GoodMark

Simple markdown preview and editing web-tool.

Project Summary

This project aims to provide a simple, lightweight Markdown previewer. Users can change and preview Markdown syntax in real time, assisting beginners and experts alike. The program simplifies Markdown to make it accessible.

Project Description

GoodMark is a web-application that converts plain-text Markdown into rendered HTML in real time to facilitate Markdown writing. It will be useful for both beginners and experts who seek a lightweight Markdown tool that renders in real time without a complete editor.

Main features:

1. A live preview that updates as Markdown is typed.
2. Code snippet syntax highlighting.
3. A simple, clean user interface, see User Interface Design.
4. Dark and light modes for your convenience.

Design

This project's design and construction involve multiple stages. I began with User Interface design to visualise my project. After designing the Software's architecture, I planned its implementation.

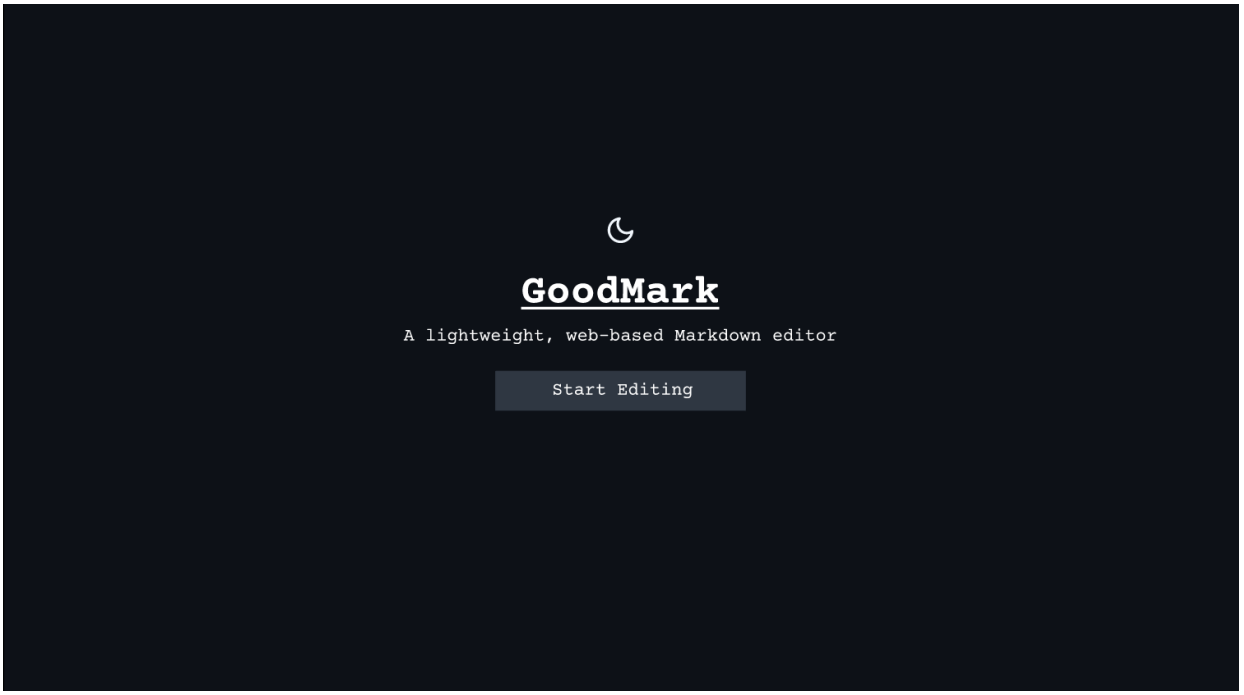
User Interface

The user interface design was created in Figma. The design prototype is here.

Product Design

GoodMark Welcome Screen

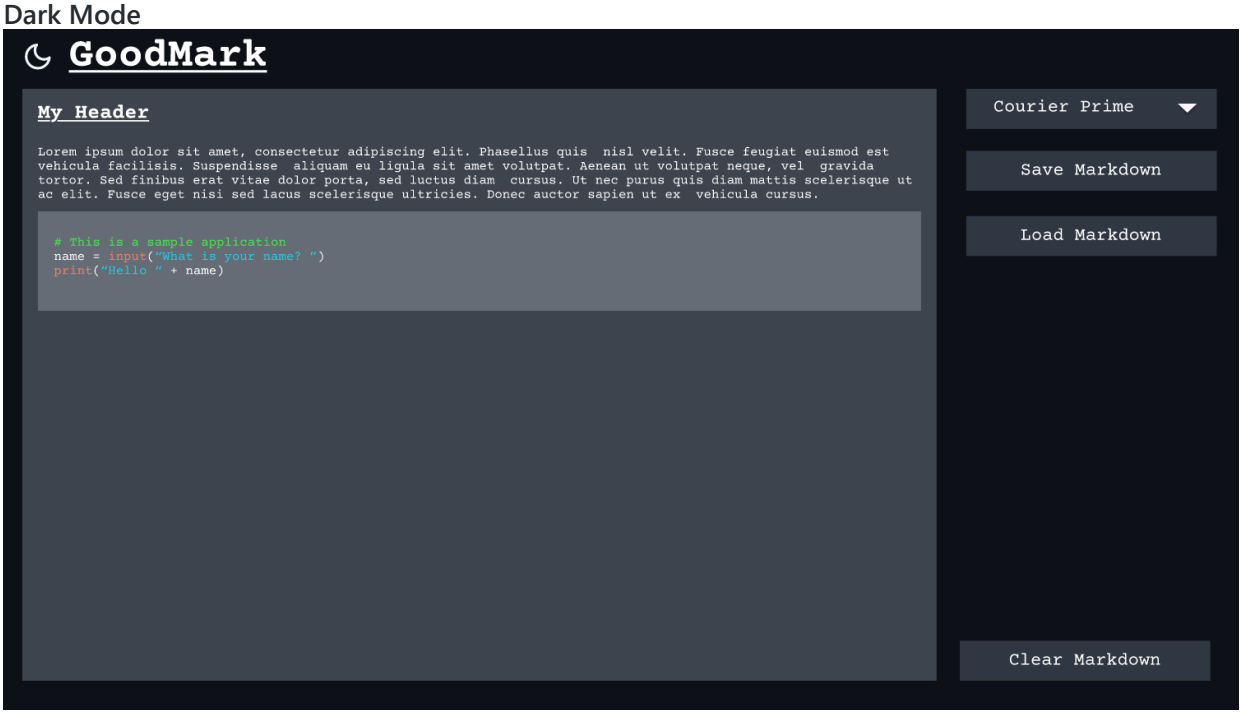
The website should open with a splash screen that introduces the product. Similar to how many apps construct their software. The game menu before the game is typical in video game design. I liked this technique for this project because it didn't overwhelm new users with an immediate editing screen and gave them a choice before accessing the editor. I plan to use a cookie on the user's browser to track if they have visited the site and pressed the "Start Editing" button before. If they have, they will not be presented with the screen again unless they clear their cookies, avoiding regular users from repeating the process while still giving newer users this splash screen. The user can also choose between bright and dark mode on this screen, which will be recorded in a browser cookie. Dark mode will be the default.



Editor Screen

When the user clicks "Start Editing" or has visited the website before and stored the cookie in their browser, they are directed to the primary Markdown editing screen. If they have never modified, this screen will display sample text. If they have, the last text they entered will be populated into the textbox and kept in a cookie. The sample text will have a headline, normal text, and a code block to demonstrate Markdown alternatives. They will also have a combo-box to select a font from a pre-defined list, a "Save Markdown" button to save their Markdown to their machine, and a "Load Markdown" button to load any Markdown file from their machine into the text box. The "Clear Markdown" option in the bottom right allows users to easily remove their Markdown, although it prompts for approval to avoid unintentional data loss. This screen's top left icon lets them choose light or dark mode like the welcome screen.

Dark Mode



Light Mode

Colour Scheme

As Markdown is widely used on GitHub, I thought it would be appropriate to use GitHub's colour scheme to preview the Markdown in a README format like this file. I think GitHub's neutral colour scheme is good for this project because it won't distract users as they write Markdown. Flashy colours can distract users. I also chose dark mode as our default colour scheme because it avoids accidental issues with bright light. For instance, if someone's eyes are adjusted to the dark, a bright light will ruin that adjustment, but a dark screen will barely change that.

Accessibility

Although GitHub's colour scheme has already been tested for accessibility, I ran some basic colour blindness simulations using Coblis to make sure the UI was accessible to colour blind users. The results are available here. Keyboard navigation will be needed to allow users without mice to navigate the tool, including the "Start Editing" button, font selection, "Load Markdown" and "Save Markdown" buttons, and "Clear Markdown" button. A user must also be able to return to the editor box. Another crucial accessibility test will ensure screen readers can read the markdown editing area. The site will be in English exclusively. Responsive design will be important: editing Markdown on a phone is not ideal, but we should allow a user to do so and scale the page as needed. No animations will be on this site, thus no need to address this, however we will add a timeout to the dark mode and bright mode switcher to prevent accidental mass-activations that could cause photosensitivity.

Software Architecture

The React frontend will provide a responsive user interface. A hosted backend will render Markdown-to-HTML, allowing us to use server-side processing. Due to the tiny size of this project, server-side processing is appropriate and will not overwhelm servers.

Backend Architecture

The backend will use Express.js and have two endpoints:

- GET `/render` converts Markdown to HTML.
- GET `/fonts` returns a list of available fonts.

Frontend Architecture

The frontend will be implemented with React, and will feature these main components:

- Welcome Page:
  - Light/dark mode selector
  - Text field for title and description
  - A "Start Editing" button
- Editing Screen:
  - A Markdown input field that converts to HTML.
  - Title and light/dark mode selector header
  - A font selector and control buttons sidebar -Several sidebar control buttons

Implementation Approach

To ensure program robustness, Test-Driven Development will be used. Before adding a feature, a test must be prepared and then the feature built to pass the test. Component-level testing with Jest's React Testing Library will ensure that all frontend components are implemented as expected. We will write integration unit tests to test each endpoint and its function and mock externals to test our code to guarantee the backend works properly. Using GitHub Actions, we will create a CI/CD pipeline that builds the project, passes all tests, and lints the code before merging a PR. The pipeline will also push PR changes to live software after merging.

Planning

For a well-rounded project, I want to use Agile sprints to provide the solution in stages. The following tools will help me design the solution.

Personas and User Stories

User stories will be my initial development tool. First, in high-level stories that explain the project's core requirements, then in user stories for each feature. To help with this, I've established three personas with their own aims and difficulties, as well as empathy maps of each. When developing a feature, I'll designate whose persona it's for. The personas are available here

High-Level User Stories

From personas and empathy maps, I generated 5 high-level user stories:

1. As a user, I want to view live previews of my Markdown content so that I can immediately see how it will render any rectify any issues.
2. As a user, I want the option to customise the view of the editor, such as, switching between light and dark mode so that the website is accessible.
3. As a user, I want the editor to be lightweight and intuitive, so that I can focus on editing Markdown rather than learning the tool I am using.
4. As a user, I want to create and edit Markdown with syntax highlighting, so that

code in the Markdown can be read easily.

5. As a user, I want the tool to be accessible via the web, so that I can avoid downloading more software to my machine.

MoSCoW Prioritisation

I've devised a MoSCoW prioritisation to identify which stories are most critical to develop and which may be left to later stages or made optional.

Must Have	Should Have	Could Have	Wont Have
Live Markdown Preview	Dark and Light Mode	Export Options	Collaboration Features
Lightweight Design	Syntax Highlighting	Offline Mode	
Web Accessibility		Templates	
Accessible Design		Responsible Web Design	

I included won't have "Collaboration Features" since I think it would improve the tool's instructional component for Educator Eric, but it's not in the MVP's scope.

Project Management

All project management approach will be done using GitHub tools, which is great because it keeps Source Code Management and Project Management together. This project will be managed utilising the following items, how they will be used, and why.

Issues, Epics and Stories

Issues on GitHub will be the main way to assign and prioritise bugs, features, infrastructure modifications, and other activities. The repositories' issue page allows you to describe any work.

Five issues types will be used in the project:

1. A user story should be used to describe a feature to be introduced.
2. Epics will be Feature Story or Infrastructure concerns. The Epic should describe all its work.
3. Bugs should be reported through bug problems, which will be handled outside of the standard project management system and not part of a Sprint, Epic, etc.
4. Project infrastructure modifications include development and project management infrastructure.
5. Other—Any issue not easily fit into these categories.

Infrastructure and Feature issues are usually inside Epics unless they are clearly identifiable as separate work. This will drive the project so that features are always linked to a broader project.

Sizing

They should be sized during Feature and Infrastructure issue generation. This project will employ shirt sizing to size each issue. Small, Medium, Large, Extra Large. Instead of

considering how long it will take to create the feature, one should assess how much effort it will demand. Issue Size ensures that sprint workloads are balanced and issues are the right size. Extra Large issues must be split into smaller ones. Labels will show sizes. Size only infrastructure, feature, and *some* other issues. Sized issues should be part of an Epic because they build a larger work.

Prioritisation

All issues except Epics should be prioritised to determine their importance. Urgent, High, Medium, and Low priorities are available. This priority should weigh the importance of completing the issue and the issues it may block. Urgent issues will be kept for crucial items that must be completed within the sprint. Prioritisation will be labelled like size.

Sprints

Project development will be done in sprints. Sprints require a specified amount of effort in a short time. Every sprint except Sprint 0, which is the initialisation sprint where all planning is done, will be one day due to project time limits. You should assign a sprint label to each issue and generate a new one if a sprint is needed. This enables iterative development. Depending on Epic size, each sprint should complete one or more Epics. Epics should be carried over into the next sprint if not finished.

GitHub Projects

We'll utilise GitHub Projects' Kanban board to manage and view work. Here is the GitHub Project for this repository: https://github.com/users/Dinoosawruss/projects/1. The board will feature columns for backlog, in progress, blocked, reviewed, and done work. The GitHub Project should represent the current state of the project by moving work across the board to the relevant column. The project board has an Epics tab that lists all epics, their status, and their sprint. A PR tab displays the status of all pull requests in the project.

The GitHub Project board gives stakeholders a simple picture of the project's status. If this were a standard open source project, anyone could see the present condition and find employment. The board will be used for internal tracking for this project to keep me informed of items in progress, on the backlog, and blocked.

Risks

Time limitation is the main project risk. Due to this, I have created a single-day sprint cycle, which means that if work is not completed in a sprint, it could postpone future sprints because there is no time to fix the issue. This could leave some functionalities out of the final project.

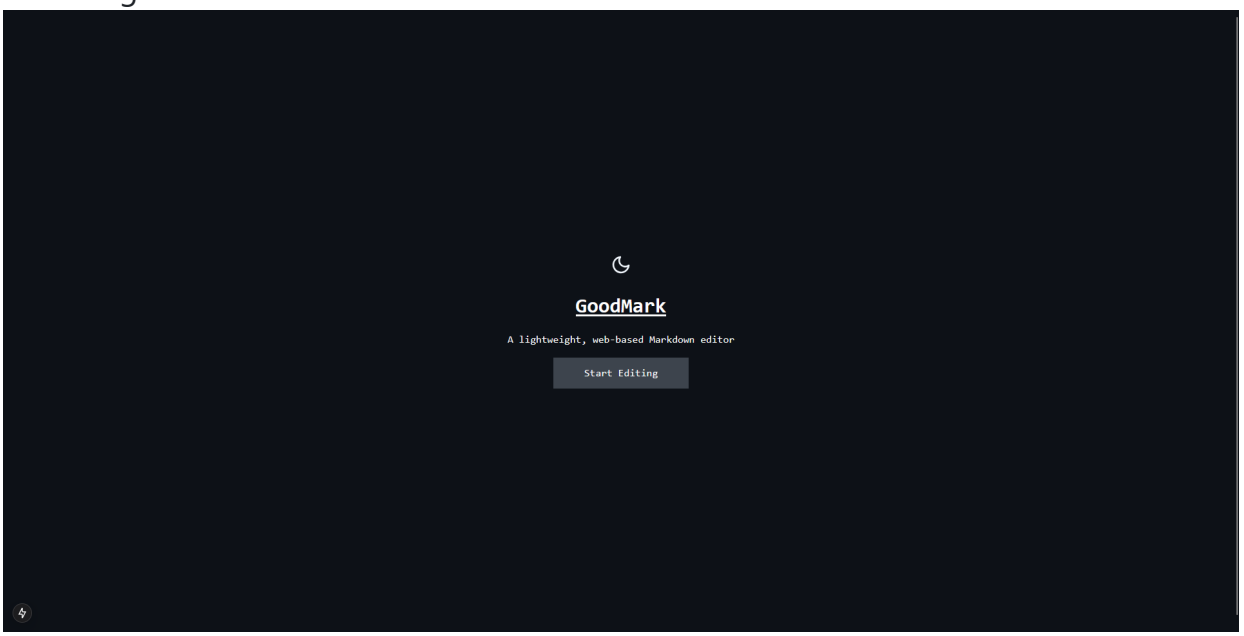
Due to server-side rendering, performance is another risk. If a huge Markdown file was used or several people utilised the tool at once, the website may have performance issues due to restricted resources.

Documentation

User

Render hosts GoodMark on the web Render. As this is a free instance, the deployment will spin down after 50 seconds of inactivity. If you load the URL, it will come back up,

but please take at least 1 minute for it to work and render your initial Markdown.
On your first visit, the Welcome Screen lets you choose light/dark mode and access the editor. Your OS preference will determine your initial preference. You'll never see this screen again.



The Editor Screen lets you edit Markdown and preview it live. The initial visit will have example Markdown to demonstrate some of the functionality, and following visits will contain whatever was in your editor. Use the "Load Markdown" button to load a Markdown or text file from your computer, and the "Save Markdown" button to save the editor's Markdown to your screen. Switch between light and dark mode using the top-left icon. Top right selector box lets you choose your typeface. Last, click "Clear Markdown" in the bottom right of your Markdown editor.

All done! Happy Markdown editing with GoodMark!

Technical

The Markdown editor is a single repository with `/frontend` and `/backend` sections. The frontend uses Next.js and has two main pages: index and `/editor`. The backend uses Express.js and has two endpoints: `POST /render` and `GET /fonts`. All Markdown

rendering is done server-side, while the frontend receives HTML.

Setup Instructions

Setting up GoodMark locally requires:

1. Install Node.js.
2. Install git (https://git-scm.com/).
3. Clone the git repository with `git@github.com:Dinoosawruss/software-engineering-summative-1.git`.
4. In the `/backend` directory, run `npm install`.
5. Launch the backend server with `npx nodemon index.js`.
6. In the `/frontend` directory, run `npm install && npm run build`.
7. Create a `.env.local` file with `NEXT_PUBLIC_BACKEND_URL=http://localhost:5000`.
8. Use `npm run dev` to start the frontend server.
9. The frontend and backend should now be available at http://localhost:3000 and http://localhost:5000.

Deployment and CI/CD

Three main CI/CD pipelines validate pull requests. First, a dependency check GitHub Action will ensure the project has no active security vulnerabilities. Next, a Build and Test check will build the project at Node.js 18, 20, and 22. While these execute, Render will deploy the project automatically. Press "View Deployment" to see this. This deploys your PR online. Once your PR is merged, these deployments will rebuild, test, and deploy to the main deployment again.

Testing

A complete unit test suite exists. Test-driven development should be used for all development to ensure code coverage and suite strength. Use `npm test` to run tests on both the frontend and backend. Both use jest for testing, with the frontend utilising `swc/jest`.

Narrative

In this section, I'll explain how I implemented the project in each sprint.

Sprint 0

Sprint 0 was dedicated to project setup and Software Engineering "admin". This featured issue templates, branch protection rules, README writing, and CI/CD pipelines including build, dependency test, and deployment. This sprint was the longest because it required a lot of pre-work, but it laid a solid foundation for future sprints. It also involved creating Figma UI designs available here and all Epic, Feature, and Infrastructure issues for the project. I then assigned all sprints.

Allocating Sprints

After deciding on all my issues and epics, I had to assign each Epic to a Sprint to start Sprint 1. This procedure focused on Epic importance to the previous one. It would be irrational to sprint the backend epic after the performance epic because it must be done first. I concluded that epics had a natural order. Sprint 1 consisted of Backend and Sever Functionality (#32) and Markdown Editing Core Features (#28), which were

necessary before continuing. UI and Accessibility (#36) was a lot of work and may affect performance, therefore I would finish it in Sprint 2. I chose Performance (#38) and Welcome Screen and Personalisation (#37) for Sprint 3. Markdown Editing Core Features (#38) required Setup React Frontend (#39) to be accomplished outside of Sprint 2, the only notable feature issue. The Epics' natural order seemed good to me, so I started Sprint 1.

Sprint 1

Sprint 1 started slowly due to issues with Sprint 0's CI/CD pipelines, but once these were resolved, I was able to start the project and add basic features. I built the backend using Express.js and implemented the `POST /render` endpoint using `marked`, a library that converts Markdown to HTML. I used Test Driven Development (TDD) to create the endpoint. First, I implemented a test to ensure that `# Hello` returns `<h1>Hello</h1>`, then implemented the answer to that test. After this, I set up the Next.js React frontend and frontend deployment as Sprint 0 had only set up backend deployment. Finally, I added Markdown rendering and calls to the `POST /render` backend. This Sprint 1 task was the most important and produced a pleasing result. Like the backend, I used TDD to add and test items. I then added Save, Load, and Clear Buttons. Final Sprint 1 addition: Prism.js syntax highlighting. Due to its large initialisation and implementation workload, some of this sprint ate into Sprint 2.

Sprint 2

Sprint 2 began with Sprint 1 work wrapping up. This contained syntax highlighting and Save, Load, and Clear button operations. I implemented font selection after this carry over work. To regulate font availability, the site created a backend endpoint `GET /fonts` and made changes to the front end to request, load, and change fonts. The process took longer than expected due to trying to use "Courier Prime" monospace, however the website would not render due to a missing comma. After fixing that issue, I added a light and dark mode setting based on your system choice and then allowed you to move between them. Accessibility features, such as ARIA components, keyboard navigation, and shortcuts `Ctrl + S` and `Ctrl + O`, were added after significant effort. I tested Chrome's and Windows Narrator's accessibility. Finally, I made the site responsive with portrait and landscape views to stack the editor and markdown preview on portrait devices.

Sprint 3

Sprint work could finally commence. This Sprint had two key themes: Welcome Screen and quality of life, and performance. I started with the Welcome Screen, which was easy to construct because I could reuse many items from the Editor screen, but it required some sophisticated file structure and testing suite adjustments. After fixing that, I could create the Welcome Screen and automated forwarding after the first visit. I then added quality-of-life features like preserving your dark/light-mode option, last Markdown in the editor, and last font selection. The implementation was straightforward as I utilised `localStorage` instead of cookies, which provides a simple key-value API. In the Bugs bust, I resolved some major issues with the testing suite, which was more complicated

because I had constructed a massive testing suite and any issues would compound and cause numerous failures. Performance was next, focussing on backend scalability. To establish the deployed instance baseline, I ran scalability tests. Due to its free status, the deployed instance had limited resources, hence it was vital to establish a baseline response time. I then implemented caching, error handling, and async API calls to improve performance. I sanitised the HTML answer to prevent Cross Site Scripting. Following these performance adjustments, I ran a scalability test and found that the deployment could handle 90 requests per second without HTTP losses. Overall, the performance was good, and the MVP was applied.

Bugs

TDD helped me catch many regressions and errors during development, minimising the number of issue tickets I had to file and fix outside the sprint. However, some emerged. Early in development, the frontend used `http://localhost:5000` for the backend, even on deployed instances. I implemented `dotenv` to pull the URL from a `.env` file, avoiding users from deploying the backend on their own machine. This worked well and required little deployment. No tickets were needed because I only detected problems during the development cycle, however during Sprint 3, I uncovered bugs and filed issues. I fixed these issues in one huge Pull Request. App bugs include the dark and light mode selector not saving the last state, the markdown preview centring after forwarding from the welcome page, malfunctioning keyboard controls, the Clear Markdown button not clearing the preview, and the default Markdown being added when the local storage key is empty. It was excellent to resolve all these issues. I wondered how these issues had gone through, and then I saw they were in locations with few unit tests and no test cases, so I added test cases to prevent regressions. In portrait mode, the font is hidden. Several CSS tweaks failed to fix this issue, so I left it open.

Evaluation

Overall, I think this effort was successful. It taught me React and Express.js and improved my understanding of Test Driven Development and web-service development. I think my app has a good testing framework, is useable, and meets my User Stories. I've also used GitHub's project management tools to track my progress and update the board. I overcame most of the technical obstacles of building the tool, including bugs, testing issues, and learning Next.js and Express.js. I think my project implementation was good, although I might have made Sprint 1 smaller and added a Sprint to reduce time pressure. I also think I should have made the Sprints 2 or 3 days longer to give me more time to finish. Other than these few issues, I think the project has been successful, and an open source contributor could easily examine this README and utilise the tools to start contributing. I think my personas would like the tool and that I met their user stories.