POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering 2
Design Document

Author(s): **Riccardo Piantoni - 10816545**

Matteo Rossi - 10798975

Jacopo Sacramone - 10752157

Academic Year: 2024-2025
Version: 1.1
Release date: 27/01/2025

Contents

1 Introduction

1.1. Scope

Students&Companies (S&C) is a platform that acts as an intermediary system facilitating the internship matching process between students and companies. It allows companies to advertise internships and students to search, receive customized recommendations, and initiate contact.

The platform defines **Recommendation** as the automated process of identifying suitable internship opportunities for students and potential candidates for companies. Following this, a **Contact** represents the phase in which students and companies communicate via the platform to conduct the selection process, including interviews and candidate selection.

The system automates key activities such as generating recommendations using various mechanisms, coordinating interviews, and collecting feedback to improve its algorithms. Additionally, it provides tools to monitor the progress of contacts and internships, manage issues through communication features, and enable universities to supervise the status of internships, ensuring compliance and resolving possible complaints effectively.

1.2. Definitions, Acronyms, Abbreviations

- **System, Platform**: these terms are used interchangeably when referring to the system-to-be-developed.
- **University, Company**: when the terms are referenced as performing an action, it means that the action is executed by a representative acting on behalf of the respective entity.
- **Party**: the term refers to the entities actively involved in the process of applying, participating and managing internships, so both Student and Company; it doesn't include the University.

Acronyms	Definition
RASD	Requirements Analysis & Specification Document
DD	Design Document
API	Application Programming Interface
REST	REpresentational State Transfer
DDD	Domain-Driven Development
DBMS	DataBase Management System
LLM	Large Language Model
JSON	JavaScript Object Notation
URL	Uniform Resource Location
URI	Uniform Resource Identifier
HTTPS	HyperText Transfer Protocol over Secure Socket Layer
SMTP	Simple Mail Transfer Protocol
UI	User Interfaces
CDC	Change Data Capture
QoS	Quality of Service
E2E	End-To-End

Table 1.1: Acronyms used in the document.

Abbreviations	Definition
S&C	Student&Company
UC	Use Case

Table 1.2: Abbreviations used in the document.

1.3. Revision history

Revised on	Version	Description
05/01/2025	1.0	Initial Release of the document
27/01/2025	1.1	Minor fixes of some details

Table 1.3: Revision history

Integrations in Version 1.1

- Updated the image of the Sequence Diagram: *SD13. Manage an Interview*, moving the break condition on top.
- Fixed the layout of the document, removing a blank page that only had one word.

1.4. Reference Documents

- [1] Di Nitto, Rossi, Camilli, *"A.Y. 2024-2025 Software Engineering 2 Requirement Engineering and Design Project"*, 2024.
- [2] ISO/IEC/IEEE 29148:2018, *"Systems and Software Engineering — Life Cycle Processes — Requirements Engineering," International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers*, 2018.
- [3] *Microsoft Learn - Cloud-native data patterns*
- [4] *Microsoft Learn, Data Partitioning guidance*
- [5] *Microsoft Learn, Relational vs. NoSQL data*
- [6] *Microsoft Learn, Data Consistency primer*
- [7] *Chris Richardson, Microservices: API Gateway Pattern*

1.5. Overview

1.5.1. Document Structure

This document is composed of six sections:

- 1st Chapter - Introduction
- 2nd Chapter - Architectural Design
- 3rd Chapter - User Interface Design
- 4th Chapter - Requirements Traceability
- 5th Chapter - Implementation, Integration and Test Plan
- 6th Chapter - Effort Spent
- 7th Chapter - References

2 Architectural Design

2.1. Overview: High-Level Components and Interactions

In this section, a general description of the architecture of the S&C system is provided. The main architectural styles to be followed are:

- **Client-Server Architecture:** This architecture was chosen for its simplicity and efficiency in managing user interactions and business logic. It is also scalable, as servers can be replicated and distributed to handle increasing user loads.
- **4-Tier Architecture:** The tier architecture and the number of levels are a natural consequence of the subdivision of functionalities across distinct layers, as it will be shown in the Deployment View. This approach supports a clear separation of concerns, ensuring each tier is responsible for a specific aspect of the system's operation. This architecture enhances scalability, modularity and fault tolerance, separating each layer, and allows the system to adapt dynamically to workload demands while maintaining high availability and performance.
- **Microservices:** The system's functionalities have been split into separated microservices, each with its own dedicated database. This decision comes with plenty of benefits: for instance, it allows the achievement of better separation of concerns and decoupling between the functionalities offered, which consequently makes it possible to implement them in parallel and also to accomplish graceful degradation of the system in case of failures. Moreover, it enables different deployment strategies, such as containerization, enhancing system scalability.
- **REST - Representational State Transfer:** The designed microservices expose interfaces based on a REST API to communicate both with each other and the outside world. This decision allows to hide underlying technical aspects of the services since the communication is achieved through a technology-neutral protocol.

2.2. Component View

In this section, the main components of the S&C system are outlined, with an accurate description of what each of them aims to accomplish. Every component is approximately derived from one of the product functions discussed in Section 2.2 of the RASD. Further, components are grouped together by affinity in high-level "macro-categories" to form microservices, obtaining what is usually referred to as "bounded contexts" in Domain-Driven Development (DDD). Components also expose interfaces with methods to be invoked from external entities and communicate with some databases.

2.2.1. Component Diagram

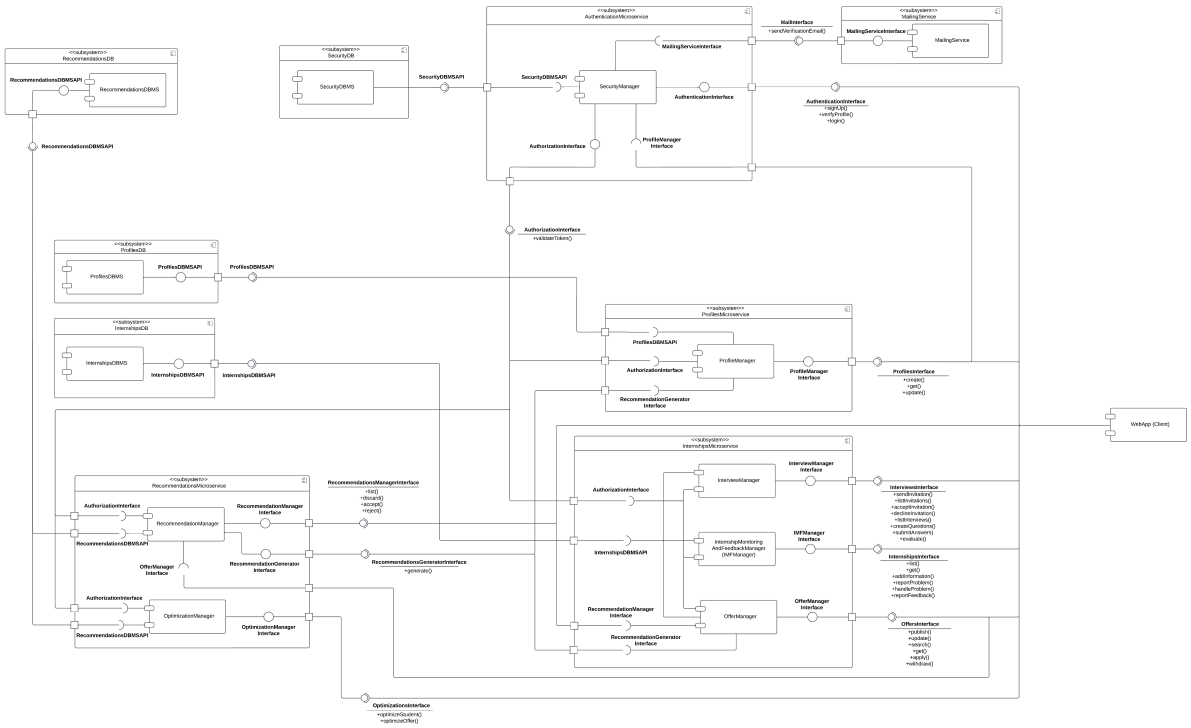

Figure 2.1: Component diagram of the S&C system.

2.2.2. Components Description

Each of the following components shall be thought of as a collection of simpler units that closely interact together to provide the offered functionalities. Specification of these units and how they form the component as a whole is left to the single teams or developers, as it is not binding towards the overall architecture of the system (given that every component must still fulfill the outlined expectations).

1. **SecurityManager:** This component handles user authentication and registration processes for Students, Companies, and Universities. It manages the creation of new accounts by validating the provided information and communicating with the ProfileManager, and it also interacts with the MailingService to verify user identities. Additionally, it manages secure login operations, ensuring that only authenticated users can access the platform, and is used by other components to verify that users have the necessary authorization to execute the incoming requests.

2. **MailingService:** This component handles the email communications needed for the ongoing account verification processes. It implements the mail server functionalities needed to send verification emails to users during the sign-up phase, each containing a link that shall be clicked to complete the registration process.

3. **ProfileManager:** This component handles the operations required to view and update user profiles. For students, sign-ups or updates to relevant details initiate the generation of new recommendations by interacting with the RecommendationManager, aimed either at them or at some Companies, ensuring they both remain aligned with current opportunities.

4. **OfferManager:** This component manages the lifecycle of internship offers within the platform. It provides companies with tools to create, update, withdraw and manage their internship announcements. When creating an offer, the OfferManager automatically extracts the relevant keywords to be applied to the offer from the description provided. The component also integrates advanced filtering mechanisms to allow students to search efficiently for relevant opportunities and view them. Moreover, the OfferManager facilitates interactions between students and companies by managing applications and allowing them to keep the status of offers up-to-date. As for the ProfileManager, this component interacts with the RecommendationManager to generate new recommendations when creating or updating relevant details of offers.

5. **InterviewManager:** This component facilitates and manages the whole interview process between students and companies. It provides tools for scheduling, conducting, monitoring and evaluating interviews. The component ensures easy communication between the parties and handles the creation, delivery and management of interview invitations. Additionally, it records evaluations and updates the status of candidates in the selection process, easing the transition from interviews to final decisions.

6. **InternshipMonitoringAndFeedbackManager (IMFM):** This component manages the monitoring, complaining and feedback processes for internships. It enables students and companies to provide updates and view detailed information about ongoing intern-

ships they are participating in. It also allows parties to report issues encountered during internships, forwarding these reports to the university for resolution. At the conclusion of an internship, the component facilitates the collection of feedback from both students and companies, so that this information can be used in the future to enhance the recommendation algorithms and ensure a continuous improvement of the system.

7. **RecommendationManager:** This component handles the operations needed to generate and manage recommendations. It identifies internship recommendations for students and student recommendations for companies by performing graph analyses on a graph database storing the relationships between students, companies, offers, skills and various feedback expressed through time: specifically, the RecommendationManager detects links between the keywords associated with students' resumes and offers published, and can also perform more complex queries which statistically identify relationships between profiles, preferences and past histories of all users on the platform, by means of collaborative filtering and content-based filtering. Last, but not least, this component also allows both of them to view and act on their received recommendations.

8. **OptimizationManager:** This component manages the operations needed to generate personalized suggestions to improve user content. It relies on a Large Language Model (LLM) that has been properly fine-tuned to identify enhancements to be applied to students' profiles and offers' descriptions to make them more appealing and increase their chances of being selected. The OptimizationManager also takes into account the opportunities (students and offers) currently available on the platform by feeding to the LLM some aggregated data resulting from whole-graph analyses, so that it can provide suggestions that are customized for the current situation.

9. **SecurityDBMS, ProfilesDBMS, InternshipsDBMS:** relational DBMSs that store data respectively about account credentials, profiles information and internship information.

10. **RecommendationsDBMS:** graph DBMS that stores data about the relationships between users, skills, internships, recommendations and useful feedback in order to identify new recommendations by simple skill keyword searching or statistical whole-graph analyses.

2.3. Deployment View

The system is deployed as a distributed microservices architecture leveraging containerization and orchestration to achieve high scalability, resiliency and modularity. Each layer has distinct responsibilities and is deployed across a collection of servers to guarantee replication and fault tolerance.

- **Presentation Layer:** represents the client in a traditional client-server architecture. Users interact with the system via a web application running in their browser.
- **Application Layer:** hosts containerized microservices that provide all business functionalities. The deployment diagram illustrates a possible allocation of these containers across the servers available to this layer. The actual number of microservice instances and their distribution during runtime are dynamically determined by the container orchestration platform based on workload demands. This ensures optimal resource utilization and adaptability to varying request volumes.
- **Service Management Layer:** responsible for managing the orchestration of microservices and auxiliary services. This layer includes the mailing service used during the registration process, the Data Layer Load Balancer, which manages the distribution of requests to database replicas in the Data Layer, and the Replication-Consistency Middleware, which handles data replication and consistency.
- **Data Layer:** manages persistent data storage using replicated databases to ensure fault tolerance and performance optimization. Data replicas are statically allocated, in order to implement strategic business decisions about scalability and geographic allocation manually, and the load balancer in the Service Management Layer ensures requests are distributed evenly across these replicas.

Communication between the components relies on REST APIs (HTTPS), with TCP/IP for lower-level communication, especially with the Service Management Layer and with the Data Layer.

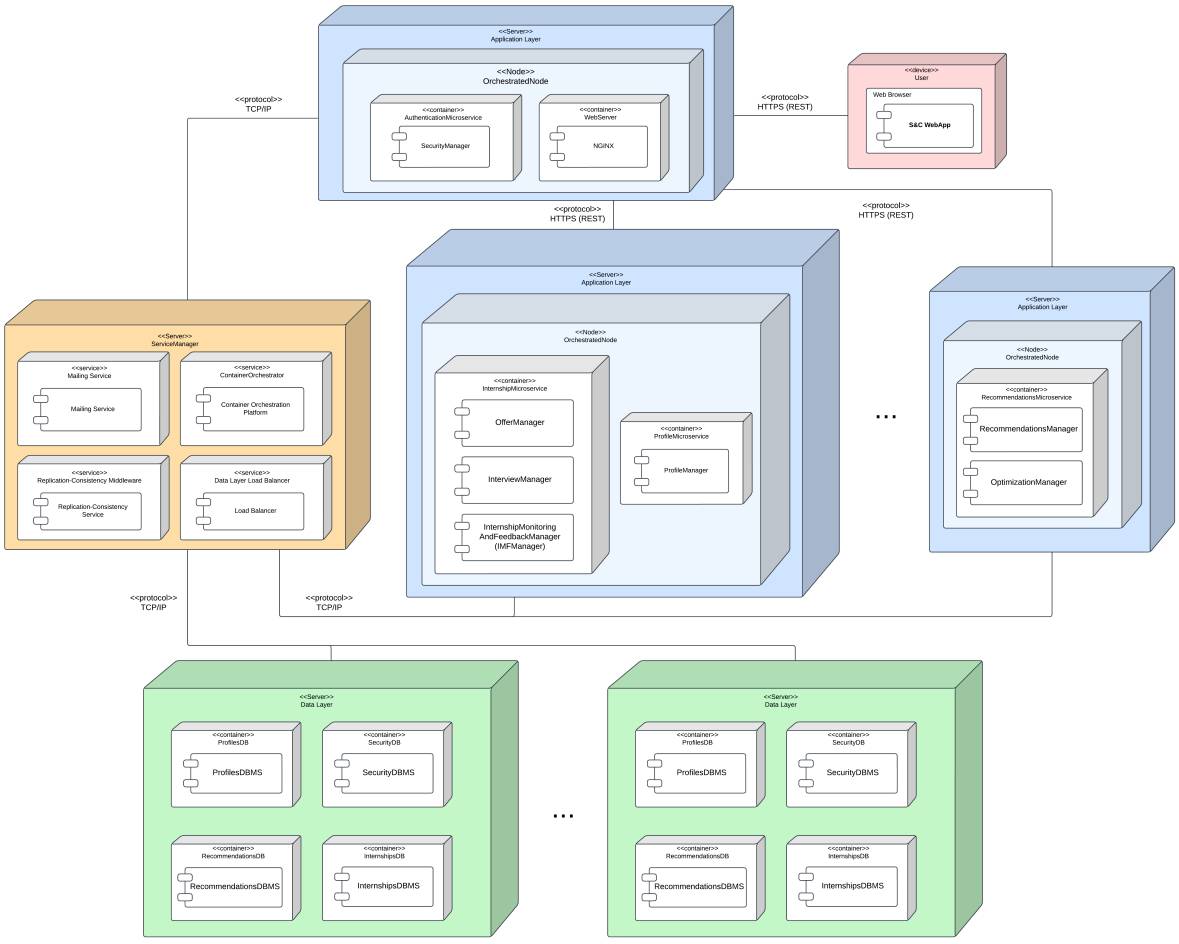

Figure 2.2: UML Deployment Diagram.

2.3.1. Detailed Layer Decisions

Web Server The entry point of the application is NGINX, a web server deployed in a container acting as a reverse proxy, hosting all API endpoints, acting as an API Gateway and forwarding the requests from the user layer to the appropriate microservices in the application layer. The component gatekeeps all traffic to microservices and collaborates with the SecurityManager in order to provide authentication and authorization mechanisms. Internally, it also performs load balancing of the incoming requests and URL rewriting a redirection. It simplifies routing logic and improves the security of the system by exposing a single public endpoint.

Scalability Each microservice and DBMS runs in an isolated container, enabling independent scaling. The container orchestrator dynamically adjusts the number of containers based on resource consumption or incoming traffic, and replication in the Data Layer improves read performance by distributing queries across replicas.

Resiliency The container orchestration platform manages failover and automatically restarts containers in case of crashes. Data replication ensures availability, even in the event of server failure and the Data Layer Load Balancer prevents overloading by evenly distributing requests.

Orchestration Platform Considerations A specific Container Orchestration Technology is not specified, even if ideally it would be preferable to use Kubernetes and deploy the containers with Docker, since they are the most widely adopted platforms and are well known to system administrators, simplifying their use for all the people involved in deploying and managing the infrastructure. In particular, the orchestrator offers many useful functionalities in the context of our architecture, such as automatic scaling, monitoring, networking and server-side service discovery, lightening the burden of managing such aspects. In the server-side discovery method, services are registered with an API server (if Kubernetes is chosen, called Kubernetes API Server), which acts as a central registry for services. Clients then query the API server to discover the available services, which are dynamically deployed and for this reason real addresses may change. The API server responds with a list of available microservices and their corresponding endpoints. When the reverse proxy makes a network request for a service, the Orchestration Platform routes the request to the appropriate endpoint using the information stored. While doing this, the orchestration platform also manages load balancing among the available microservices.

Replication and Data Consistency The Data Layer adopts a Multi-Leader Replication protocol to maintain consistency across replicas, accommodating the geographical distribution of data nodes. This approach enables multiple leaders to handle write operations concurrently, reducing latency and ensuring high availability. In this setup, write operations can be performed on any leader node. Changes are then propagated asynchronously to other leaders and replicas, ensuring eventual consistency across the system. Read operations are distributed across multiple replicas to optimize performance and handle high query volumes effectively. This replication model provides enhanced fault tolerance, as the failure of a single leader does not disrupt write operations. However, potential conflicts from concurrent writes have to be managed using conflict resolution strategies, ensuring data integrity across all replicas.

2.4. Component Interfaces

In this section the component interfaces defined are described, highlighting for each one the expected inputs and outputs.

2.4.1. REST API Endpoints

Each microservice of the system exposes one or more interfaces, which are the resource endpoints of the defined REST API. The Service Orchestrator is responsible for carrying out the Service Discovery process among the physical servers, as it is the one that knows how microservices are deployed on the different machines at every instant, and can thus route the requests aimed at the specific resources to the correct location for the corresponding microservice. The structure of an API request directed to a microservice of the system must adhere to the following structure:

`https://DOMAIN/api/VERSION/INTERFACE/FUNCTION`

In particular:

- The communication protocol must be HTTPS.
- `DOMAIN` is the domain name of the S&C system.
- `/api/` is just syntactic sugar to highlight that the request is aimed at an API resource.
- `VERSION` is the version of the API to be addressed (for compatibility purposes).
- `INTERFACE` is the first-level hierarchy, indicating the specific interface to be addressed (not the actual microservices, as they could expose more than one interface): every interface keeps the same name indicated in the Component View in Section 2.2.1, but without the final "Interface" keyword.
- `FUNCTION` is the API procedure of the specified interface that is being requested.

The following is a list of all the API endpoints, grouped by interface: for each one, the expected input parameters, responses[1] (with eventual important headers), and output values are listed. For additional security, the endpoints must always be invoked through the HTTPS POST method: input parameters and eventual output values are always provided respectively in the body of the request and the response, and both must be in standard JSON format.

[1]Both here and in the Sequence Diagrams in Section 2.5, all **400 Bad Request** responses for requests about nonexisting resources (such as invalid IDs) are omitted for brevity. The same goes for **401 Unauthorized** responses, generated and forwarded by the SecurityManager upon denial resulting from calls to its `/validateToken` endpoint: even if omitted at the moment, such calls shall be present in the implementation code whenever a request that involves access to any protected resource is received and whenever access authorization of the requestor needs to be verified.

/authentication

- `/signUp`
  parameters: { email: String, password: String, accountType: Enum(Student, Company, University) }
  responses:
  - **303 See Other:**
    header: "Location: loginPage",
    body: { cause: Enum(alreadyRegistered, emailSent) }
  - **400 Bad Request:**
    body: { error: Enum(missingFields, invalidPassword) }
- `/verifyProfile`[2]
  parameters: { userID: int }
  responses:
  - **308 Permanent Redirect:**
    header: "Location: updateProfilePage"
    body: { cause: Enum(profileVerified) }
- `/login`
  parameters: { username: String, password: String [3] }
  responses:
  - **308 Permanent Redirect:**
    header: "Location: dashboardPage"
    body: { cause: Enum(successfulLogin), token: String }
  - **308 Permanent Redirect:**
    header: "Location: updateProfilePage"
    body: { cause: Enum(incompleteProfile), token: String }
  - **400 Bad Request:**
    body: { error: Enum(wrongCredentials, unverifiedAccount) }

[2]As an exception, this is the only endpoint to be accessed through a GET request, as it is intended to be invoked by clicking the link in the verification email.
[3]Note that it is not the actual password, but only its hash value.

/mail

- `/sendVerificationEmail`
  parameters: { email: String, confirmationLink: String }
  responses:
  - **204 No Content**

/authorization

- `/validateToken`
  parameters: { token : String }
  responses:
  - **200 OK:**
    body: { authorized: true }
  - **401 Unauthorized:**
    body: { authorized: false }

/profiles

- `/create`
  parameters: { userID: int, user: User }
  responses:
  - **204 No Content**
- `/get`
  parameters: { userID: int }
  responses:
  - **200 OK:**
    body: { user: User }
  - **400 Bad Request:**
    body: { cause: Enum(invalidUser) }
- `/update`
  parameters: { userID: int, user: User }
  responses:
  - **200 OK:**
    body: { popup: Enum(profileUpdated), user: User }

  - **400 Bad Request:**
    body: { cause: Enum(missingFields) }

/offers

- `/publish`
  parameters: { companyID: int, offer: Offer }
  responses:
  - **307 Temporary Redirect:**
    header: "Location: offerPage"
    body: { popup: Enum(offerPublished), offerID: int }
  - **400 Bad Request:**
    body: { cause: Enum(missingFields, nonCompliantInformation) }
- `/update`
  parameters: { companyID: int, offerID: int, offer: Offer }
  responses:
  - **200 OK:**
    body: { popup: Enum(offerUpdated), offer: Offer }
  - **400 Bad Request:**
    body: { cause: Enum(missingFields, nonCompliantInformation) }
- `/search`
  parameters: { filters: [filter: Filter] }
  responses:
  - **200 OK:**
    body: { offersList: [offer: Offer] }
- `/get`
  parameters: { offerID: int }
  responses:
  - **200 OK:**
    body: { offer: Offer }
- `/apply`
  parameters: { studentID: int, offerID: int }
  responses:

  - **200 OK:**
    body: { popup: Enum(applicationSubmitted) }
  - **400 Bad Request:**
    body: { cause: Enum(expiredDeadline) }

- `/withdraw`
  parameters: { companyID: int, offerID: int }
  responses:
  - **308 Permanent Redirect:**
    header: "Location: dashboardPage"
    body: { cause: Enum(successfulWithdrawal) }

/interviews

- `/sendInvitation`
  parameters: { studentID: int, offerID: int, date: DateTime, type: Enum(inPerson, inPlatform), otherDetails: String }
  responses:
  - **200 OK:**
    body: { popup: Enum(invitationSent) }
- `/listInvitations`
  parameters: { userID: int}
  responses:
  - **200 OK:**
    body: [{ offerID: int, offer: Offer, date: DateTime, type: Enum(inPerson, inPlatform), otherDetails: String }]
- `/acceptInvitation`
  parameters: { offerID: int, studentID: int }
  responses:
  - **200 OK:**
    body: { popup: Enum(invitationAccepted) }
- `/declineInvitation`
  parameters: { offerID: int, studentID: int, reason: String }
  responses:

  - **200 OK:**
    body: { popup: Enum(invitationDeclined) }

- `/listInterviews`
  parameters: { userID: int}
  responses:

  - **200 OK:**
    body: [{ interviewID: int, interview: Interview }]

- `/createQuestions`
  parameters: { companyID: int, interviewsIDs: [interviewID: int], questions: [content : String] }
  responses:

  - **200 OK:**
    body: { popup: Enum(questionsCreated), questions: [{ questionIDs: int, question: Question }] }

- `/submitAnswer`
  parameters: { studentID: int, interviewID: int, questionID: int, answer: String }
  responses:

  - **200 OK:**
    body: { popup: Enum(answerSubmitted), answer: Answer }

- `/evaluate`
  parameters: { studentID: int, offerID: int, status: Enum(Selected, Rejected), feedback: String }
  responses:

  - **200 OK:**
    body: { popup: Enum(interviewEvaluated) }

/internships

- `/list`
  parameters: { userID: int }
  responses:

  - **200 OK:**
    body: [{ internshipID: int, internship: Internship }]

- `/get`
  parameters: { internshipID: int, userID: int }
  responses:
  - **200 OK:**
    body: { internship: Internship }
- `/addInformation`
  parameters: { internshipID: int, userID: int, information: String }
  responses:
  - **200 OK:**
    body: { popup: Enum(informationAdded) }
- `/reportProblem`
  parameters: { internshipID: int, userID: int, problem: String }
  responses:
  - **200 OK:**
    body: { popup: Enum(problemReported) }
  - **400 Bad Request:**
    body: { cause: Enum(missingFields) }
- `/handleProblem`
  parameters: { internshipID: int, problemID: int, status: Enum(Unhandled, In Progress, Solved, Hidden) }
  responses:
  - **200 OK:**
    body: { popup: Enum(statusUpdated), problemID: int, status: Enum(Unhandled, In Progress, Solved, Hidden) }
- `/reportFeedback`
  parameters: { internshipID: int, userID: int, feedback: String }
  responses:
  - **200 OK:**
    body: { popup: Enum(feedbackReported) }

/recommendations/generator

- `/generate`[4]
  parameters: { studentID: int, student: Student }
  responses:
  - **204 No Content**
- `/generate`[4]
  parameters: { offerID: int, offer: Offer }
  responses:
  - **204 No Content**

/recommendations/manager

- `/list`
  parameters: { userID: int }
  responses:
  - **200 OK:**
    body: { listOfRecommendations: [recommendation: Recommendation] }
- `/discard`
  parameters: [recommendationID: int]
  responses:
  - **200 OK:**
    body: { popup: Enum(successfullyDiscarded) }
- `/accept`
  parameters: [recommendationID: int]
  responses:
  - **200 OK:**
    body: { popup: Enum(successfullyAccepted) }
- `/reject`
  parameters: [recommendationID: int]
  responses:

[4]Calls to this API endpoint are overloaded, as the new or updated Student or Offer needs to be sent to the RecommendationManager, because changes might still not have been recorded in the RecommendationsDBMS because of eventual consistency.

  - **200 OK:**
    body: { popup: Enum(successfullyRejected) }

/optimizations

- `/optimizeStudent`
  parameters: { userID: int }
  responses:
  - **200 OK:**
    body: { suggestedOptimizations: String }
- `/optimizeOffer`
  parameters: { offerID: int }
  responses:
  - **200 OK:**
    body: { suggestedOptimizations: String }

2.4.2. Other Interfaces

No additional interfaces are needed for the S&C system, as no components inside a specific microservice ever need to communicate.

2.5. Runtime View

In this section, the dynamic behavior of the S&C system is portrayed. For each Use Case identified in Section 3.1 *Use Cases and Activity Diagrams* of the RASD, the corresponding Sequence Diagram is provided, highlighting communication and messages exchanged between the components. Once again, irrelevant details (such as: most client-side interactions between the User and the WebApp, e.g. form filling or negligible button clicking; call of most `get` functions of the API; extremely common operations such as calls to the `validateToken` function of the API) have been left out, as they would add further unnecessary complexity to the model, preventing a clear understanding of how components interact together.

SD1. Sign Up by a Student

The Student clicks the "Sign Up" button on the sign up page and submits their data via the `.../api/v1/authentication/signUp/...` API endpoint. The request is forwarded to the `SecurityManager`, which validates the inputs and checks in the `SecurityDBMS` if the Student is already registered. If all checks succeed, it stores the Student data and calls the `MailingService` API to send a verification email to the specified email address through Simple Mail Transfer Protocol (SMTP). When the Student receives the email, if they click on the confirmation link within 24 hours, a call to the `.../api/v1/authentication/verifyProfile/...` API endpoint is made: the account is marked as verified in the `SecurityDBMS`, and the Student is redirected to the page for updating their profile, which takes place according to SD5 - Update User Profile. If the timer for the verification instead expires, the account data is deleted from the `SecurityDBMS`.

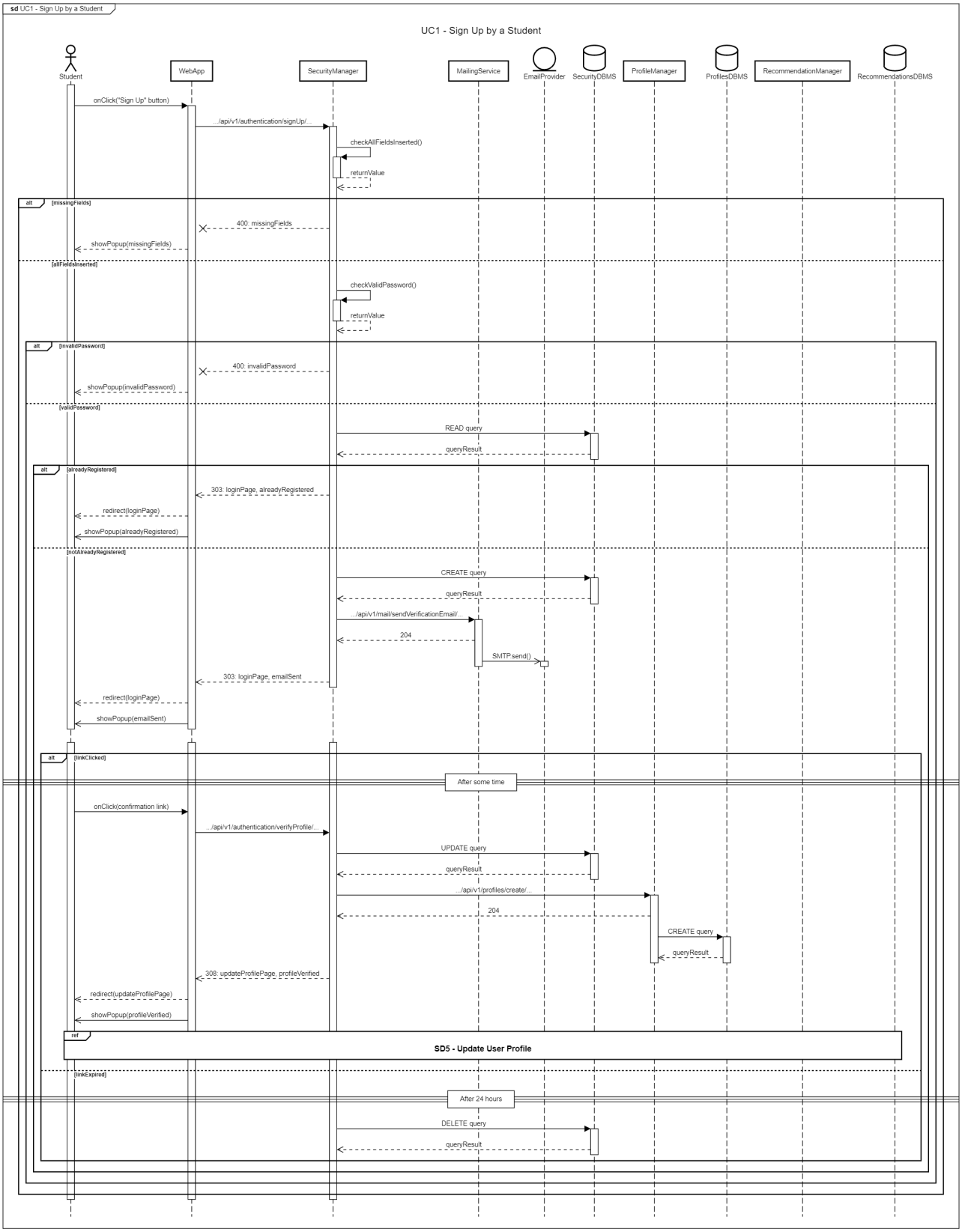

Figure 2.3: Sign Up by a Student

SD2. Sign Up by a Company

This Sequence Diagram is the same as the previous one, the only difference being that is is started by a Company and not by a Student.

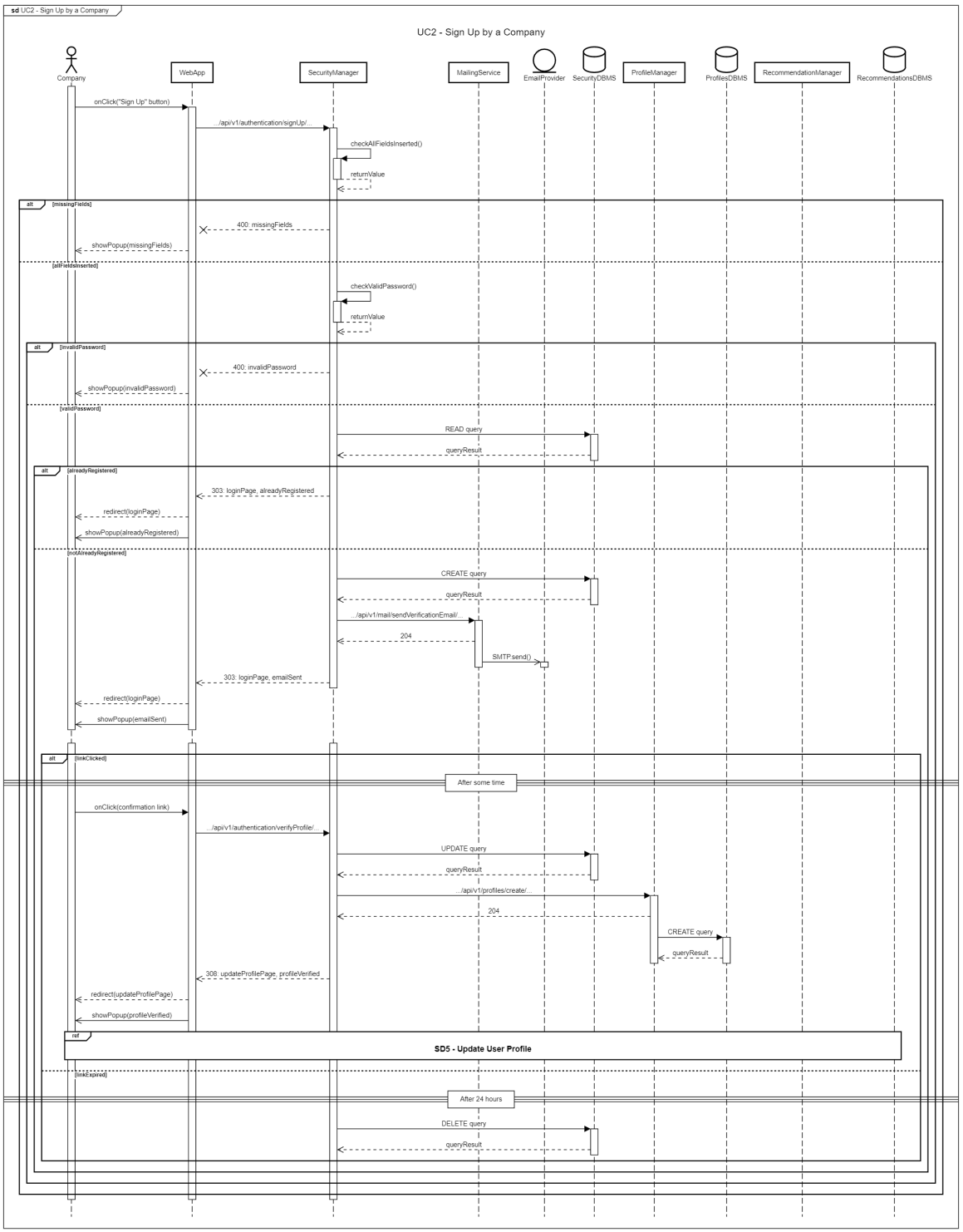

Figure 2.4: Sign Up by a Company

SD3. Sign Up by a University

The same goes for this Sequence Diagram, which is in turn started by a University.

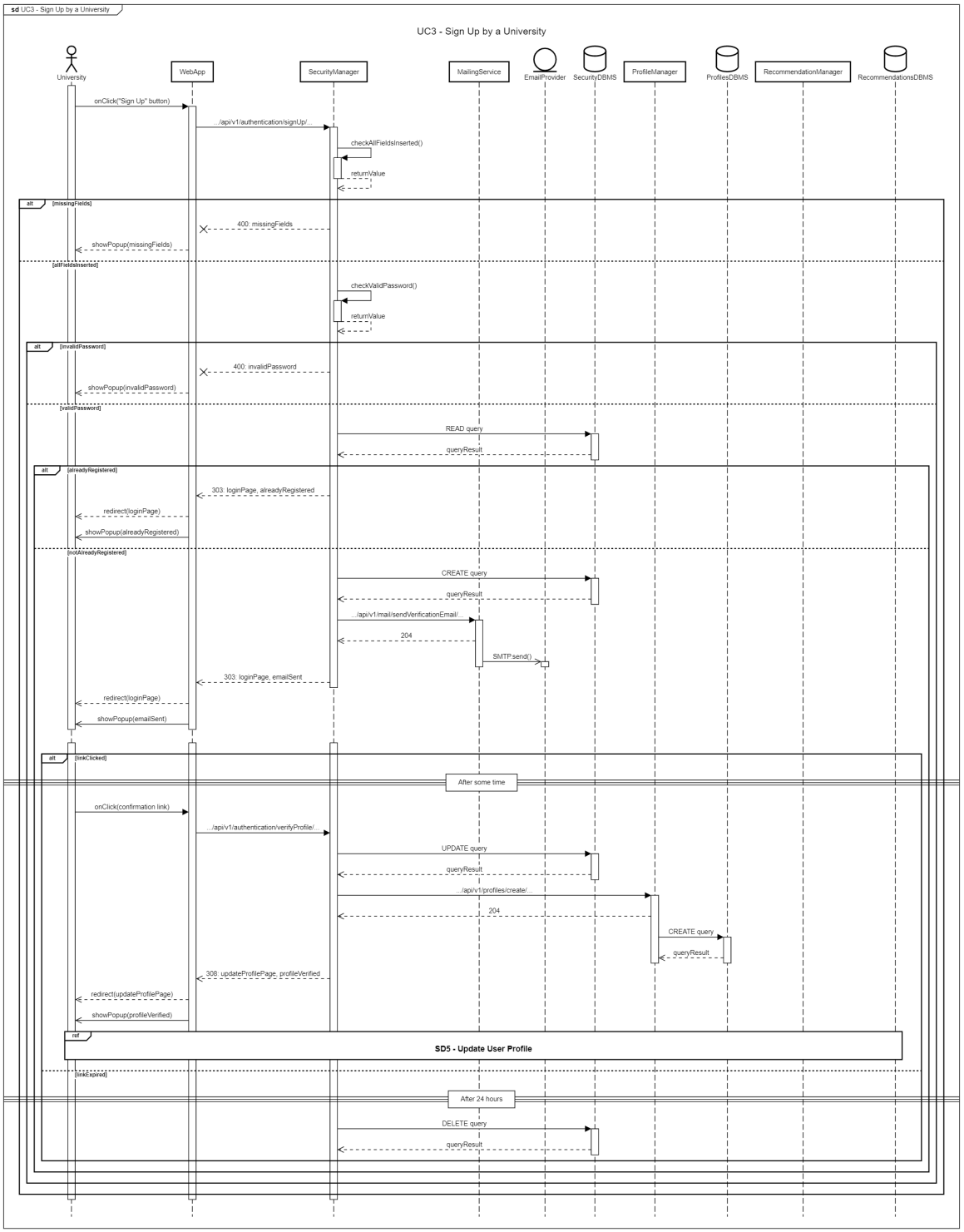

Figure 2.5: Sign Up by a University

SD4. Log In by a User

The User initiates the process by clicking the "Login" button on the login page, which sends a request to the `.../api/v1/authentication/login/...` API endpoint. The `SecurityManager` queries the `SecurityDBMS` to validate the provided credentials. If the credentials are correct and the account is verified, the `SecurityManager` generates an authentication token, which is sent back to the `WebApp`. Then, the `SecurityManager` checks whether the User has completed their profile during the signup process. If the profile is incomplete, the User is sent the token and is redirected to the page for updating their profile in order for them to insert the required information; otherwise, they are redirected to the dashboard page with a success status and the token for authentication.

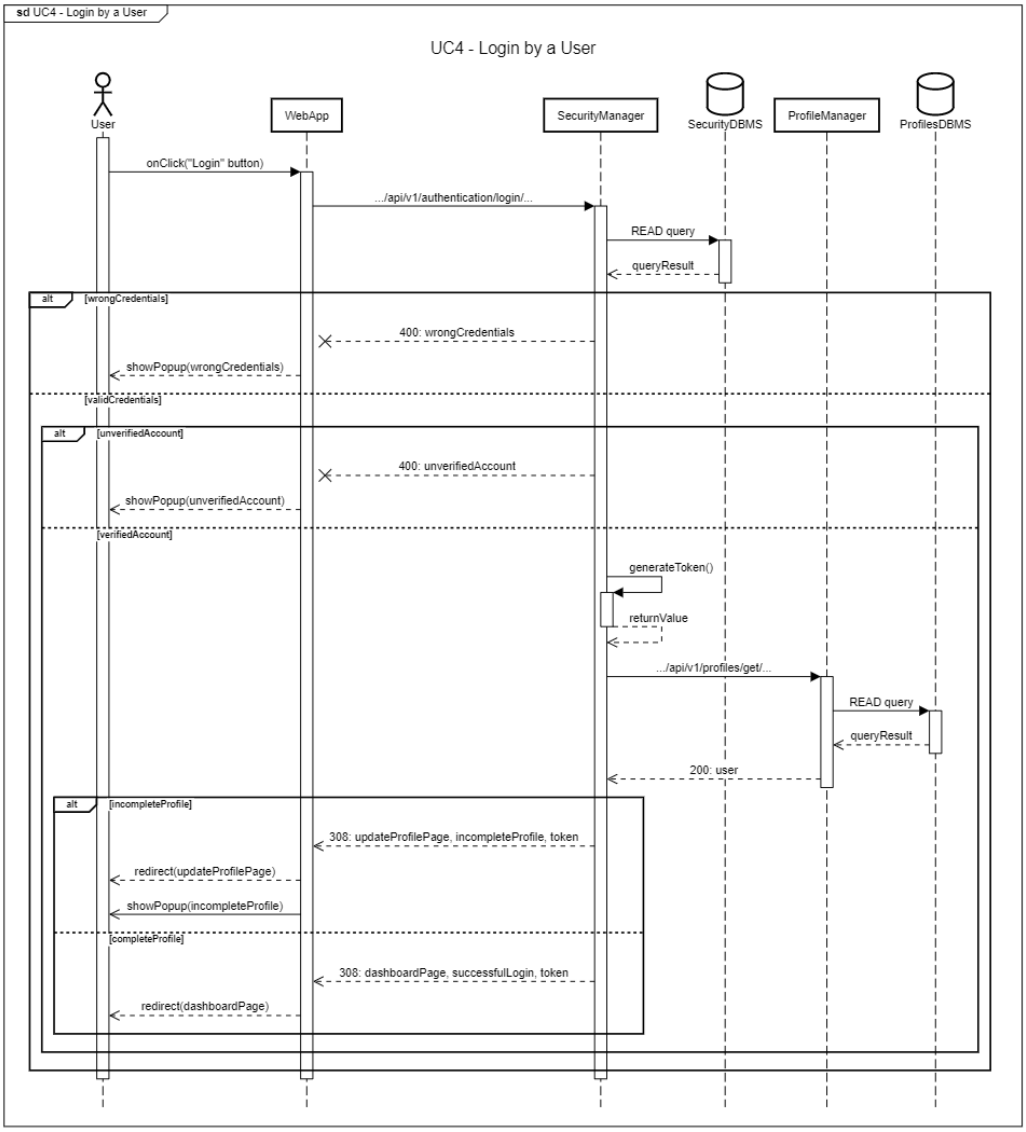

Figure 2.6: Log In by a User

SD5. Update User Profile

The User initiates the action by clicking the "Apply Changes" button after having inserted updated profile information in the profile update page, triggering a request to the `.../api/v1/profiles/update/...` API endpoint. The `ProfileManager` validates the input to ensure all required fields are present. If the input is valid, the `ProfileManager` updates the profile data in the `ProfilesDBMS`, and then notifies the successful update to the WebApp, which shows a popup to the User. The `ProfileManager` then triggers the `RecommendationManager` to execute the logic of SD10 - Generate Recommendations for detecting possible new recommendations to be generated.

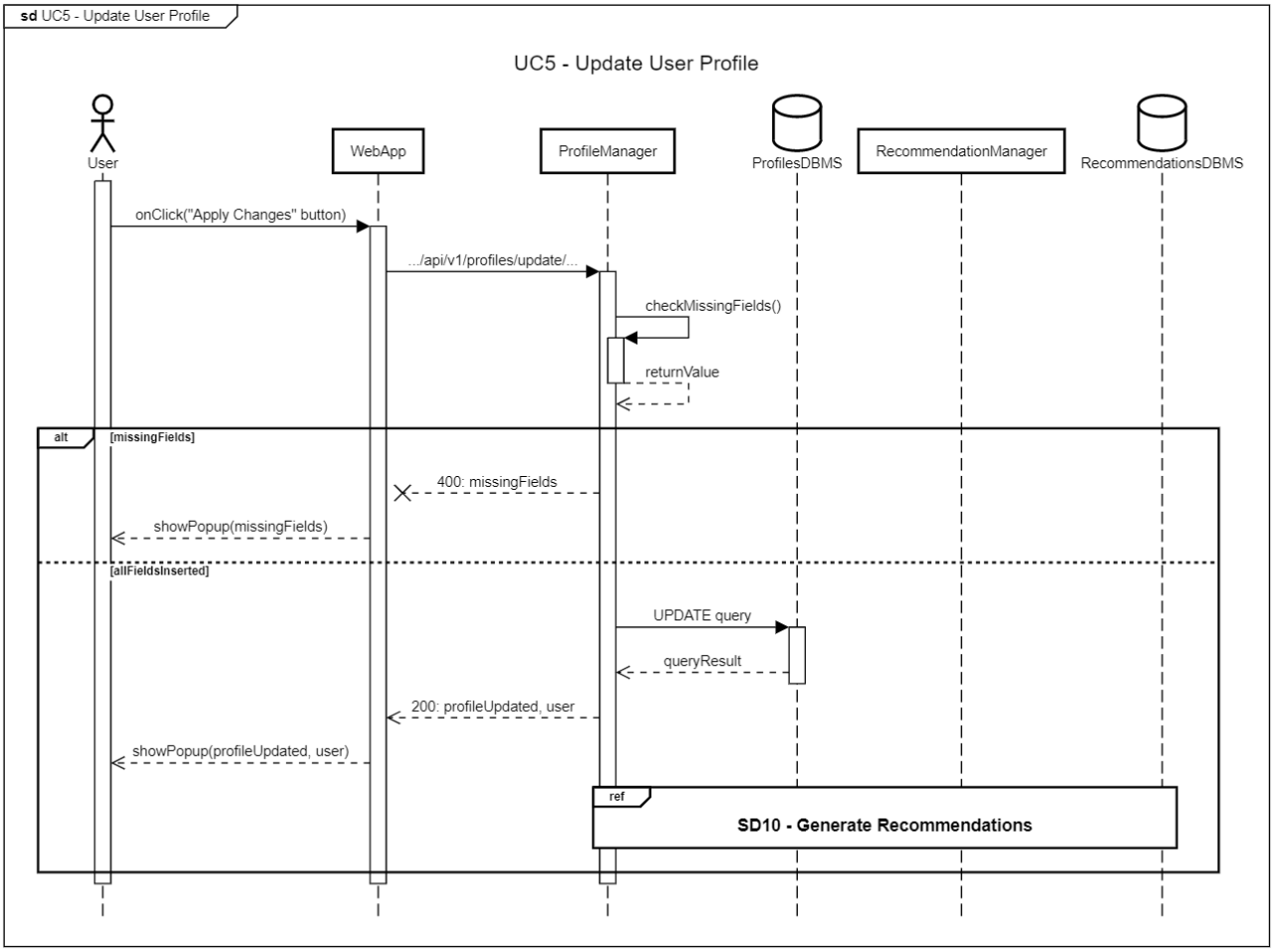

Figure 2.7: Update User Profile

SD6. Publish an Internship Offer

The Company initiates the action by clicking the "Submit" button after having inserted information about the new offer to be posted in the offer creation page, which sends a request to the `.../api/v1/offers/publish/...` API endpoint. The `OfferManager` validates the input to ensure all required fields are completed. If all fields are provided and all the information is compliant the `OfferManager` creates a new entry in the `InternshipsDBMS`. Once the query succeeds, the `OfferManager` confirms the offer's publication and the Company is notified with a popup indicating that the offer has been successfully published. The component then triggers the `RecommendationManager` to generate recommendations as part of SD10 - Generate Recommendations.

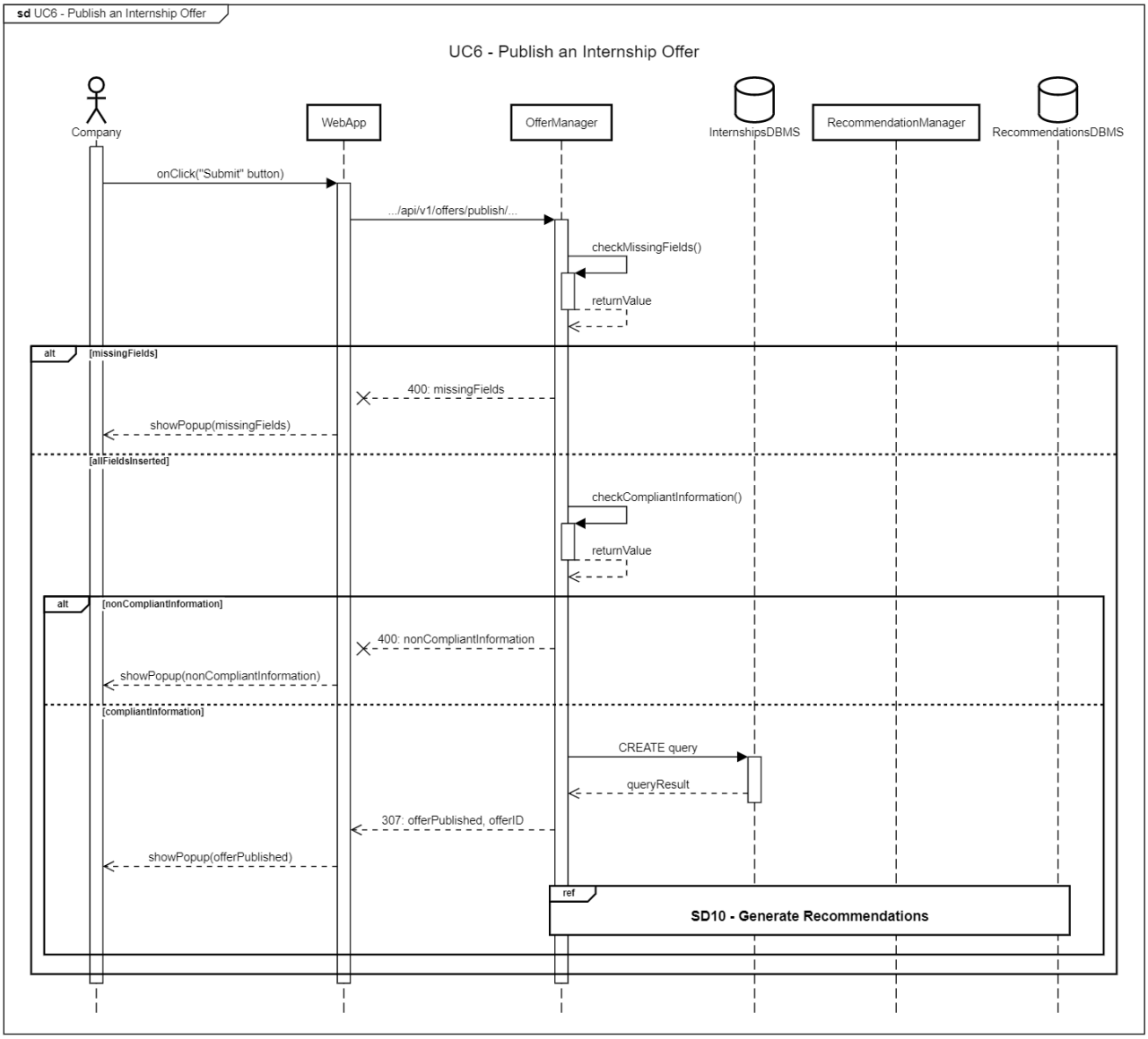

Figure 2.8: Publish an Internship Offer

SD7. Update Internship Offer

The Company initiates the action either by confirming the withdrawal or by applying changes to one of its offers on the page dedicated to that specific offer. For the withdrawal, the `.../api/v1/offers/withdraw/...` API endpoint is the one where the request is sent. The `OfferManager` retrieves related "Unhandled" recommendations by interacting with the `RecommendationManager`, which in turn queries the `RecommendationsDBMS`, and subsequently asks the exposed API to discard them. Then the `OfferManager` doesn't DELETE the offer, but only performs an UPDATE query for its status in the `InternshipsDBMS`, so that its data doesn't get lost. Finally, the component redirects the Company to the dashboard with a success alert. When updating offer information, the request is instead forwarded to the `.../api/v1/offers/update/...` API endpoint. The `OfferManager` validates the input for missing fields and compliance. If valid, the updated information is stored in the `InternshipsDBMS`, and the system notifies the Company of the successful update, in the end triggering the `RecommendationManager` to generate new recommendations as defined in SD10 - Generate Recommendations.

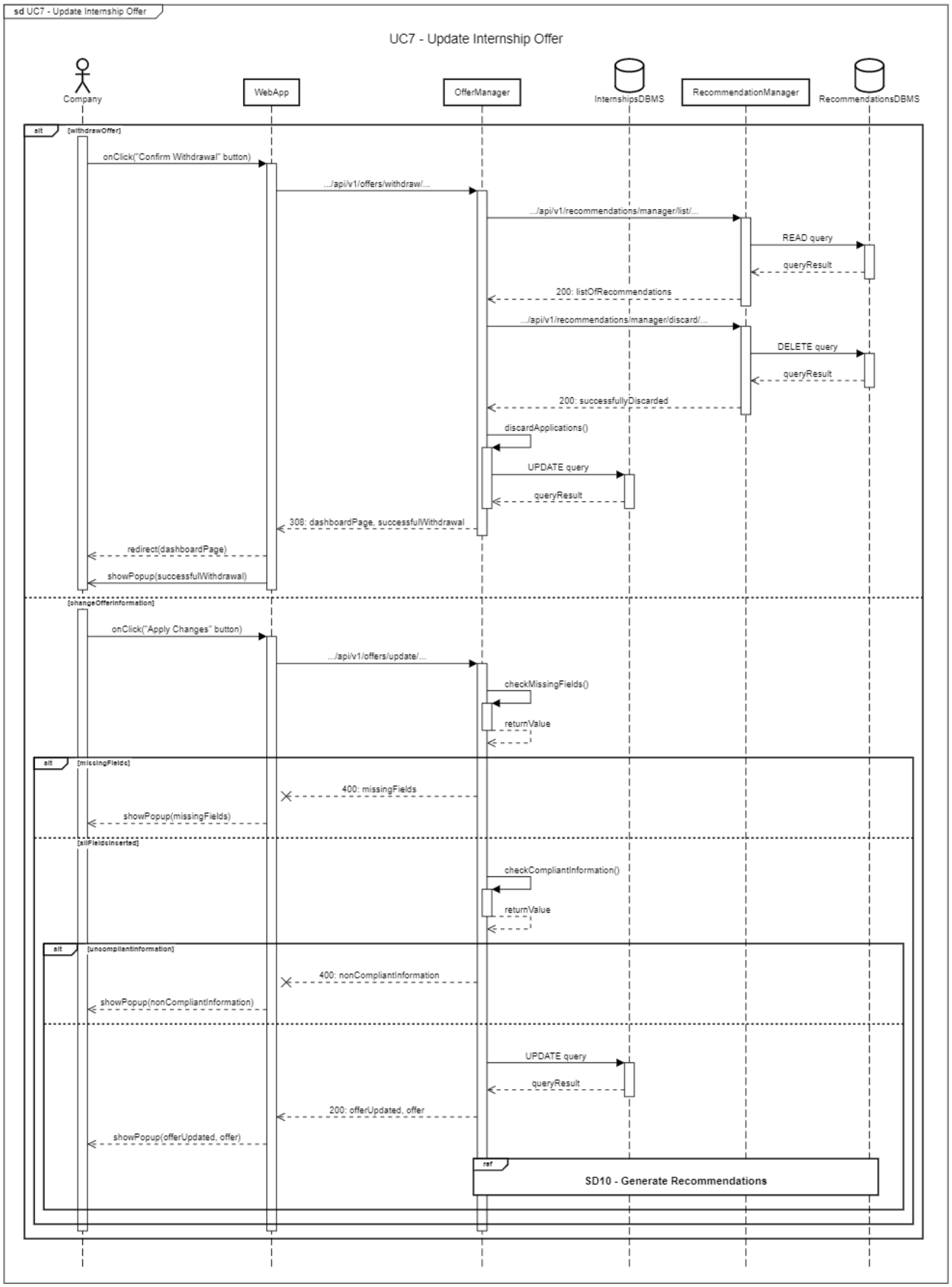

Figure 2.9: Update Internship Offer

SD8. Search Internship Offers

The Student initiates the action by clicking the "Submit" button on the dashboard page after having inserted the filter criteria, triggering a request to the `.../api/v1/offers/search/...` API endpoint. The `OfferManager` processes the request and queries the `InternshipsDBMS` using a READ operation to fetch the relevant internship offers which match the search query criteria. The `OfferManager` returns the query results to the `WebApp`, which displays the results to the Student. If no results are found, the system shows a custom message of the list of internships instead.

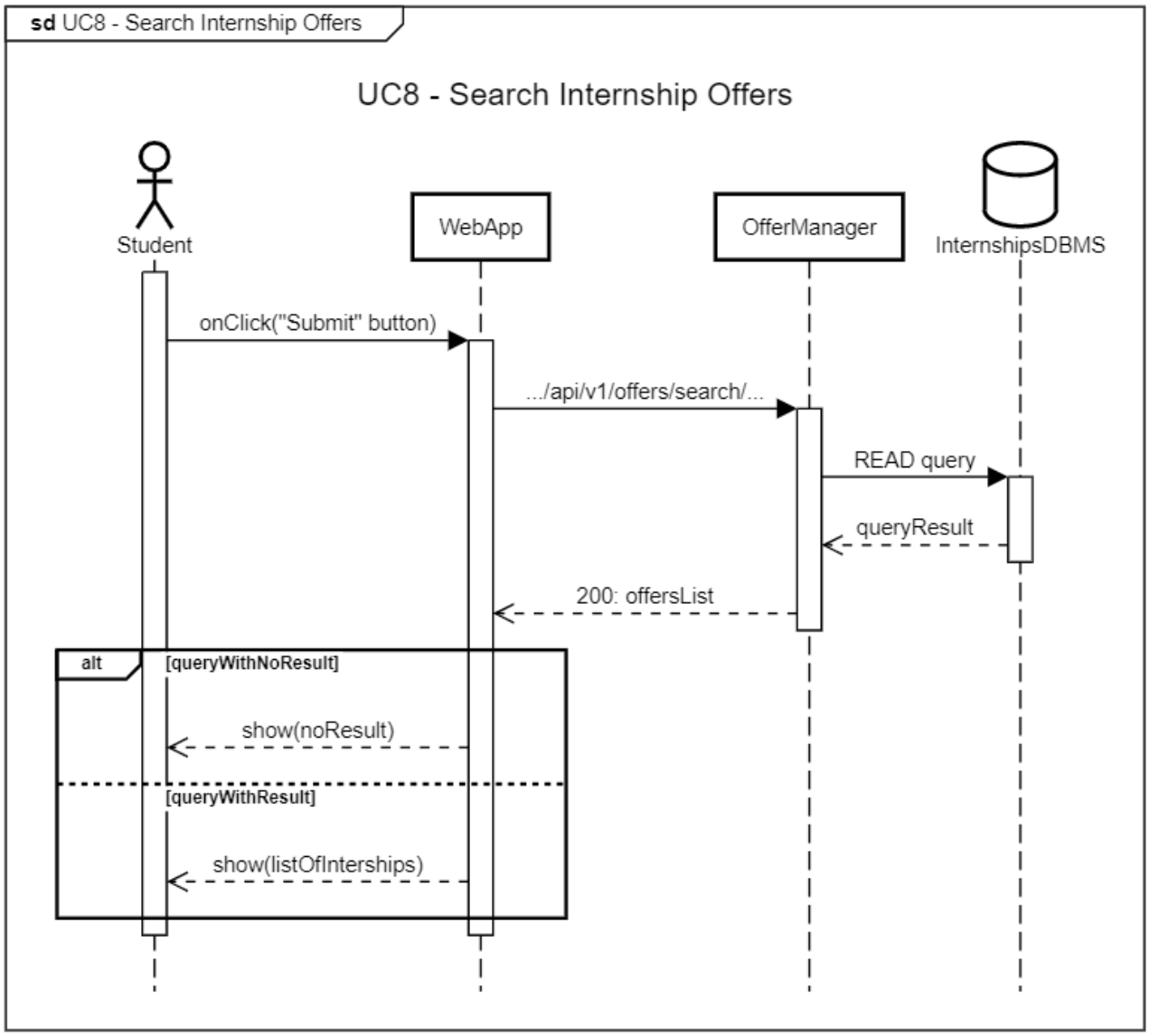

Figure 2.10: Search Internship Offers

SD9. Apply to an Internship Offer

The Student submits an application for an internship offer by clicking the "Apply" button on an offer's page, invoking the `.../api/v1/offers/apply/...` API endpoint. The `OfferManager` first verifies the deadline through the `InternshipsDBMS`: if the deadline has expired, an error is shown to the Student; otherwise, the `OfferManager` calls the `RecommendationManager`, which interacts with the `RecommendationsDBMS` to accept the recommendations of the applicating Student about that offer and to discard the recommendations of the offering Company about that Student for that Offer, as expected by the functional requirements. Finally, the application is recorded in the `InternshipsDBMS`, and the Student is notified of the successful submission.

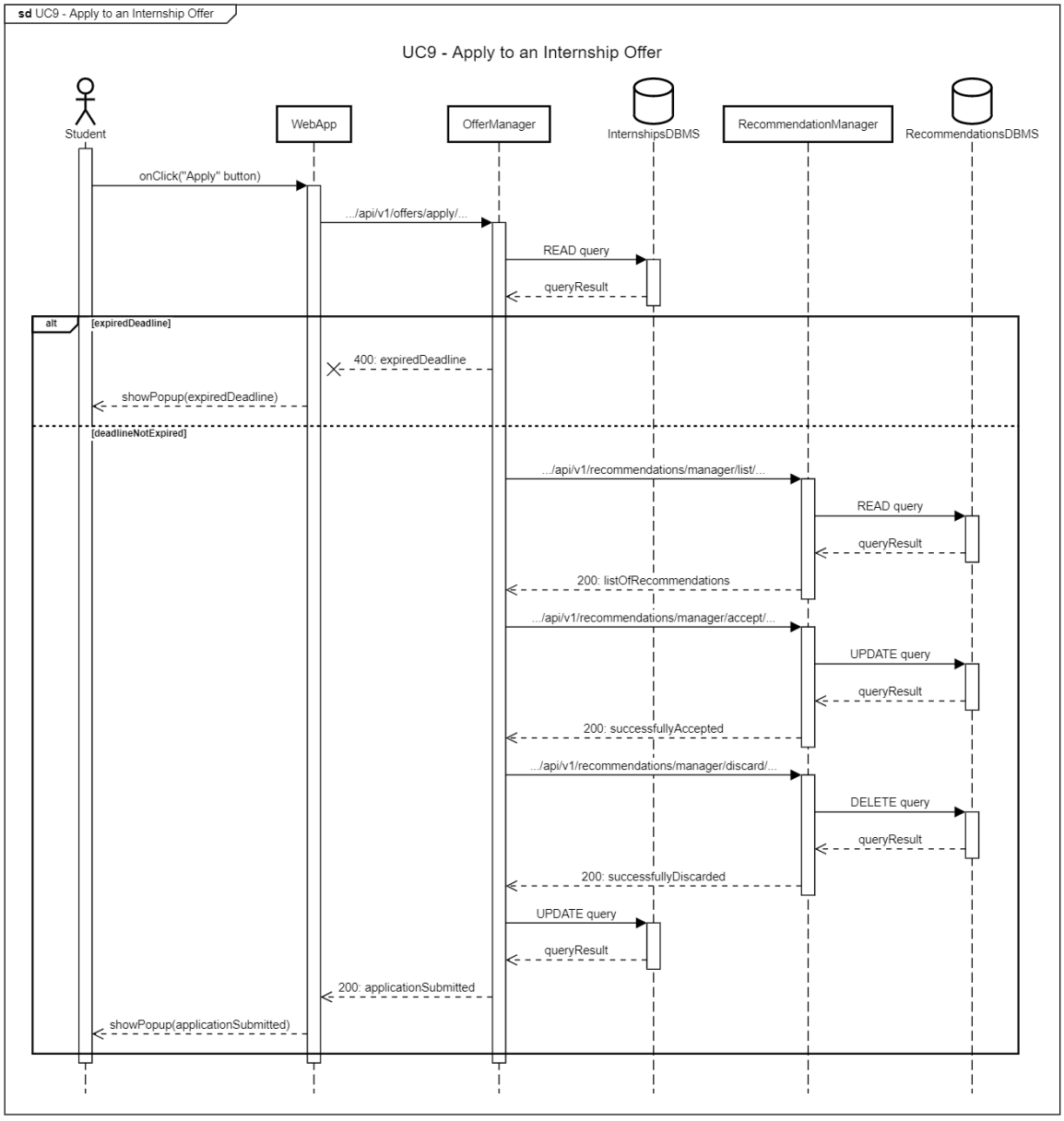

Figure 2.11: Apply to an Internship Offer

SD10. Generate Recommendations

The process can be triggered either by a profile update through the `ProfileManager` or an internship offer update via the `OfferManager`, both invoking the `.../api/v1/recommendations/generator/generate/...` API endpoint. Upon invocation, the `RecommendationManager` queries the `RecommendationDBMS` to retrieve relevant data for generating new recommendations. Once the query is completed and the data is processed, the new recommendations are pushed into the `RecommendationDBMS`.

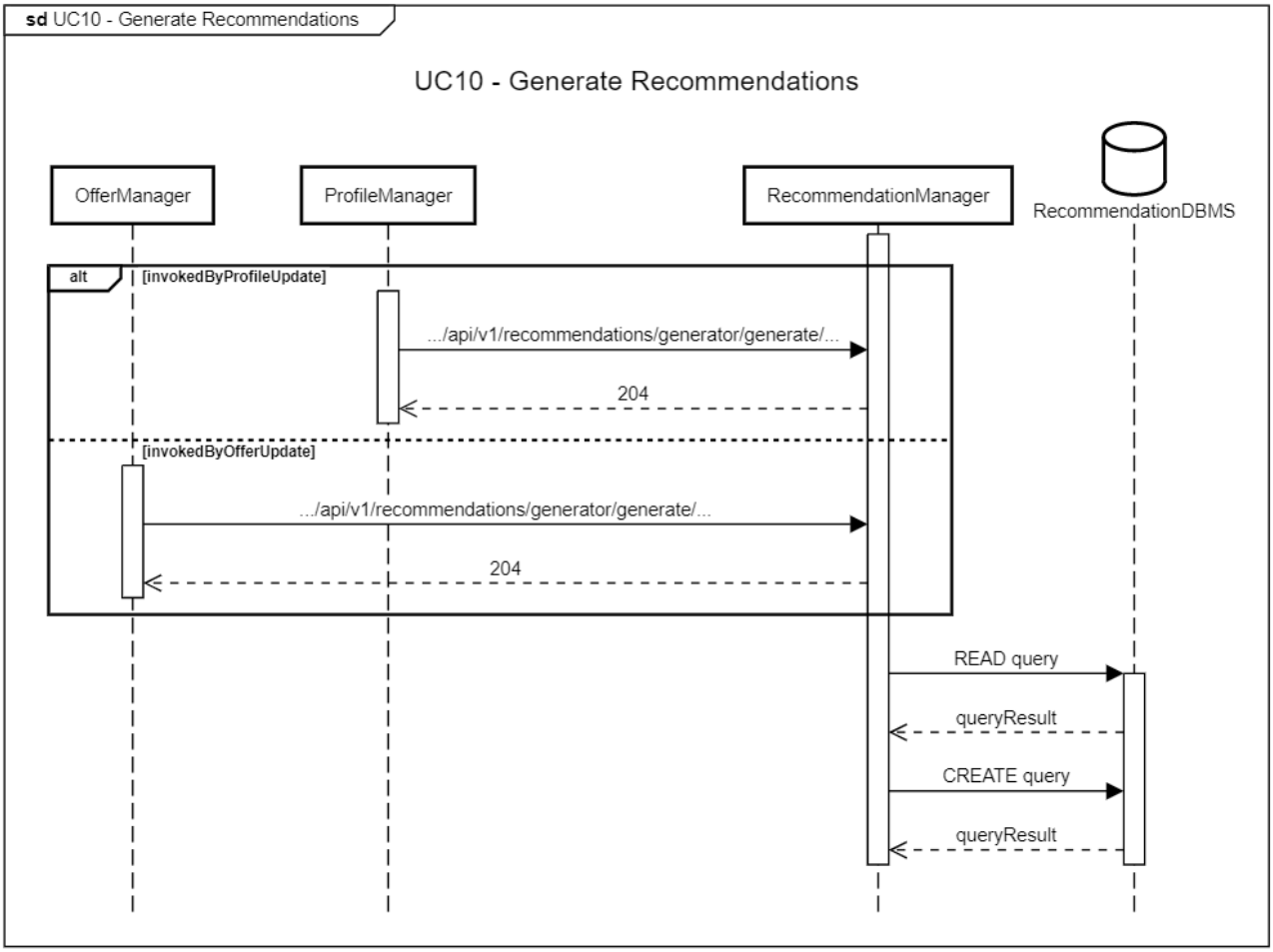

Figure 2.12: Generate Recommendations

SD11. Manage Internship Recommendations

For each of their "Unhandled" recommendations, the Student can either accept it, reject it, or postpone the decision. Whether the Student clicks the "Accept" or the "Reject" button on a recommendation's widget, triggering the corresponding `.../api/v1/recommendations/manager/...` API endpoint of the `RecommendationManager`, the status of the recommendation in the `RecommendationsDBMS` is updated accordingly. In case of acceptance, an application for the offer from the Student is also generated, as discussed in the functional requirements, by invoking SD9 - Apply to an Internship Offer. A confirmation popup for the success of the operation is finally displayed to the Student. Recommendations postponed (for which no action is taken) remain unaltered.

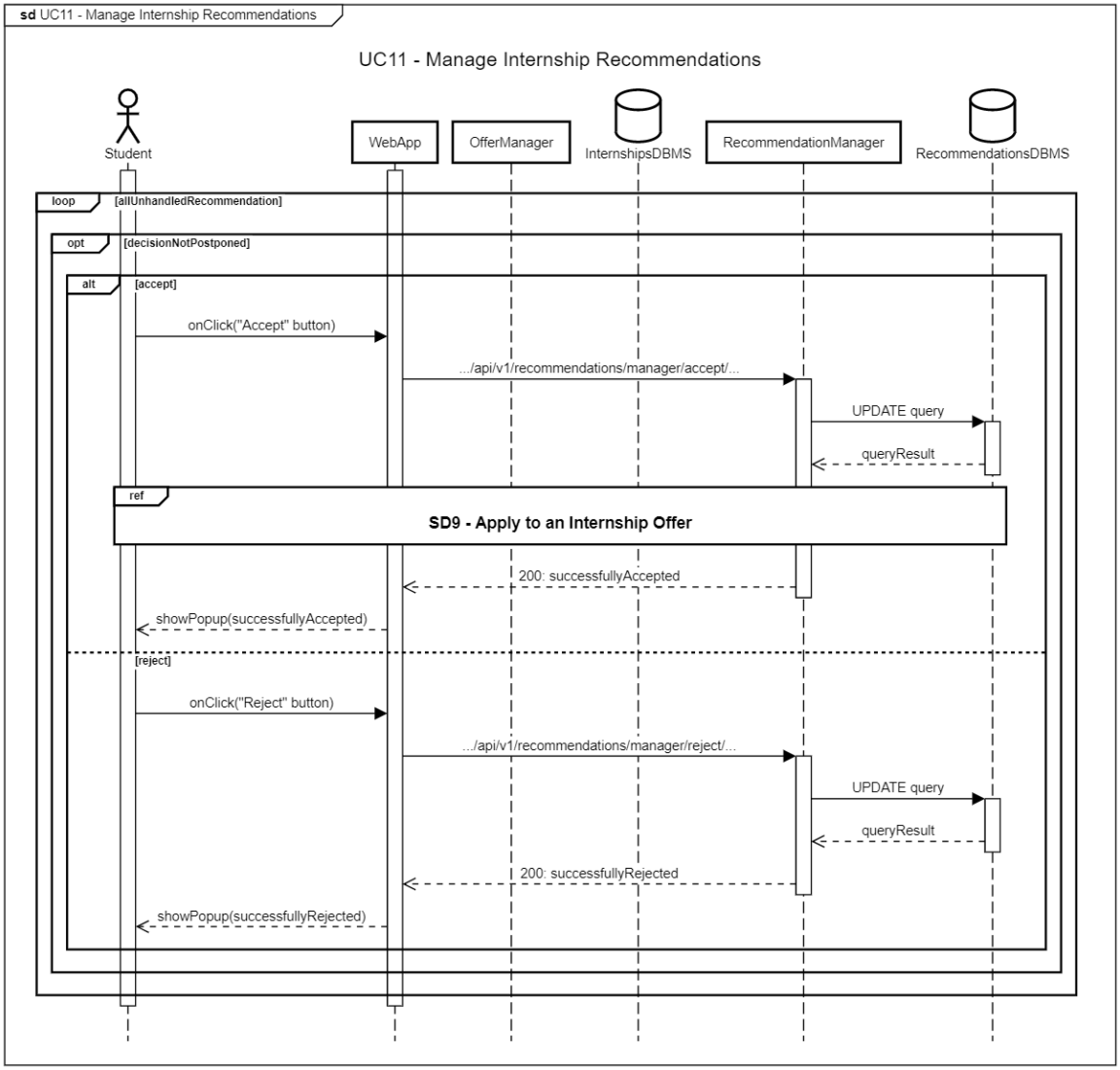

Figure 2.13: Manage Internship Recommendations

SD12. Manage Students Recommendations

Similar to the previous one, the following Sequence Diagram describes the process of recommendations management by a Company. The only difference is that, instead of generating an offer application, when accepting a recommendation a new one is generated for the corresponding Student if they don't already have one.

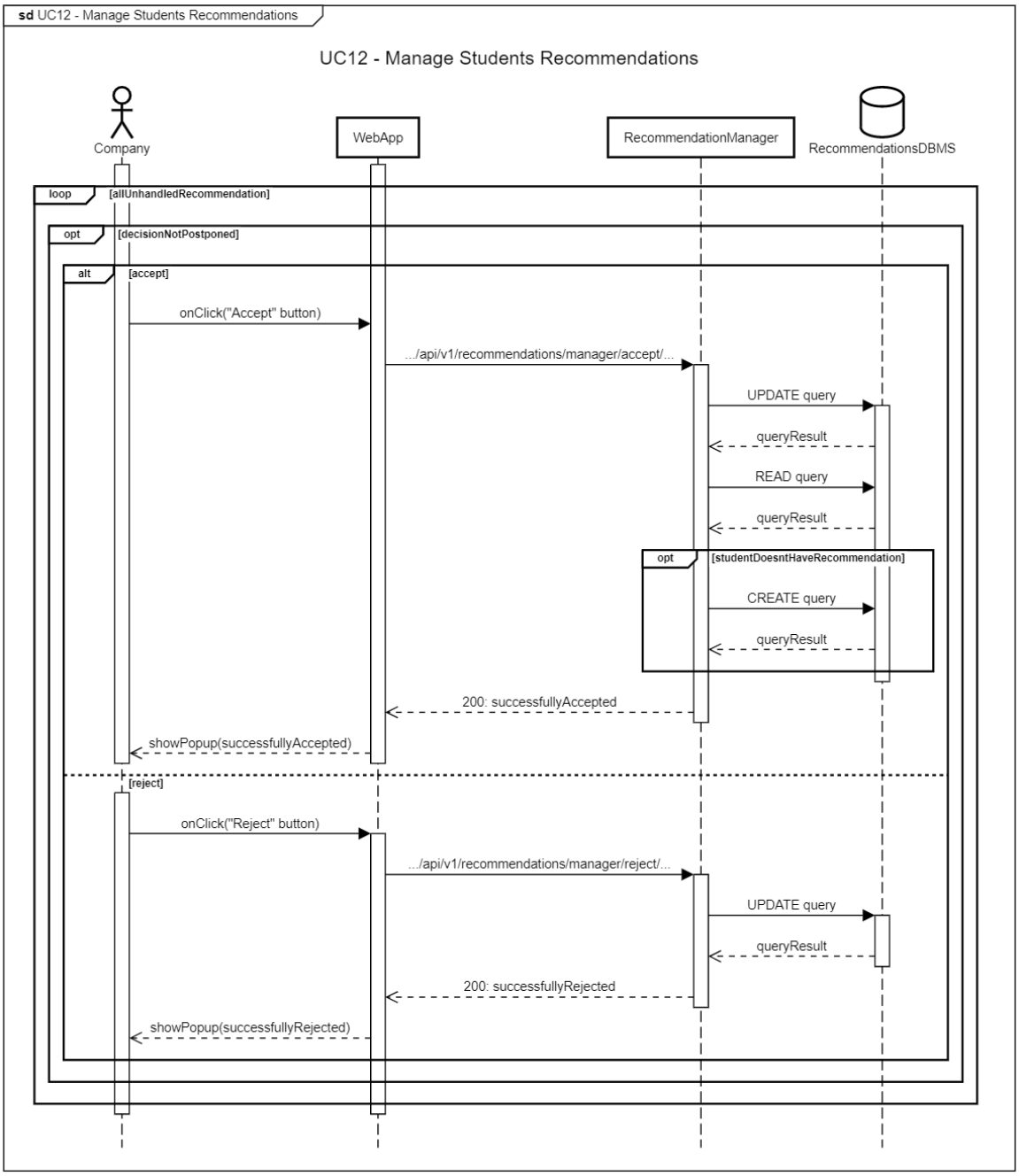

Figure 2.14: Manage Students Recommendations

SD13. Manage an Interview

This Sequence Diagram illustrates the process of scheduling, conducting, and evaluating an interview between a Company and a Student.

1. The Company first clicks the "Send Invitation" button on its dashboard, sending an interview invitation through the `.../api/v1/interviews/sendInvitation/...` API endpoint, managed by the `InterviewManager`, which creates an entry in the `InternshipsDBMS` for the invitation. The Student may accept or decline the invitation by clicking the corresponding buttons, which invoke the respective `.../api/v1/internships/...` API endpoint:

   - If accepted, the `InterviewManager` updates the status in the `InternshipsDBMS`, and the Student is notified.
   - If declined, the `InternshipsDBMS` is updated accordingly, and the Company may decide to try to reschedule it based on the motivation provided by the Student.

   This phase loops until either the Student accepts one invitation, or the Company decides that no agreement can be reached and stops the process.

2. For in-platform interviews only, the Company posts some questions via the "Post Questions" button, which calls the `.../api/v1/interviews/createQuestions/...` API endpoint, and when the scheduled time comes the Student involved in the interview replies to each question and submits their answer through the "Submit" button of the specific question in its dashboard, which interacts with the system through the `.../api/v1/interviews/submitAnswer/...` API endpoint. The questions and the answers are recorded in the `InternshipsDBMS`, and popups are shown to both parties to confirm their submissions.

3. After the interview, the Company evaluates the candidate by submitting the results through the specific "Evaluate" button, calling `.../api/v1/interviews/evaluate/...` API endpoint. The result and eventual feedback are stored in the `InternshipsDBMS` by updating the interview status, and the Company is notified of the successful evaluation.

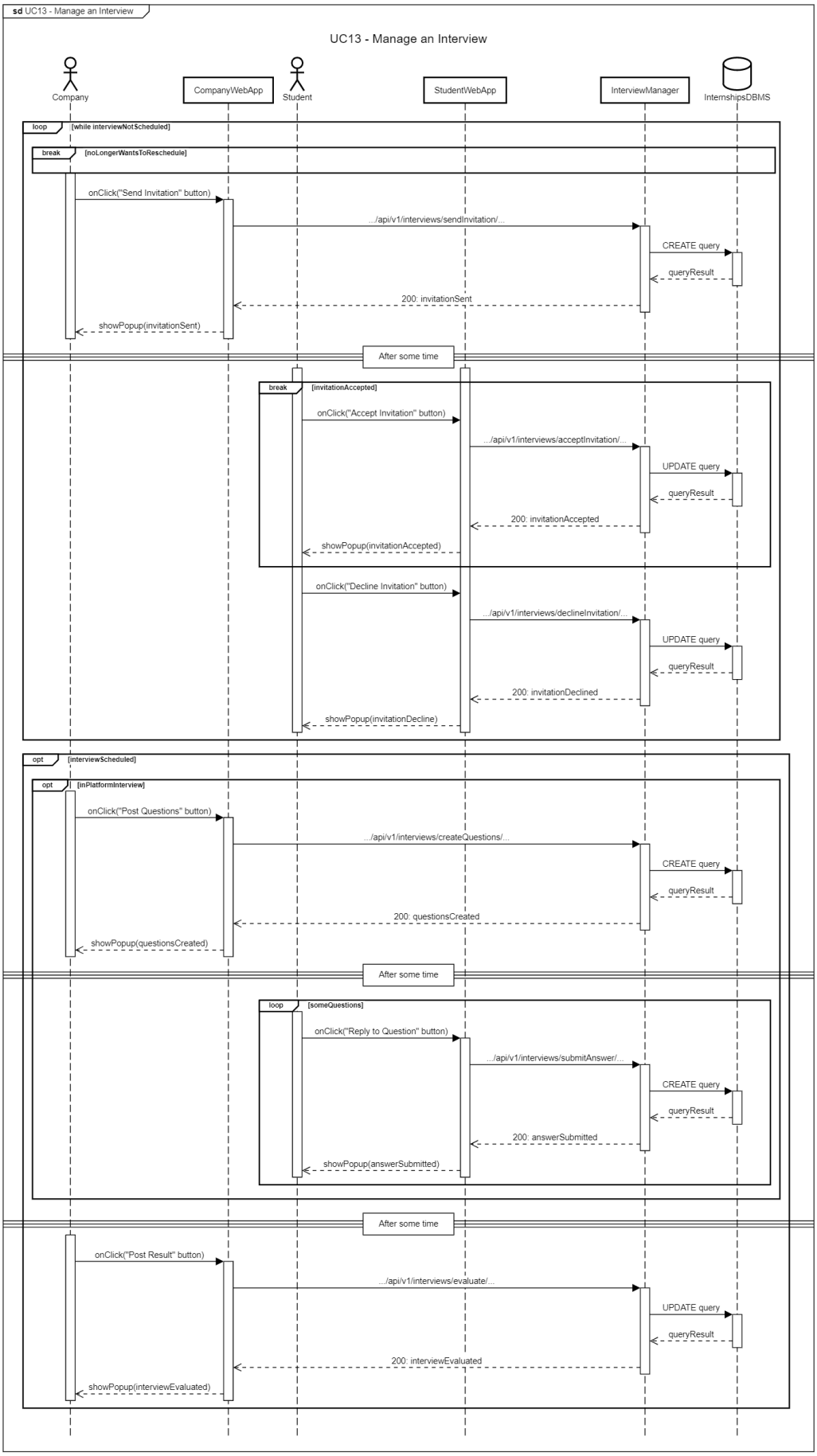

Figure 2.15: Manage an Interview

SD14. Provide Information for an ongoing Internship

The Party initiates the process by clicking the "Post Information" on an ongoing internship's page, which triggers a request to the `.../api/v1/internships/addInformation/...` API endpoint handled by the `IMFManager`. The `IMFManager` adds the new internship data in the `InternshipsDBMS`. Once the operation is successfully completed, the `WebApp` receives a confirmation and displays a popup notifying the Party of the successful submission.

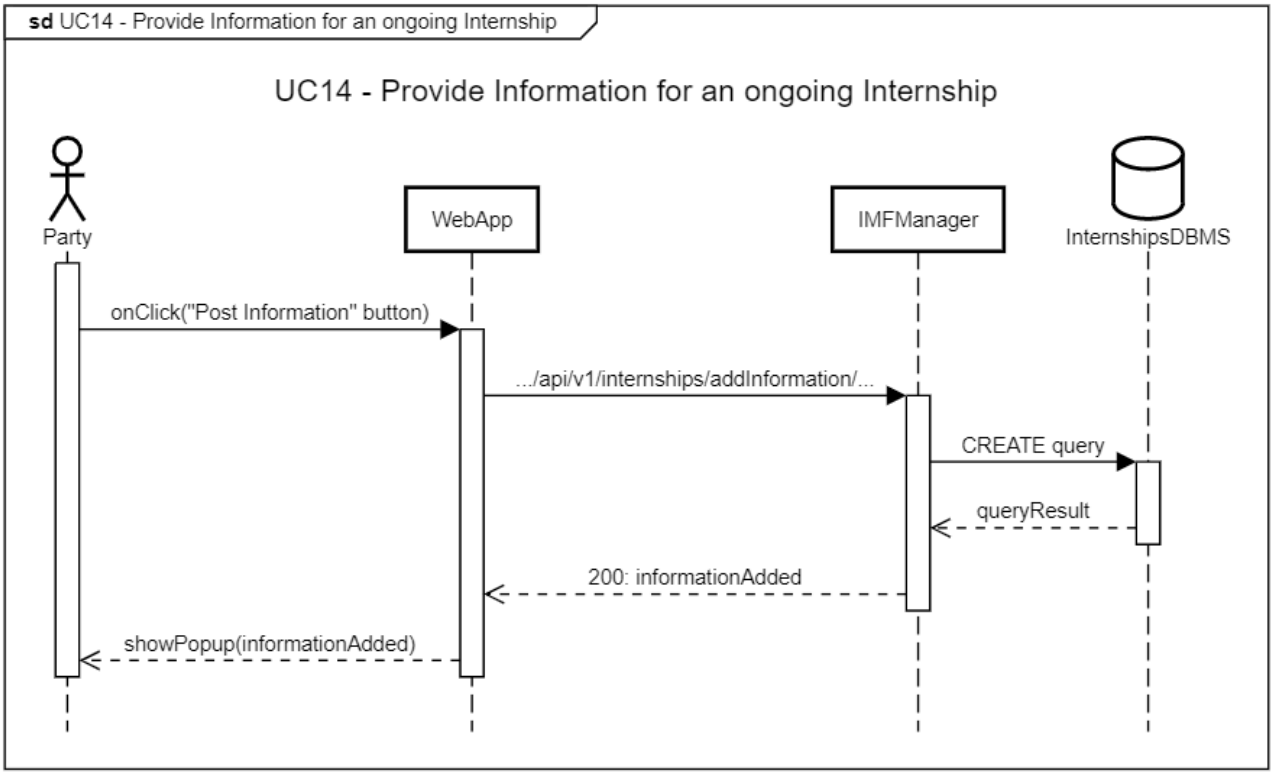

Figure 2.16: Provide Information for an ongoing Internship

SD15. Monitor an ongoing Internship

The Party selects an internship widget, sending a request to the `.../api/v1/internships/get/...` API endpoint handled by the `IMFManager`. The `IMFManager` retrieves the internship details from the `InternshipsDBMS`. Upon successfully fetching the data, the `WebApp` displays the internship details to the Party. Additionally, in the visualized section the Party can also optionally provide new information about the internship if they need to, referring to the process detailed in SD14 - Provide Information for an ongoing Internship.

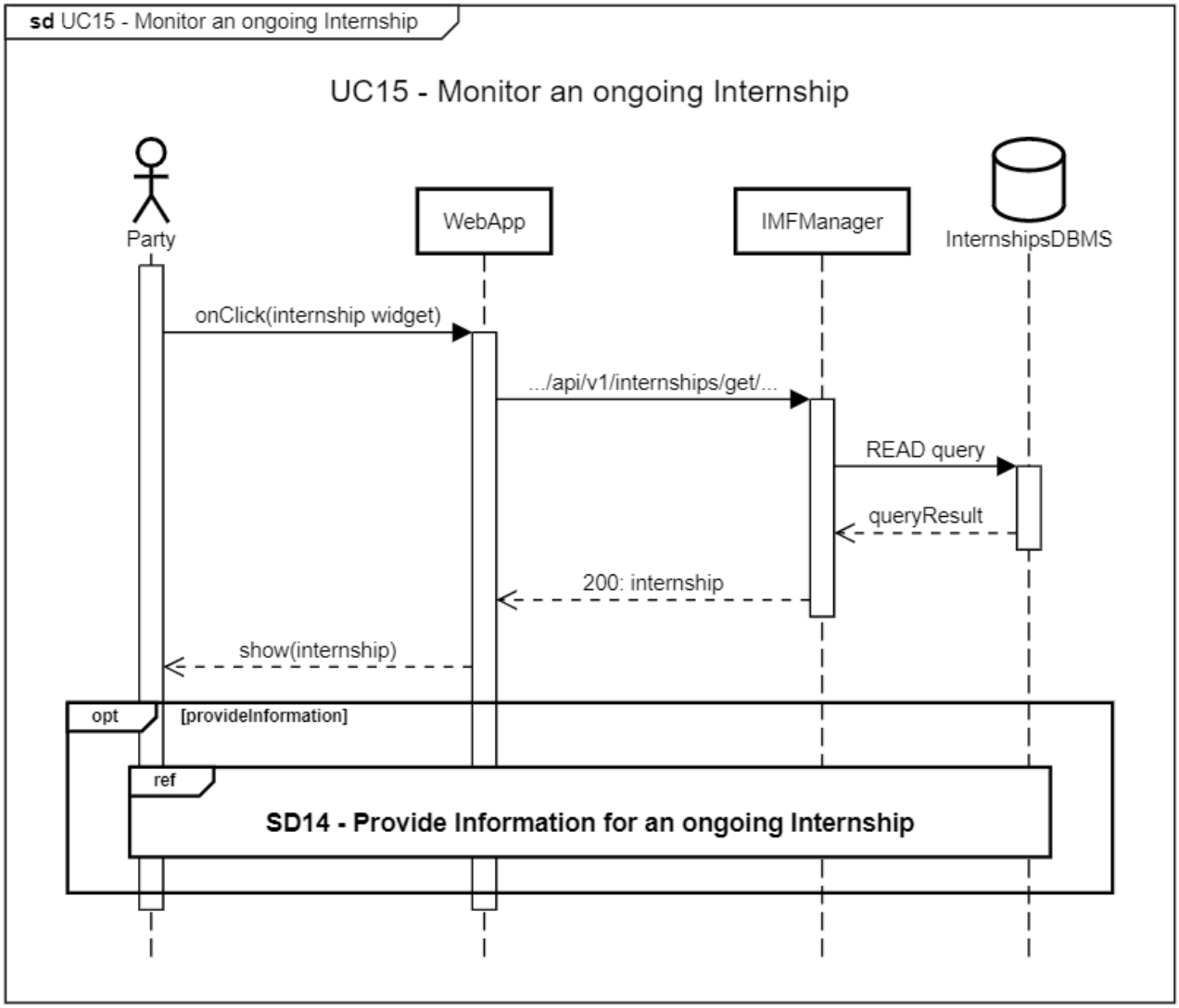

Figure 2.17: Monitor an ongoing Internship

SD16. Report Problems during an Internship

The Party initiates the process by clicking the "Report" button on an ongoing internship's page, triggering a request to the `.../api/v1/internships/reportProblem/...` API endpoint exposed by the `IMFManager`, which checks the submitted fields for completeness. If all fields are provided, the `IMFManager` records the problem in the `InternshipsDBMS` creating a new entry. Once the problem is successfully reported, the `WebApp` notifies the Party with the appropriate confirmation popup.

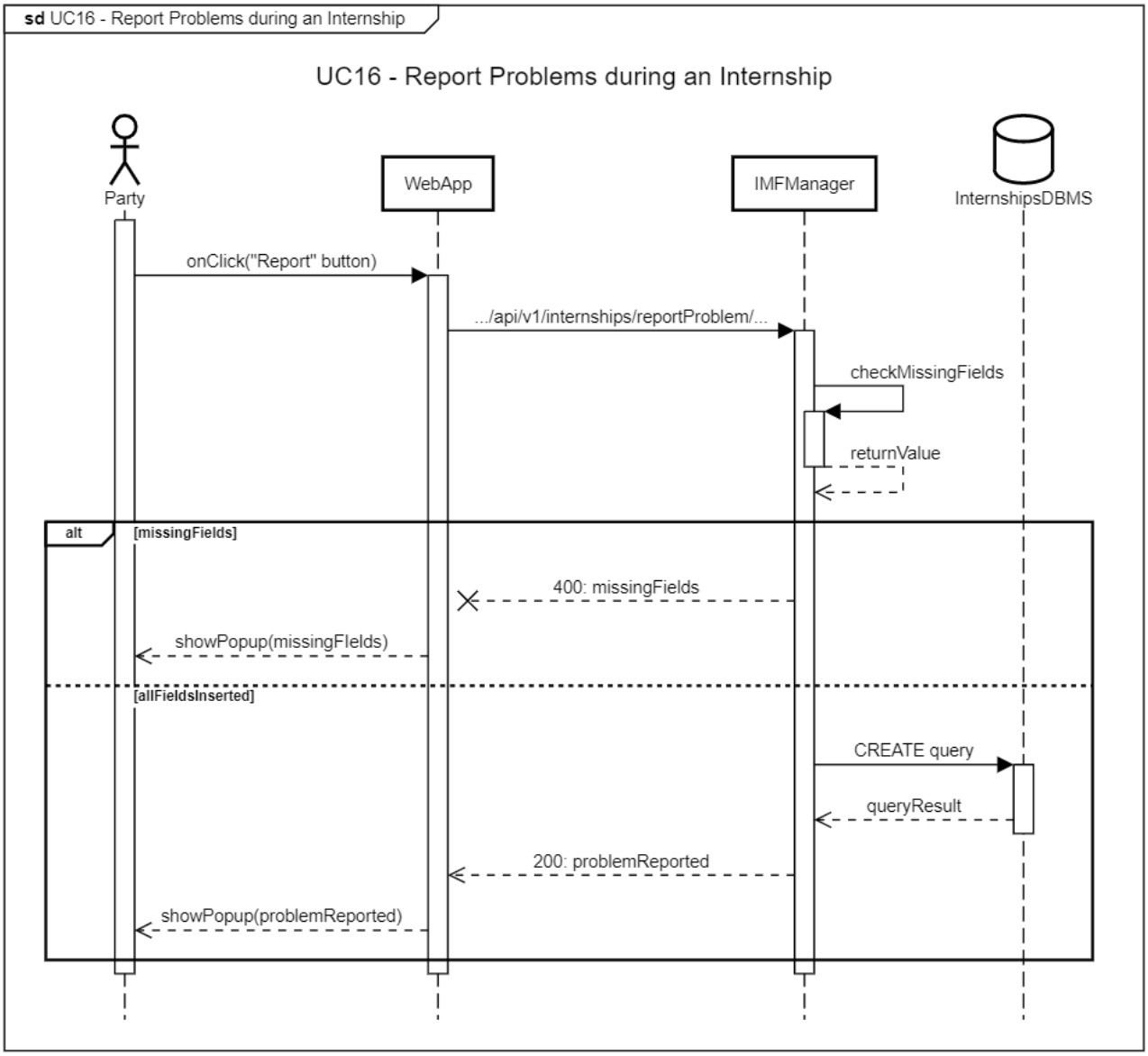

Figure 2.18: Report Problems during an Internship

SD17. Handle Problems during an Internship

The University initiates the process by clicking one of the "Mark as In Progress", "Mark as Solved" or "Hide" buttons on a problem's widget in its dashboard page, triggering a request to the `.../api/v1/internships/handleProblem/...` API endpoint, managed by the `IMFManager`. The `IMFManager` updates the problem's status in the `InternshipsDBMS` according to the button clicked by the University. Upon successful completion, the `WebApp` receives confirmation and notifies the University with a popup indicating that the problem has been correctly marked as solved.

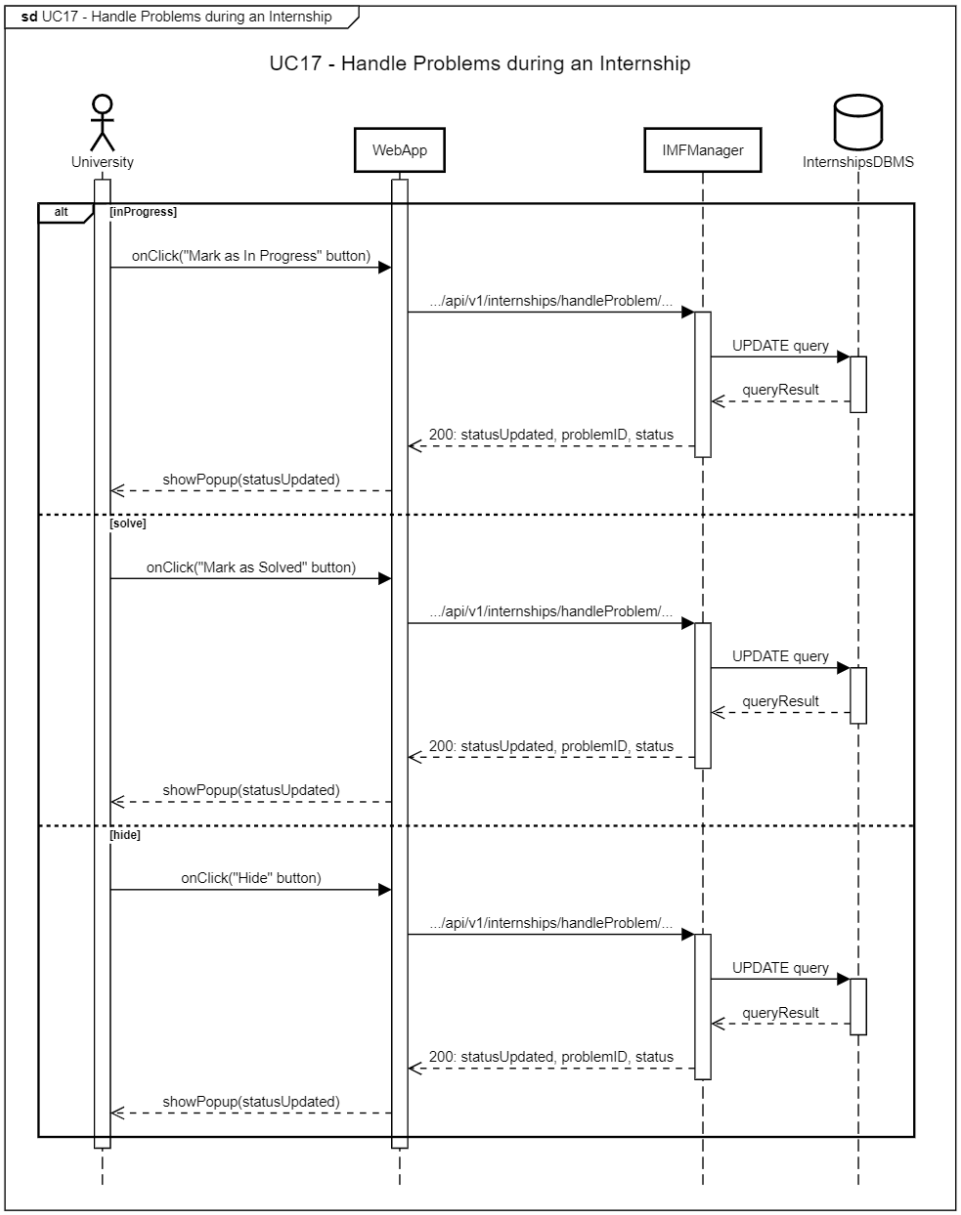

Figure 2.19: Handle Problems during an Internship

SD18. Report Feedback after an Internship

The Party begins by clicking the "Submit" button in the widget of an ended internship on their dashboard page, sending a request to the `.../api/v1/internships/reportFeedback/...` API handled by the `IMFManager`, which processes the request and updates the relevant records in the `InternshipsDBMS`. Once the feedback is successfully recorded, the `WebApp` is notified of the successful operation and displays a confirmation popup to the Party.

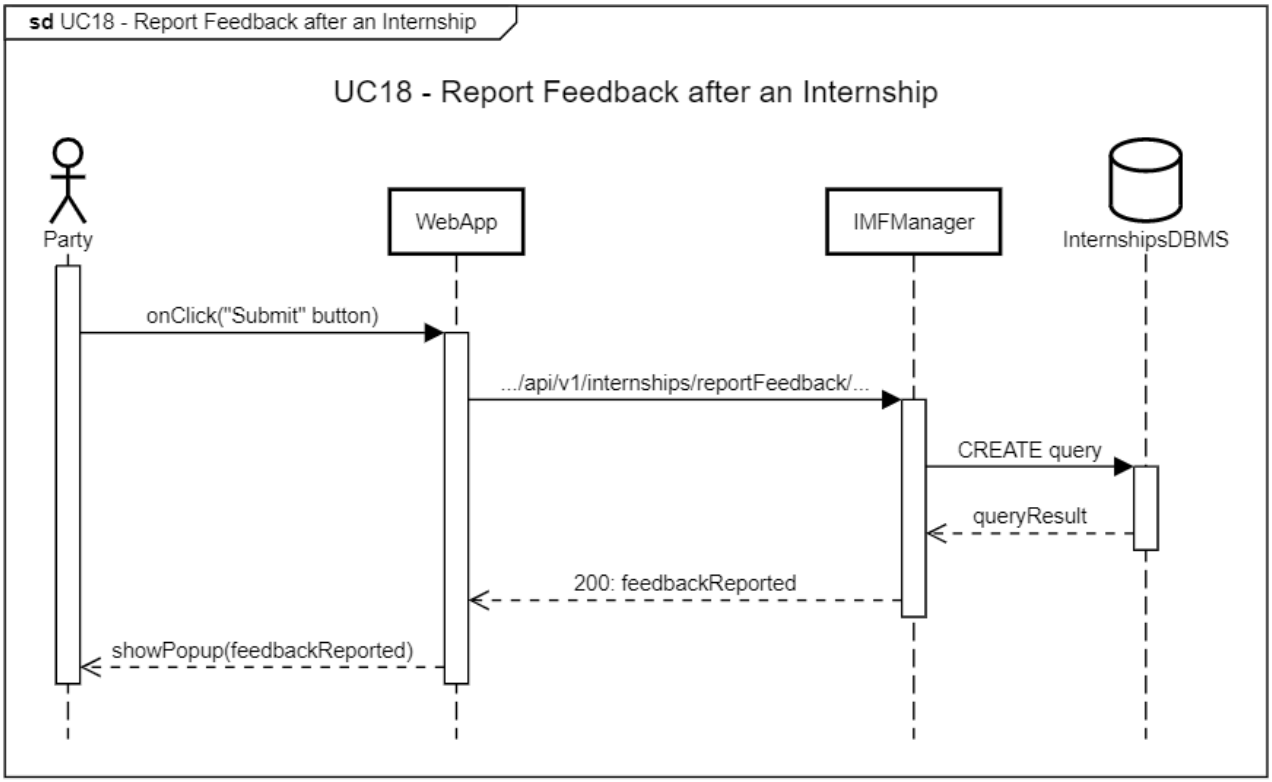

Figure 2.20: Report Feedback after an Internship

SD19. Suggest Optimizations for a Student Profile

The Student initiates the action by clicking the "Improve Profile" button on their profile page, triggering a request to the `.../api/v1/optimizations/optimizeStudent/...` API endpoint, which is managed by the `OptimizationManager`. The `OptimizationManager` retrieves relevant data from the `RecommendationsDBMS` and processes the information to generate optimization suggestions. Once completed, the `OptimizationManager` sends the suggestions back to the `WebApp`, which displays them to the student.

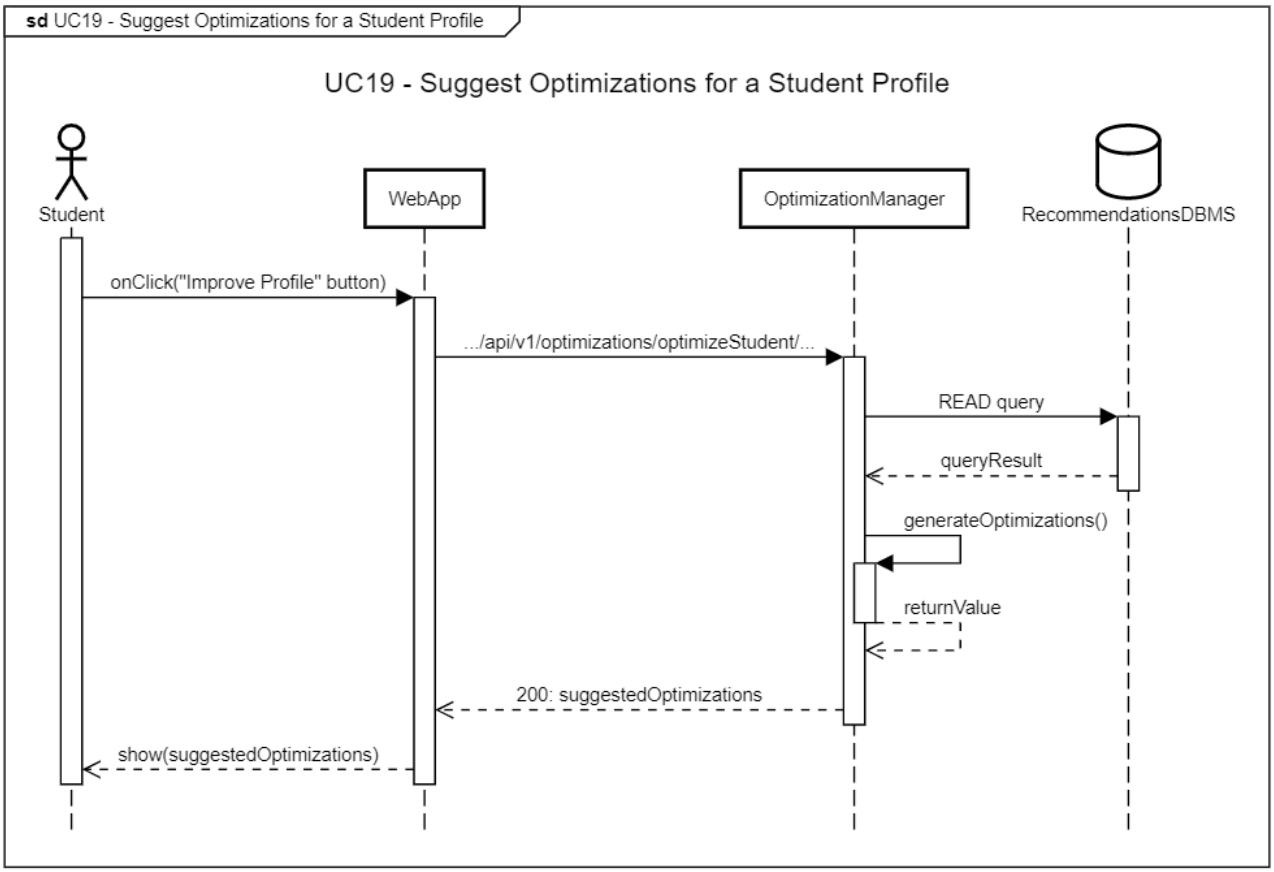

Figure 2.21: Suggest Optimizations for a Student Profile

SD20. Suggest Optimizations for an Internship Offer

This Sequence Diagram is the same as the previous one, with the only difference being that it is started by a Company and not by a Student, and thus is about optimizing an offer and not a profile.

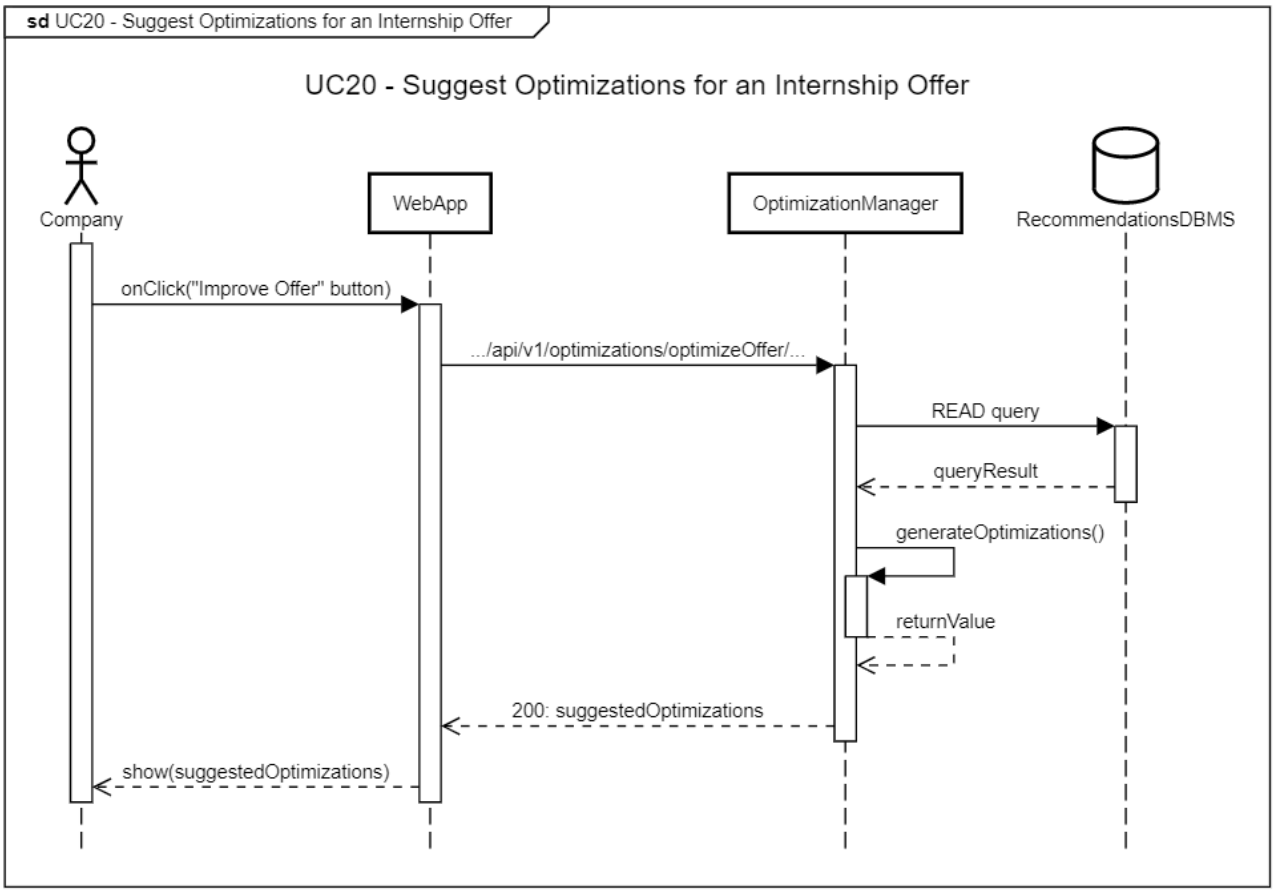

Figure 2.22: Suggest Optimizations for an Internship Offer

2.6. Selected Architectural Styles and Patterns

For the S&C system, various styles and patterns allow for better management of the key aspects of the system:

- **Database-per-service pattern:** When implementing a system as a microservices architecture, it is strongly advised to partition the domain data according to the bounded contexts identified, obtaining one separate database for each service [3]. This pattern comes with many advantages: independent scalability between the services, localized impact of schema evolutions or failures, and the possibility of using different technologies for distinct databases according to the needs of the specific domains. As a consequence, multiple databases are present in the S&C system: the SecurityDB, the ProfilesDB, the InternshipsDB and the RecommendationsDB: for instance, if one of these databases goes down, the other microservices are not impacted at all.
- **Functional partitioning:** As just stated, data is partitioned among the various databases according to the bounded context of each microservice: in order to reduce cross-service queries to the bare minimum, bounded contexts have been identified as the different macro-functionalities that the system is required to offer, therefore adhering to a functional partitioning model [4]. However, monitoring of the workload and distribution of data across all the databases should be constantly monitored, so that the database architecture can be redesigned as needed by performance issues, for example by adding horizontal partitioning (sharding) to the partitioning already in place.
- **Eventual consistency:** As the (in)famous CAP theorem states, it is impossible for a distributed system to achieve consistency, availability and partition tolerance (CAP) all at the same time [5]. Given the choice of structuring the S&C platform as a distributed system and the somewhat strict requirements on availability outlined in Section 3.5.2 *Availability* of the RASD, the system is designed to be available and partition-tolerant alone, settling on only the eventual consistency of data across the spread-out databases and replicas. As the S&C application is not handling critical processes, eventual consistency isn't really a problem [6], as it doesn't matter if information about any offer isn't immediately up-to-date, or if a recommendation is not generated instantly after a new offer becomes available, or if some recommendations do not take into account the absolute latest information available.
- **Event Sourcing pattern:** A common choice to achieve data replication across a cluster of databases is implementing an event-driven architecture, also known as publish-subscribe: changes in the data one of the replicas generate events that get sent to all the

other replicas. As stated before, tools based on the Raft protocol might be helpful: an example of such tools is Apache Kafka, which can be used as a channel to share the events between all the database copies. Kafka also supports multiple topics, enabling replication for different clusters of databases at the same time, and can also be configured to guarantee exactly-once semantics for ensuring events are not duplicated. Moreover, additionally to standard data replication, keeping data semantically separated comes with a cost, as some data is to be replicated across the different databases: this is the case for the RecommendationsDB, which is meant to gather data from the ProfilesDB and from the InternshipDB, representing something similar to what is called a **Materialized View**. As this can also be seen as a form of data replication, the standard Kafka distribution can also help: however, given that the involved databases are implemented with heterogeneous technologies, another suitable choice is to employ Kafka Connect with Change Data Capture (CDC), detecting changes in the databases by defining source connectors on the ProfilesDB and on the InternshipsDB and replicating them through a sink connector on the RecommendationsDB.

- **API Gateway pattern:** By introducing this pattern, the system establishes a single entry point for all client requests, unifying interactions with other components. Using NGINX, the system manages request routing to microservices and performs URL rewriting for necessary redirections. This approach enhances security by exposing only one public endpoint and optimizes internal traffic management. The benefits of adopting this pattern in our system include isolating clients from the internal partitioning of microservices, shielding them from the complexity of determining service instance locations, reducing the number of requests, and simplifying the client by shifting the logic for interacting with multiple services to the API Gateway [7].

- **Circuit Breaker pattern:** In order for graceful degradation of the system to be completely achieved, it is necessary to take full advantage of the microservices architecture by implementing the Circuit Breaker pattern in the API Gateway directing the communications. In this way, microservices don't block or suffer from other services' slowdowns or failures, as the "link" between them can be opened and closed at will according to their current status, so that each service can keep functioning and offering limited functionalities also when it isn't able to make requests to another one.

3 User Interface Design

3.1. General Overview

The design of the user interfaces (UI) for the S&C platform plays an important role in ensuring an engaging user experience. In the following section, through the use of possible UI interfaces, a comprehensive overview of the platform and its functionalities will be given, highlighting the principles and features that make the platform intuitive and accessible for its diverse user base.

The UI features a clean and consistent layout, obtained with uniform visual elements across pages and intuitive icons and labels, ensuring users can quickly find what they need and understand the differences between each section.

UI interfaces are not provided for every possible page described in the RASD & DD documents, but only for the main pages which include the majority of the functionalities. Further interfaces can be made following the style provided in these layouts. Design choices, colors, fonts, and other elements can be modified accordingly in order to follow strategic business decisions to better engage the users.

Each webpage shown is responsive and adapts to all types of devices.

3.2. User Interfaces Layouts

3.2.1. Home Page

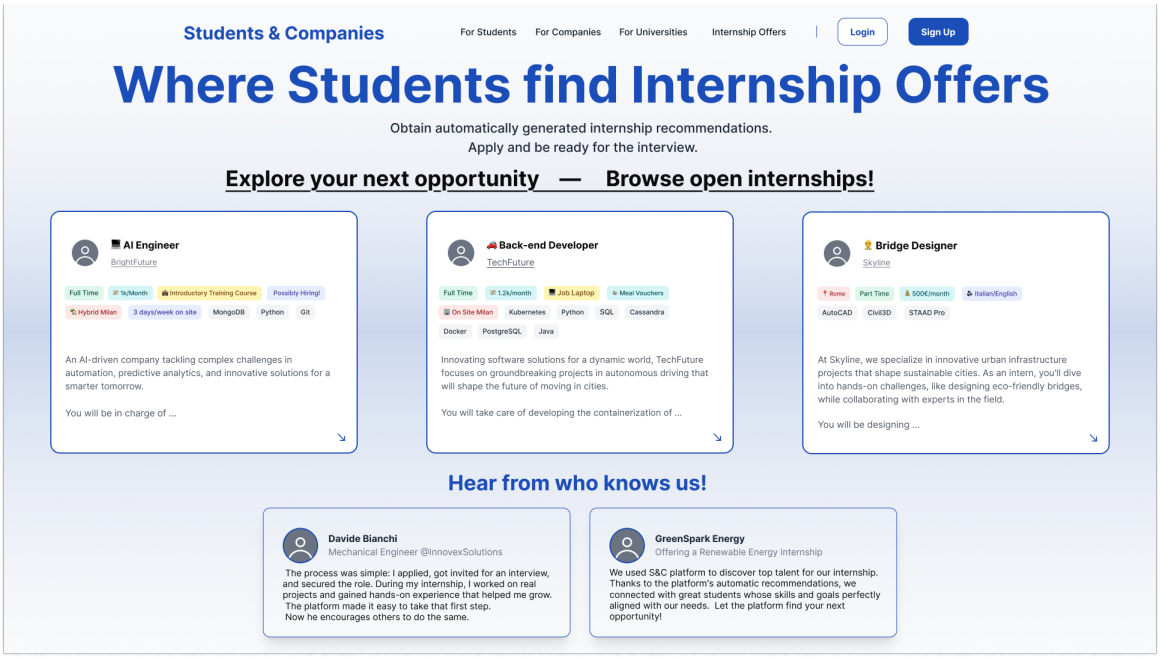

Figure 3.1: Landing Page of the WebApp

The Home Page of S&C is a multiple entry point for all the user groups. Its purpose is to guide students, companies, and universities to tailored content. The navigation bar contains links to "For Students", "For Companies", "For Universities", which redirects to custom landing pages. In this case, the page displayed is specifically made for students. The "Internship Offers" redirects to a browse-only carousel of internship offers, without any additional functionality prior to being registered. The middle section displays some of the active internship opportunities in an interactive grid layout that can be manually moved by the user or automatically updates the showcased content after a certain time. A testimonials section features true stories to build trust and credibility. The page can also be extended vertically and made scrollable, to include more content and optionally a footer.

3.2.2. Sign Up Page

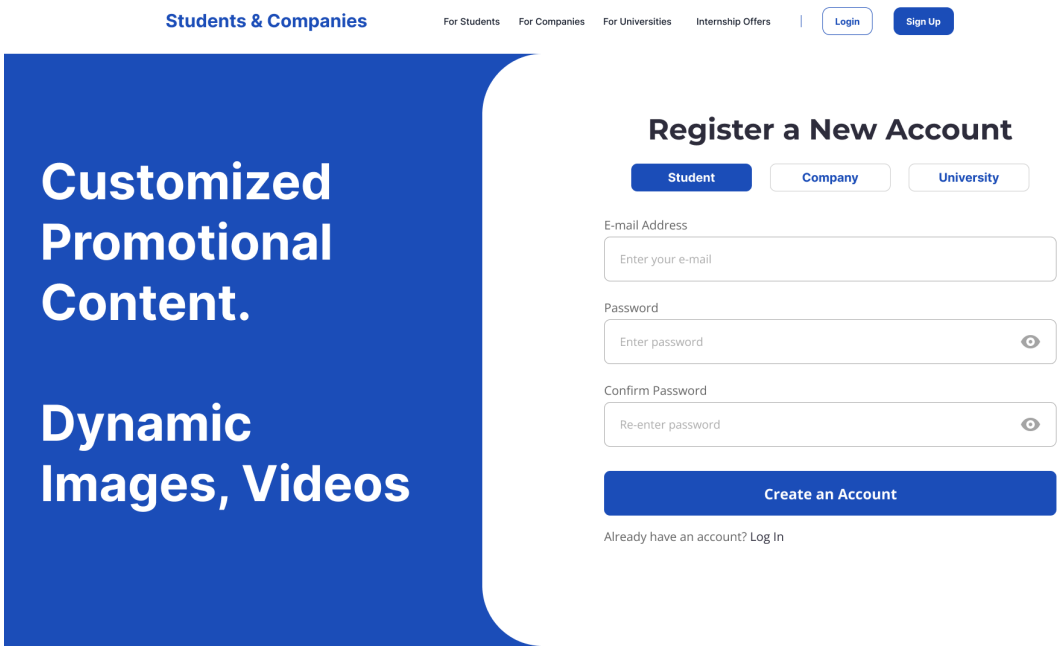

Figure 3.2: Sign Up - Initial Step

The sign up page contains the initial step of the registration processes for all user groups. Additional information required to customize each profile will be collected in a subsequent form, accessible after the link in the confirmation email is accepted. The navigation bar from the landing page remains consistently visible throughout the sign up process.

3.2.3. Student Dashboard

The student dashboard contains all functionalities available to students. It is divided into three distinct views, accessible via a dropdown menu integrated into the navbar, only visible once a student is logged in.

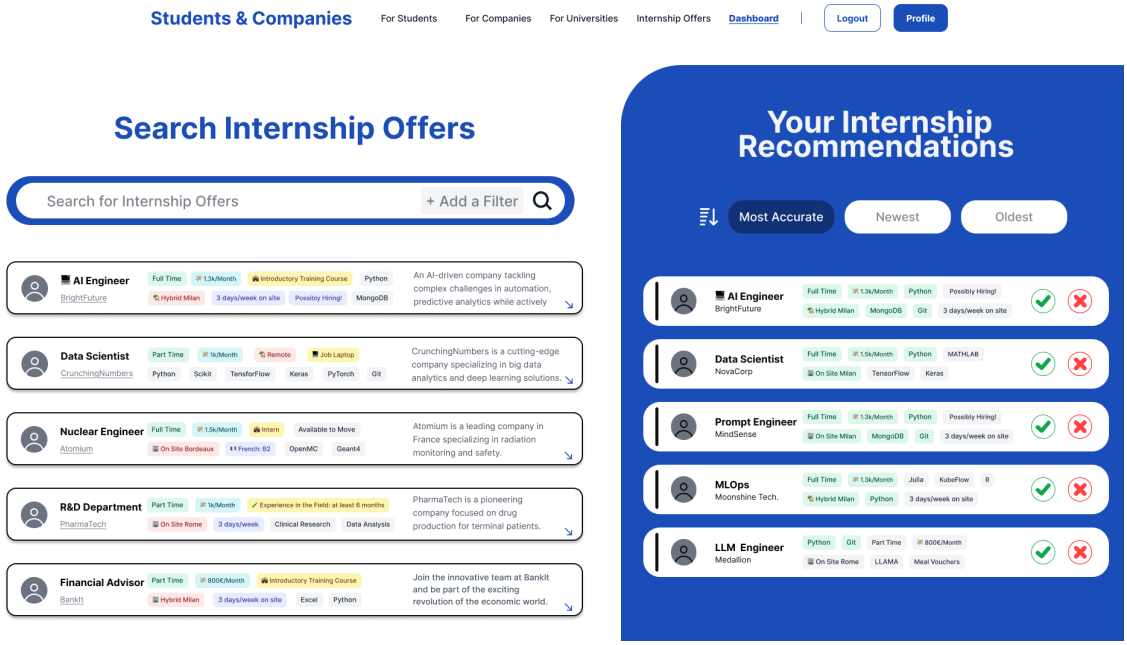

Figure 3.3: Student Dashboard - Main View

The main view, displayed by default whenever a student is redirected to the dashboard, is made by two sections: Search Internship Offers and Your Internship Recommendations.

On the left, the search functionality allows students to explore internship offers using a text-based search bar. The search can be further refined with predefined filters, such as keywords found in posted internship descriptions. If no input is provided, the system displays a random selection of internship offers. After initiating a search, the results appear in a vertical list layout. Each internship offer is an interactive element that opens a modal to display detailed information and includes the option to apply directly.

On the right, the Recommendations section shows the internship recommendations automatically generated by the system. Students can choose to accept or decline these recommendations and sort them based on predefined criteria, as indicated by the three intuitive sorting buttons. Each recommendation is interactive, allowing students to view offer details via a modal. As previously detailed in the document, students can apply directly through the internship offers or the recommendations with identical technical functionality.

Both sections are dynamic and adapt in real-time to user interactions.

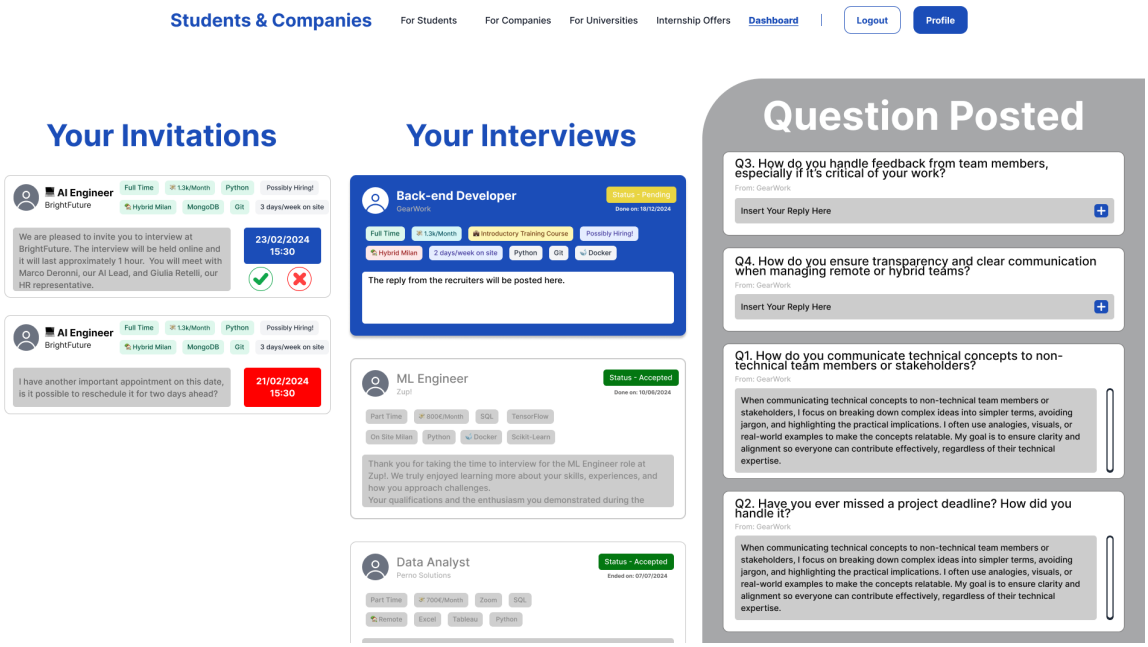

Figure 3.4: Student Dashboard - Second View

The second view, accessible through the drop-down menu, contains all the functionalities related to interviews and is divided into three sections: Your Invitations, Your Interviews, and a Message Board for questions related to the internships you have applied for.

On the left side, the Your Invitations section displays all pending, accepted, and refused interview invitations for internship offers where the interview process is still ongoing. Invitations are sorted chronologically, with the most recent ones appearing at the top. If a student declines an invitation, a modal will appear, asking to provide a reason or message to the hiring company.

The middle section, Your Interviews, lists all completed interviews. Each entry includes a summary of the position applied for, the interview date, and the current status of the application. When the company ends the process, feedback from the recruiter is displayed within each interview entry. Similar to the Invitations section, interviews are organized chronologically from top to bottom.

On the right side, the Message Board displays questions posted by companies conducting interviews for the internships you've applied to. Unanswered questions are shown at the top, while those you've already responded to appear below. Since questions from different companies may appear in a mixed layout, each question identifies the company that posted it. Alternatively, a filtering option by company can be added to obtain a separated view for each company. Once a selection process is completed, associated questions are automatically removed from the board.

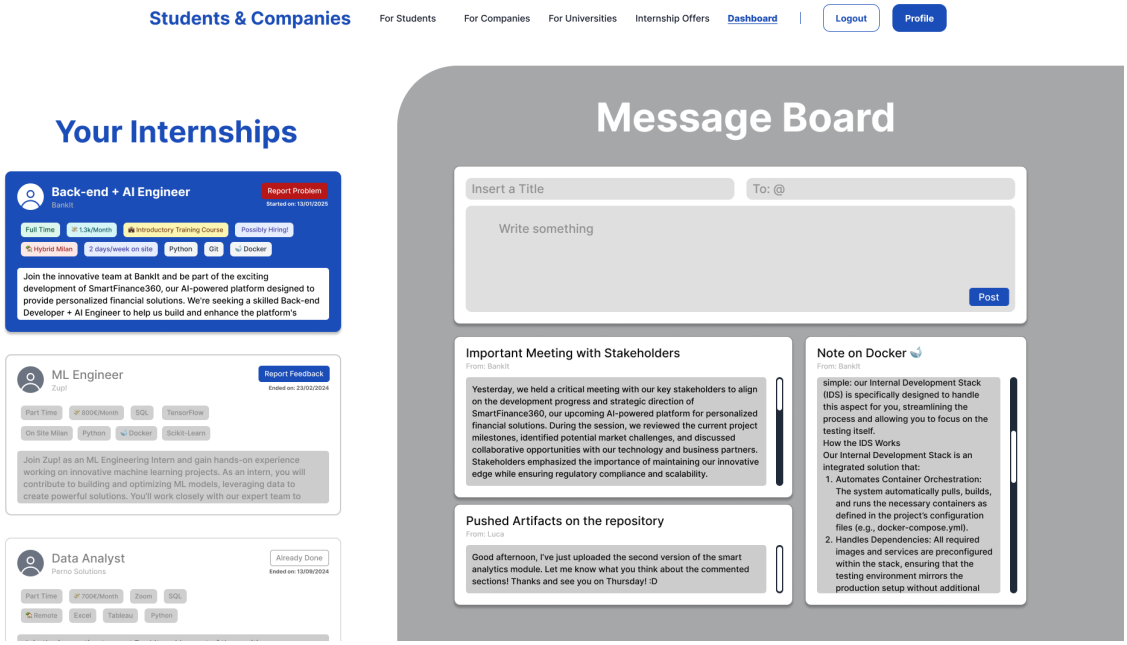

Figure 3.5: Student Dashboard - Third View

The third view, accessible through the dropdown menu, provides all functionalities related to ongoing and completed internships, organized into two sections: Your Internships and a Message Board for communication with the company.

On the left, the Your Internships section displays details of both ongoing and completed internships. Each ongoing internship includes a summary of the applied position, the starting date, and a "Report Problem" feature that initiates a process managed by the university to address issues. Completed internships are also listed here, with the option to provide feedback if it has not already been submitted. As in other sections, internships are sorted chronologically, with the most recent at the top.

On the right, the Message Board provides a dedicated space for displaying all communications between the student(s) and the company. At the top, students can post new messages, while previous messages are displayed below in chronological order. Older messages can be accessed by scrolling further down the section, ensuring all messages remain accessible.

3.2.4. Profile Page - Student

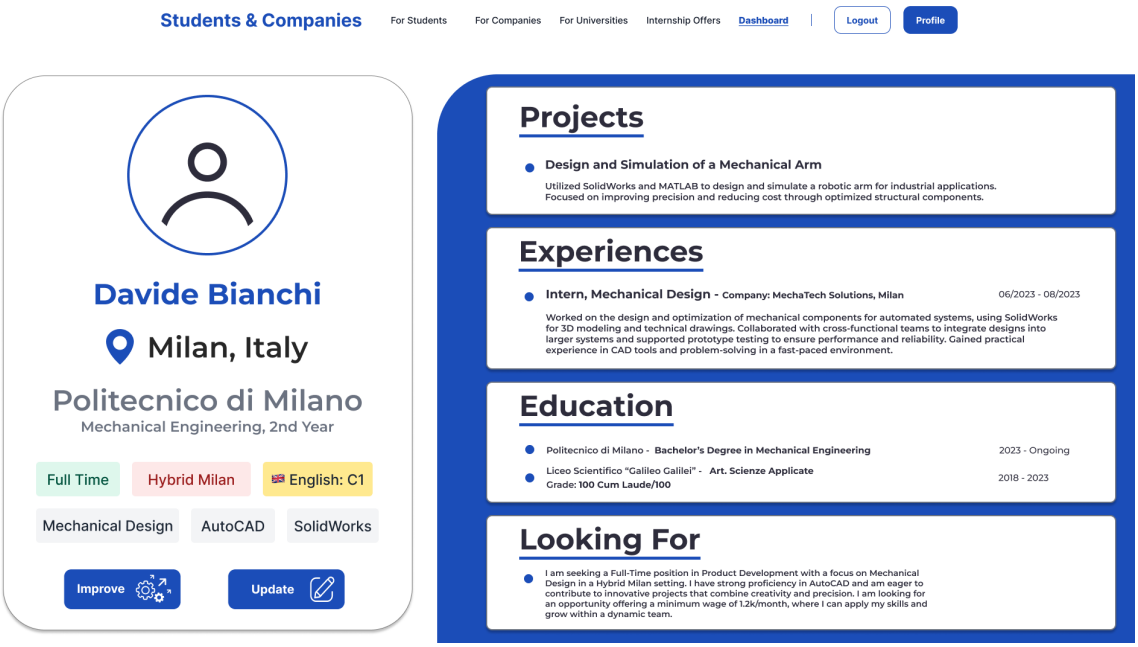

Figure 3.6: Profile Page - Student

The Profile page, accessible through the navigation bar, contains all information related to the student.

On the left, a summary of the student's profile is displayed, highlighting the most relevant details along with a subset of keywords extracted from their complete profile. Below this summary, two key actions are available: Suggest Optimizations for a Student Profile, which provides recommendations to enhance the profile, and Update Profile, allowing students to edit their information.

On the right, the main subsections of the curriculum are presented. These sections are used to identify and extract keywords that define the student's keywords.

Other profile pages expected in the platform, for Company and University, are similar in the general layout and only differ in the content shown, and so for this reason their UI is not provided in this document.

3.2.5. Company Dashboard

The company dashboard contains all functionalities available to companies. It is divided into three distinct views, accessible via a dropdown menu integrated into the navbar, only visible once a company is logged in.

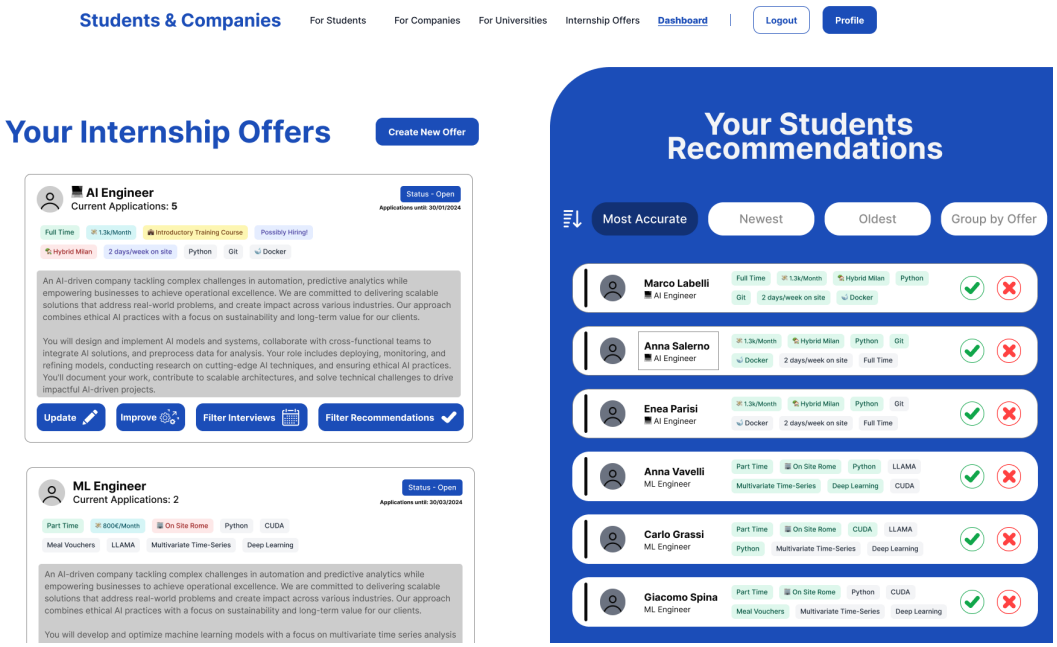

Figure 3.7: Company Dashboard - Main View

The main view, displayed by default whenever a company is redirected to the dashboard, is composed of two sections: Your Internship Offers and Your Students Recommendations.

On the left, all open internship offers are listed, displaying the description provided during posting and relevant keywords extracted from the description. To perform the insertion of a new internship offer on the platform, companies can use the associated button, which opens a modal containing a form with all the required fields to be compiled. Each internship is an interactive element that can be expanded into a modal for detailed visualization, including its description and a list of applicants. Below each internship, there are further options: one to edit the offer's details or eventually withdraw it and another to receive suggestions for improving its content. A Filter Interviews button allows navigation to the Interviews section of the dashboard, focusing exclusively on interviews and invitations for that specific offer. By default, the Interviews section displays invitations and interviews for all internships currently in the Selection phase. Additionally, a Filter Recommendations button performs a similar action, filtering the Recommendations section (on the right) to suggestions relevant to the selected internship.

On the right, the Recommendations section shows the student recommendations automatically generated by the system. Companies can accept or decline these recommendations and sort them using four intuitive sorting buttons. Each recommendation highlights the compatibility between internship offer keywords and student profiles. Recommendations are interactive, enabling companies to view detailed student profiles through a modal.

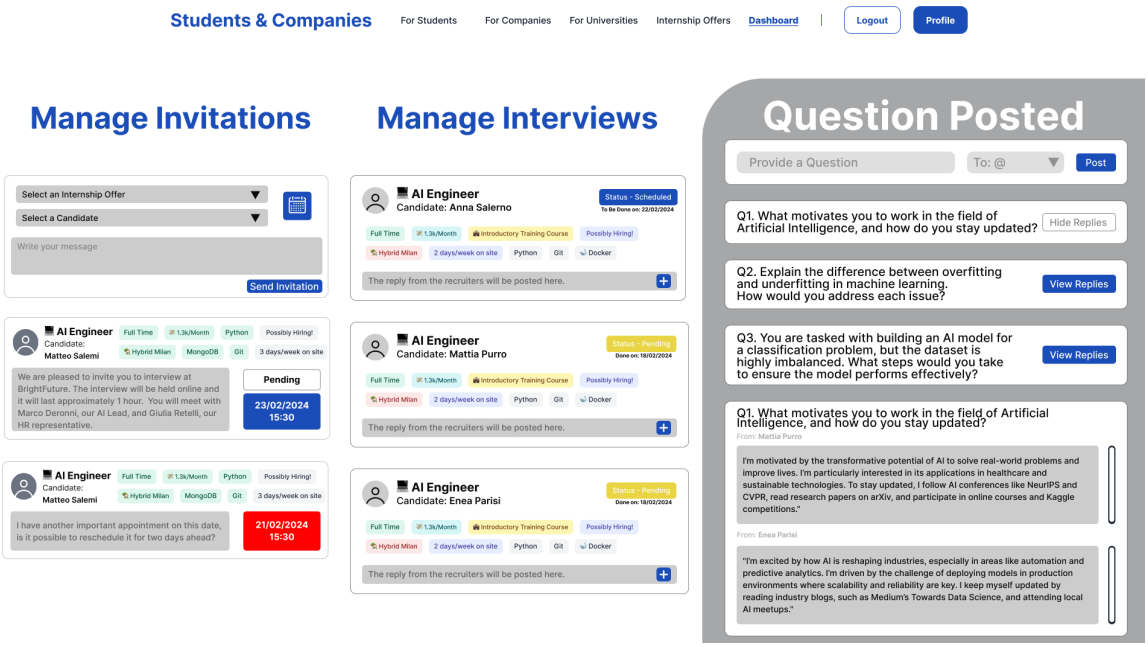

Figure 3.8: Company Dashboard - Second View

The second view, accessible via the drop-down menu or the previously mentioned shortcut, contains all functionalities related to interviews. It is divided into three sections: Manage Invitations, Manage Interviews, and a Message Board for providing questions to candidates during interviews.

On the left, companies can create new invitations by selecting the associated internship offer and choosing a candidate from a dropdown menu, specifying a proposed date and time, and providing a description. Below this, all refused and pending invitations are displayed, while accepted invitations automatically move to the middle section.

The middle section shows all scheduled interviews, sorted chronologically. Here, companies can provide feedback on each interview, which includes a status field that can be updated by clicking on it to finally accept or decline the candidate.

On the right, the Message Board allows companies to post new questions to candidates, selecting one or multiple candidates through a dropdown menu. Companies can also monitor replies to previous questions, with options to view or hide specific questions using dedicated buttons.

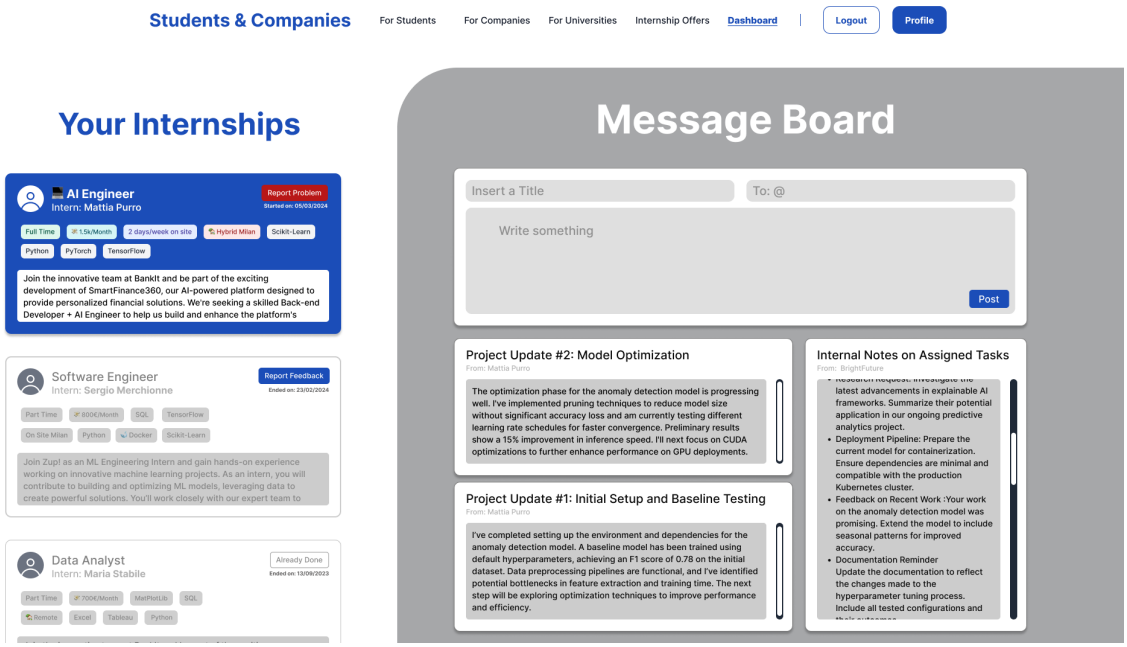

Figure 3.9: Company Dashboard - Third View

The third view, accessible through the dropdown menu, provides all functionalities related to ongoing and completed internships, organized into two sections: Your Internships and a Message Board for communication with the students.

On the left, the Your Internships section displays details of both ongoing and completed internships. Each ongoing internship includes a summary of the applied position, the starting date, and a "Report Problem" feature that initiates a process managed by the university to address issues. Completed internships are also listed here, with the option to provide feedback if it has not already been submitted. As in other sections, internships are sorted chronologically, with the most recent at the top.

On the right, the Message Board provides a dedicated space for displaying all communications between the company and the student(s). At the top, companies can post new messages, while previous messages are displayed below in chronological order. Older messages can be accessed by scrolling further down the section, ensuring all messages remain accessible.

3.2.6. University Dashboard

The university dashboard contains all functionalities available to universities. It is composed of a minimal view, accessible via the dashboard link integrated into the navbar, only visible once a university is logged in.

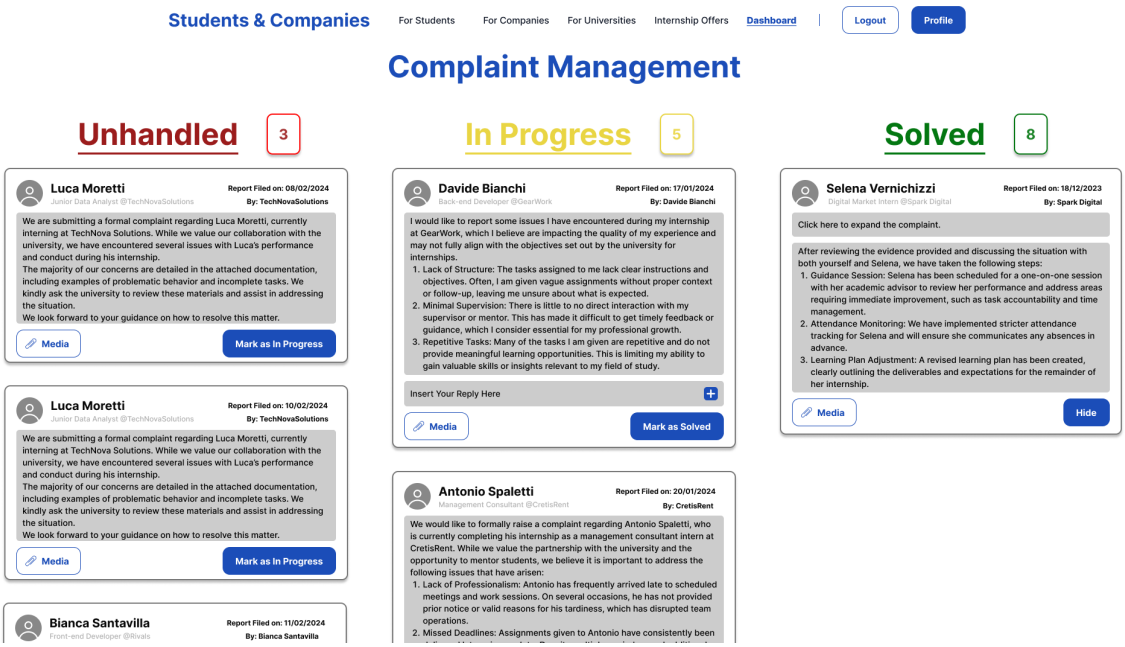

Figure 3.10: University Dashboard - Complaint Management View

The default view, displayed whenever a university is redirected to the dashboard, consists of three sections that form the Complaint Management Dashboard. This dashboard enables the university to execute the "Handle Problems" functionality for their students.

The layout is intuitive, featuring a three-column pipeline that organizes complaints based on their status: Unhandled, In Progress, or Solved. Complaints are displayed in reverse chronological order within each column, with the oldest reports at the top and the newer ones below.

Each report includes comprehensive details about the student, their role, the date the report was filed, and the reporting party. A description of the issue is presented in a dedicated text section, and any attached media can be accessed by clicking the Media button.

When the status of a report changes, it automatically moves to the corresponding adjacent column. Once a report is marked as solved, the university has the option to hide it from the dashboard view to reduce clutter.

To improve focus, the university can hide any of the three columns by clicking the corresponding label. The remaining columns will dynamically adjust to occupy the available space equally.

4 Requirements Traceability

This chapter shows how the functional and non-functional requirements of the S&C system described in the RASD are mapped to each component. First, we describe the mapping between the requirements listed in Section 2.2.1. *Requirements* of the RASD document and the components identified in the Component Diagram section of the DD document. Then, we comment on how Sections 3.3 *Performance Requirements* and 3.5 *Software System Attributes* of the RASD document are achieved through the decisions taken for the S&C system.

4.1. Functional Requirements Traceability

The following table lists all the requirements linked within each component.

MailingService	[R1] Upon request, the system shall allow the User to sign up to the platform, as long as they submit all the required information, they don't already have a profile in the platform and their identity and role (Student, Company or University) are verified.
SecurityManager	[R1] Upon request, the system shall allow the User to sign up to the platform, as long as they submit all the required information, they don't already have a profile in the platform and their identity and role (Student, Company or University) are verified. [R2] Upon request, the system shall allow the requesting User to log in to the platform, granting him access to their profile as long as their authentication is successful.
ProfileManager	[R3] Upon request, the system shall allow the requesting User to update their profile, as long as they provide all the necessary information.
OfferManager	[R4] Upon request, the system shall allow a Company to publish a new internship offer, as long as it provides all the required information and the latter is compliant with platform guidelines. [R5] Whenever a Company publishes an internship offer, the system shall add it to the list of all the internship offers.

	[R6] Upon request, the system shall allow the requesting Company to update information for any of their open internship offers, as long as it provides all the necessary information. [R7] Upon request, the system shall allow the requesting Company to withdraw any of their open internship offers. [R8] Upon request, the system shall allow a Student to search for desired internship offers by applying optional filters to the list. [R9] Whenever receiving a list of filter attributes for searching internship offers, the system shall return the list of all the offers matching the selected criteria. [R10] Upon request, the system shall allow the requesting Student to apply to an internship offer, as long as that offer's application deadline has not expired. [R11] Whenever a Student applies for an internship offer, the system shall add them to the list of candidates for that offer. [R14] Whenever an internship offer is withdrawn by its publishing Company, the system shall discard all applications to that offer. [R23] Whenever a Student accepts one of their received recommendations, the system shall apply the requesting Student to the internship offer to which the recommendation refers.
InterviewManager	[R25] After the application deadline of an internship has expired, the system shall allow the publishing Company to contact a Student who had previously applied to that offer in order to plan a future interview with them, if none has been planned yet. [R26] Whenever a Student receives an interview proposal, the system shall allow that Student to either accept it or refuse it by providing a reason.

	[R27] Whenever an interview has to be carried out in-platform, the system shall allow the interviewing Company to submit questions to the Student involved. [R28] Whenever a Company submits questions to a Student for an in-platform interview, the system shall allow that Student to answer those questions, reporting them to the interviewing Company. [R29] Upon request, the system shall allow a Company to evaluate the answers received from a Student in one of their interviews, by registering that interview's result. [R30] Whenever a Company evaluates an interview (both in-platform and in-person), the system shall inform the corresponding Student of the registered outcome. [R31] Whenever the interview results for all the candidates for an internship offer have been registered into the platform, the system shall close the selection process of that offer.
InternshipMonitoringAndFeedbackManager (IMFM)	[R32] Upon request, the system shall allow the requesting Party to provide new information about any of the ongoing internships in which that Party is involved. [R33] Upon request, the system shall yield to the requesting Party all the information about one of the ongoing internships it is involved in. [R34] Upon request, the system shall allow the requesting Party to report a problem occurring in one of the ongoing internships it is involved in. [R35] When receiving a problem report about an internship from a Party, the system shall forward it to the University of the Student involved in that internship.

[R36] Upon request, the system shall allow the requesting University to handle a received problem regarding an ongoing internship in which one of its Students is taking part.

[R37] Upon request, the system shall allow the requesting Party to report feedback about an internship in which it has been actively involved, if that internship has been completed.

[R38] Whenever receiving feedback about a completed internship, the system shall process it in order to improve the process for generating recommendations for the future.

RecommendationManager

[R12] Whenever a Student applies for an internship offer, the system shall mark all the "Unhandled" recommendations of that Student about that internship offer as "Accepted".

[R13] Whenever a Student applies for an internship offer, the system shall discard all the "Unhandled" recommendations of the Company offering it about that Student in the context of that offer.

[R15] Whenever an internship offer is withdrawn by its publishing Company, the system shall discard all generated recommendations linked to that offer.

[R16] Whenever a recommendation aimed at a Party is generated, the system shall add that recommendation to that Party's profile, as long as there is not another "Unhandled" recommendation about the other Party in the context of the same offer.

[R17] Whenever a new Student signs up to the platform, the system shall generate, for every internship offer matching that Student's data, a recommendation about them aimed at the Company advertising that offer, as long as the latter's application deadline has not expired.

[R18] Whenever a Student updates their profile, the system shall generate, for every internship offer matching that Student's updated data, a recommendation about them aimed at the Company advertising that offer, as long as the latter's application deadline has not expired.

[R19] Whenever a Company publishes a new internship offer, the system shall generate, for every Student matching with that internship offer's data, a recommendation about it aimed at that Student.

[R20] Whenever a Company updates data for an internship offer, the system shall generate, for every Student matching with that internship offer's updated data, a recommendation about it aimed at that Student.

[R21] Whenever an internship offer is withdrawn by its publishing Company or its application deadline expires, the system shall discard all the recommendations about it, regardless of whether they have been accepted or not.

[R22] Upon request, the system shall allow the requesting Party to manage their received recommendations by accepting or refusing them, if those have not already expired.

[R23] Whenever a Student accepts one of their received recommendations, the system shall apply the requesting Student to the internship offer to which the recommendation refers.

[R24] Whenever a Company accepts one of their received recommendations, the system shall generate a symmetric recommendation to the corresponding Student and add it to the latter's list of recommendations, as long as the generated recommendation is not already present in it.

OptimizationManager	[R39] Upon request, the system shall provide a Student with targeted suggestions for optimizing its profile, enabling the Student to improve its appeal and relevance for obtaining more internship offers in the future, if such optimizations can be found. [R40] Upon request, the system shall provide a Company with targeted suggestions for optimizing a selected internship offer, enabling the Company to make it more attractive to Students and to improve its visibility for the future, if such optimizations can be found.

Table 4.1: Mapping between Components and Requirements.

4.2. Non Functional Requirements Traceability

4.2.1. Performance Requirements

The performance requirements outlined in Section 3.3 *Performance Requirements* of the RASD document are addressed through some design decisions (already discussed in Section 2.6) and the adoption of scalable and efficient technologies in the S&C system. Below, we explain how they are achieved:

- **Handling 10,000 concurrent users:** The scalability of the system is ensured by its distributed 4-Tier Architecture and containerized microservices. The container orchestration platform dynamically manages the allocation of resources based on workload, ensuring that at least up to 10,000 concurrent users can interact with the system without significant performance degradation. Furthermore, the microservices architecture allows for horizontal scaling, enabling the system to handle increased user loads if required by future business decisions.
- **Recommendations accuracy and scalability:** The recommendation system is implemented as a dedicated microservice, relying on a machine learning model, that should guarantee both an accuracy and F1 score of at least 0.8, to be previously verified during the testing phase and continuously monitored when the system will be running. Its design enables batch processing of data and scaling to handle up to 1,000,000,000 unique user profiles efficiently, by actively operating on the corresponding data storages.
- **Low response time for requests ($<$ 5 seconds)** The system offers fast response times by employing RESTful APIs and efficient load balancing in all the involved layers. Requests are distributed evenly across server instances of Application and Data Layer. For computationally intensive processes such as recommendations, asynchronous handling and caching of previous queries ensure prompt delivery of results to the user.
- **Real-time data updates ($<$ 0.01 seconds)** The introduction of a distributed cache in the Data Layer should guarantee low-latency access to updated information, when retrieving already fetched data.

4.2.2. Software System Attributes

The software system attributes outlined in Section 3.5 *Software System Attributes* of the RASD document have already been hinted at several times in the context of this Design Document. Below, we explain how they are achieved:

- **Reliability:** The replication of the services, accomplished through the containerization approach, and of the databases, carried out with the Event Sourcing pattern, makes the system highly scalable: new instances of the microservices can be dynamically deployed as needed by the current requests load, basically guaranteeing that the Quality-of-Service (QoS) remains stably high throughout the system's lifecycle, unless exceptional circumstances happen. In addition, another consequence of the (geographic) distribution and replication of nodes is that the system becomes greatly fault-tolerant, which means that it is able to resist service and network failures to a certain extent, without necessarily interrupting the provision of functionalities and achieving graceful degradation.
- **Availability:** The system offers the availability previously declared in the RASD (two-nines) through the use of replicated databases, container orchestration for dynamic scaling, and a load balancer that distributes incoming requests evenly across servers so that they can be managed in parallel, therefore reducing the risk of downtime and ensuring consistent performance under varying workloads. Further, the choice of adopting eventual consistency reduces the overhead in synchronizing the databases, so that they can fulfill incoming requests more readily.
- **Security:** Authentication and authorization mechanisms relying on token-based access control, a web server acting as an external entry point for filtering and managing incoming requests, and encrypted communication via HTTPS, collectively ensure a secure system.
- **Maintainability:** The adoption of the microservices architecture makes the system extremely modular, and the resulting separation of concerns enables the possibility to make updates to a service without completely affecting or bringing down the other ones. Moreover, thanks to containerization, it is possible to deploy additional services offering new functionalities as needed, without impacting the already running system.
- **Portability:** As previously discussed in the RASD document, since the application is offered in the form of a WebApp, it inherently supports compatibility across various operating systems. Considering the architectural aspects of the system, as the platform is deployed through containers and the communication is mainly made through standard and technology-neutral protocols, it can be easily migrated regardless of the underlying infrastructure.

5 | Implementation, Integration and Test Plan

5.1. Overview

This section describes how the overall development process for building the system shall be carried out. Having three separate plans for implementation, integration and testing wouldn't be effective, because the three phases can (and should) be greatly intertwined to maximize efficiency in the development process: thus, a single, chronological plan is delineated.

In fact, the decision to design the system by following the microservices architecture makes it possible to split its functionalities among different units, enforcing the separation of concerns between them and allowing the developers to implement and test each of them separately and in parallel. However, this also means that the different subsystems shall be carefully integrated together after their completion, and consequently extensive integration testing shall be performed.

5.2. Development Plan

The development shall focus on ensuring that the most impactful and business-critical components are implemented first. The functionalities that set this platform apart from the other similar available products are recommendations and optimizations, located in the RecommendationManager and OptimizationManager components respectively: due to this consideration, the highest priority and a greater amount of development and testing resources are to be devoted to these components. Indeed, their implementation involves perfecting the Artificial Intelligence algorithms and models outlined in Section 2.2.2 to generate recommendations and optimizations, and therefore requires thorough analyses to guarantee the correctness of the training and fine-tuning processes for models, because these are quite resource-consuming (in terms of both time and computation power) and, consequently, they should be iterated as few times as possible.

The platform's other functionalities are offered by the remaining microservices, each responsible for a specific subset of tasks. This modular design enables taking full advantage of the modularity and parallelizability offered by the microservices architecture, allowing teams to work on them simultaneously without being tightly coupled to the progress of other components. Such an approach is particularly efficient, as every requirement is almost always mapped to only one component. Each microservice rarely communicates with the others, only in limited cases to offer particular functionalities, such as authorization and recommendation generation,

and they do so only through a standard interface that is the REST API. For this reason, the implementation should proceed without any interruptions or dead times.

Units within the same component are to be implemented and tested following a bottom-up strategy: that is, development shall start by building the basic units, and then these shall be progressively put together to form the whole component, taking advantage of the fact that this strategy doesn't require the developers to write complex stubs. External calls from other components shall be simulated in a standard and decoupled way by invoking the corresponding REST API endpoint: as far as it concerns unit testing, only the correctness of the calls and their inputs shall be validated, also applying mutational fuzzing techniques from valid URIs with coherent parameters.

After components are complete and working, they can be integrated together. Integration testing shall be carried out by considering the pairs of components that interact together according to the Component View in Section 2.2.1: they shall launch the expected calls to one another's API endpoints, in order to verify that the contract of each exposed interface is correctly interpreted by them both.

System (E2E) testing shall start immediately after integration testing, in order to ensure consistency with the requirements defined in the RASD, and validation should also be assessed by stakeholders. System testing shall be split among functional testing, aimed at straightforwardly evaluating the satisfaction of the functional requirements, and performance testing, for the remaining non-functional requirements. The latter shall verify the system's capability to withstand increasingly heavy loads (Load Testing, for requirements such as availability and scalability) and to achieve graceful degradation in case of the failure of a subset of the microservices offered (Stress Testing).

6 Effort Spent

Group Member	Effort Spent in each Section	
Riccardo Piantoni	Introduction	$0h$
	Architectural Design	$17.5h$
	User Interface Design	$0h$
	Requirements Traceability	$0.5h$
	Implementation, Integration and Test Plan	$1.5h$
	Reasoning	$7.5h$
Matteo Rossi	Introduction	$1h$
	Architectural Design	$5h$
	User Interface Design	$17.5h$
	Requirements Traceability	$1.5h$
	Implementation, Integration and Test Plan	$0.5h$
	Reasoning	$3h$
Jacopo Sacramone	Introduction	$0h$
	Architectural Design	$15.5h$
	User Interface Design	$2h$
	Requirements Traceability	$0h$
	Implementation, Integration and Test Plan	$0h$
	Reasoning	$7h$

Table 6.1: Effort spent by each member of the group.

7 References

7.1. Used Tools

- **Overleaf**: LaTeX collaborative writing.
- **SequenceDiagram.com with IntelliJ IDEA plugin**: sequence diagrams.
- **Lucidchart**: all other diagrams.
- **Figma**: UI Interfaces.

List of Figures

List of Tables