Comprehensive Exercise Report

Team DIP392-wedding_app of Section 001
Tural Asgarov - 221ADB123
Tural Mammadli – 221ADB166
Rza Mirzayev – 221ADB213
Khayal Aghazada 221ADB119,
Sadulla Abdukodirov 221ADB187,
Ahmad Ughurluzada 211ADB063

Requirements/Analysis

Week 2

Journal

The following prompts are meant to aid your thought process as you complete the requirements/analysis portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- After reading the client's brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
  - The Wedding Management App is a user-friendly software designed to help efficiently plan and manage weddings.
    - User roles and account management
    - Guest management
    - Venue booking
    - Vendor tracking
    - Budget management
    - Task scheduler with reminders
    - Dashboard overview of guests, expenses, and pending tasks
- After reading the client's brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
  - What specific user roles are expected (e.g., wedding planner, vendor, guest)?
  - Are there any particular design preferences or branding guidelines for the UI?
  - How detailed should the vendor tracking be (e.g., just contact information, or service-specific details)?
  - What is the expected level of automation for task scheduling and reminders?
  - Should the system support any integrations with external services (such as calendar apps or payment gateways)?

Instructor's answers (documented during the discussion):

- The application should support at least three roles: admin (wedding planner), vendors, and guests.
- The UI should be clean and intuitive, adhering to a simple design without complex styling requirements.

    - Vendor tracking must include basic contact details and the services they offer.
    - Task scheduling should allow for manual entry with simple automated reminders.
    - No external integrations are required for this project.

- Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.

  We are generally comfortable with the main technologies (Java, Spring Boot, PostgreSQL) mentioned in the brief. However, we will review additional best practices regarding designing a task scheduler with reminders to ensure a smooth user experience.

    - *References:*
        - Java Spring Boot Documentation
        - PostgreSQL Official Documentation

- Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
    - **Wedding Planners/Administrators:** Typically experienced event organizers managing multiple weddings, responsible for overall planning and coordination.
    - **Vendors:** Service providers who offer products or services (e.g., catering, photography) and need to update their availability and contact information.
    - **Guests:** Individuals invited to the wedding who may interact with the system for RSVP or viewing event details.

- Describe how each user would interact with the software
    - **Wedding Planners:** Log in to create and manage wedding events, oversee guest lists, schedule tasks, manage budgets, book venues, and monitor overall event progress via the dashboard.
    - **Vendors:** Access their profiles to update service details and availability, receive notifications about event requirements, and communicate with the planners.
    - **Guests:** Use the app to view invitations, confirm attendance (RSVP), and receive event updates.
    -
- What features must the software have? What should the users be able to do?
    - With the system dynamically adjusting features depending on whether the user is an administrator, vendor, or guest, the application will let users register accounts and log in. By adding, adjusting, tracking RSVPs, and issuing reminders, wedding planners may fully control guest lists so that every element is in order. By means of simple capabilities such calendar views and availability checks, the app also facilitates venue booking, hence facilitating event scheduling. The program keeps a comprehensive database for vendors that records necessary contact details and highlights their provided offerings. A strong budget management tool that tracks costs and creates thorough financial reports helps users also control their financial plans.

By enabling the creation of tasks with deadlines, automated reminders, and progress tracking—all of which help to improve the user experience—a dedicated task planner also At last, the dashboard offers a consolidated view of important indicators such guest counts, spending, and outstanding chores so allowing customers to quickly check the whole state of wedding preparation.

- Other notes:
    - The project is designed to be completed within a 20-30 hour timeframe, focusing solely on implementing the essential functionalities required for a basic yet functional user interface. Given the time constraints, advanced features such as real-time collaboration, deep analytics, or AI-based recommendations are not included in the scope. The development will adhere to the agreed-upon tech stack, which consists of Java, Spring Boot, and PostgreSQL, as established in the initial agreement made via Fiverr.

Software Requirements

Detailed Project Description (Overview)

This Wedding Management App is designed for wedding planners to streamline and simplify the planning process. The software focuses on core functionalities such as creating and managing user accounts, overseeing guest lists, scheduling and tracking tasks, managing vendors, booking venues, and organizing budgets. The system will be built using Java, Spring Boot, and PostgreSQL, with an emphasis on delivering a functional and intuitive user interface. Advanced features such as real-time collaboration, deep analytics, or AI-based recommendations are intentionally excluded to keep the project within a 20-30 hour development window. Ultimately, the solution will give Abel and his team a central hub to manage the complexities of wedding planning efficiently, ensuring key tasks, vendor details, and budgetary information are always accessible.

Functional Requirements

1. **User Account and Role Management**
    - The system shall allow users to create an account with unique login credentials.
    - The system shall differentiate between three primary user roles (Admin, Vendor, Guest) and provide role-based access to features.
    - Admin users shall be able to view, add, update, or remove other user accounts.
2. **Guest Management**
    - Admin users shall be able to add, edit, and delete guest profiles.
    - The system shall allow Admin users to track RSVPs and update guest statuses.
    - Automated reminders shall be sent to guests who have not responded within a configurable timeframe.
3. **Venue Booking**
    - Admin users shall be able to view and book available venues within the system.
    - The system shall display a calendar with venue availability and existing bookings.
    - Admin users shall receive confirmation notifications once a venue is successfully booked.
4. **Vendor Tracking**
    - Admin users shall be able to maintain a database of vendors, including contact details and service types.
    - Vendors shall be able to update their availability and service information in their own profiles.

- The system shall notify Admin users when a vendor updates availability or service details.

5. **Budget Management**
    - The system shall allow Admin users to input budget targets and log expenses under specific categories.
    - The system shall calculate total expenses and provide a summary comparing actual costs to the allocated budget.
    - Admin users shall be able to export budget data in a basic format (e.g., CSV or PDF).
6. **Task Scheduler and Reminders**
    - Admin users shall be able to create tasks with deadlines, assign them to specific user roles, and mark their completion status.
    - Automated reminders shall be sent to the responsible users when tasks approach or exceed their deadlines.
    - The system shall display a task overview page with filters for due dates, statuses, and assigned roles.
7. **Dashboard Overview**
    - The system shall provide a dashboard that summarizes guest counts, expenses, pending tasks, and vendor availability.
    - Admin users shall be able to customize the dashboard view to display key metrics in real time.
    - The dashboard shall update automatically as data is modified in the system.

External & User Interface Requirements

1. **User Interface Simplicity**
    - The interface shall be designed for ease of use, following a clean and intuitive layout without complex styling or animations.
    - The system shall use responsive design principles to ensure usability on both desktop and mobile devices.
2. **Accessibility**

    - Fonts, colors, and layout shall comply with basic accessibility standards to accommodate users with visual or motor impairments.

3. **Notifications**

    - The system shall provide email or in-app notifications for critical actions such as RSVPs, venue booking confirmations, and task reminders.

    - Users shall have the option to enable or disable specific notification types in their profile settings.

System Features

1. **Database Integration**

    - The system shall utilize PostgreSQL for secure and reliable data storage.

    - Database tables shall be structured to maintain data consistency across user accounts, guests, vendors, venues, tasks, and budget entries.

2. **Security**

    - The system shall implement basic security measures such as password hashing, role-based access control, and secure session handling.

    - Only authorized roles shall be permitted to view or edit sensitive data (e.g., budget details, vendor contracts).

3. **Scalability and Maintainability**

    - The system architecture (Java, Spring Boot, PostgreSQL) shall be designed to accommodate potential future feature expansions.

    - Code shall follow standard best practices, allowing for straightforward maintenance and updates.

Nonfunctional Requirements

1. **Performance**

    - The system shall respond to user actions (e.g., logging in, saving data) within two seconds under normal usage conditions.

    - The database shall handle up to 1,000 guest entries, 100 vendor entries, and multiple events without noticeable performance degradation.

2. **Reliability and Availability**
    - The system shall have a maximum downtime of 2% per month, excluding scheduled maintenance.
    - In the event of unexpected crashes, data integrity shall be preserved via regular backups or transaction logging.
3. **Usability**
    - The user interface shall be intuitive enough for a new user to perform basic tasks (e.g., creating an account, adding guests) with minimal guidance.
    - Basic tooltips or help text shall be provided for critical features.
4. **Portability**
    - The application shall be deployable on common operating systems such as Windows, macOS, or Linux, provided Java is installed.
    - The system shall be compatible with major modern browsers when accessed via a web-based front end.

Sample User Stories (Optional)

1. **As an Admin (Wedding Planner),** I want to manage multiple wedding events so that I can keep track of all tasks, vendors, and budgets separately.
2. **As a Guest,** I want to RSVP easily so that the planner knows whether I will attend the wedding.
3. **As a Vendor,** I want to update my service offerings and availability so that wedding planners can book me for relevant dates.
4. **As an Admin (Wedding Planner),** I want a dashboard view that shows overall expenses, tasks, and guest counts so that I can monitor event progress at a glance.

Black-Box Testing

Instructions: Week 4

Journal

- What does input for the software look like (e.g., what type of data, how many pieces of data)?
    - The software receives various forms of data including textual information (names, email addresses, vendor service descriptions), numerical values (budget figures, expense amounts), and dates (for scheduling tasks, booking venues, and tracking RSVP deadlines). For example, during account creation users provide a username, password, and role selection, while guest management requires names, contact details, and RSVP statuses. Each module expects multiple pieces of data that together create a comprehensive dataset for each functionality.
- What does output for the software look like (e.g., what type of data, how many pieces of data)?
    - The outputs include confirmation messages for successful operations (such as account creation, booking confirmations, or task completion notifications) and error messages when invalid data is entered. Additionally, the system generates dashboards displaying key metrics like guest counts, budget summaries, and pending tasks. Data is presented in various formats such as text lists, tables, or summary charts, depending on the context of the information.
- What equivalence classes can the input be broken into?
    - Inputs can be categorized into valid and invalid equivalence classes. Valid inputs include correctly formatted email addresses, passwords meeting security criteria, and numerical values within acceptable ranges (e.g., positive numbers for budget entries). Invalid inputs might be empty fields, improperly formatted email addresses, excessively long or short strings for usernames, or negative numbers where only positive values make sense. This classification helps in testing how the system processes expected data versus erroneous data.
- What boundary values exist for the input?
    - For text fields, boundary values include the minimum and maximum allowed characters (for instance, a username might have a minimum of 3 and a maximum of 20 characters). In numerical fields like budget management, the lower boundary is often zero (indicating no expense), and an upper boundary could be defined by system constraints or practical limits. Date inputs must also be tested at their boundaries, such as the earliest acceptable date (possibly today's date or a date in the past for historical data) and the farthest future date that the system should handle.
- Are there other cases that must be tested to test all requirements?
    - Inputting special characters or SQL injection strings to ensure robust data validation and security.

    - Using boundary dates (e.g., leap year dates, end-of-month dates) to test the scheduling functionality.

    - Entering maximum allowable characters in text fields to ensure the system handles lengthy inputs gracefully.

    - Testing simultaneous operations such as updating a vendor's information while a task reminder is being sent, to confirm the system's consistency and reliability.

    - Ensuring that error messages and fallback options are correctly triggered when input data violates constraints.
    -
- Other notes:
    - For our Wedding Management App, the black-box testing strategy is focused exclusively on the functional aspects defined by the client's requirements. This means we will evaluate features like user account creation, event scheduling, vendor management, and budget tracking solely from an end-user perspective without considering the internal code structure. By doing so, we can ensure that every component from managing user roles and guest lists to updating the dashboard and performs as expected with a wide range of input types. Regular testing of both valid and invalid interactions will verify that the system not only provides the correct feedback and maintains data integrity, but also consistently meets the expected behavior outlined in our project specifications.

Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

Test ID	**Description**	**Expected Results**	**Actual Results**
TC001	User Registration with valid data (all required fields provided, valid username, password, and role).	The system should create the account successfully and display a confirmation message.	200 OK (Postman). The API returns a success message (e.g., {"message": "User registered successfully!"}). User data is correctly saved to the database.
TC002	User Registration with missing required fields (e.g., no username provided).	The system should reject the registration attempt and display an error message indicating that required fields are missing.	400 Bad Request (Postman). The API returns an error message detailing which required fields are missing (e.g., {"errors": ["Username is required", "Email cannot be empty"]}). No user is created in the database
TC003	User Login with valid credentials.	The system should authenticate the user and navigate them to the appropriate landing page based on their role.	200 OK (Postman). The API returns a JSON response containing a JWT, user ID, username, email, and roles. The user is successfully authenticated.
TC004	User Login with invalid credentials (e.g., wrong password).	The system should deny access and display an error message indicating invalid login credentials.	401 Unauthorized (Postman). The API returns an error message indicating invalid login credentials (e.g., {"message": "Error: Invalid username or password"}).
TC005	Add Guest with valid details (e.g., valid name, contact information, and RSVP status provided by the admin).	The guest should be successfully added to the guest list, and a confirmation message should be displayed.	200 OK or 201 Created (Postman, assuming an endpoint like POST /api/weddings/{weddingId}/guests). The API returns a success

			message and the created guest object. The guest is correctly added to the specified wedding's guest list in the database.
TC006	Add Guest with invalid details (e.g., blank name or incorrectly formatted contact information).	The system should reject the input and display an appropriate error message.	400 Bad Request (Postman). The API returns an error message detailing the validation errors (e.g., "Guest name cannot be blank," "Invalid email format for guest"). No guest is added to the database
TC007	Venue Booking with a valid date where the venue is available.	The system should successfully book the venue, update the availability status, and send a confirmation notification.	200 OK or 201 Created (Postman, assuming an endpoint like POST /api/bookings or POST /api/venues/{venueId}/book). The API returns a success message and/or the booking confirmation details. The venue's availability is updated in the database, and a confirmation (e.g., email, in-app notification) is triggered.
TC008	Venue Booking with a date that falls on an unavailable period (boundary test for date selection).	The system should prevent the booking and display an error message indicating that the venue is unavailable on the selected date.	400 Bad Request or 409 Conflict (Postman). The API returns an error message stating the venue is unavailable for the selected date(s). No booking is created.
TC009	Budget Management: Adding an expense with a value within the available budget limit.	The expense should be recorded, and the remaining budget should update correctly.	200 OK or 201 Created (Postman, assuming an endpoint like POST /api/weddings/{weddingId}/budget/expenses). The API returns a success message and/or the created

			expense object. The expense is recorded, and the wedding's remaining budget is correctly updated in the database.
TC010	Budget Management: Attempting to add an expense that exceeds the current budget (boundary test).	The system should either warn the user or reject the expense entry, preventing overspending.	400 Bad Request (Postman). The API returns an error message indicating the expense exceeds the available budget (or a warning if the system allows exceeding with a flag). The expense is not recorded, or if recorded with a warning, the budget reflects the overage
TC011	Task Scheduling: Adding a new task with a valid deadline (future date) and description.	The task should be added to the event's task list, and a reminder should be scheduled accordingly.	200 OK or 201 Created (Postman, assuming an endpoint like POST /api/weddings/{weddingId}/tasks). The API returns a success message and/or the created task object. The task is added to the wedding's task list in the database, and any associated reminder mechanisms are initiated
TC012	Task Scheduling: Entering a boundary value for the deadline (e.g., the current date).	The system should handle the deadline appropriately, either accepting it or displaying a prompt if immediate action is required.	200 OK or 201 Created (Postman). The task is successfully added. If the deadline is the current date, it might be flagged as 'due today' or an immediate reminder/notification might be triggered depending on system logic.
TC013	Dashboard Overview: After adding multiple guests, tasks, and budget expenses, verify that the dashboard metrics update correctly.	The dashboard should accurately reflect the total guest count, cumulative expenses, and the number of pending tasks.	200 OK (Postman, for the API endpoint serving dashboard data, e.g., GET

			/api/dashboard/{weddingId}). The API returns data accurately reflecting the total guest count, confirmed guests, pending guests, total budget, amount spent, remaining budget, total tasks, completed tasks, and pending tasks based on the data operations performed in previous steps.
TC014	Vendor Update: Updating vendor service details with valid information.	The vendor’s details should be updated successfully, with the new information displayed in their profile.	200 OK (Postman). The API returns a success message and/or the updated vendor object. The vendor's details are correctly updated in the database
TC015	Vendor Update: Providing invalid input (such as empty service type) when updating vendor details.	The system should reject the update and display an error message regarding invalid input.	400 Bad Request (Postman). The API returns an error message detailing the validation errors (e.g., "Service type cannot be empty"). The vendor's details are not updated in the database

Design

Instructions: Week 6

Journal

- List the nouns from your requirements/analysis documentation.
    - From the requirements and analysis documentation, the main nouns include user, account, role, guest, RSVP, wedding event, wedding planner, vendor, venue, budget, expense, task, dashboard, notification, and service. These represent both the data and functionalities inherent to the Wedding Management App.

- Which nouns potentially may represent a class in your design?
    - Several of these nouns can be directly mapped to classes in the design. For example, "User" would serve as the base class with specialized versions like AdminUser, VendorUser, and GuestUser derived from it. "Wedding Event" naturally represents the core event class, and similarly, "Venue," "Budget," "Guest," "Task," "Vendor," and "Dashboard" can be modeled as individual classes to encapsulate their respective functionalities and data.

- Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
    - Within these classes, attributes are drawn from other nouns or key properties. The User class might include fields such as id, username, password, and role. The Guest class would have attributes like guestId, name, contactInfo, and rsvpStatus. In the Wedding Event class, fields would include eventId, eventName, eventDate, and associations like venue, budget, a guest list, and tasks. Similarly, the Venue class would have a venueId, name, location, and availability, while the Budget class would maintain budgetId, totalBudget, and a collection of expenses. The Task class would contain taskId, description, deadline, and status, and the Vendor class would include vendorId, name, serviceType, contactInfo, and availability. Finally, the Dashboard class would aggregate totalGuests, totalExpenses, and pendingTasks.

- Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.
    - One design option is to create a detailed, modular structure where each noun becomes its own class. In this design, there would be a distinct User class with derived classes for different roles, a separate Wedding Event class that connects to independent Venue, Budget, Guest, Task, Vendor, and Dashboard classes. The advantage of this approach is the clear separation of concerns, which improves maintainability and scalability. However, it could introduce complexity when integrating these classes, as more classes mean more points of interaction and potential for miscommunication between components.

  An alternative design option would be to reduce the number of classes by merging closely related functionalities. For example, combining the Budget and Dashboard classes could simplify the financial overview aspects by handling both expense tracking and summary displays in one class. Similarly, one might consider merging Guest and RSVP-related functionality into a single class. The benefit of this approach is a simpler design with fewer classes, which might be easier to implement under time constraints. On the downside, this design risks reduced clarity and separation of responsibilities, potentially complicating future expansions or maintenance.

- Which design do you plan to use? Explain why you have chosen this design.
    - the first design option with a detailed modular structure. This design's clear separation of concerns helps ensure that each component whether user management, event scheduling, or financial tracking remains focused and easier to update independently. The approach is more scalable and aligns well with object-oriented principles, which is particularly useful for managing the diverse functionalities of the Wedding Management App.

- List the verbs from your requirements/analysis documentation.
    - Key verbs from the documentation include register, login, logout, manage, view, add, remove, update, send, calculate, book, track, display, and schedule. These verbs indicate the primary actions users will perform or that the system will execute.

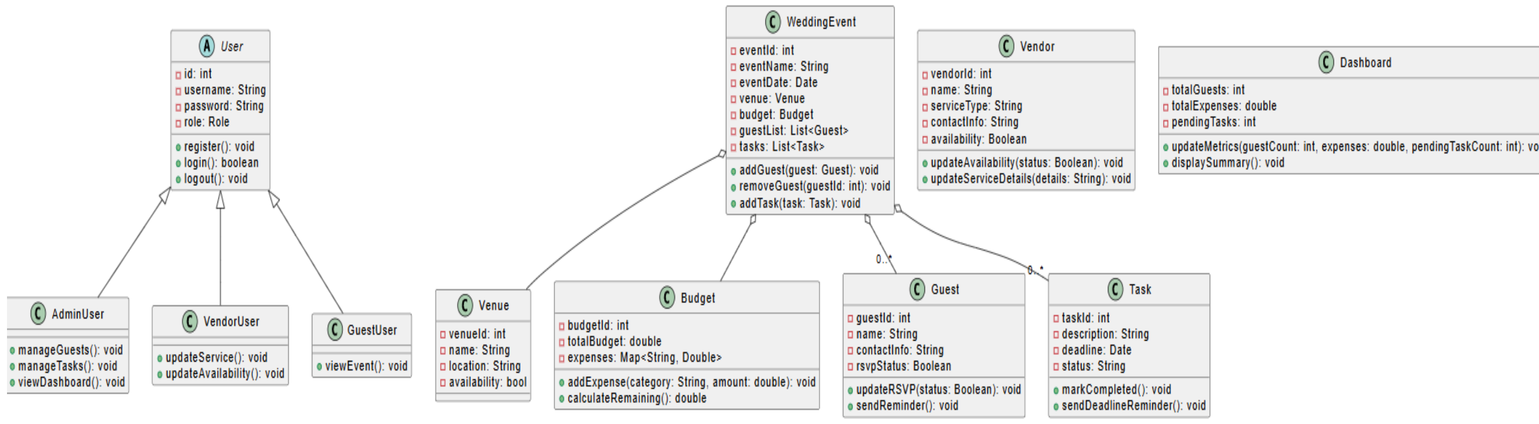
- Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
    - Corresponding to these verbs, potential methods include register(), login(), and logout() in the User class. The AdminUser class could have methods like manageGuests(), manageTasks(), and viewDashboard(), while the VendorUser class might include updateService() and updateAvailability(). For the GuestUser class, a viewEvent() method would be essential. The Wedding Event class would incorporate addGuest(), removeGuest(), and addTask() methods to manage event details. The Guest class might include updateRSVP() and sendReminder(), and the Budget class could feature addExpense() and calculateRemaining(). The Task class would likely include markCompleted() and sendDeadlineReminder(), while the Dashboard class would have updateMetrics() and displaySummary().
- Other notes:
    - This exercise has reinforced the importance of mapping requirements directly to design elements. The chosen modular design not only supports clear separation of responsibilities but also facilitates potential future enhancements, such as introducing a Notification class to manage alerts and reminders across various functionalities. Overall, the design process remains iterative, and further refinement may occur as the implementation phase progresses.

Software Design

Overview of the Design

The application is divided into several core modules: user management, guest management, venue booking, vendor tracking, budget management, task scheduling, and dashboard overview. Each module is implemented using classes that encapsulate both data (fields) and behavior (methods). The main user roles (Admin, Vendor, Guest) are represented through inheritance and role-based logic.

UML Class Diagram:



The design begins with a foundational User class that encapsulates common properties such as a unique identifier, username, password, and a role designation, along with methods for registering, logging in, and logging out. From this abstract class, specialized user types are derived. For instance, the AdminUser class extends User and introduces methods to manage guest lists, create and track tasks, and view a comprehensive dashboard. Similarly, the VendorUser class is intended to provide functionalities for updating service details and availability, while the GuestUser class offers a method for viewing event details.

In the event management domain, the WeddingEvent class is central. It maintains essential information like the event identifier, name, date, and associations with other key elements such as the venue and budget. It also holds collections for guests and tasks. This class will include methods to add or remove guests and tasks. The Venue and Budget classes serve as auxiliary structures; the Venue class holds details like venue identifier, name, location, and availability status, whereas the Budget class is responsible for tracking the overall budget and expenses. Methods in the Budget class enable the addition of expenses and the calculation of remaining funds.

The design further includes a Guest class that stores individual guest details along with methods to update their RSVP status and trigger reminders. A separate Task class manages details about individual tasks, including deadlines and status, and it provides methods to mark tasks as completed or to send deadline reminders. A Vendor class maintains vendor-specific information and includes methods to update both availability and service details. Finally, the Dashboard class aggregates key metrics such as total guest count, total expenses, and pending tasks, with methods to update and display these metrics.

Each class is carefully planned to encapsulate related data and functionality, ensuring a clear separation of concerns while facilitating ease of maintenance and potential future enhancements.

Implementation

Instructions: Week 8

Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompt below and feel free to add additional notes.

- What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.

To implement wedding manager application, we have used several key concepts from our Applied System Software course:

Software Development Life Cycle (SDLC): we have followed an SDLC framework (likely an Agile-inspired iterative approach rather than strict Waterfall) to guide the project through requirements gathering (the features list), design (system architecture, database schema for PostgreSQL, API specifications), implementation (React frontend, Spring Boot backend), testing (using the provided test cases and Postman), deployment, and maintenance.

System Architectures:

Client-Server: project is inherently client-server: the React frontend is the client, and the Spring Boot application is the server. 3-Tier Architecture: This is evident in the design: React (Presentation Tier), Spring Boot (Application/Logic Tier), and PostgreSQL (Data Tier).

API Design (RESTful): we have designed and implemented RESTful APIs using Spring Boot for communication between frontend and backend, defining clear JSON request/response formats and using standard HTTP methods. JWTs were used for securing these APIs.

Database Design: designed the PostgreSQL database schema with JPA entities in Spring Boot, defining relationships and using Spring Data JPA for data operations.

Security Concepts: implemented authentication (JWTs) and authorization (role-based access) via Spring Security, along with password hashing (BCrypt) and input validation.

Implementation Details

# Wedding Management System

A comprehensive wedding management application built with Spring Boot, PostgreSQL, and a modern frontend.

## Features

- **User Management**: Register and authenticate users with different roles (Couple, Vendor, Admin)
- **Venue Management**: Browse, search, and book wedding venues
- **Guest Management**: Manage guest lists, track RSVPs, and manage dietary requirements
- **Task Management**: Create, assign, and track wedding tasks with notifications
- **Vendor Management**: Manage vendors, contracts, and payment status
- **Budget Management**: Track wedding expenses and stay within budget

## Technologies

### Backend
- Java 17
- Spring Boot 3.2.1
- Spring Security with JWT authentication
- Spring Data JPA
- PostgreSQL

### Frontend
- React
- Bootstrap / Material-UI
- Axios for API communication
- React Router for navigation

### Prerequisites
- Java 17 or higher
- PostgreSQL 12 or higher
- Node.js and npm (for the frontend)

### Database Setup
1. Create a PostgreSQL database named `wedding_manager`
2. Configure the database connection in `src/main/resources/application.properties`

### Running the Backend
```bash
cd wedding-manager
./mvnw spring-boot:run
```

### Running the Frontend
```bash
cd wedding-manager-frontend
```

```
npm install
npm start
```

## API Documentation

### Authentication
- `POST /api/auth/signup`: Register a new user
- `POST /api/auth/signin`: Authenticate a user and receive a JWT token

### Venues
- `GET /api/venues/public/available`: Get all available venues
- `GET /api/venues/public/search/city/{city}`: Search venues by city
- `GET /api/venues/public/search/state/{state}`: Search venues by state
- `GET /api/venues/public/search/capacity/{capacity}`: Search venues by capacity
- `GET /api/venues/public/{id}`: Get venue details by ID
- `GET /api/venues/owner`: Get venues owned by the current vendor
- `POST /api/venues`: Create a new venue (for vendors)
- `PUT /api/venues/{id}`: Update a venue (for venue owners)
- `DELETE /api/venues/{id}`: Delete a venue (for venue owners)

### Weddings
- `GET /api/weddings`: Get all weddings for the current couple
- `GET /api/weddings/{id}`: Get wedding details by ID
- `POST /api/weddings`: Create a new wedding
- `PUT /api/weddings/{id}`: Update a wedding
- `PUT /api/weddings/{id}/addCouple`: Add a partner to the wedding
- `DELETE /api/weddings/{id}`: Delete a wedding

### Guests
- `GET /api/guests/wedding/{weddingId}`: Get all guests for a wedding
- `GET /api/guests/wedding/{weddingId}/status/{status}`: Get guests by invitation status
- `GET /api/guests/{id}`: Get guest details by ID
- `POST /api/guests/wedding/{weddingId}`: Add a new guest to a wedding
- `PUT /api/guests/{id}`: Update a guest
- `PUT /api/guests/{id}/status/{status}`: Update a guest's invitation status
- `DELETE /api/guests/{id}`: Delete a guest
- `GET /api/guests/count/wedding/{weddingId}`: Get the total guest count for a wedding
- `GET /api/guests/count/wedding/{weddingId}/status/{status}`: Get the guest count by status

### Tasks
- `GET /api/tasks/wedding/{weddingId}`: Get all tasks for a wedding
- `GET /api/tasks/wedding/{weddingId}/status/{status}`: Get tasks by status
- `GET /api/tasks/wedding/{weddingId}/overdue`: Get overdue tasks
- `GET /api/tasks/assigned`: Get tasks assigned to the current user
- `GET /api/tasks/{id}`: Get task details by ID
- `POST /api/tasks/wedding/{weddingId}`: Create a new task
- `PUT /api/tasks/{id}`: Update a task
- `PUT /api/tasks/{id}/status/{status}`: Update a task's status

- `DELETE /api/tasks/{id}`: Delete a task

### Vendors
- `GET /api/vendors/wedding/{weddingId}`: Get all vendors for a wedding
- `GET /api/vendors/wedding/{weddingId}/type/{type}`: Get vendors by type
- `GET /api/vendors/wedding/{weddingId}/payment/deposit/{isPaid}`: Get vendors by deposit payment status
- `GET /api/vendors/wedding/{weddingId}/payment/full/{isPaid}`: Get vendors by full payment status
- `GET /api/vendors/{id}`: Get vendor details by ID
- `POST /api/vendors/wedding/{weddingId}`: Add a new vendor to a wedding
- `PUT /api/vendors/{id}`: Update a vendor
- `PUT /api/vendors/{id}/payment/deposit/{isPaid}`: Update a vendor's deposit payment status
- `PUT /api/vendors/{id}/payment/full/{isPaid}`: Update a vendor's full payment status
- `DELETE /api/vendors/{id}`: Delete a vendor

## Security

The application uses JSON Web Tokens (JWT) for authentication. Protected endpoints require a valid JWT token in the Authorization header.

Testing

Instructions: Week 10

Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.
    - We have not changed any requirement
- List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.

1. Class: AuthController.java (Located in com.wedding.manager.controller or similar)

    Purpose: Handles HTTP requests for user authentication (signup, signin).

    - Methods to Test & Focus:
        - registerUser(...) or signup(...) :
            - Equivalence Classes (EC):
                - Valid registration (unique username/email, all fields correct).
                - Duplicate username.
                - Duplicate email.
                - Missing required fields (e.g., username, password).
                - Invalid email format.
                - Password doesn't meet criteria (if any).
            - Boundary Values (BV): Min/max length for username, password, email (if defined).
            - Paths: Successful creation; username exists; email exists; validation failure.
        - authenticateUser(...) or signin(...) :
            - EC: Valid credentials; invalid password; non-existent username; empty credentials.
            - Paths: Successful authentication (JWT returned); authentication failure (bad credentials/user not found).

2. Class: UserDetailsServiceImpl.java (Located in com.wedding.manager.security.service)

    Purpose: Implements Spring Security's UserDetailsService to load user data for authentication.

    - Method to Test & Focus:
        - loadUserByUsername(String username):
            - EC: Username exists; username does not exist; username is null/empty.
            - Paths: User found and UserDetails returned; UsernameNotFoundException thrown.

3. Class: JwtUtils.java (Located in com.wedding.manager.security.jwt)

    Purpose: Generates, validates, and parses JWTs.

    - Methods to Test & Focus:
        - generateJwtToken(Authentication authentication):
            - EC: Valid Authentication object.
            - Paths: Successful token generation.
        - getUserNameFromJwtToken(String token):
            - EC: Valid token; malformed token; expired token; incorrectly signed token; empty/null token.
            - Paths: Username extracted; exception/error due to invalid token.

        - validateJwtToken(String authToken):
            - EC: Valid token; malformed token; expired token; unsupported token; incorrectly signed token; empty/null token.
            - Paths: Token valid (returns true); token invalid due to various reasons (logs error, returns false).
    - BV (for token validation/generation): Token expiration (just before/after); validity of app.jwtSecret and app.jwtExpirationMs from properties.

4. Class: A representative Service class for a core feature, e.g., GuestServiceImpl.java **Purpose: Handles business logic for Guest management (CRUD operations).**

    - Methods to Test & Focus (Example: addGuestToWedding(GuestDto guestDto, Long weddingId)):
        - EC:
            - Valid guest details, valid weddingId.
            - Invalid guest details (e.g., missing name).
            - weddingId does not exist.
            - User (e.g., couple) does not have permission to add guests to the specified wedding.
        - BV: Guest name length; number of guests allowed per wedding (if a limit exists).
        - Paths: Guest successfully added; validation error for guest details; wedding not found; authorization failure.
    - General Testing Notes for these Classes:
        - Dependencies/Mocks: When unit testing services or controllers, dependent classes (like repositories, other services, JwtUtils) should be mocked to isolate the class under test.
        - Exception Handling: Test paths where exceptions are expected (e.g., UsernameNotFoundException, custom business exceptions like WeddingNotFoundException) and ensure they are handled gracefully, returning appropriate error responses.
        - Security Context: For methods in services or controllers that rely on the authenticated user (e.g., SecurityContextHolder.getContext().getAuthentication()), ensure tests set up a mock security context.

- Other notes:
    - None

Testing Details

1. AuthControllerTests.java: Unit tests AuthController for user signup and signin API endpoints. Verifies request mapping, input validation (mocked), interaction with services, and correct HTTP success/error responses.
2. UserDetailsServiceImplTests.java: Unit tests UserDetailsServiceImpl for loading user data during authentication. Verifies retrieval of existing users and correct exception handling for non-existent users.
3. JwtUtilsTests.java: Unit tests JwtUtils for JWT generation, parsing, and validation. Verifies handling of valid and invalid (malformed, expired, bad signature) tokens.
4. ServiceTests.java (e.g., GuestServiceTests.java, VenueServiceTests.java): Unit tests for service classes of respective features. Verify business logic, CRUD operations, input validation, and interaction with repositories (mocked).
5. ControllerTests.java (e.g., GuestControllerTests.java, VenueControllerTests.java): Unit tests for controller classes of respective features. Verify API endpoint mapping, request/response handling, interaction with services (mocked), and security rule enforcement (e.g., role-based access) at the controller level

Presentation

Instructions:Week 12

Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

- Give a brief description of your final project
    - Our final project is a "Wedding Manager," a full-stack web application designed to assist users (Couples, Admins like Abel, and Vendors) in planning and managing wedding events. It features a Spring Boot (Java) backend with PostgreSQL for data persistence and a React (JavaScript) frontend. The core aim is to streamline planning, reduce errors, and improve the overall experience.
- Describe your requirement assumptions/additions.
    - Users have basic web literacy. A stable internet connection is required. The primary user, Abel, requires admin oversight.
    - Implemented distinct user roles (Couple, Vendor, Admin) with role-specific dashboards and functionalities. Added JWT-based security for API authentication and authorization. Included placeholder pages for all core features identified.
- Describe your design options and decisions. How did you weigh the pros and cons of the different designs to make your decision?
    - Architecture:
        - Option 1 (Monolith): Simpler initial setup. Con: Less scalable, harder to maintain for distinct frontend/backend technologies.
        - Option 2 (Microservices): Highly scalable, independent deployment. Con: Increased complexity, more overhead for this project size.
        - Option 3 (Client-Server with distinct Frontend/Backend - N-Tier): Good separation of concerns, allows specialized tech stacks (React for UI, Spring Boot for backend logic), moderately scalable. Con: Requires managing two separate deployments.
            - Decision: We chose Option 3 (Client-Server/N-Tier). This provides the best balance of modularity, maintainability, and allows using the strengths of React for a dynamic UI and Spring Boot for robust backend services and data management with PostgreSQL. The pros of clear separation and technology fit outweighed the cons of managing two codebases for a project of this nature.

      Database:
            - Option 1 (NoSQL e.g., MongoDB): Flexible schema. Con: Less ideal for relational data like users, bookings, and defined entities.
            - Option 2 (SQL e.g., PostgreSQL, MySQL): Strong relational integrity, ACID compliance. Con: Requires schema definition upfront.
            - Decision: We chose PostgreSQL (SQL). Its relational nature is well-suited for structured wedding data (users, events, bookings, vendors, tasks, budget items), and it offers robust features.

- How did the extension affect your design?
    1. Initial errors (like "Module not found" in frontend, router issues, database connection problems in backend) required focused debugging and configuration changes.

2. Integrating PostgreSQL involved modifying application.properties, ensuring correctJDBC drivers and Hibernate dialects, and adjusting pom.xml.
3. Security configurations (WebSecurityConfig.java, JwtUtils.java) were refined to correctly handle public endpoints (like signup) and secure protected ones, addressing 401 Unauthorized errors.
4. The design remained largely N-Tier, but the implementation details within each tier were iterated upon to achieve stability and correct functionality.

- Describe your tests (e.g., what you tested, equivalence classes).
    - AuthControllerTests: Tested user signup/signin APIs.
        - ECs: Valid/invalid inputs (duplicate username/email, missing fields, bad credentials).
    - UserDetailsServiceImplTests: Tested user data loading for authentication.
        - ECs: Existing/non-existing users.
    - JwtUtilsTests: Tested JWT generation and validation.
        - ECs: Valid/invalid tokens (malformed, expired, bad signature).
    - Black-box API testing was performed using Postman for user registration and login flows (TC001-TC004), verifying HTTP responses and basic data persistence. Equivalence classes for these API tests included valid data, missing fields, and invalid credentials.
    -
- What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
    - Programming Concepts: Reinforced the importance of robust security configurations (Spring Security), correct API design for client-server interaction, and proper ORM (Hibernate/JPA) setup for database communication. Debugging configuration issues (properties files, dependencies) is a key skill.
    - Software Process (SDLC): Experienced the iterative nature of development ,identifying issues (bugs, configuration errors), debugging, fixing, and re-testing is crucial. The value of clear requirements (even if initially broad) and a structured testing plan (like the test cases) became evident. Early and frequent testing of core functionalities (like authentication) is vital.

- What functionalities are you going to demo?
    - Given the current state and what has been confirmed working:
        - User Registration: Show a successful new user signup via frontend
        - User Login: Demonstrate a successful login with the registered user via frontend,
        - Database Verification : Briefly show the created user and roles in pgAdmin to confirm backend-database integration.
        - Frontend : Navigate to placeholder pages created for core features to show the basic routing and UI structure is in place (e.g., Dashboard, Venue List).

- Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes of the presentation.)
    - There are 6 people in the group, each member will speak approximately 2 minutes relating to their role in the project.
- Other notes:
    - none