FoodLocator

By CSCI201 Team 7

Olivia Liau, Laila Gayden, Bona Suh,

Siiri Haapasalo, Yougendy Mauricette,

Joyce Zhou, Abhijay Sharma, Jason Byun

Table of Contents

Project Proposal

Concept: Food Locator and Rater

Our team is interested in pursuing a Food Locator and Rater application for the various food places available on campus, designed to help other Trojans choose where to eat. The app will use the user's geographical location to display nearby food places on campus, along with their corresponding ratings given by fellow Trojans.

High-Level Requirements

We need to create an application that helps USC students find and rate food places on campus. The app should use the student's location on a map and show nearby dining options along with their names, hours, and ratings from other students. Users should be able to leave their own ratings and reviews, edit or delete them, and search for food by name or type. The app should display results on a map or in a list and clearly show which places are open or closed. It should work smoothly on both phones and computers and let students find and rate a food place quickly and easily.

Technical Specifications

I - Login System (12 hours)

- Implement authentication and authorization with Google SSO restricted to USC domains (i.e usc.edu)
- Restricted access to not logged in users, they will be unable to add reviews, view only

II - Map System (20 hours)

- Main screen centered on map of USC Campus and USC village
- Integrate interactive map
- Option for locating user
- Add pins for food locations
- Clicking a pin opens a modal with
    - Location name, average rating, distance, category type
    - Top 3 reviews
    - Buttons for View All Reviews, Write Review, Get Directions

III - Review Screen (18 hours)

- Modal showing all reviews for a selected location
- Display review list with pagination, sorting (Most Recent, Highest Rated, Most Useful)
- Users can write and submit new reviews with
    - Star rating system
    - Text input with inline validation
    - Tags (“vegan”, “cheap”, “quick service”)
- Implement upvote/downvote or “Mark as Helpful”

IV - Database Architecture (16 hours)

- MySQL as the relational database
- Core Tables
    - Users (userID, email, name)
    - Locations (locationID, name, address, category)
    - Reviews (reviewID, locationID, userID, rating, title, body createdAt)

Detailed Design

Overview

This section of the document describes the detailed design for a food locator web interface. The application is designed to allow USC students to log in and view all the food options available on campus and at the village, accompanied by reviews left by other Trojans. The backend will be implemented using Java with a MySQL database, and the front end will be built using HTML and CSS. Hardware requirements are simply any computer capable of running Chrome browser, Eclipse, and MySQL.

Authentication

Workflow:

1. User opens the application, can either click “Log in” or “Sign up”
2. Users either sign up with their own accounts and passwords. Will not support outside account authentication like Google.
3. After a successful login, the app redirects the user back to the app front page.

All user data will be stored in the database table named users. Log-in information will be compared against the data saved there.

User versus guest:
Both a guest and user will be able to see all the restaurant information. A user will be able to write reviews. A guest is basically view-only while a user can add information.

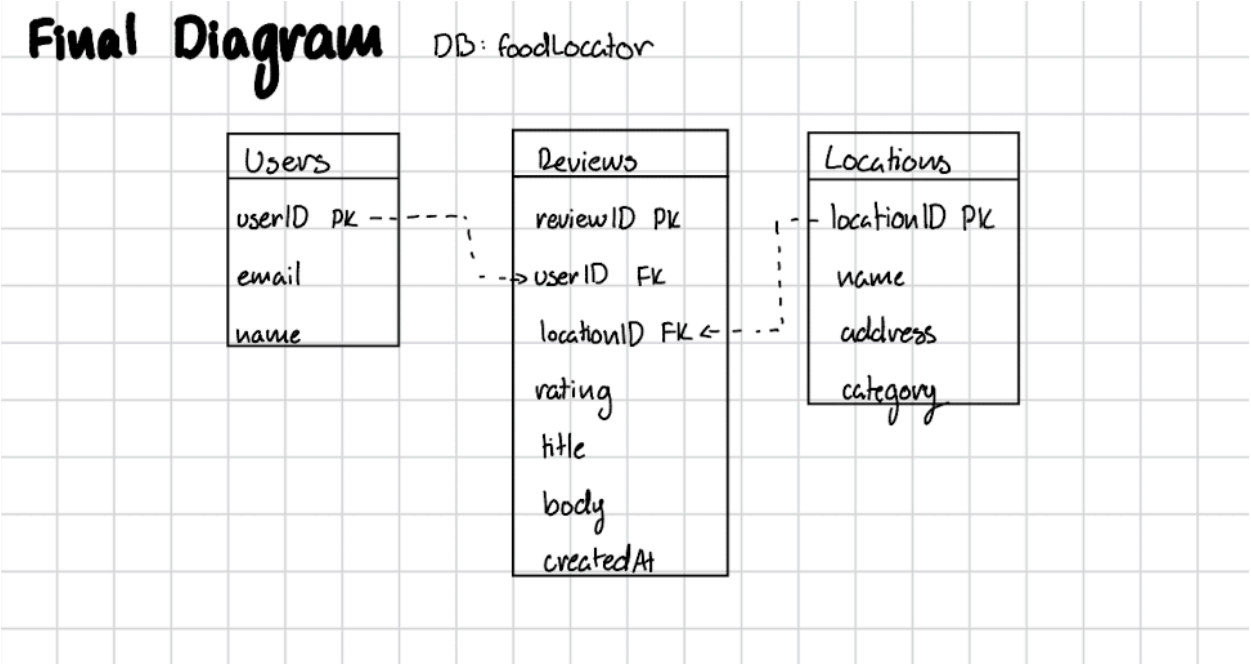
Mockups:

Database

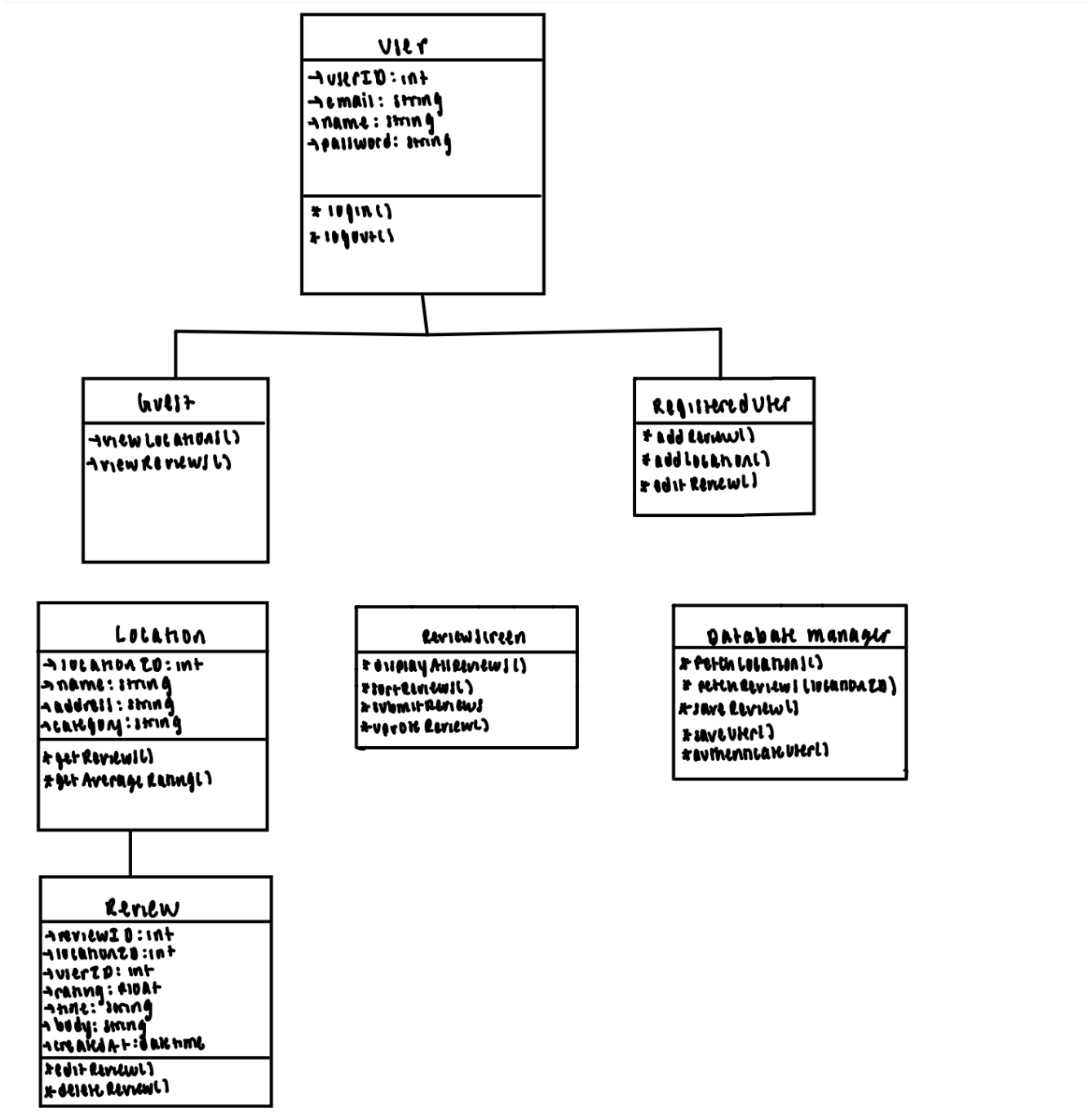
The database will be on MySQL, and it will be named foodLocator.
Tables:

- Users
    - userID, (INT AUTO_INCREMENT PRIMARY KEY)
    - email, (VARCHAR(225) UNIQUE NOT NULL)
    - name, (VARCHAR(225) NOT NULL)
    - password, (VARCHAR(225) NOT NULL)
- Reviews
    - reviewID (INT AUTO_INCREMENT PRIMARY KEY)
    - locationID (INT NOT NULL) ← Foreign key
    - userID (INT NOT NULL) ← Foreign key
    - rating (DECIMAL(2,1) CHECK (rating >= 0 AND rating <= 5))
    - title (VARCHAR(225) NOT NULL)
    - body (VARCHAR(2000) NOT NULL)
    - createdAt (DATETIME DEFAULT CURRENT_TIMESTAMP)
- Locations
    - locationID (INT AUTO_INCREMENT PRIMARY KEY)
    - name (VARCHAR(225) NOT NULL)
    - address (VARCHAR(225) NOT NULL)
    - category (VARCHAR(225) NOT NULL)

Diagram:



Class Diagram & Inheritance Hierarchy



Data Collection System

The data collection system will be for collecting newly added comments and reviews by users to the program, so that they can be saved into the database and always shown. The data will be stored in the reviews table in the same database, foodLocator. This has to be run every time a new review is left on a food place.

Insert algorithm that works like this

- Check the users input to ensure that it is valid
- Use SQL Query to ensure its not a duplicate
- Once Validated Insert Review
- Update/Synchronize the data
- Store/Log the meta data (restaurant and customer ids)

Web Interface

Map Screen

The main page is an interactive map of the USC Campus and Village, with clickable red pins on it to represent where the various food locations are. There will also be an option of a sidebar on the left side that has a list of all the nearby food options.

At the bottom of the map there is also a button for locating the user. At the top of the page there is a navbar with options for map, list, and login.

When clicking on a pin, it opens up a modal with the selected location's name, ratings, distance and its category type at the top for the user. Below that information will be the top 3 reviews by other students for that same place. There will then be three buttons at the bottom of the modal for the user to click on: View All Reviews, Write Review, and Get Directions.

Review Screen

The review screen will be a modal, from which it is possible to view and write reviews for the selected location. This modal must support list and pagination, it will be scrollable.

The reviews can be accessed by selecting one of the red pins on the map. Doing so will open the corresponding food place's modal. Users are then given the option to either scroll through existing reviews for that place, or create a new one if they want. The latter is only for users that are logged in.

This review system will use the existing reviews table in the foodLocator database, with columns for reviewID, locationID, userID, rating, title, body, createdAt. The review system works on a scale of 1 to 5, and there will be guards that will not allow for a review out of those bounds.

Layout:

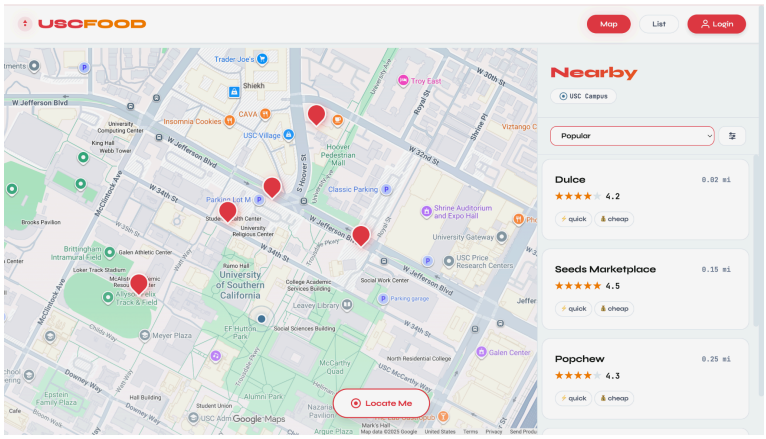
- Header: Location Name and Average Rating
- Review Item: author, stars, title, body, tags, createdAt

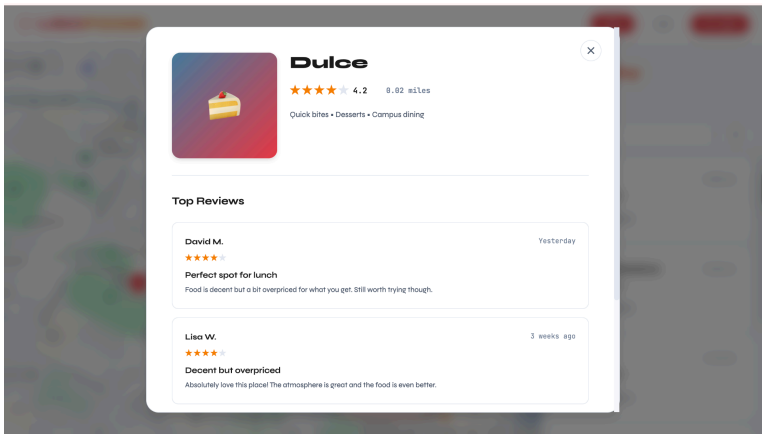
Interactions:

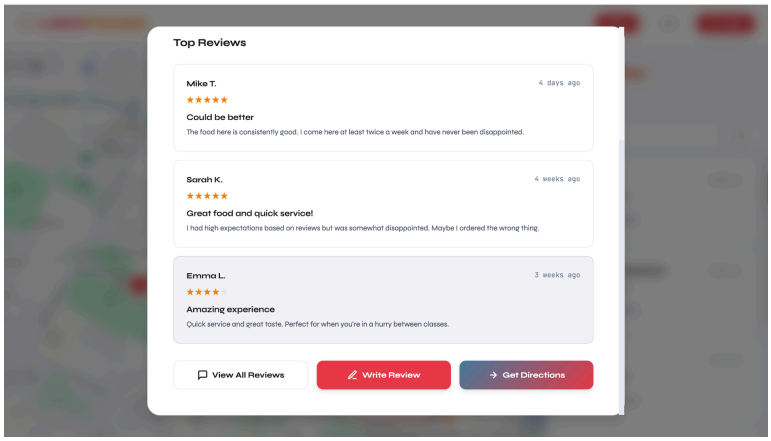
- Tag filters (“vegan”, “cheap”, “quick service”)
- Helpful voting (upvote or “Mark as Helpful”)

Algorithm Walkthrough:

Mockups with filler values:







Testing Plan

Authentication System

1. **Black Box Testing - Login - Successful Authentication**
   Input: User clicks on "Login" and logs in with a valid email and password
   Expected Output: User is redirected back to the main application screen; user's past class schedules are fetched from the backend and are accessible on the main application screen
2. **Black Box Testing - Login - Invalid Email**
   Input: User clicks on "Login" with an invalid email address
   Expected Output: User fails to login, remains on the login page with a red error message under email address reading "Invalid email address"
3. **Black Box Testing - Login - Invalid Password**
   Input: User clicks on "Login" with an invalid password
   Expected Output: User fails to login, remains on the login page with a red error message under password reading "Invalid Password"
4. **Black Box Testing - Creating an Account - Invalid Email**
   Input: User clicks on "Create Account" with an invalid email address
   Expected Output: User fails to create account, remains on the create an account with a red error message under email address reading "Invalid Email"
5. **Black Box Testing - Creating an Account - Weak Password**
   Input: User clicks on "Create Account" with a weak password. A password is considered weak if it fails to meet any of the requirements:
   ○ 8 or more characters long
   ○ At least 1 uppercase letter
   ○ At least 1 lowercase letter
   ○ At least 1 number (0-9)
   ○ At least 1 special character (!,@,#,$,%,^,&,*,(,))
   Expected Output: User fails to create account, remains on the create an account page with a red error message under password reading "Password Too Weak"
6. **Black Box Testing - Creating an Account - Email already taken**
   Input: User clicks on "Create Account" with an email that is already registered with the website.
   Expected Output: User fails to create account, remains on the create an account page with a red error message under email reading "Email already taken"
7. **Black Box Testing - Password Reset - Successful Password Reset**
   Input: User requests a password reset and follows the instructions in the email to reset their password.
   Expected Output: User successfully resets the password and receives a confirmation message. They can log in with the new password.

8. **Black Box Testing - Password Reset - Failed Password Reset**
   Input: User requests a password reset and either:
   - Enters an email that is not associated with an account
   - Enters a valid email, but does not follow instructions in the email within 10 minutes

   Expected Output: Error message reading “Email is not linked to an account”, or requested password reset link will time out

Database Architecture

1. **White Box Testing - Database Schema**
   Run the SQL scripts to create the tables in the database, the program should then create them correctly without errors and linked appropriately, all under the same schema
2. **Unit Testing - Database Schema - Autoincrement**
   Add in multiple entries to each table without specifying their IDs, and the program should auto-increment starting from 1 and remain unique for the userID, reviewID, and locationID
3. **Black Box Testing - Non-Null Constraints**
   Try to insert an user without an email, a review without a title, or a location without a name to each of the tables. They should all fail, and an error message should be displayed for NOT NULL violation.
4. **Unit Testing - Rating Validation**
   Try to insert reviews with a rating of 5, 0, 6, -4 to the table. Out of those, only the values between 0 and 5 should be entered, and for any rating outside the range it should be rejected and an error message should pop up about an invalid value for the field.

Web Interface

1. **Black Box Testing - Reviews pop up**
   Input: User clicks on a pin on the map
   Expected Output: A tab should pop up with the corresponding places information and its reviews
2. **Black Box Testing - Writing a Reviews**
   Input: the user presses the button for writing a review
   Expected output: A screen where the user can enter their review should pop up
3. **Stress Test - Several Reviews**
   Add a few hundred reviews for a place and ensure that they all stay saved into the system and show up correctly and that the user is able to scroll through them smoothly
4. **Black Box Testing – Duplicate Review Detection**

Input: Submit same review body twice within 30s.
Expected Output: Second attempt blocked with “Duplicate review detected.

5. **Black Box Testing – Edit/Delete Permissions**
   Input: User tries to edit/delete another user’s review.
   Expected Output: Action blocked only your own reviews can be edited/deleted successfully.

Map System

1. **Black Box Testing - Map Initialization**
   Input: User opens the application
   Expected Output: Map loads centered on USC campus (coordinates: 34.0224° N, 118.2851° W) with appropriate zoom level showing all campus food locations
2. **Black Box Testing - Pin Rendering**
   Input: Map loads with existing food locations in database
   Expected Output: All food location pins appear on map with correct coordinates; clicking a pin displays location name and average rating
3. **Black Box Testing - Geolocation Permission Granted**
   Input: User grants location access when prompted
   Expected Output: Blue dot appears on map showing user's current location; nearby locations are highlighted or filtered
4. **Black Box Testing - Geolocation Permission Denied**
   Input: User denies location access when prompted
   Expected Output: Map still loads centered on USC; user sees all locations but without proximity-based filtering; informational message appears: "Enable location access to see nearby options"
5. **Black Box Testing - Get Directions Button**
   Input: User clicks "Get Directions" on a location's info popup
   Expected Output: Opens device's default maps application (Google Maps or Apple Maps) with directions from user's current location to selected food location
6. **Stress Test - Multiple Pins**
   Input: Load map with 50+ food location pins
   Expected Output: All pins render within 3 seconds; map remains responsive to zoom and pan interactions without lag
7. **White Box Testing - Map API Integration**
   Input: Application makes API call to mapping service
   Expected Output: API responds with valid map tiles; API key is properly authenticated; error handling catches failed API calls and displays fallback message

Deployment

Step 1: Push the application to the live production server

1. Ensure the production server has the required environment installed (Java, Tomcat, MySQL).
2. Upload the project build files to the server
3. Confirm that the version being deployed is fully tested and approved

Step 2: Create the Production Database
In MYSQL:
Create the Database

```
CREATE DATABASE IF NOTEXISTS foodLocator;
USE foodLocator;
```

Necessary tables

```
CREATE TABLE Users(
   userID INT AUTO_INCREMENT PRIMARY KEY,
   email VARCHAR(255) UNIQUE NOT NULL,
   name VARCHAR(255) NOT NULL,
   password VARCHAR(255) NOT NULL
);

CREATE TABLE Locations(
   locationID INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   address VARCHAR(255) NOT NULL,
   category VARCHAR(255) NOT NULL
);

CREATE TABLE Reviews(
   reviewID INT AUTO_INCREMENT PRIMARY KEY,
   locationID INT NOT NULL,
   userID INT NOT NULL,
   rating DECIMAL(2,1) CHECK (rating >= 0 AND rating <= 5),
   title VARCHAR(255) NOT NULL,
   body VARCHAR(2000) NOT NULL,
   createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY(locationID) REFERENCES Locations(locationID),
   FOREIGN KEY(userID) REFERENCES Users(userID)
);
```

Step 3: Configure Database Connection in the Application
Example
DB_URL = "jdbc:mysql://localhost:3306/foodLocator";
DB_USER = "root";
DB_PASS = "yourPassword";
*Ensure the MySQL Connector JAR is in the buildpath

Step 4: Build and Deploy to Tomcat

Option A — Deploy a .war file

1. Build your project in Eclipse/IntelliJ.
2. Export as WAR file.
3. Place the WAR file into the tomcat/webapps/ directory.
4. Start Tomcat.
5. Tomcat will automatically unpack the WAR.

Option B — Deploy manually (exploded folder)

1. Copy your project folder into tomcat/webapps/foodLocator/.
2. Restart Tomcat.

Step 5: Verify the deployment

1. Confirm the application loads by visiting it in the browser
2. Run the database connectivity test by using the applications

Step 6: Perform full regression testing on the production server

- Use testing plan to test systems

Step 7: Configure Production Server Settings

1. Configure HTTPS (via certbot or Tomcat SSL)
2. Set correct file permissions for:
    a. /logs
    b. /WEB-INF
3. Change default MySQL password for security
4. Limit user permissions (e.g., separate app user account)

Step 8: Hosting, DNS, and Routing Setup

1. Point the DNS record to the server's IP

2. Verify these permissions/websites are working

Step 9: Final deployment and instruction

1. Notify all end users that the application is now live
2. Provide the final production URL
3. Provide instructions for accessing the system

Team AI Logs

Olivia Liau

I developed the frontend for a USC campus food discovery web application featuring an interactive static map system, user authentication modals, review display with pagination and filtering, and comprehensive geolocation services. The project demonstrates modern web development practices including responsive design, accessibility compliance, and secure client-side architecture.
PROMPT: I queried the following to Claude "I'm coding a food locator in Java using Eclipse and Tomcat 9.0 which uses a user's location to identify nearby restaurants and display their ratings. Generate HTML, CSS, JS files with a map system centered on USC campus, clickable pins showing location details and top 3 reviews, and a review screen supporting sorting (most recent, highest rated, most helpful) and tag filtering (vegan, cheap, quick service)."
ISSUES
Issue #1: Missing Map Background Image - The initial implementation referenced a static map image usc-map.png in the CSS (background-image: url('usc-map.png')), but this file didn't exist in the project. This caused the map container to display as a blank white/gray area with no visible USC campus map, making it impossible to understand the geographic context of restaurant locations.
Issue #2: Map Markers Not Displaying - The location markers weren't appearing on the map interface. The JavaScript code in app.js used coordinate-to-pixel conversion functions (latLngToPixel() and addPlaceMarker()) but the markers weren't rendering visually, leaving users unable to see where restaurants were located on campus.
Issue #3: Missing Login Button - The navigation bar lacked a login button entirely. The initial HTML only included "Map" and "List" navigation buttons along with a "Locate Me" button, but there was no way for users to access authentication features or create accounts, preventing user-specific functionality like writing reviews.
Issue #4: Original Theme Not Suitable - The AI generated a dark color scheme with --bg-main: #0A0A0F and light text, which produced readability issues and wasn't appropriate for an academic project requiring professional appearance and high accessibility standards.

FIXES
Fix #1: Map Image Documentation - I requested the AI create comprehensive documentation explaining how to obtain and add the usc-map.png file. The AI generated MAP_IMAGE_REQUIRED.txt with three methods: taking a Google Maps screenshot centered on coordinates 34.0224, -118.2851; downloading an official USC campus map; or creating a custom map covering the bounds (North: 34.0280, South: 34.0200, East: -118.2820, West: -118.2880).
Fix #2: Marker Styling and Positioning - I worked with the AI to add proper CSS classes for map markers. The solution included .map-marker styles with absolute positioning, 32px dimensions, red background (var(--primary)), white border, and teardrop shape using border-radius: 50%

50% 50% 0 with transform: rotate(-45deg). The markers also received hover effects scaling to 1.15x with enhanced shadows, making them visually prominent and interactive.
Fix #3: Login Button Implementation - I requested: "Add a login button to the top right of the navigation bar and move the locate me button to the bottom of the map." The AI added a .login-btn styled in primary red with a user icon, positioned in the top-right navigation. The "Locate Me" button (originally in that position) was repositioned as a floating element at the map bottom using position: absolute; bottom: 32px; left: 50%.
Fix #4: Light Theme Conversion - I explicitly requested: "Use a light modern theme instead." The AI modified CSS variables from dark backgrounds to --bg-main: #F8F9FA, --bg-elevated: #FFFFFF, changed text from white to --text-primary: #1A1A1A, and reduced shadow opacity from 0.4 to 0.1.

These fixes were essential for creating a functional application. Without the map image and visible markers, users cannot locate restaurants geographically, defeating the core purpose. The login button enables critical features like writing reviews and personalized recommendations. Finally, the light theme ensures professional appearance. Together, these modifications created a semi-functional prototype ready to be connected to the database.

.

Laila Gayden

For this project, my main responsibility was producing all of the documentation that describes how our website should work. This included writing the high-level requirements, the technical specifications, and the deployment document. I drafted all of these pieces myself, relying heavily on what we had clarified during our weekly meetings and earlier assignments. Since we had already discussed the system flow, features, and expectations many times, I had a strong foundation to work from and could describe the functionality in a clear and organized way.

While I was developing the documentation, I had to make a number of detailed decisions—how to explain different components, what level of depth each section needed, and how to describe the system in a way that someone else could easily understand. For the technical specifications especially, I needed to break down system behavior, outline the structure of our components, and make sure everything aligned with what the team had planned. The deployment document required me to think through each step of setup and configuration so that someone could reliably replicate our development environment.

As I worked, I reviewed and revised the documents multiple times to make sure everything was accurate and consistent. I checked for clarity, corrected vague descriptions, and made sure the terminology matched across all three documents. After completing the drafts, I used AI as a final step—not to write the documents for me, but to help refine the language, smooth out transitions, and catch areas where the writing felt repetitive or overly technical. I compared the AI-suggested edits with my original text to ensure the meaning stayed true to our actual design decisions. In a few cases, the AI tried to generalize sections too much, so I re-edited those parts to maintain precision.

Overall, the AI served mainly as an editing tool at the end of the process. The structure, content, decisions, and explanations all came from me, but AI helped polish the final product so the documentation was cleaner, more readable, and more professional.

Bona Suh

For this project, beyond developing the backend, I also helped Olivia debug and finalize the frontend. Most of the debugging work was done manually, but I used AI strategically to accelerate troubleshooting when stuck, generate example fixes, and verify CSS/JS behavior. My role was not to have AI write the solution, but to use it as a tool to test approaches and reduce trial-and-error time.

Clarity:

Throughout development, I tested the frontend interface to ensure the map, markers, review display, and login modal behaved correctly. We quickly identified that certain visual elements were either not rendering or not positioned properly. The map displayed, but the markers were invisible due to missing CSS, and UI components occasionally overlapped or broke at different screen sizes. Our goal during debugging was to make the map interactive, intuitive, and visually clean.

Specificity:

While working through the issues, we had to make decisions regarding marker placement logic, event listeners, and styling rules. For example, markers were generating in JavaScript but not visible because the AI-generated CSS lacked shape, color, and z-index. We inspected the DOM, confirmed the JS functions were firing, then adjusted CSS to use absolute positioning and custom styling.

We also noticed that the login modal existed in HTML but had no trigger button, and the "Locate Me" button placement conflicted with navbar spacing. We moved the button to the bottom of the map, added a login button to the top right, and rewired modal visibility logic so the UI became accessible and functional.

Correctness:

When issues surfaced, I used ChatGPT selectively to test potential fixes—for example, asking it to rewrite the marker CSS or help identify why a listener wasn't firing—but I always reviewed the output line-by-line and manually modified it to fit our file structure.

A specific example: when troubleshooting pagination glitches in the review list, AI generated a revised indexing snippet. Before using it, we manually tested edge cases (last page, filter applied, zero results) and corrected off-by-one behavior. We verified everything visually in the browser and through console logs to ensure correctness instead of trusting auto-generated code blindly.

Depth:

Helping debug the frontend was important because visuals and functionality must work together for a food locator to make sense—the backend can only store places, but without visible markers and working UI, users cannot interact with them. Consistent styling, functional buttons, and clean JavaScript logic directly improved usability and accessibility.

By combining manual debugging with targeted AI assistance, we accelerated front-end fixes without sacrificing control or reliability. Working through this process also strengthened our workflow—Olivia and I could divide the problem, test ideas quickly, iterate faster, and ultimately deliver a more polished, user-ready interface.

Siiri Haapasalo

For this project, I was in charge of creating and populating the backend as well as finalizing this document. I manually coded my part, and then used AI at the end to help simplify the process of filling the backend and making sure everything was correct.

Clarity:

From all our weekly meetings and assignments, I already had a very good idea of what the backend would look like. I just needed to create 3 tables, one for users, locations, and reviews. And then populate the locations table with the foodplaces in the backend.

Specificity:

During the actual development, I had to make decisions on the constraints for the columns and ultimately how our data would be formatted, and how many characters to allow for the various fields. I also needed to figure out how to connect the tables to each other.

When it came to filling out the locations table, I realized that it will be important for each of the entries to be formatted consistently, so like Suite # vs Ste # in the addresses. I also made the decision to use Apple Maps for all of the addresses for consistency's sake. But I quickly realized this was going to be a tedious process to manually add in all the places as even within Apple Maps some of the addresses were formatted differently.

Correctness:

I realized that when setting the different VARCHAR() values, that the bodies of the reviews would need way more characters, so I opted for: body VARCHAR(2000) NOT NULL, to allow for longer reviews. I also decided to stick with having all the addresses have Ste #, rather than Suite # to make them slightly shorter and also consistent.

After I had manually filled in a couple of the food places from the Village, I realized that this process could be optimized by asking ChatGPT: "Send me a list of all the food places on USC's campus and the USC village formatted to match this sql so I can copy paste them
INSERT INTO Locations (name, address, category) VALUES
('Cava', '3201 S Hoover St, Ste 1840', 'Mediterranean'),
('IL Giardino', '3201 S Hoover St', 'Italian'),
…"

Then after it sent me a list of places, before adding them to the query, I double checked that the addresses, names, and categories were correct and consistent. I actually found that it gave me a couple wrong places, it especially messed up the various Starbuck addresses, it gave me some that were not on campus nor the village.

Depth:

Consistent database entries are important because that makes all the code where they are accessed easier and simpler to write and it helps avoid inconsistencies in the final product. Now all the addresses will show in a consistent way, and corrrectly. Also, without increasing the characters for the reviews, users would not be able to leave longer reviews.

Yougendy Mauricette

(PrePromp Uploads)
For this project Jason and I were in charge of the UI and Auth, I worked mainly on the UI and Frontend while he worked on the Servlets. I used Lab 12 and Assignment 3 as templates for my LLM. I began by editing the Styles.css that was in my assignment 3 to match what we were needing for our login and signup. And once I had a pretty decent looking UI I submitted screenshots of my teammates UI so that Chatgpt would be able to match the Color Scheme and Fonts.
(Clarity)

My prompt was “Using my Signup HTML and Login HTML
Use this style for all my HTML/CSS: Primary color: #E63946 (USC red style) Soft border-red variant for outlines Background: #FFFFFF and #F7F8FA Text: dark gray #333 and medium gray #666 Font: Inter or SF Pro (rounded modern) Buttons: pill-shaped, bold, red fill for primary, thin border for secondary Navbar: white background, logo left, button group right Cards: white, rounded, slight shadow, icon tags Icons: minimal outline Everything is clean, rounded, spaced out like the USCFOOD demo page”
To ensure it followed the color and visual scheme of my teammates.
(Specificity)

The major issue I had was actually identifying the colors and fonts since I did not have direct access at the time to my teammates files and originally It wasnt working because AI has accidentally referenced my old Lab 12 files rather than the new one.
(Correctness)

but I was able to get it close enough with the help of screenshots and color correcting tools. And once i went through and added the proper lines like Login and Signup HTML it worked properly.
(Depth)

Once I and LLM had the proper css and HTML linked properly tested them inside my local Tomcat environment, adjusting path linking, merging small parts from Assignment 3, and making sure the UI remained consistent across both pages. This required verifying that the CSS was loading properly, that the pages responded correctly when resized, and that the layout matched the rounded, modern USCFOOD style.

Joyce Zhou

For this part of the project, I implemented the complete database integration layer including connection management, data access objects, and RESTful API servlets. I used GitHub Copilot IDE extension to help generate boilerplate code and understand proper Java design patterns, particularly the DAO pattern for separating database logic from business logic and the Singleton pattern for connection management. One major challenge was ensuring SQL injection protection, which I addressed by using PreparedStatements throughout all DAO classes to safely parameterize queries. I also worked with JDBC for database connectivity, implementing CRUD operations across three main tables (Users, Locations, Reviews) with proper foreign key relationships and aggregate queries for calculating average ratings. The servlet layer was built following RESTful API conventions with proper HTTP methods (GET, POST, DELETE) and JSON serialization using the Gson library, along with session-based authentication using HttpSession. Another challenge was understanding how to properly handle ResultSet processing and converting database results into Java objects, which required careful attention to data types and null checking. GitHub Copilot was particularly helpful for generating servlet structure and explaining how to set appropriate response headers and status codes (200, 201, 401, 404) for different scenarios. Overall, the implementation follows MVC architecture with clear separation between models (POJOs), data access layer (DAOs), and controllers (Servlets), making the codebase maintainable and testable.

Abhijay Sharma

On this project, I teamed up with Olivia to build the review page for the Food Locator app. She laid most of the foundation for the page using Claude to draft the starting HTML, CSS, and JS. She covered things like the pop-up design, page switches, and how filters sort reviews. Afterwards, I manually went in and reviewed the code and made minor changes where necessary to better tailor the functionality to what we were looking to achieve with the review page, as well as to test and fix any bugs present. Some problems that I found with the AI generated code was wrong labels messing up comment displays. The popup looked off next to the map view, colors didn't match. Also caught a glitch where certain filter combos showed blank spots even if there were matches.

I fixed these problems by going through the JavaScript event triggers, making sure the reviews loaded right from the database reply, yet tweaking the CSS so things looked smooth across the app. Then I checked how the "Mark as Helpful" vote button worked and fixed a bug where likes disappeared after reloading the page.

Jason Byun

For this part of the project, I implemented the full authentication workflow for the Food Locator application, including sign-up, login, and session-based access control for writing reviews. I updated the project to use Jakarta Servlet since Tomcat 11 no longer supports the legacy javax imports, which required modifying the pom.xml file and replacing all servlet imports with their jakarta equivalents to ensure the application compiled correctly.

I created dedicated SignupServlet and LoginServlet classes to handle form submissions and input validation. A major issue I encountered was form parameters being returned as null. This was caused by calling getWriter() before accessing getParameter(), which consumes the input stream. Reordering the logic resolved the issue. I also configured JDBC database connectivity and fixed a connection failure caused by an incorrect database password in db.properties.

To manage user authentication, I implemented session checks using HttpSession, allowing the JSP views to dynamically show or hide UI elements such as the Logout button or the Write Review option based on whether the user is logged in. I also reorganized static assets into css/, js/, and jsp/ directories and updated relative paths to ensure stylesheets and scripts were properly loaded.

Before pushing to GitHub, I added a .gitignore entry for db.properties to prevent storing database credentials in version control and documented setup instructions for local configuration. Overall, my contribution focused on integrating servlet-based authentication, database connectivity, and session management to create a secure and consistent user experience. Most of these actions were contributed by suggestions and code written by Copilot.