Developer Manual

1. Introduction

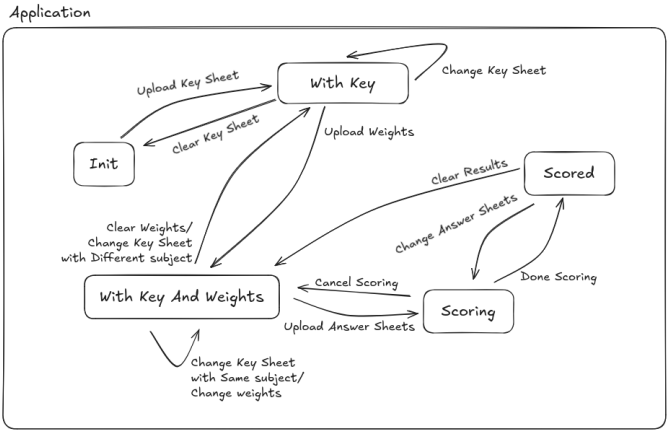
Quikscore is an application that can easily and accurately grade multiple answer sheets and store them safely for further usage in a database.

Technologies Used:

- **Frontend:** Vue.js (for building user interfaces) with TypeScript (for static type safety).
- **Backend:** The Rust-based Tauri framework, combining the frontend with a Rust backend for efficient API handling and system-level performance.
- **Authentication Server:** With Node.js as a runtime environment with the Express framework for API end points.
- **Image Processing & OCR:** Tesseract library for extracting and processing text from scanned answer sheets.
- **Database:** MongoDB Atlas for cloud-based, scalable, and secure data storage.

2. System Architecture

Quikscore's internal application state can be represented with the figure below.
(src-tauri/src/state.rs in Enum AppStatePipeline)



1. **Init**

   This is the initial state of the program. Start by uploading an answer key sheet.

2. **With Key**

   The user has uploaded the answer key. From here, the user can upload weights in CSV containing that answer key's subject ID or change the answer key.

3. **With Key and Weights**

   The user has uploaded both the answer key and its corresponding weights. From here, the user can upload the answer sheets to score. Or change either the answer sheet or the weights file.

4. **Scoring**

   The user has uploaded the answer sheets, and the scoring process has started. From here, the user can cancel the scoring process.

5. **Scored**

   The program has finished, and the results are displayed onscreen. The user can then clear the answer sheets, or upload a new batch.

3. Installation & Setup (Developer Environment)

- For Mac/Linux
  - With Nix (Recommended)
    1. Follow the devenv "Getting Started" guide.
       If you wish to also enable automatic environment activation, follow the "Automatic shell activation" guide.
    2. If you didn't setup automatic shell activation, run devenv shell to enter the development environment. First setup may take a while. If you had setup automatic shell activation, for the first time direnv will prompt you to approve its contents. Run direnv allow, then the developer environment will be automatically activated for you when you enter the project directory.
  - Without Nix

    These are general instructions. Adapt them to your specific setup.
    1. Install core build tools
       - Linux: gcc, g++, make, pkg-config
       - macOS: Xcode Command Line Tools (xcode-select --install) and Homebrew
    2. Install rustup from https://rustup.rs. The rust-toolchain.toml file witll automatically tell Rust what toolchain and components to use.
    3. (Optional) Run cargo install cargo-nextest cargo-tarpaulin for enhanced testing and development tools.
    4. Install required libraries.
       - Ubuntu/Debian based:

sudo apt update

sudo apt install build-essential pkg-config libopencv-dev tesseract-ocr libtesseract-dev libleptonica-dev libssl-dev llvm-dev libclang-dev zlib1g-dev libgtk-3-dev libpango1.0-dev libcairo2-dev libharfbuzz-dev libgdk-pixbuf2.0-dev libatk1.0-dev libatspi2.0-dev libgirepository1.0-dev libwebkitgtk-6.0-dev librsvg2-dev

- Fedora (RPM based):

  sudo dnf install gcc gcc-c++ make pkgconf-pkg-config opencv-devel tesseract tesseract-devel leptonica-devel openssl-devel llvm-devel clang-devel zlib-devel gtk3-devel pango-devel cairo-devel harfbuzz-devel gdk-pixbuf2-devel atk-devel at-spi2-atk-devel gobject-introspection-devel webkit2gtk4.1-devel librsvg2-devel

- macOS (Homebrew):

  brew install opencv tesseract leptonica openssl llvm zlib

5. (Optional) For prototyping things in /playground, set up a python virtual environment and install opencv-python and numpy.

6. Set up these environment variables:

```
export WEBKIT_DISABLE_COMPOSITING_MODE=1 # only on linux
export LIBCLANG_PATH=$(llvm-config --libdir)
```

```
export RUST_LOG_STYLE=always
export RUST_LOG=debug
```

- For Windows
  1. Install Node.js from the official website.
  2. Enable Yarn by installing Corepack using the command: npm install -g corepack
  3. Install Rust via rustup from https://rustup.rs
  4. install Chocolatey by running the command:
     Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
     If this fails, run Set-ExecutionPolicy Unrestricted and rerun the command.
  5. Using Chocolatey, install OpenCV and LLVM libraries with the command:
     choco install llvm opencv -y. Confirm that the folder C:\tools\opencv exists.
  6. Set the following environment variables:
     OPENCV_INCLUDE_PATHS : C:\tools\opencv\build\include,
     OPENCV_LINK_LIBS : +opencv_world4110
     OPENCV_LINK_PATHS : +C:\tools\opencv\build\x64\vc16\lib.
     Add C:\tools\opencv\build\x64\vc16\bin to the system PATH.
  7. Install msys2, then open the UCRT64 version of msys2.
     then, run the following to update the system and

install the dependencies.
pacman -Syu –noconfirm
then, msys2 will restart. start it back up, then run
pacman -Syu --noconfirm # just to make sure everything is updated
pacman -S mingw-w64-ucrt-x86_64-tesseract-ocr mingw-w64-ucrt-x86_64-openssl –noconfirm

8. After that, you need to define 12 environment variables. If you use powershell, paste the script below. If not, create them manually.

```
$env:MSYS2="C:\msys64"
[Environment]::SetEnvironmentVariable("OPENCV_INCLUDE_PATHS", "C:\tools\opencv\build\include", "User")
[Environment]::SetEnvironmentVariable("OPENCV_LINK_PATHS", "C:\tools\opencv\build\x64\vc16\lib", "User")
[Environment]::SetEnvironmentVariable("OPENCV_DLL_PATH", "C:\tools\opencv\build\x64\vc16\bin", "User")
[Environment]::SetEnvironmentVariable("OPENCV_LINK_LIBS", "opencv_world4110", "User")
[Environment]::SetEnvironmentVariable("LEPTONICA_INCLUDE_PATH", "$env:MSYS2\ucrt64\include", "User")
[Environment]::SetEnvironmentVariable("LEPTONICA_LINK_PATHS", "$env:MSYS2\ucrt64\lib", "User")
[Environment]::SetEnvironmentVariable("LEPTONICA_DLL_PATH", "$env:MSYS2\ucrt64\bin", "User")
[Environment]::SetEnvironmentVariable("LEPTONICA_LINK_LIBS", "leptonica", "User")
[Environment]::SetEnvironmentVariable("TESSE
```

RACT_INCLUDE_PATHS", "$env:MSYS2\ucrt64\include", "User")
[Environment]::SetEnvironmentVariable("TESSERACT_LINK_PATHS", "$env:MSYS2\ucrt64\lib", "User")
[Environment]::SetEnvironmentVariable("TESSERACT_DLL_PATH", "$env:MSYS2\ucrt64\bin", "User")
[Environment]::SetEnvironmentVariable("TESSERACT_LINK_LIBS", "tesseract", "User")

Note: For an explanation of these environment variables, see the readme.

The instructions above assume the default installation path for MSYS2: C:\msys64. If you have installed MSYS2 to a different location, change C:\msys64 to your install location.

9. Navigate to the project folder and run yarn install to install all project dependencies.
10. To preview the application in development mode, run yarn tauri dev.
11. To config the external server please install Nodejs and Node Package Manager (NPM). The navigate to server and run npm install to download all the dependencies used for the authentication server.

4. Code Structure

This is a brief directory structure that you will be most likely be working on.

Directory Structure

Project Root

```
├── server/
├── src-tauri/
│   ├── src/
│   └── tests/assets/
└── src/
    ├── assets/
    └── components/
```

Folder and File Descriptions

- **server/**: Contains all secret and handle API end point

for authorization.

- **src-tauri/**: Contains all Rust backend code that Tauri uses to handle system-level operations, API endpoints, and integration with the frontend.
- **src-tauri/src/**: Main Rust source files implementing backend logic.
- **src-tauri/tests/assets/**: Sample files or images used for backend testing.
- **src/**: Frontend application source code written in Vue.js with TypeScript.
- **src/assets/**: Static assets like images, icons, and CSS files used by the UI.
- **src/components/**: Reusable Vue components and states.

Naming Conventions and Coding Standards

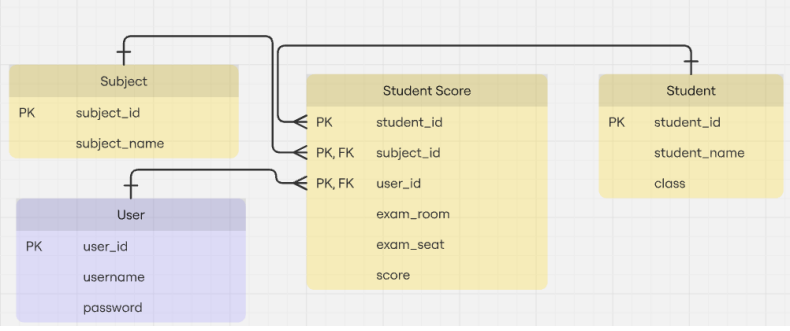
- **Rust (backend)**:
  - Use snake_case for variables, functions, and file names.
  - Use PascalCase for struct and enum names.
  - When in doubt, run cargo clippy for linting and rustfmt for auto-formatting.
- **Vue.js / TypeScript (frontend)**:
  - Component file names use PascalCase.
  - Folder names use lowercase and hyphens, e.g., components/, assets/.
  - Variables and functions use camelCase.
- **ExpressJS (exterior server)**:
  - Use to contain every secret like jwt secret or some important APIs.
- **General**:
  - Keep file names descriptive and match the component or module purpose.
  - Organize folders by feature or functionality rather than type

5. API Documentation

- **GET** /login for authorizing users.

6. Database Design

Right now, our team are only able to create an unnormalized relation of Student Score which could lead to some anomalies which are impractical. This is because of the limitation of time we have. So, our team couldn't finish the database system as planned, but here is our design.



For the above figure, there are 4 objects which are user, student, subject and student score. The relatedness is also shown in the figure.

7. Deployment & Contribution Guide

We use a "git-feature-branch" development model, where new features and bugs are assigned a "story" (A Github Issue) with members assigning themselves and/or others to said issue. When you want to work on a feature, you base a new git branch off main, do your work, then at the end you open a pull request to merge your branch back into the main branch. If relevant files are changed, this would then run some CI checks to check the correctness of your code. If they all pass, you are free to merge. If not, look at the logs of the CI job that failed (at the end), fix those issues, then commit the changes to your branch. This would then rerun the CI tasks.

Binary builds of Quikscore are automatically run by our CD pipeline when a successful commit to the main branch is done.

8. Testing

To run test, first navigate to src-tauri directory and then run the **cargo nextest run** command to run unit test within the src-tauri directory. When you implement new function, you should also add unit test to increase coverage.

9. Contribution Guidelines

1. Branching Strategy

- **main branch**: always contains stable, production-ready code. Never directly push code into main
- **Feature branches**: for integration of features before merging to main and always link it to an associated isssues.

2. Making Changes

Before starting work, pull the latest changes from main and create your feature branch.
Write clean, modular code following the existing coding conventions listing in Code Structure.

3. Committing Changes

Write clear, descriptive commit messages.

10. Maintenance & Future Enhancements

The limitations of our project which could be further improve are;

1. Users only allow to use specific format of answer sheet. It might be better to dynamically scored the answer sheet is possible.

2. We did not provide a feature for users to view their score in database directly.

3. Database relations are not normalized yet.