Technical Documentation – Frontend

LEO-Based Assessment Tool

---

1. Introduction

This document provides the **technical documentation** for the **frontend application** of the *LEO-Based Assessment Tool*. It describes the frontend architecture, technology stack, component structure, data flow, backend integration, and key technical decisions.

The frontend is implemented as an **Electron-based desktop application** using **React, Vite, and TypeScript**. It serves as the primary interaction layer for teachers and students and communicates with the backend via RESTful APIs.

---

2. System Context

2.1 Role of the Frontend

The frontend is responsible for:

- User interaction and navigation
- Visualization of LEO structures and progress
- Input and validation of assessment data
- Displaying recommendations and mastery levels

All business logic, grading rules, and data persistence are handled by the backend. The frontend focuses on presentation, usability, and interaction.

2.2 High-Level Architecture

The overall system consists of:

- **Frontend**: Electron + React (this component)
- **Backend**: Spring Boot REST API
- **Database**: Neon PostgreSQL (cloud-based)

The frontend communicates with the backend exclusively via HTTP using JSON payloads.

---

3. Technology Stack

The frontend application is built using the following technologies:

- **Electron** – desktop application framework

- **React** – component-based UI framework
- **Vite** – development server and build tool
- **TypeScript** – type-safe JavaScript
- **HTML & CSS** – UI structure and styling
- **Node.js / npm** – dependency management and tooling

---

4. Application Architecture

4.1 Single-Page Application (SPA)

The frontend is implemented as a **Single-Page Application (SPA)**. Page navigation is handled on the client side without full page reloads, providing a responsive user experience.

Electron wraps the SPA into a cross-platform desktop application that can run on Windows, macOS, and Linux.

4.2 Component-Based Design

The frontend follows a **component-based architecture** using React.

Key principles:

- Separation of concerns
- Reusable UI components
- Clear distinction between pages and shared components

---

5. Project Structure

The frontend project is organized as follows:

```
app/
├── electron/                # Electron main process (desktop window,
lifecycle)
├── public/                  # Static assets
├── src/
│   ├── api/                 # Backend communication (HTTP requests, error
handling)
│   ├── components/          # Reusable UI components
│   │   └── UI.jsx
│   ├── pages/               # Main application views
│   │   ├── teacher/         # Teacher dashboard and feature tabs
│   │   │   └── tabs/
│   │   ├── student/         # Student dashboard and views
│   │   ├── Login.jsx        # Authentication view
│   │   └── Landing.jsx      # Entry page
```

```
│   ├── App.jsx                # Root React component
│   ├── main.jsx               # Frontend entry point
│   └── mockData.js            # Mock data for development/testing
├── docker-compose.yml         # Docker configuration
├── Dockerfile                 # Frontend Docker image definition
├── index.html                 # HTML entry file
├── package.json               # Dependencies and scripts
├── vite.config.js             # Vite configuration
└── README.md
```

---

6. UI Components & Pages

6.1 UI Components

Reusable UI components are stored in the `components` directory.

Responsibilities: - Buttons, cards, inputs, dialogs - Layout and visual consistency - Shared styling across the application

---

6.2 Pages

Pages represent the main views of the application:

- **Login Page** – user authentication
- **Teacher Dashboard** – course management, LEO creation, assessments
- **Student Dashboard** – progress overview and recommendations

Teacher functionality is organized into **tabs**, allowing clear separation of features.

---

7. State Management

State is managed using **React Hooks**:

- `useState` for local component state
- `useEffect` for lifecycle events and data fetching

State includes: - Authenticated user information - Selected course and student - LEO lists and assessment data - Progress and recommendation results

No external state management library is used, keeping complexity low.

---

8. Data Flow

The typical data flow is as follows:

1. User performs an action in the UI
2. Frontend triggers an API request
3. Backend processes the request and returns JSON data
4. Frontend updates local state
5. UI re-renders dynamically based on updated state

This unidirectional data flow ensures predictable behavior and easier debugging.

---

9. Backend Integration

9.1 API Communication

- Communication via **RESTful HTTP endpoints**
- JSON request and response payloads
- Base API URL configured via environment variables (`VITE_API`)

9.2 Authentication & Authorization

- Users authenticate via login endpoint
- Authentication tokens are stored locally
- Tokens are attached to subsequent API requests
- Backend enforces role-based access (Teacher / Student)

Frontend views adapt dynamically based on the user role.

---

10. Visualization & User Experience

The frontend focuses on **clarity and usability**:

- List-based visualization of LEOs and their status
- Clear indicators for mastery levels
- Progress visualization using structured UI elements
- Recommendation lists for next possible LEOs

Complex graphical dependency diagrams were intentionally avoided to maintain usability and precision.

---

11. Error Handling

Error handling is implemented at multiple levels:

- API error responses are captured and displayed to users
- Validation errors are shown directly in the UI

- Network and backend errors are logged for debugging

This ensures transparent feedback without exposing technical details to end users.

---

12. Build & Deployment

12.1 Development Build

- Vite provides fast hot-reload during development
- Electron loads the Vite development server

12.2 Production Build

- Vite builds optimized frontend assets
- Electron packages the application
- Docker container is used for deployment on AWS EC2

This approach ensures consistent behavior across environments.

---

13. Key Technical Decisions

- **Electron** chosen for cross-platform desktop support
- **React** for modular, maintainable UI development
- **Vite** for fast development and efficient builds
- **TypeScript** to improve type safety and code quality
- SPA architecture for responsive user experience

---

14. Testing Considerations

Frontend testing focuses on:

- Manual UI testing
- Validation of API integration
- End-to-end feature verification during sprints

Automated backend testing ensures overall system correctness.

---

15. Related Repositories

- Frontend Repository: https://github.com/piy678/SENGPRJ_Group6_FrontendPart
- Backend Repository: https://github.com/piy678/SENGPRJ_Group6

---

16. Summary

The frontend of the LEO-Based Assessment Tool provides a structured, intuitive, and responsive user interface for outcome-based assessment.

Its component-based architecture, clear data flow, and tight backend integration ensure maintainability, scalability, and a positive user experience.

Group 6 — SENGPRJ
Supervisor: *Thomas Mandl*