Software Requirements Specification Document

A2 Tiim – Karel Berg, Erik-Tiit Proos, Cen Pulst, Andreas-Valentin Jalasto, Karl Markus Saareli

Internship Portal Web Application

[**1. Introduction 2**](#_w1koua2xt9lh)

[1.1 Purpose 2](#_fp7xqbwrr6w0)

[1.2 Scope 2](#_b4a9qk86e3uq)

[**2. General Description 2**](#_zhshxccaf6mo)

[2.1 Product Perspective 2](#_4k3w5ib6y6p)

[2.2 Product Features 2](#_dv5g74iy5ssl)

[2.3 User Classes and Characteristics 2](#_nf21qalk2g5y)

[2.4 Operating Environment 2](#_33te788ie7a5)

[2.5 Design and Implementation Constraints 2](#_2gd4qjtrtvkk)

[**3. Functional Requirements 3**](#_hw692qk8fb2b)

[3.1 Start [Fill in…] them in from 2.2 point 3](#_7u883nssh4tz)

[**4. Non-Functional Requirements 3**](#_fau1po9gtpzj)

[4.1 Performance 3](#_t47wovx0ru0p)

[4.2 Security 3](#_mkjfksg7hmdi)

[4.3 Usability 3](#_5emtptcmn0s7)

[4.4 Compatibility 3](#_yxf0qvuwixtt)

[4.5 Reliability 3](#_bsg83dy7en6z)

[4.6 Scalability 3](#_fqmwnmjkuv9m)

[4.7 Legal and Compliance 3](#_4zqe3ica3div)

[**5. System Models 3**](#_xo537s8rgulw)

[5.1 Use Case Diagram 3](#_e9tmsqepn7is)

[5.2 Sequence Diagrams 4](#_k74ntoknnts7)

[5.3 Data Flow Diagrams 4](#_unatw8fgyifi)

[**6. External Interface Requirements 4**](#_3rhcbtox4299)

[6.1 User Interfaces 4](#_lhx9v72lc8f6)

[6.2 Hardware Interfaces 4](#_p2kbi5625puf)

[6.3 Software Interfaces 4](#_xjsd4fa49yiw)

[6.4 Communication Interfaces 4](#_nttg5vpz7mbn)

[**7. Testing and Validation 4**](#_wzafb0mbpvv3)

[7.1 Test Case 4](#_ywjpvbqjisxm)

[7.2 Validation Criteria 4](#_p5gkj4lknhxr)

[7.3 User Acceptance Testing 5](#_t1w2da11s3te)

[**8. Maintenance and Support 5**](#_5lgbgf29ibji)

[8.1 Maintenance Procedures 5](#_dzwfodpwgd3s)

[8.2 Support and Helpdesk 5](#_48yoxtuevw88)

[**9. Appendices 5**](#_uowp3tksltw9)

[9.1 Glossary 5](#_j9jmwzopmnwz)

[9.2 Revision History 5](#_78g26l7mvuz2)

## 1. Introduction

### 1.1 Purpose

The purpose of this document is to specify the requirements for the development of a web application that allows companies to post internship listings and for students to submit CVs for internship applications.

### 1.2 Scope

This document outlines the functional and non-functional requirements for the Internship Portal web application.

## 2. General Description

### 2.1 Product Perspective

The Internship Portal web application operates as a standalone system, interacting with external databases for data storage. It offers internship posting and candidate CV submission features.

### **2.2 Product Features**

Key features of the web application include:

* User management (login, registration)
* Internship posting
* Internship search posting ?
* CV, (motivation letter?) submission
* Students can search and filter internship postings
* Students and companies can contact staff or admin
* Students can apply to the posted internships through the web application by sending the company a message with their CV and optionally a motivational letter.

### **2.3 User Classes and Characteristics (personas)**

* Students – Can post CV’s and apply for internship.
* Employers – Users who post internship listings.
* School staff – Users who can check their class students’ application status.
* Admin – Users who will approve of school staff registration, so there would be no fake staff being created. Admins can manage the accounts of all registered users.
* Super Admin – The crème of the crop of accounts. Super admins can do everything an admin can but also have access to the whole infrastructure of the web app. The accounts will be heavily secured, and the passwords will be stored locally and not in the database.

### 2.4 Operating Environment

The system will be hosted on a web server with compatibility for modern web browsers.

### 2.5 Design and Implementation Constraints

The application should be built using Express + Drizzle with React (Vite) and UI component library.

## **3. Functional Requirements**

### 3.1 Internship posting

1. Employers should be able to create job postings by providing following info:
   1. company details (Short description/history, company name, location (Main HQ), phone number, email, registration code)
   2. job details (location, job requirements, job title/description, salary, work term, )
   3. application timeline
   4. applicant spots
2. Employers can edit or delete their job postings. Employers can edit most of everything in job details, except job location (home => office), worktime, salary, applicant spot, timeline.
3. The system should provide an internship posting ID for each listing.
4. Hides outdated posts.
5. Hides posts when job spots are full.

### 3.2 CV submission

1. Must have: Upload CV in in the system by providing following info: (Name, age, phone number, email, experience/skill, language level, interests/hobbies, summary/short introduction, desired position).
2. Nice to have: System can integrate with LinkedIn profile.

### 3.3 Contact

1. Students and companies can contact staff or admin through a dedicated button, which brings you to another page with all the contacts of the registered staff and admin(s). There will also be a button for a chat function.

### 3.4 Post filtering

1. Students can filter the posts by company name, job tags, profession, salary (If a company has not posted a salary to the post, it would not show up with the filter), work location (home, office or both).

### 3.5 Status

1. Students will have an application status list on the same page where they search for internship postings which shows the students the company name and the position which they’ve applied to and the current status of the application(unseen, seen, under review, accepted, declined).

## **4. Non-Functional Requirements**

### 4.1 Performance

1. For example: System should respond to user interactions withing 2 seconds. System should support 1000 concurrent users.

### 4.2 Security

Internship postings can only be edited or deleted by their owners.

Passwords should be hashed and salted.

Optional 2FA.

An email will be sent if there is account activity detected in another location or machine.

### 4.3 Usability & compatibility

We support all Chromium based browsers and Edge.

Option of remembering sign in on device.

Website will support most mobile device sizes.

### 4.5 Reliability

Users can access the website 98% of the time

### 4.6 Scalability

The service will be completely scalable without many errors.

### 4.7 Legal and Compliance

The system must comply with data privacy regulations, such as GDPR.

Intellectual property rights of internship postings and CVs should be respected.

## 5. System Models

### 5.1 Use Case Diagram

[Include a use case diagram illustrating the interactions between actors and the system. For example, a use case diagram could depict how internship seekers register, apply for internships, and update their profiles.]

### 5.2 Sequence Diagrams

[Provide sequence diagrams for key interactions, like the sequence of events when an employer posts an internship listing and when an internship seeker applies for an internship.]

### 5.3 Data Flow Diagrams

[If applicable, include data flow diagrams showing the flow of data within the system, such as how user profiles are updated and how internship listings are published.]

## 6. External Interface Requirements

### 6.1 User Interfaces

[Fill in…]

### 6.2 Hardware Interfaces

[Outline any specific hardware interfaces the system requires, for example, integration with payment processing hardware.]

### 6.3 Software Interfaces

[Outline any specific software interfaces the system requires, for example, integration with email service, database, etc.]

### 6.4 Communication Interfaces

[Fill in…]

HTTP/HTTPS Protocol: The application should communicate with users' browsers using the HTTP/HTTPS protocol.

APIs: Employ external APIs for geolocation services to determine location-based internship listings.

## 7. Testing and Validation

### 7.1 Test Case – Erik-Tiit Proos

1. **Backend –** 
   1. **Schema –**
      1. Environment Variables: Test if the environment variables for the database connection are correctly loaded.
      2. Database Connection: Test if a connection to the database can be established.

1.1.3 Table Creation: Test if the users, companies, and offers tables are created with the correct columns and types.

* + 1. Primary Keys: Test if the primary keys for each table are correctly set.
    2. Foreign Keys: Test if the foreign key relationships are correctly established.
    3. Unique Constraints: Test if the unique constraints on certain fields are correctly applied.
    4. Not Null Constraints: Test if the not null constraints on certain fields are correctly applied.
  1. **Authentication for JWT –** 
     1. No Token: Test the case where no token is provided. The server should respond with a 401 status and a message indicating that authentication is required.
     2. Invalid Token: Test the case where an invalid token is provided. The server should respond with a 403 status.
     3. Valid Token: Test the case where a valid token is provided. The server should call the next middleware function and the req.user property should be set to the decoded JWT payload.
  2. **Creating Business account –** 
     1. Successful API Request: Test the case where the API request is successful and returns a 200 or 201 status. The function should return the response data.
     2. Failed API Request: Test the case where the API request fails and returns a status other than 200 or 201. The function should throw an error.
     3. Network Error: Test the case where the API request encounters a network error. The function should throw an error.
  3. **Issuing JWTs –** 
     1. Token Generation: Test if the function generates a token when provided with a user object. The function should return a string.
     2. Payload Content: Test if the payload of the generated token contains the correct user information.
     3. Token Expiry: Test if the generated token has the correct expiry time (24 hours in this case).
     4. Token Verification: Test if the generated token can be successfully verified with the secret key.
  4. **Testing the API endpoints – Company, Users and Offers –**
     1. GET All Companies: Test if the endpoint returns all companies. If no companies exist, it should return a 404 status.
     2. GET Company by RegNo: Test if the endpoint returns a company by its registration number. If the company doesn't exist, it should return a JSON response, otherwise a 201 status with a message.
     3. GET Company by Email: Test if the endpoint returns a company by its email. If the company doesn't exist, it should return a JSON response, otherwise a 201 status with a message.
     4. GET Company by Name: Test if the endpoint returns a company by its name. If the company doesn't exist, it should return a JSON response, otherwise a 201 status with a message.
     5. GET Company by ID: Test if the endpoint returns a company by its ID. If the company doesn't exist, it should return a 404 status.
     6. POST New Company: Test if the endpoint creates a new company and returns a 201 status.
     7. PUT Update Company: Test if the endpoint updates an existing company. If the company doesn't exist, it should return a 404 status.
     8. DELETE Company: Test if the endpoint deletes an existing company. If the company doesn't exist, it should return a 404 status.

* + 1. GET All Offers: Test if the endpoint returns all offers. If no offers exist, it should return a 404 status.
    2. GET Offer by ID: Test if the endpoint returns an offer by its ID. If the offer doesn't exist, it should return a 404 status.
    3. POST New Offer: Test if the endpoint creates a new offer and returns a 201 status.
    4. PUT Update Offer: Test if the endpoint updates an existing offer. If the offer doesn't exist, it should return a 404 status.
    5. DELETE Offer: Test if the endpoint deletes an existing offer. If the offer doesn't exist, it should return a 404 status.
    6. GET All Users by Role: Test if the endpoint returns all users by role. If no users exist, it should return a 404 status.
    7. GET User by ID: Test if the endpoint returns a user by its ID. If the user doesn't exist, it should return a 404 status.
    8. POST New User: Test if the endpoint creates a new user and returns a 201 status.
    9. POST User Login: Test if the endpoint authenticates a user and returns a JWT token. If the user doesn't exist or the password is incorrect, it should return a 401 status.
    10. GET Check Auth: Test if the endpoint verifies a JWT token and returns a 200 status. If the token is invalid or not provided, it should return a 401 or 403 status.
    11. PUT Update User: Test if the endpoint updates an existing user. If the user doesn't exist, it should return a 404 status.
    12. DELETE User: Test if the endpoint deletes an existing user. If the user doesn't exist, it should return a 404 status.
  1. **Routing for backend –** 
     1. Server Startup: Test if the server starts up correctly on the specified port.
     2. Middleware: Test if the middleware (cors, cookieParser, authenticateJWT) are correctly applied.
     3. Routes: Test if the routes are correctly set up. This includes testing if the routes are accessible and return the expected responses.
     4. Authentication: Test the '/check-auth' route to ensure it correctly authenticates a JWT and returns a 200 status if successful.

1. **Frontend** –
   1. – **Business View –**
      1. **– Company Application** –
         1. Rendering: Test if the component renders without crashing.
         2. Initial Values: Test if the initial values of the form fields are correctly set from localStorage.
         3. Validation: Test if the validation functions validateApplicantPosistion and validateFieldOfWork work correctly.
         4. Blur Event: Test if the handleBlur function correctly sets localStorage, validates the input, and adds/removes the 'border-red' class.
         5. Form Submission: Test if the form correctly alerts the form values upon submission.
      2. **– Company info –** 
         1. Rendering: Test if the component renders without crashing.
         2. Initial Values: Test if the initial values of the form fields are correctly set from localStorage.
         3. Validation: Test if the validation functions validateEmail, validatePhone, validateName, and validateReg work correctly.
         4. Blur Event: Test if the handleBlur function correctly sets localStorage, validates the input, and adds/removes the 'border-red' class.
         5. Form Submission: Test if the form correctly alerts the form values upon submission.
      3. **- Offer info –**
         1. Rendering: Test if the component renders without crashing.
         2. Initial Values: Test if the initial value of the date input field is correctly set from localStorage.
         3. Blur Event: Test if the handleBlur function correctly sets localStorage.
         4. Date Change: Test if the handleDateChange function correctly updates form state and localStorage.
         5. Form Submission: Test if the form correctly sends data to the API, alerts success, removes form-related items from localStorage, and redirects to "/student-offer" upon successful submission. Also test if it correctly handles errors during submission.
      4. - **Offer validation and info –**
         1. Rendering: Test if the component renders without crashing.
         2. Initial Values: Test if the initial values of the form fields are correctly set from localStorage.
         3. Validation: Test if the validation functions validateJobdecsPosition and validateExpectationsForTheCandidate  work correctly.
         4. Blur Event: Test if the handleBlur function correctly sets localStorage, validates the input, and adds/removes the 'border-red' class.
         5. Tag Addition: Test if the handleKeyDown function correctly adds tags to the state and localStorage when the spacebar is pressed.
         6. Tag Removal: Test if the removeTag function correctly removes tags from the state and localStorage.
   2. **- User login –**
      1. Rendering: Test if the component renders without crashing.
      2. Initial Values: Test if the initial values of the form fields are correctly set to empty strings.
      3. Validation: Test if the validation schema correctly requires the email and password fields.
      4. Form Submission: Test if the form correctly calls the loginUser function with the form values upon submission. Also test if it correctly handles successful and unsuccessful login attempts.
      5. Navigation: Test if the "Register" button correctly navigates to the '/user-register' route.
   3. **- User registration –**
      1. Rendering: Test if the component renders without crashing.
      2. Initial Values: Test if the initial values of the form fields are correctly set to empty strings.
      3. Validation: Test if the validation schema correctly requires and validates the name, email, password, confirmPassword, and course fields.
      4. Form Submission: Test if the form correctly calls the registerUser function with the form values upon submission. Also test if it correctly handles successful and unsuccessful registration attempts.
      5. Navigation: Test if the "Login instead" link correctly navigates to the '/user-login' route.
   4. **- Student view –** 
      1. **- Internship cards –**
         1. Rendering: Test if the component renders without crashing.
         2. Initial Values: Test if the initial values of the state variables are correctly set.
         3. Search Functionality: Test if the search functionality correctly filters the data based on the search term.
         4. Modal Opening: Test if clicking the "Learn More" button opens the modal with the correct offer data.
         5. Modal Closing: Test if the modal closes correctly.
   5. - **Controllers –** 
      1. **- Business form –** 
         1. Successful Request: Test if the functions correctly send a POST request to the API and handle a successful response.
         2. Failed Request: Test if the functions correctly handle a failed request.
         3. Error Throwing: Test if the functions correctly throw an error when the response status is not 200 or 201.
      2. **- Login registration –**
      3. Successful Registration: Test if registerUser correctly sends a POST request to the API and handles a successful response.
      4. Failed Registration: Test if registerUser correctly handles a failed request.
      5. Successful Login: Test if loginUser correctly sends a POST request to the API and handles a successful response.
      6. Failed Login: Test if loginUser correctly handles a failed request.
      7. Successful Auth Check: Test if checkAuth correctly sends a GET request to the API and handles a successful response.
      8. Failed Auth Check: Test if checkAuth correctly handles a failed request.

### 7.2 Validation Criteria

[Define the criteria for validating that the system meets the specified requirements, e.g., successful completion of user acceptance testing.]

### 7.3 User Acceptance Testing

[Describe how the user acceptance testing will be conducted, including the involvement of stakeholders and the acceptance criteria to be met.]

## 8. Maintenance and Support

### 8.1 Maintenance Procedures

[Fill in…]

Routine Backups: Regularly backup the database to prevent data loss in case of system failures.

Security Updates: Stay updated with security patches and updates for the server, database, and web application to address vulnerabilities.

Bug Fixing: Establish a procedure for addressing and fixing reported bugs and issues.

### 8.2 Support and Helpdesk

Provide a contact channel for user support, such as email or a dedicated support form.

Define support hours and response times for user inquiries or issues.

## 9. Appendices

### 9.1 Glossary

[Fill in…]

SRS: Software Requirements Specification

CV: Curriculum Vitae

GDPR: General Data Protection Regulation

### 9.2 Revision History

[Include a section for recording changes and revisions made to the SRS document, with dates and descriptions of modifications.]