Multidisciplinary project Rapport

A report submitted as part of the requirements
for the
Pluridisciplinary Project
of Final year of Preparatory cycle
at The Higher School of Computer Science Sidi
Bel Abess,8 Mai 1945

Supervisor Dr. Serhane Oussama

Grow Path

Connecting Students and Companies for Real-World Experience

May 23, 2025

Contents

Team Members:

Member Name	Role	Contact
Hasrane Mouloud	Mobile/ web	m.hasrane@esi-sba.dz
Goumri Zakaria	web	z.goumri@esi-sba.dz
Bouroumana moundher	Backend	m.bouroumana@esi-sba.dz
Sadouki Wassim	Backend	w.sadouki@esi-sba.dz
Amour Mohamed Islam	Mobile	mi.amour@esi-sba.dz
BENAMMOUR RIHAB MERIEM	ui/ux	Rm.benammour@esi-sba.dz

1 Introduction

In Algeria, especially at the Higher School of Computer Science (ESI), 4th-year students often struggle to secure internships that provide real-life technical exposure. Our platform bridges this gap by connecting students with companies seeking interns or solutions to technical problems. This project includes a web frontend, backend, and a mobile app to support all users involved in this ecosystem.

2 Project Objectives

Help students gain real-world experience through internships and projects.

- Provide companies with easy access to student talent.
- Simplify team formation, application submission, and communication.
- Provide dashboards and tracking tools for companies and students.

3 Specifications and needs analysis

1. Role-based account creation for students, companies, and admins.
2. Posting and browsing of internships and technical challenges.
3. Team formation and collaborative application submission.
4. Application state tracking and notifications.
5. Integrated chat between companies and students.
6. Admin dashboard for user and post moderation.

4 User Roles and Permissions

- **Students:** Browse and apply to internships, form teams.
- **Companies:** Post opportunities, review and accept applicants.
- **Admin:** Moderate users and content, verify accounts.

5 System Architecture

5.1 Frontend

Stack: React.js (for Web) and Flutter (for Mobile)
Features:

- Modular architecture using shared components.
- Authentication, profile management, application status, and teams management.

- Interactive dashboards for both companies and students.
- real-time messaging via WebSocket

5.2 Mobile App Architecture

Stack: Flutter using BLoC pattern

Technologies:

- Get_it – Dependency injection
- dio – HTTP client
- freezed – Model serialization
- Bloc - state management
- Flutter Secure Storage -tokens storage

Folder Structure and Architecture:

Key Principles: This architecture combines Clean Architecture principles with BLoC pattern for state management, ensuring separation of concerns, testability, and scalability.

Core Benefits :

- Separation of Concerns: Each layer has a distinct responsibility
- Testability: Business logic is isolated and easily testable
- Scalability: Modular structure supports growing applications
- Maintainability: Clear boundaries make code easier to maintain

Core Directory Structure

Purpose: Shared app infrastructure and cross-cutting concerns

```
core/
├── dioservices/        # Custom Dio client with interceptors
├── dio_client.dart # Base HTTP configuration
#Auth/error /interceptors
├── connection/         # Network monitoring
│   └── network_info.dart
└── failure/            # Error handling
    ├── failure.dart   # Base Failure class
```

Key Components

- DioServices: Centralized HTTP client configuration
- Connection: Network connectivity monitoring
- Failure: Standardized error handling across the app

Bloc architecture implementation :

Layer	**Responsibility**	**Dependencies**
Application	UI + BLoC state management	Depends on Domain
Domain	Business logic, entities, Repositories	No dependencies (pure Dart)
Data	Data sources, models,	Depends on Domain

Dependency Rule: Presentation → Domain ← Data

Inner layers know nothing about outer layers, ensuring loose coupling**.**

Usage of Service locator (get_it) to ensure safe, scalable DI (dependency Injection)

Bloc Implementation:

Unidirectional Data Flow

UI Event → BLoC Processing → State Update → UI Rebuild

Complete Data Flow Example:

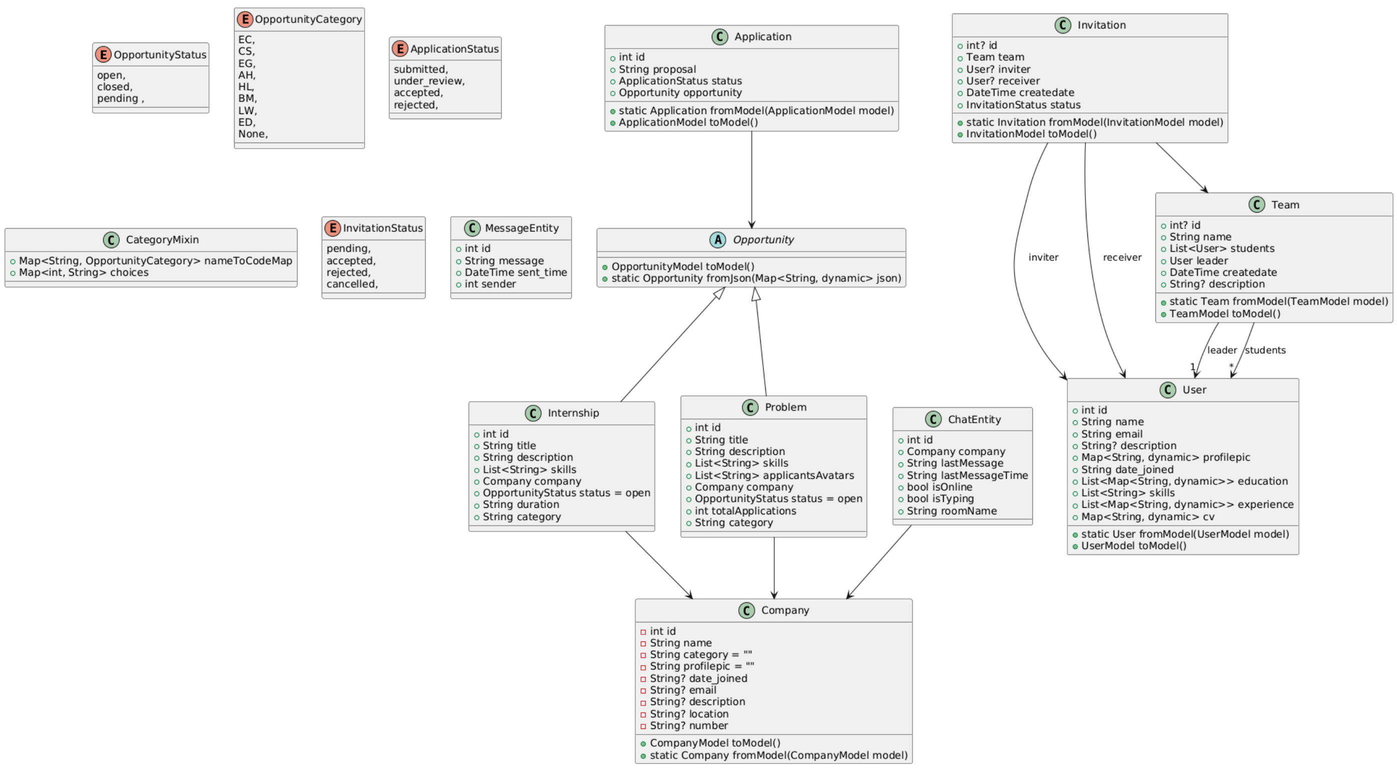
Step-by-Step Process:

1. UI Interaction: User taps login button
2. Event Dispatch: LoginButtonPressed event sent to BLoC
3. Repository Call: The BLoC Calls the AuthRepository
4. Data Source: Repository calls remote/local data source
5. Response Chain: Data flows back through layers
6. State Update: BLoC emits new state
7. UI Update: Widget rebuilds based on new state

Shared and Utils modules :

```
shared/

├── widgets/           # Reusable components

│   └── loading_indicator.dart

└── pages/             # Common screens

    └── no_connectionscrean_screen.dart

ustils/

 └── theme # Theme Provider

 └── error_page # Global user Friendly Error page

 └── locator # Service Locator Responsible for the Di
```

business (domain) layer:



5.3 Frontend Web (React.js)

The frontend of the Grow Path platform was developed using React 19 with TypeScript and follows a modular, component-driven architecture. It ensures accessibility, responsiveness, and a seamless user experience for students, companies, and administrators. The system integrates real-time messaging, AI-powered features, and dynamic application workflows—all optimized for performance and maintainability.

1. Features and architecture:

1. Core Features

A. Authentication System

- JWT-based Authentication
  - Secure token management
  - Role-based access control (Student, Company, Admin)
  - Protected routes implementation

    - Social authentication (Google, LinkedIn)

B. Real-time Chat System

- WebSocket Integration
    - Instant messaging between users
    - Message persistence and history

C. AI Interview System

- Gemini AI Integration
    - Context-aware interview sessions
    - Smart response generation
    - Progress tracking
    - Session persistence
- Web Speech API Integration
    - Speech-to-Text for user responses
    - Text-to-Speech for AI feedback
    - Real-time transcription

D. CV Builder

- Interactive Interface
    - Drag-and-drop section reordering
    - importing data from database
    - Multiple export formats (Latex)

Animations and UX

- Framer Motion is used to animate modal transitions, page changes, and in-app notifications.
- Toasts and feedback messages (e.g., from Gemini) are animated and thematically styled based on status (success, warning, error).
- Smooth scroll, typing indicators, and hover states enhance the overall user experience.

Performance Optimization

- React.lazy and Suspense are used for code splitting, loading large components (e.g., dashboards) only when needed.
- Memoization (useMemo, useCallback, React.memo) ensures re-renders are optimized for performance-critical sections.
- All static assets and images are served with lazy loading and responsive sizing.
- React Query's caching and stale-while-revalidate behavior reduce unnecessary API calls.

Tooling, Deployment & CI/CD

- Built with Vite: Enables faster hot-module reloads and faster production builds compared to Webpack.
- Dockerized Setup: The frontend is deployed via a multistage Dockerfile (build → Nginx).

- Nginx serves the compiled static frontend and handles caching of assets.
- Environment configuration is managed through Vite's import.meta.env API with .env files for development and production.
- GitHub Actions (optional): Linting, formatting, and deployment workflows.

Developer Experience

- Reusable UI library: Shared button, form, modal, and card components reduce code duplication.
- Custom hooks: (useAuth, useWebSocket, useToast) encapsulate logic for maintainability and reuse.

-OOP in Services (Inspired by Angular)

All API logic is encapsulated in dedicated service classes (e.g., AuthService, PostService, TeamService). These classes expose clearly defined methods (e.g., login(), getUserById(), applyToOpportunity()) which encapsulate all request logic, headers, error handling,

-Singleton Pattern for Service Instances

To prevent redundant instantiations of service classes and maintain shared state (e.g., headers, config), we use the Singleton Pattern. Each service class has a static getInstance() method that returns the same instance every time. This guarantees:

- A single source of truth for service logic
- Controlled access to configuration (e.g., auth tokens)
- Predictable behavior in asynchronous workflows

This structure improves testability and allows us to mock services easily in unit tests.

3. API Call Flow: Service → Hook → Component

We follow a clean and consistent architecture for data fetching and logic handling:

1. Service Method: Handles all API calls (Axios request, error parsing, token injection).
2. Custom Hook: Calls the service and encapsulates logic (caching, loading states, React Query integration).
3. Component: Uses the hook to access the data, showing loaders, errors, or the result.

This cycle keeps UI logic separate from data logic, making the app easier to maintain and debug.

4. Firebase Notifications (Cloud Messaging)

To enable push notifications (especially for students and companies), we integrated Firebase Cloud Messaging (FCM) in the frontend:

- On login, the browser requests a device token via the Firebase SDK.

- The token is sent to the backend and stored in the user’s record.
- When a backend event is triggered (e.g., application accepted, message received), the backend uses FCM to send a push notification.
- The frontend listens using the Firebase onMessage API and shows a custom toast or native browser notification.

2. Project architecture :

The frontend web application of Grow Path was designed with a modular architecture in mind, enabling separation of concerns, scalability, and ease of collaboration between team members. Each feature and domain in the application is encapsulated into independent modules or domains, following best practices inspired by modern frontend architectures such as Atomic Design and feature-based folder structure.

```
src/
├── api/           # API services and interceptors
│   ├── axios.ts      # Axios configuration
│   └── endpoints.ts  # API endpoints
├── components/      # Shared UI components
│   ├── ui/         # Basic UI components
│   └── layout/     # Layout components
├── lib/           # Utility functions
│   ├── hooks/      # Custom hooks
│   └── utils/      # Helper functions
├── modules/        # Feature-based modules
│   ├── auth/       # Authentication module
│   ├── chat/       # Chat module
│   ├── profile/    # User profile module
│   └── interview/  # AI interview module
└── stores/        # State management
    ├── auth.ts     # Authentication store
    └── ui.ts       # UI state store
```

5.4 Backend

The backend of **Grow Path** was designed to provide a robust, secure, and scalable API infrastructure that serves as the core logic and data management layer of the application. The platform supports two main user roles — students and companies — enabling internship applications, challenge submissions, real-time communication, and secure authentication.

Features:

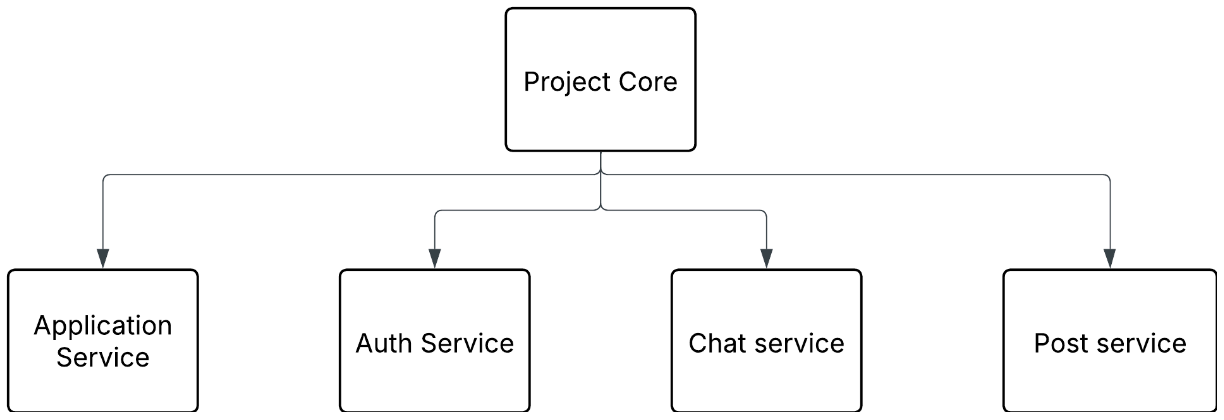
- RESTful API design with authentication and role-based access
- Background task processing with Celery and RabbitMQ
- Notification system using Redis pub/sub
- Secure file handling

Technologies:

- django + django rest framework.
- artillery for load test.
- supbase for store files and secure them.
- firebase for push notification .
- websocket for real time communication

System Architecture

- Core Apps:



1_Auth Service:

Purpose:

Handles all authentication and authorization-related functionalities across the platform, ensuring secure access control and user identity management.

Key Features:

- JWT-based Authentication System: Issues JSON Web Tokens upon successful login, enabling stateless authentication for API requests.

- User Registration and Profile Management: Allows users to register, update their profiles, and manage account settings.

- Role-based Access Control (RBAC): Defines specific permissions and access levels for Students, Companies, and Admins.

- Password Reset and Email Verification: Enables users to securely reset forgotten passwords and verify their email addresses upon registration.

- Social Authentication (Google, LinkedIn): Supports OAuth2-based login via Google and LinkedIn for quicker onboarding and increased security.

Implementation Details:

- Framework: Built using Django REST Framework (DRF), leveraging its built-in authentication mechanisms.

- Custom User Model: Extends AbstractUser to incorporate custom fields (e.g., user role, company name for company users).

- JWT Authentication: Utilizes the SimpleJWT package to generate and validate access and refresh tokens.

- Custom Permissions: Implements DRF custom permission classes to enforce RBAC.

- Email Services: Integrates with Django’s email backend (SMTP or third-party like SendGrid) to send verification and reset links.

- Social Auth: Uses django-allauth or social-auth-app-django to handle OAuth2 flows from Google and LinkedIn.

2_Application Service:

Purpose:

Manages all student application workflows for job/internship opportunities and collaborative team formations.

Key Features:

- Application Submission and Management: Allows students to apply to opportunities posted by companies and track the status of each application.

- Team Formation and Collaboration: Students can create or join teams to submit collaborative applications.

- Application Status Tracking: Maintains a lifecycle state (e.g., Submitted → Under Review → Accepted/Rejected).

- Team Member Approval System: Requires team leader or all members to approve new join requests to maintain collaboration integrity.

- Application Notifications: Sends email and in-app notifications for application updates, status changes, and team events.

Implementation Details:

- Custom Application Model: Tracks each application's metadata including status, opportunity reference, submission date, and team.

- Team Model: Represents teams of students, each linked to applications. Includes join requests, member roles, and approval logic.

- Approval System: Custom logic to manage member invitations and consensus-based approvals before submission.

- Notification System: Uses Django signals to trigger email or in-app notifications upon events like application submission or team changes.

- Search Capabilities: Elasticsearch integration allows efficient querying and filtering of applications using fields like status, date, or keywords.

3_Post Service:

Purpose:

Enables companies to post and manage job/internship opportunities and allows users to browse and search for relevant posts.

Key Features:

- Opportunity Creation and Management: Companies can create, update, and delete postings, including application deadlines and role descriptions.

- Search and Filtering: Advanced filtering by tags, categories, locations, deadlines, and other metadata.

- Post Analytics: Tracks views, applications received, and engagement over time.

- Category and Tag Management: Enables organization and discoverability of posts using taxonomies.

- Application Tracking: Companies can monitor the application count and status for each post.

Implementation Details:

- Custom Post Model: Contains rich metadata, including title, description (rich text), deadline, required skills, etc.

- Rich Text Support: Uses django-ckeditor or similar package for formatting job descriptions.

- Search Functionality: Integrates Elasticsearch to index and retrieve posts based on full-text search, filtering, and relevance ranking.

- Tags and Categories: Leverages django-taggit or a custom tagging system to allow users to browse by interest area or skillset.

- Analytics Tracking: Middleware or custom model fields log each view and interaction with posts.

4_Chat Service:

Purpose:

Facilitates real-time communication between users, particularly team members and companies, enhancing collaboration.

Key Features:

- Real-Time Messaging: Allows users to send and receive messages instantly via WebSocket connections.

- Message History: Stores all messages so users can view past conversations even after logout or disconnection.

- Read Receipts: Displays delivery and read statuses for sent messages to ensure effective communication.

Implementation Details:

- Real-Time Communication: Uses Django Channels to handle asynchronous

WebSocket connections, enabling low-latency messaging.

- Channels Layer (Redis): Redis acts as the channel layer backend, supporting group messaging and message broadcasting.

- Message Model: Each message is saved to a PostgreSQL database with metadata like timestamp, sender, receiver, and read status.

- Chat Rooms: Implements private and group chat logic using unique room names derived from user or team identifiers.

-

Infrastructure:

1. Docker:

- **Why:** Containerized apps to ensure they run the same everywhere.
- **Use Case:** Run Django, Redis, RabbitMQ, Celery, and Elasticsearch in isolated, consistent environments.
- **Benefit:** Easy setup, portability, and scaling.

2. Celery (Background Jobs):

- **Why:** Runs long tasks asynchronously.
- **Use Case:** Sending emails, processing uploads.
- **Benefit:** Keeps app fast and responsive.

3. Redis (as Cache):

- **Why:** Fast in-memory key-value store used to reduce database load.
- **Use Case:** Caching API responses, sessions, or frequently accessed data.
- **Benefit:** Boosts performance by serving data quickly.

4. RabbitMQ (as Message Broker for Celery):

- **Why:** Queues and routes tasks to Celery workers.
- **Use Case:** Delivering background jobs reliably to Celery.
- **Benefit:** Ensures reliable, scalable task distribution.

5. Elasticsearch (Search Engine)

- **Why:** Fast, powerful full-text search.
- **Use Case:** Searching products, users, or articles.
- **Benefit:** Better search speed and accuracy than SQL.

6 Database Design

- PostgreSQL relational schema with tables for users, applications, messages, opportunities, and teams.
- Celery workers store background task results and statuses in Redis.

6.1 ***Entity-Relationship Diagram***

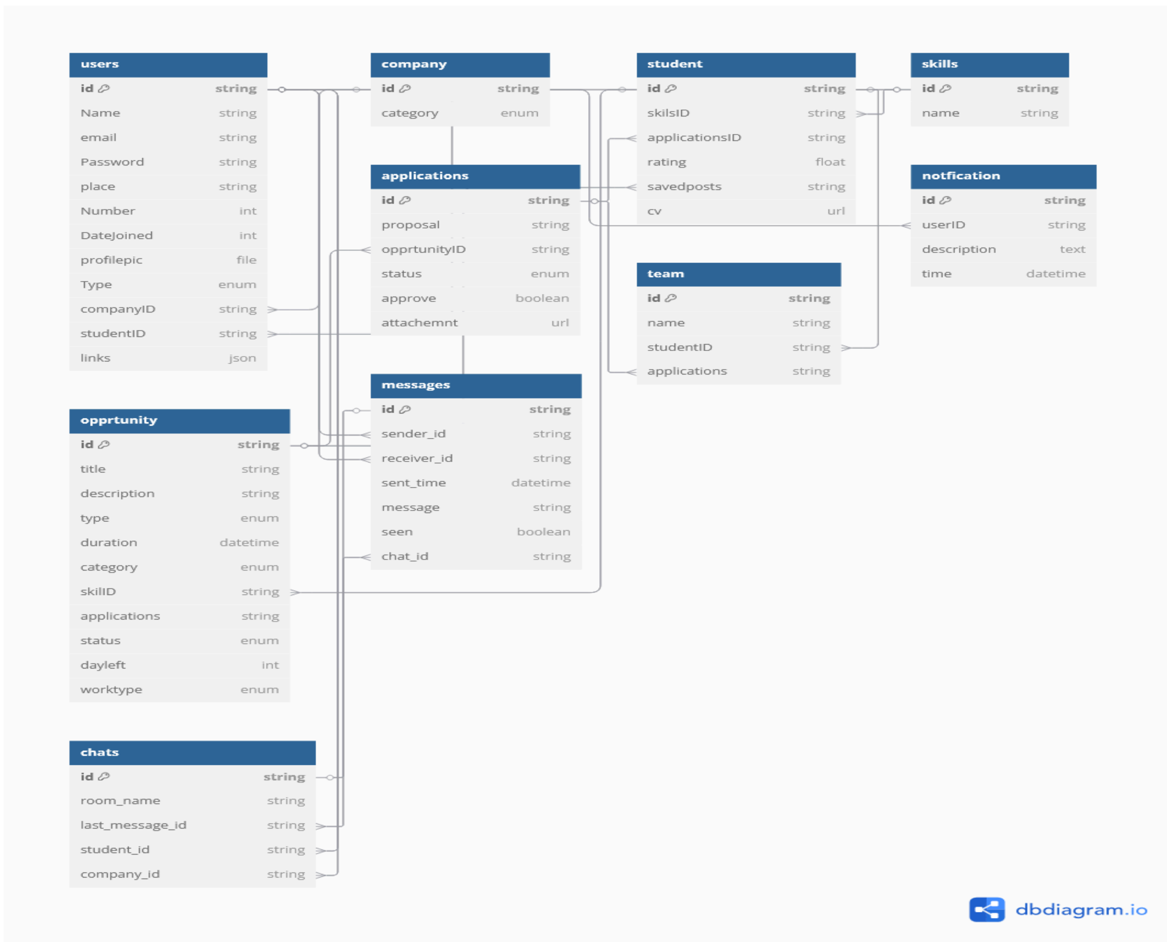


Figure 1: Entity-Relationship Diagram

7 UI and UX

7.1 Web UI

- Tailwind CSS + Shadcn UI for consistent styling.
- Responsive design with intuitive layout for students and companies.

7.2 Mobile UI

- Flutter widgets and reusable components.
- State-managed using BLoC for predictable data flow.

8 Key Flow Diagrams

8.1 Company Flow

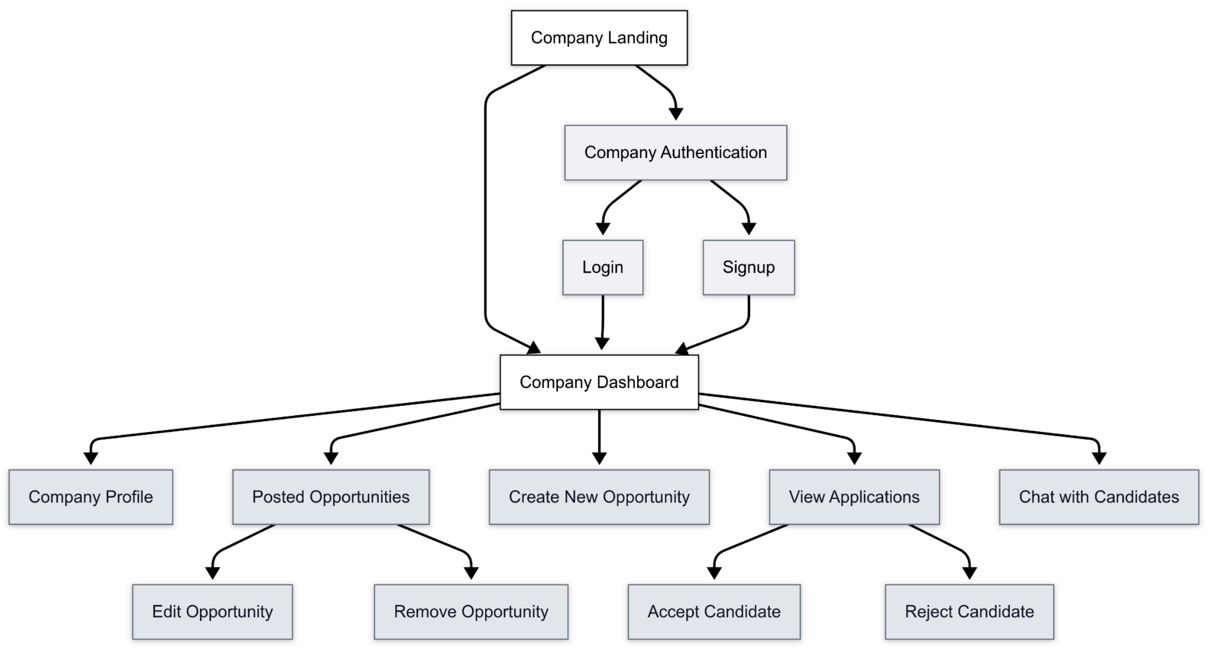


Figure 2: Flow diagram for company

8.2 Student Flow

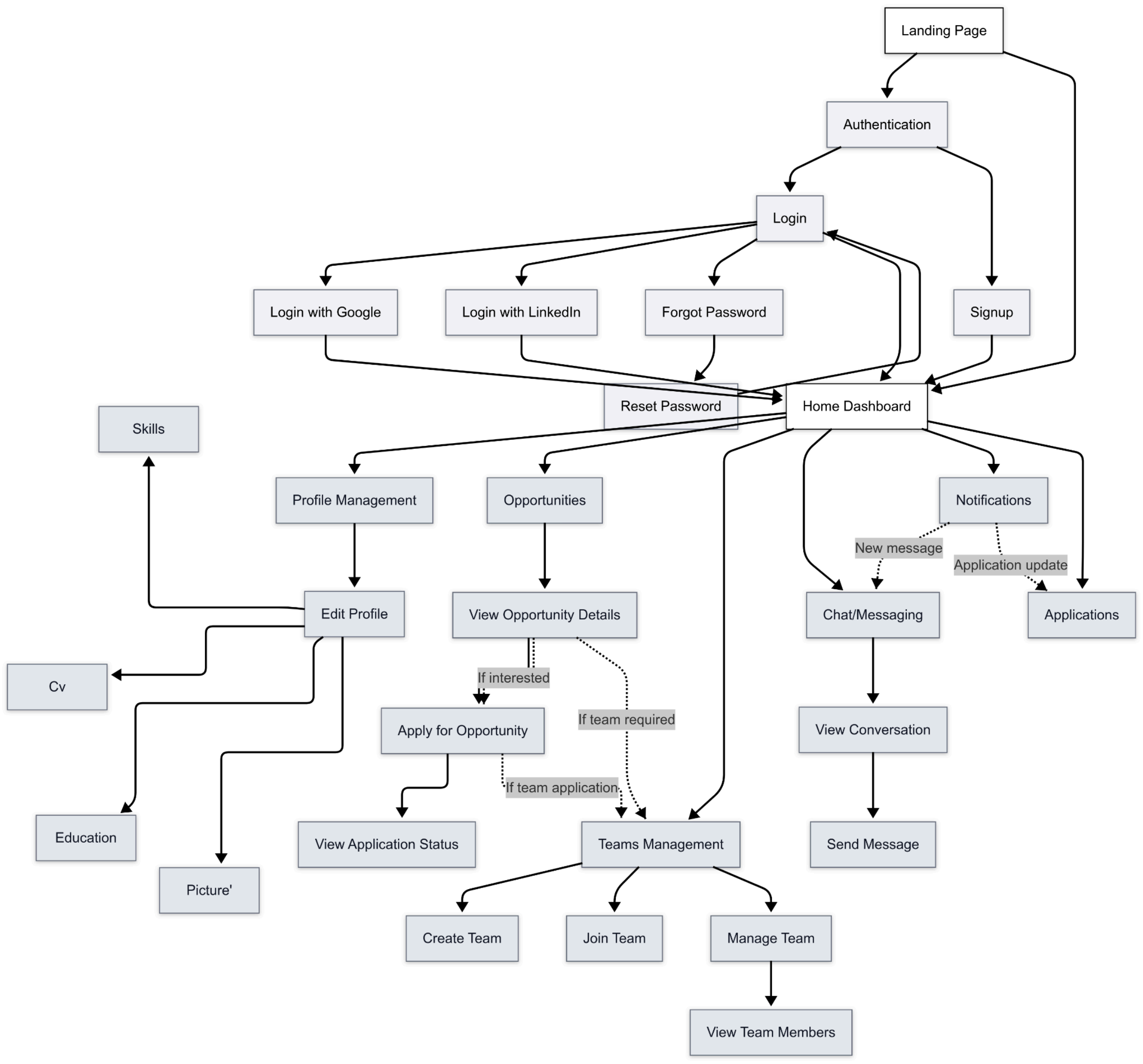


Figure 3: User Flow Diagram Across Web and Mobile Student

9 Tools and Technologies Used

- **Frontend:** Next.js, Tailwind CSS, Shadcn UI
- **Mobile:** Flutter, BLoC, Dio, GetIt, Freezed
- **Backend:** Django, PostgreSQL, Redis, Celery, RabbitMQ
- **File Storage:** Supabase
- **Design:** Figma
- **DevOps:** GitHub, Docker, CI/CD

10 Conclusion

We designed and implemented a full-stack platform that enables students to engage in real-world challenges and internships while helping companies recruit efficiently. This system demonstrates scalable architecture and thoughtful UX across web and mobile platforms. The platform serves as a model for future educational-industry collaborations.

Acknowledgments

We would like to thank our supervisor, Mr.Serhane for his continuous guidance and support. Our thanks also extend to all our peers and contributors.