SOFTWARE DESIGN SPECIFICATION

ConnectTree Application

Version 1.0

Jordan Kelley
Buchard Joseph
David Martinez

November 8, 2024

Contents

1 System Description

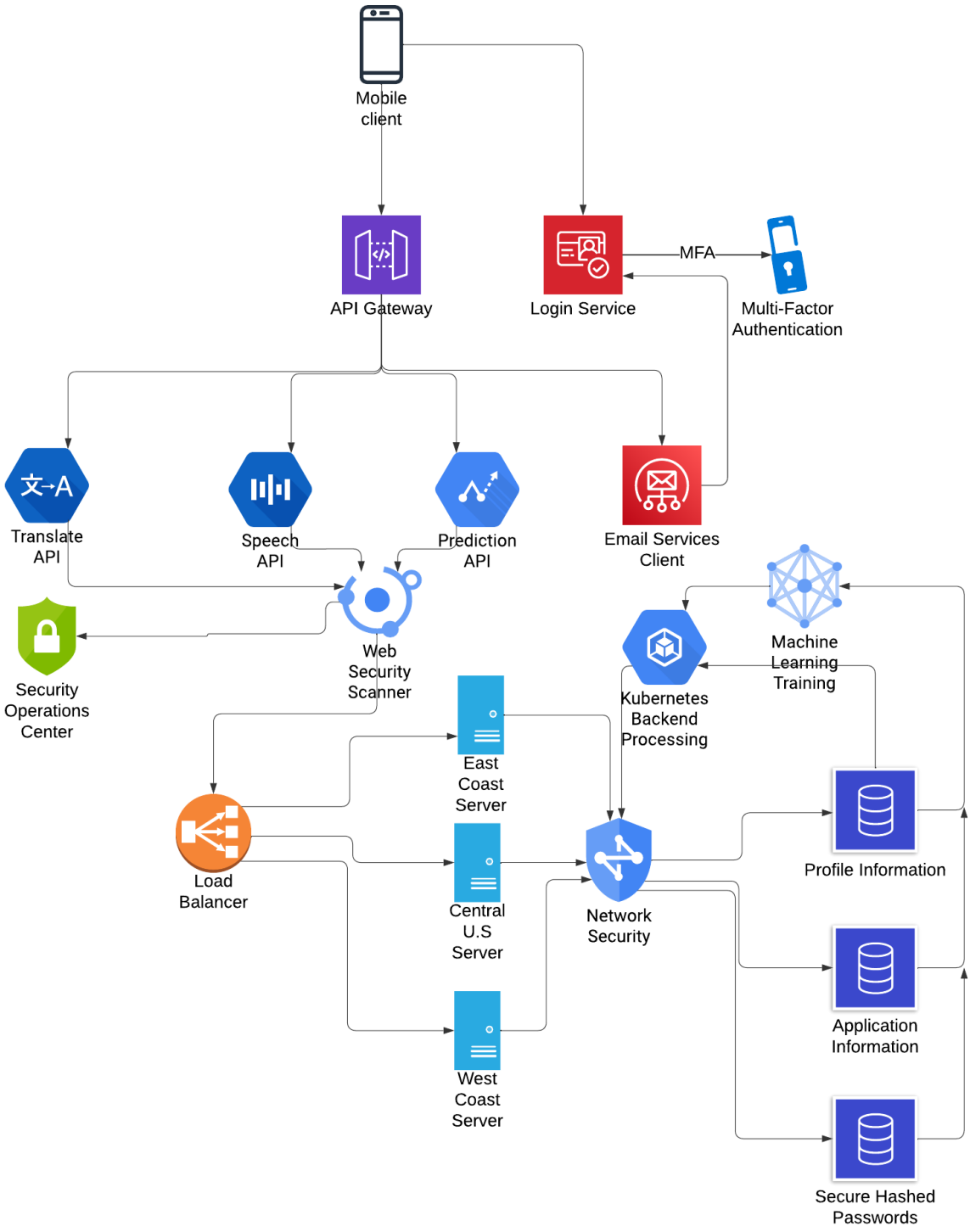
Purpose

This system is designed to create a networking tool for both professionals and the everyday person. Unlike other networking tools, this app aims to be less focused on the social media aspect of things and rather having a social network within easy grasp. Different views will allow the user to view their contacts as lists, categories, and even a broad web of connected bubbles.

Target Audience

The primary audience for the application will be professionals looking to have a structured and organized network of contacts. However, the secondary audience will be application to a normal customer base.

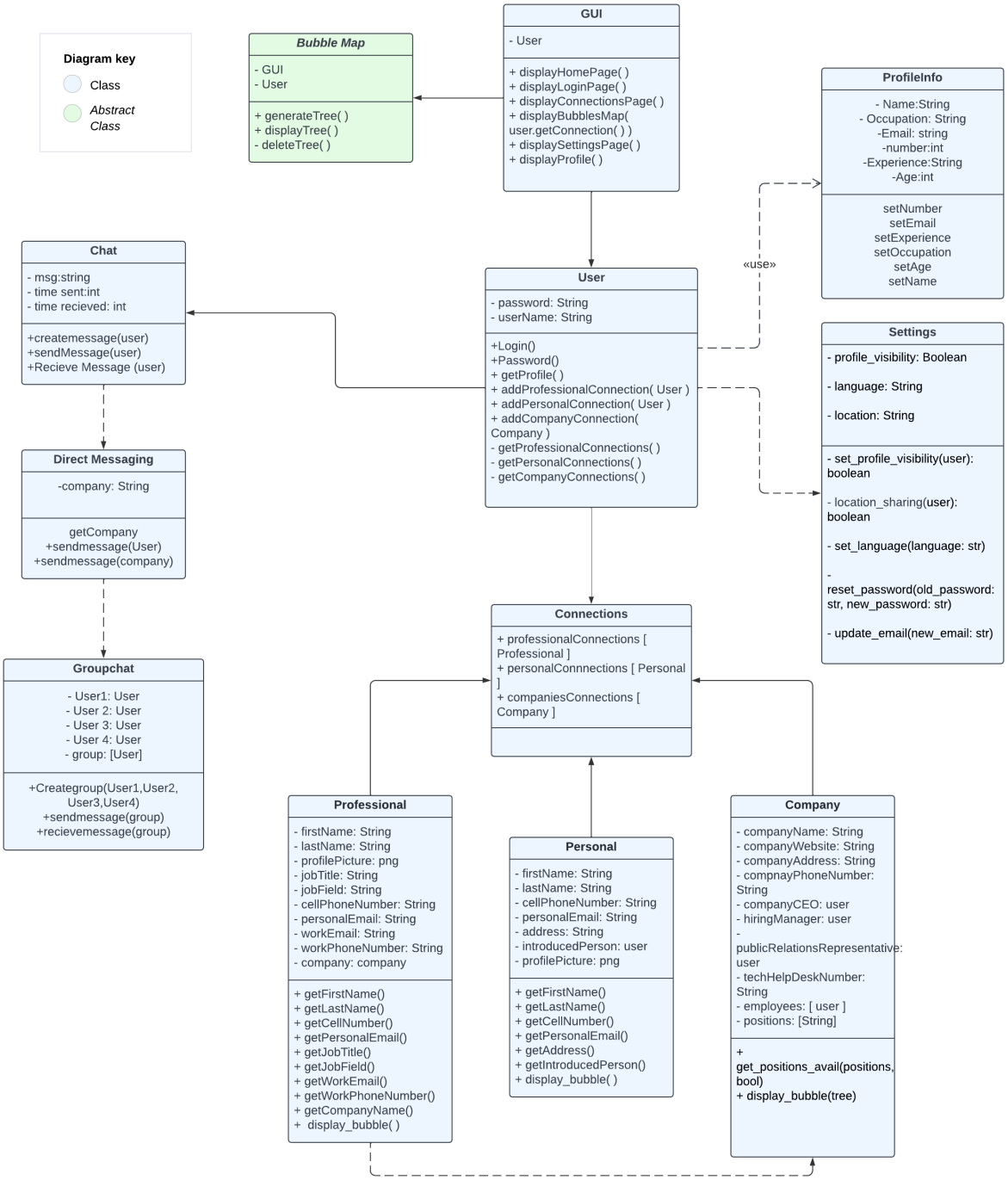
2 Software Architecture Overview



2.1 Architecture Diagram Descriptions

- User: User first logs in through the login service. From there the login service will use a Multi-Factor Authentication (MFA) for better security practices. Once the user has been verified, the login service will then return a valid certificate to the user device to then be used for access to the system.
- API Gateway: Once the user has a valid certificate, they get connected to the API gateway. This gateway will then connected the user to multiple API's to handle different tasks:
  - Translate API: Translates text from users through a 3rd party translation service.
  - Speech API: Takes speech from the user and transposes it to text.
  - Prediction API: Uses users previous system interactions to predict what they may do next.
  - Email Services Client: Connects User to outside email client.
- Web Security Scanner: After the API(S) perform their task, the traffic is then routed through a security software. If there exist any threat, the traffic is halted and a log is sent to the Security Operations Center (SOC).
  - SOC: This will be a third party security monitoring service that will access threats and then send sercurity patches to the web security scanner as needed.
- Load Balancer: This will take all incoming traffic and distriubte the traffic amongst three servers based on geographic location and server load.
  - East Coast Server: Handles all traffic with IP address from the east of 90 Degrees West.
  - Central U.S Server: Handles all traffic with IP address between 110 Degrees West and 90 Degrees West. This will also be the first server to take extra load from East and West Servers.
  - West Coast Server: Handles all traffic with IP address West of 110 Degrees West.
- Second Network Security Scanner: After being assigned a server, traffic between servers and Databases will require an additional security scan and encryption.
- Databases:
  - Profile Information: This database should hold all profiles and their relevant information.
  - Application Information: This database should hold all information relevant to the application framework. Think images, algorithms, API information, data analysis, etc...
- Machine Learning Training: This will take information from user interactions with the databases and then try and develop better data processing / storage methods and try and figure out how to analyze user interactions with the system.
- Kubernetes Backend Processing: Performs application rendering of information on the server side rather than the client side. This allows for the app to work with different levels of client software and neglects the need to consider client hardware capabilities.

2.2 ULM Class Diagram



3 Description of Classes, Attributes, and Operations

3.1 User

This class represents a system user with personal login capabilities.

Attributes

- **password (String)**: The user's password for login.
- **userName (String)**: The username for login.

Operations

- **Login()**: Authenticates the user.
- **Password()**: Verifies the user's password.
- **getProfile()**: Retrieves the user's profile information.

3.2 Connections

Handles the different types of connections a user can have.

Attributes

- **professionalConnections (List¡User¿)**: A list of professional connections.
- **personalConnections (List¡User¿)**: A list of personal connections.
- **companyConnections (List¡Company¿)**: A list of company connections.

Operations

- **addProfessionalConnection(User user)**: Adds a professional connection.
- **addPersonalConnection(User user)**: Adds a personal connection.
- **addCompanyConnection(Company company)**: Adds a connection to a company.
- **getProfessionalConnections()**: Returns a list of professional connections.
- **getPersonalConnections()**: Returns a list of personal connections.
- **getCompanyConnections()**: Returns a list of company connections.

3.3 Settings

Manages user settings related to privacy and preferences.

Attributes

- **profile_visibility (bool)**: Indicates if the profile is visible to others.
- **language (String)**: Preferred language setting.
- **location (String)**: User's location.

Operations

- **set_profile_visibility(bool visibility)**: Sets the visibility of the user's profile.
- **location_sharing(bool shareLocation)**: Shares or hides the user's location.
- **set_language(String language)**: Sets the user's preferred language.
- **reset_password(String old_password, String new_password)**: Resets the user's password.
- **update_email(String new_email)**: Updates the user's email.

3.4 ProfileInfo

Contains detailed personal information about the user.

Attributes

- **Name (String)**: User's full name.
- **Occupation (String)**: User's current occupation.
- **Email (String)**: User's email address.
- **Number (String)**: User's phone number.
- **Experience (String)**: User's work experience.
- **Age (int)**: User's age.

Operations

- **setName(String name)**: Updates the user's name.
- **setOccupation(String occupation)**: Updates the user's occupation.
- **setEmail(String email)**: Updates the user's email address.
- **setNumber(String number)**: Updates the user's phone number.
- **setExperience(String experience)**: Updates the user's experience.
- **setAge(int age)**: Updates the user's age.

3.5 Personal

Details a personal connection; most of its functions are derived from personal info.

Attributes

- **personalConnectionName (String)**: Name of the personal connection.
- **cellPhoneNumber (String)**: The number of your personal connection, assuming they have made their profile.
- **personalEmail (String)**: Email address of the personal connection.
- **address (String)**: Home address.
- **introducedPerson (String)**: Person who introduced the connection.
- **profilePicture (Image)**: Profile picture of the connection.

Operations

- Getters for each attribute (e.g., **getPersonalConnectionName()**, **getCellPhoneNumber()**, etc.).
- **display_bubble()**: Displays user details visually.

3.6 Company

Represents a company in the user's network.

Attributes

- Attributes derived from **ProfileInfo**, with additional details:
  - **CEO (String)**: Name of the CEO.

Operations

- **get_positions_avail(List¡String¿ positions, bool isAvailable)**: Shows available positions within the company.
- **display_bubble(Tree tree)**: Displays information in a visual format.

3.7 Professional

Details a professional connection.

Attributes

- Professional and contact details similar to those found in **Personal**.

Operations

- Getters for professional details.
- **displayDetails()**: Displays these details visually.

3.8 Direct Messaging

Facilitates messaging between users.

Attributes

- **recipient (String)**: Registered user you want to send a message to.
- **company (Company)**: Company to which you want to send a message.

Operations

- **getRecipient()**: Retrieves details about the message recipient.
- **getCompany()**: Retrieves details about the company recipient.
- **sendMessage(String message)**: Sends a message to either a company or an individual.

3.9 GUI

Manages the graphical user interface components.

Operations

- Methods to display different user interface pages like **home**, **login**, **connections**, **settings**, and **profile** pages.

3.10 Groupchat

Manages messaging within a group.

Attributes

- **recipientsList (List¡String¿)**: Recipients' names, which should be registered names from **ProfileInfo**.
- **groupIdentifier (String)**: Group chat name.

Operations

- **createGroup(String groupName, List¡String¿ members)**: Creates a new group chat.
- **sendMessage(String message)**: Sends a message to the group.
- **receiveMessage()**: Receives messages in a group.

3.11 Bubble Map

Visual representation of connections or data.

Operations

- **generateTree()**: Generates a hierarchical tree structure.
- **displayTree()**: Displays the tree visually.
- **deleteTree()**: Deletes the tree structure.

3.12 Chat

Manages chat messages.

Attributes

- **msg (String)**: Contains the message.
- **timeSent (DateTime)**: Timestamp of when the message was sent.
- **timeReceived (DateTime)**: Timestamp of when the message was received.

Operations

- **createMessage(String message)**: Creates a new message.
- **sendMessage()**: Sends the message.
- **receiveMessage()**: Receives a message.

4 Development Plan and Timeline

Stage	Tasks	Assigned to	Duration (Weeks)
1. Requirements Analysis	Define app objectives, user interactions, and connection flow (user profiles, professional, personal connections)	Developer 1	1
	Gather and document detailed functional and non-functional requirements for ConnectTree	Developer 2	1
	Identify constraints, platform dependencies, security considerations for connections	Developer 3	1
2. System Design	Create UI/UX mockups, including connection bubbles, user interfaces	Developer 1	2
	Define software architecture, class diagrams, and data structures for tree connections	Developer 2	2
	Define API contracts for managing connections and messaging, database schema design	Developer 3	2
3. Implementation	Develop front-end, including user interfaces and bubble map navigation for connections	Developer 1	4
	Implement back-end services for handling professional, personal, and company connections	Developer 2	4
	Set up database integration, user authentication, and real-time messaging services	Developer 3	4
4. Integration and Testing	Unit testing of UI components and navigation	Developer 1	2
	Integration testing between front-end, back-end services, and database	Developer 2	2
	System testing, ensure compliance with App Store guidelines and security tests for connections and messaging	Developer 3	2
5. Deployment	Prepare ConnectTree for submission to App Store	Developer 1	1
	Finalize and review all technical documentation and user manual	Developer 2	1
	Deploy app to TestFlight and collect feedback from beta testers	Developer 3	1
6. Maintenance	Monitor app performance, fix bugs, release updates based on user feedback	All Developers	Ongoing

5 Test Plan

5.1 Test Set 1

Unit Test

Feature: Testing the getProfessionalConnections() method of User Class.
Test vector: Create an object of Connections using a constructor with a initialized array of professional connections, Call method as User.getProfessionalConnections()
Expected result: The Method should return the expected array of Professional Connections.

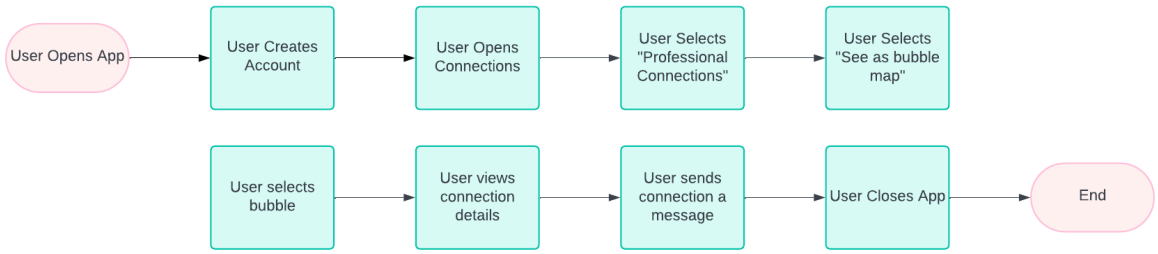
User Joe = new User( );
Connections professionalConnections [ ] = Connection1, Connection2 ;
assertEquals(Joe.getProfessionalConnections( ), professionalConnections);

Functional Test

Feature: Testing Login Service
Test vector: Create a User with a password.
One: pass correct User Password and Username to login service. Check login results.
Two: pass incorrect User Password and Correct UserName to login service. Check login results.
Three: pass correct User Password and Incorrect UserName to login service. Check login results.
Four: pass Incorrect User Password and Incorrect UserName to login service. Check login results.

String userName = Joe1244;
String password = 657abc;
User Joe = new User(userName, password);
assertEquals(Joe.logon(userName, password)); // Expected True
assertEquals(Joe.logon(userName, 657cba)); // Expected False
assertEquals(Joe.logon(Joe4421, password)); // Expected False
assertEquals(Joe.logon(Jane123, 4431Bds)); // Expected False

System Test



5.2 Test Set 2

Unit Test

Feature: getJobField() method of the user class
Test vector: Provides the current user or selected users job occupation/ title
Expected result: The method should return the Field of work the user is in

User Greg = new user()
function testgetJobTitle(): professional = new Professional()
professional.setJobTitle("Backend Developer")
assert professional.getJobTitle() == "Backend Developer"

Functional Test

Feature: Testing group chat feature
Test vector: Creating a group chat with various user counts

1st Test: group chat with two users groupChat = new GroupChat()
groupChat.createGroup(User1, User2)
result = groupChat.sendMessage("User1", "Hello User2!")
assert result == true
message = groupChat.receiveMessage("User2")
assert message == "Hello User2!"

2nd Test: A group with 4 users // should be created successfully, and all users can send/ receive messages
groupChat = new GroupChat()
groupChat.createGroup(User1, User2, User3, User4)
result =groupChat.sendMessage("User3", "Hi everyone!")
assert result == true
for user in [User1, User2, User4]:
message =groupChat.receiveMessage(user)
assert message == "Hi everyone!"

3rd test: Group with no user// should fail to create and return an error
groupChat = new GroupChat()
try: groupChat.createGroup() No users provided assert false Group chat creation should not succeed
Except Exception as e:
assert str(e) == "Cannot create a group chat without users"

System Test

Feature: Testing contact management functionality, ensuring users can add, categorize, and view contacts seamlessly.

Test vector: Simulate a user adding contacts, categorizing them, and viewing them in list, category, and bubble formats. Ensure information is updated correctly across all views.

Expected Result: Users, after creating an account, can see a display where they can manage and update their profile and personal contact information at will

5.3 Test Set 3

Unit Test

Feature: Testing the setName(String name): Updates the user's name.
Test vector: Provide an unordered list as input and check if the output is correctly sorted.
Expected result: The method should return a sorted list, and you would also test edge cases like an empty list or a list with one element.

Functional Test

Feature: Testing generateTree(): Generates a hierarchical tree structure.
Test vector: Test the tree generation with different sets of professional and personal connections to ensure a variety of hierarchical structures are created.
Expected result: The method should generate a correct hierarchical tree structure for both professional and personal connections. The method should also handle edge cases, such as an empty tree, gracefully.

System Test

Feature: Verifying the display bubble(): Displays user details visually.
Test vector: Create multiple Personal or Company objects with varying attributes (e.g., different names, emails, phone numbers) and test how the details are presented in the visual bubble format.
Expected result: The method should correctly display the user's details (name, phone number, email, etc.) in a visual bubble format. It should ensure that all the information is visible and that the display adapts well to different data lengths or missing information.

6 SQL Database Implementation

6.1 Strategy

Given that this system necessitates the need to store sensitive information, such as user details, the system requires the need for a robust and secure data management strategy. In response, we have choose to use a SQL based data management strategy. Split among three separate relational databases, we can effectivity organize our data in a secure manner.

6.2 Why SQL?

After spending time in class seeing how SQL and non-SQL compared, it was obvious SQL was a clear winner.

- Structured Data Relationships: Through the use of relational databases (Profile, Application, and Hashed Passwords), data is secure and easy to manage within an organized manner.
- Complex Queries: Data retrieval and manipulation seemed more efficient when using the smaller set of SQL query instructions. This not only allows for better standardization, but allows different members to access the data easily.

6.2.1 Three Databases Over One

- Secure Data Storage: Sensitive information such as User information must be stored in a secure manager, which is why we chose to give it its own database. Similar to that reason, we choose to implement a separate database for hashed passwords, offering our users the ultimate protection from password leaks.
- Performance: Allowing there to be multiple databases enhances overall performance. Allowing each database to be optimized based on its expected workload, such as read-heavy operations on application data versus write-heavy operation on User data.
- Scalability: Already thinking of expansion of the application, allowing our databases to be scaled based on their usage patterns was of upmost importance. For instance, as our User audience grows, we can scale up our User database; while maybe not needing to expand the database for simple application data.

6.3 Trade-offs Analysis

SQL vs. NoSQL
Justification for SQL
Our need for complex transactions and strong data integrity aligns with SQL's strengths. The overhead of schema management in SQL is acceptable given our structured data. *Trade-off:* We sacrifice some scalability and flexibility that NoSQL offers but gain reliability and consistency.

Multiple Databases vs. Single Database

Justification for Multiple Databases:
Enhances security and performance.
Allows for specialized optimization of each database.
Trade-off:
Increased complexity in data management.
Potential overhead in ensuring data consistency across databases.