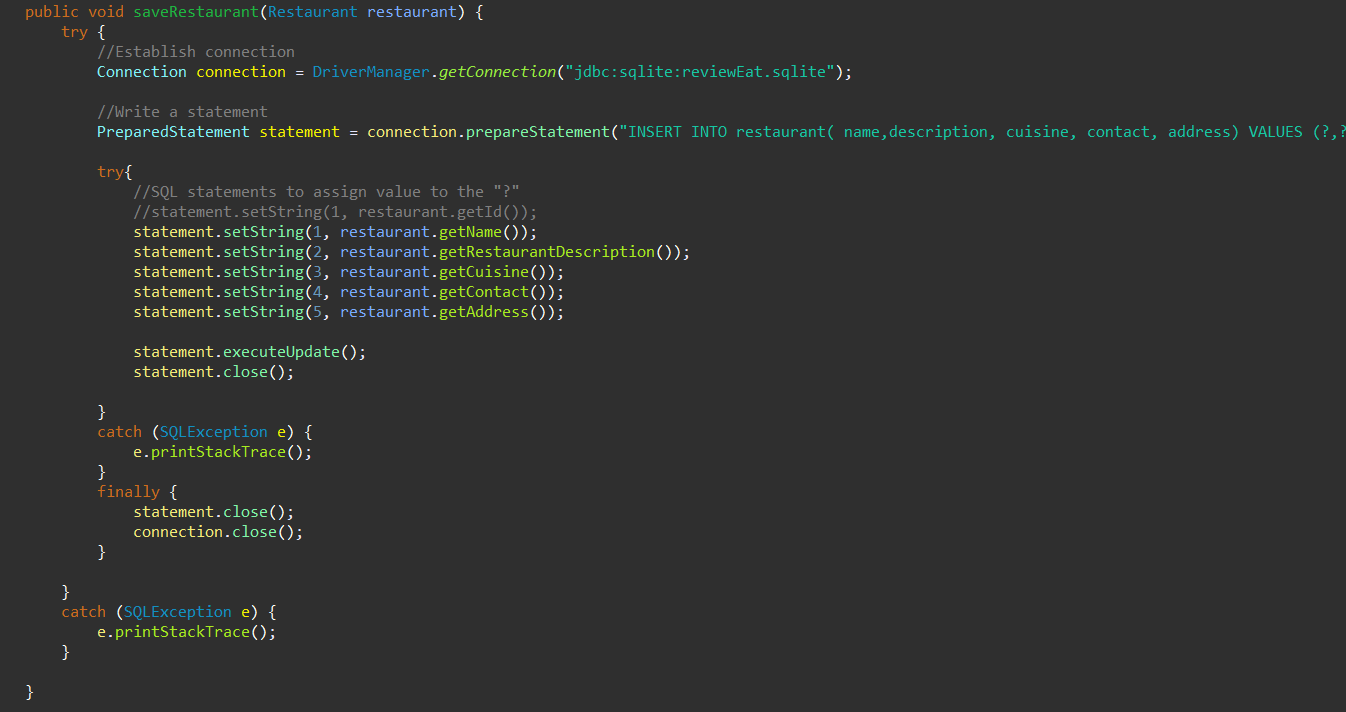
IO Exceptions in java is used to detect failure in an input or output operation. In my project, IO Exceptions are used by the controller classes when change of scene is necessary. For example:



Above mentioned method is supposed to change the window to “deleteAccountFXML.fxml”. But if that file is not found, then the program throws an IO Exception

* **SQL Exception**

SQL Exception in java is thrown when there is an error in the database i.e. for accessing database and others. The SQL Exceptions handled in following way:

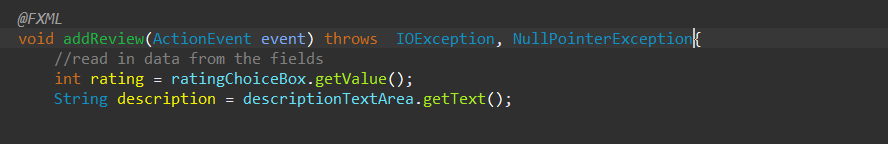


The code is from dataSQLite class which consists of all the SQLite commands. I have used try catch blocks in the code where if the program throws any SQL errors, it is handled by the catch block.

* **Null Pointer Exception**

Null Pointer Exception is thrown when object of null value is attempted to be referenced. In my program, the controller class

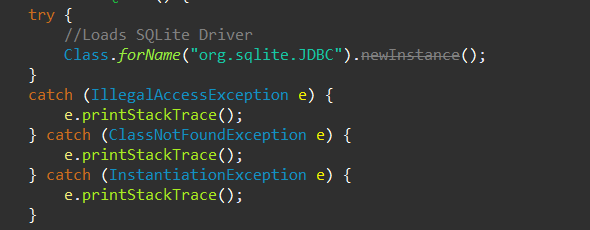
Null Pointer Exception is thrown when object of null value is attempted to be referenced. In my program, the controller classes pass on data between each other (i.e. passes an object reference across). This can possibly trigger a Null Pointer Exception. For example:



This is a piece of code from a add review controller class. A user object has been passed onto this class from another class and is used by the addReview method. Hence, I have used throws NullPointerException in this case.

* **Illegal Access Exception, Class Not Found Exception, Instantiation Exception**

These are the exceptions that are thrown when an SQLite Database file is created, and I have handled it in the following manner by using try/catch blocks:



## GUI and Events Handling

JavaFX has been used to develop the Graphical User Interface for the project. I have made use of **FXML** which is an XML based markup language for defining user interface in Java applications. [[1]](#footnote-2)I have made use of **Scene Builder** [[2]](#footnote-3)to define the user interface. Scene Builder is a software that allows rapid application development by providing drag and drop feature for building a user interface.

FXML files contains the script for defining the user interface and Scene Builder allows to manipulate the FXML files graphically. This procedure was used during the development of the project. Each FXML files are then controlled by another java class. Those files are called controllers. I have a controller for each of the FXML files in the project.

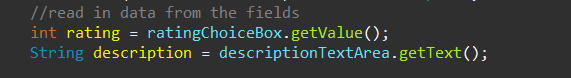
There are interactable and non-interactable GUI elements that are used in the software. This section of document will describe interactable/dynamic GUI elements that are used in the software and logic used behind it.

### Text Area

Text Area shares the functionality with text box. The only difference is that the text area can take in multiple lines of input while text area only takes single line of input. However, the code behind it handles the information in similar manner to text box.



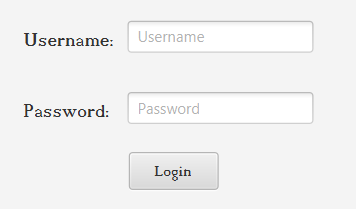
This is an example of a text area being used. The code that handled the input from description is as follows:



Here, we are just assigning the input from the text area to a variable.

### Text Box

Text Box in Review Eat has been used to get information from the users. For example, text box has been used in login page to get username and password from the user.



This is a screen shot of login page. Behind the text box, simple instructions have been used. The software reads the text entered in the text box and it sets the value as a variable which is then used for validations.



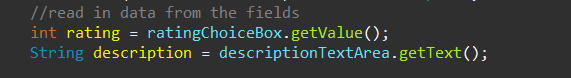
This is a piece of code which sets the value read from the text box to different variables. Also, all the text box used the software has some text written onto them as a guidance to the user of what type of information should be entered in that specific text box.

### Choice Box

The choice box is the GUI element that allows user to select an item from a list. The list is predefined. In Review Eat, choice box has been implemented to get input from the user. For example:

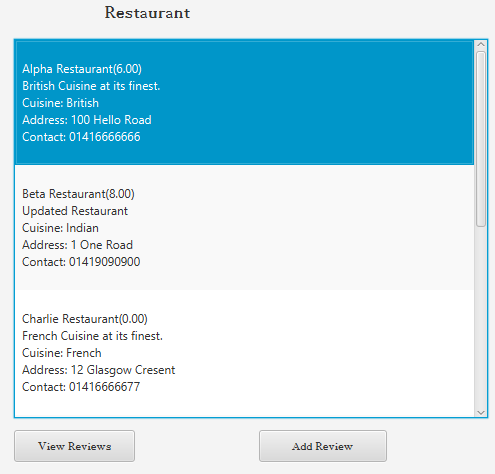


Here, a choice box is used to get a rating from the user. This is an example of taking an input from the user. Like text boxes, this input is assigned to some variable. But unlike text boxes, choice box limits the input i.e. only items listed can be chosen by the user.

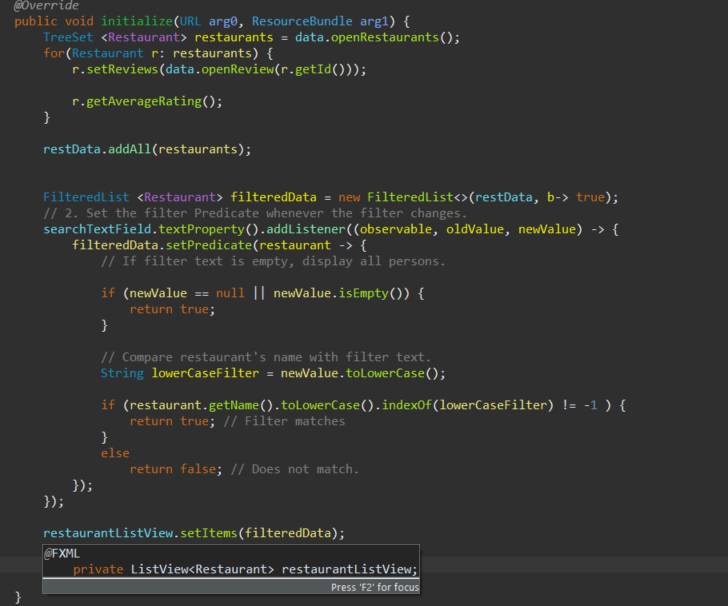


### List View

List View is one of the elements provided by Java FX which allows you to display data in a list format. The list is scrollable. Also, each item in the list can be selected as well. List views have in been used in Review Eat in various windows. The major one being on the user view window where restaurants are listed. The restaurants are selectable. Here is an example of list view where items are selectable:



The item highlighted on blue is selected item. The selection could be tracked, and the item could be used for other purposes.



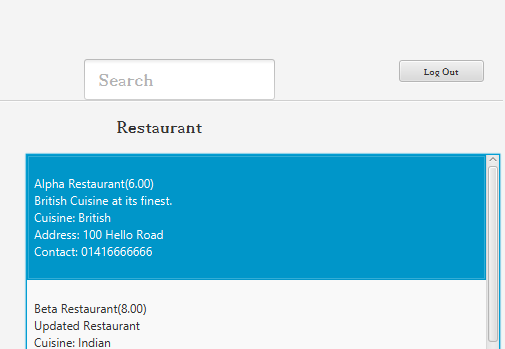
In above mentioned code, “restaurantListView” represents the list view itself. We are reading the values from a database (data for restaurant) and passing the data through a code which allows search functionality (mentioned in next section). Then, the list view is populated with that data and it is viewable. This piece of code is run when user view window is loaded therefore, this code is executed every time the page is loaded.



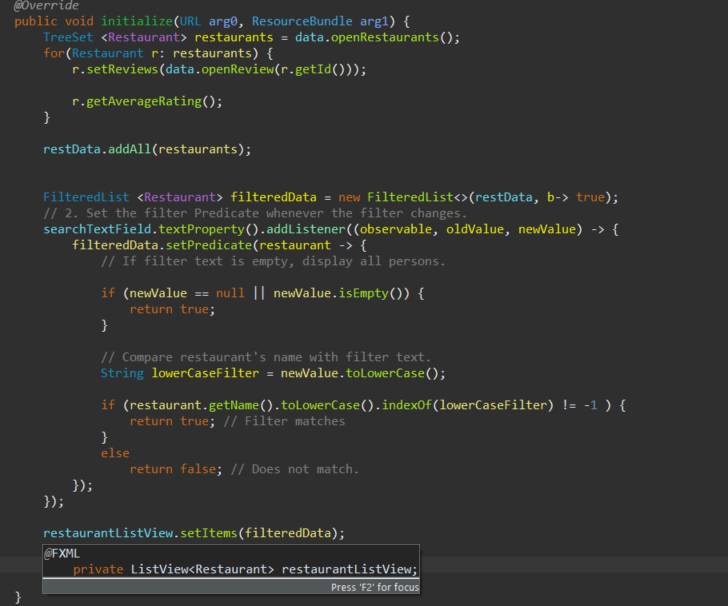
This piece of code is responsible for keeping track of selected item. So, when an item is selected from a list, it is assigned to a variable. The variable/ data in the variable is either passed onto another window or for something else.

### Search Box

User view page of my project features a search box which is basically is a text box that is implemented in a special manner. As the name suggests, this text box is meant to search for items in the list view. Like normal text boxes, a user can type characters and the search box basically keeps track of each character that is typed into it and then it tries to find an item in the list and returns it. The returned list is empty if the characters entered by the user does not matches with any items in the list.



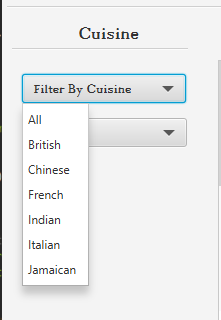
The search box has the word “Search” written into it.



Search box is defined by “searchTextField”. Here after getting data for the restaurant from the database, we have added an event listener using lambda expression. The event listener keeps track of any data input in the search box. If a user starts typing something in the search box, the search box immediately runs the above-mentioned code. The code basically keeps track of values in the search box before and after the something has been typed into it. Then it checks for matches with restaurant’s name. If there are matches, then the list (i.e. items in the list view) gets updated with the matching list. And, if no matches are found then empty list is returned.

### Menu Buttons

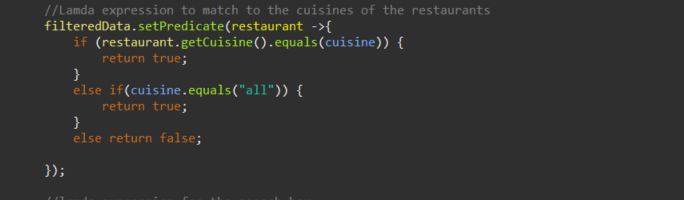
Menu buttons are another GUI element that is used in the Review Eat GUI. Menu buttons has a title attached to it and it has various options. Each option is then assigned with a function to perform a task. There are a couple of menu buttons that are used. One of them is a menu button titled “Filter By Cuisine”.



The menu button looks like a choice box of instead of taking standard inputs from a list, menu buttons allows user to choose an option to perform a function. The function itself is identified by the title of menu button. Each function is assigned with different function. For example:



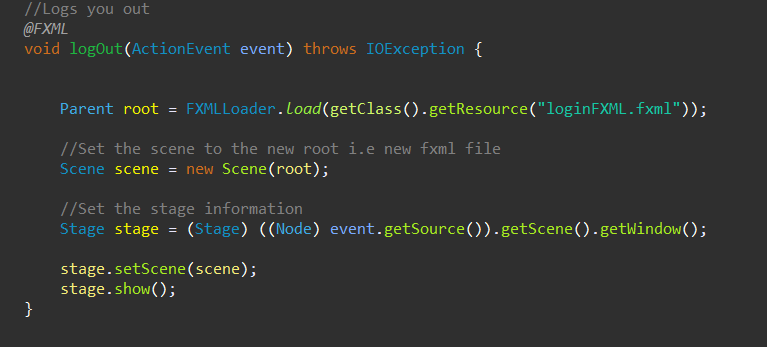
Each option of our GUI element is attached to a function.



This is the code that really deals with what each option does. So, based on option chosen by a user, the code looks for matching “cuisine” and returns the restaurant that has selected cuisine.

### Buttons

Buttons are another GUI element that is used for the project. Buttons in Review Eat has been used for a couple of different purposes. The first one is navigation. The navigational buttons include “Cancel” button, “Back” button, “Add Restaurant” button (on Admin view page), “Add Review” button (on User view page), “Log Out” button and others. All these buttons navigate you to a different page. Similarly, buttons are also used for submitting information. This includes “Login” button, “Add Review” button (on Add Review page), “Add Restaurant” button (on Add Restaurant page) and others. The events are handled differently to accomplish these features in the software. Following is an example of code behind navigational buttons.



This is function attached to log out button. The logic behind the code is quite simple. All we are doing is changing the current scene to another by loading another FXML file. In this case the log out button changing the scene to “loginFXML.fxml” file.

Here is an example of a button that is used to submit data:



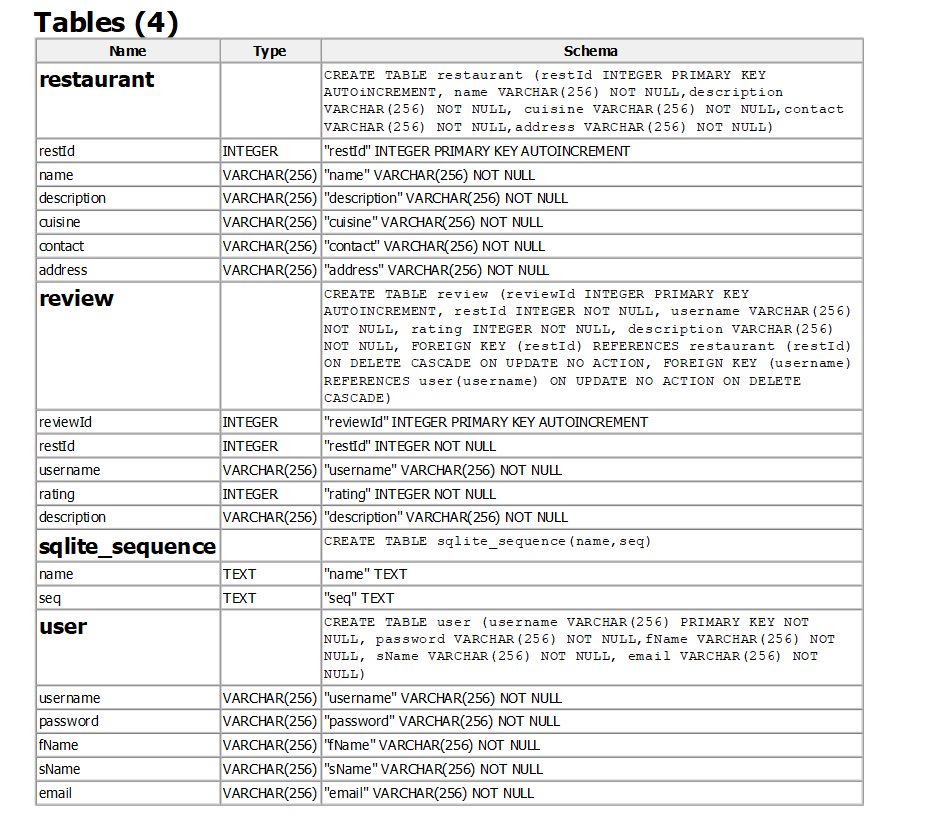
This is a method to handle event of “Login” button from login page. Login page contains two text boxes where user can enter their username and password. As mentioned before, the input read from the text boxes are assigned to a variable, but they are only assigned to the variables once a button is pressed. So, in this instance of login page, the data read from the text boxes are assigned to a variable, the variables are checked with data in database. If the inputs are valid then we set the user as current user. Then we check if they have admin privilege. If they do not, then “userViewReviewEatFXML” (an FXML file) is opened. Also, the data for current user is also passed onto controller of “userViewReviewEatFXML” where it is used if the user wants to add a review.

## Database and CRUD Operations

An SQLite database is used as data storage for our program. The database designed was mentioned in the planning stage, this part of documentation includes an updated ERD. Also, the code behind some CRUD operations are explained below.

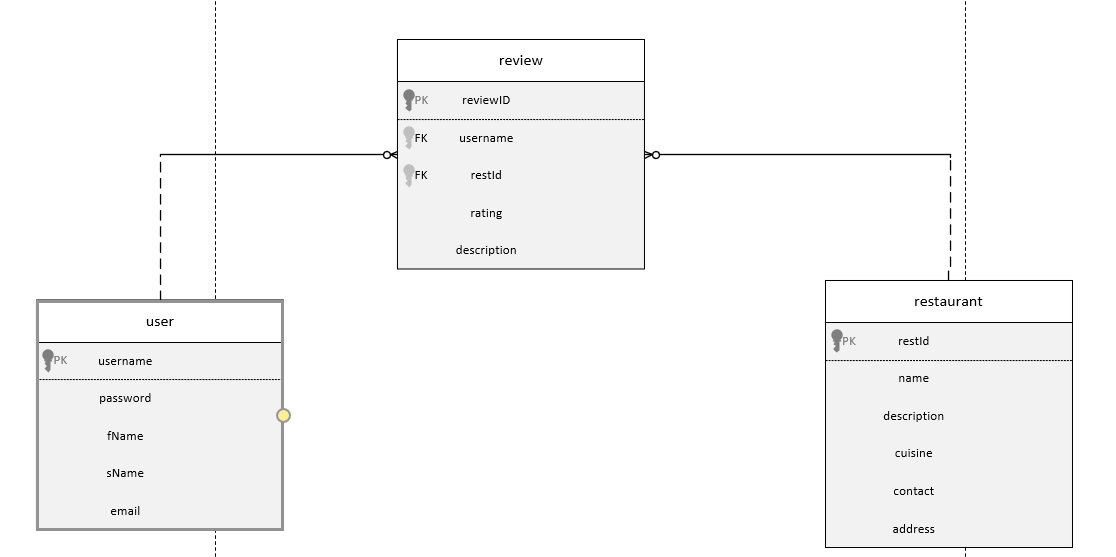
### Database and ERD

Following entities are used for the software:



The “sqlite\_sequence” is auto generated by the database itself. The table that are used are restaurant (to stored restaurant details), review (to store review) and user (to store user’s details)

The updated ERD is as follows:



### CRUD Operation

The dataSQLite class deals the database side of the program. This section of the documentation contains a detailed description of dataSQLite class and methods used in this class to achieve CRUD.

The dataSQLite class is responsible for establishing a connection with the database, creating tables if necessary and performing CRUD operations. It has been implemented using a singleton design pattern. Singleton pattern allows us to ensure that only a single object can be instantiated from the class. It is crucial because if there were more than one instance of dataSQLite class, this would mean that there would be more than one database which is not ideal.

To begin with, dataSQLite class establishes a connection with jdbc to create a .sqlite file. When a connection is established, the class uses prepared statements to create necessary tables if they do not already exist. The entities are defined along with primary and foreign keys. This is all done by the constructor of dataSQLite class. It uses SQLite commands to perform CRUD operations.

Furthermore, prepared statements are used for every operation is the class. Likewise, parametrised statements where question marks are uses as place holders. Using the place holders allows caching benefits.

Here is an overview of methods in dataSQLite class and their functionality:

* saveRestaurant

Parameters: restaurant (Restaurant object)

Output: --

Explanation: This method uses INSERT command on the restaurant table and saves information on the database. It takes the restaurant object and uses a prepared statement to insert values to attribute of a restaurant table.

* saveReview

Parameters: user (User object), restaurant (Restaurant object), review (Review object)

Output: --

Explanation: This method uses INSERT command on the review table to save the review. It gets username from the User object, restaurant id from Restaurant object, and Review object and populates them on the review table.

* saveUser

Parameters: user (User object)

Output: --

Explanation: Like saveRestaurant and saveReview, this method also uses an INSERT command on the user table to save information about a user to the system. The method extracts information from the user object and populates them on user table.

* checkUsername

Parameters: username (String)

Output: Boolean

Explanation: This method is to check if another user has already taken a username. It makes use of SELECT command to select username from the user table. If the username exists in the system, then, it returns true, and if it does not, then it would return false.

* checkReview

Parameters: username (String), restId (int)

Output: Boolean

Explanation: As checkUsername, checkReview method also makes use of SELECT command to select a review using a username and restaurant if (restId). It checks if the user has already added a review to a restaurant. The system does not allow a user to add a review on a restaurant twice.

* Validate

Parameters: username (String), password (String)

Output: Boolean

Explanation: This is the method used to check if the username and password entered by the user match with the data in the database. It makes use of SELECT statement on user table to select a row with matching username and password, and if the row is found it returns true, else, it returns false.

* openUser

Parameter: username (String)

Output: user (User object)

Explanation: This method is used to retrieve information about a user. It makes use of SELECT DISTINCT command on the user table to find the information about the user, creates a user object and returns the user object. It returns null if no match is found.

* openAllUsers

Parameter: --

Output: ArrayList <User>

Explanation: openAllUsers is a method to retrieve information for all users rather than just one. It uses SELECT command to select all data from the user table. An ArrayList is created, and data retrieved from the database gets added to that ArrayList as User objects. Then, it returns the ArrayList.

* openRestaurants

Parameter: --

Output: TreeSet <Restaurant>

Explanation: Like openAllUsers, the purpose of this method is to retrieve information about all the restaurants. It uses SELECT command to select all data from the restaurant table. A treeset has been created, and data from the database gets populated into the treeset as Restaurant object. Then, it returns the treeset.

* openReview

Parameter: --

Output: HashMap<String, Review>

Explanation: Each restaurant object contains a hashmap with username as the key and Review object as its value. So, when we open restaurant, we need to open reviews for those restaurants as well. This method does just that. Again, it uses SELECT command to get information from the database, populates it to a hashmap and returns the hashmap.

* removeUser

Parameter: username (String)

Output: --

Explanation: removeUser method removes the user from the database. It uses DELETE command. It looks for the username provided in the database, and when found, that row of data is then deleted.

* removeRestaurant

Parameter: restaurant (Restaurant object)

Output: --

Explanation: This method is used to delete information about restaurant. Again, it uses DELETE command and works with the same principle as removeUser.

* removeReview

Parameter: username (Username)

Output: --

Explanation: This method has been created to brute force method to delete review when a user is deleted. The tables were set up such when a user is deleted, any data from related tables would also be deleted. But during the test process, I noticed that the program was not working as I wanted it to. Therefore, this method is supposed to delete any reviews from the review table, which contain the given username. It uses DELETE command on review table.

* removeReview

Parameter: restaurant (Restaurant)

Output: --

Explanation: This method is the same as the previous, but instead of a username, this method deletes the reviews using the restaurant id of provided Restaurant object.

* updatePassword

Parameter: username (String), newPassword (String)

Output: --

Explanation: This method updates a user’s password with the new one. It takes in username which is used to search for the user in the database. Here, we use UPDATE and SET command with some constraints (uses WHERE command for condition). The row of data for the user is searched using while command and when found, the password is updated to newPassword (String).

* updateRestaurant

Parameter: restId (int), contact (String), address (String), description (String)

Output: --

Explanation: Like updatePassword method, this method also updates information in the database. It uses UPDATE and SET command along with WHILE command. The WHERE command is used to find a row of matching data, and when found, its columns are updated with the provided information.

\****NOTE: All operations requiring to check for a match in the rows uses WHERE command to find the matching data.***

1. <http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf> [↑](#footnote-ref-2)
2. <https://gluonhq.com/products/scene-builder/> [↑](#footnote-ref-3)