POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# ITD

## Computer Science and Engineering

**Github**:https://github.com/giovanni-vaccarino/VaccarinoPalladinoVacis
Installation instructions in chapter 6

Author(s): **Giovanni Vaccarino**
**Vittorio Palladino**
**Nicolò Vacis**

# Contents

# 1 Introduction

## 1.1. Scope

The following document has the key purpose of providing a detailed description on how the implementation and testing of the S&C platform has been performed, focusing on the following sections:

- **Implemented Functionalities:** In this section it will be listed the successfully implemented functionalities of the platform, making a mapping with the requirements stated in the RASD document.
- **Adopted Development Frameworks:** In this section there will be discussed the technology stack chosen, providing its advantages and disadvantages.
- **Source Code Structure:** In this section an insight of the code structure will be provided.
- **Testing:** In this section it will be discussed the testing part.
- **Installation Instructions:** This section provides detailed instructions for running the S&C platform.

## 1.2. Acronyms

- **S&C**: Students&Companies platform

## 1.3. Revision History

| Version | Date | Description |
|---|---|---|
| 1.0 | February, 2 2025 | Initial release |

# 2 | Implemented Requirements

**[R1]: The S&C shall allow users to log in to their accounts using their registered email and password.**
*This requirement has been implemented.* Users can successfully log in to their accounts with their email and password.

---

**[R2]: The S&C shall allow users to reset their password via a "Forgot Password" feature, with verification sent to their registered email.**
*This requirement has been implemented.* Users can reset their password via the "Forgot Password" feature, and the system sends a verification email.

---

**[R3]: The S&C shall allow users to log out securely from their accounts.**
*This requirement has been implemented.* Users can securely log out from their accounts without issues.

---

**[R4]: The S&C shall allow users to register on the platform.**
*This requirement has been implemented.* Users can create accounts on the platform.

---

**[R5]: The S&C shall allow students to edit profiles and upload CV.**
*This requirement has been implemented.* Students can edit their profiles and upload their CVs successfully.

---

**[R6]: The S&C shall allow students to edit personal details, skills, and interests in their profiles.**
*This requirement has been implemented.* Students can update personal details, skills, and interests in their profiles.

---

**[R7]: The S&C shall allow students to submit assessments, including answering open-ended, true or false, and multiple-choice questions.**
*This requirement has been implemented.* Students can submit assessments with both open-ended and multiple-choice questions.

---

**[R8]: The S&C shall provide students with an internship search tool that includes filtering options for location, job title, and publication date.**
*This requirement has been implemented.* Students can search for internships with filters for location, job title, and publication date.

---

**[R9]: The S&C shall allow students to get tailored internship suggestions based on their interests and skills.**
*This requirement has been implemented.* Students receive tailored internship suggestions based on their specified interests and skills.

---

**[R10]: The S&C shall allow the recommendation engine to improve over time based on feedback from students and companies.**
*This requirement has been implemented.* The recommendation engine improves over time using feedback from students and companies.

---

**[R12]: The S&C shall allow real-time tracking of application statuses, showing students the progress of applications at each stage.**
*This requirement has been implemented.* Students can track their application statuses in real-time, viewing progress at each stage.

---

**[R13]: The S&C shall allow companies to view submitted applications.**
*This requirement has been implemented.* Companies can access and view the applications submitted by students.

---

**[R14]: The S&C shall allow companies to accept or reject candidates directly from the submitted applications.**
*This requirement has been implemented.* Companies can accept or reject candidates directly from the submitted applications.

---

**[R15]: The S&C shall allow companies to create internship postings, specifying job roles and required skills.**
*This requirement has been implemented.* Companies can create detailed internship postings specifying job roles and required skills.

---

**[R16]: The S&C shall allow companies to edit existing internship postings after creation.**
*This requirement has* **not** *been implemented.*Companies cannot edit their existing internship postings after they have been created.

---

**[R17]: The S&C shall allow companies to have a dashboard to review candidate applications.**
*This requirement has been implemented.* Companies can use the dashboard to review.

---

**[R18]: The S&C shall allow companies access to a comprehensive page to track applications status.**
*This requirement has been implemented.* Companies have access to a detailed page for tracking application statuses.

---

**[R19]: The S&C shall allow companies to have a recommendation for candidates based on internship requirements.**
*This requirement has been implemented.* Companies have a recommendation for candidates relating to one of their internships, based on the internship requirements.

---

**[R20]: The S&C shall allow companies to access a "Potential Candidates" feature to view potential matches based on student profiles.**
*This requirement has been implemented.* Companies can use the "Potential Candidates" feature to find student matches.

---

**[R21]: The S&C shall allow both students and companies to provide feedback, covering overall satisfaction.**
*This requirement has been implemented.* Both students and companies can provide overall satisfaction feedback.

---

**[R22]: The S&C shall allow companies to view feedback provided for candidates from previous internships.**
*This requirement has been implemented.* Companies can access feedback for candidates from previous internships.

---

**[R23]: The S&C shall allow students to view feedback provided for companies from previous interns.**
*This requirement has been implemented.* Students can view feedback for companies given by previous interns.

**[R25]: The S&C shall allow a reporting feature for users to report inappropriate or irrelevant internship environments, ensuring a safe and professional place.**

*This requirement has been implemented.* Users can report inappropriate or irrelevant internship environments, from the feedback feature.

# 3 | Adopted Frameworks

## 3.1. Technology Stack Selection

In this section, the rationale for choosing the specific frameworks and languages for the backend and frontend development of the job-boarding platform is discussed.

### 3.1.1. Backend: ASP.NET Core with C#

The backend framework chosen for the application is ASP.NET Core, utilizing the C# programming language. The primary reasons for this choice are outlined below:

- **Cross-Platform Support:** ASP.NET Core is a modern, cross-platform framework that enables the development of web applications on Windows, Linux, and macOS, ensuring deployment flexibility.
- **Performance:** ASP.NET Core is known for its exceptional performance and speed, which are critical for handling concurrent requests in a scalable job-boarding platform.
- **Robust Ecosystem:** The .NET ecosystem offers a wide range of libraries, tools, and integrations, which accelerates development and reduces the need for third-party dependencies.
- **Built-In Pattern Implementations:** ASP.NET Core provides easy access to widely adopted software patterns such as Dependency Injection (DI) and Mediator, making it straightforward to implement clean architecture principles and maintain a modular codebase.
- **Security:** ASP.NET Core includes built-in features for managing authentication, authorization, and data protection, making it easier to build secure applications.
- **Community and Support:** ASP.NET Core has a large and active community, ensuring access to extensive documentation, tutorials, and third-party tools.

**Disadvantages:**

- **Learning Curve:** For developers unfamiliar with .NET Core, the framework can have a steep learning curve, especially when adopting patterns such as DI and Mediator.
- **Overhead for Small Projects:** The robust ecosystem and extensive features may

introduce unnecessary overhead for simpler applications.

### 3.1.2. Frontend: React with TypeScript

For the frontend, the chosen framework is React, combined with TypeScript. The reasons for this selection are as follows:

- **Component-Based Architecture:** React's component-based design simplifies the creation of reusable and modular UI components. This feature of React has been crucial during the frontend development.
- **Rich Ecosystem:** React has a vast ecosystem of libraries and tools that we've used, such as React Router for navigation and state management solutions like Redux. Moreover, for the design, we've chosen to use the MUI library.
- **Dependency Injection Support:** Libraries like `Inversify` made it easier to implement the Dependency Injection (DI) pattern on the frontend, promoting clean architecture principles and maintaining consistency between the frontend and backend.
- **Type Safety with TypeScript:** TypeScript enhances JavaScript by adding static typing, reducing runtime errors and improving code maintainability in large applications.
- **Performance:** React's virtual DOM efficiently updates the user interface, ensuring fast rendering and a responsive user experience.
- **Popularity and Community Support:** React is one of the most popular frontend frameworks, with a strong community and extensive resources for learning and troubleshooting.

**Disadvantages:**

- **Boilerplate Code:** While React provides flexibility, managing state and other common tasks often requires additional libraries and setup, leading to boilerplate code.
- **Tooling Complexity:** Setting up a React project with TypeScript, state management, and DI libraries can be complex and time-consuming for developers new to the ecosystem.

### 3.1.3. Overall Justification and Limitations

The combination of ASP.NET Core with C# for the backend and React with TypeScript for the frontend provides a well-rounded, modern technology stack.

**Limitations:**

- The initial learning curve for both ASP.NET Core and React with TypeScript may increase the onboarding time for new developers.

- Integration between the frontend and backend may require careful coordination, particularly when implementing DI patterns and managing shared state.

# 4 | Source Code Structure

## 4.1. Backend

This section provides an overview of the source code structure for the backend of the platform. The code is organized into three main layers: **Business**, **Data**, and **Service**, each serving a distinct purpose. Additionally, a **Shared** directory contains common utilities and resources.

### 4.1.1. Business Layer

The **Business** layer is responsible for handling the business logic of the application. This layer is structured to follow the **Command Query Responsibility Segregation (CQRS)** pattern. The organization is as follows:

- For each section of the application (e.g., *Authentication*, *Internship*, *Student*, etc.), there is a dedicated folder.
- Within each section folder, the relevant **use cases** are implemented.
- Each use case folder contains:
  - The **Command** or **Query**: These define the data and parameters required for executing the use case, and also its return value.
  - The corresponding **Use Case**: This contains the logic for processing the command or query.

### 4.1.2. Data Layer

The **Data** layer manages all database-related configurations and entity definitions. It is organized as follows:

- A file for the **database setup**, which includes configuration settings, database initialization. This file also handles **entity relations** and cascading policies, ensuring proper database constraints and relationships.
- A folder named `Entities`, which contains the definitions of all the **entities** in the application. Each entity corresponds to a table in the database and includes properties and navigation relationships. Each entity extends the **EntityBase**, that represents the base properties of an entity: an **autogenerated ID**, a **createdDate**

and an **updatedDate**

### 4.1.3. Service Layer

The **Service** layer serves as the interaction point between the application and the external users or systems. This layer includes:

- A `Contracts` folder containing the **DTOs (Data Transfer Objects)**, which are organized into subfolders corresponding to each section (e.g., *Authentication*, *Internship*, etc.).
- A `Controllers` folder that includes all the **controllers** responsible for processing HTTP requests and returning responses.
- A `Middlewares` folder that contains any middlewares or policies(for authorization)

### 4.1.4. Shared Folder

Outside the main three layers, a **Shared** folder contains resources and utilities that are reused across the application. This includes:

- A folder named `Enums`, which holds all the **enumerations** used throughout the application.
- A folder named `Services`, which contains shared services such as:
  - The **Storage Service** for handling file uploads and storage operations.
  - The **Email Service** for sending emails.
  - The **Background Matching Job Service**, a folder that contains many files for handling the matching background job.

## 4.2. Frontend

This section provides an overview of the source code structure for the frontend of the platform. The code has many files in the root folder, that handle for example routing or theming, but the core of the code is organized into several directories:

### 4.2.1. Pages Folder

The `pages` folder contains all the **pages** of the application. Each page represents a distinct view or route, such as the login page, home page, or internship details page.

### 4.2.2. Components Folder

The `components` folder contains all the **reusable UI components** in the application. The components are further categorized into subfolders based on their type, such as:

- `layout`: Components related to the layout, such as headers, footers, and sidebars.

- `buttons`: Custom button components used across the application.
- `list-items`: Components for rendering lists and list items.
- Other types of reusable components.

### 4.2.3. Hooks Folder

The `hooks` folder contains all the **custom hooks** created for the application. These hooks are often wrappers that encapsulate reusable logic.

### 4.2.4. Models Folder

The `models` folder is used to define all the **interfaces and types** for the application's data models. For example:

- `auth`: Contains interfaces/types related to authentication, such as login payloads or user models.
- `application`: Contains interfaces/types for internship applications.

Since we are working in TypeScript, this folder is crucial to have a specific point where to define our types.

### 4.2.5. Core Folder

The `core` folder contains essential services and utilities that form the core infrastructure of the frontend. It is structured as follows:

- `config`: This folder contains files for setting up and managing the **configuration service**.
- `ioc`: This folder is dedicated to **inversion of control (IoC)** and is set up using `Inversify`. It manages dependency injection across the application.
- `store`: This folder is used for **Redux state management**. It includes:
  - Configuration files for setting up the Redux store.
  - A `slices` folder, where each slice represents a distinct piece of application state, such as authentication state or internship data.
- `API`: This folder handles API-related logic. It is further divided into:
  - `setup-api`: Contains files for setting up the API infrastructure, including factories and decorators.
  - Subfolders for each API (e.g., `auth`, `internship`), where each subfolder includes:
    - An **interface** defining the API contract(e.g IInternshipApi).
    - One or more **implementations** of the API(e.g InternshipApi).

## 4.3. Patterns and Architectural Choices

This section explains the design patterns and architectural principles adopted across both the backend and frontend, where we have always tried to follow the **SOLID** principles.

### 4.3.1. Backend

- **Dependency Injection (DI)**: Dependency Injection is used to manage dependencies, promoting loose coupling between components, simplifying testing.
- **Command Query Responsibility Segregation (CQRS)**: The CQRS pattern is used in the **Business Layer**, where commands and queries are handled separately. This allows for better scalability and separation of responsibilities.
- **Mediator**: The Mediator pattern is used to coordinate communication between different components, especially in the context of processing use cases.
- **Mapper**: A mapper is employed to handle conversions between **entities** and **DTOs (Data Transfer Objects)**, ensuring that the data structure used in the service layer and data layer remains consistent.

### 4.3.2. Frontend

- **Dependency Injection (DI)**: Dependency Injection is implemented using `Inversify` to manage services and dependencies across the application.
- **Builder**: The builder pattern is utilized in the setup for the configuration service.
- **Factory**: Factories are used to create instances of various services, including API setup.
- **Decorator**: Decorators are applied for instance to API setup, in order to decorate the HttpClient to handle RefreshToken procedures.
- **State Management (Redux)**: The **store** directory utilizes Redux for managing global application state. Redux slices represent different parts of the state, such as authentication.

# 5 | Testing

As detailed in the Design Document (Chapter 5), we adopted a bottom-up implementation approach during the development of the platform. For each section of the platform, including Authentication, Internship management, and more, we implemented the corresponding Use Cases and conducted Unit Tests in parallel. Specifically, we utilized the `xUnit` testing framework and developed an `IsolatedUnitTests` class. This class provided an isolated environment for creating unit tests, including all necessary dependencies and mock objects.

To illustrate, for the `RegisterUseCase`, we conducted the following key unit tests:

- Successful registration of a student.
- Successful registration of a company.
- Handling of already registered email addresses.
- Validation of password strength.

We aimed for a line code coverage of at least 97% in each section, ensuring that every line of code functioned as intended. Performing these unit tests during the development of each section was critical for maintaining consistent functionality over time, essentially forming a continuous integration (CI) workflow.

After implementing and unit testing all sections of the platform, we moved to integration testing. This phase was designed to verify that the individual building blocks of the platform worked in the right way together. Integration testing included comprehensive scenarios such as:

- For students: registration, profile editing, and applying to internships.
- For companies: registration, profile editing, and internship creation.

In addition to verifying correct functionality and database effects, we ensured that HTTP requests returned the correct response codes for each interaction. As with unit testing, `xUnit` was used for integration tests.

The final phase of testing involved End-to-End (E2E) testing, conducted using `Cypress`, to simulate real-world scenarios and ensure the platform functioned correctly from the user's perspective. E2E testing included:

- Verification of the user interface: ensuring it rendered correctly across different browsers and devices.

- Testing critical user workflows: such as student registration, internship application, and company creation of internships.
- Validation of edge cases: such as handling invalid user inputs or ensuring robust error recovery mechanisms.
- Performance testing: measuring response times for key user actions to ensure the platform's responsiveness.

The E2E tests not only validated the correctness of the platform but also ensured a smooth and intuitive user experience. `Cypress` provided an efficient framework for automating these scenarios, enabling us to identify and address issues early. This comprehensive testing strategy ensured that our platform met the highest standards of reliability and usability.

### 5.0.1. Static Analysis

In addition to unit, integration, and end-to-end testing, we also performed static analysis on the codebase to ensure code quality at static time. Static analysis has been done using linters which helped us identify potential issues such as syntax errors, unused variables, and violations of coding standards. For example, `ESLint` helped ensure TypeScript type safety and React best practices. Additionally, we employed code formatters such as `Prettier` to enforce consistent code formatting across the project. While formatters do not analyze code behavior, their contribution to maintaining readability and consistency aligns with the goals of static analysis.

# 6 | Installation Instructions

This chapter provides detailed instructions for setting up and running the platform locally, including both the backend (written in ASP.NET Core) and the frontend (developed in React with TypeScript). Follow these steps carefully to ensure proper configuration and execution.

## 6.1. Prerequisites

Before proceeding with the installation, ensure that the following prerequisites are met:

- **Backend Requirements**:
  - .NET SDK (version 8.0 or later)
  - A code editor (e.g., Rider or Visual Studio Code)
- **Frontend Requirements**:
  - Node.js (version 18.x or later) and npm (version 9.x or later)
  - A code editor (e.g., Visual Studio Code)

## 6.2. Setting up

1. Copy the folder in the zip file named "VaccarinoPalladinoVacis" wherever you prefer on your device. If you have access to the github repository, you can find that zip also in the Deliverables folder.

### 6.2.1. Setting Up the Backend

The backend is implemented as an ASP.NET Core application. Follow these steps to set it up:

1. Consider the backend folder(assuming you're in the "VaccarinoPalladinoVacis" folder:

   ```
   cd SC/backend
   ```

2. Configure the **environment variables**.

In the zip file you can find a directory called env-files that contains all the necessary file to copy. In particular for the backend you will copy:

- An `appsettings.json` file containing the required configuration (e.g., database connection string, API keys, etc.) Place this file in the root of the backend project directory.
- An `appsettings.Development.json` file containing the required configuration for the development settings. Place this file in the root of the backend project directory.
- A `launchSettings.json` file containing the required settings for launching. Place this file into a directory called `"Properties"`, that must be placed in the root of the backend project directory.

3. Restore dependencies:

   ```
   dotnet restore
   ```

4. Run the application locally:

   ```
   dotnet run
   ```

   The backend will typically run on `https://localhost/api:5000` by default. **Ensure the port is 5000**

### 6.2.2. Setting Up the Frontend

The frontend is a React application written in TypeScript. Follow these steps to set it up:

1. Consider the frontend folder(assuming you're in the "VaccarinoPalladinoVacis" folder:

   ```
   cd SC/backend
   ```

2. Install dependencies:

   ```
   npm install
   ```

3. Run the application locally:

   ```
   npm run dev
   ```

   The frontend will typically be available at `http://localhost:5173`. **Ensure the running port is 5173, in order to don't face CORS policies issues**.

## 6.3. Running the Platform Locally

Once both the backend and frontend are set up, follow these steps to run the entire platform:

1. Start the backend (if not already started) by running `dotnet run`.
2. Start the frontend (if not already started) by running `npm start`.
3. Open a browser and navigate to `http://localhost:5173` to access the platform.

## 6.4. Troubleshooting

- **Frontend API errors**: Ensure the running ports are correct(5000 for backend and 5173 for frontend).
- **Dependency installation issues**: Make sure the correct versions of .NET SDK, Node.js, and npm are installed.

## 6.5. Running the tests

1. Navigate to the BackendSolution folder (assuming you're in the 'VaccarinoPalladinoVacis' folder):

   ```
   cd SC/BackendSolution
   ```

2. Run the tests(before executing it, **ensure** that there are not any active executions of the backend application):

   ```
   dotnet test
   ```

## 6.6. Mail-Related Features

All features related to mail sending, such as password recovery or email verification, require authorization through AWS SES (Simple Email Service). Since we are using SES in sandbox mode, these features are currently unavailable unless the email address is authorized. Due to this limitation, we have not restricted access for users who have not been verified.

# 7 Effort spent

| Member of Group | Effort Spent | Hours |
|---|---|---|
| Giovanni Vaccarino | Introduction | 0h |
| | Implemented Requirements | 0h |
| | Adopted Frameworks | 2h |
| | Source Code Structure | 2h |
| | Testing | 2h |
| | Installations | 1h |
| Vittorio Palladino | Introduction | 1h |
| | Implemented Requirements | 1h |
| | Adopted Frameworks | 1h |
| | Source Code Structure | 0h |
| | Testing | 1h |
| | Installations | 1h |
| Nicolò Vacis | Introduction | 0h |
| | Implemented Requirements | 1h |
| | Adopted Frameworks | 1h |
| | Source Code Structure | 2h |
| | Testing | 1h |
| | Installations | 2h |

Table 7.1: Effort spent by each member of the group

# 8 References


- **2024-2025 Software Engineering 2** - RASD Document
- **Aws RDS** - https://aws.amazon.com/it/rds/
- **Aws SES** - https://aws.amazon.com/it/ses/
- **Aws S3** - https://aws.amazon.com/it/s3/
- **Microsoft .NET Microservices** - https://learn.microsoft.com/it-it/dotnet/architecture/micr container-microservice-net-applications/test-aspnet-core-services-web-apps
- **ASP.NET Core Dependency Injection** - https://learn.microsoft.com/en-us/aspnet/core/fu injection?view=aspnetcore-9.0
- **InversifyJS** - https://github.com/inversify/InversifyJS
- **Cypress Testing Framework** - https://docs.cypress.io/app/get-started/why-cypress
- **S.O.L.I.D Design Principles** - https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- **HTTP CORS Documentation** - https://developer.mozilla.org/en-US/docs/Web/HTTP/C
- **React Internals FAQ** - https://legacy.reactjs.org/docs/faq-internals.html
- **AWS SES Sandbox Guide** - https://www.bigmailer.io/aws-ses-sandbox/