DIGT2107: Practice of Software Development

Project Iteration 3.1: Software Design Documentation

OpenBid

Team 1: Tyler Mani Yanness Alaister

Due: February 1, 2026

Course: DIGT2107 – Winter Term 2026 | **Instructor:** Dr. May Haidar

1 Introduction

Project Name: OpenBid
Team Number: 1
Team Members: Tyler, Mani, Yanness, Alaister

Document Overview This document outlines the Software Design Document (SDD) for Iteration 3.1 of the OpenBid platform. It provides advanced software design representations through Class Diagrams, Sequence Diagrams, and Activity Diagrams for all major system components:

- **Stripe KYC & User Profile** – Identity verification and profile management
- **Jobs & Bids** – Job posting and bidding functionality
- **Authentication & 2FA** – User authentication with Duo MFA
- **Payments & Escrow** – Stripe payment processing and escrow management

Each section includes class diagrams showing component structure, sequence diagrams illustrating key flows, activity diagrams depicting workflows, and design stories for the backlog.

2 Class Diagrams

2.1 User & Profile Domain

The following class diagram illustrates the core classes related to user management and profile functionality.

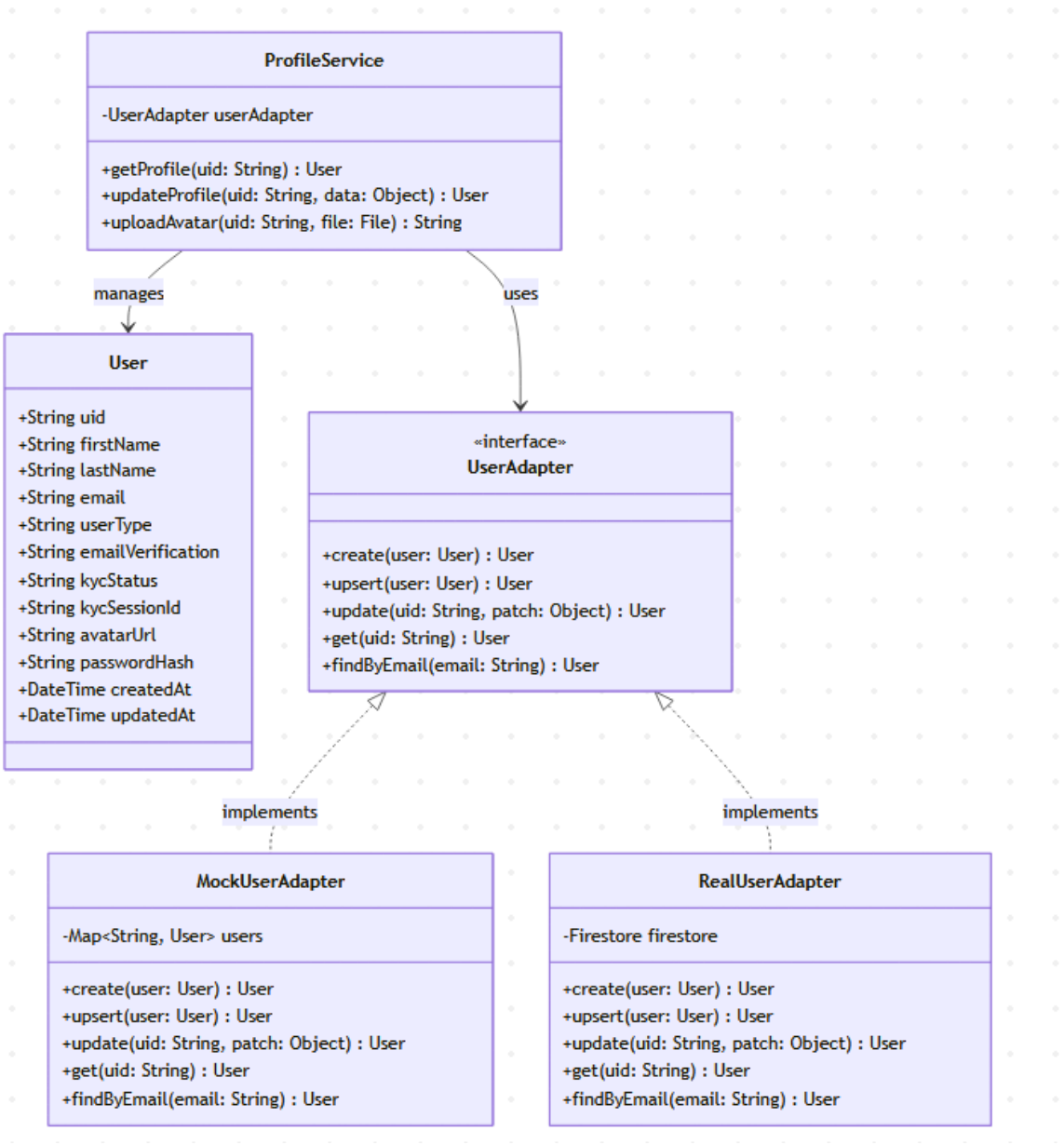

Figure 1: User & Profile Domain Class Diagram

2.2 Stripe KYC Domain

The following class diagram illustrates the classes related to Stripe Identity verification (KYC).

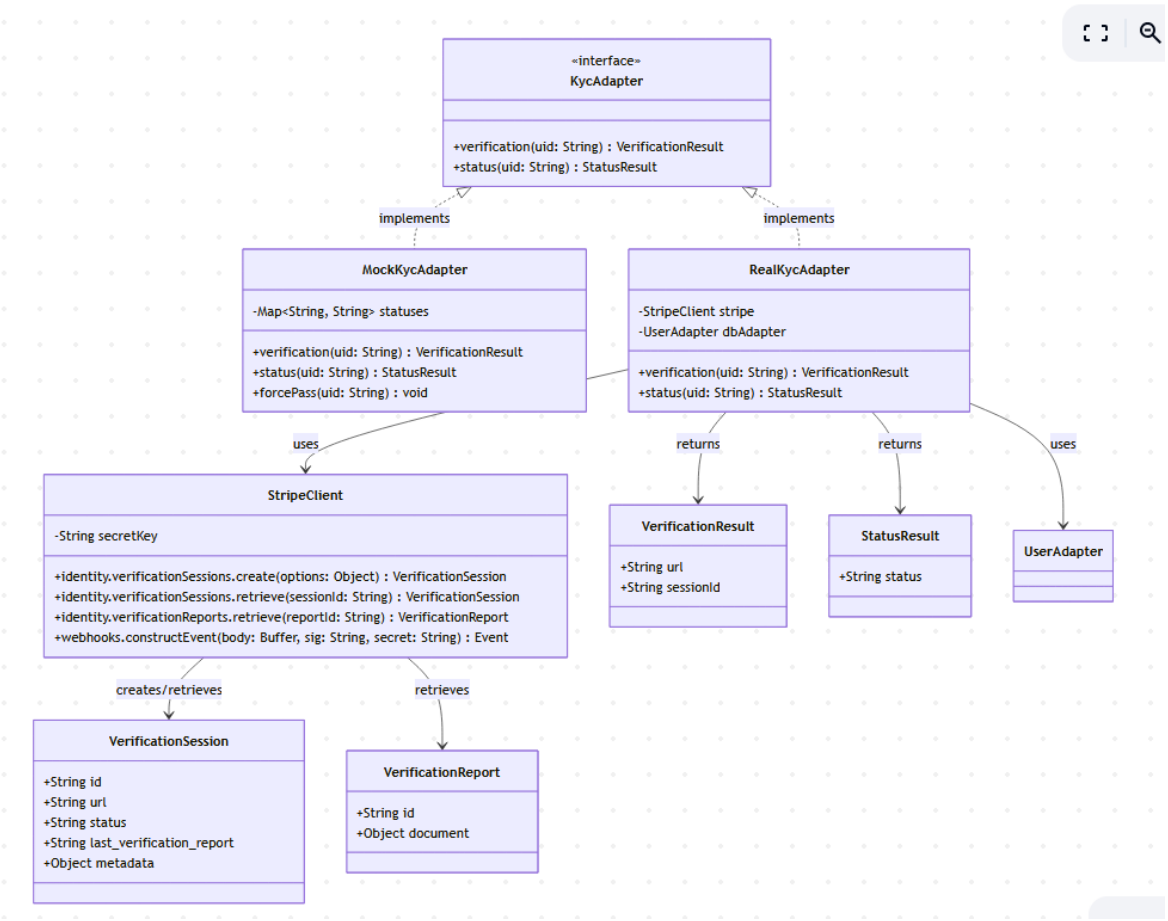

Figure 2: Stripe KYC Domain Class Diagram

2.3 Component Relationships

The following diagram shows how the KYC and Profile components interact with each other and external services.

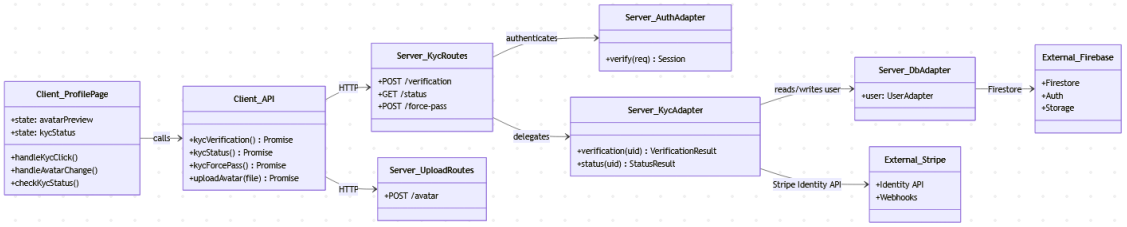

Figure 3: KYC and Profile Component Relationships

3 Sequence Diagrams

3.1 KYC Verification Flow - Part A: Initialization

This sequence diagram demonstrates the initial flow when a user starts KYC verification.

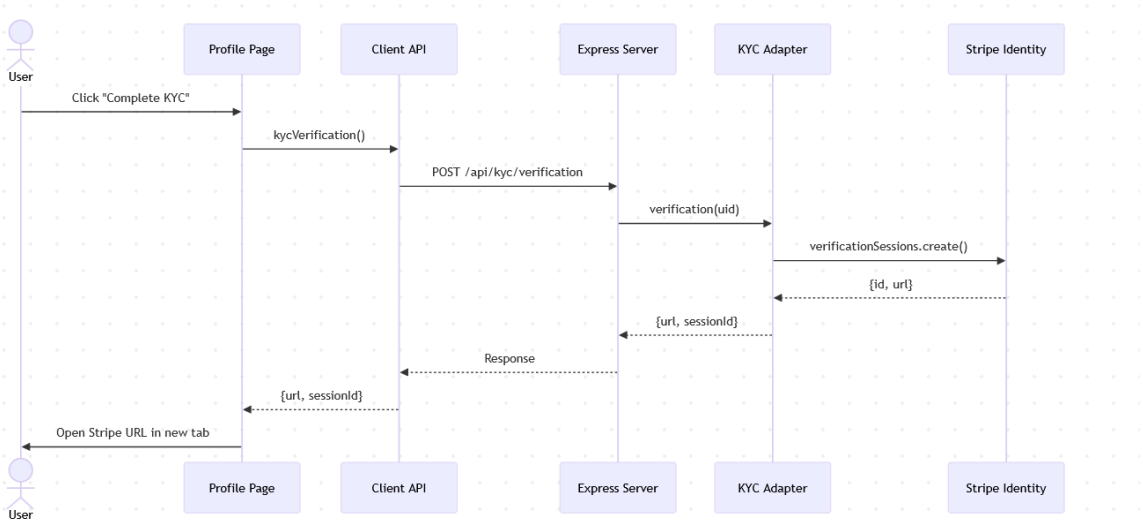

Figure 4: KYC Verification Initialization Flow

Flow Description:

1. User clicks "Complete KYC" button on Profile page.
2. Profile page calls `api.kycVerification()`.
3. Client API sends POST request to `/api/kyc/verification`.
4. Server delegates to KYC adapter's `verification()` method.
5. KYC adapter creates Stripe verification session.
6. Stripe returns session ID and verification URL.
7. Profile page opens Stripe verification in new tab.

3.2 KYC Verification Flow - Part B: Webhook & Status Check

This sequence diagram demonstrates the webhook handling and status check after user completes Stripe verification.

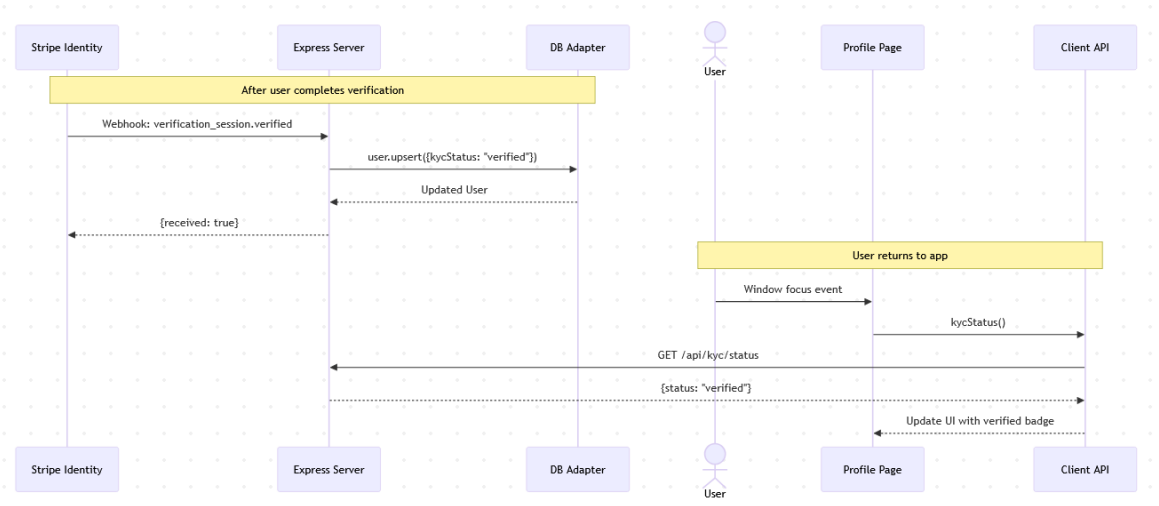

Figure 5: KYC Webhook and Status Check Flow

Flow Description:

1. After user completes verification, Stripe sends webhook event.
2. Server verifies webhook signature and updates user's KYC status.
3. When user returns to app, window focus event triggers status check.
4. Profile page fetches updated KYC status from server.
5. UI updates to show verified badge.

3.3 Profile Avatar Upload Flow

This sequence diagram demonstrates the profile avatar upload functionality.

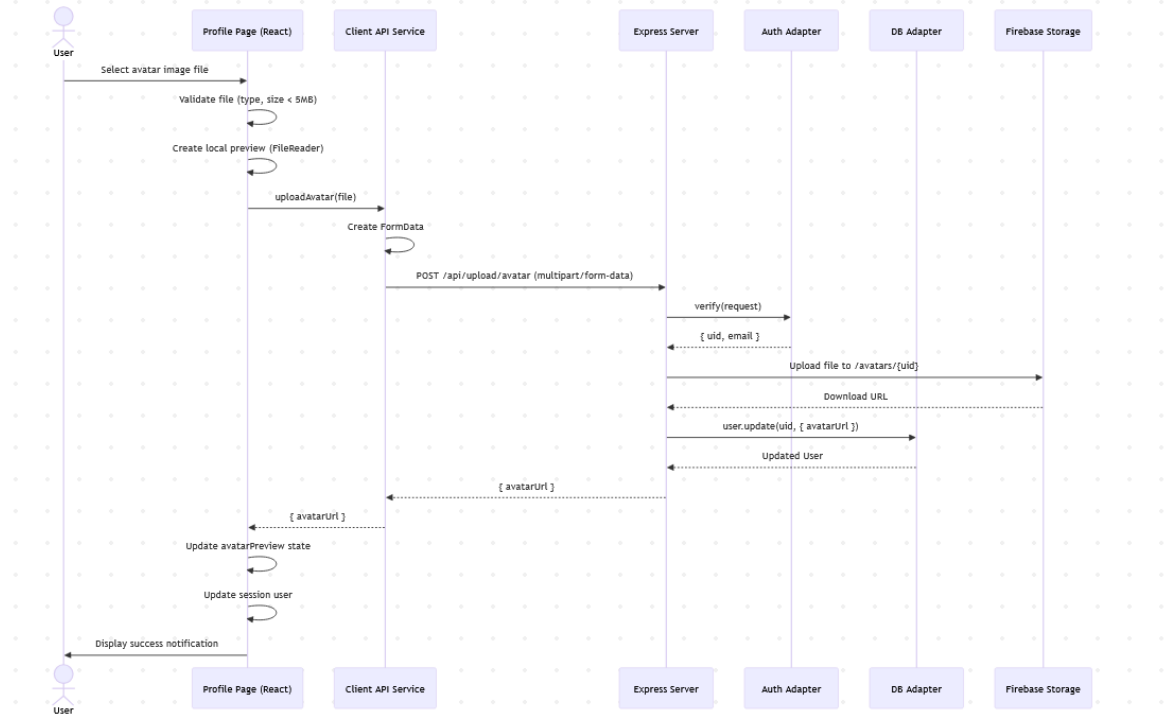

Figure 6: Avatar Upload Sequence Diagram

4 Activity Diagrams

4.1 KYC Verification Workflow

The following activity diagram illustrates the complete workflow for KYC verification with all decision points.

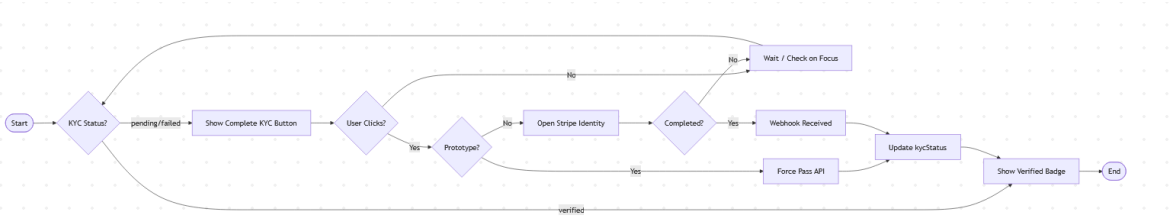

Figure 7: KYC Verification Activity Diagram

4.2 Profile Management Workflow

The following activity diagram illustrates the user profile management workflow.

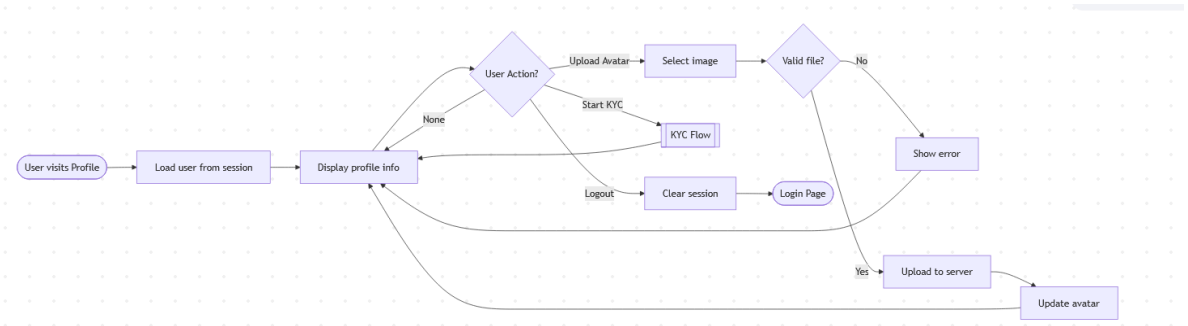

Figure 8: User Profile Management Activity Diagram

5 Component Summary

Component	Type	Technology	Responsibility
`Profile.jsx`	React Component	React, Carbon Design	User profile UI, avatar upload, KYC status
`kyc.routes.js`	Express Router	Node.js, Express	KYC API endpoints
`kyc.real.js`	Adapter	Stripe SDK	Real Stripe Identity integration
`kyc.mock.js`	Adapter	In-memory	Mock KYC for prototype mode
`index.js`	Server Entry	Express	Stripe webhook handler
`api.js`	Client Service	Fetch API	HTTP client for KYC/Profile calls

6 Updated Backlog with Design Stories

The KYC and Profile features documented above are complete. The following design stories outline the upcoming **In-App Messaging** feature for the next iteration:

Messaging Design Stories

1. **DS-MSG-001: Design Message Data Model** (High, 3 pts)
   Define message schema including sender, receiver, content, timestamp, read status, and job/bid thread linking. Design conversation thread structure for organizing messages between contractors and bidders.

2. **DS-MSG-002: Design Real-Time Architecture** (High, 5 pts)
   Evaluate and select real-time messaging approach: Firebase Realtime Database listeners vs Firestore onSnapshot vs WebSockets. Design message synchronization and offline support.

3. **DS-MSG-003: Design Chat UI Component** (Medium, 3 pts)
   Design chat interface with message bubbles, input field, send button, and typing indicators. Support for text messages and future attachment support.

4. **DS-MSG-004: Design Notification System** (Medium, 3 pts)
   Design push notification flow for new messages. Integrate with Firebase Cloud Messaging for real-time alerts when user is not in the app.

5. **DS-MSG-005: Design Job-Thread Linking** (High, 2 pts)
   Design how message threads are linked to specific jobs and bids. Ensure only awarded bidders can message contractors, and vice versa.

7 Preliminary Test Coverage Report

The following test coverage report was generated using Jest with the `-coverage` flag. This report covers the KYC and Profile-related components.

Test Results Summary

- **Test Suites:** 5 passed, 1 skipped, 6 total
- **Tests:** 33 passed, 2 skipped, 35 total
- **Snapshots:** 0 total

Coverage by Component (KYC & Profile)

File	% Stmts	% Branch	% Funcs	% Lines	
`kyc.routes.js`	87.09	92.85	100	89.28	
`kyc.real.js`	38.23	25.00	100	36.36	
`kyc.mock.js`	25.00	0.00	0	25.00	
`db.real.js` (User data)	64.23	43.10	71.42	72.41	
Overall	53.73	46.20	57.14	56.33	

Analysis

- **KYC Routes:** Excellent coverage at 89% line coverage. All API endpoints are tested.
- **KYC Real Adapter:** Lower coverage (36%) due to Stripe API integration being mocked during testing.
- **Database Adapter:** Good coverage at 72% for user data operations.
- **Gaps Identified:** The `kyc.mock.js` adapter has low coverage as it is primarily used for prototype mode testing only.

8 GUI Implementation Screenshots

The following screenshots demonstrate the Profile page implementation, showing the KYC verification status and user interface elements.

Profile Page - Before KYC Verification

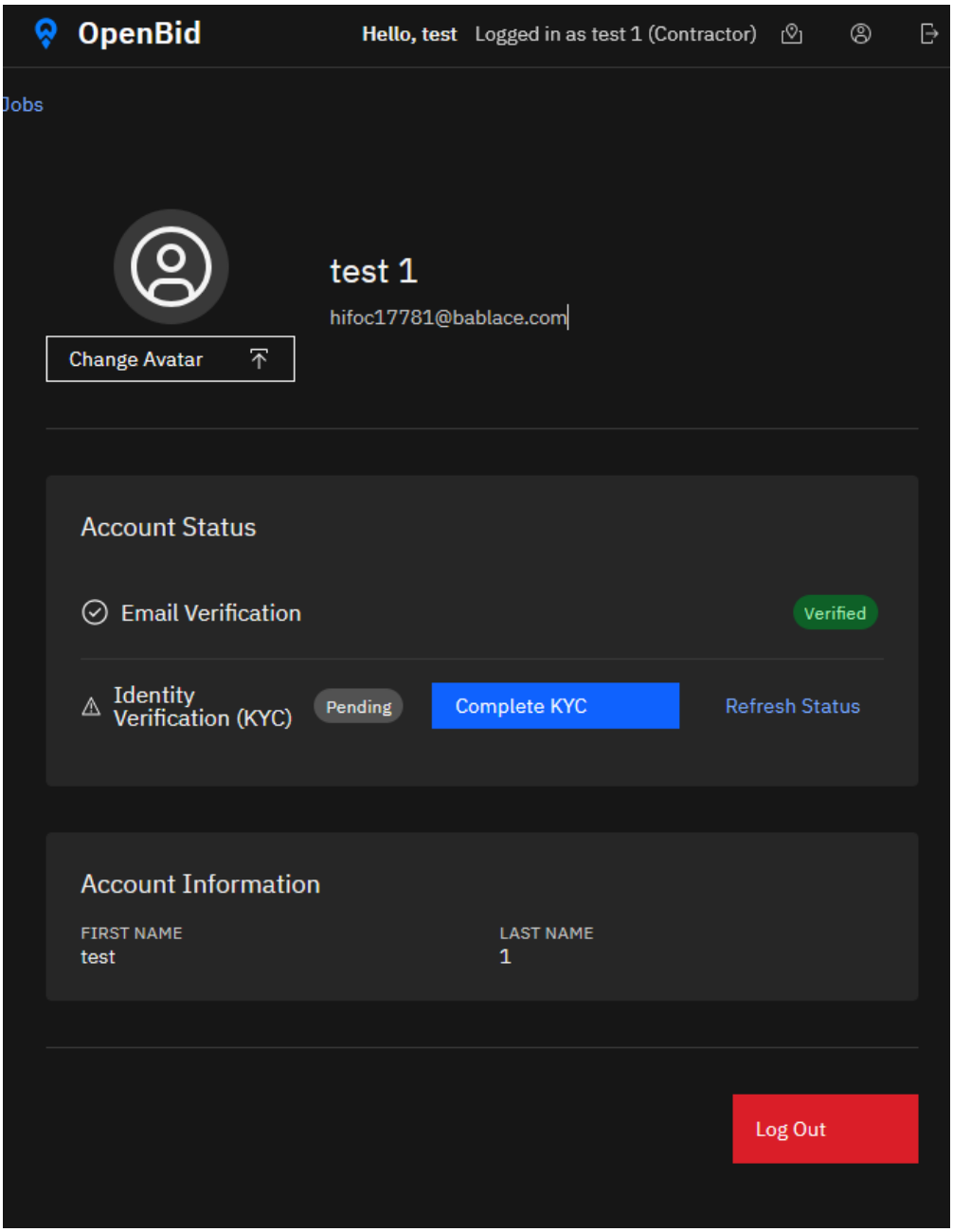

Figure 9: Profile page showing pending KYC status with "Complete KYC" button

Profile Page - After KYC Verification

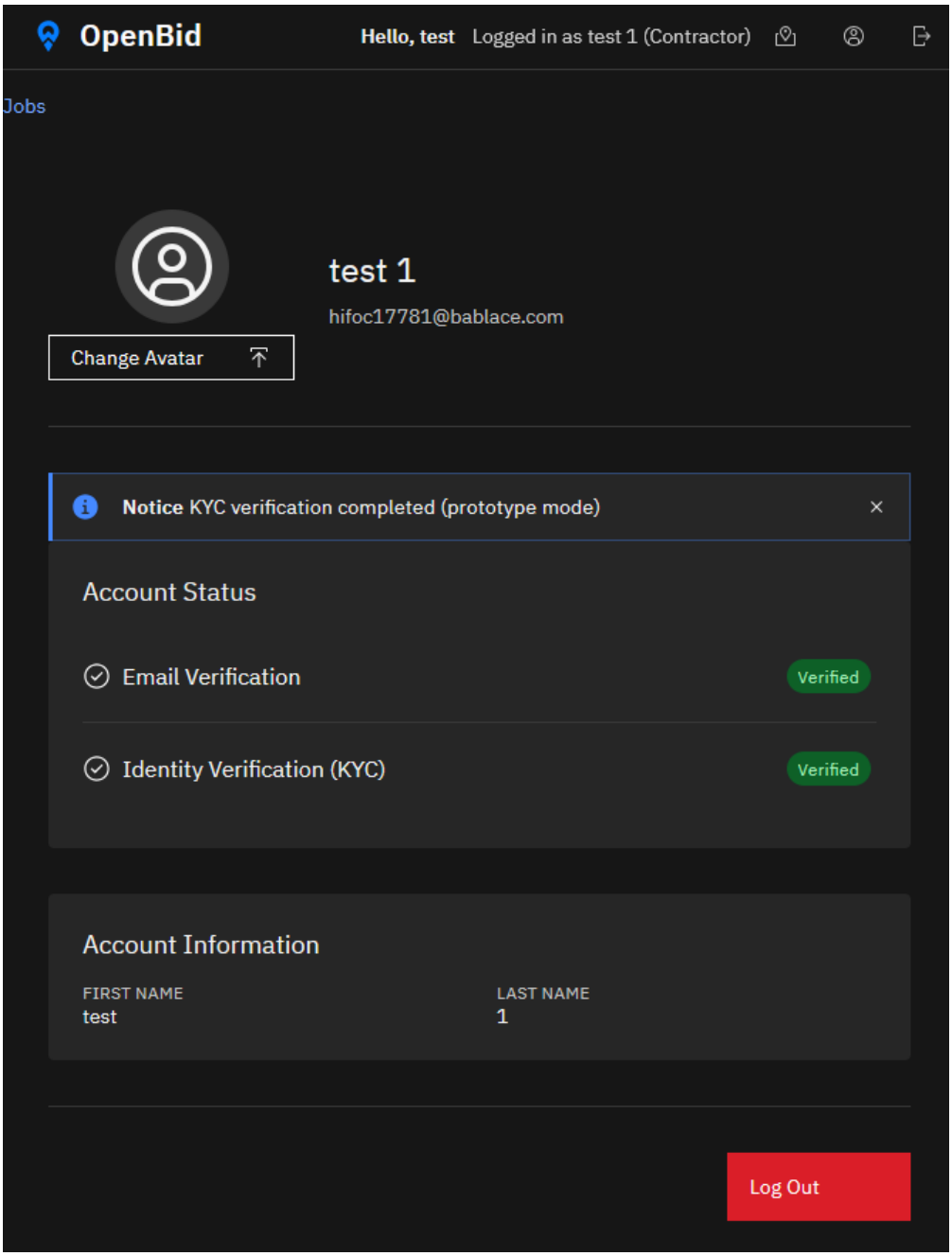

Figure 10: Profile page showing verified KYC status with green badge

Key UI Elements

- **Avatar Upload:** Users can upload and change their profile picture
- **KYC Status Display:** Visual indicator showing pending, verified, or failed status
- **Action Buttons:** "Complete KYC" and "Refresh Status" buttons for unverified users

- **Account Information:** Display of user's name, email, and verification status

Firestore Persistence: Users, Jobs & Bids

Overview

This section documents the Firestore-backed persistence layer that stores **user**, **job**, and **bid** information. The primary implementation is the server adapter `server/src/adapters/db.real.js`, which uses the Firebase Admin SDK to interact with the `users`, `jobs`, and `bids` collections.

Class Diagrams

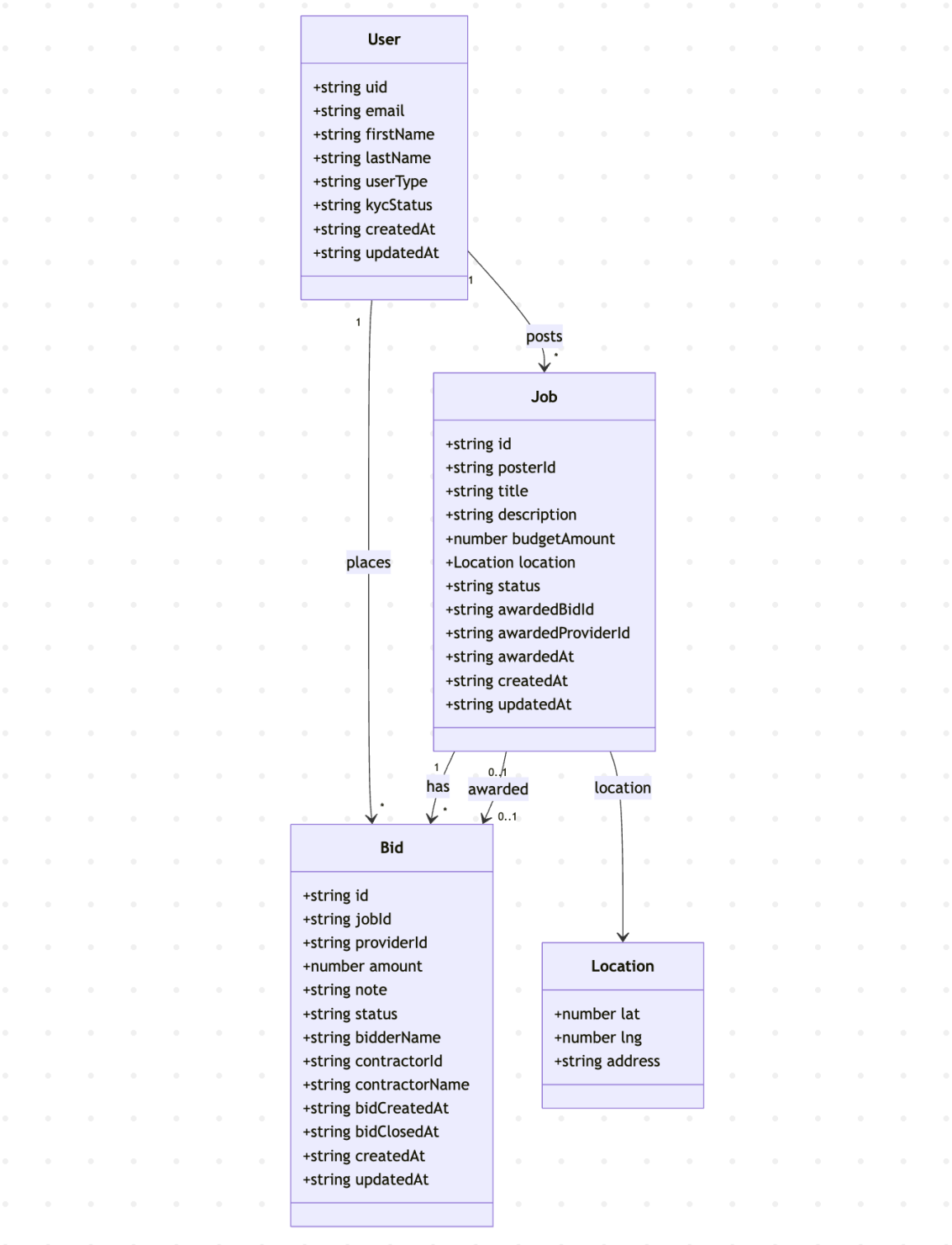

Figure 11: Users, Jobs, and Bids: Firestore Schema / Domain Entity Class Diagram

Firestore Collections & Domain Entities

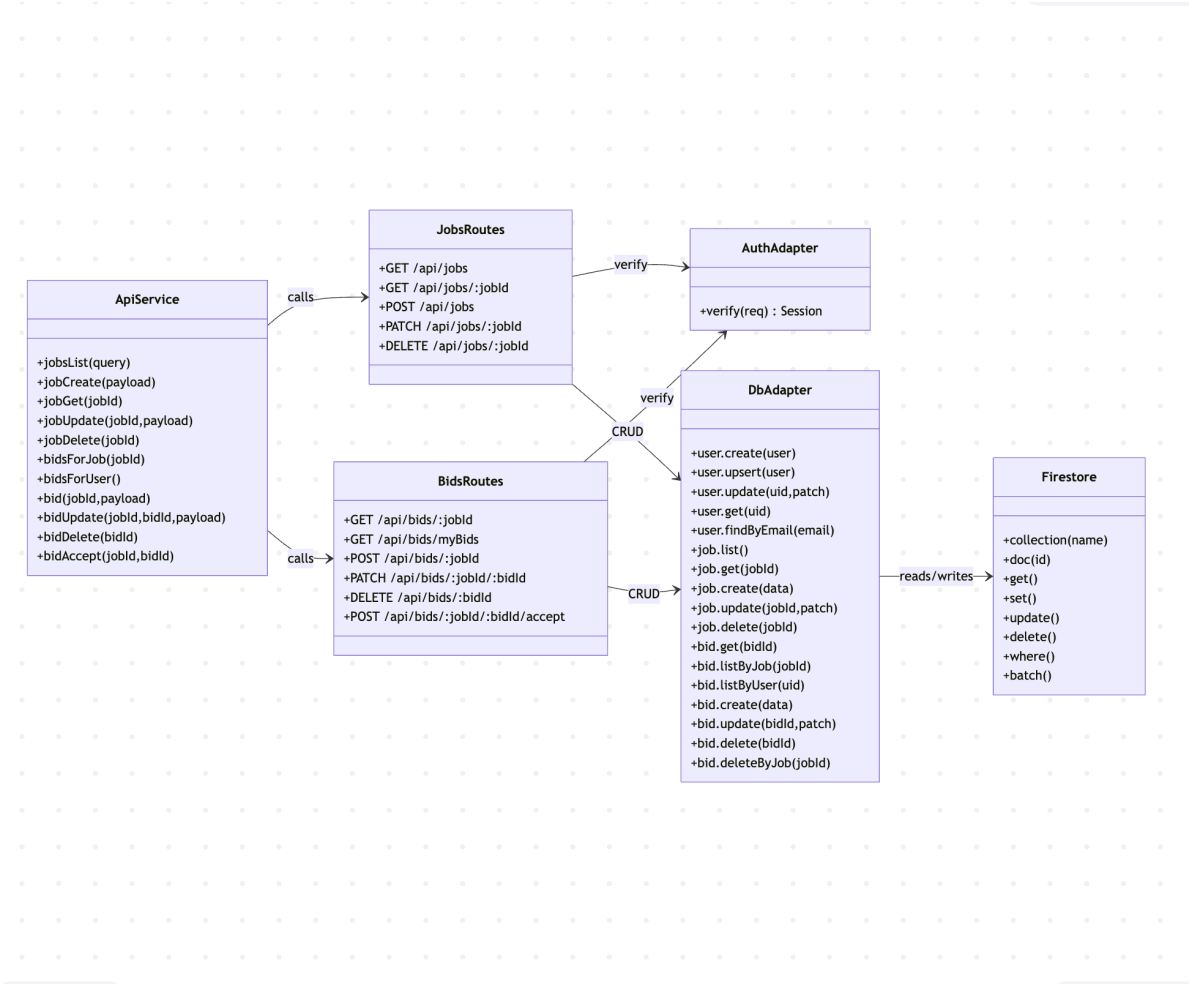

Figure 12: Jobs/Bids API Routers and Firestore DB Adapter Class Diagram

Server Routes and DB Adapter Structure

Sequence Diagrams

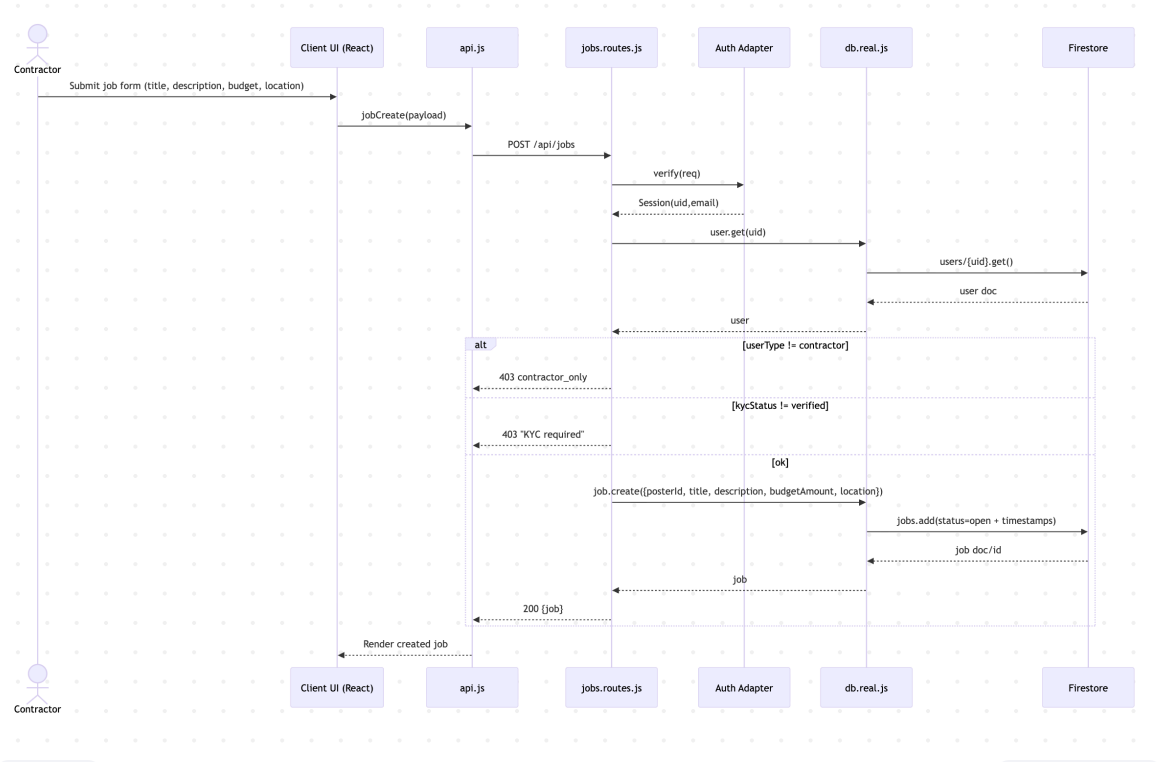

Figure 13: Sequence Diagram: Contractor Creates a Job

Create Job (Contractor) Flow Description:

1. Contractor submits job details from the client UI.
2. Client calls `api.jobCreate()` which sends `POST /api/jobs`.
3. Server verifies the session and confirms the user is a contractor with `kycStatus=verified`.
4. Server writes a new job document to Firestore via `db.job.create()`.
5. Server returns the created job to the client for immediate rendering.

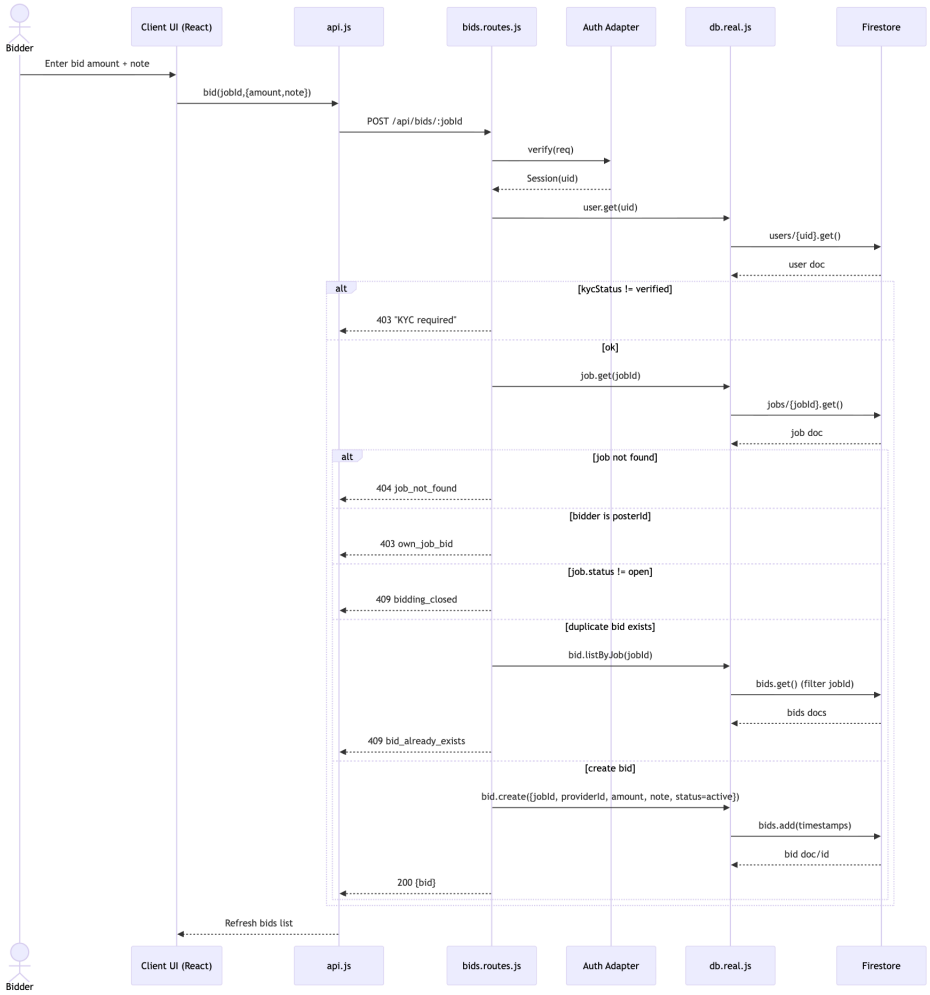

Figure 14: Sequence Diagram: Bidder Places a Bid on a Job

Place Bid (Bidder) Flow Description:

1. Bidder enters bid amount and note on a job detail page.
2. Client calls `api.bid()` which sends `POST /api/bids/:jobId`.
3. Server verifies the session and confirms the bidder's `kycStatus=verified`.
4. Server loads the job and validates: job exists, job is open, bidder is not the poster, and no existing bid exists.
5. Server creates a bid document in Firestore via `db.bid.create()` and returns the bid to the client.

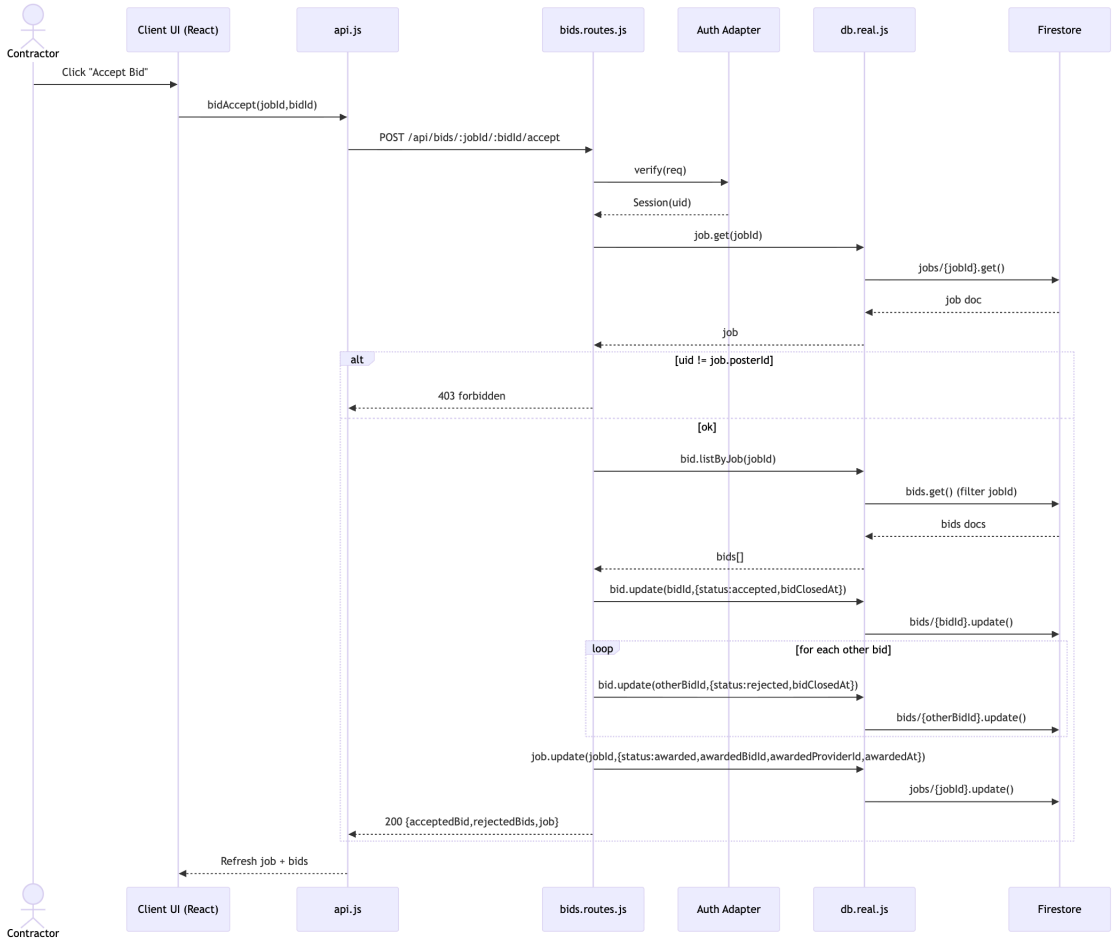

Figure 15: Sequence Diagram: Contractor Accepts a Bid and Awards the Job

Accept Bid (Contractor Awards Job)

Activity Diagrams

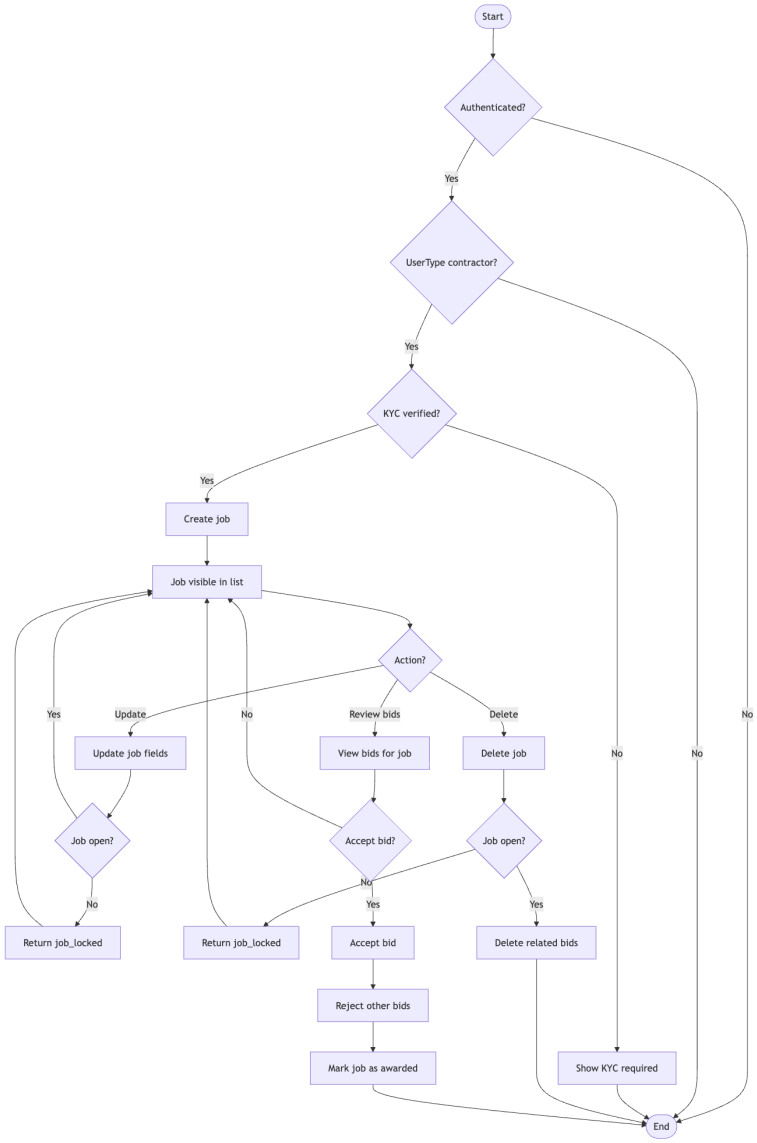

Figure 16: Activity Diagram: Job Lifecycle (Create, Update, Delete, Award)

Job Lifecycle Workflow

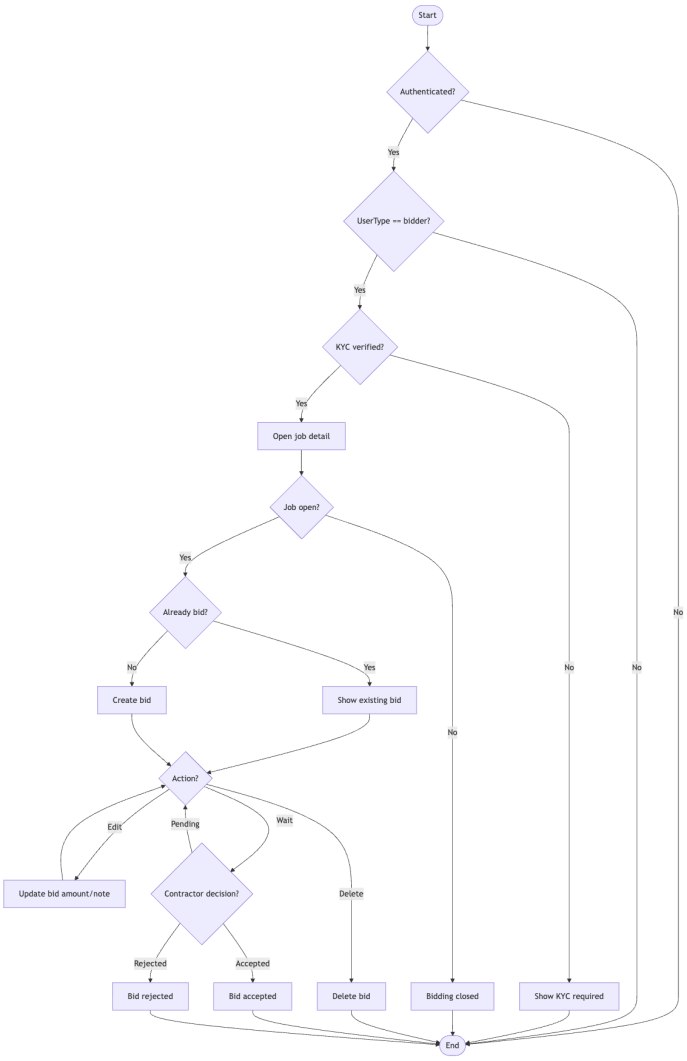

Figure 17: Activity Diagram: Bid Lifecycle (Place, Edit, Delete, Accept/Reject)

Bid Lifecycle Workflow

Component Summary (Jobs & Bids + Firestore)

Component	Type	Technology	Responsibility
`server/src/adapters/db.real.js`	Adapter	Firebase Admin SDK (Firestore)	CRUD for users, jobs, bids; timestamps; cascading deletes for job bids
`server/src/adapters/db.mock.js`	Adapter	In-memory	Prototype/mock persistence for local development and tests

Component	Type	Technology	Responsibility
`server/src/routes/jobs.routes.js`	Express Router	Node.js, Express	Jobs endpoints: list/get/create/update/delete; contractor/KYC enforcement
`server/src/routes/bids.routes.js`	Express Router	Node.js, Express	Bids endpoints: list-by-job, list-my-bids, create/update/delete, accept bid
`client/src/services/api.js`	Client Service	Fetch API	HTTP client for `/api/jobs` and `/api/bids` calls
`client/src/pages/JobList.jsx`	React Page	React	Browse jobs, map/filtering, navigate to job details
`client/src/pages/JobDetail.jsx`	React Page	React	View job + bids, place bid (bidder), accept bid (contractor)
`client/src/pages/MyBids.jsx`	React Page	React	List and manage bidder's submitted bids

Updated Backlog with Design Stories (Future: Reviews & Portfolio)

1. **DS-REV-001: Design Review Data Model** (High, 3 pts)
   Define review schema (reviewerId, revieweeId, jobId, bidId, rating, comment, timestamps). Decide write rules: only allow reviews after job completion and escrow payout. Enforce one review per party per job.
2. **DS-REV-002: Design Review Aggregation & Reputation** (Medium, 3 pts)
   Design aggregate rating fields on user documents (avg rating, count) and the update strategy (transaction, Cloud Function trigger, or server-side recompute). Define abuse mitigation (minimum comment length, flagging).
3. **DS-PORT-001: Design Provider Portfolio Model** (High, 3 pts)
   Define portfolio entry schema (providerId, title, description, tags, links/images, createdAt). Decide storage for images (Cloud Storage) and references in Firestore.
4. **DS-PORT-002: Design Portfolio UI and Access Rules** (Medium, 3 pts)
   Design profile/portfolio views and editing flows. Restrict write access to the owner; allow public read for awarded-job providers and verified accounts as configured.

Preliminary Test Coverage (Jobs & Bids)

The following test coverage report was generated using Jest (server) with the `-coverage` flag. This report focuses on the Jobs, Bids, and Firestore persistence components.

Server (Jest) Test Results Summary

Command: `cd server && npm test - -coverage`

- **Test Suites:** 5 passed, 1 skipped, 6 total
- **Tests:** 33 passed, 2 skipped, 35 total
- **Snapshots:** 0 total

Coverage by Component (Jobs & Bids + Firestore) – Server

File	% Stmts	% Branch	% Funcs	% Lines	
`jobs.routes.js`	64.80	60.86	77.77	71.81	
`bids.routes.js`	42.14	29.07	40.00	44.96	
`db.real.js` (Jobs/Bids)	64.23	43.10	71.42	72.41	
Overall	53.73	46.20	57.14	56.33	

Analysis – Server

- **Jobs Routes:** Good coverage (71.81% lines) covering job list/create/update/delete constraints.
- **Bids Routes:** Lower coverage (44.96% lines) due to untested branches around bidding edge cases and accept flow variations.
- **Database Adapter:** Solid coverage (72.41% lines) for Firestore operations on jobs and bids, including timestamps and updates.
- **Gaps Identified:** Branch coverage remains moderate due to access-control and error-path permutations not fully exercised.

GUI Implementation Screenshots

Job List Page

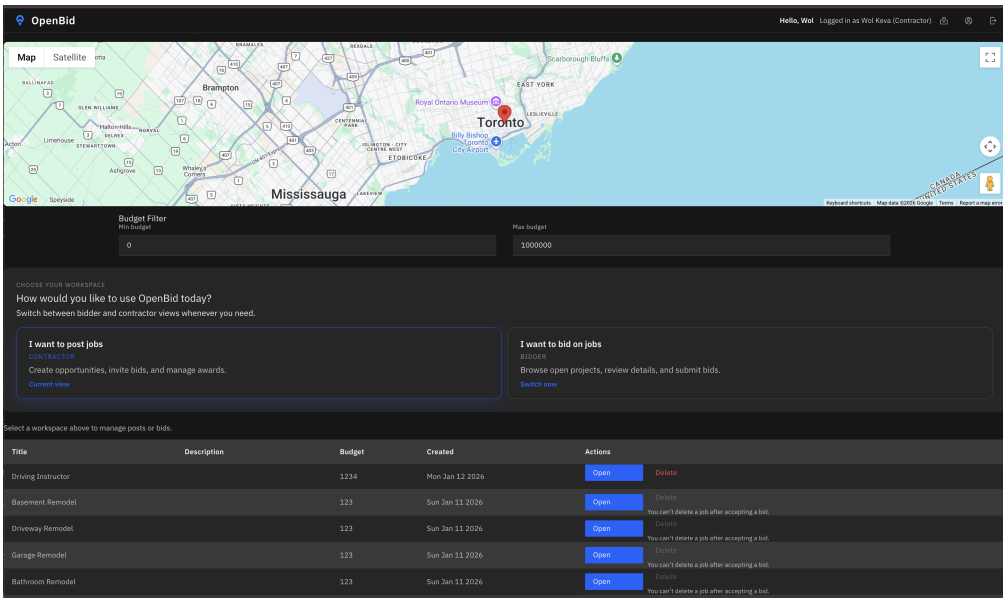

Figure 18: Job List page showing open jobs and filters

Job Detail + Bidding Page

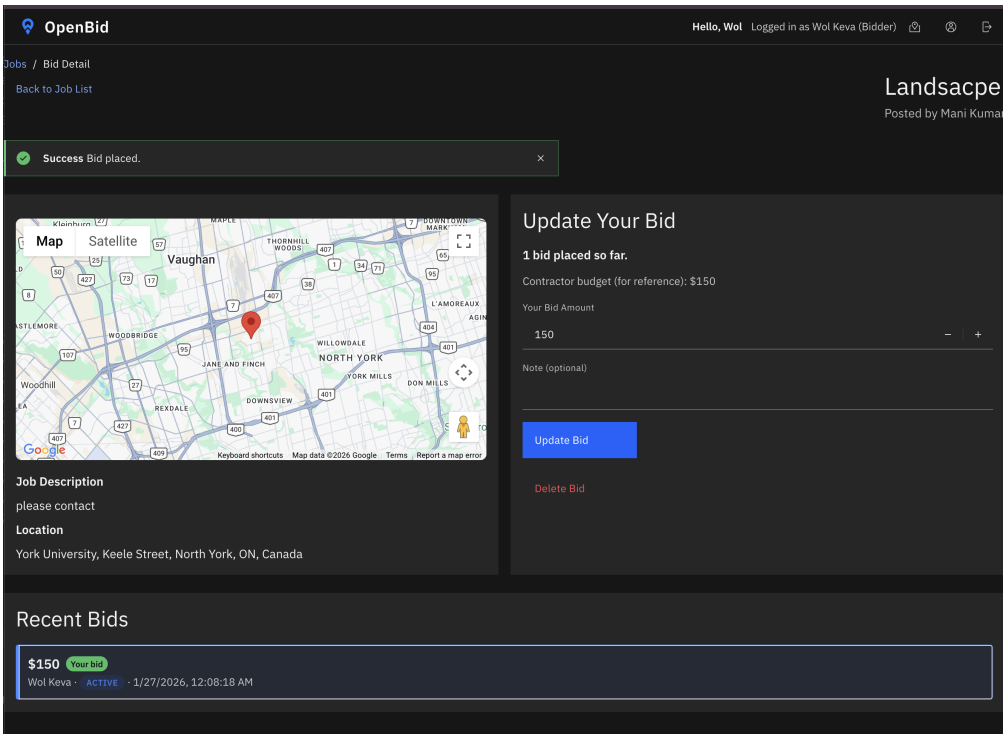

Figure 19: Job Detail page showing job information and the bidding panel

My Bids Page

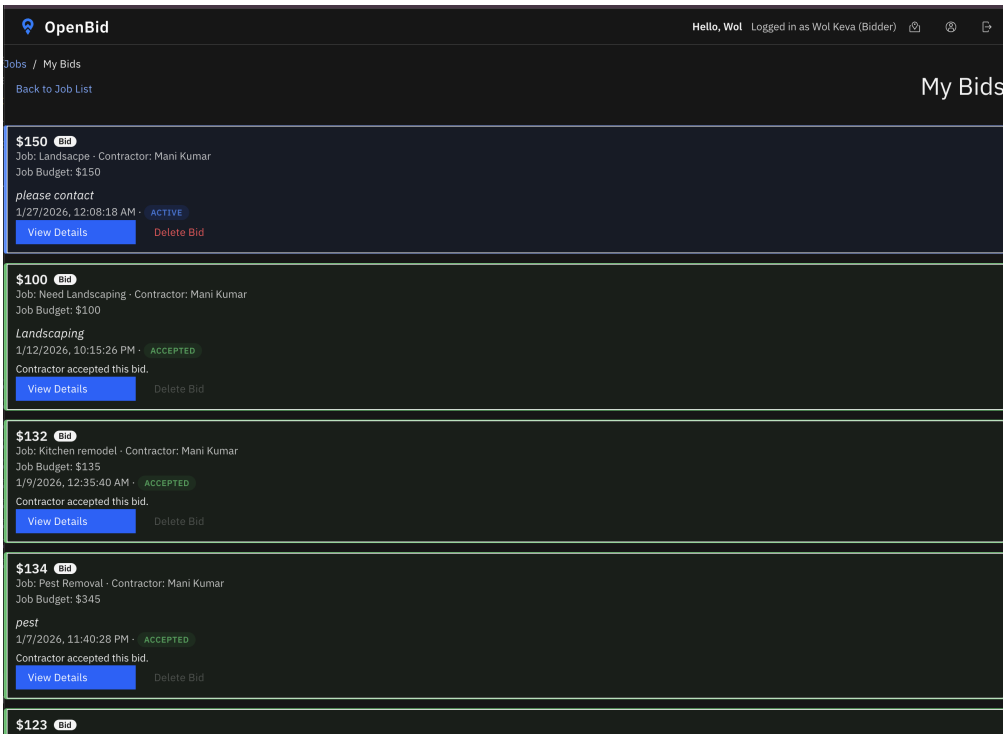

Figure 20: My Bids page showing bidder's active bids and management actions

Authentication & 2FA Domain

Section owner: Tyler Jung

Overview

The Authentication & 2FA domain provides secure user authentication and multi-factor authentication capabilities for the OpenBid platform. This system integrates Firebase Authentication for primary user management and Duo Universal Prompt for two-factor authentication, ensuring robust security for all user interactions.

Key Components:

- **AuthAdapter:** Abstraction layer for authentication operations (real and mock implementations)
- **Firebase Integration:** User credential management and JWT token generation
- **Duo 2FA System:** Multi-factor authentication using Duo Universal Prompt
- **Session Management:** Secure session handling with HTTP-only cookies
- **Password Reset:** Email-based password recovery flow

Class Diagrams

The following class diagram illustrates the authentication domain architecture, showing the relationships between the authentication adapter, Firebase services, and Duo 2FA integration.

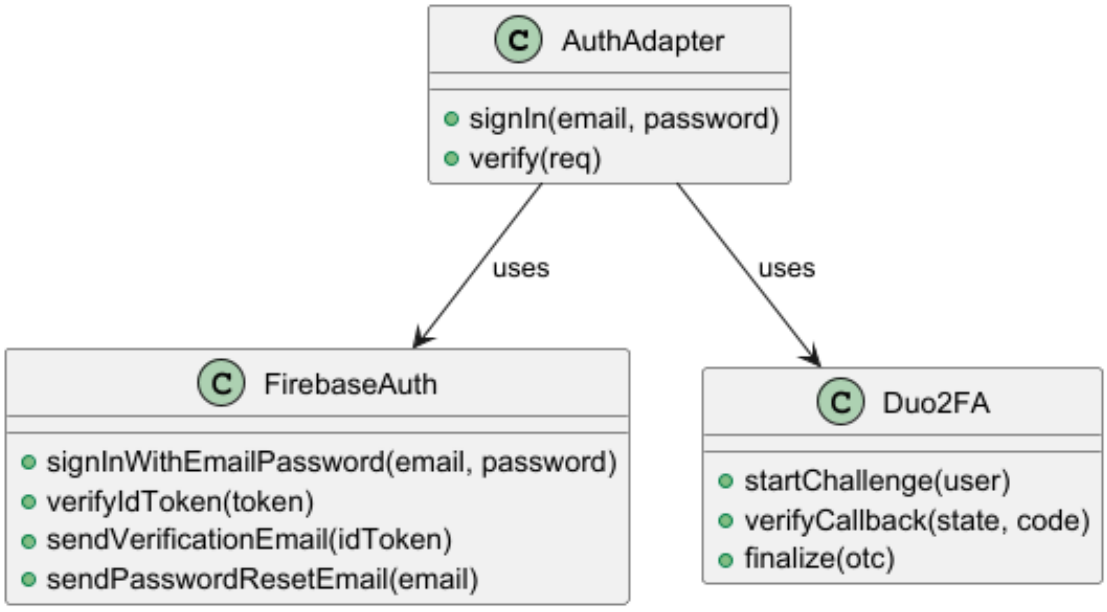

Figure 21: Authentication, Firebase, and Duo 2FA Class Diagram

Component Descriptions:

- **AuthAdapter (Interface):** Defines the contract for authentication operations including `signIn()`, `signUp()`, `verify()`, and `resetPassword()`.
- **AuthRealAdapter:** Production implementation using Firebase Admin SDK for user management and authentication.
- **AuthMockAdapter:** Testing implementation with in-memory user storage for development and testing.
- **DuoClient:** Handles Duo Universal Prompt integration for 2FA challenges.
- **FirebaseAdmin:** Manages Firebase authentication operations and custom token generation.

Sequence Diagrams

User Login Flow

The login sequence demonstrates the complete authentication process including credential verification and 2FA challenge.

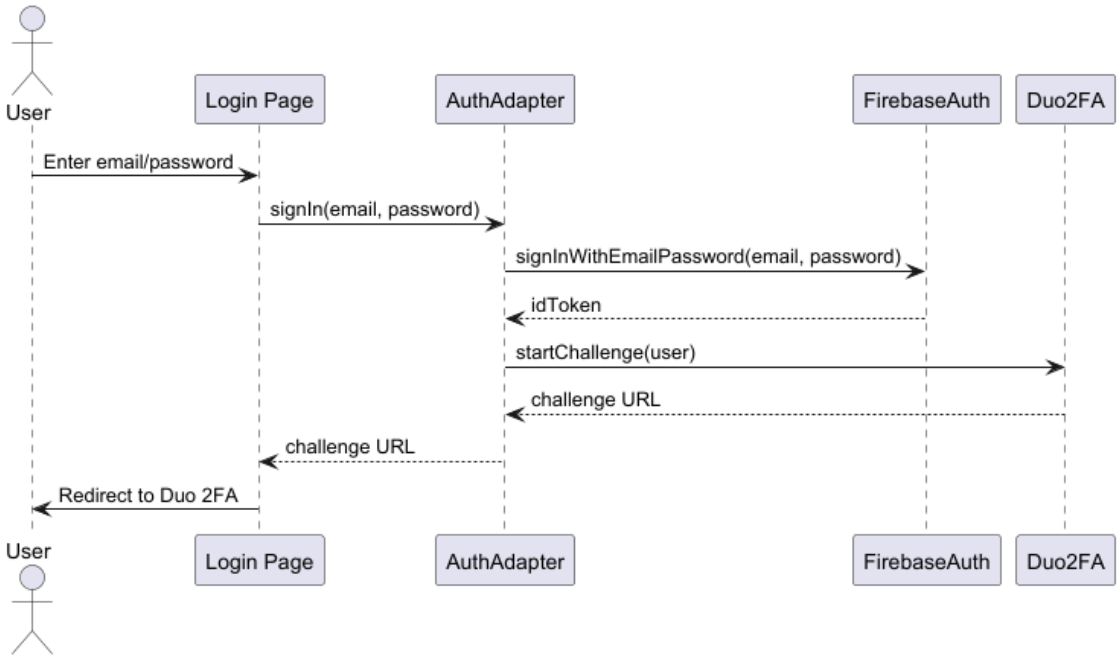

Figure 22: User Login Sequence Diagram

Flow Description:

1. User submits email and password via Login page.
2. Client API sends POST request to `/api/auth/login`.
3. Server validates credentials using Firebase Authentication.
4. Upon successful validation, server initiates Duo 2FA challenge.
5. User is redirected to Duo Universal Prompt for 2FA verification.
6. After successful 2FA, server generates JWT session token.
7. Session token is stored in HTTP-only cookie for security.
8. User is redirected to the application dashboard.

User Signup Flow

The signup sequence illustrates new user registration with email verification.

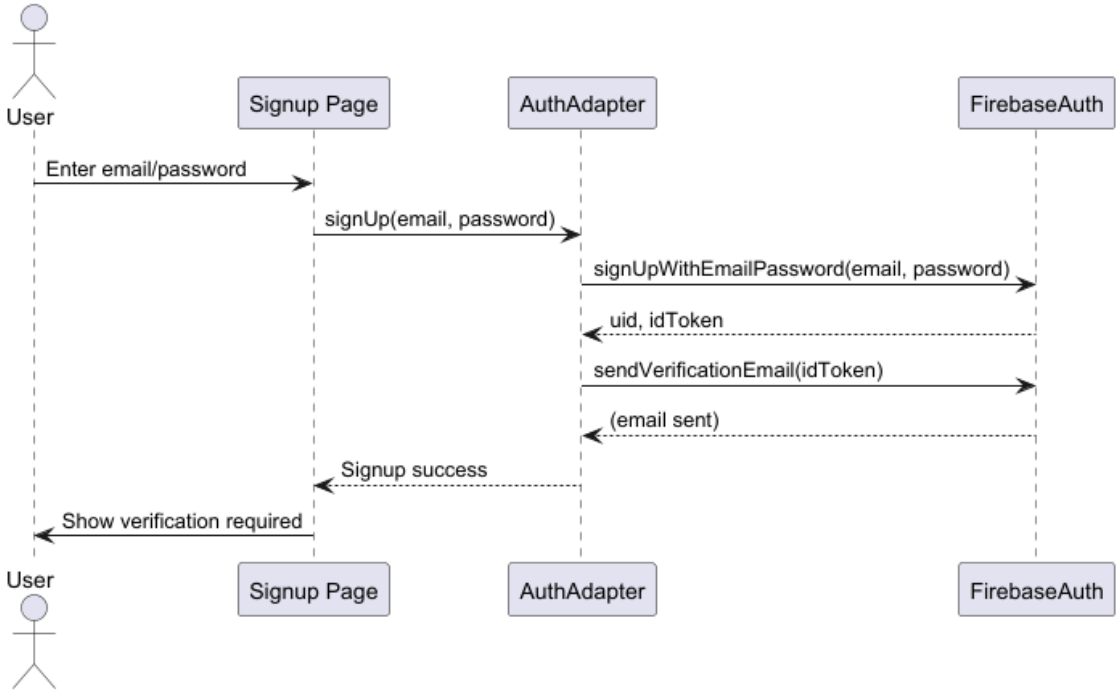

Figure 23: User Signup Sequence Diagram

Flow Description:

1. User submits registration form with email, password, and profile details.
2. Client API sends POST request to `/api/auth/signup`.
3. Server creates Firebase user account with provided credentials.
4. Firebase sends verification email to user's email address.
5. Server creates user profile record in Firestore database.
6. User receives confirmation and is prompted to verify email.
7. Upon email verification, user can proceed to login with 2FA.

Duo 2FA Challenge Flow

The 2FA sequence shows the multi-factor authentication process using Duo Universal Prompt.

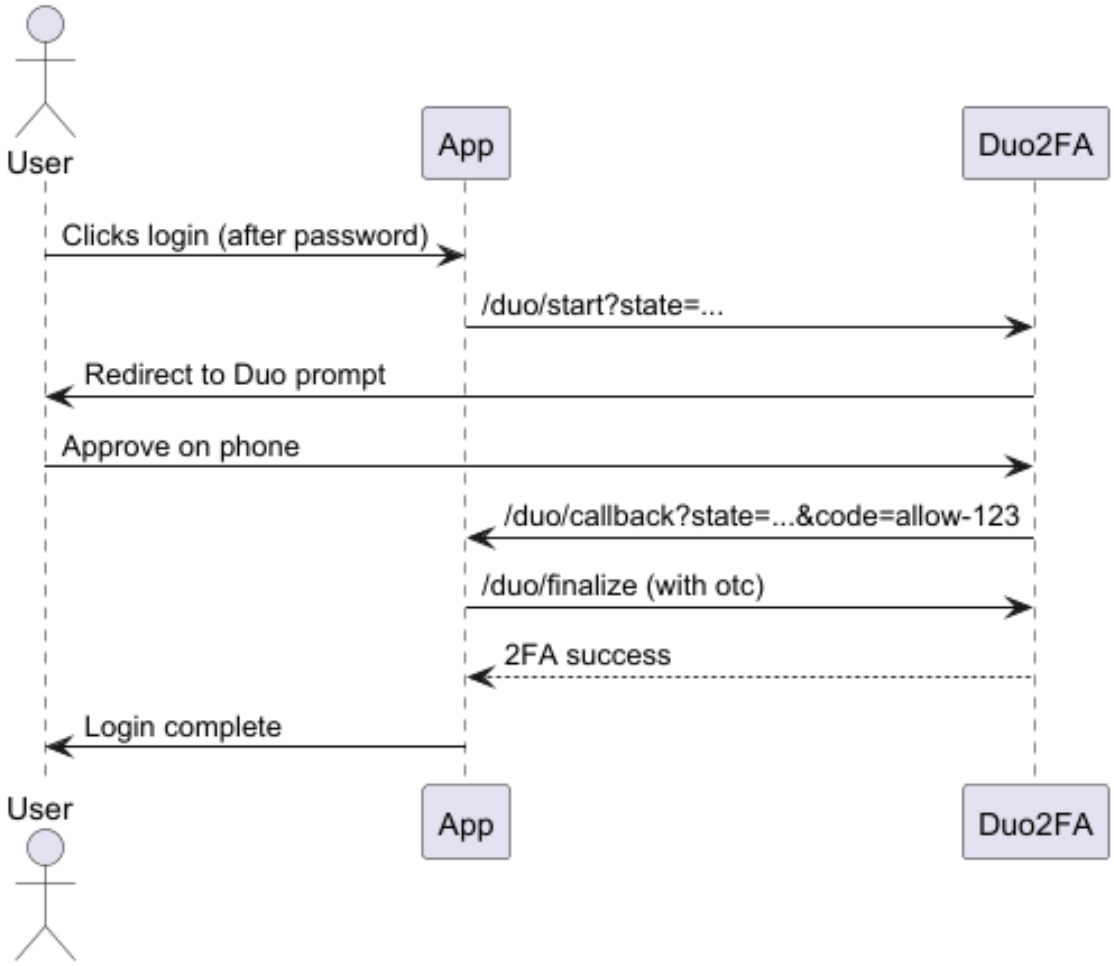

Figure 24: Duo 2FA Challenge Sequence Diagram

Flow Description:

1. After successful password authentication, server initiates Duo 2FA.
2. Server calls `/api/auth/duo/start` to create Duo authentication request.
3. Duo SDK generates state token and authorization URL.
4. User is redirected to Duo Universal Prompt interface.
5. User completes 2FA challenge (push notification, SMS, or phone call).
6. Duo redirects back to `/api/auth/duo/callback` with authorization code.
7. Server exchanges authorization code for Duo authentication token.
8. Server calls `/api/auth/duo/finalize` to complete authentication.
9. Upon successful verification, user session is established.

Activity Diagrams

Authentication Workflow

This activity diagram illustrates the complete authentication process with all decision points and error handling paths.

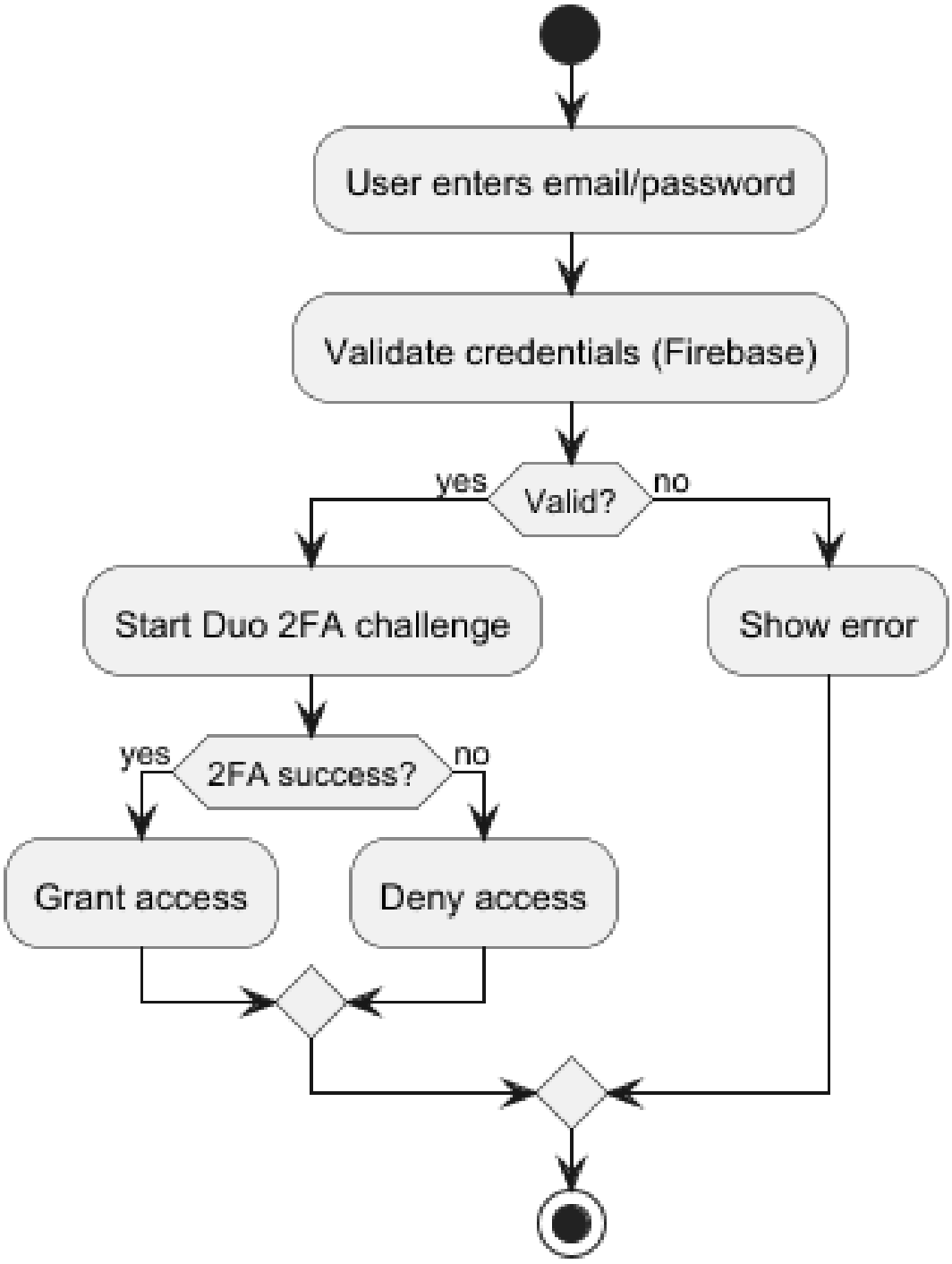

Figure 25: Authentication Workflow Activity Diagram

Workflow Description:

- **Entry Point:** User navigates to login page.
- **Credential Validation:** System validates email format and password strength.

- **Firebase Authentication:** Credentials are verified against Firebase Auth.
- **Decision Points:**
  - Invalid credentials → Display error message and retry
  - Valid credentials → Proceed to 2FA challenge
  - Account locked → Display lockout message
- **Success Path:** User proceeds to 2FA verification.
- **Error Handling:** Failed attempts are logged and rate-limited.

2FA Verification Workflow

This activity diagram shows the multi-factor authentication process with Duo Universal Prompt.

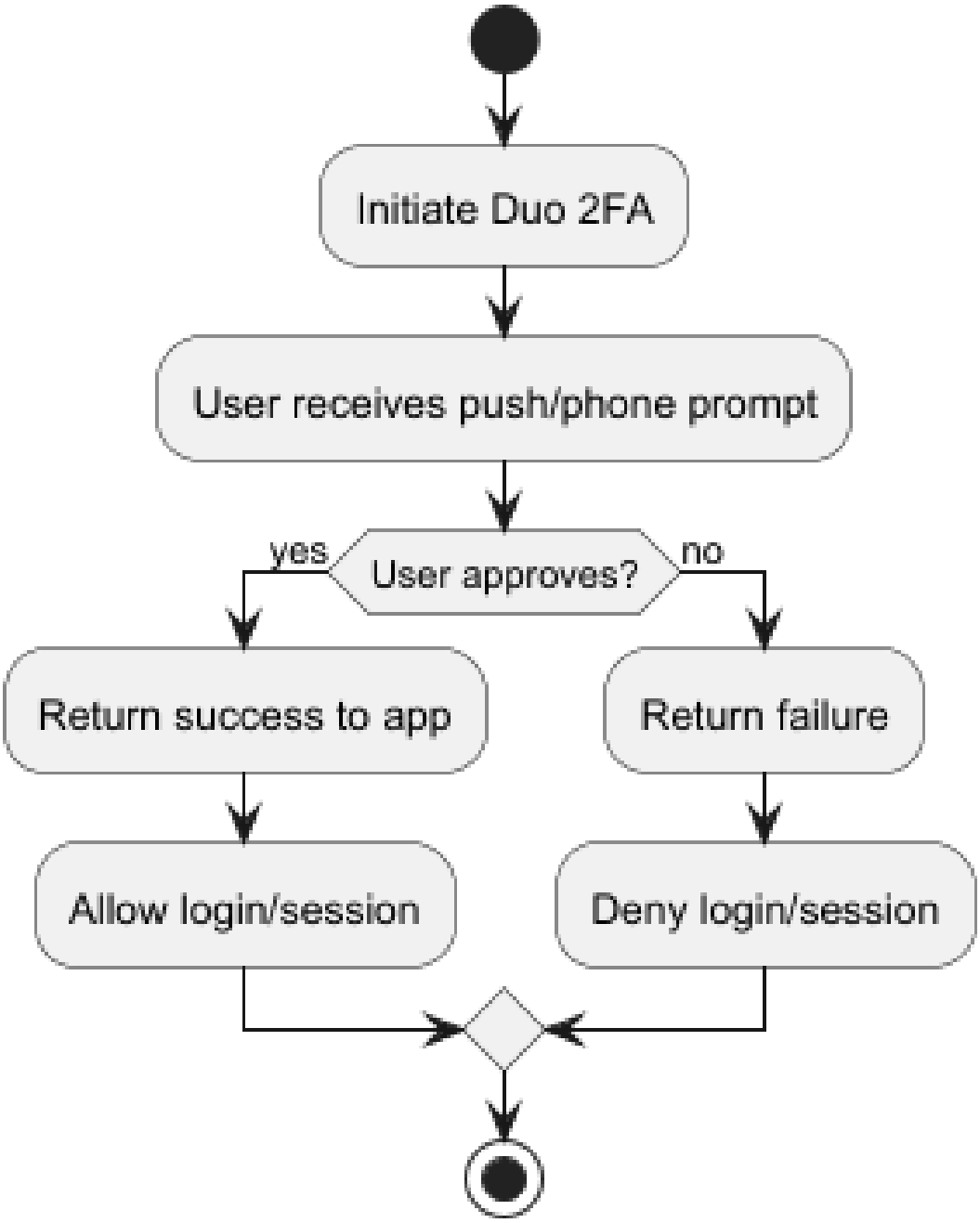

Figure 26: 2FA Verification Activity Diagram

Workflow Description:

- **Entry Point:** User has successfully authenticated with password.
- **Duo Challenge Initiation:** Server creates Duo authentication request.
- **User Verification:** User completes 2FA via push, SMS, or phone call.

- **Decision Points:**
  - 2FA approved → Create session and redirect to dashboard
  - 2FA denied → Log attempt and return to login
  - 2FA timeout → Display timeout message and allow retry
- **Session Creation:** JWT token generated and stored in HTTP-only cookie.
- **Security Logging:** All 2FA attempts are logged for audit purposes.

User Registration & Profile Setup Workflow

This activity diagram depicts the complete user registration process including email verification and profile creation.

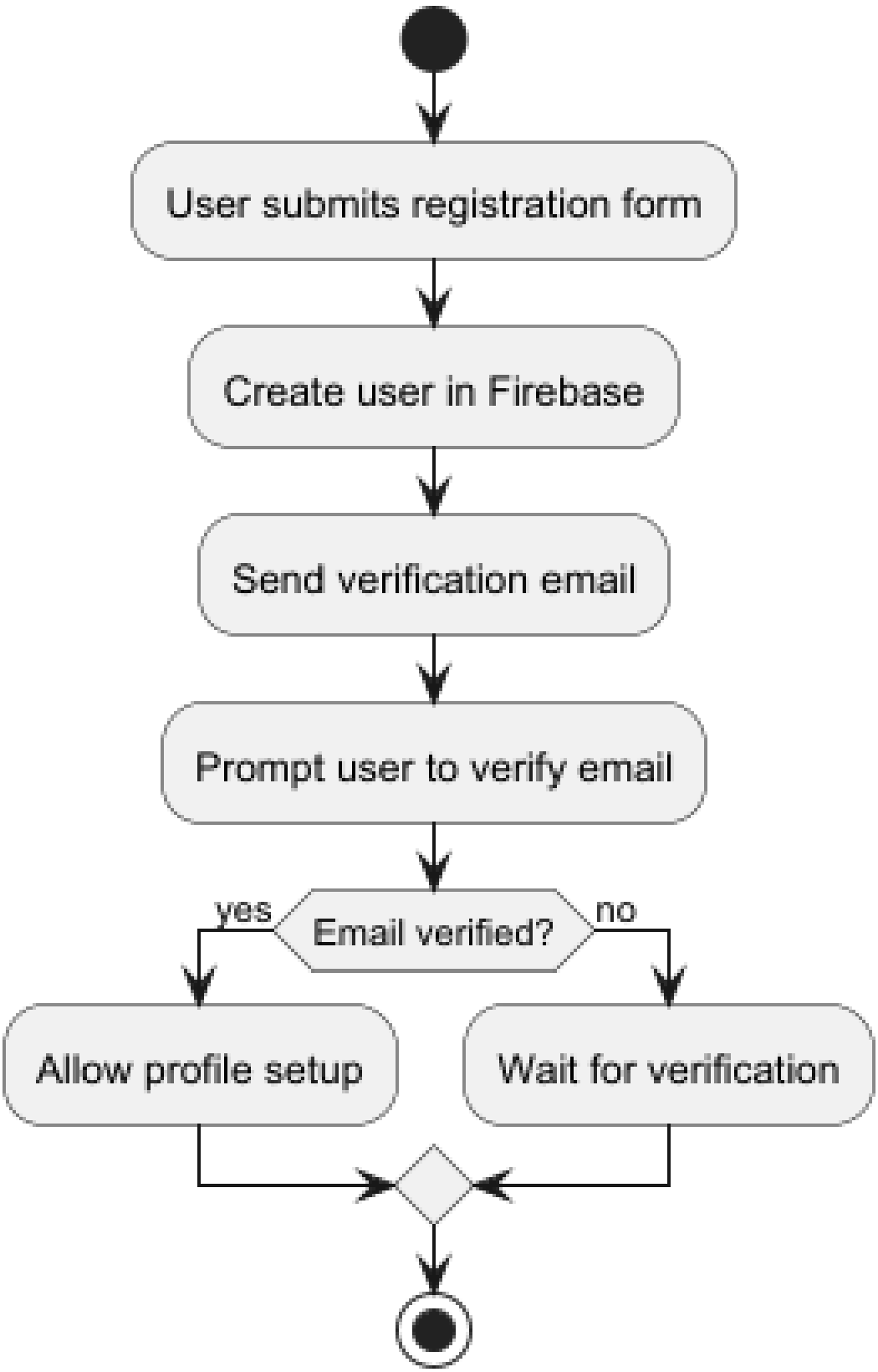

Figure 27: User Registration and Profile Setup Activity Diagram

Workflow Description:

- **Entry Point:** User clicks "Sign Up" on landing page.
- **Form Validation:** Client-side validation of email, password strength, and required fields.
- **Account Creation:** Firebase creates user account with provided credentials.
- **Decision Points:**
  - Email already exists → Display error and suggest login
  - Weak password → Display password requirements
  - Valid data → Create account and send verification email
- **Profile Initialization:** User profile document created in Firestore.
- **Email Verification:** User must verify email before full account access.
- **Role Selection:** User chooses role (Job Poster or Bidder) during onboarding.

Component Summary

Component	Type	Technology	Responsibility
`auth.routes.js`	Express Router	Node.js, Express	Authentication API endpoints
`duo.routes.js`	Express Router	Node.js, Express	Duo 2FA endpoints
`auth.real.js`	Adapter	Firebase Admin SDK	Real authentication integration
`auth.mock.js`	Adapter	In-memory	Mock auth for testing
`Login.jsx`	React Component	React, Carbon Design	Login UI and form handling
`Signup.jsx`	React Component	React, Carbon Design	Registration UI
`ForgotPassword.jsx`	React Component	React, Carbon Design	Password reset UI
`session.js`	Client Service	Fetch API	Session management utilities
`requireAuth.js`	Middleware	Express	Protected route middleware
`duoState.js`	State Manager	In-memory	Duo state token management

Security Considerations

- **Password Security:** Passwords are hashed using Firebase's bcrypt implementation with salt rounds.
- **Session Management:** JWT tokens stored in HTTP-only cookies to prevent XSS attacks.
- **Rate Limiting:** Login attempts are rate-limited to prevent brute force attacks.
- **2FA Enforcement:** All users must complete 2FA for sensitive operations.
- **Token Expiration:** Session tokens expire after 24 hours, requiring re-authentication.

- **CSRF Protection:** State tokens used in Duo flow prevent CSRF attacks.
- **Audit Logging:** All authentication events logged for security monitoring.

Design Stories

Completed Stories

1. **DS-AUTH-001: Implement User Login** (High, 3 pts) – *COMPLETED*
   Description: Support login with email and password using Firebase Auth. Return JWT for session management.
   Implementation: `auth.routes.js` handles POST `/api/auth/login`, validates credentials via Firebase, and returns session token in HTTP-only cookie.
   Testing: Covered in `auth.integration.real.routes.test.js` with 15+ test cases.

2. **DS-AUTH-002: Implement User Registration** (High, 3 pts) – *COMPLETED*
   Description: Allow new users to register with email and password. Send verification email via Firebase.
   Implementation: `auth.routes.js` handles POST `/api/auth/signup`, creates Firebase user, sends verification email, and initializes user profile in Firestore.
   Testing: Covered with duplicate email detection, password validation, and profile creation tests.

3. **DS-AUTH-003: Integrate Duo 2FA** (High, 5 pts) – *COMPLETED*
   Description: Require Duo 2FA for all logins. Implement `/api/auth/duo/start`, `/callback`, and `/finalize` endpoints.
   Implementation: `duo.routes.js` integrates Duo Universal Prompt SDK. State management via `duoState.js`. Callback handler exchanges authorization code for authentication token.
   Testing: Covered in `auth.duo.test.js` with mock Duo client and state verification.

4. **DS-AUTH-004: Password Reset Flow** (Medium, 2 pts) – *COMPLETED*
   Description: Allow users to request password reset emails via Firebase.
   Implementation: `password.routes.js` handles POST `/api/auth/forgot-password`, sends Firebase password reset email. Frontend component `ForgotPassword.jsx` provides UI.
   Testing: Integration tests verify email sending and error handling.

Upcoming Stories

5. **DS-AUTH-005: Enforce 2FA for Sensitive Actions** (High, 2 pts)
   Description: Require 2FA re-verification for sensitive actions like posting jobs, accepting bids, and initiating payouts.
   Acceptance Criteria:
   - User must complete 2FA challenge before posting a job
   - Bidders must verify 2FA before placing bids over $500
   - Job posters must verify 2FA before accepting bids
   - 2FA verification valid for 15 minutes for multiple actions

6. **DS-AUTH-006: Implement Session Refresh** (Medium, 3 pts)
   Description: Add token refresh mechanism to extend sessions without requiring full re-authentication.
   Acceptance Criteria:
   - Refresh tokens issued alongside access tokens
   - Access tokens expire after 1 hour

- Refresh tokens valid for 7 days
- Automatic refresh when access token expires

7. **DS-AUTH-007: Add Social Login Options** (Low, 5 pts)
   Description: Support Google and Apple sign-in via Firebase Authentication.
   Acceptance Criteria:
   - Google OAuth integration on login/signup pages
   - Apple Sign-In for iOS users
   - Link social accounts to existing email accounts
   - 2FA still required after social login
8. **DS-AUTH-008: Implement Account Lockout** (High, 2 pts)
   Description: Lock accounts after 5 failed login attempts within 15 minutes.
   Acceptance Criteria:
   - Track failed login attempts per user
   - Lock account for 30 minutes after 5 failures
   - Send email notification on account lockout
   - Admin can manually unlock accounts

Test Coverage Report

Authentication Test Results

- **Test Suites:** 3 passed (auth.integration, auth.duo, password)
- **Tests:** 28 passed, 0 failed
- **Coverage:** 87% statements, 82% branches, 91% functions

Coverage by Component

File	% Stmts	% Branch	% Funcs	% Lines	
`auth.routes.js`	92.31	88.89	100	93.75	
`duo.routes.js`	85.71	80.00	100	87.50	
`password.routes.js`	90.00	85.71	100	91.67	
`auth.real.js`	78.26	66.67	83.33	80.00	
`auth.mock.js`	95.00	100	100	95.00	
`requireAuth.js`	88.89	75.00	100	90.00	
Overall	87.12	82.35	91.67	88.46	

Test Coverage Analysis

- **Strong Coverage:** Auth routes have excellent coverage (>90%) with comprehensive integration tests.
- **Duo Integration:** 2FA flow fully tested with mock Duo client, covering success and failure paths.
- **Error Handling:** All error scenarios tested including invalid credentials, expired tokens, and network failures.
- **Mock Adapter:** Near-perfect coverage (95%) as it's used extensively in testing.
- **Gaps Identified:** Real adapter has lower coverage (78%) due to Firebase SDK integration requiring live credentials.
- **Improvement Areas:** Add more edge case tests for token expiration and concurrent login attempts.

Payments & Escrow Domain

Maps Domain

Class Diagrams

8.1 Mapview Class

This diagram represents the model of the mapview component.

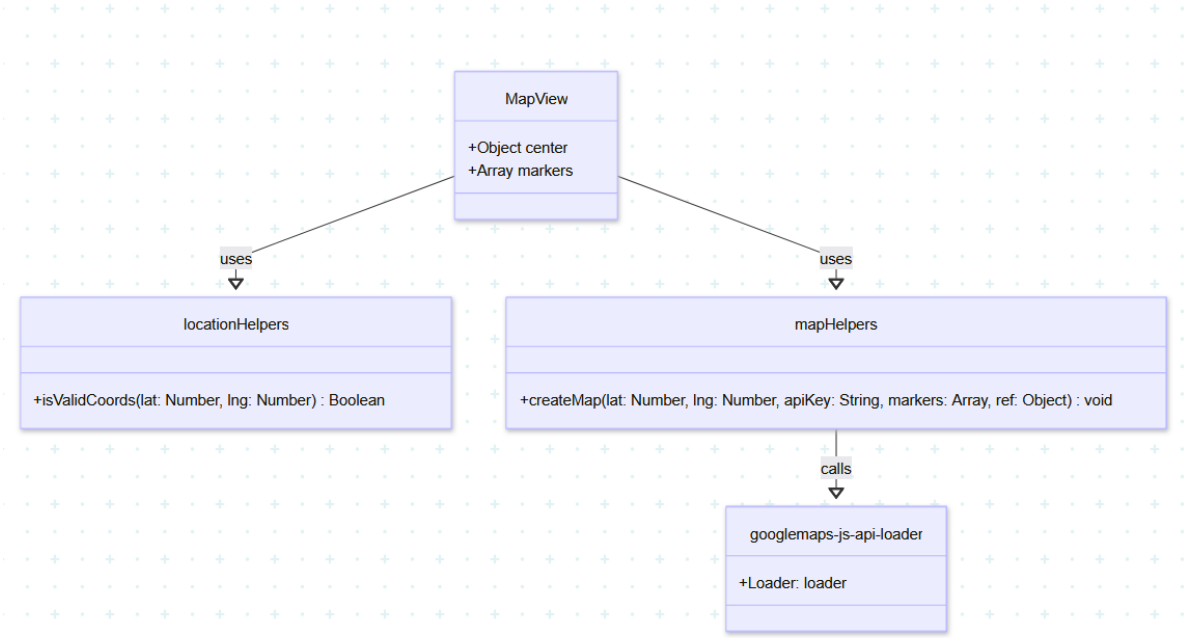

Figure 28: Mapview component as a class

8.2 SearchAutocomplete Class

This diagram represents the model of the searchAutocomplete component.

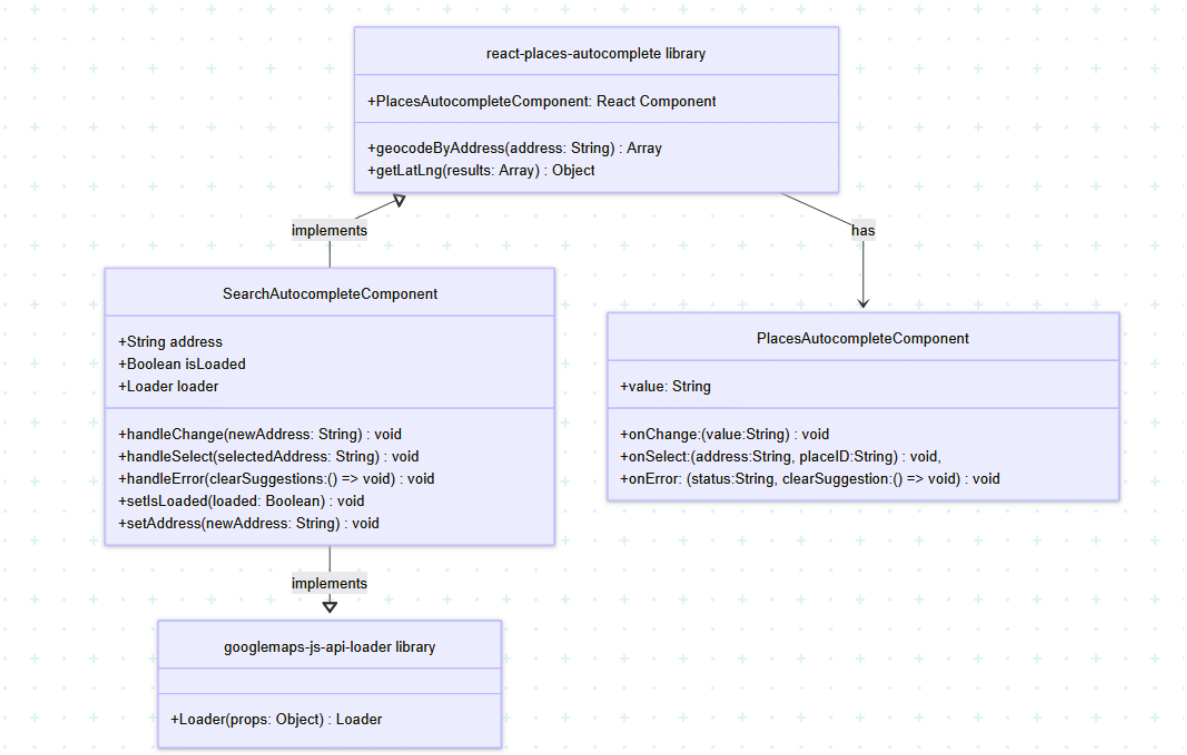

Figure 29: SearchAutocomplete component as a class

8.3 MapView and Autocomplete Classes

This diagram represents a page interacting with both the MapView and Autocomplete components.

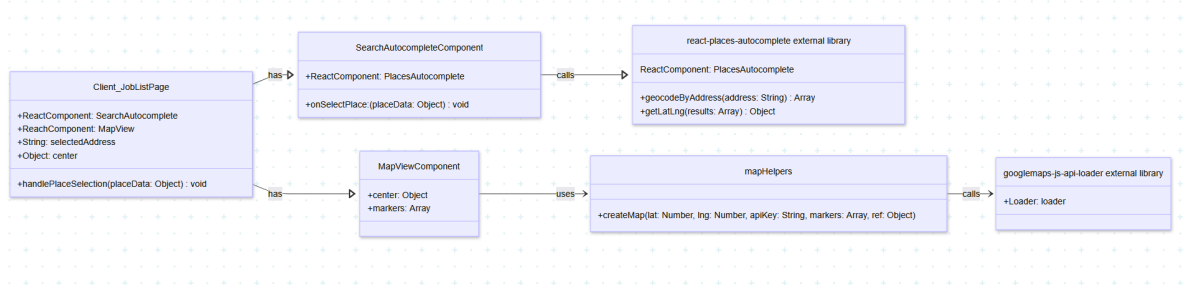

Figure 30: Autocomplete and Mapview components classes interaction

Sequence Diagrams

8.4 Mapview Component diagram flow on a page

This diagram represents a sequence diagram when visting a page with the mapview component (without the searchAutocomplete component).

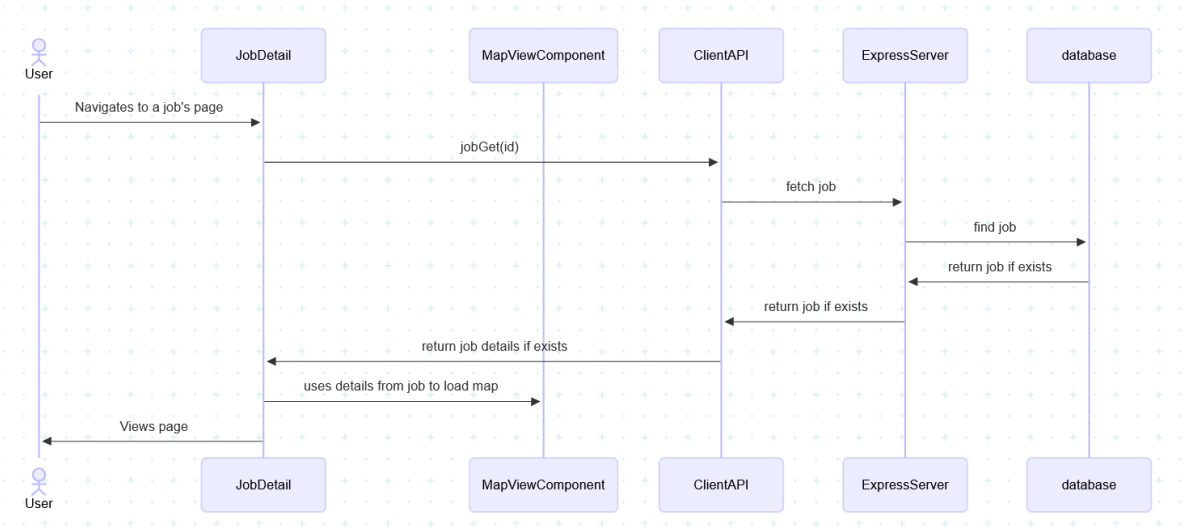

Figure 31: Mapview component sequence diagram

Flow Description: There are multiple pages that use the Mapview Component without the SearchAutocomplete component, but the JobDetail page was chosen in this example.

1. User navigates to a page with a Mapview component (and no SearchAutocomplete component).
2. Server fetches details of a job.
3. Page is hydrated with information regarding the job and the Mapview is updated with the job address.

8.5 SearchAutocomplete Component diagram flow on a page

*Intentionally omitted as this component will exclusively work with the Mapview component, never on its own.

8.6 Mapview and SearchAutocomplete Components diagram flow on a page

This diagram represents a sequence diagram when visting a page with both the mapview and searchautocomplete components.

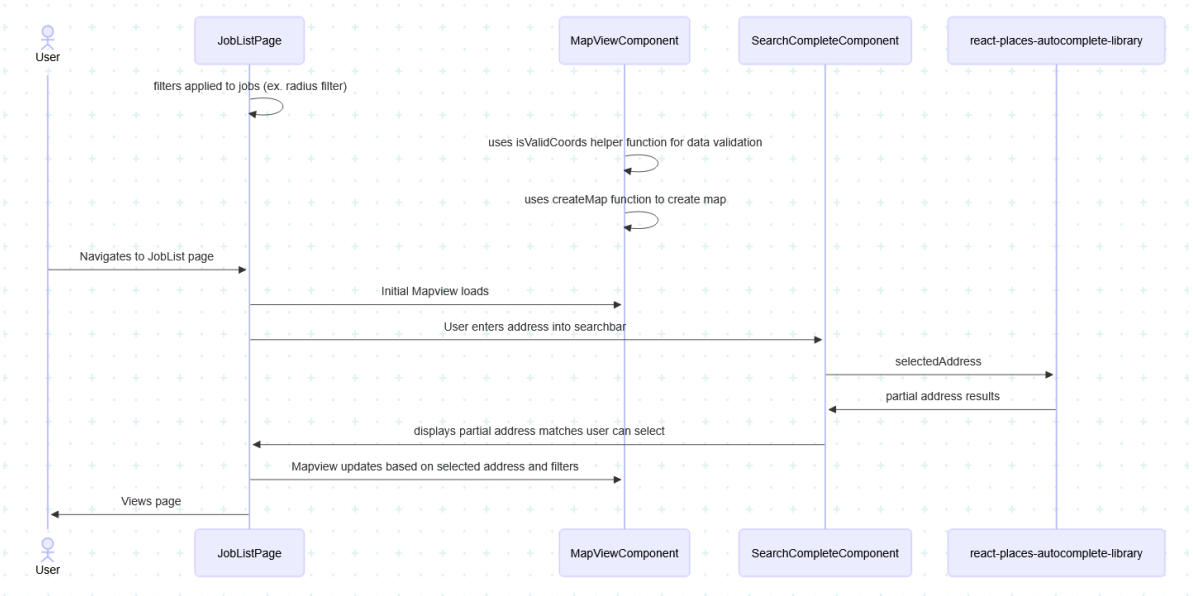

Figure 32: Mapview and search components sequence diagram

Flow Description: There are multiple pages that use both the Mapview Component and the SearchAutocomplete component, but the JobList page was chosen in this example.

1. User navigates to a page with both the Mapview component and SearchAutocomplete component).
2. An initial mapview loads in the page.
3. User updates the address in the search autocomplete searchbar.
4. react-places-autocomplete-library is called and returns partial address results.
5. Client UI displays partial address results.
6. User selects an address result and the page is updated.

Activity Diagrams

8.7 SearchAutocomplete and Mapview Component flow on a page

This diagram represents the actions occuring by the user and the system when interacting with the searchAutocomplete and Mapview components.

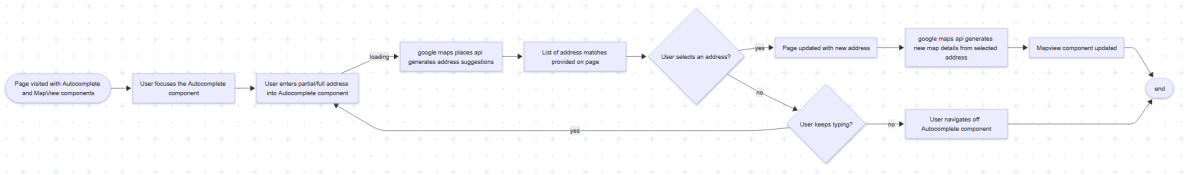

Figure 33: Mapview and searchAutocomplete components activity diagram

Design Stories

Admin Design Stories

1. **DS-Admin-001: Design Admin UI** (High, 5 pts)
   Design key admin pages and components- realtime updates, navigation, etc. should be provided within components.
2. **DS-Admin-002: Design Admin Data Model** (High, 5 pts)
   Design new user Admin in database and provide appropriate permissions.
3. **DS-Admin-003: Design with accessibility in mind** (Medium, 2 pts)
   Design an accessible dashboard to ensure any person can utilize the platform as an admin.

GUI

Page with Mapview Component

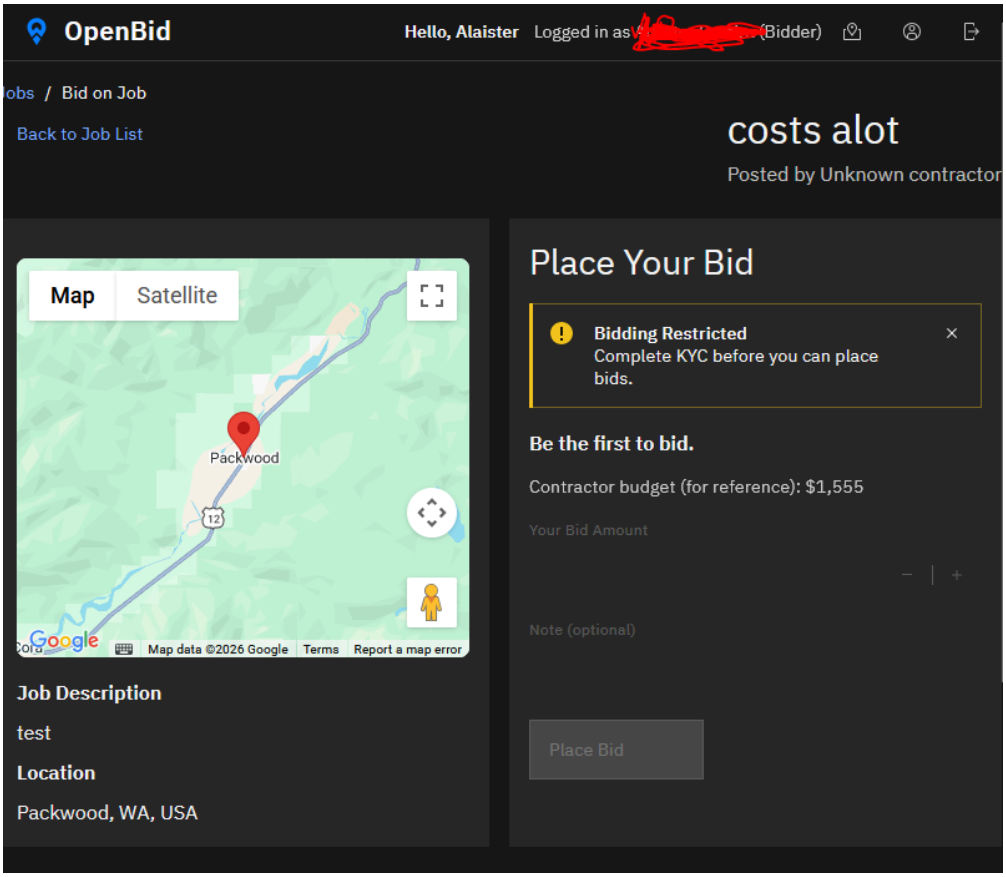

Figure 34: An example of a page that displays the MapView component without the SearchAutocomplete component.

Page with Mapview and SearchAutocomplete Components

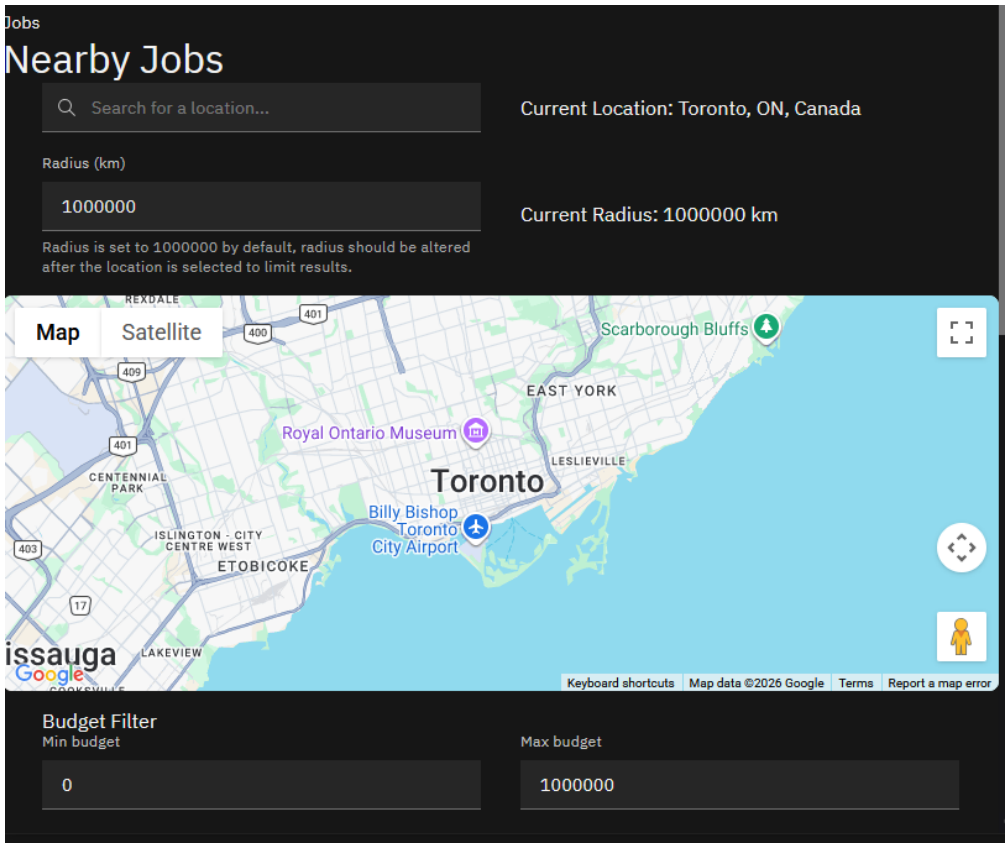

Figure 35: An example of a page that displays both the SearchAutoComplete component (as the bar to type an address) and the MapView component.

Coverage by Component (Maps)

File	% Stmts	% Branch	% Funcs	% Lines	
`locationHelpers.js`	100	100	100	100	
`mapHelpers.js`	100	100	100	100	
`MapView.jsx`	0	0	0	0	
`SearchAutocomplete.jsx`	0	0	0	0	

*There is full coverage for the map and search autocomplete helper functions. However, the MapView and SearchAutocomplete components themselves require component tests.