GNG2101

Design Project User and Product Manual

Companion Accessibility Tool

Submitted by:

Companion Dev Team, GNG2101B Group 1.5

Jonathan Soliman, 300345528

William Levesque, 300352375

Andrius Avenido, 300344515

Aidan Nikolaus, 300309124

Jacob Cearns, 300361219

December 1, 2024

University of Ottawa

Table of Contents

List of Acronyms and Glossary

Table 1. Acronyms

Acronym	Definition

Table 2. Glossary

Term	Acronym	Definition
User Interface	UI	The visual elements of an application that users interact with, like buttons and menus.
Application Programming Interface	API	A set of protocols that allows different software to communicate with each other.
Comma-Separated Values	CSV	A file format used for storing tabular data, where each line represents a row and columns are separated by commas.

1 Introduction

This User and Product Manual (UPM) provides the information necessary for future users, developers, and engineers to effectively use and view prototype documentation for Companion, an accessibility-focused chatbot and resource website designed to assist professors and educators in improving digital accessibility of course materials. Built with the following popular web technologies such as MongoDB, Express.js, React.js, Node.js (MERN Web Stack), and utilizing OpenAI Services, this tool combines accessibility resources with a powerful chatbot interface to simplify the process of identifying, evaluating, and improving coursed content accessibility.

Key Assumptions made for this manual include:

1. Users have a basic understanding of web interfaces and technologies
2. Users possess working knowledge of Canvas Ally and are familiar with its role in evaluating accessibility requirements
3. Professors and Educators may not have in-depth technical expertise, but aim to improve accessibility compliance in their materials
4. Users understand OpenAI services and the role of the GPT-4o model

This manual serves the following purposes:

- Educate users on how to navigate and utilize Companion efficiently
- Clarify how to upload and analyze CSV files from Canvas Ally to address accessibility benchmarks
- Act as a reference for core features and settings
- Providing detailed prototype documentation for Companion

This document is intended for:

- Professors and educators aiming to learn how to use Companion in length with Ally to empower them with the ability to improve their course accessibility
- Developers and Technical teams involved in maintaining or enhancing Companion

Note: Familiarity with Canvas Ally and its role in accessibility assessments is essential for users to effectively interact with Companion.

Security, Safety, and Privacy Considerations

Companion is designed with user security, safety, and privacy as priorities:

1. **Data Security**:
    - Uploaded CSV files are processed securely and stored temporarily for analysis. Files are not stored permanently unless the user explicitly saves their data.
    - Communication between the user's browser and the server is encrypted via HTTPS.
2. **Privacy Protections**:
    - User interactions and chat histories are saved only with explicit consent and can be deleted by the user at any time.
    - The chatbot leverages OpenAI services, which comply with robust privacy standards.

3. **Accessibility and Safety**:
    - Customizable themes ensure inclusivity for users with varying abilities.
    - The chatbot interface is built to comply with accessibility standards, including WCAG.
4. **Authentication Security**:
    - User accounts are secured with strong authentication protocols, and sensitive data (e.g., passwords) is hashed and stored safely.

By incorporating these measures, Companion ensures a safe, secure, and accessible experience for all users while prioritizing privacy and data protection.

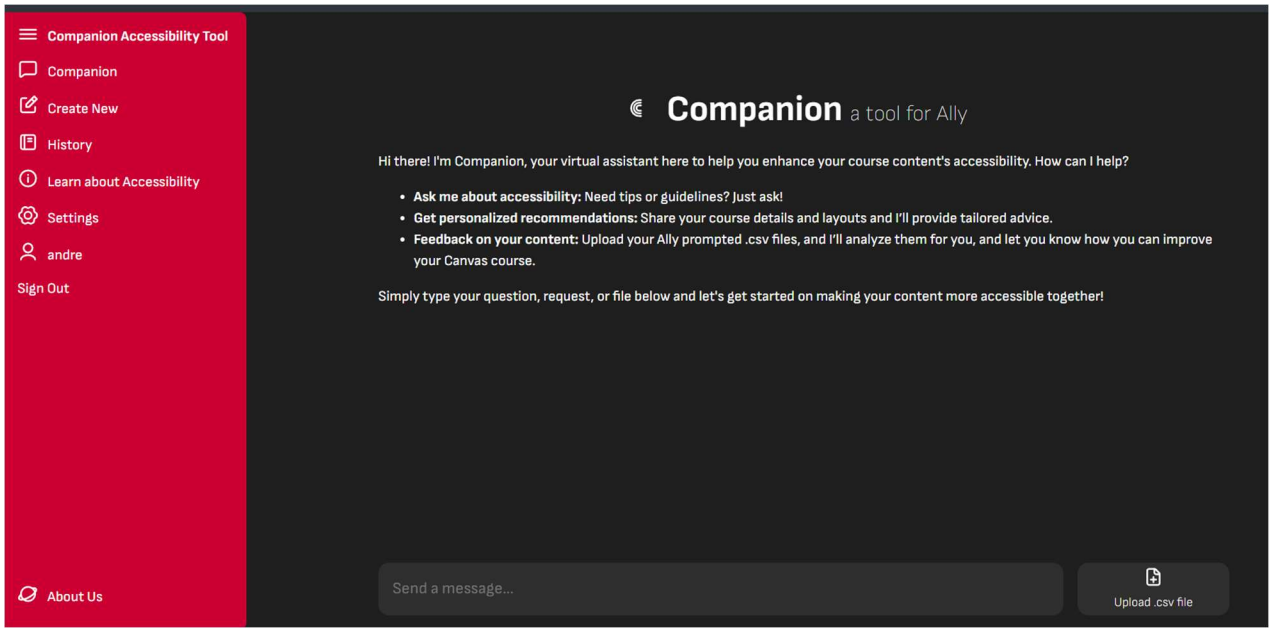
2 Overview

Accessibility in digital education platforms is a growing concern, yet many tools provide technical, overwhelming, or unclear feedback. Professors often struggle to interpret these recommendations, leading to unaddressed accessibility gaps that affect students with disabilities. This makes it harder for institutions to create inclusive learning environments, impacting student engagement and equity. The Companion Accessibility Tool addresses this problem by simplifying accessibility feedback, making it actionable, understandable, and easy to implement. Companion is specifically made to help analyze Canvas Ally csv files. This helps educators ensure their materials are accessible, improving the learning experience for all students.

Fundamental Needs of the User

1. **Clarity:** Professors need clear, concise, and actionable feedback on accessibility issues.
2. **Efficiency:** Users require quick and simple solutions to make their course materials compliant without significant time investment.
3. **Customization:** A flexible tool that allows personalized themes and settings for ease of use, such as font size or dark mode.
4. **Integration**: A system that can easily process data from existing accessibility programs like Ally (in Canvas) without needing advanced technical expertise.
5. **Interactive Support**: A chatbot or interface that can provide real-time assistance and answer questions about accessibility.

Screenshot of Final Prototype



What Differentiates Companion from Others

1. **Actionable Guidance**: While other tools focus on technical details, Companion simplifies feedback into practical steps.
2. **Chat Interface**: The chatbot provides an interactive, conversational way to understand and fix accessibility issues.
3. **Customization**: Features like adjustable font size, themes, and saved conversations cater to accessibility for the user themselves.
4. **Multi-Functionality**: Beyond just feedback, it serves as a hub with accessibility resources, benchmarks, and interactive tools.
5. **Focus on Education**: Tailored specifically for educators in higher education to align with their workflow and goals.

Key Features or Major Functions

- **CSV File Processing**: Professors upload files from Ally or similar tools for detailed feedback.
- **Chatbot Interface**: Users receive feedback in a conversational format powered by GPT-4, tuned for accessibility.
- **Customization Settings**: Options for font size, themes, and dark mode to enhance usability.
- **Resources and Tools**: A built-in library of accessibility guidelines, benchmarks, and interactive components.
- **Conversation Saving**: Users can save chat histories to refer to prior feedback.

System Architecture and User Access Mode

System Construction: The Companion Accessibility Tool is a **web-based application** built on the **MERN stack** (MongoDB, Express.js, React, Node.js). The backend leverages the ChatGPT API for processing conversational feedback.

- **User Access Mode**:
    - Professors interact with the system through a **graphical user interface (GUI)**.
    - **CSV file uploads** allow accessibility data input.
    - Chatbot feedback and settings are available via the website interface.
    - Saved conversations are accessible when user is logged in

- **Special Conditions**: The system is designed to be fully accessible itself, adhering to WCAG standards for inclusive design.

2.1 Conventions

<desc>: These tags are used throughout the document and pictures; they indicate placeholders for information for the user to provide such as put your password here: <password>.

2.2 Cautions & Warnings

API Key for OpenAI Services

Users must obtain a valid API key from OpenAI to access the ChatGPT API. Without the key, the backend functionality relying on the API will not work.
API keys are unique and must be kept secure to prevent unauthorized usage. Sharing your key may result in excessive charges or service disruptions.

Action:

- Visit OpenAI's API Page to create an account and generate an API key.
- Store the key securely, typically in an environment file (.env), and avoid hardcoding it directly into the application code.

Connection to MongoDB

The MongoDB database may require your IP address to be added to its access list to allow proper backend functionality. Without this, the application will fail to connect to the database.

Action:

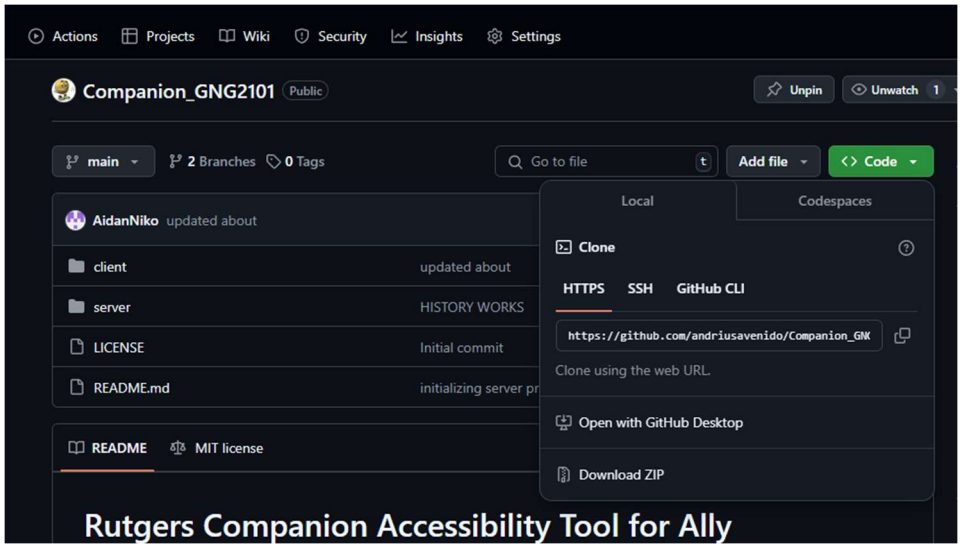
- Ensure your IP address is whitelisted in the MongoDB cluster's network access settings. This can be managed through the MongoDB Atlas interface or your self-hosted database settings.
- Refer to the official MongoDB documentation for details on managing network access: https://www.mongodb.com/docs/.

3 Getting started

The source code for Companion can be found at https://github.com/andriusavenido/Companion_GNG2101. This section will walkthrough how to run Companion locally. Important information, software requirements, and extra steps can be found after this piece.

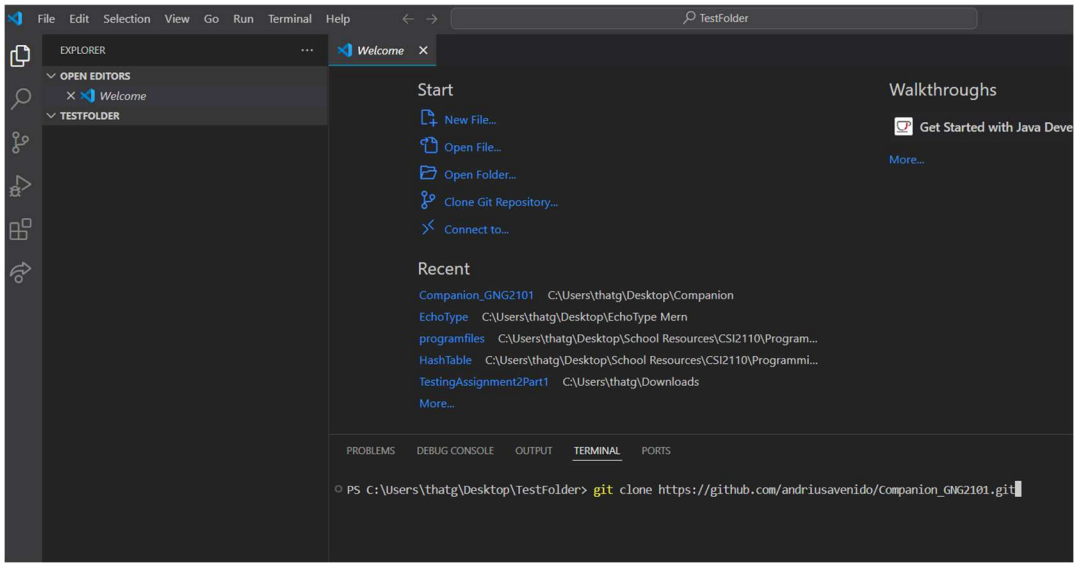
Downloading the Source Code

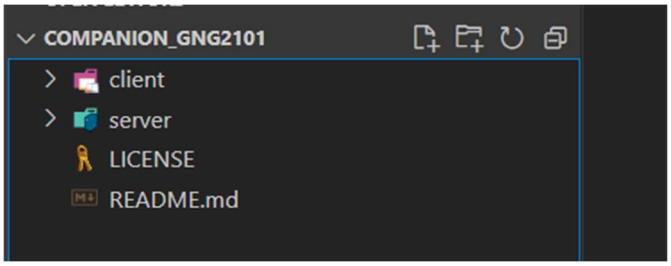
1. Navigate to https://github.com/andriusavenido/Companion_GNG2101 to view the source code.
2. Click the green “Code” button and its local section. Make sure that HTTPS is selected. Copy the link show. This will be the link to the source code that we will tell **Git** (view below for more information) to download the resources from.



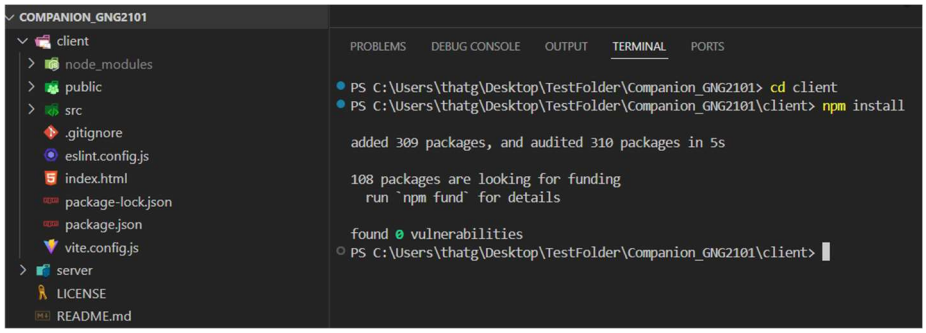
3. Create a folder on your computer (name it anything). Open Visual Studio Code and open the folder. Click “Terminal” and open a new terminal and write in the line: git clone

<link>. All of the source code should be downloaded.



4. Open the inner folder inside the folder you are in so that the source code folder is the main thing VS Code will be viewing as such you should see only two folders and some extra files. 

The website code is stored in client, and the server code is stored in server. These two folders run two separate applications that talk to each other. Both will be running when using the website.
5. We now need to download additional resources. The source code hosted on GitHub does not contain a folder called “node_modules” for both the client and server folders. This folder contains important code that will help run the project (this code was created by other open-source resources and can accumulate in large size; therefore, it is typically not stored online).
6. Open your terminal once more. Write “cd client”. Your terminal is now looking at that folder specifically. Write the command “npm install”, and the node modules will be

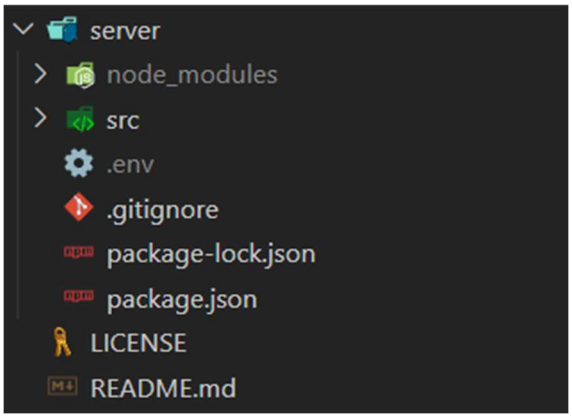
downloaded. 

Do the same for the server. Open a new terminal, type “cd server” and run the command “npm install”

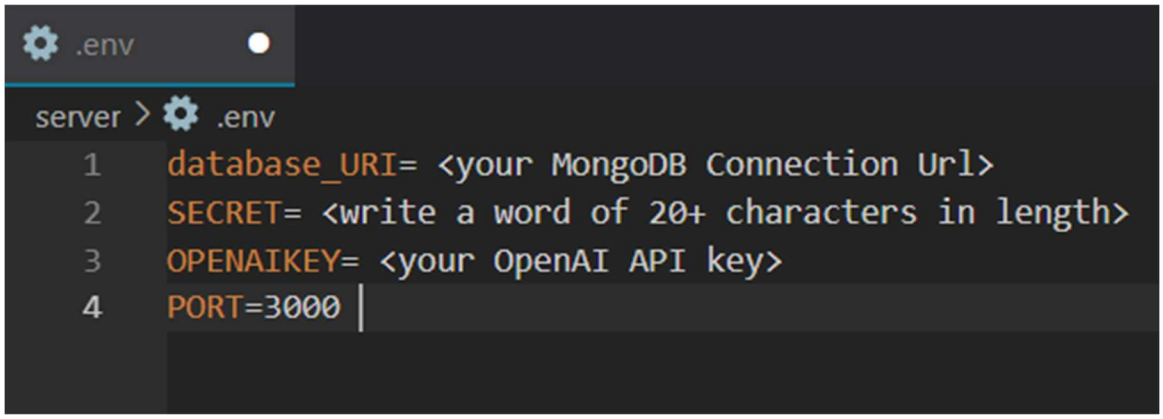
7. Congratulations, you have now downloaded the source code on your computer locally. You can review the code and make changes. Before running the website, we need to do more important security configurations.

Configuring Specific Files

1. To run our server, we need to setup a file that will allow us to store our sensitive information such as keys and passwords.
2. Open the server folder and create a file called “.env” at the top most directory area.

Your folder should look like this 

3. In that .env file, write the following text.



   a. database_URI: paste your MongoDB connection URL here
   b. SECRET: here write any word over 20 characters. This is a component used to hash passwords. It can be anything.
   c. OPENAIKEY: paste your OpenAPI Key
   d. PORT: write 3000.
4. Save the file (CTRL + S)

Congratulations! You have successfully downloaded the source code on to your local computer. You are now ready to run and use Companion.

Running the source code can be found below.

3.1 Configuration Considerations

Important considerations and requirements should be acknowledged when trying to run Companion. While you will interact with developer tools, no programming skills are required. Furthermore, keep and password or key or link found when accessing these different sites very secure and private as they maybe (and probably are) sensitive.

1. **OpenAI API Key and Service**
   a. Companion uses OpenAI's services to access the GPT-4o Large Language Model to power its chatbot
   b. You will need to sign up for an OpenAI account and generate an API key. Payment will be required to enable the key to have access to GPT-4o. Can be found at https://platform.openai.com/docs/overview
   c. **IMPORTANT:** Keep this key private and secure, it is required for the chatbot to work.

2. **MongoDB Atlas**
    a. Companion uses a cloud database to store information like user accounts and saved chat histories.
    b. MongoDB Atlas is a free, cloud-based database service that will be required to setup, found at https://www.mongodb.com/products/platform/atlas-database
    c. You'll create an account and setup a "connection string" which tells Companion where to find our database.
3. **Git and Github**
    a. Git is a tool used to download the Companion source code which is hosted on GitHub. Download Git here: https://git-scm.com/
    b. Expertise in Git is not required, installing and using simple commands as outlined above is enough. Here is a resource: https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository
4. **Node.js and Node Package Manager (NPM)**
    a. Node.js is a platform that allows Companion's server code to run locally on your computer.
    b. Node package manager comes bundled with Node.js and is used to install additional tools and resources with single simple commands. It will help you run everything including the front-end (visual webpage) and back-end (server) on your computer. https://nodejs.org/en/download/package-manager
5. **Visual Studio Code (VS Code)**
    a. VS Code is a free and easy-to-use code editor that allows you to view and modify the source code easily. You will use it to view and run the source code effectively and make minor changes to specific config files (such as adding in your API key).
    b. Beginner-friendly and great for managing the project: https://code.visualstudio.com/download
6. **A Modern Web Browser**
    a. Companion works through a website interface, so you will need a modern browser like Google Chrom or Microsoft edge to use it.

3.2 User Access Considerations

The Rutgers Companion prototype is designed for diverse user groups: professors and educators seeking to improve course material accessibility using Canvas Ally data; administrators and accessibility specialists overseeing compliance and providing feedback; students and teaching assistants assisting with content preparation; developers and technical support teams maintaining the system; and accessibility advocates using the tool for training or evaluation. Access varies by

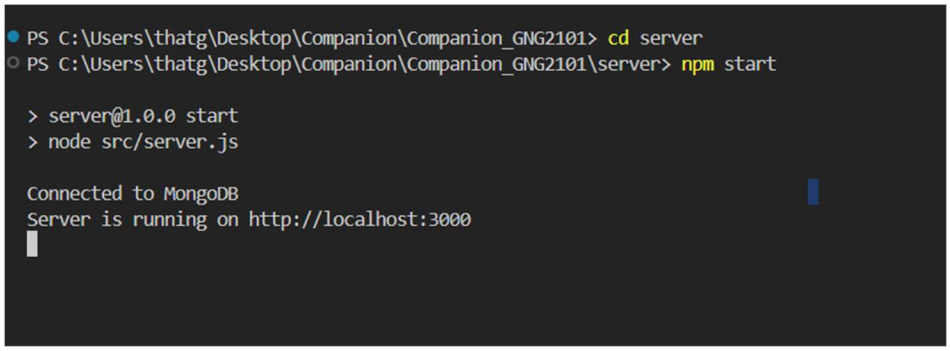
role—professors and students need working knowledge of Canvas Ally, administrators may require broader permissions, and developers need advanced access for debugging and updates. All users require an OpenAI API key, authentication credentials, and a modern web browser. Restrictions include limited access to sensitive data, demo-mode limitations for external advocates, and adherence to privacy and security protocols.

3.3 Accessing/setting up the System

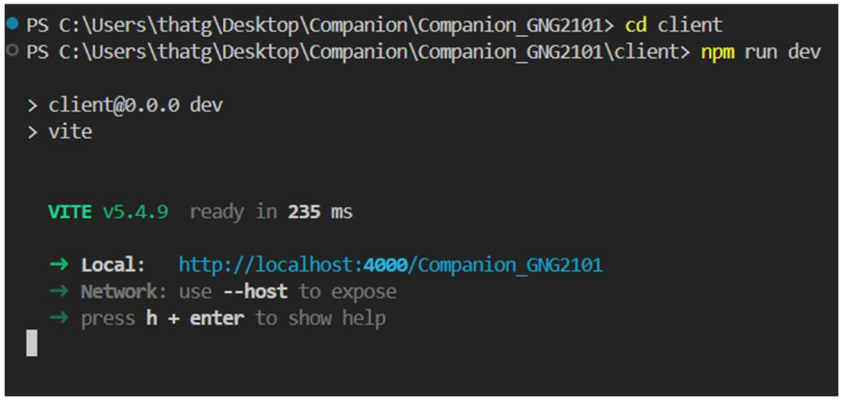
Running the Source Code

Running the source code is very simple and just requires a few commands.

1. Open a new terminal. Write “cd server” and then enter the command “npm start”. Your server should be running



2. Open another terminal (don’t close the server one). Write “cd client” and then enter the command “npm run dev”. Your front end should now be running. Press the link and it should direct you to the Companion webpage running locally on your computer.



Information on how to use Companion itself can be found in Section 4.

3.4 System Organization & Navigation

Here is a quick explanation of our project setup. As explained, there are two main folders for this project in the source code: client and server. The client and server software will be run simultaneously, and the client will talk to the server when doing tasks such as talking to GPT-4o. Client has all the website code and Server has all the server code:

Client Side: This is the part of the application that users interact with directly (what you see on the screen).

- There are multiple pages used for navigation: Home → About → Accessibility → History → Options.
- Page Navigation is controlled by a file called App. Other pages and systems of the client side can call this to move pages directly.
- Each page has their own section for website design (what it looks like) and a section for code logic (what the page will do).
- There are CSS files used to style the pages (like what color this button should be etc.)
- The top folder has different specific software-related configuration files
- There are multiple different folders and files used for more complex logic and processing which is explained in more detail under prototype design.

Server Side: This part runs behind the scenes and handles things like saving data, processing requests, and communicating with other services (like OpenAI).

- There is a main file called server. It contains the core logic and settings such as setting up database connection etc.
- There are files called controllers. They are the managers of the server.
    - Conversation Controller: Manages logic for saving and retrieving conversations
    - User Controller: Manages user tasks such as logging in and registering.
- There are files called Models. They define how data is stored in the database such as a user model which describes how things such as password should be stored.
- There are routes, that manage the HTTPS requests sent by the client.
- Finally, there are services tasked with specific tasks. Such as the OpenAI service file which contains all related logic towards talking to GPT-4o (specific rules and prompts).

3.5 Exiting the System

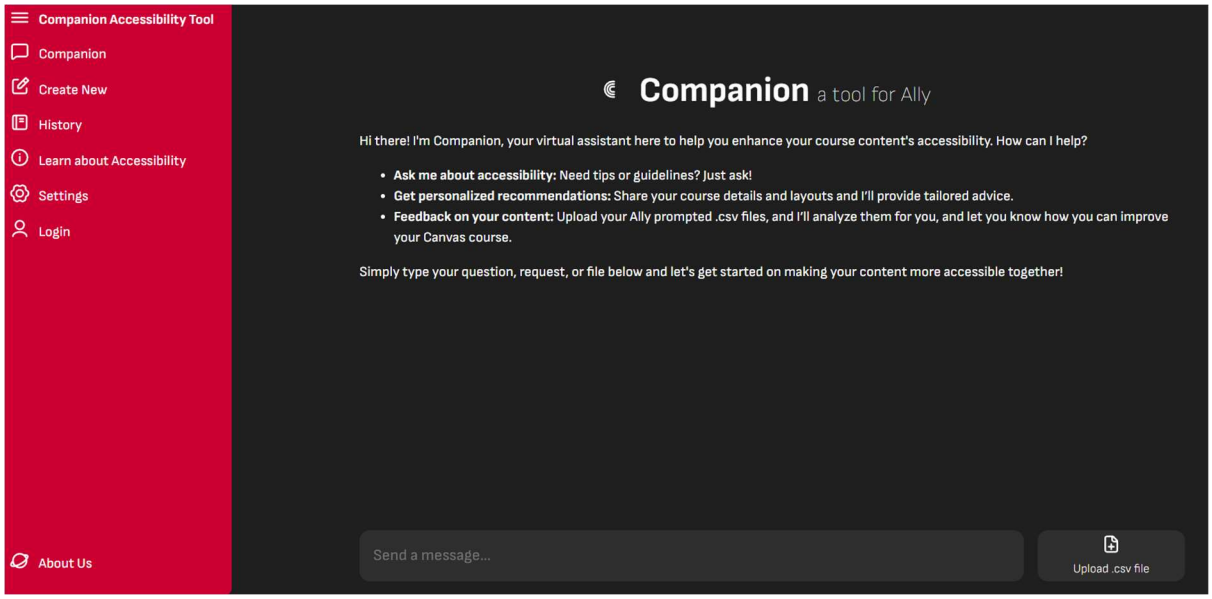
Turning off the locally run server is very simple. All you must do is exit/close your code editor, VS Code, which will stop your system from running the software in the terminals. MongoDB and OpenAI services run 24/7 so there is no requirement to close them.

4 Using the System

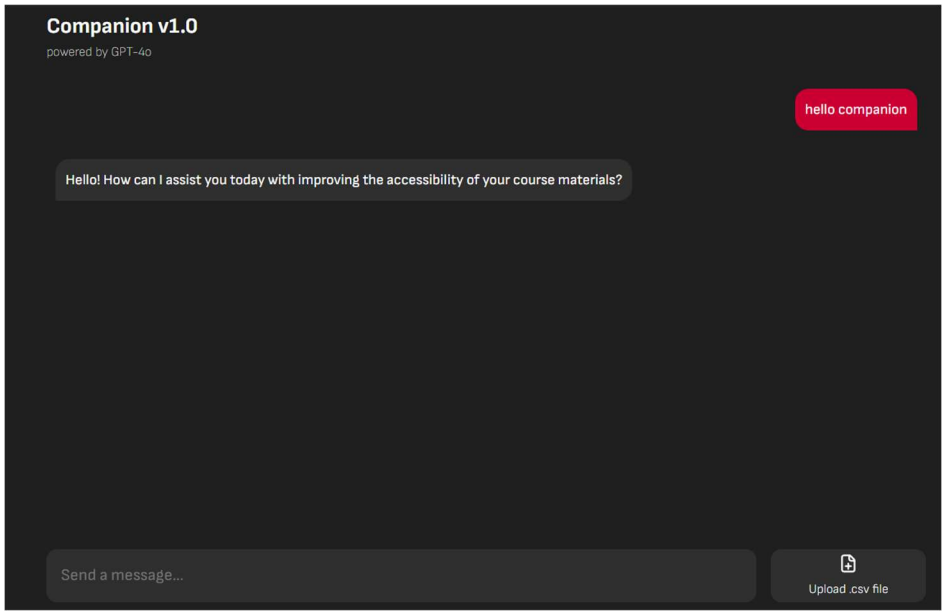
The following sub-sections provide detailed step-by-step instructions on how to use the various functions or features of the Companion Accessibility Tool Website.

4.1 Companion Virtual Assistant

The home page and landing page of the website features the companion virtual assistant. It features a text input box where users can type in messages and an upload drop box where users can upload their companion csv files.



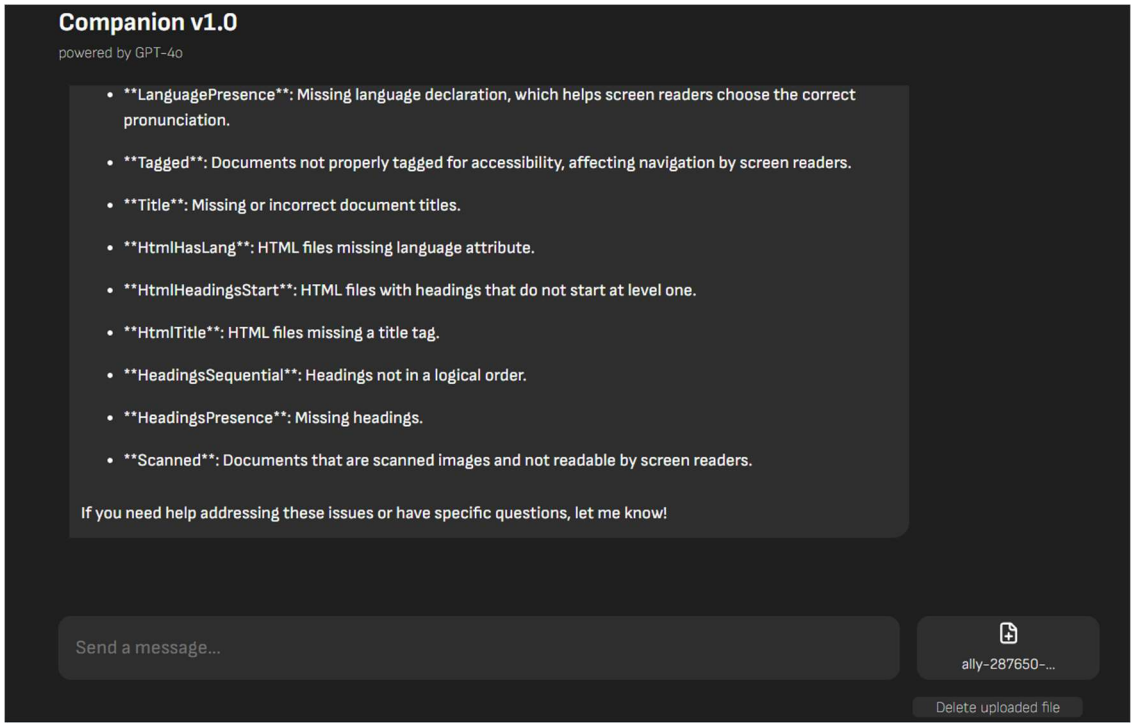
To begin, type in a message and press Enter. Companion will respond to your message.



You can ask it anything regarding to accessibility and it will respond effectively. Companion can remember the entire conversation and what it analyzed.

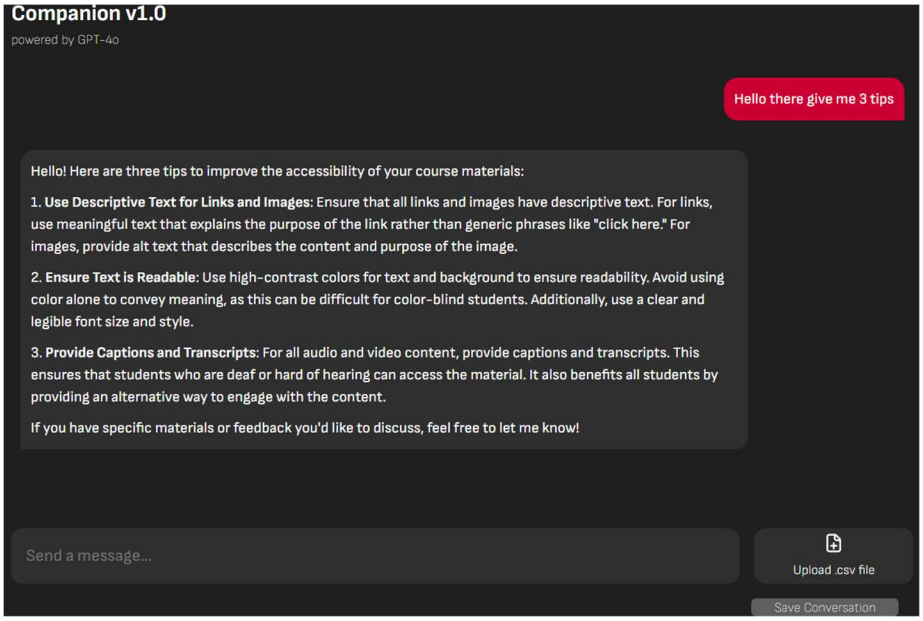
4.1.1 File Uploading

To upload a file, simply press the upload csv file. A file window should prompt you to select your specific file, press open and it should be saved on the Companion website. Now you can ask Companion to analyze it. You can delete your file with the button or upload a new one to override it.



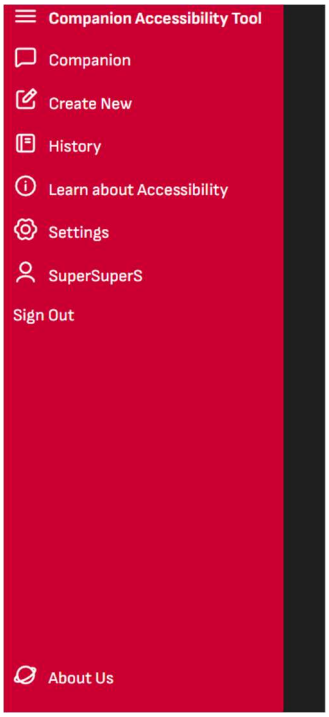
4.1.2 Saving Conversations

When logged in, you can save the state of the current conversation by clicking the save conversation button at the bottom of the page.



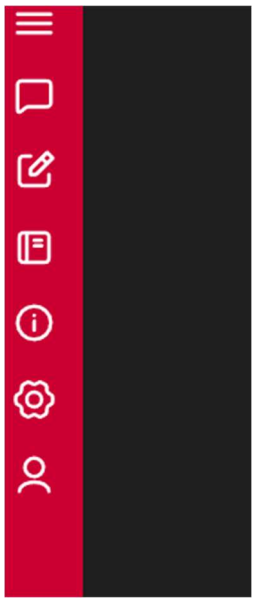
4.2 Sidebar/Navbar for Page Navigation

Users can look at the left of the webpage to see the Sidebar. They can click on any section to navigate to that page effectively.



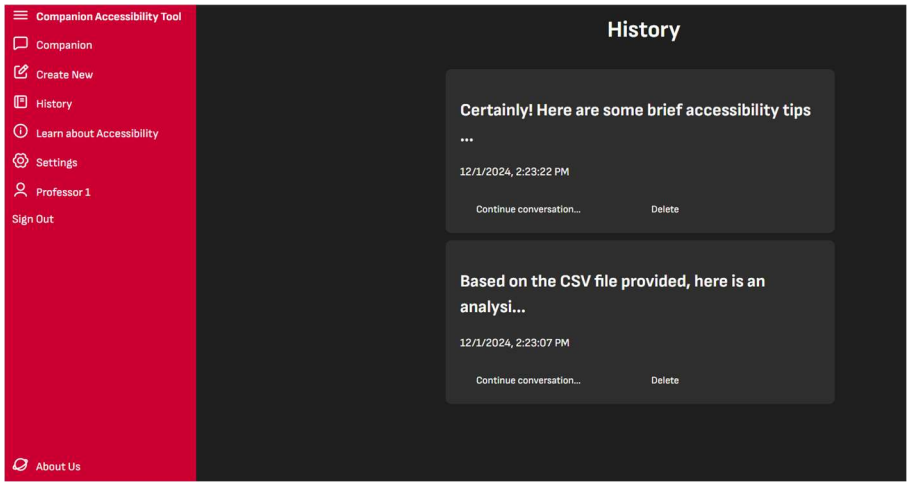
4.2.1 Toggle Expansion

Press the Hamburger Icon (3 horizontal lines stacked) on the side bar to close it. It can be pressed again to open it.



4.3 History Page

The history page contains all the conversations that users have saved. They can scroll through them using the scroll wheel if there is a lot. They can also hover each section.



4.3.1 Continue Conversation

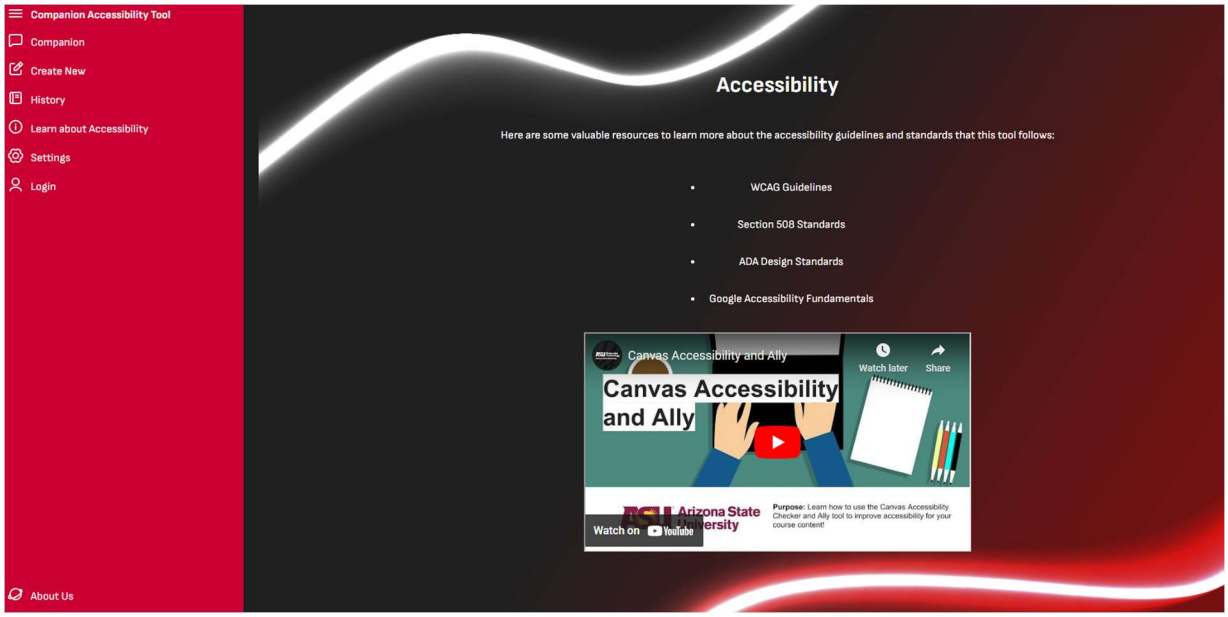
Press the Continue Conversation… link to continue the conversation. You will be redirected to the chat interface with the conversation history selected.

4.3.2 Delete Conversation

Press the delete link on any conversation to delete it.

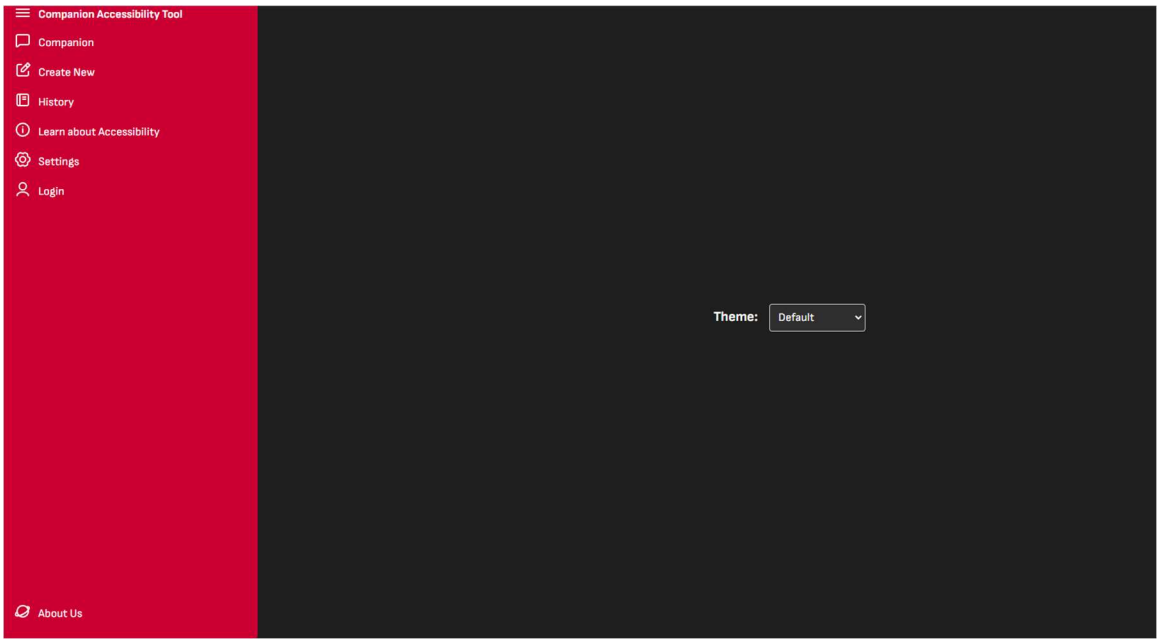
4.4 Accessibility Resources

The accessibility resources page provides users various links and resources to help them with accessibility rules. The can select any of the bulleted links which will redirect upon mouse press and can watch the video.



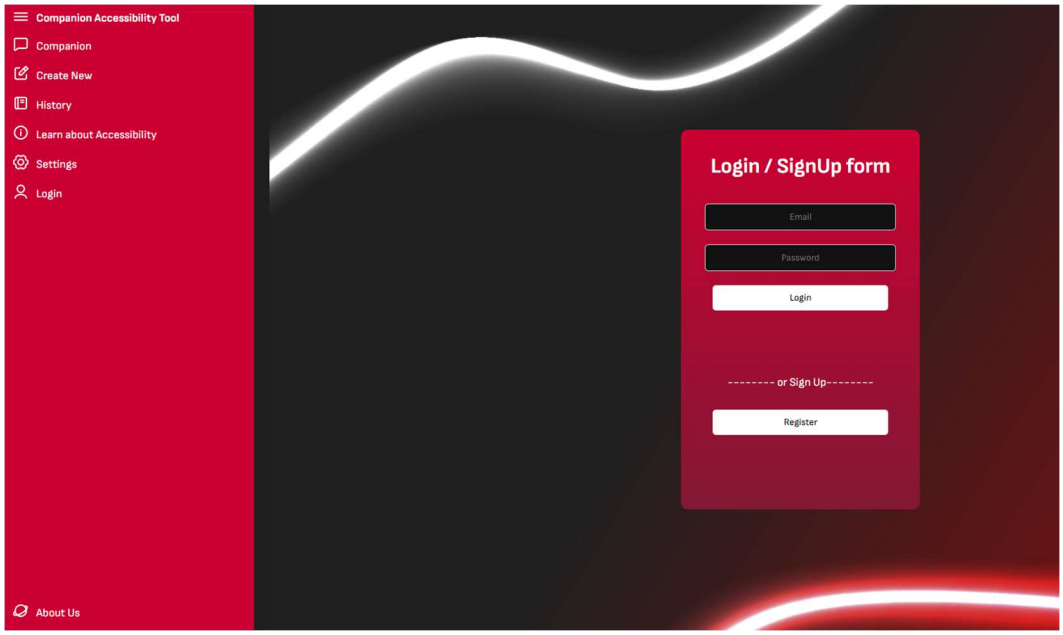
4.5 Theme Settings

The settings page allows users to select different themes for the website. They can click the menu, and it will drop down three options: Default, Light, and High Contrast. They can select any one and it will change the colors of the pages accordingly. Users can choose their preference.

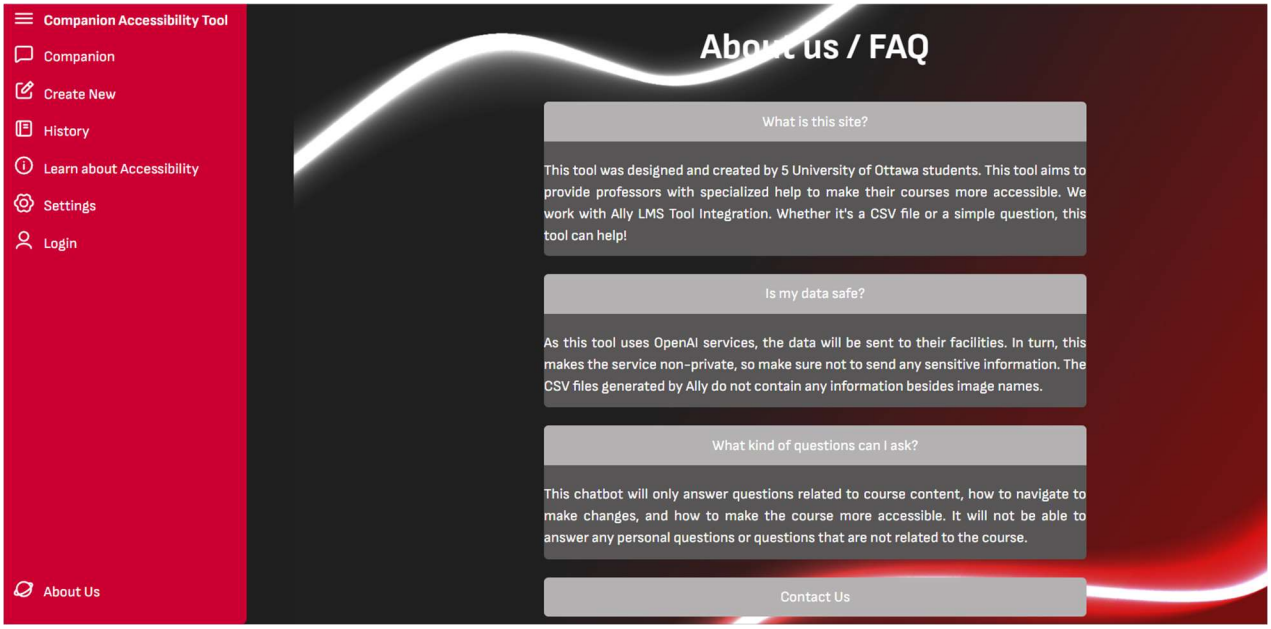


4.6 Login / Sign Up Form

The login/signup form is a simple, user-friendly feature that allows users to create an account or log in to their existing account. It ensures secure access to the Companion Accessibility Tool, allowing users to save conversation history. Users can input information into specific fields.



4.7 About Page



This is the about page where users can click on different sections to open up information about the website. They can also view an email to contact the Companion Team.

5 Troubleshooting & Support

5.1 Error Messages or Behaviors

Below are common error messages or behaviors a user may encounter, their likely causes, and corrective actions:

Frontend Errors:

- "Network Error": Indicates connectivity issues between the client and server.
    - **Cause:** Server downtime or client internet issues.
    - **Action:** Check the internet connection. If the server is down, wait for notification of resolution from support.
- UI Freezing or Non-responsive: The interface becomes unresponsive to user input.
    - **Cause:** Rendering conflicts or excessive client-side data processing.
    - **Action:** Refresh the page. If the issue persists, clear the browser cache or try a different browser.

Backend Errors:

- "400 Bad Request" *during Login*: The request sent to the server was invalid.
    - **Cause:** Missing or improperly formatted login credentials (e.g., empty email or password fields).
    - **Action:** Ensure all required fields are filled and formatted correctly before resubmitting.
- "401 Unauthorized": User authentication failed.
    - **Cause:** Invalid credentials or an expired token.
    - **Action:** Verify login credentials. If using a token-based system, reauthenticate to generate a new token.
- "500 Internal Server Error": General server-side failure.
    - **Cause:** Unhandled exceptions in Node.js, MongoDB errors, or ChatGPT API issues.
    - **Action:** Report the issue to support with a timestamp and error details.
- "Database Connection Error": Failure to access MongoDB.
    - **Cause:** MongoDB server is down or incorrect connection credentials.
    - **Action:** Restart the database server. Verify database credentials in the environment configuration file.

ChatGPT API Errors:

- "API Key Invalid or Missing": Unable to connect to ChatGPT API.
  - **Cause:** Incorrect or expired API key.
  - **Action:** Verify and update the API key in the backend environment settings.
- "Rate Limit Exceeded": ChatGPT usage has exceeded the allowed limit.
  - **Cause:** Excessive API calls in a short time frame.
  - **Action:** Implement rate-limiting or wait for the limit to reset.

5.2 Special Considerations

- Ensure the backend server and MongoDB database are hosted on reliable platforms with robust monitoring to minimize downtime.
- Monitor login flow and authentication token systems regularly to ensure no expiration or validation errors.
- For ChatGPT API, monitor usage limits regularly and implement fallback mechanisms for high-traffic scenarios (e.g., provide predefined responses).
- Maintain compatibility with assistive technologies to ensure a seamless experience for all users.

5.3 Maintenance

Regular maintenance procedures include:

- **Database Maintenance:**
  - Run weekly backups of the MongoDB database.
  - Monitor database performance metrics and resolve slow queries.
- **Codebase Maintenance:**
  - Review logs for recurring errors and address them promptly.
  - Update dependencies (Node.js, React, etc.) monthly to patch security vulnerabilities.
- **ChatGPT API Maintenance:**
  - Periodically check for updates to the API documentation.
  - Renew the API key before expiration.

5.4 Support

For resolving issues or gaining assistance with the technologies used in the application, users and developers can access the following resources:

JavaScript

- **MDN Web Docs**: Comprehensive documentation on JavaScript syntax, features, and APIs.
    - Website: https://developer.mozilla.org/en-US/docs/Web/JavaScript
- **Stack Overflow**: A community-driven platform for troubleshooting JavaScript-specific errors.
    - Website: https://stackoverflow.com

MongoDB

- **Official MongoDB Documentation**: Covers topics from database setup to advanced querying techniques.
    - Website: https://www.mongodb.com/docs/
- **MongoDB University**: Offers free courses on database design and usage.
    - Website: https://university.mongodb.com/

ChatGPT API

- **OpenAI Documentation**: Detailed guides on using the ChatGPT API, including examples for integration.
    - Website: https://platform.openai.com/docs/
- **OpenAI Support**: For troubleshooting API-related issues or account-specific queries.
    - Contact: Use the support chat feature on https://platform.openai.com/

React

- **React Documentation**: Official resources for understanding React concepts and building applications.
    - Website: https://reactjs.org/docs/getting-started.html

- **FreeCodeCamp React Tutorials**: Beginner-friendly tutorials and projects to enhance understanding.
    - Website: https://www.freecodecamp.org/learn/

Node.js

- **Node.js Documentation**: Offers guides on Node.js APIs and development best practices.
    - Website: https://nodejs.org/en/docs/
- **Node.js Slack Community**: A platform for developers to discuss issues and share solutions.
    - Website: https://nodejs.org/en/get-involved/community/

General Programming Support

- **GitHub Discussions**: Check repositories for open discussions on related issues or features.
    - Website: https://github.com/
- **W3Schools**: Beginner-friendly tutorials and examples for learning the core web technologies.
    - Website: https://www.w3schools.com/

These resources offer reliable support and guidance for troubleshooting and understanding the tools and frameworks used in this application.

6 Product Documentation: The Final Prototype

This section highlights the specific documentation for the design and implementation of the final prototype of the Companion Accessibility Tool. This section is specifically designed for developers or engineers looking to recreate the Companion Accessibility Tool; thus, a level of web design and development is assumed.

Here are the various technologies required and discussed:

- **React**: A JavaScript library used for building user interfaces. It simplifies the creation of dynamic, reusable UI components and ensures seamless user experiences through virtual DOM updates. Contains its own subscript of HTML called JSX, allowing for more developer control.

- **Node.js**: A runtime environment that enables the execution of JavaScript code on the server side. It is used to handle backend logic and integrate the frontend with the database and APIs.
- **npm**: The Node Package Manager, used to manage dependencies and libraries required for the project. It simplifies package installation, version control, and dependency management.
- **MongoDB**: A NoSQL database used to store and retrieve data for the application. It is highly flexible and scalable, making it ideal for managing unstructured or semi-structured data.
- **Express.js**: A lightweight and flexible web application framework for Node.js. It is used to build the backend server and manage API routes and middleware.
- **HTML**: The standard markup language for structuring the web application's content. It provides the foundational elements for building web pages.
- **CSS**: A styling language used to design and layout web pages. It ensures the application's visual appearance is user-friendly and aligns with accessibility standards.
- **Vite**: A modern frontend build tool that is faster than traditional bundlers like Webpack. It improves development speed with its efficient Hot Module Replacement (HMR) and optimized production builds.
- **HTTPS**: The secure version of HTTP, used to ensure secure communication between the web application and the server by encrypting data transfers.
- **JavaScript**: The primary programming language for creating interactive and dynamic web functionalities in the frontend and backend.

Core Idea:

The Companion Accessibility Tool was planned with a range of complex features:

- AI Interactivity
- Chatbot Interface
- CSV File Uploads and Analysis
- Accessibility benchmarks and feedback
- Customizable User Settings
- User Authentication
- Saved Chat Histories
- Interactive Tools and Resources

To implement these features efficiently, it was essential to choose web technologies that supported rapid development while ensuring scalability and reliability. To meet these needs, the popular **MERN stack** (MongoDB, Express.js, React, and Node.js) was selected as the foundation for the website's development. This choice provided several advantages:

- **MongoDB**: A NoSQL database that offers flexibility for managing and storing diverse data formats, making it ideal for handling user data, saved conversations, and accessibility configurations.
- **Express.js**: A lightweight and robust web application framework for Node.js, simplifying the creation of RESTful APIs for seamless communication between the frontend and backend.
- **React**: A powerful library for building dynamic user interfaces. Its component-based architecture and virtual DOM capabilities enabled the creation of an interactive and accessible user experience.
- **Node.js**: A server-side runtime environment that allowed for fast and scalable backend development, providing the foundation for integrating AI models and handling authentication processes.

This combination of technologies not only streamlined the development process but also ensured that the planned features could be implemented in a performant, maintainable, and user-friendly manner.

Other Considerations

During the planning phase, various technologies and languages were considered:

- **Python**: Known for its simplicity and extensive AI and machine learning libraries, Python was a strong candidate for handling the AI interactivity features. However, integrating Python with modern web frameworks would have added complexity to the stack.
- **PHP**: A popular backend language for web development. While it provides robust server-side capabilities, its traditional use in monolithic applications made it less suitable for a modern, interactive, and component-based architecture.
- **Ruby on Rails**: Its rapid prototyping capabilities and developer-friendly environment were appealing. However, the lack of widespread adoption for AI and real-time interactivity presented challenges.
- **Java**: While powerful and scalable, Java's verbosity and heavier frameworks (e.g., Spring) would have slowed development compared to lightweight, modern alternatives.

Ultimately, **JavaScript** was chosen as the single language for the entire stack. This decision was driven by its ability to power both the frontend and backend seamlessly, enabling efficient data exchange via JSON (JavaScript Object Notation) and reducing the need for context-switching between different languages.

The **MERN stack** (MongoDB, Express.js, React, and Node.js) was selected as the foundation of development, complemented using **OpenAI services** for integrating AI features. This stack provided:

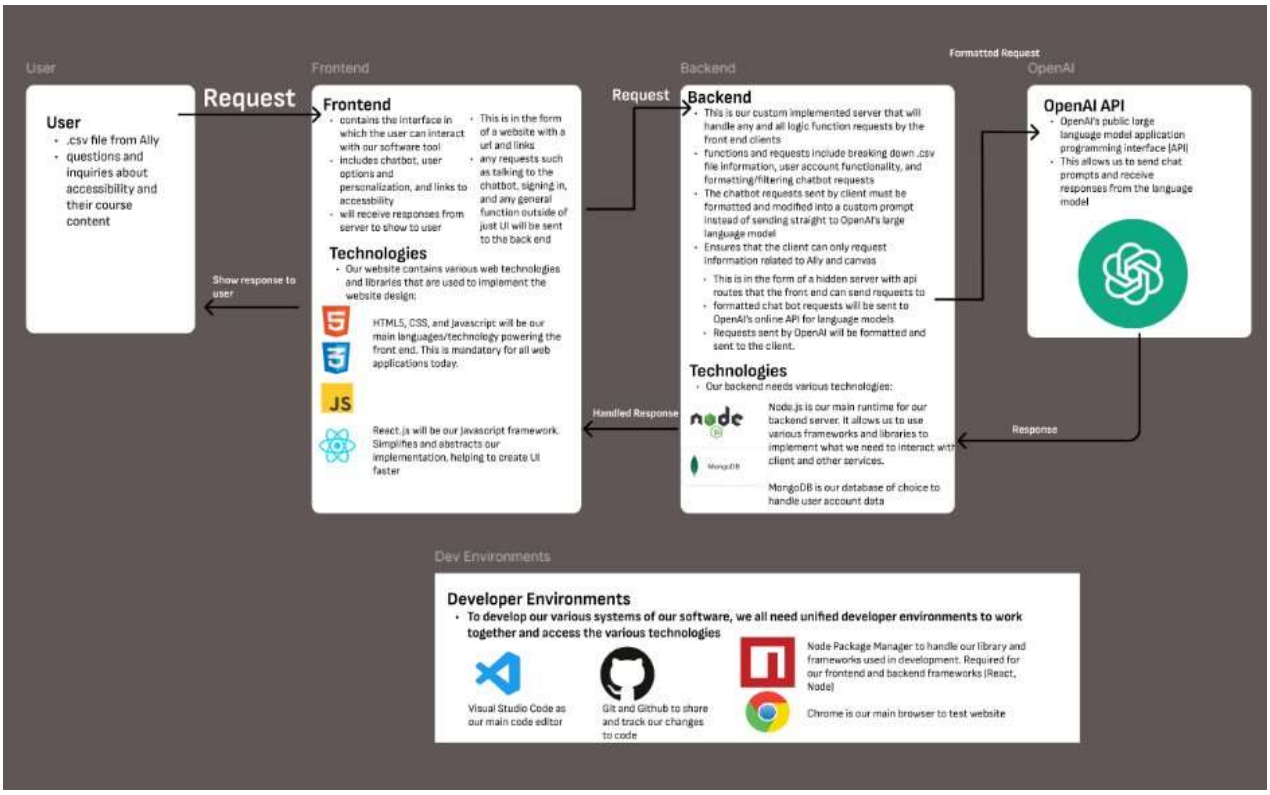
- A unified development environment, streamlining communication between the frontend, backend, and database.
- Native JSON support, enabling efficient data handling and API integration.
- Compatibility with modern tools and APIs, such as OpenAI services, for embedding AI interactivity into the platform.
- Easy Website Testing and easier integration with HTML and CSS

Overall, sticking to JavaScript not only simplified the development process but also empowered the use of JSON for data transmission and integration with cutting-edge services. This ensured the Companion Accessibility Tool could be built rapidly while remaining scalable and accessible.

Systems Diagram

Here is a detailed system interaction diagram to breakdown the sub systems of our project:

Figure #. System Interaction Diagram



****

IMPORTANT: Below is our break down of how the systems were designed. However, knowledge and how to use specific technologies are not discussed since they are assumed that the user can use and research them on their own**. <u>This is not a tutorial on the MERN stack, git, npm, or other related technologies!</u>** A guide on building a house wouldn't teach you how to use a hammer or how to mix cement, but would instead focus on the architectural plans, layout, and materials used to construct the house.

****

6.1 Front-End (Website Client Side)

This section breaks down the front-end implementation. All specific code and implementation can be found at https://github.com/andriusavenido/Companion_GNG2101/.

6.1.1 BOM (Bill of Materials)

Item Name	Units	Qty	Unit cost	Extended Estimated cost
GPT-4o API Key	API Tokens	1	$2.50 / 1M input tokens $1.25 / 1M cached** input tokens $10.00 / 1M output tokens	For one month of development testing, and assuming 50 requests a day (15 000 input and output tokens) is roughly $0.1875 per day or $5.65 for one development month
MongoDB Atlas	n/a	1	0$ for small scale projects	0$
Developer Tools	n/a	1	0$	0$
Total product cost (without taxes or shipping)				**$5.65 USD**
Total product cost (including taxes and shipping)				**$6.48 + $0 Shipping USD**

Note: This BOM is shared with the Back End Subsystem.

6.1.2 Equipment list

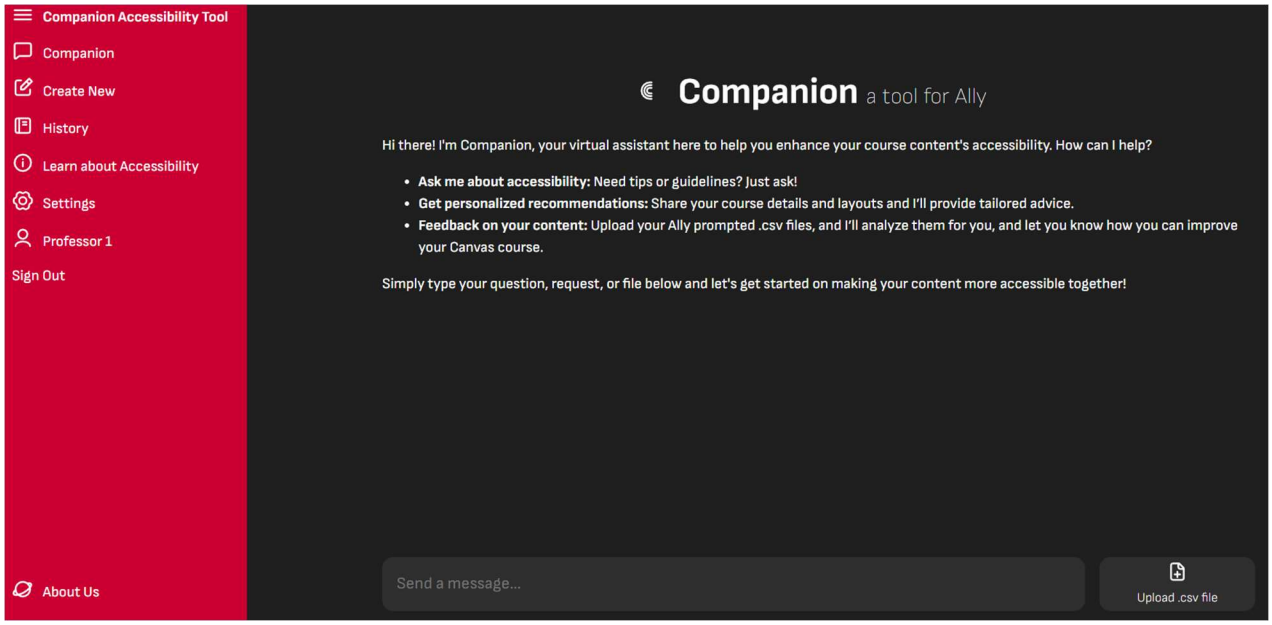
- **Visual Studio Code**
- **Modern Browser**
- **React and JavaScript Front-End Technologies**
- **Vite Web Bundler**

6.1.3 Instructions/Design Breakdown

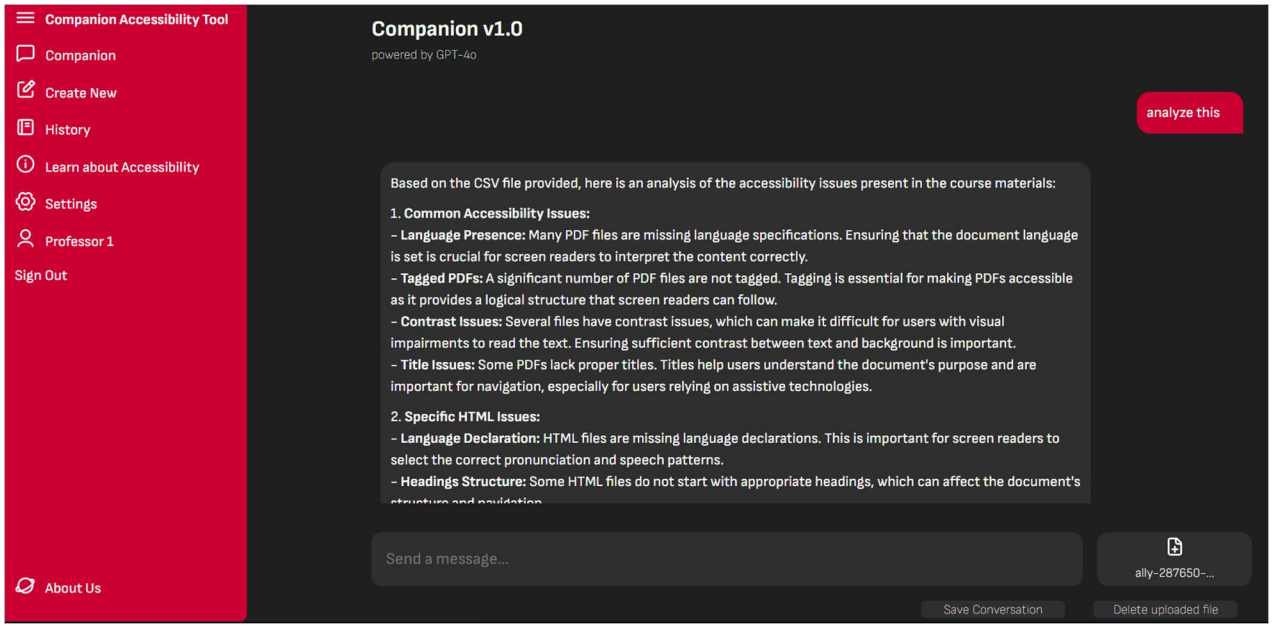
Interface Design

The focus was to create a user interface that is friendly, simple, and accessible. The colors were chosen and modified based on client wishes. Plain CSS was used to style the pages. Here are our designs that can be used as reference to recreate the layouts:

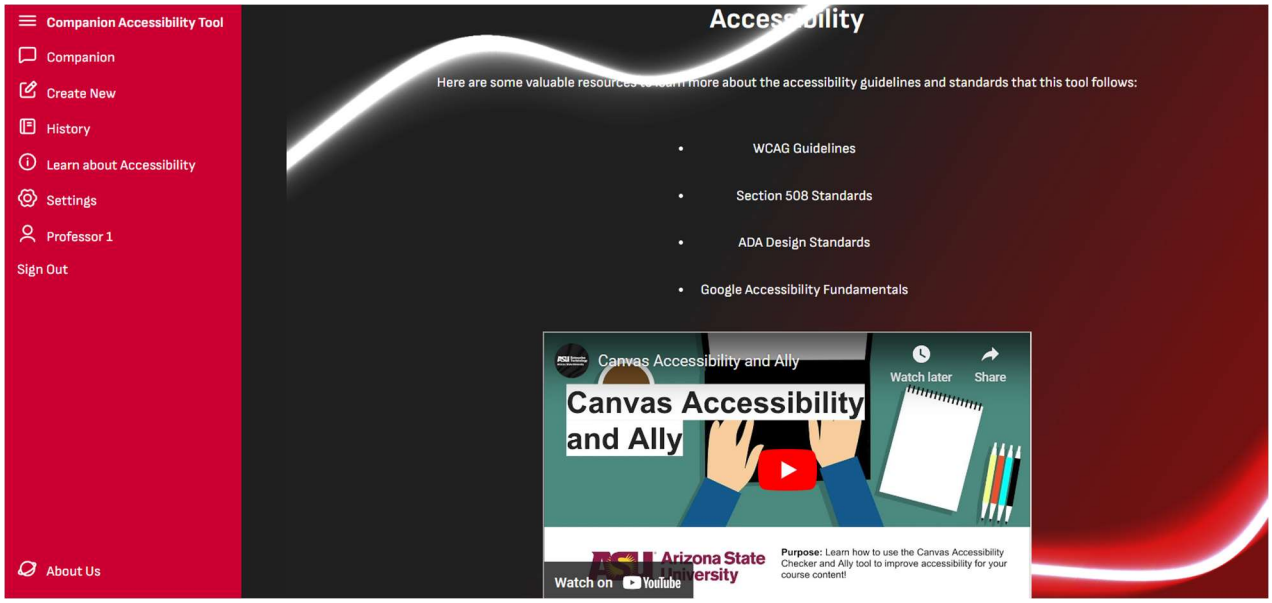
Home Page and Navbar:



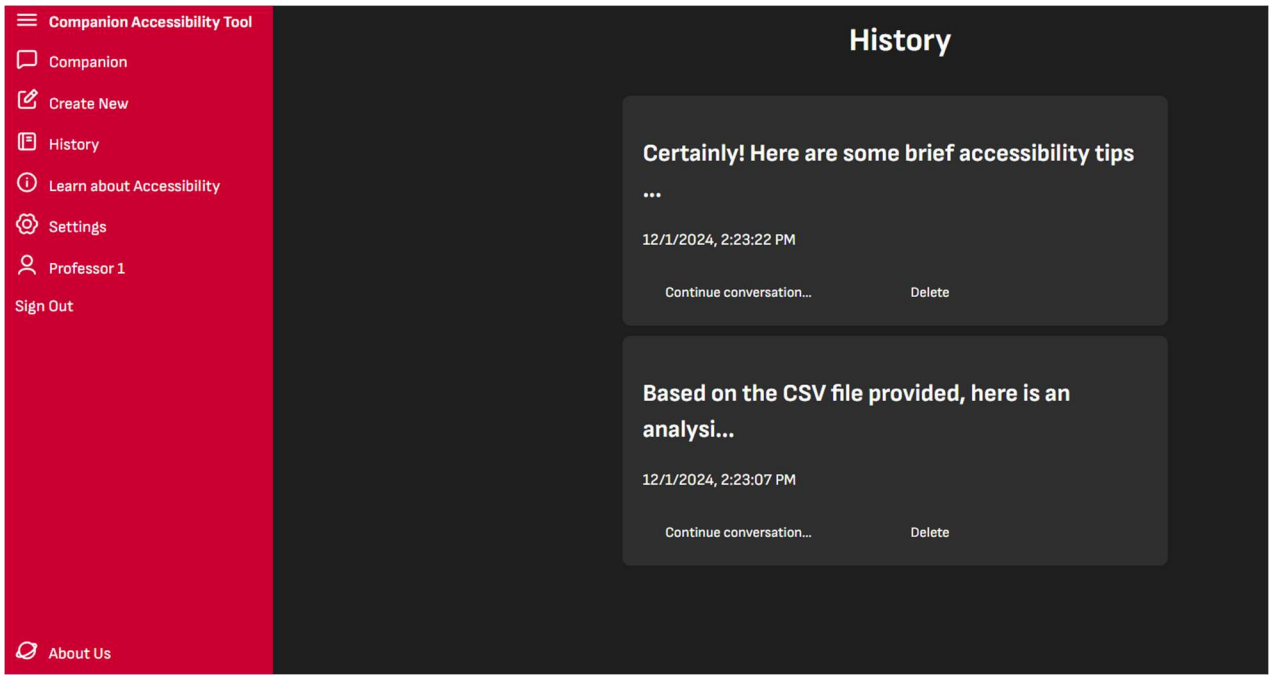
Home Page with Chat Started and file uploaded



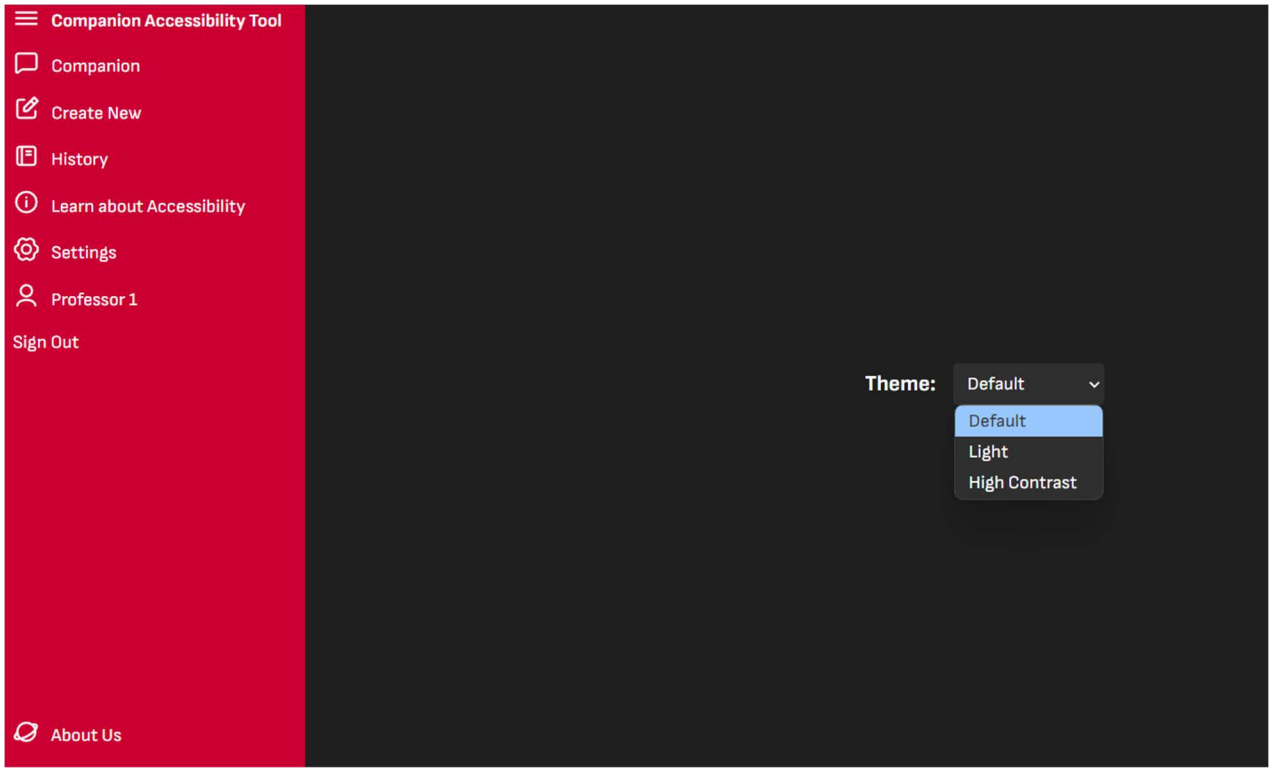
Accessibility Resource Page



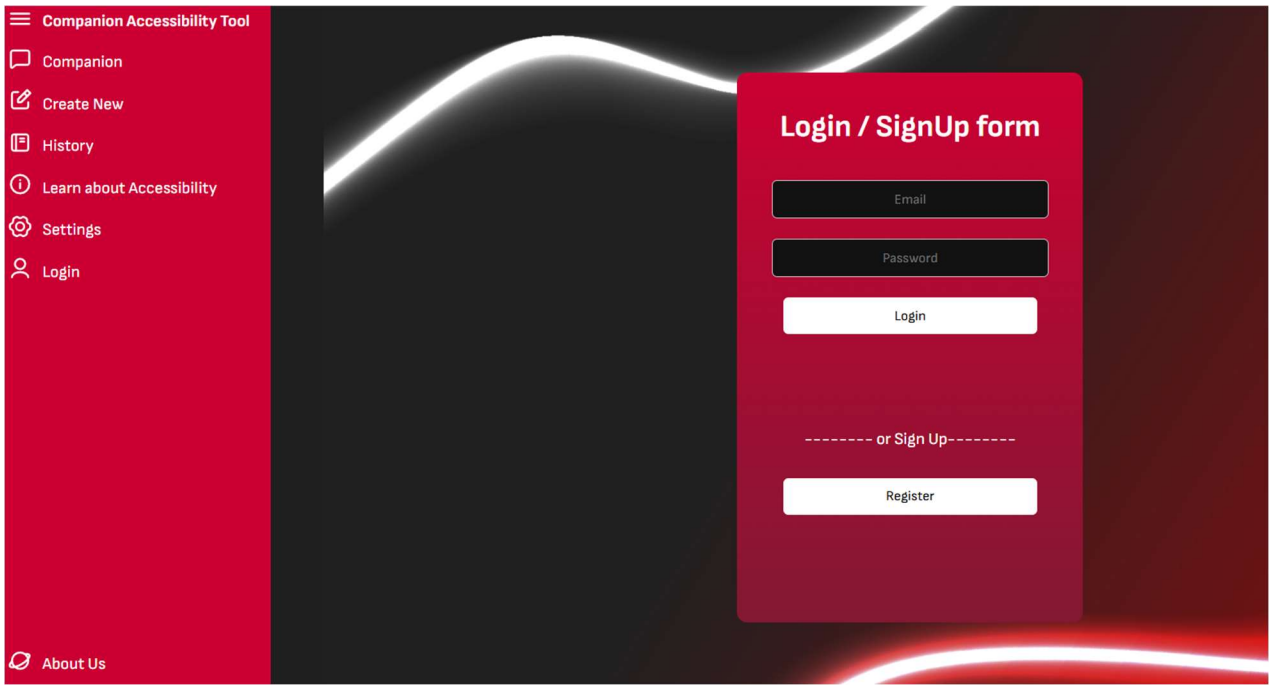
History Page

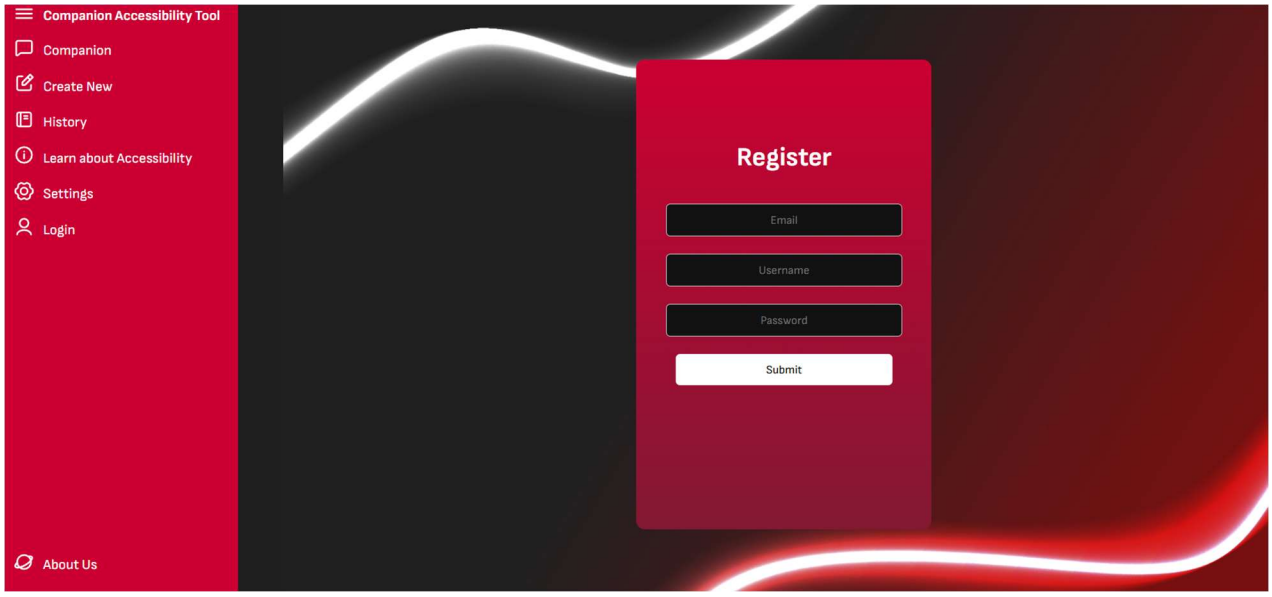


Theme Settings

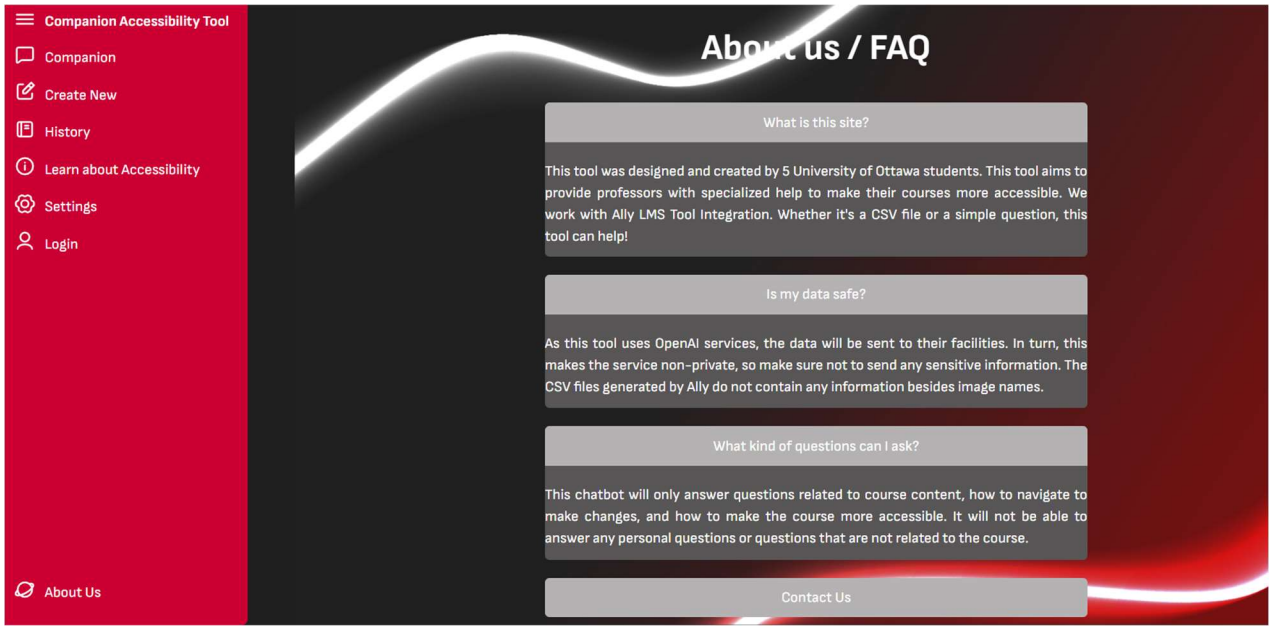


User Login and Signup

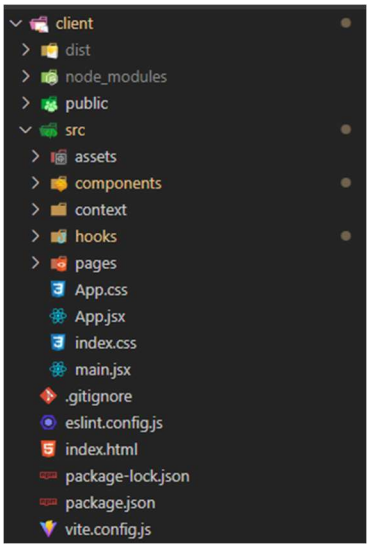




About Us Page



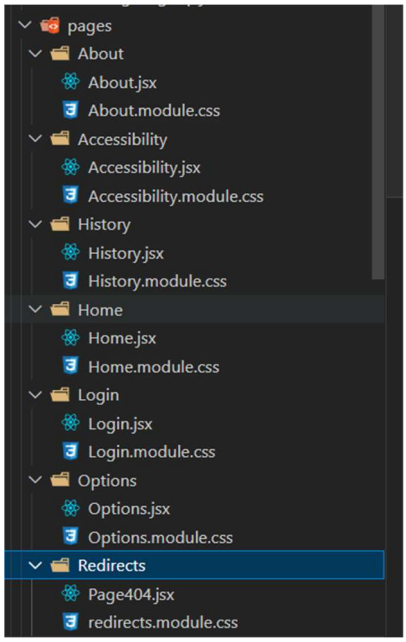
Client Folder Structure



This is the structure of our client folder in our project.

Pages

In our React application, each of our page components were structed in on pages folder. Each page had a subfolder containing its own .JSX file and .module.css file. This allowed us to isolate development for functionality and styling for each page. This also ensured any CSS would not overlap to a different page.



Navbar, Page Navigation, and Page Routing

The router from the React Router Library was used for page navigation. This allowed for ease of use by utilizing its custom “Link” tag in our Navbar component to redirect page navigation quickly. Here is our App.jsx page where all the routing is setup. Some page routes are protected by user authentication.

```
function App() {
  //protect routes using auth context
  const {user} = useAuthContext();


  return (
    <Router basename ="/Companion_GNG2101">
      <div className="topWrapper">
      <Navbar></Navbar>
      <div className="pages">
       <Routes>
         <Route path = "/" element = {<Home/>}></Route>
         <Route path = "/about" element = {<About/>}></Route>
         <Route path = "/history" element = {user? <History/>:<RequireLogin/>}></Route>
         <Route path = "/accessibility" element = {<Accessibility/>}></Route>
         <Route path = "/login" element = {<Login/>}></Route>
         <Route path = "/options" element = {<Options/>}></Route>
         <Route path="*" element={<Page404/>} />
       </Routes>
      </div>
     </div>
     </Router>

  )
}
```

Here is a section of our Navbar component that utilizes the Link property where if the user clicks on it, they will navigate to a different page. The Navbar also utilized state variables to control if it is expanded or not. The animation was done in CSS. ReactSVG icons from its named library were used as well to have accessible icons for viewing.

```
return (
  <div
    className={`${isExpanded ? styles.navbarExpanded : styles.navbar} ${
      !isExpanded ? styles.closed : ""
    }`}
  >
    <div className={styles.navHeader}>
      <button className={styles.hamBtn} onClick={toggleBar}>
        <ReactSVG src={hamburger} className={styles.icon} />
        {isExpanded && <span>Companion Accessibility Tool</span>}
      </button>
    </div>
    <nav className={styles.menu}>
      <ul>
        <li>
          <Link to="/" className={styles.menuLink}>
            <ReactSVG src={chat} />
            {isExpanded && <span>Companion</span>}
          </Link>
        </li>
        <li>
          <Link onClick={handleCreateNew} className={styles.menuLink}>
            <ReactSVG src={create} />
            {isExpanded && <span>Create New</span>}
          </Link>
        </li>
        <li>
          <Link to="/history" className={styles.menuLink}>
```

React Context, History and Authentications

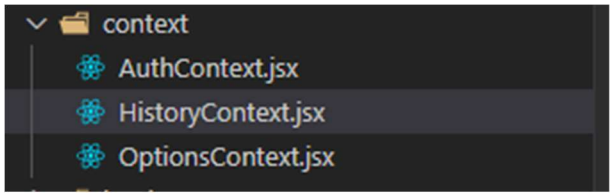
Our application has various global states that need to be managed:

- **History**: When a user wants to continue their previous conversation on the history tab, the application will need to temporarily store the conversation (stored in a list format, see Back-end for more information) and send it to the Home page that contains the chat interface and start it.
- **User Authentication:** We need to store the user object and send it to all components that need it for validation such as the history page (which needs the user id to fetch the histories of the user).
- **Theme:** We need to store which theme was selected to send it to other UI components to change color.

Thus, we utilized **React Context** to handle these states. Here is our main.jsx file to wrap our application with the contexts. The providers will send the information to anything inside of it (which is the App that contains the entire application).

```
createRoot(document.getElementById('root')).render(
  <StrictMode>
    <AuthContextProvider>
      <HistoryContextProvider>
        <OptionContextProvider>
          <App />
        </OptionContextProvider>
      </HistoryContextProvider>
    </AuthContextProvider>
  </StrictMode>,
)
```

We also have our different context files that handle the higher level of state:



Here is an example of the HistoryContext file:

```
import { createContext, useContext, useState } from "react";

const HistoryContext = createContext();

export const HistoryContextProvider= ({children})=>{
    const [messageHistory, setMessageHistory] = useState([]);

    const clearMessageHistory = () =>{
        setMessageHistory([]);
    }

    return (<HistoryContext.Provider value={{messageHistory, setMessageHistory, clearMessageHistory}}>
        {children}
    </HistoryContext.Provider>);
}

export const useHistoryContext = () =>{
    return useContext(HistoryContext);
}
```

Any child components can access messageHistory and the setMessageHistory and clearMessageHistory functions. An example is our chatHandler file (see hooks for explanation) where when constructed, will load the messageHistory into its messages list that can be viewed. Simply: user clicks continue conversation → Redirect to chat interface → initially load chat handler → set messages from history context → finally load chat ui so user can view the past messages in the chat ui.

```
const {messageHistory, clearMessageHistory} = useHistoryContext();//

//on mount, set history context into messages
useEffect(()=>{
    setMessages(messageHistory);
    clearMessageHistory();//reset history context
},[]);
```

Other contexts work in a similar principle where they contain some logic and state that are called by other components. See the source code for the specific implementation.

React Hooks, Fetching Data, Streaming Responses

In our frontend, we must call the server to retrieve or send information. Therefore, we used 3 different **React Hooks,** separate files to abstract logic, that can be used in any component to do specific server fetch and send operations:

- **useHistoryManager:** This hook is used to fetch conversations history of a user from the server (where the server fetches it from database, see models in the back-end to see how

these are structured). This hook also has functions that can be used to create or delete a conversation. Here is a sample of the file.

```
import { useAuthContext } from "../context/AuthContext";
import { useState, useEffect } from "react";

const useHistoryManager = () => {
    const {user} = useAuthContext();
    const [conversationHistory,setConversationHistory]= useState([]);

    // Fetch conversations when the component mounts
     useEffect(() => {
        if (user) {
            fetchConversations();
        }
    }, [user]);

    const fetchConversations = async()=>{
        try {
            const response = await fetch("/api/conversations/", {
              method: "GET",
              headers: {
                Authorization: `Bearer ${user.token}`, // Assuming JWT is used for auth
              },
            });

            if (!response.ok) {
              throw new Error("Failed to fetch conversations");
            }

            const data = await response.json();
            setConversationHistory(data);
          } catch (error) {
            console.error("Error fetching conversations:", error);
          }
    }
```

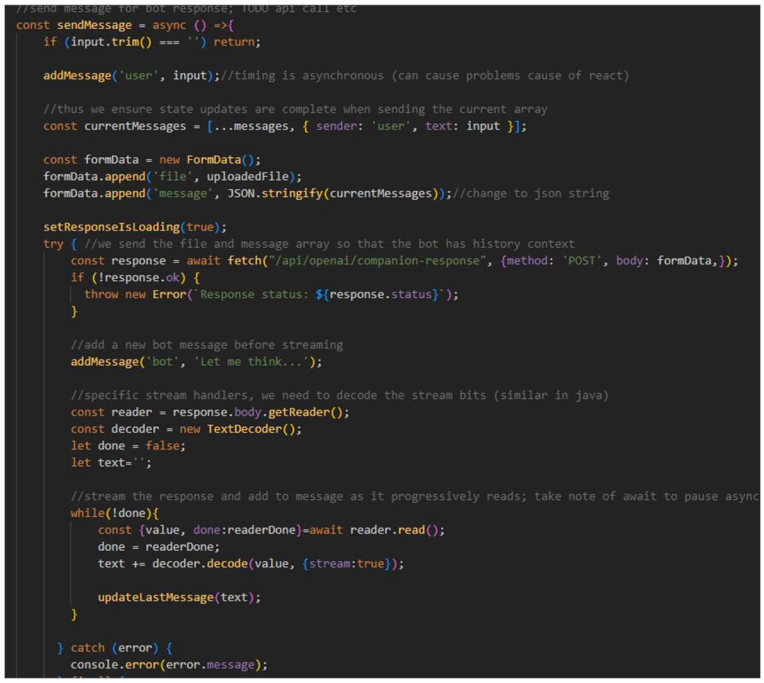
When this hook is constructed, it does an initial call, setting the conversationHistory of the specific user. This can be then used in the UI to show the information. Here is the History page that utilizes this hook:

```
const {conversationHistory,setConversationHistory,deleteConversation} = useHistoryManager();

const handleDeletion = async (id) =>{···
}

return (
    <div className={styles.pageContainer}>
      <header className={styles.header}>
          <h1>History</h1>
      </header>

      <div className={styles.conversationList}>
          {conversationHistory.length === 0 ? (
              <p>No conversations found.</p>
          ) : (
              [...conversationHistory]
              .sort((a, b) => new Date(b.timestamp) - new Date(a.timestamp)) // Sort by lat
              .map((conversation) => (
                <Conversation
                  key={conversation._id} // Assuming each conversation has a unique _id
                  id={conversation._id}
                  title={conversation.title}
                  messages={conversation.messages}
                  timestamp={conversation.timestamp}
                  handleDelete={() => handleDeletion(conversation._id)}
                />
              ))
```

It calls the history manager and then formats all the information in its markup.

- **useLoginSignup:** This hook contains functions that delegate HTTPs POST methods to sign or login the user into the server. It sends a request and then the server will respond with the user object that was created or found.
- **useChatHandler:** This hook is the most crucial in the program, it handles the states of chat messages sent by the server from our AI assistant Companion. This hook contains the

functions for uploading a file, sending messages to the AI, and receiving messages from the AI. This hook also will manage the stream of response data that OpenAI sends and by incrementally appending the new data into the messages list, the application will be able to show a generative animation of the response.



```
//send message for bot response, TODO api call etc
const sendMessage = async () =>{
    if (input.trim() === '') return;

    addMessage('user', input);//timing is asynchronous (can cause problems cause of react)

    //thus we ensure state updates are complete when sending the current array
    const currentMessages = [...messages, { sender: 'user', text: input }];

    const formData = new FormData();
    formData.append('file', uploadedFile);
    formData.append('message', JSON.stringify(currentMessages));//change to json string

    setResponseIsLoading(true);
    try { //we send the file and message array so that the bot has history context
        const response = await fetch("/api/openai/companion-response", {method: 'POST', body: formData,});
        if (!response.ok) {
          throw new Error(`Response status: ${response.status}`);
        }

        //add a new bot message before streaming
        addMessage('bot', 'Let me think...');

        //specific stream handlers, we need to decode the stream bits (similar in java)
        const reader = response.body.getReader();
        const decoder = new TextDecoder();
        let done = false;
        let text='';

        //stream the response and add to message as it progressively reads; take note of await to pause async
        while(!done){
            const {value, done:readerDone}=await reader.read();
            done = readerDone;
            text += decoder.decode(value, {stream:true});

            updateLastMessage(text);
        }

      } catch (error) {
        console.error(error.message);
```

This is snippet of the file, where the hook continuously updates the message array after calling for a AI response. This hook is used in the chat interface and when the message list is updated, the UI is updated (thanks to React's complex state updates). See source code for the specific implementation.

Chat Interface, Auto Response Formatter

The Home page jsx file also handles the chat interface. When the user starts typing, state is updated to switch the layout into the chat interface. This is the most complex page file since it calls multiple

things such as the chat handler hook, authentication, message history, and even manages UI updates.

```
const Home = ({}) => {

  const {user}=useAuthContext();
  const [beganConversation, setBeganConversation] = useState(false);
  const {
    messages,
    input,
    setInput,
    sendMessage,
    handleFileUpload,
    uploadedFile,
    responseIsLoading
  } = useChatHandler();

  const {messageHistory} = useHistoryContext();
  const{createConversation} = useHistoryManager();

  const messagesEndRef = useRef(null); //reference used to snap to for scro
  const fileInputRef = useRef(null); //reference for file input
  const [saved, setSaved]=useState(false);
```

Here we can see all the functions and properties the Home file is using from other areas in the program. In addition, messages in the message list are shown in the UI, and they are also parsed into an auto formatter. This was added so that the raw text responses from the AI can be more legible.

```
  {beganConversation && <div className={styles.messages}>
    {messages.map((message) =>(
      <div key = {message.id} className={`${styles.message} ${message.sender ==='user' ? styles.user: styles.bot}`}>
        {parseMessageToJSX(message.text)}
      </div>
    ))}
    <div ref={messagesEndRef}></div>
    </div>}
```

Our parseMessageToJSX can be seen here: 

```
function parseMessageToJSX(messageText) {
  const lines = messageText.split('\n');

  return lines.map((line, index) => {
    // Check if line has more than one space (add padding for extra whitespace
    if (line.trim().length === 0) {
      return <div key={index} className={styles.line_space}/>;
    }

    // Render bold text (e.g., **bold**)
    if (line.startsWith('**') && line.endsWith('**')) {
      return <strong key={index}>{line.slice(2, -2)}</strong>;
    }

    // Render italic text (e.g., *italic*)
    if (line.startsWith('*') && line.endsWith('*')) {
      return <em key={index}>{line.slice(1, -1)}</em>;
    }

    // Render bullet points (e.g., - bullet point)
    if (line.startsWith('- ')) {
      return <ul key={index}><li>     {line.slice(2)}</li></ul>;
    }

    // Replace bold (**bold**) and italic (*italic*) in the middle of text
    const lineElements = [];
    let parts = line.split(/(\*\*[^*]+\*\*|\*[^*]+\*)/g); // Split by bold/ita
    parts.forEach((part, idx) => {
      if (part.startsWith('**') && part.endsWith('**')) {
        lineElements.push(<strong key={idx}>{part.slice(2, -2)}</strong>);
      } else if (part.startsWith('*') && part.endsWith('*')) {
        lineElements.push(<em key={idx}>{part.slice(1, -1)}</em>);
      } else {
        lineElements.push(part);
```

It contains a lot of complex regex and formatting. Overall, all these components allow the application to have one clean and effective chat interface for use.

6.2 Back-End (Server Side)

This section breaks down the back-end implementation. Note: MongoDB and OpenAI services will be discussed, but setting up those services on other websites will not be discussed. That should be researched by the users themselves since this guide has a level of assumption. All specific code and implementation can be found at https://github.com/andriusavenido/Companion_GNG2101/.

6.2.1 BOM (Bill of Materials)

Item Name	Units	Qty	Unit cost	Extended Estimated cost
GPT-4o API Key	API Tokens	1	$2.50 / 1M input tokens $1.25 / 1M cached** input tokens $10.00 / 1M output tokens	For one month of development testing, and assuming 50 requests a day (15 000 input and output tokens) is roughly $0.1875 per day or $5.65 for one development month
MongoDB Atlas	n/a	1	0$ for small scale projects	0$
Developer Tools	n/a	1	0$	0$
Total product cost (without taxes or shipping)				**$5.65 USD**
Total product cost (including taxes and shipping)				**$6.48 + $0 Shipping USD**

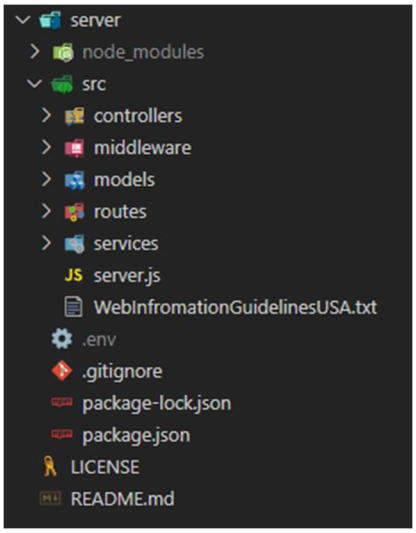
Note: This BOM is shared with the Front-End subsystem

6.2.2 Equipment list

- **Node.js**
- **Mongoose**
- **MongoDB**
- **OpenAI API**
- **Visual Studio Code**
- **Postman (for testing)**
- **Express.js**
- **NPM, Nodemon**

6.2.3 Instructions

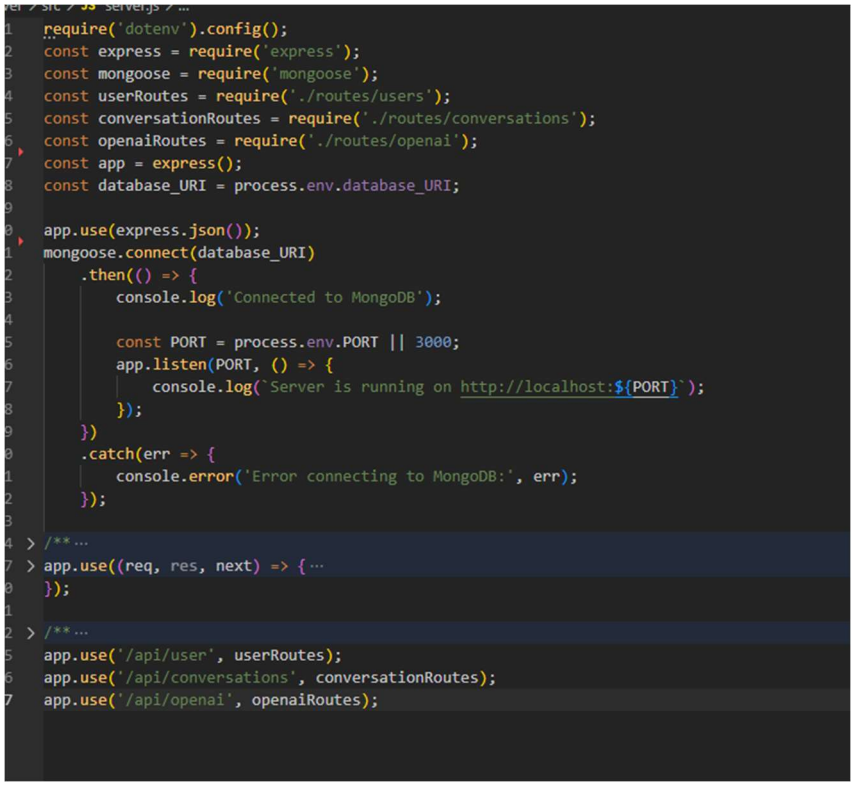
Our server is much simpler than the Front-End, however it is still quite important and complex.



We organized our server folder into different sections:

Express and Routing

Our server is build using the Express framework which allows us to create a web server with ease and have it handled all the specific implementations. The server is run from one file called server.js that utilizes different things from different areas of the project.



```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const userRoutes = require('./routes/users');
const conversationRoutes = require('./routes/conversations');
const openaiRoutes = require('./routes/openai');
const app = express();
const database_URI = process.env.database_URI;

app.use(express.json());
mongoose.connect(database_URI)
    .then(() => {
        console.log('Connected to MongoDB');

        const PORT = process.env.PORT || 3000;
        app.listen(PORT, () => {
            console.log(`Server is running on http://localhost:${PORT}`);
        });
    })
    .catch(err => {
        console.error('Error connecting to MongoDB:', err);
    });

/**...
app.use((req, res, next) => {...
});

/**...
app.use('/api/user', userRoutes);
app.use('/api/conversations', conversationRoutes);
app.use('/api/openai', openaiRoutes);
```

- dotenv, is used to handle our environment file (that contains keys and passwords)
- express/app runs the server.
- We import different routes that are used to handle the different HTTPs calls that can be made to this server from the client. One used for users, conversations, and OpenAI calls.
- Mongoose is a library used to connect to the MongoDB database

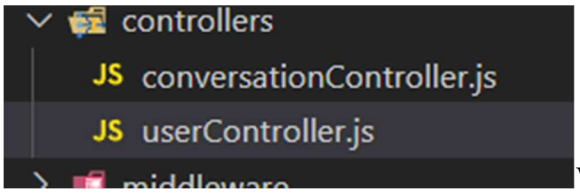
MongoDB, Mongoose, and Models

As seen above, we utilize MongoDB and Mongoose to handle database connections. We store both conversations and user accounts in the database and define the information of each in the following models:

```
const conversationSchema = new Schema ({
    user_id:{
        type: Schema.Types.ObjectId,
        ref: 'User',
        required:'true',
    },
    title:{
        type: String,
        required: true,
    },
    messages:[ {
        id: {
            type: Number,
            required: true,
        },
        sender: {
            type: String,
            required: true,
        },
        text: {
            type: String,
            required: true,
        },
    },],
    timestamp: {
        type: Date,
        default: Date.now,
    }
},{collection:'conversations'});
```

```
const Schema = mongoose.Schema;

const userSchema = new Schema({
    email: {
        type: String,
        required: true,
        unique: true
    },
    username:{
        type: String,
        required: true,
    },
    password:{
        type: String,
        required: true,
    }
},{collection: 'users'});
```

HTTPS Controllers



We use two different controller files that contain all the logic used for the conversations and user HTTPs routes. These controllers contain logic for talking to the database (creating, deleting, etc.). See source code for specific implementation.

Authentication

Our server also handles specific authentication for user login and signups. We utilize JWT to handle tokens for authentication with the front end to ensure our data is encrypted. We also use encryption libraries to hash passwords securely:

```
// add salt and hash password for encryption to database
const salt = await bcrypt.genSalt(12);
const hash = await bcrypt.hash(password, salt);

//add to db
const user = await this.create({email, username, password: hash});
```

See source code for specific implementation. See PD E (in design files) for more details as well.

CSV File Reader

We have a file that aggregates the data read from a CSV file. The files are passed through our OpenAI route as such:

```
const express = require('express');
const multer = require('multer');
const router = express.Router();

// Grab services
const { getOpenAIResponse } = require('../services/openaiService');
const { csvToJson } = require('../services/csvService');

// Set up multer to store files in memory
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });


router.post('/companion-response', upload.single('file'), async (req, res) => {
    const file = req.file;

    try {
        let csvJSON = null;
        if (file){
            csvJSON = await csvToJson(file.buffer);
        }
```

where we use the library, multer, to handle the storage of the file. This file is then sent to the file reader to be aggregated.

```
function csvToJson(buffer) {
    return new Promise((resolve, reject) => {
        const results = [];
        const parser = parse(buffer, {
            columns: true, // Use the first line as column names
            skip_empty_lines: true
        });

        parser.on('data', (row) => {
            // Filter error columns with value '1'
            const errors = Object.keys(row).filter(key => row[key] === '1' && key !== 'Id' && k
            if (errors.length > 0) {
                results.push({
                    Name: row['Name'],
                    FileType: row['Mime type'],
                    Id: row['Id'],
                    Url: row['Url'],
                    Errors: errors
                });
            }
        });

        parser.on('end', () => {
            resolve(results); // Return the parsed JSON
        });

        parser.on('error', (error) => {
            reject(error); // Handle any parsing errors
        });

        parser.write(buffer);
        parser.end();
    });
}
```

This CSV reader is specifically tailored towards the CSV files outputted/generated by Ally.

OpenAI Service

The OpenAI Service file is the most crucial file in the server. It is accessed through a distinct HTTPs route and handles all logic pertaining to communicating with the OpenAI API.

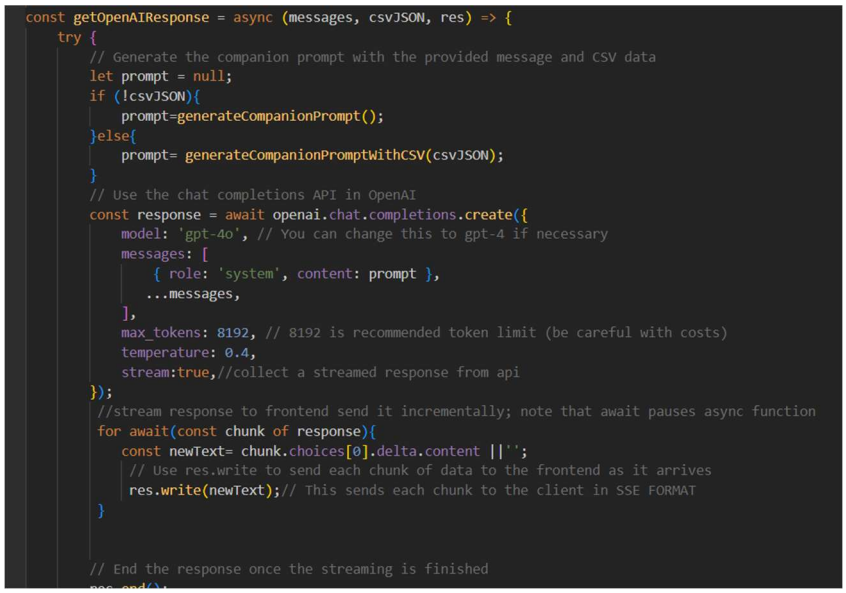
```
router.post('/companion-response', upload.single('file'), async (req, res) => {
    const file = req.file;

    try {
        let csvJSON = null;
        if (file){
            csvJSON = await csvToJson(file.buffer);
        }

        const parsedMessages = JSON.parse(req.body.message);//unstring the json messages
        const messages = convertMessagesToGPTRoles(parsedMessages);  //convert to gpt ac

        // Send the parsed CSV JSON and message to OpenAI service and stream the respons
        res.setHeader('Content-Type', 'application/json');
        await getOpenAIResponse(messages, csvJSON,res); //pass res object to directly st

        // // Send the OpenAI response back to the client
        // res.json({ openaiResponse });

    } catch (error) {
        // Handle any errors that occur during the process
        console.error(error);
        res.status(500).json({ error: error.message });
    }
});
```

This is the route for the AI service that the front end will call. In the service file we have two functions with our custom engineered

prompt that contains the Companion AI personality and rules. This is the prompt that is sent initially to the OpenAI API at every call.

```
/**
 * Create custom prompt with all of our parameters
 * @param {JSON} jsonData to be attatched into parameter
 *
 * TODO: we want to add more things when creating prompt, like rules and regulations etc..
 */

//TODO: prompt is not finished -> more testing required
//TODO: ADDING CHAT HISTORY from front end
const generateCcompanionPromptWithCSV = (csvData) => {···
};


const generateCcompanionPrompt = () =>{···
}
```

Finally, we have the main function in the service that calls the API. We chose to get the responses in streamed chunks so that the responses would come faster and allow us to have a generative effect in the frontend.



```
const getOpenAIResponse = async (messages, csvJSON, res) => {
    try {
        // Generate the companion prompt with the provided message and CSV data
        let prompt = null;
        if (!csvJSON){
            prompt=generateCompanionPrompt();
        }else{
            prompt= generateCompanionPromptWithCSV(csvJSON);
        }
        // Use the chat completions API in OpenAI
        const response = await openai.chat.completions.create({
            model: 'gpt-4o', // You can change this to gpt-4 if necessary
            messages: [
                { role: 'system', content: prompt },
               ...messages,
            ],
            max_tokens: 8192, // 8192 is recommended token limit (be careful with costs)
            temperature: 0.4,
            stream:true,//collect a streamed response from api
        });
         //stream response to frontend send it incrementally; note that await pauses async function
         for await(const chunk of response){
            const newText= chunk.choices[0].delta.content ||'';
             // Use res.write to send each chunk of data to the frontend as it arrives
             res.write(newText);// This sends each chunk to the client in SSE FORMAT
         }


        // End the response once the streaming is finished
```

Summary of the Cycle: Front end calls with message → go to openai route → go to openai service → receive OpenAI streamed response → stream response continuously to front-end.

6.3 Testing & Validation

Testing was done using various tools such as Vite, web browsers, Postman, and external users to test various metrics. Note: ChatGPT was used to format our table below:

Test Case	Metric	Expected Value	Test Description	Actual Result	Pass/Fail
1. User Authentication	Authentication Success Rate	100% successful logins with valid credentials	Test that a user can successfully log in with correct username and password and receive a valid JWT token.	100% successful logins with valid credentials	Pass
2. Chat Response Time	Average Response Time (Latency)	< 2 seconds	Measure how long it takes for the chatbot to respond after sending a message.	Average response time: 1.5 seconds	Pass
3. CSV File Upload	File Upload Success Rate	100% of valid CSV files are processed successfully	Test that a valid CSV file uploads correctly and is parsed into structured data without errors.	100% of valid CSV files processed successfully	Pass

4. Theme Switching (Light/Dark)	Theme Toggle Success Rate	100% of users can toggle between light and dark modes	Test that users can switch between light and dark themes, and the change persists after page reload.	100% success rate for theme toggle, persists after reload	Pass
5. Accessibility Features (Screen Reader)	Accessibility Compliance (%)	100% of chat content is accessible via screen readers	Test that all chat content, buttons, and navigation elements are properly read by screen readers.	100% compliance with screen readers	Pass
6. Error Handling in Chatbot	Error Handling Success Rate	100% of errors are caught and appropriate messages displayed	Test that any failed requests to the OpenAI API return a meaningful error message without crashing the system.	All errors returned appropriate error messages	Pass

7 Conclusions and Recommendations for Future Work

Lessons Learned

Throughout the development process, we identified several areas for improvement that will guide future iterations. One key lesson was the importance of creating a responsive user interface that adapts seamlessly to different screen sizes. Challenges like unnecessary scrollbars or overlapping elements underscored the need for thorough testing across multiple devices to ensure a consistent user experience. Another lesson was the need for better history management. Specifically, when continuing a chat from a saved history, saving it again would create a new entry instead of updating the existing one, leading to unnecessary duplication and confusion for users. Finally, handling large inputs, such as error-filled CSV files, highlighted the need to preprocess and streamline data before sending it to the ChatGPT API to improve efficiency and reliability.

Future Work Recommendations

Given more time, we would focus on refining key areas of the prototype. Accessibility features could be expanded to include a hotbar for quick toggles like high contrast, font size adjustments, and text-to-speech options. Existing features, such as the contrast toggle, could also be made more intuitive and user-friendly. The history feature would be reworked so that continuing a chat from history updates the original session rather than creating a new saved entry. This improvement would simplify session management and enhance the user experience. Additionally, optimizing how large files are handled by preprocessing data and splitting oversized requests would ensure smoother interactions with the ChatGPT API.

Suggestions for Future Teams

Future teams can build upon these lessons by prioritizing session management, accessibility, and data handling. Enhancing session management would allow seamless transitions between saved and active chats, reducing redundancy and improving workflow. Accessibility features could be tailored to user needs, ensuring the application serves a broader audience. Finally, implementing intelligent preprocessing and error-checking systems for data inputs would improve ChatGPT's response accuracy and efficiency. Regular user testing and feedback would further help refine these features and guide the application's evolution, ensuring a robust and user-friendly tool.

8 Bibliography

APPENDICES

9 APPENDIX I: Design Files

Below we have various design files located in the MakerRepo link: https://makerepo.com/andriusavenido/2111.companion-lms-accessibility-tool-by-companion-dev-team-. The files pertain to older prototypes.

Table 3. Referenced Documents

Document Name	Document Location and/or URL	Issuance Date
PD E.docx	https://makerepo.com/andriusavenido/2111.companion-lms-accessibility-tool-by-companion-dev-team-	2024-12-01
PD_B,C , and D(1).docx	https://makerepo.com/andriusavenido/2111.companion-lms-accessibility-tool-by-companion-dev-team-	2024-12-01

10 APPENDIX II: Other Appendices