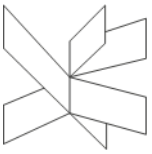


HireFire – Project Report

Hamsa Abdullah Sheikhdon (354974)

Tymoteusz Krzysztof Żydkiewicz (355413)

Caranfil Cristian (331164)

Damian Michał Choina (354789)

Jakub Maciej Bączek (354814)

Supervisors:

Jakob Trigger Knop

Joseph Chukwudi Okika

Number of characters: 54789

Software Technology Engineering

3rd Semester

19.12.2025

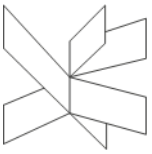
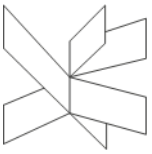
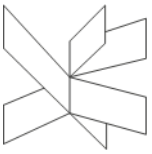
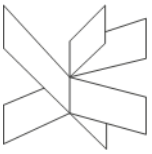


Table of content

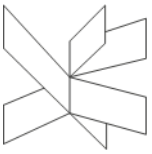






1. Abstract

This document describes the creation of an application used for job-hunts. The development aims to help potential employees with their job search while enhancing the ability of recruiters to seek out new talents. It opts to minimize the amount of time both parties have to spend during the application process. The applicants apply for or ignore job opportunities without having to spend more time than is required to read the listing. Same goes for recruiters - they decline and accept applicants without having to sacrifice time unnecessarily. If they feel more information is needed, a real-time chat



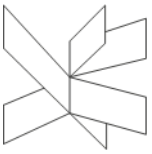
feature is available for successful applications. This provides for a more genuine experience and gets both parties in direct contact at an early stage of the recruitment process.

The system is implemented using a 3-tier architecture and thus consists of a client and 2 servers bearing various responsibilities which are described in detail in the following sections of this report. The advanced design provides exceptional security, by leveraging the https protocol and a complex hashing function used for passwords. It additionally allows for potential future modifications and extensions to be implemented with ease and without the disruption of the general flow of the system thanks to a clear separation of concerns present in every tier of the application.

2. Introduction

The world of digital recruitment is undergoing major and rapid reformation as both job seekers and employees convey growing displeasure in the traditional hiring platforms. Despite the abundance of known job portals, such as LinkedIn, Indeed, Jobnet and etc, research shows that applicants who frequently use these portals come face to face with unclear application processes, long waiting times for responses, a more stressed recruitment process and very limited assistance within these portals. These problems underscore a normed gap between what applicants want and what they expect in a job portal: clarity, efficiency and usability. Within this domain, there has been an increasing demand for job portals that revamp the whole recruitment process in a way that is optimised in the formerly mentioned ways: clarity, efficiency and usability. By doing this we can achieve an overall better user experience, a more streamlined process for job applications and an interactive alternative to mainstream online recruitment networks.

HireFire intercepts this problem by reconsidering how applicants interact with job postings and recruitment workflows. A mention in the problem statement within the project description, more and more job seekers want the process to be simple and

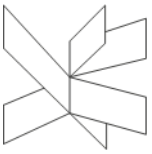


transparent. Many find existing platforms frustrating to use, as they are often overloaded with information and lack user-friendly design. The solutions we have developed are numerous and effective, as we have introduced a straightforward and intuitive model, where applicants can easily browse postings and even chat with recruiters through a simple, accessible interface.

Henceforth, in practicality the substance of HireFire really aims beyond mere academic exploration and exercise by examining a unique way for job portals to present and administer applicant interactions. The below sections now will further elaborate on this foundation. The sections go in detail about the development process of HireFire, the design, implementation and the conclusion after thoughtful testing of the HireFire software. By closely inspecting these components in detail, this report aims to illustrate how HireFire aims to provide a more streamlined and applicant-friendly approach to navigating the modern job market.

3. Analysis

The recruitment industry has always faced effectiveness challenges in terms of hiring and job-search workflow management, the stakeholders - including the main customers: job seekers, and employers - have varied requirements and expectations regarding the operation of the system, emphasizing the need for efficient project execution and effective communication. The analysis section of this report addresses these requirements by providing diagrams to outline the system’s structure clearly, such as use case and sequence diagrams and detailed information about the main users. To facilitate a deeper understanding of the system’s functionality and purpose, all the use case descriptions derived from the requirements will be introduced in a brief format. The most complex use case descriptions will be presented in full dress format. This detailed examination helps in understanding how these use cases interact with and support the system's architecture and user needs.



3.1 Requirements

The requirements for HireFire are concluded from matters defined in Project Description: analysis of the problem domain, delimitation and problem formulation. As a result, the requirements are categorised into functional and non-functional, following the SMART (Thomas & Hunt, 2020, #), FURPS (Larman, 2004, #) and INVEST principles (Buglione & Abran, 2013, #).

3.1.1 Actors description

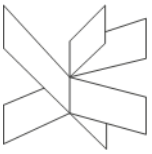
The actor descriptions demonstrate how functionality and access to features are distributed across different user types. The actors represent the 4 types of users the system has.

Company Representative - Users who represent a company in the system. The main functionalities they require from the system are to register and manage the company profile, create and manage recruiter accounts, and remove the company profile from the platform.

Recruiter - The system allows recruiters to manage job listings. Their main functionalities are to publish, edit, close, and remove job listings, view applicant profiles, accept or decline applications, and chat with matched applicants to continue the recruitment process.

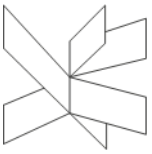
Applicant - Users who search for job opportunities. The main functionalities they require from the system are to browse recommended job listings based on skills and location, view company details, create and update a professional profile, track application status, chat with recruiters after matching, and remove their account.

Admin - The system allows admins to control company verification. The main functionality they require from the system is to review and approve or remove companies.



3.1.2 Functional requirements

1. As a company representative, I want to create my representative account so that I can manage my company presence.
2. As a user, I want to log in to the system with my username and password so that I can securely access all functions relevant to my account.
3. As a user, I want to log out from the system, so that I can finish my session.
4. As a company representative, I want to register a company in the system so that I can share my information with potential applicants.
5. As a company representative, I want to create recruiter accounts so that authorized employees can represent us on the platform.
6. As an applicant, I want to create a professional profile with my skills, experience, and location so that recruiters can learn more about me.
7. As an Admin, I want to review, approve or remove company registration requests so that only legitimate companies are verified and receive a verification badge.
8. As a company representative, I want to edit my representative account details so that my contact and identity data are correct.
9. As a company representative, I want to edit the company profile so that applicants always see up-to-date information about us.
10. As a recruiter, I want to publish job listings so that applicants can discover and apply to them.
11. As an applicant, I want to browse through available job listings so that I can quickly find positions that interest me.
12. As an applicant, I want to view only job listings that match my skills and preferred location, so that I can focus on opportunities that are relevant to me.
13. As an applicant, I want to see the detailed information about companies responsible for the job listings so that I know exactly what I’m applying for.
14. As a recruiter, I want to view full applicant profiles so that I can evaluate their skills and experience before making a decision.
15. As a recruiter, I want to accept or decline applicants so that I can choose the ones who match my requirements.
16. As an applicant, I want to see the status of my applications so that I know if I am under review, declined, or matched.
17. As an applicant, I want to instantly connect with recruiters when there is a match so that I can start a conversation about the job.



18. As a recruiter, I want to be able to chat with applicants that I match with so that I can get in contact with them.
19. As a recruiter, I want to edit job details so that I can correct mistakes or update requirements without creating a new post.
20. As a recruiter, I want to be able to close job listings so that candidates no longer apply to positions that are already filled or inactive.
21. As a recruiter, I want to be able to remove job listings so that outdated or irrelevant postings do not clutter the platform.

22. As a company representative, I want to edit recruiter accounts so that our team's information and access remain accurate and up to date.
23. As a company representative, I want to remove recruiters, so that former or unauthorized employees can no longer represent us on the platform.
24. As an applicant, I want to edit my profile so that I can keep my personal information, skills, and preferences up to date.
25. As an applicant, I want to remove my account so that I can permanently delete my data and stop using the platform.
26. As a company representative, I want to remove my profile from the system so that I can permanently withdraw our presence and data from the platform.
27. As a company representative, I want to permanently delete the company profile so that the companies which no longer exist are not in the system.

3.1.3 Non-Functional Requirements

1. The system will be distributed and heterogeneous, implemented using Java, C#, Blazor, and CSS.
2. The system will follow a three-tier architecture, consisting of:
   - a data tier (server connected to the database),
   - a logic tier (application server handling business logic),
   - a presentation tier (client application).
3. Communication between servers will be established using gRPC.
4. Communication between the server and the client will be established using REST.
5. The user interface of the presentation tier will be implemented in Blazor (C#).

6. The database will be developed using DataGrip and implemented in PostgreSQL.
7. The system will support hash-based authentication as a security measure.
8. The system will include a real-time chat feature implemented using SignalR.

3.2 Use case diagram

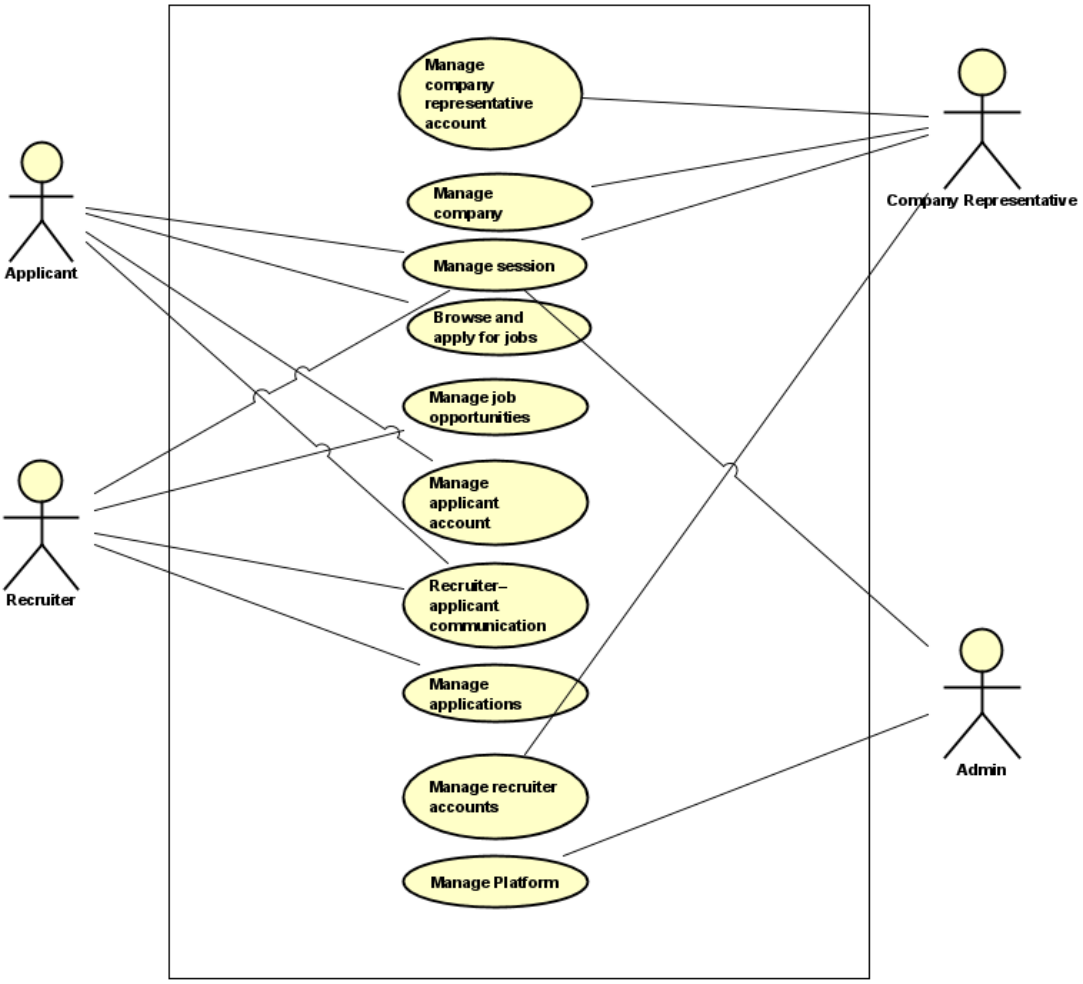
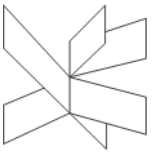


Figure 1 Use Case Diagram

The use case diagram outlines the interactions within the system between the actors: Company Representative, Recruiter, Applicant and Admin.The functions they are

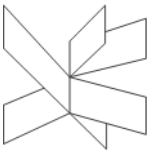


enabled to perform are showcased through the use cases. (Connolly & Beg, 2015, #) (Connolly et al., 2015, #).

Link between requirements and use cases

Use cases	Requirements
Use case 1:Manage Recruiter Accounts	5, 22, 23
Use case 2: Manage Company	4, 9, 26, 27
Use case 3: Browse and Apply for Jobs	11, 12, 13
Use case 4: Manage Job Listings	10, 19, 20, 21
Use case 5: Manage applicant account	6, 24, 25
Use case 6: Recruiter–Applicant Communication	17, 18
Use case 7: Manage Applications	14, 15, 16
Use case 8: Review company registration request	7
Use case 9: Manage company representative account	1, 8
Use case 10: Manage session	2, 3

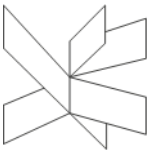
Table 1: Link between requirements and use cases



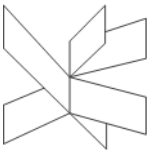
3.3 Use case description

Use case 3

Use case	Browse and Apply for Jobs
Summary	Actor views Job Listings and can apply for them
Actors	Applicant
Pre-condition	1. Actor has the application open 2. Actor is logged in and has Applicant privileges
Post-condition	Actor applied for the positions he liked
Base sequence	1. View the recommended Job Listings [ALT1] 2. A Job Listing is shown 3. Apply - Scenario A Decline - Scenario B View Profile - Scenario C View sent applications - Scenario D



	Scenario A: Apply for position	A.1 Choose to apply for the position A.2 An application is created and has status pending, actor is shown another position.
	Scenario B: Decline position	B.1 Choose to decline the position B.2 The actor is shown another position
	Scenario C: View Company Profile	C.1 Choose to view the Company Profile responsible for job listing C.2 The full company information is displayed
	Scenario D: View application status	D.1 Choose to view sent applications status



	<table><tr><td></td><td>D.2 Actor see all his applications and theirs status [ALT2]</td></tr></table> Use case ends
Alternate sequences	[ALT1] There is no available job listing for this applicant. Actor is informed Use case ends [ALT2] Applicant has no sent applications, actor is informed
Note	This use case covers requirement 11,18

Table 2 Use case 3

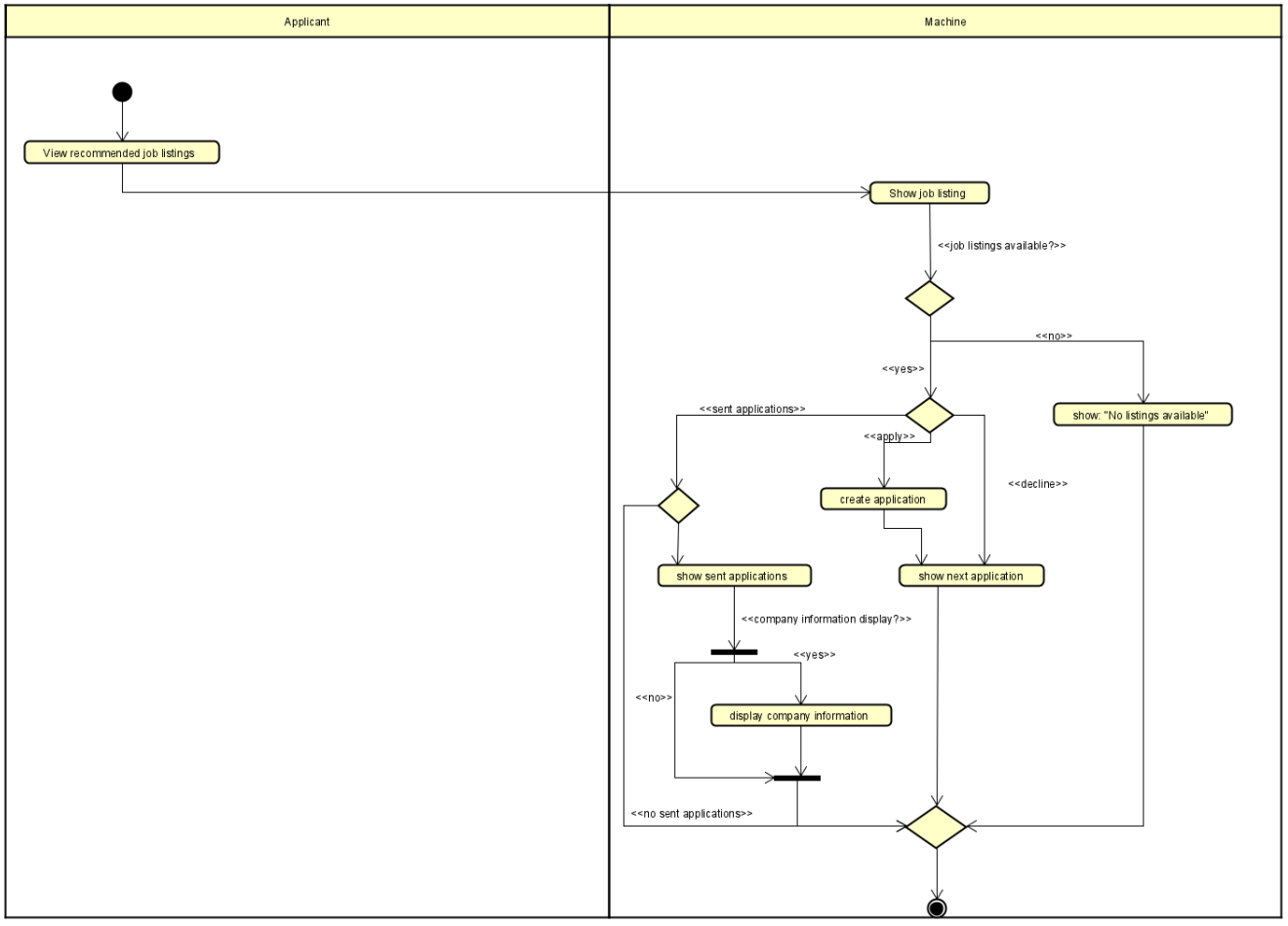
This use case describes a core feature of HireFire: enabling an Applicant to browse available job listings and take action on them. The functionality includes viewing recommended job listings, applying for a selected position, declining a position, and viewing the company profile responsible for a job listing. In addition, the Applicant can view the status of previously sent applications.

Note: For all the use cases refer to attached Appendix D - Analysis

3.4 Activity Diagram

The activity diagram is used for visualization of the steps involved in the use case description. According to “Browse and Apply for Jobs”, the diagram Figure 2 illustrates how the applicant actor views the recommended job listing and how the system

responds by displaying available positions. The schema includes possible actions the Applicant can take, such as applying for a job, declining a position, viewing company information or checking previously sent applications and the status.



3.5 System sequence diagram

A sequence diagram shows how objects are working together, the dynamic interaction between objects and add the time dimension to the sequence of messages sent between objects. The following diagram illustrates the interaction between the applicant and the System for Use Case 3 “Browse and Apply for Jobs”. Each diagram focuses on a specific scenario of the use case and shows how the Applicant’s actions trigger responses from the System.

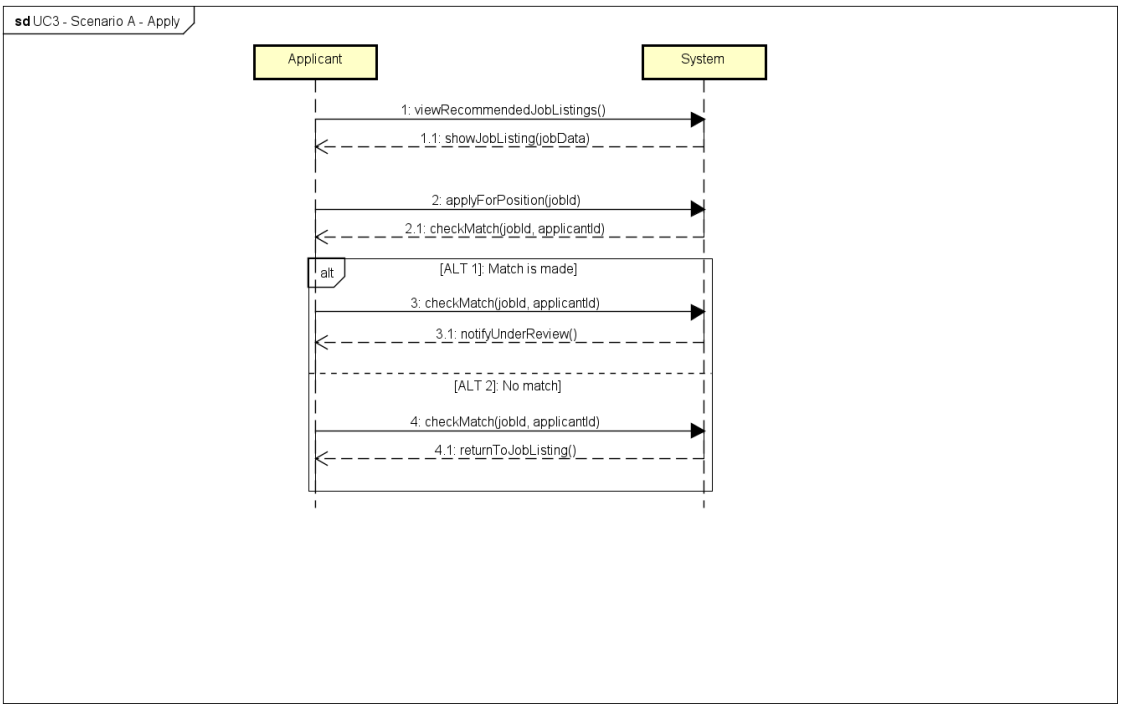


Figure 3 Sequence diagram - Use Case 3 - Scenario A

Figure 3 describes the scenario in which the Applicant chooses to apply for a job. The process begins when the Applicant requests to view recommended job listings, and the System responds by showing the Job Listing. The scenario represents the main flow of

submitting a job application. Figure 4 shows the scenario where the Applicant decides to decline a job position. The start of the sequence is represented by the Applicant requesting the recommended job listings. Once the listing is displayed, the actor declines the position, and the System shows the next job listing.

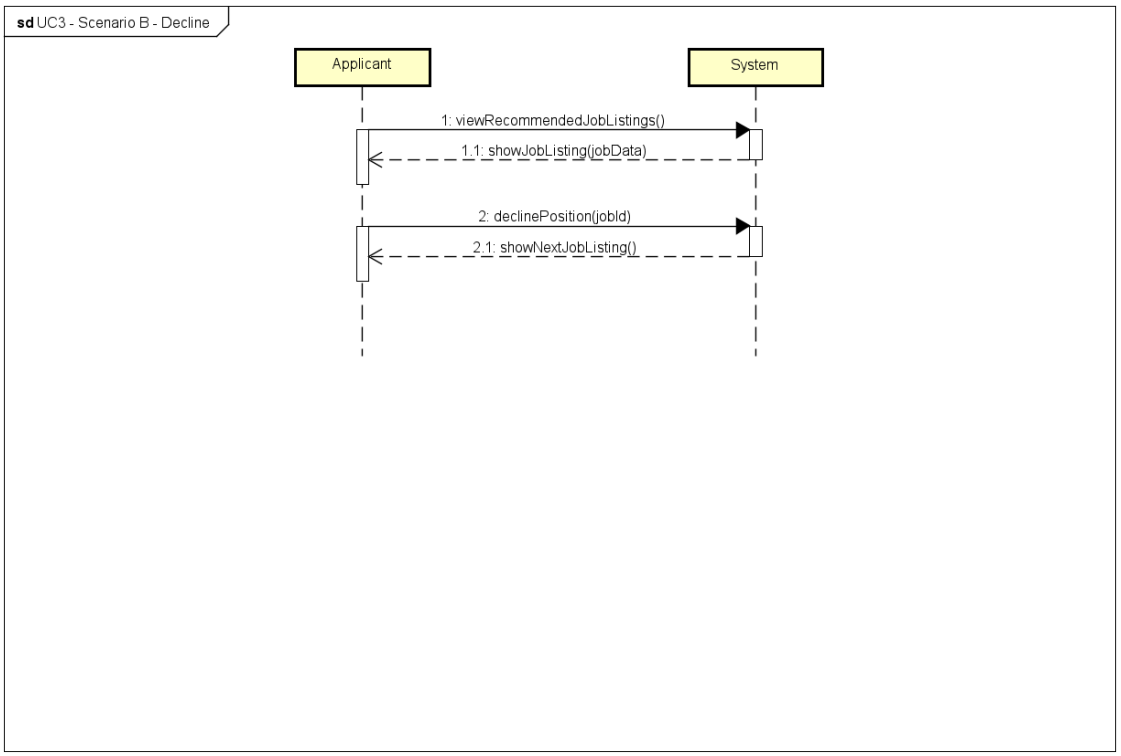


Figure 4 Sequence diagram - Use Case 3 - Scenario B

The sequence diagram from Figure 5 displays how the Actor is opening the company profile from the accepted job listings.

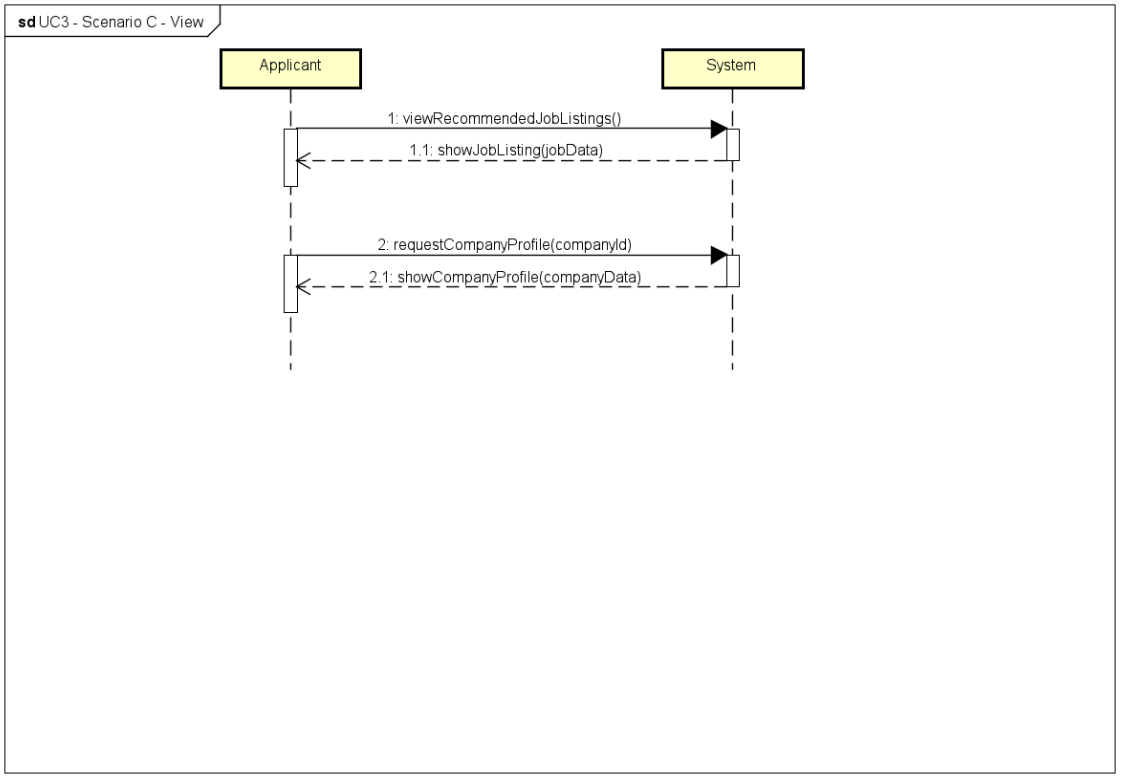


Figure 5 Sequence diagram - Use Case 3 - Scenario C

3.6 Domain Model

The domain model serves as a high-level blueprint of the HireFire system, made from our functional requirements. It maps out the primary entities and roles within the

application, establishing the rules that govern their interaction.

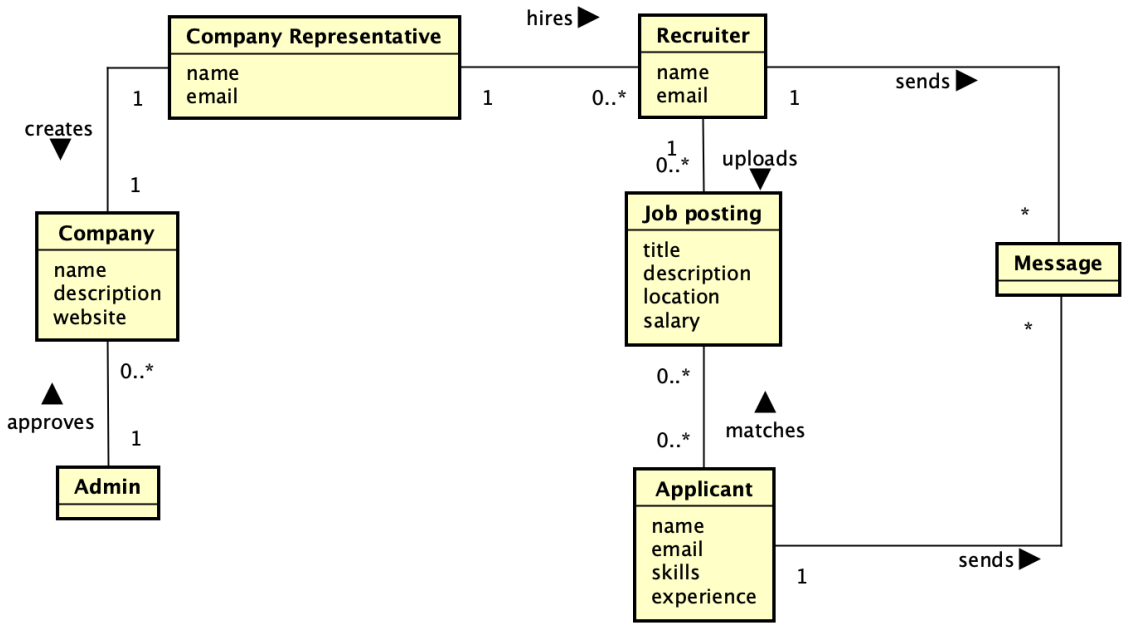
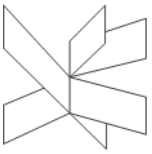


Figure 6 Domain Model

Through the development of the domain model, we visualized the system's core dynamic - interaction between the Company hierarchy and the Applicant. The model illustrates how Company Representatives establish organizations, enabling Recruiters to manage Job Listings. It defines the Application as the critical entity connecting an Applicant to a specific listing. Defining these relationships early set clear rules for the chat feature, ensuring messages are only exchanged when a valid application exists.

3.7 Security

During the development of a distributed system like HireFire, which relies on a 3-tier architecture, we have identified several potential security threats. These threats target the communication channels, API endpoints and data integrity.



3.7.1 Broken Access Control

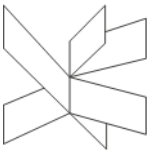
The system uses predictable numbers to identify specific pages or resources. This creates a risk where a user could access someone else's data simply by changing the number in the website address. If the system fails to verify that the user has permission to view that specific page, unauthorized people could easily read private messages or see personal details belonging to other applicants.

3.7.2 Man-in-the-Middle Attacks

The system involves communication across three distinct layers. If these channels are unencrypted, an attacker on the network could intercept sensitive data, including login credentials and personal messages. In this attack, the attacker positions themselves between the user and the system, allowing them to silently monitor or manipulate traffic without being detected. For this reason, the authentication process requires particularly strong security measures.

3.7.3 Denial of Service & Brute Force attacks

The system faces significant risks from automated attacks that threaten its availability and user security. A primary threat is a Denial of Service (DoS) attack where attackers can overload the server by flooding it with fake requests, causing the platform to crash or become too slow for legitimate users.
Additionally, the system is vulnerable to Brute Force attacks on user accounts. Attackers can use automated scripts to endlessly guess passwords until they break in. This puts user accounts - especially those with weak passwords - at high risk of being compromised.



3.7.4 Credential Theft & Data Interception

User account security depends on keeping credentials safe both when they are stored and when they are sent across the network. If passwords are stored in plain text or protected with weak hashing methods, a compromised database could allow attackers to recover passwords and impersonate users.They could also intercept login details over the network (discussed in the Man-in-the-Middle section).

4. Design

In this phase, the report will look into how the system is structured. First section consists of the system's architecture and reason for our choices. Both the domain model and the class diagram will be provided to aid understanding of the system design.

4.1 Architecture overview

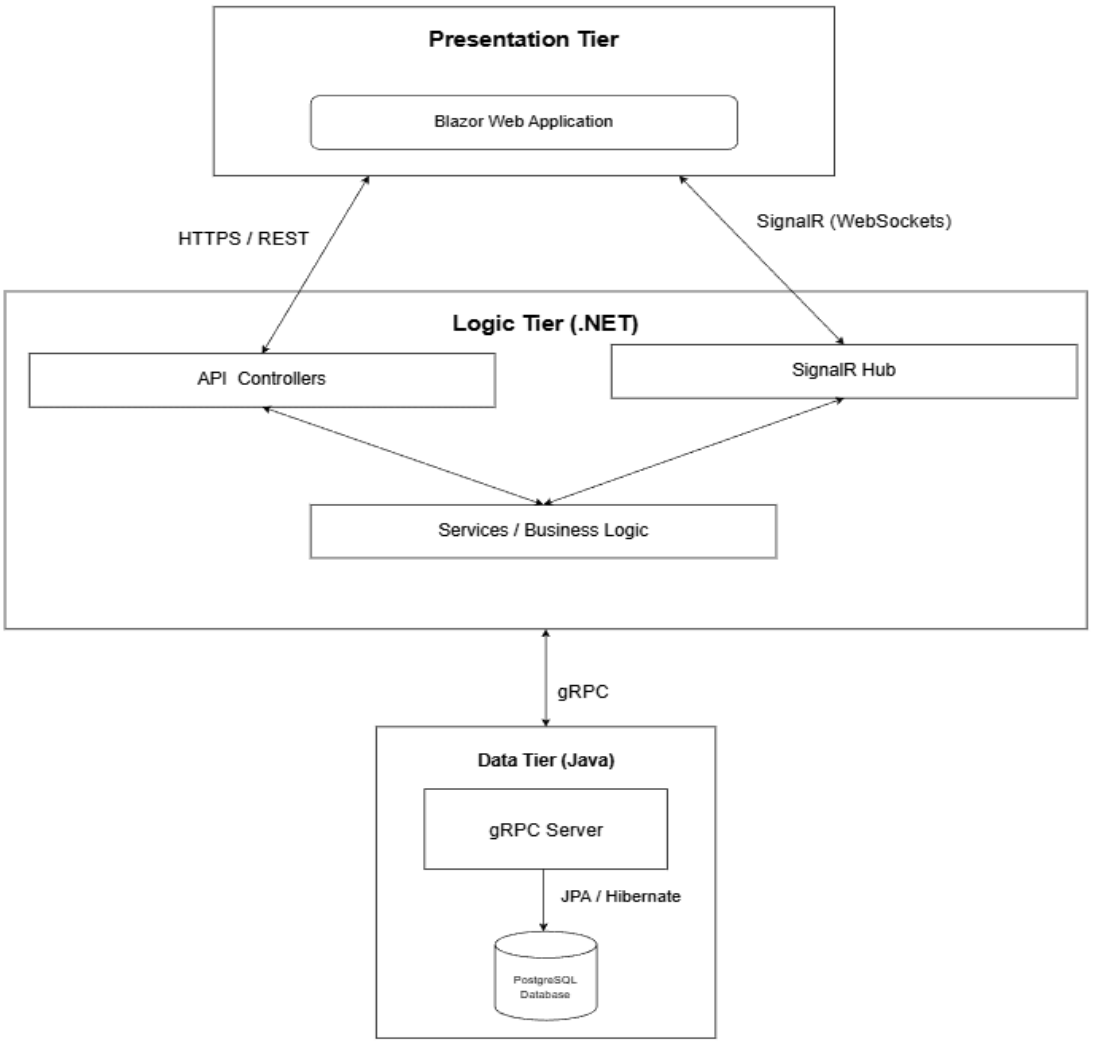
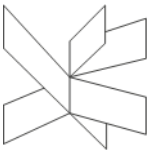


Figure 7 Architectural Model

The HireFire system is implemented using a layered, distributed client-server architecture. Each tier has a clearly defined responsibility which ensures separation of concerns, scalability and maintainability. The project is divided into three main tiers:

- **Client** (Presentation Layer)



- **Logic Tier** (Application Layer)
- **Data Tier** (Persistence Layer)

The client is a Blazor Server application responsible for presentation, user interaction and real-time UI updates. The client communicates with the Logic Tier primarily through RESTful HTTP APIs for standard business operations such as authentication, job listings, application and profile management.

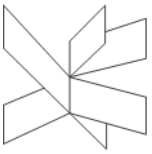
The logic acts as a layer that manages and connects different parts of the system. It exposes REST endpoints which are being used by the client and internally communicates with domain-specific backend services using gRPC for efficient service-to-service communication.

For real-time features (such as chat), the system uses SignalR hubs hosted within Logic Tier, enabling bidirectional communication between clients while persisting data through gRPC calls to backend services.

The Data Tier is implemented as a separate Java-based service responsible for data persistence and database access. It uses PostgreSQL relational database to store all core domain entities, including users, companies, job listings applications, matches and chat messages. All database operations are encapsulated within the Data Tier and sent through gRPC services, ensuring that no other layer accesses the database directly.

4.2 Technologies

In this project we have chosen a mix of different technologies to take advantage of their diverse capabilities for each part of the system. We used .NET (C#) for the website and business logic and Java Spring Boot for the database management, connected by gRPC.



4.2.1 Communication Protocol

- **gRPC**: We used a remote procedure call framework to let the Logic Tier talk to the Data Tier. Instead of sending text like normal APIs, gRPC packs data into a binary format called Protocol Buffers. This makes it much faster and smaller to send over the network. It also ensures that the C# code and Java code understand each other perfectly without errors.
- **SignalR** (WebSockets): For the live chat feature, we used the APS.NET library called SignalR. This tool handles difficult parts of setting up a live connection. It allows the Logic Tier to push new messages instantly to the user's screen without the website having to constantly ask the server if there are new messages.
- **REST / HTTP**: For standard actions that don't need to be instant (like saving a profile), the Blazor client sends standard HTTP requests to the Logic Tier, The client uses "HttpClient" to send data in JSON format.

4.2.2 Frameworks and Languages

- **ASP.NET & Blazor Server**: .We chose Blazor Server for the user interface, because it lets us write web pages using C# instead of e.g. JavaScript. The Logic Tier uses ASP.NET Core to act as the main gateway that organizes data for the client and handles business logic.
- **Java Spring Boot**: The Data Tier is a separate application built with Java Spring Boot. We chose this framework, because it has great tools for connecting to a database as well as excellent support for hosting the gRPC server.

4.2.3 Libraries & Tools

- **Lombok**: A java library that automatically generates repetitive code parts (getters, setters etc.) so our classes remain clean and easy to read.

- **JPA / Hibernate**: These tools handle the connection between our Java code and the PostgreSQL database, automatically converting data between the two.

4.3 Sequence Diagram

Early on in the **Elaboration** phase of the project the team decided to create the sequence diagram shown on the figure below.

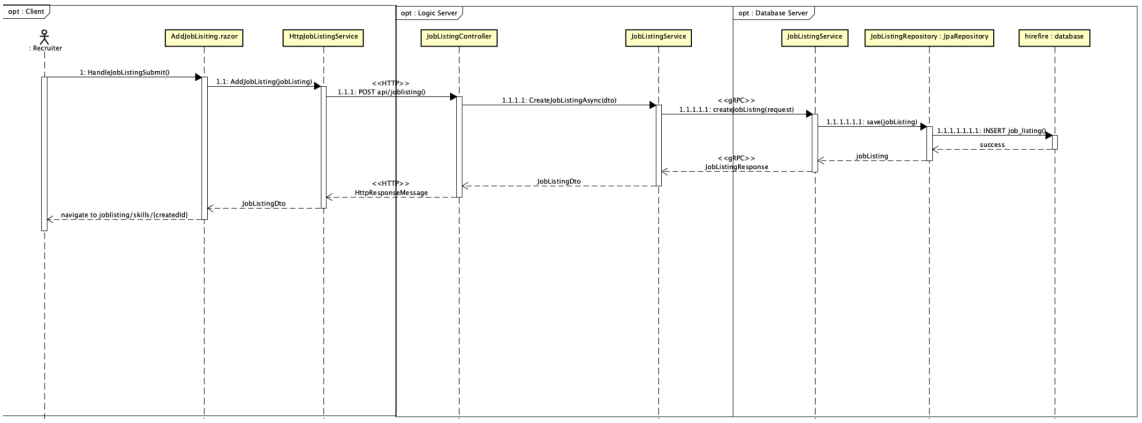


Figure 8 Sequence Diagram - Create job listing

The diagram illustrates the end-to-end flow for creating a job listing. It is extremely helpful in showcasing the interactions of integral parts of each tier of the system. It exposed the important classes and methods needed for the implementation of Use Case 4. The other features present in the system follow a pattern very similar to the one shown in the diagram so the team did not find the creation of other Sequence Diagrams necessary.

4.4 UI's structure and functionality

The application user's interface (UI) was designed to provide an intuitive and reliable experience. Light blue and white colors were chosen for the view pages in order to

ensure a calm, secured and relaxed workspace. Next paragraphs will describe the used UI structure and functionality.

Blazor Server is used to implement the application's user interface, which provides an interactive and responsive web experience that is tailored for different types of users, such as applicants, company representatives, recruiters, and administrators. App.razor creates the application shell and loads international styles and scripts, and Routes.razor handles navigation and authorization through cascading authentication state and role-based route protection. All pages are rendered in a shared main layout. This ensures visual consistency by providing a common header, footer, and content area, as well as handling logout functionality through the authentication provider.

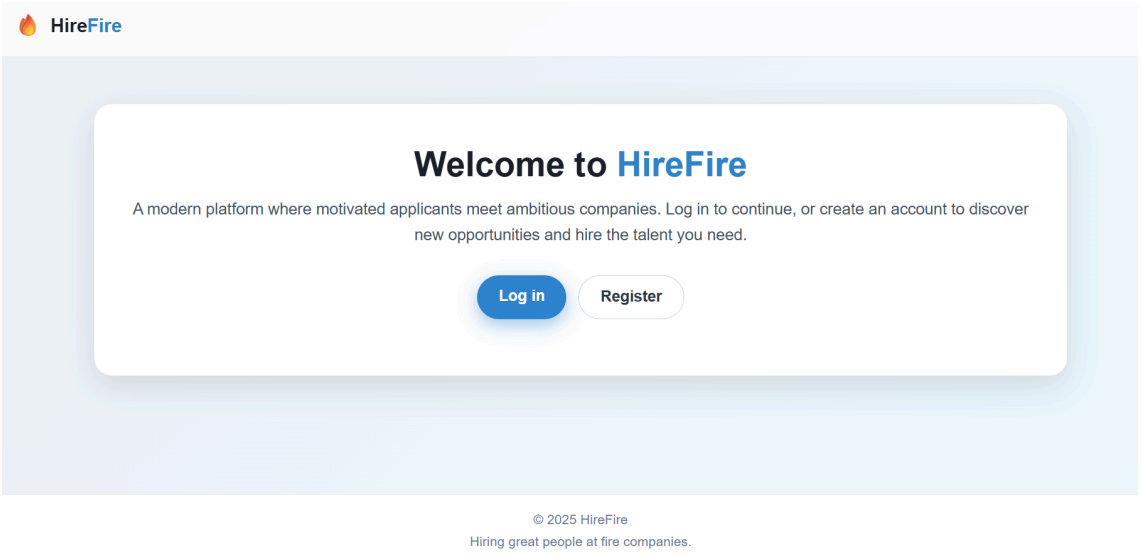


Figure 9 Home Page

The home page prompts users to register and log in. By completing an authentication form, users connect directly to the authentication provider. The onboarding process constantly changes based on the user type. This allows applicants or company representatives to fill out customized forms and receive immediate responses via notifications.

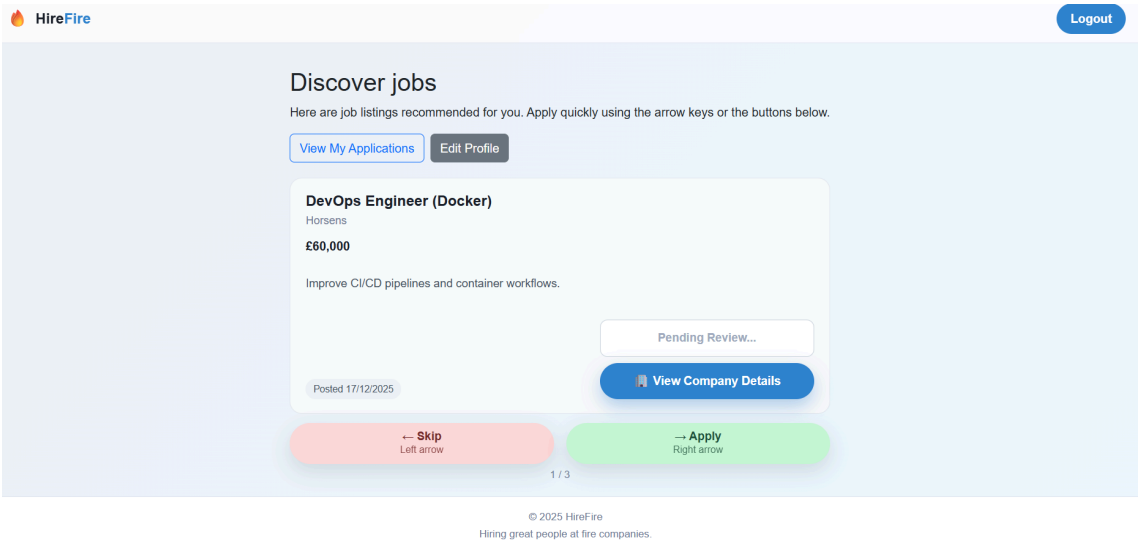


Figure 10 Applicant Main Interface

The interface adapts to the user's role after logging in. A swipe-based dashboard helps candidates interact primarily, shows suggested job offers and allows interaction using a keyboard and buttons to apply or decline positions. Additional views allow candidates to manage profiles, which include detailed monitoring of skill levels that can be adjusted, as well as reviewing submitted applications and begin communicating with recruiters when matches are found. Dashboards allow company representatives to monitor recruiters, profiles, and the company. The interactive interface allows recruiters to edit and modify job postings, identify required and optional skills, evaluate candidates, and track suitable candidates.

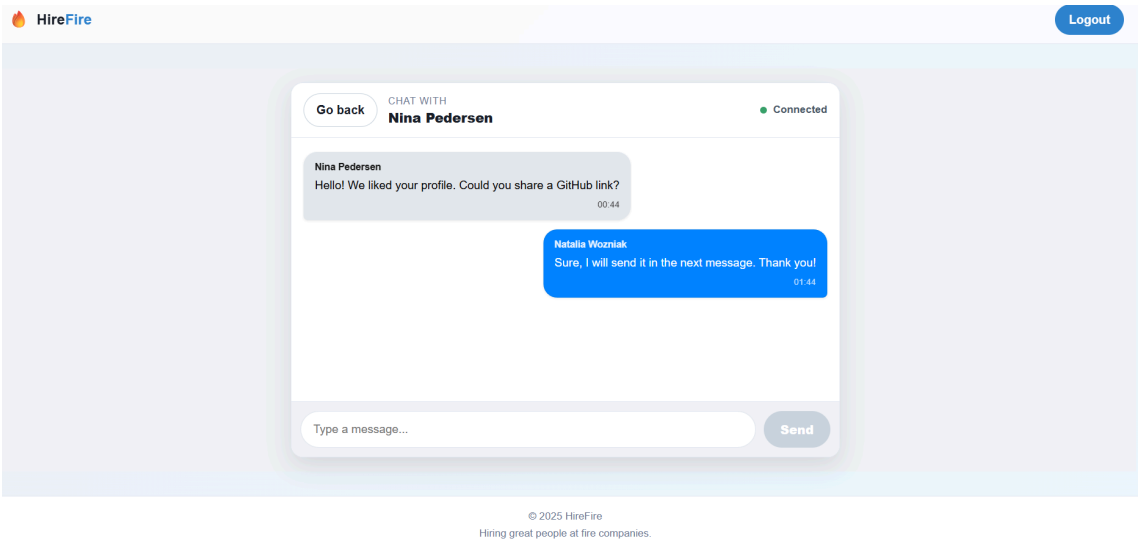


Figure 11 Applicant - Recruiter Chat

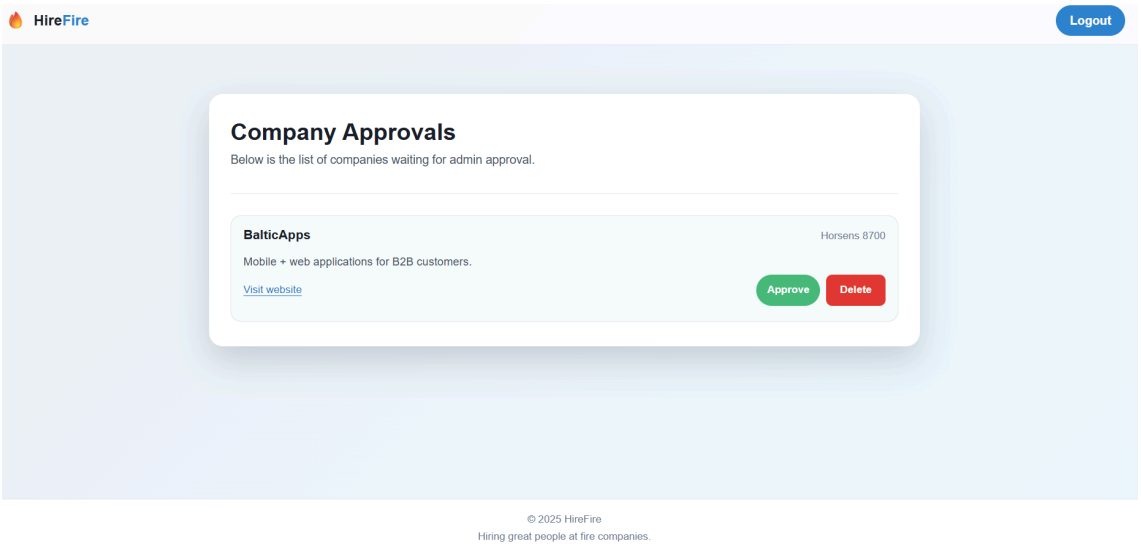
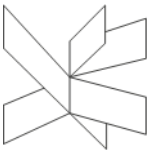


Figure 12 Administrator Page

A simple yet powerful dashboard is useful for administrators to manage access, ensure system integrity, and authorize or deny access to company information. A dedicated chat interface, derived from SignalR, allows roles to interact in real time with each other. This interface not only allows for automatic message updates and scrolling but also allows for structured discussions about specific programs. To ensure each view is



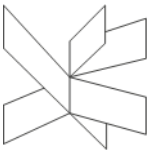
unique while maintaining overall consistency, a combination of individual component styles and CSS styles is used across the board. To create a modular, role-based user interface, it is essential to choose a design that aligns with the system's functional requirements. This design ensures consistent navigation, clarity, and ease of use.

4.5 Design Pattern choices

The system makes use of several software design patterns to maintain a clean structure and reduce coupling between components. The distributed nature of the application benefits significantly from predictable and consistent communication patterns. The following section outlines the key design patterns applied throughout the project.

4.5.1 Data Transfer Object Pattern

DTOs are used extensively across the entire system to encapsulate the data exchanged between layers. The Client communicates with the Logic Tier using DTOs serialized over REST, while the Logic Tier communicates with the Data Tier through Proto-generated DTOs via gRPC. The pattern ensures that only the required data is transferred between layers and decouples the internal database schema from the external API contract, providing security, flexibility, and maintainability (*Baeldung, 2025. The DTO Pattern (Data Transfer Object)*).



4.5.2 Dependency Injection

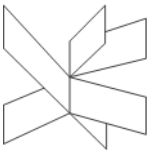
Both Logic and Data Tier applications heavily utilize Dependency Injection, where business logic is encapsulated in Service classes injected into Controllers or Hubs. Date Tier abstracts database operations using Spring Data Repositories injected into Services by defining interfaces that extend JpaRepository, allowing the framework to automatically generate SQL queries from method names at runtime, thereby eliminating redundant code and decoupling the business logic from the underlying database implementation. This architecture ensures loose coupling and high testability, allowing dependencies like database repositories to be easily swapped with mocks during unit testing (*GeeksforGeeks, 2025 Dependency Injection(DI) Design Pattern*).

4.5.3 Proxy Pattern

The Client Tier uses the Proxy Pattern to communicate with the Logic Tier. Instead of making raw HttpClient calls inside Razor components, we wrap these calls in the services (e. G. *HttpApplicantService*). This abstraction benefits the UI components by hiding endpoints and serialization logic, allowing them to simply call methods like *_service.GetByJobAsync(id)* while the service handles HTTP complexity, error parsing, and DTO mapping (*GeeksforGeeks, 2025 Proxy Design Pattern*).

4.5.4 Observer Pattern

For the real-time chat feature, we implemented the Observer Pattern using SignalR. The Logic Tier's Hub acts as the subject, maintaining a list of connected clients (Observers) grouped by their *ApplicationId*. The Blazor client functions as the observer, subscribing to events like *ReceiveMessage*. When the Hub receives a new message, it automatically notifies only the relevant observers in that specific chat room, enabling instant communication without the need for the client to constantly request updates from the server (*Microsoft (2025). Observer design pattern Microsoft Learn*).



4.6 Class diagram (simplified)

The graphical representation of the system's structure is presented in the simplified class diagram below. This diagram illustrates the main components within each tier and their relationships. To improve readability and focus on architectural decisions, helper classes, specific DTOs, and minor utility methods have been omitted as well as their properties and methods.

The Client, built using Blazor, acts as the interface for the user. It is organized into Pages and Services.

- **Pages**: The .razor files (e.g., Login.razor) serve as the View, handling user input and rendering the UI.
- **Services**: Classes such as HttpApplicantService and HttpCompanyService act as the bridge between the UI and the backend. They are responsible for sending HTTP REST requests to the Logic Tier.
- **Authentication**: The AuthProvider class extends AuthenticationStateProvider, managing the user's login state to control access to different views.

Logic Tier, implemented in .NET, serves as the system's brain. It processes client requests and enforces business rules before data is persisted.

- **Controllers**: Classes like ApplicantController and JobListingController expose RESTful API endpoints. They receive HTTP requests from the Client and return HTTP responses.
- **Services**: The controllers delegate complex processing to the Service layer (e.g., ApplicantService). These services contain the core business logic and do not access the database directly. Instead, they communicate with the Data Tier using gRPC clients, ensuring strict separation from the physical storage.

Data Tier, developed with Java and Spring Boot, is responsible for data persistence and integrity.

- **Services**: Classes extending the generated gRPC bases (e.g., ApplicantServiceGrpc) handle incoming remote procedure calls from the Logic Tier.
- **Repositories**: The system utilizes the Repository pattern (e.g., ApplicantRepository, JobListingRepository) which extends JpaRepository. These interfaces abstract the underlying SQL queries.
- **Entities**: Classes such as Applicant, Company, and JobListing represent the database tables. They are mapped using JPA/Hibernate annotations, allowing the application to manipulate data as Java objects rather than raw SQL.

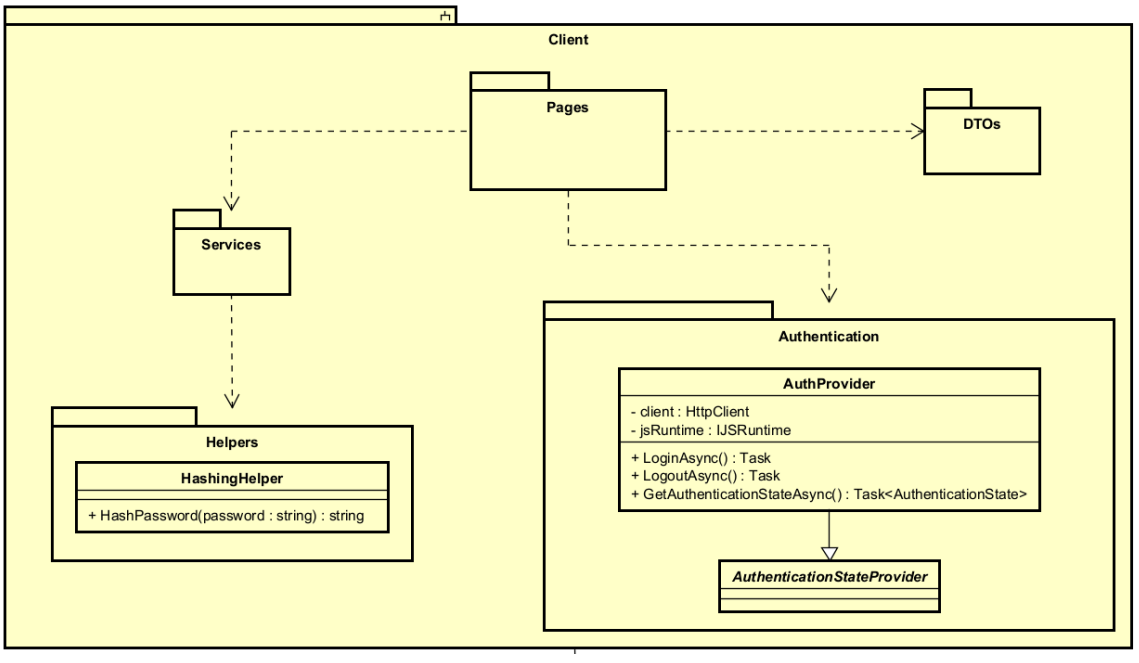


Figure 13 Class diagram for the Client

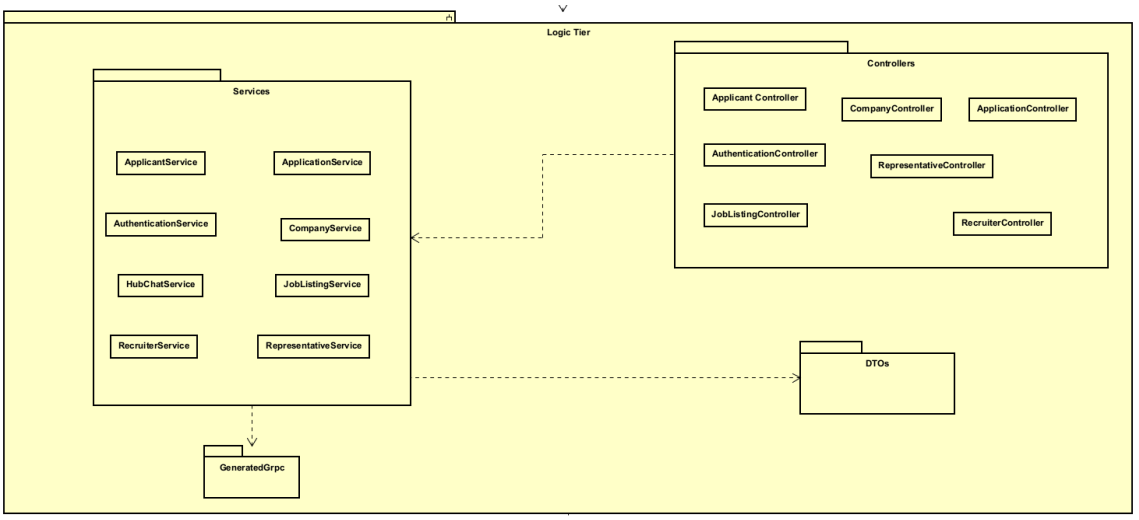


Figure 14 Class diagram for the Logic Tier

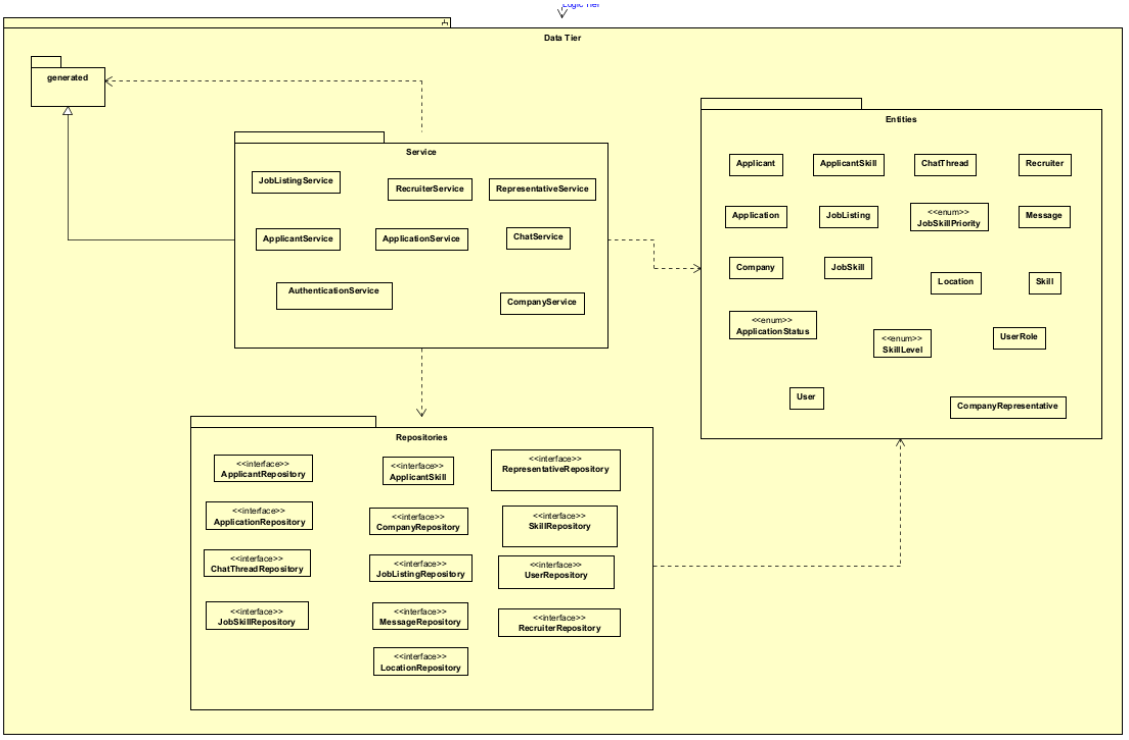


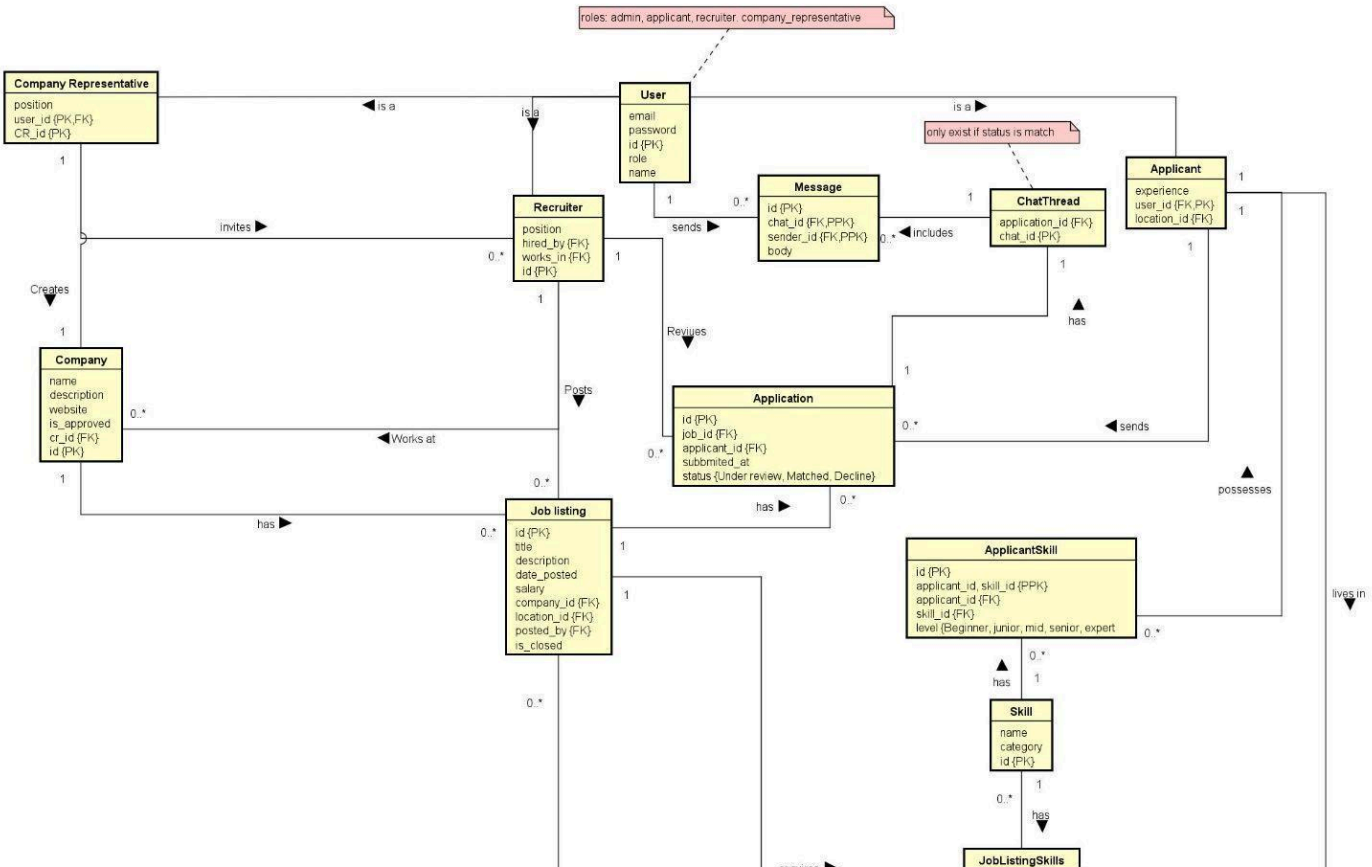
Figure 15 Class diagram for the Data Tier

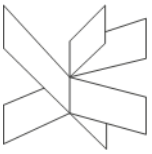
4.7 Database

This section shows how the database for HireFire is designed. It stores key information about companies and their representatives, recruiters, applicants, job listings, locations, and messages.

The Figure 16 GRD diagram below shows the database structure, what is stored in each entity, and how the entities are connected using primary and foreign keys. The company stores basic company details. Company Representative is linked to a company and can create recruiter accounts. Recruiter can post Job Listing records, and each job listing is linked to a Location.

Applicant stores profile details such as skills and experience, and is linked to a Location as well. Applicants and job listings can be connected through matches, which represents the many-to-many relationship between them. Message stores the chat history with a timestamp and message content, so communication between recruiters and applicants can be saved and shown in the system.





The database aims to be organized using 3rd Normal Form (3NF) to reduce data repetition and unnecessary dependencies. Based on the implemented schema, the database fulfills 1NF, 2NF, and 3NF.

First Normal Form (1NF)

Each column stores a single value and there are no repeating groups in one table. (Connolly, T. M., & Begg, C. E. 2015)

Example:

users store atomic values like email, password, role, and name.

Skills are not stored as a list in applicant or job_listing. Instead, they are stored in separate tables (applicant_skill and job_listing_skills), where one row represents one skill connection.

Second Normal Form (2NF)

The database is in 2NF because tables are already in 1NF and non-key attributes depend on the whole key. (Connolly, T. M., & Begg, C. E. 2015)

Example:

Most tables use a single-column primary key (id or user_id), so partial dependency problems do not occur.

In junction tables like application and job_listing_skills, extra attributes (like status or priority) describe the relationship and depend on that relationship.

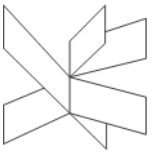
Third Normal Form (3NF)

The database is in 3NF because it is in 1NF and 2NF, and it avoids transitive dependencies. (Connolly, T. M., & Begg, C. E. 2015)

Example:

Location data is stored only in location and referenced using foreign keys, instead of being repeated in multiple tables.

User data is stored in users, and role-specific details are stored in separate tables (recruiter, applicant, company_representative).



Many-to-many relationships are handled using junction tables (applicant_skill, job_listing_skills, application), which reduces repetition.

4.8 Security

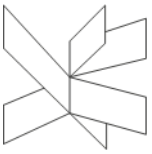
The security architecture for The HireFire system is designed to prioritize authentication, secure data transmission and access control. The primary design objective is to ensure that sensitive user data is protected throughout its lifecycle - from input on the client side to storage in the database.

4.8.1 Access Control & Authentication State

To manage user sessions effectively, the project should include a custom authentication provider. Upon a successful login, the system shall store basic user identity information in the browser's session storage. This client-side state is intended to restrict navigation ensuring that pages requiring authentication remain inaccessible to other users.

4.8.2 Encryption in transit

To mitigate the risk of Man-in-the-Middle attacks, the architecture requires strict Encryption in Transit. The design mandates that all communication between the web client and the Logic Tier must occur over HTTPS. Additionally, the internal communication link between the Logic Tier and the Data Tier is to be secured using gRPC over TLS. This strategy is intended to ensure that sensitive credentials and messages remain encrypted while travelling through the network.



4.8.3 Mitigation of Availability Attacks

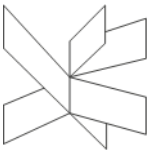
To safeguard system availability against Denial of Service and Brute Force attacks, the system should have control mechanisms. Rate limiting strategies are suggested to restrict the number of requests allowed from a single user or IP addresses within a set timeframe. On the other hand the system might still be vulnerable to Distributed Denial of Service attacks. Additionally, an account lockout policy would help to temporarily disable user access after a predefined number of failed login attempts in order to avoid brute-force attacks.

4.8.4 Secure Password Handling

To prevent Credential Theft, instead of transmitting plain-text passwords to the server the system should hash it by using e.g. SHA-256 algorithm immediately upon user input. These hashes will be compared during the logging process rather than checking the plain password. By ensuring that the passwords are stored in unreadable format, the potential impact of a database breach will be reduced.

The security mechanisms implemented in the system are mainly focused on authentication and basic access control. When a user logs in, credentials are sent from the web client to the Logic Server through the authentication API. The password is hashed using the SHA-256 algorithm in the AuthenticationService before being forwarded via gRPC to the database server. The database layer then verifies the received hash against the stored value. The same process applies for company representatives and recruiters, so the passwords are never stored in plain text, reducing the impact of a potential data breach.

AuthProvider handles the authentication state, that stores basic user information in the browser’s session storage after a successful login. This data is used to restrict access to client-side pages. Pages that require authentication remain inaccessible unless the



user is logged in. Since no server–issued cookie or token is used, authorization is enforced only at the user interface level.

4.9 Version control

GitHub was used for version control throughout the entire project. All of the tiers of the system were put into separate folders on the repository and .gitignore files were created to prevent unnecessary clutter of the repository workspace.

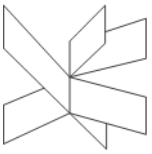
The implementations of tasks generally followed the following pattern when it came to GitHub interactions:

1. Initially the developer would create a new branch based on main
2. All of the updates related to the task were committed and pushed to their branch
3. After the task was finished, the developer would pull the main branch into the task branch. This was to avoid merge conflicts when merging their branch back into main
4. The developer would then open a new pull request and wait for pier review
5. After the pull request was reviewed by another team member, the task branch was merged back into main and the task was considered completed

The preceding conventions allowed the team to avoid frequent merge conflicts and provided a smooth and consistent workflow.

5. Implementation

This section of the report will be dedicated to discussing the roles of various parts of the subsystems, explaining how the tiers interact and going into detail on the security



features. The descriptions will be backed using code snippets to better clarify the inner workings of the application.

5.1 Structure of Application Tiers

This section is dedicated to describing the structure of each of the system's tiers.

5.1.1 Client structure

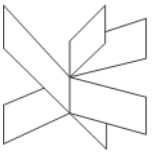
The client side of HireFire consists of the following packages:

- **Authentication** - classes responsible for managing the authentication state of the user
- **Components** - components used for displaying the UI
- **DTOs** - data transfer objects grouped into entity subfolders i.e. AddApplicantDto.cs used to make requests for adding applicants is stored under DTOs/Applicant
- **Helpers** - helper classes
- **Services** - service interfaces and their implementations responsible for communicating with the Logic Server

5.1.2 Logic Server structure

Logic Server packages and their responsibilities:

- **Controllers** - classes responsible for exposing endpoints for use by the Client
- **DTOs** - data transfer objects grouped into entity subfolders i.e. CreateApplicantDto.cs used to interpret requests for adding applicants is stored under DTOs/Applicant
- **GeneratedGrpc** - helper classes autogenerated based on .proto files



- **Proto** - .proto files defining the service methods and return/request message formats used for communication with the Database Server through gRPC
- **Services** - classes responsible for communicating with the Database Server through gRPC
- **Services/Helper** - helper classes for the Services

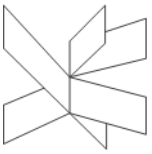
5.1.3 Database Server structure

Most of the important classes in the Database Server are located in the com.example.databaseserver package, found in the src/main/java folder. They are divided into the following packages:

- **Entities** - entity classes including annotations that allow for mapping to the database
- **Repositories** - repository interfaces extending JpaRepository, used to access the various database tables
- **Services** - service classes containing the methods intended to be exposed to the Logic Server using gRPC

The com.example.databaseserver package additionally contains the DatabaseServerApplication class which is used to start the server.

It is also important to mention that the .proto classes defining the request/response message formats and methods exposed to the Logic Server using gRPC are stored under src/main/proto.

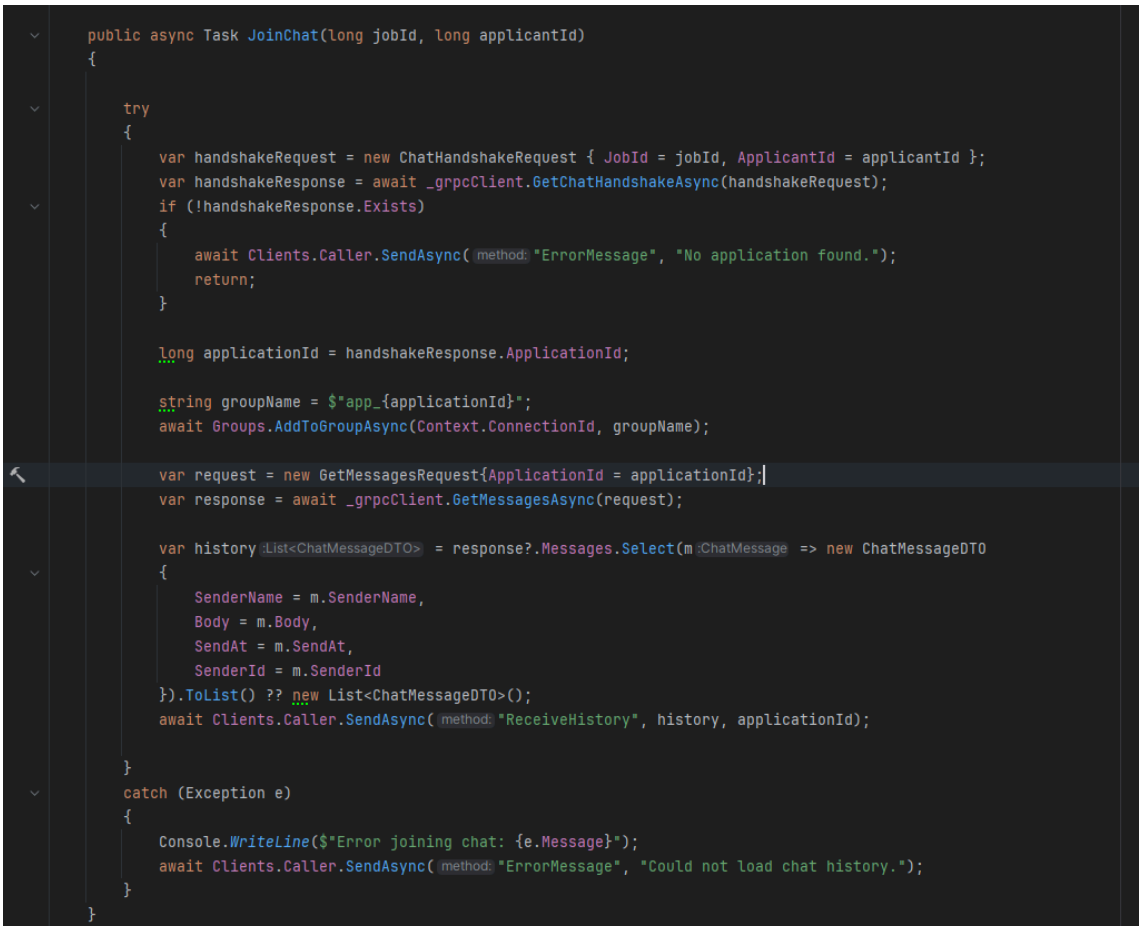


5.2 Real-Time communication

To enable instant communication between applicants and recruiters, our system implements websockets using SignalR. This allows the server to push messages to connected clients instantly.

5.2.1 The Chat Hub

The core of the implementation is the HubService. This class acts as a central traffic controller. Instead of broadcasting every message to every user (which would be a privacy breach), the Hub uses a “Group” mechanism.
When a user opens a chat, they are automatically added to a unique group named after the specific Job Application ID. This ensures that messages are strictly isolated between the two relevant participants.

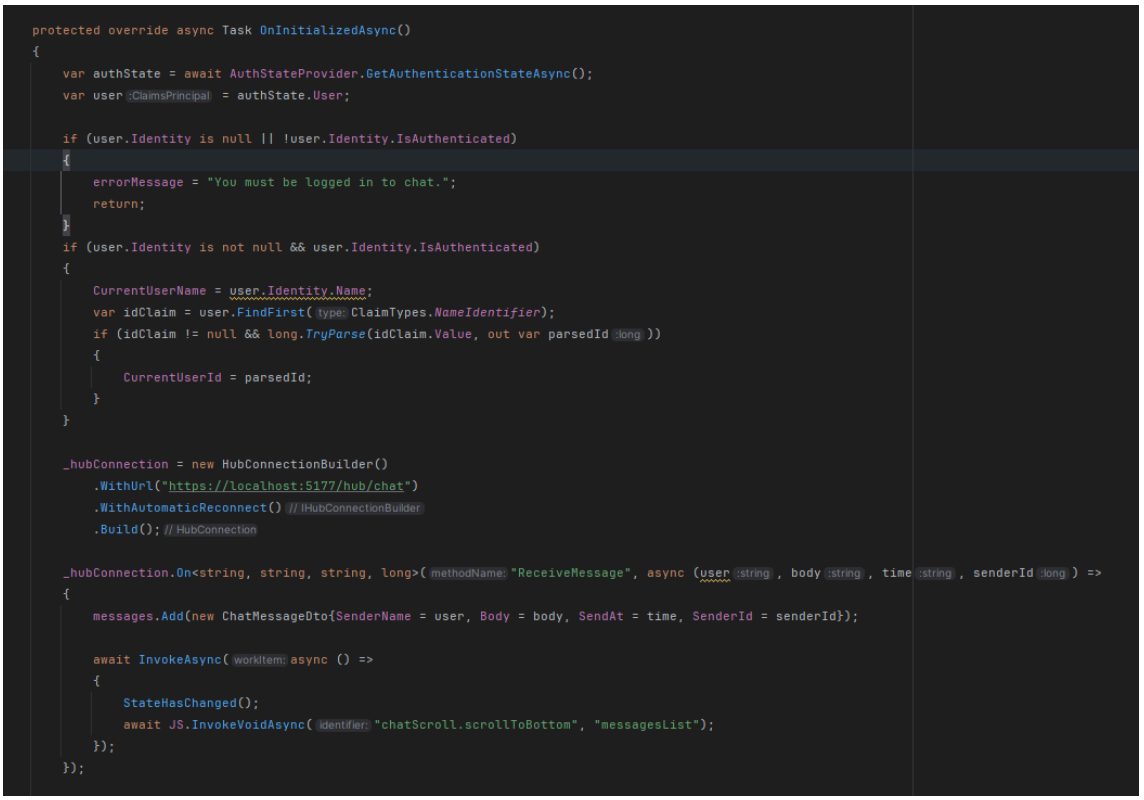


```
public async Task JoinChat(long jobId, long applicantId)
{

    try
    {
        var handshakeRequest = new ChatHandshakeRequest { JobId = jobId, ApplicantId = applicantId };
        var handshakeResponse = await _grpcClient.GetChatHandshakeAsync(handshakeRequest);
        if (!handshakeResponse.Exists)
        {
            await Clients.Caller.SendAsync( method: "ErrorMessage", "No application found.");
            return;
        }

        long applicationId = handshakeResponse.ApplicationId;

        string groupName = $"app_{applicationId}";
        await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

        var request = new GetMessagesRequest{ApplicationId = applicationId};
        var response = await _grpcClient.GetMessagesAsync(request);

        var history :List<ChatMessageDTO> = response?.Messages.Select(m :ChatMessage => new ChatMessageDTO
        {
            SenderName = m.SenderName,
            Body = m.Body,
            SendAt = m.SendAt,
            SenderId = m.SenderId
        }).ToList() ?? new List<ChatMessageDTO>();
        await Clients.Caller.SendAsync( method: "ReceiveHistory", history, applicationId);

    }
    catch (Exception e)
    {
        Console.WriteLine($"Error joining chat: {e.Message}");
        await Clients.Caller.SendAsync( method: "ErrorMessage", "Could not load chat history.");
    }
}
```

Figure 17 Chat Hub on the Logic Tier (JoinChat method)

5.2.2 Client Integration

To avoid constant server requests, the client establishes a continuous connection when the chat component loads. It sets up a 'listener' that waits specifically for the 'ReceiveMessage' signal. When the server detects a new message, it triggers this signal on the client, causing the browser to update the chat window immediately without requiring a page reload.

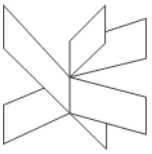


```
protected override async Task OnInitializedAsync()
{
    var authState = await AuthStateProvider.GetAuthenticationStateAsync();
    var user :ClaimsPrincipal = authState.User;

    if (user.Identity is null || !user.Identity.IsAuthenticated)
    {
        errorMessage = "You must be logged in to chat.";
        return;
    }
    if (user.Identity is not null && user.Identity.IsAuthenticated)
    {
        CurrentUserName = user.Identity.Name;
        var idClaim = user.FindFirst( type: ClaimTypes.NameIdentifier);
        if (idClaim != null && long.TryParse(idClaim.Value, out var parsedId :long ))
        {
            CurrentUserId = parsedId;
        }
    }

    _hubConnection = new HubConnectionBuilder()
        .WithUrl("https://localhost:5177/hub/chat")
        .WithAutomaticReconnect() // IHubConnectionBuilder
        .Build(); // HubConnection

    _hubConnection.On<string, string, string, long>( methodName: "ReceiveMessage", async (user :string , body :string , time :string , senderId :long ) =>
    {
        messages.Add(new ChatMessageDto{SenderName = user, Body = body, SendAt = time, SenderId = senderId});

        await InvokeAsync( workItem: async () =>
        {
            StateHasChanged();
            await JS.InvokeVoidAsync( identifier: "chatScroll.scrollToBottom", "messagesList");
        });
    });
```

Figure 18 Chat.razor page (OnInitializedAsync method) p1

```
    _hubConnection.On<List<ChatMessageDto>, long>( methodName: "ReceiveHistory", async (history :List<ChatMessageDto> , appId :long ) =>
    {
        activeApplicationId = appId;
        messages.AddRange(history);
        await InvokeAsync( workItem: async () =>
        {
            StateHasChanged();
            await JS.InvokeVoidAsync( identifier: "chatScroll.scrollToBottom", "messagesList");

        });
    });
    _hubConnection.On<String>( methodName: "ErrorMessage", (reason :string ) =>
    {
        errorMessage = reason;
        InvokeAsync(StateHasChanged);
    });

    await _hubConnection.StartAsync();
    await _hubConnection.SendAsync( methodName: "JoinChat", JobId, ApplicantId);
}
```

Figure 19 Chat.razor (OnInitializedAsync method) p2



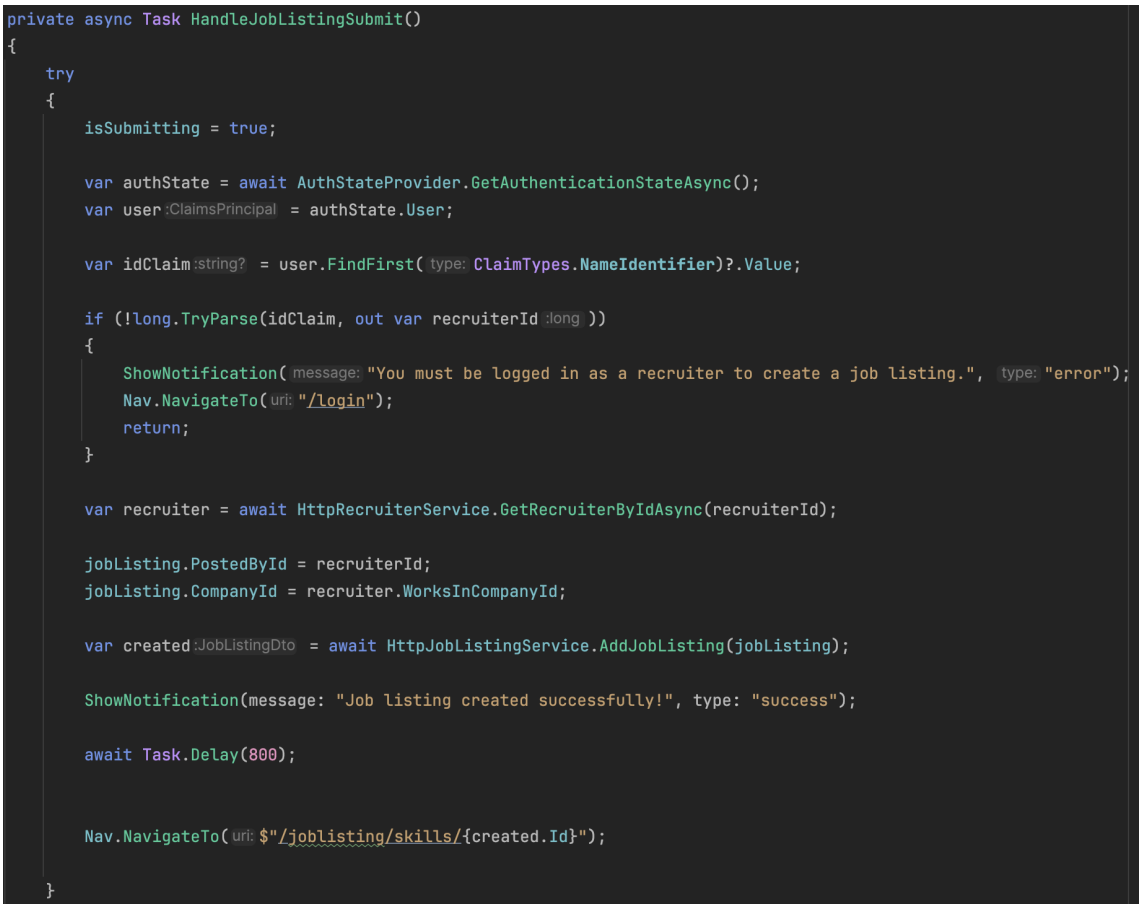
5.3 End-to-End Action Flow

This section of the report will be dedicated to explaining how the action of adding a job listing travels through all tiers of the system.

For the purpose of this demonstration, it is assumed that the user is logged in as a recruiter and already filled in the form presented on the /create-job-listing page with appropriate information.

5.3.1 Client - `AddJobListing.razor`

After pressing the “Create Job Listing” button on the /create-job-listing page the method shown below is triggered.



```
private async Task HandleJobListingSubmit()
{
    try
    {
        isSubmitting = true;

        var authState = await AuthStateProvider.GetAuthenticationStateAsync();
        var user :ClaimsPrincipal  = authState.User;

        var idClaim :string?  = user.FindFirst( type: ClaimTypes.NameIdentifier)?.Value;

        if (!long.TryParse(idClaim, out var recruiterId :long ))
        {
            ShowNotification( message: "You must be logged in as a recruiter to create a job listing.",  type: "error");
            Nav.NavigateTo( uri: "/login");
            return;
        }

        var recruiter = await HttpRecruiterService.GetRecruiterByIdAsync(recruiterId);

        jobListing.PostedById = recruiterId;
        jobListing.CompanyId = recruiter.WorksInCompanyId;

        var created :JobListingDto  = await HttpJobListingService.AddJobListing(jobListing);

        ShowNotification(message: "Job listing created successfully!", type: "success");

        await Task.Delay(800);


        Nav.NavigateTo( uri: $"/joblisting/skills/{created.Id}");

    }
```

Figure 20 AddJobListing.razor (HandleJobListingSubmit method)

It gets the Id of the currently logged in user from the `AuthenticationStateProvider` and proceeds to retrieve the necessary recruiter information from the Logic Server using the `HttpRecruiterService`. It then proceeds to call the `HttpJobListingService.AddJobListing()` method, passing in the necessary arguments based on the form data provided.

5.3.2 Client - `HttpJobListingService.cs`

The `AddJobListing()` method uses the `HttpClient` injected through the constructor to send a POST request to the api/joblisting endpoint of the Logic Server and awaits a response.

```
public async Task<JobListingDto> AddJobListing(CreateJobListingDto request)
{
    HttpResponseMessage httpResponse = await client.PostAsJsonAsync( requestUri: "api/joblisting", request);
    string response = await httpResponse.Content.ReadAsStringAsync();

    if (!httpResponse.IsSuccessStatusCode)
        throw new Exception(response);

    return JsonSerializer.Deserialize<JobListingDto>(
        response,
        new JsonSerializerOptions { PropertyNameCaseInsensitive = true }
    )!;
}
```

Figure 21 HttpJobListingService.cs (AddJobListing method)

5.3.3 Logic Server - `JobListingController.cs`

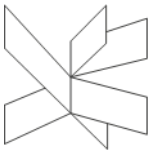
The request is received in the `CreateJobListing()` as it is the method exposed at the api/joblisting endpoint for POST requests.

It verifies the request, calls the `JobListingService.CreateJobListingAsync()` method, passing in the received DTO and awaits the response.

Figure 22 JobListingController.cs (CreateJobListing method)

5.3.4 Logic Server - `JobListingService.cs`

The `CreateJobListingAsync()` method creates a new secure gRPC channel leveraging the `GrpcChannelHelper` class. The channel is used to create a `JobListingServiceClient`. A `CreateJobListingRequest` is then constructed, the class is autogenerated based on the message format defined in `job.proto`. The newly created request is sent to the Database Server using the `JobListingServiceClient` and the response is awaited.

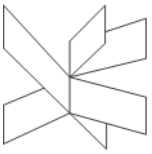


```
public async Task<JobListingDto> CreateJobListingAsync(CreateJobListingDto dto)
{
    using var channel = GrpcChannelHelper.CreateSecureChannel(_grpcAddress);
    var client = new HireFire.Grpc.JobListingService.JobListingServiceClient(channel);

    var request = new CreateJobListingRequest
    {
        Title       = dto.Title,
        Description = dto.Description ?? string.Empty,
        Salary      = dto.Salary.HasValue
            ? dto.Salary.Value.ToString( format: "0.##", CultureInfo.InvariantCulture)
            : string.Empty,
        CompanyId   = dto.CompanyId,
        City          = dto.City,
        Postcode      = dto.Postcode,
        Address       = dto.Address,
        PostedById  = dto.PostedById,
        IsClosed    = false
    };

    var reply :JobListingResponse?  = await client.CreateJobListingAsync(request);

    return MapToDto(reply);
}
```

Figure 23 AddJobListing.razor (HandleJobListingSubmit method)

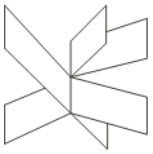
5.3.5 Database Server - `JobListingService.java`

The `createJobListing()` evoked by the Logic Server utilizes the `CompanyRepository` and `RecruiterRepository` to verify that both the company and recruiter associated with the job listing exist. It then saves the location associated with the new job listing as well as the listing itself using the appropriate JPA repositories. It creates the response based on the message format defined in `job.proto` using the `mapToResponse()` helper method and returns it by calling `.onNext()` and `onCompleted()` on the `StreamObserver`.



```java
@Override  1 usage  baczek113 +1 *
@Transactional
public void createJobListing(CreateJobListingRequest request,
                             StreamObserver<JobListingResponse> responseObserver) {
    try {
        Company company = companyRepository.findById(request.getCompanyId())
                .orElseThrow(() -> new RuntimeException("Company not found"));
        Recruiter recruiter = recruiterRepository.findById(request.getPostedById())
                .orElseThrow(() -> new RuntimeException("Recruiter not found"));
        Location location = new Location(
                request.getCity(),
                request.getPostcode(),
                request.getAddress()
        );
        Location savedLocation = locationRepository.save(location);
        BigDecimal salary = null;
        if (!request.getSalary().isBlank()) {
            salary = new BigDecimal(request.getSalary());
        }
        OffsetDateTime now = OffsetDateTime.now();
        JobListing job = new JobListing(
                request.getTitle(),
                request.getDescription(),
                now,
                salary,
                company,
                savedLocation,
                recruiter,
                request.getIsClosed()
        );
        JobListing saved = jobListingRepository.save(job);
        JobListingResponse response = mapToResponse(saved);
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
```

Figure 24 JobListingService.java (createJobListing method)



If the input data is valid and no errors occur as the request propagates through the system, each awaited operation completes successfully, and the resulting data is ultimately returned to the frontend.

5.4 Security Implementation

This section covers security mechanisms implemented i n the system, with a focus on secure authentication process and access control. The implementation ensures that sensitive credentials are handled securely before being stored in the database.

5.4.1 Authentication and hashing

When the user logs in, credentials are sent from the web client to the Logic Server via the REST API. Hashing process takes place immediately on the client side so that plain-text passwords never leave the user's browser.
When a user submits the logic form, the *HandleLogin* method intercepts the request. It utilizes a *HashingHelper* to transform the password into a secure hash using the SHA-256 algorithm (*Microsoft. (2023). Password Hashing in ASP.NET Core. Microsoft Learn*). A new request is then created containing hash and the user's email.

```
using ...

namespace WebApp.Helpers;

public class HashingHelper
{
    public static string HashPassword(string password)
    {
        if (string.IsNullOrEmpty(password)) return string.Empty;

        using var sha256 = SHA256.Create();
        var bytes :byte[] = sha256.ComputeHash( buffer: Encoding.UTF8.GetBytes(password));
        return BitConverter.ToString(bytes).Replace("-", "").ToLowerInvariant();
    }
}
```

Figure 25 HashingHelper class

```
@code {
    private LoginRequestDto loginRequest = new();

    private async Task HandleLogin()
    {
        try
        {
            var secureRequest = new LoginRequestDto
            {
                Email = loginRequest.Email,
                Password = HashingHelper.HashPassword(loginRequest.Password)
            };
            await ((AuthProvider)AuthProvider).LoginAsync(secureRequest);


            var authState = await AuthProvider.GetAuthenticationStateAsync();
            var user :ClaimsPrincipal  = authState.User;

            var role :string?  = user.FindFirst( type: ClaimTypes.Role)?.Value;


            var target :string  = role switch
            {
                "applicant"             => "/applicant/dashboard",
                "company_representative" => "/company-representative/dashboard",
                "admin"                 => "/admin",
                "recruiter"             => "/recruiter/dashboard",
                _                       => "/"
            };

            navigationManager.NavigateTo(target);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
```

Figure 26 HandleLogin method

5.4.2 Session Storage & State Management

The system uses a custom *AuthProvider* to manage the authentication state. Upon a successful login response from the API, the provider serializes the user's identity data and stores it in the browser's *sessionStorage.* This ensures the user remains logged in during their session

```
public async Task LoginAsync(LoginRequestDto request)
{
    HttpResponseMessage httpResponse = await client.PostAsJsonAsync( requestUri: "api/authentication", request);
    string response = await httpResponse.Content.ReadAsStringAsync();
    if (!httpResponse.IsSuccessStatusCode)
    {
        throw new Exception(response);
    }

    LoginResponseDto loginResponseDto = JsonSerializer.Deserialize<LoginResponseDto>(response, new JsonSerializerOptions { PropertyNameCaseInsensitive = true })!;
    if (!loginResponseDto.Success)
    {
        throw new Exception("Incorrect credentials");
    }

    string serialisedData = JsonSerializer.Serialize(loginResponseDto);
    await jsRuntime.InvokeVoidAsync( identifier: "sessionStorage.setItem", "currentUser", serialisedData);

    List<Claim> claims = new List<Claim>()
    {
        new( type: ClaimTypes.Name, loginResponseDto.Name),
        new( type: ClaimTypes.Role, loginResponseDto.Role),
        new( type: "Email", loginResponseDto.Email),
        new( type: ClaimTypes.NameIdentifier, loginResponseDto.Id.ToString())
    };

    ClaimsIdentity identity = new ClaimsIdentity(claims,  authenticationType: "apiauth");
    ClaimsPrincipal claimsPrincipal = new ClaimsPrincipal(identity);
    NotifyAuthenticationStateChanged( Task.FromResult(new AuthenticationState(claimsPrincipal)) );
}
```

Figure 27 LoginAsync method in the AuthProvider class

5.4.3 Access Control

Access control is enforced at the Client tier using Blazor's *<AuthorizeView>* component. This allows the system to selectively render content based on the user's role. Pages that require specific privileges are wrapped in these tags to prevent unauthorized navigation.

```
<AuthorizeView Roles="company_representative">
    <Authorized Context="_">
        <div class="register-shell">
            <div class="register-card">
                <div class="register-header">
                    <h2>Create Recruiter Account</h2>
                    <p>
                        Add a recruiter who will be able to post jobs and talk
                        to applicants on behalf of this company.
                    </p>
                </div>

                <div class="register-form-area">
                    <EditForm Model="@recruiterProfile" OnValidSubmit="HandleRecruiterSubmit">
                        <DataAnnotationsValidator/>

                        <div class="form-grid">
                            <div class="form-group full-width">
                                <label for="r-name">Full name</label>
                                <InputText id="r-name" class="form-control" @bind-Value="recruiterProfile.Name"/>
                                <ValidationMessage For="@(() => recruiterProfile.Name)" class="validation-message"/>
                            </div>

                            <div class="form-group full-width">
                                <label for="r-email">Work email</label>
                                <InputText id="r-email" class="form-control" @bind-Value="recruiterProfile.Email"/>
                                <ValidationMessage For="@(() => recruiterProfile.Email)" class="validation-message"/>
                            </div>
```

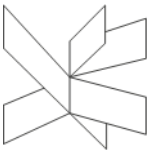
Figure 28 RegisterRecruiter razor page p1

```
                        </div>
                    </div>
                </div>
            </div>
        </Authorized>
        <NotAuthorized>
            <p>You must be logged in as a company representative to create a recruiter.</p>
            <a href="/login">Go to login</a>
        </NotAuthorized>
    </AuthorizeView>
```

Figure 29 RegisterRecruiter razor page p2

5.4.4 Secured Communication Channels1

To prevent Man-in-the-MIddle attacks, the system must implement encrypted communication across all distributed tiers (*Cloudflare. What is TLS (Transport Layer Security)?. Cloudflare Learning Center*).



- **Client ⇔ Logic Tier**: The Blazor Client is configured to communicate with the ASP.NET Core API strictly over HTTPS. The application ensures that all REST API calls and SignalR WebSocket connection are encrypted, protecting sensitive data such as chat messages.
- **Logic Tier ⇔ Date Tier:** The communication between these tiers is secured using gRPC over TLS. In the development environment, the .NET Logic Tier was specifically configured to trust the Java server's self-signed PKCS12 certificate, ensuring that even internal backend traffic remains encrypted.

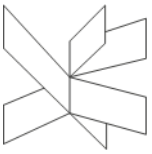
6. Testing

The test section contains Unit testing which is a form of white box (Lakhotia, 2011, #) testing, and Test Cases based on the Use Case Descriptions. The paragraph also contains a small section focusing on the Integration test.

The black box (Beizer, 1995, #) strategy was chosen to test the Use Case Descriptions. After the implementation, the Test Cases were used to test the correct UI functionality from the different actors point of view. Test Cases contain tables with actions following specific use case description marked color pink and exception test cases marked by color blue.

The white box Unit testing approach was chosen due to the multiple classes used in the Model. The tests were written using the JUnit5 (Gulati & Sharma, 2017, #) library. Classes covered by tests are JobListing and User and are key for the correct functionality of the application. The whole white box strategy was implemented with the standard structure of ZOMB+E. (Contieri, 2020).

The Integration test (Blokdyk, 2019, #) was introduced due to showing connections between layers and classes.



6.1 Test Specifications

The following subsection presents two types of testing. Black-box testing includes example test cases and exception tests such as “View recruiters”, “Create recruiter”, “Remove recruiter”, “Edit recruiter”, “Recruiter–Applicant Communication”. The next subsection focuses on Unit Testing for the Employee class. Code snippets are included to give a clearer understanding of the system’s logic. For additional test cases, the reader is encouraged to consult the Appendix, which contains the full collection of tests and exception scenarios.

6.1.1 Test cases - BlackBox

Test case: Manage Recruiter Accounts - View recruiters - **PASSED**

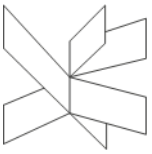
Step	Action	Reaction
1)	Actor opens the Company view and chooses to view recruiters.	The system displays the list of recruiters.

Table 3 Test case: Manage Recruiter Accounts - View recruiters

Test case: Manage Recruiter Accounts - Create recruiter - **PASSED**

Step	Action	Reaction
1)	Actor is viewing recruiters and chooses to create a new Recruiter.	The system displays a create recruiter form.
2)	Actor enters recruiter data (position - cio, name - Jakub , email - j@gm.pl, password - 123).	The system accepts the input.
3)	Actor saves and confirms.	The system creates the recruiter account and returns to the recruiters list with the new recruiter visible.

Table 4 Test case: Manage Recruiter Accounts - Create recruiter



Exception test case: Manage Recruiter Accounts - Create recruiter - Incorrect information - **PASSED**

Step	Action	Reaction
1)	Actor chooses to create a new Recruiter.	The system displays the create recruiter form.
2)	Actor enters incorrect recruiter information (empty password).	The system detects invalid input upon save attempt.
3)	Actor saves and confirms.	The system informs the actor that the information is incorrect and returns to data entry.

Table 5 Exception test case: Manage Recruiter Accounts - Create recruiter - Incorrect information

Exception test case: Manage Recruiter Accounts - Create recruiter - Email already assigned - **PASSED**

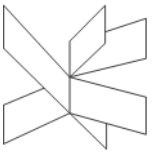
Step	Action	Reaction
1)	Actor chooses to create a new Recruiter.	The system displays the create recruiter form.
2)	Actor enters recruiter data including an email - j@gm.pl that already has an assigned account.	The system detects the conflict upon save attempt.
3)	Actor saves and confirms.	The system informs the actor the email is already assigned and returns to data entry.

Table 6 Exception test case: Manage Recruiter Accounts - Create recruiter - Email already assigned

Test case: Manage Recruiter Accounts - Remove recruiter - **PASSED**

Step	Action	Reaction
1)	Actor is viewing recruiters and selects a recruiter to remove.	The system asks for removal confirmation.
2)	Actor confirms removal.	The system removes the recruiter account.
3)	Actor returns to the recruiters list.	The removed recruiter is no longer visible; changes are visible.

Table 7 Test case: Manage Recruiter Accounts - Remove recruiter



Test case: Manage Recruiter Accounts - Edit recruiter - **PASSED**

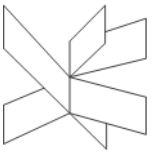
Step	Action	Reaction
1)	Actor is viewing recruiters and selects a recruiter to edit.	The system displays the recruiter's current data in an edit form.
2)	Actor modifies recruiter data. Changes his email on t@gm.pl	The system accepts the changes.
3)	Actor confirms and saves.	The system updates the recruiter account and shows the updated information.

Table 8 Test case: Manage Recruiter Accounts - Edit recruiter

Exception test case: Manage Recruiter Accounts - Edit recruiter - Incorrect information - **PASSED**

Step	Action	Reaction
1)	Actor selects a recruiter to edit.	The system displays the edit form.
2)	Actor modifies recruiter data with incorrect information. (removes email)	The system detects invalid input upon save attempt.
3)	Actor confirms and saves.	The system informs the actor that the information is incorrect and returns to editing.

Table 9 Exception test case: Manage Recruiter Accounts - Edit recruiter - Incorrect information



Exception test case: Manage Recruiter Accounts - Edit recruiter - Email already assigned - **PASSED**

Step	Action	Reaction
1)	Actor selects a recruiter to edit.	The system displays the edit form.
2)	Actor changes the email to one that already has an assigned account - h@gm.pl.	The system detects the conflict upon save attempt.
3)	Actor confirms and saves.	The system informs the actor the email is already assigned and returns to editing (C.2).

Table 10 Exception test case: Manage Recruiter Accounts - Edit recruiter - Email already assigned

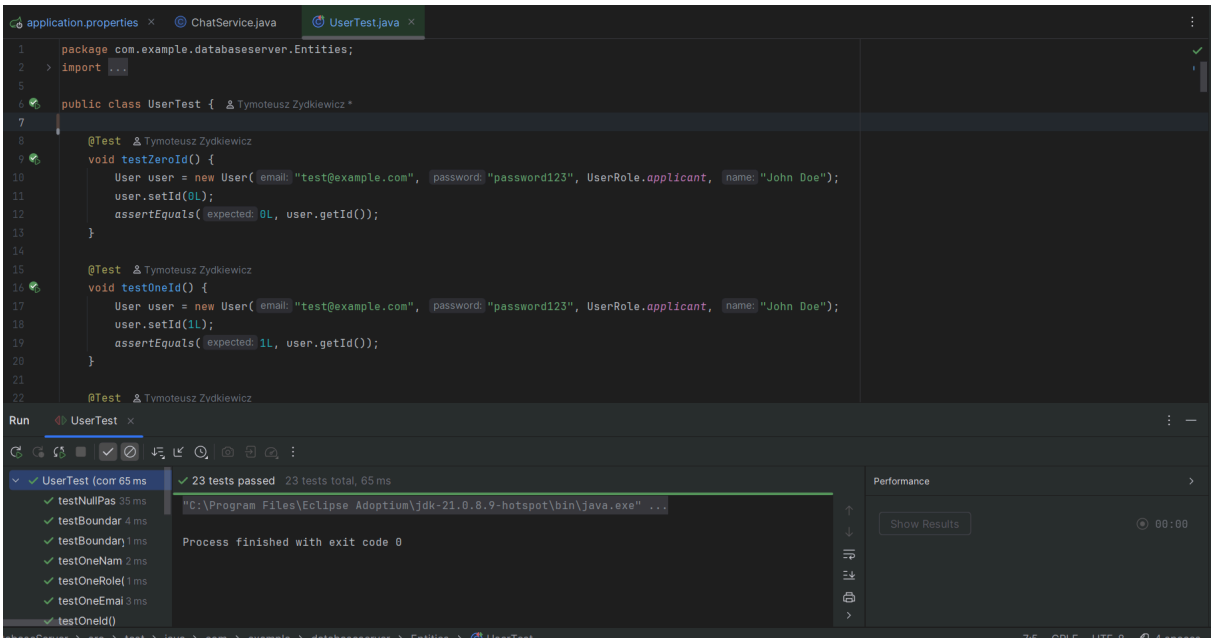
Test case: Recruiter–Applicant Communication - Open conversation and send message - **PASSED**

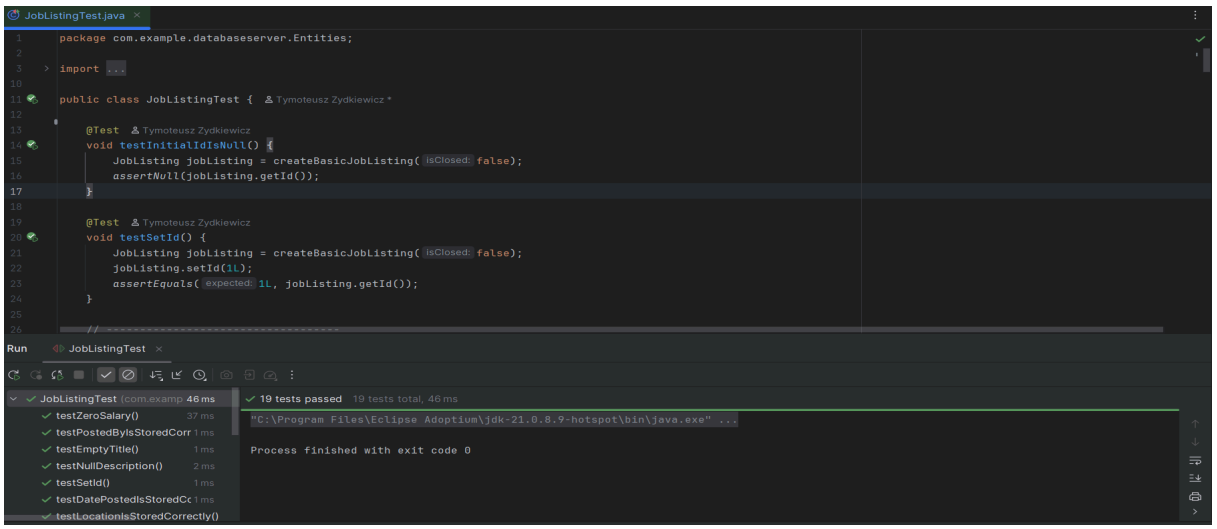
Step	Action	Reaction
1)	Actor opens the Matches/Conversations view.	The system displays matches and available conversations.
2)	Actor selects a match/conversation thread.	The system loads and displays the conversation history.
3)	Actor composes a message and clicks “Send”.	The system persists the message, updates conversation metadata (timestamp/preview/unread counters), and displays the sent message in the thread.
4)	Receiver side.	The system delivers the message to the matched user; the receiver can read it and continue exchanging messages.

Table 11 Test case: Recruiter–Applicant Communication - Open conversation and send message

6.1.2 Unit testing - WhiteBox

The classes User and JobListing were tested through unit testing.





6.1.3 Integration tests

The integration test was created to verify the register company representative functionality across multiple layers of the system:
Client → LogicTier → DataTier → Database

To track the full login flow, System.out.println() printouts were inserted across all involved classes. These printouts help confirm that the signal correctly travels through all layers, ending in either a successful or failed login response.

DataTier:

```
DB-SERVER: Received CreateRepresentative request from LogicServer
DB-SERVER: Mapping request to CompanyRepresentative entity
DB-SERVER: CompanyRepresentative created, sending response back to LogicServer
```

Figure 32 Data Tier

LogicTier:

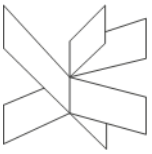
```
LOGIC: HTTP POST /api/representative received - creating company representative
LOGIC: Mapping RepresentativeDto to gRPC request for DatabaseServer
LOGIC: Sending CreateRepresentative gRPC request to DatabaseServer
LOGIC: Received CreateRepresentative gRPC response from DatabaseServer
LOGIC: Company representative created in LogicServer, returning response to Client
```

Figure 33 Logic Tier

Client:

```
CLIENT: Create Company Representative button clicked
CLIENT: Sending CreateCompanyRepresentative request to LogicServer
CLIENT: Received response from LogicServer for CreateCompanyRepresentative
CLIENT: Create Company Representative request finished
```

Figure 34 Client Tier



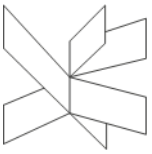
7. Discussion

Thanks to all of the stages of development, the team ended up delivering a secure, reliable, efficient and user-friendly system used for handling job-applications. The finished application fulfills all requirements defined by the team.

Accomplishments

- The system accomplishes the main goal of the project - providing the users with a simpler, more transparent way to handle job applications on both ends of the process.
- The architectural complexity satisfies the initial expectations. The system's division into 3 separate subsystems acts as a way to enforce the separation of concerns. This enables for ease of future modification and extension.
- From the user's perspective the system is pleasant to the eye and intuitive which allows for anyone unfamiliar with the application to use it with ease, regardless of their role.
- The application utilizes an advanced matching algorithm to ensure that the right job offers are suggested to the right applicants. This satisfies both the recruiters and applicants, as the likelihood of their workspace being cluttered by irrelevant applications/job offers is exceptionally low.
- The application's chat works in real-time. This allows for convenient communication between the applicants and recruiters.
- The system is highly secure thanks to the use of HTTPS and password hashing.
- Despite the application's high structural complexity it is highly responsive and no significant delays are present when interacting with the UI. When necessary a loading status is also shown when the Client is awaiting a response from the other application tiers.

Possible improvements:



- Introduction of more advanced input validation to the Logic and Database Servers could possibly improve the systems reliability and security.
- Error handling.

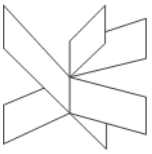
Despite some room for improvement as mentioned above, the team is satisfied with the finished product. Its architecture is highly advanced, it is fully functional and it satisfies the common vision established during Inception.

8. Conclusion and Recommendations

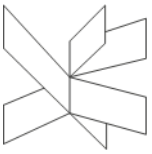
The project resulted in the creation of a unique application used for job searches and talent acquisition. The friendly, intuitive and easy to use interface combined with a smart matching algorithm for suggested job listings makes the application process simple, quick and pleasant for both the hiring and applying party.

Though the project has been concluded successfully and is considered a finished product, the team managed to come up with ideas on potential future improvements:

- Giving applicants the ability to upload CV's could allow the recruiters to gain deeper insight into the pending job applications.
- The addition of Job Listing filtering from the applicant perspective and Application filtering from the recruiter perspective.
- The location scope of the Job Suggestion algorithm could potentially be improved. As of now it looks for job opportunities within the same city as the applicant, this could potentially be changed to work based on the area of the country.
- The addition of industries other than IT could bring in more users. This, depending on the industry, could however require the reshaping of the entire system.



- Extending the administrators capabilities could be extremely useful - at present many issues that could occur are simply unsolvable without directly accessing the database.
- Moving the company management from one company representative account to another could potentially be useful - as of now if the current representative is no longer available, they would most likely have to hand their account over in order for the company to continue operating on the platform.



9. References

Baeldung. (2025). The DTO Pattern (Data Transfer Object). Retrieved from https://www.baeldung.com/java-dto-pattern

GeekforGeek. (2025). Proxy Design Pattern. Retrieved from https://www.geeksforgeeks.org/system-design/proxy-design-pattern/

GeeksforGeeks. (2025). Dependency Injection (DI) Design Pattern. Retrieved from https://www.geeksforgeeks.org/dependency-injection-di-design-pattern/

Microsoft. (2025). *Observer Design Pattern*. Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern

Microsoft. (2023). *Password Hashing in ASP.NET Core*. Microsoft Learn. Retrieved from **https://learn.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/password-hashing**

Cloudflare. (n.d.). *What is TLS (Transport Layer Security)?* Cloudflare Learning Center. Retrieved from https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

Lakhotia, K. (2011). Search-Based Testing. Lap Lambert Academic Publishing GmbH

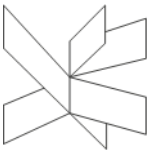
Larman, C. (2004). Applying UML and Patterns. Prentice Hall PTR.

Beizer, B. (1995). Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley.

Blokdyk, G. (2019). Software Testing a Complete Guide - 2020 Edition. Emereo Pty Limited.

Buglione, L., & Abran, A. (2013). 2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement (IWSM MENSURA). IEEE. https://ieeexplore.ieee.org/document/6693222/authors

Connolly, T. M., Connolly, T., & Beg, C. E. (2015). Database Systems: A Practical Approach to Design, Implementation, and Management. Pearson.



Contieri, M. (2020, 11 11). Zombie Testing: One Behavior at a Time. hackernoon.https://hackernoon.com/zombie-testing-one-behavior-at-a-time-9s2m3zjo

Gulati, S., & Sharma, R. (2017). Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5. Apress.

Thomas, D., & Hunt, A. (2020). The Pragmatic Programmer: Your Journey to Mastery. Addison-Wesley.

10. Appendices

Appendix A - Project Description

1. Project Description (PDF DOCX)
2. Group Contract (PDF DOCX)

Appendix B - Analysis

1. Use case descriptions (PDF)
2. Use case diagram (PNG)
3. Sequence diagram ( )
4. Domain model ( )
5. Requirements (PDF file)
6. Activity Diagrams ( )

Appendix C - User guide

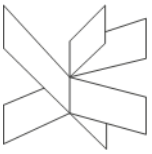
1. User guide (PDF DOCX)

Appendix D - Design

1. Class diagram simplified (PNG ASTA)
2. Sequence Diagram (PNG ASTA)
3. Architecture Overview ( )

Appendix E - Database

1. EER diagram (PNG)
2. GRE diagram (PNG ASTA)
3. SQL code



Appendix F - Test

1. Test cases HireFire(PDF DOCX)

Appendix G - Process Report :

1. The Cultural Map (PNG)
2. Personal Profiles (PDF)
3. Process Report (PDF)

Appendix H - Code

1. Zip folder (Java app, libraries used)

Appendix I - Video

1. HireFire presentation (MP4)

Appendix J - Github Repository

1. GithubRepoLink (txt)