SDLC Life Cycle – Screen Reader Project

**Stage 1 –** Planning & Requirement Analysis:

The Screen Reader prototype is an internal application that will be designed to mimic real world screen readers such as JAWS or NVDA and will be used for accessibility testing or user education. It will allow users to upload HTML files and will then simulate how a screen reader would interpret and announce the content. Parsing HTML content and DOM to extract accessible content: “aria-label, alt” etc.

Will provide unique features such as:

* Play/Pause Button
* Stop/End Button
* Speed Adjustment
* Keyboard navigation and shortcuts (H = Next Heading, TAB = Skip Lines, J = Skip Abbreviations)
* Upload multiple files/folder (See more below)
* See how screen reader interprets.

Requirements:

* No API calls or access to internet (100% local)
* Accessibility: WCAG2.1/2.2, ARIA, and AODA standards.
* Clean UI easy to understand even for a first grader.
* Process reasonably large HTML documents without lag.

Constraints:

* No Internet APIs or server communication (runs completely offline and no data)
* Simulate realistic screen reader (not just read text)
* Must support assistive features like focus, keyboard navigation and ARIA parsing.

Technologies Used:

* HTML/CSS/JavaScript
* TypeScript (Strong-typed logic for DOM parsing, TTS and keyboard manipulation.
* Tailwind CSS (Fast, responsive accessible UI)
* Windows built in text to speech (windows voices such as David, Zira, to read the text)
* Docker (Optional to secure deployment)

Use Cases:

* Accessibility testing in private environments.
* Developers previewing how content is read by a screen reader.
* Users learning how screen readers will interpret their HTML + ARIA content.

Key feature: Upload Folder/Multiple HTML files.

Instead of just uploading one HTML file repeatedly, allow users to upload an entire folder of html files and simulate screen reader output per page.

* Turn multiple html files/web pages into a “booklet” with page list navigation such as “previous page, or next page”.
* Automatically parse each html inside that folder.
* Preload everything for faster user experience.
* Dropdown to jump to any uploaded HTML file.

Key feature: Reader panel to show how the screen reader interprets.

* Show the user a visual panel on what the screen reader is picking up.
* A currently reading line which will show which line the screen reader is on.
* Pause button will show where text is paused.

Example would be:

|  |
| --- |
| Reading page1.html… |
| Heading level 1: Welcome to the website |
| Link: Learn more |
| Form field: First Name, required. |
| Button: Submit |

(Highlighted colour will indicate where the screen reader TTS is).

Other optional features: other than making the user experience for developers very easy to use such as multiple folder/file uploading:

* Allow the users to upload their own voice and export/download the .txt file that the screen reader will generate to visually see the text being displayed and which they can run through system-level TTS tools (windows narrator, macOS voiceover etc.) in which ever case they need to use their own voice.

Completely optional and does not need a uploading your own voice option.

To conclude, the application must remain completely internal and local, should be a tool to test accessibility and simulate screen readers.

Objectives & Goals (subject to change):

Simulate Screen Reader

* Emulate how assistive tech like JAWS, NVDA, and Voiceover interpret web content.
* ARIA roles, landmarks, labels, and semantic HTML.

File & Folder Upload Support

* Allow users to upload individual HTML files or entire HTML folders (to make it easier and not upload a single file at a time and the entire folder)
* Automatically detect and load multiple pages with navigation (next/prev etc.)

Offline Text-to-Speech Output

* Use browser-native speechSynthesis API (offline API for audio output and runs completely locally, Microsoft uses it internally)
* Include visual text display of what is being read (for feedback or clarity)

User Playback Controls

* Play, Pause, and Stop
* Adjustable speed (slow, normal, fast)
* Skip to next element or landmark

Keyboard Navigation Simulation

* Support realistic navigation shortcuts.
* (H – Next Heading), (F – Form Fields), (L – Links), (R – for Regions/Landmarks)

No API or Internet Dependency

* Compliant with strict privacy use cases
* No external requests, databases, or cloud services
* As mentioned before speechSynthesis runs completely locally

Simple Accessible UI

* Built with Tailwind CSS and accessible markup
* Support keyboard-only interaction
* Optionally show currently “focused” element in visual reader panel

Future Docker Compatibility

* Allow containerization at the end of the project for internal deployment (so no: downloading, no “this only runs on your machine and not mine) Docker will containerize the project so it will run the same in everyone’s computer.

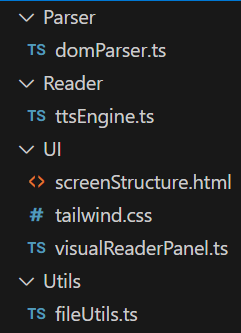
**Stage 2 –** System Design:

Architecture Design:

App Type: (HTML/CSS, JavaScript, TypeScript) local only

Folder Structure:

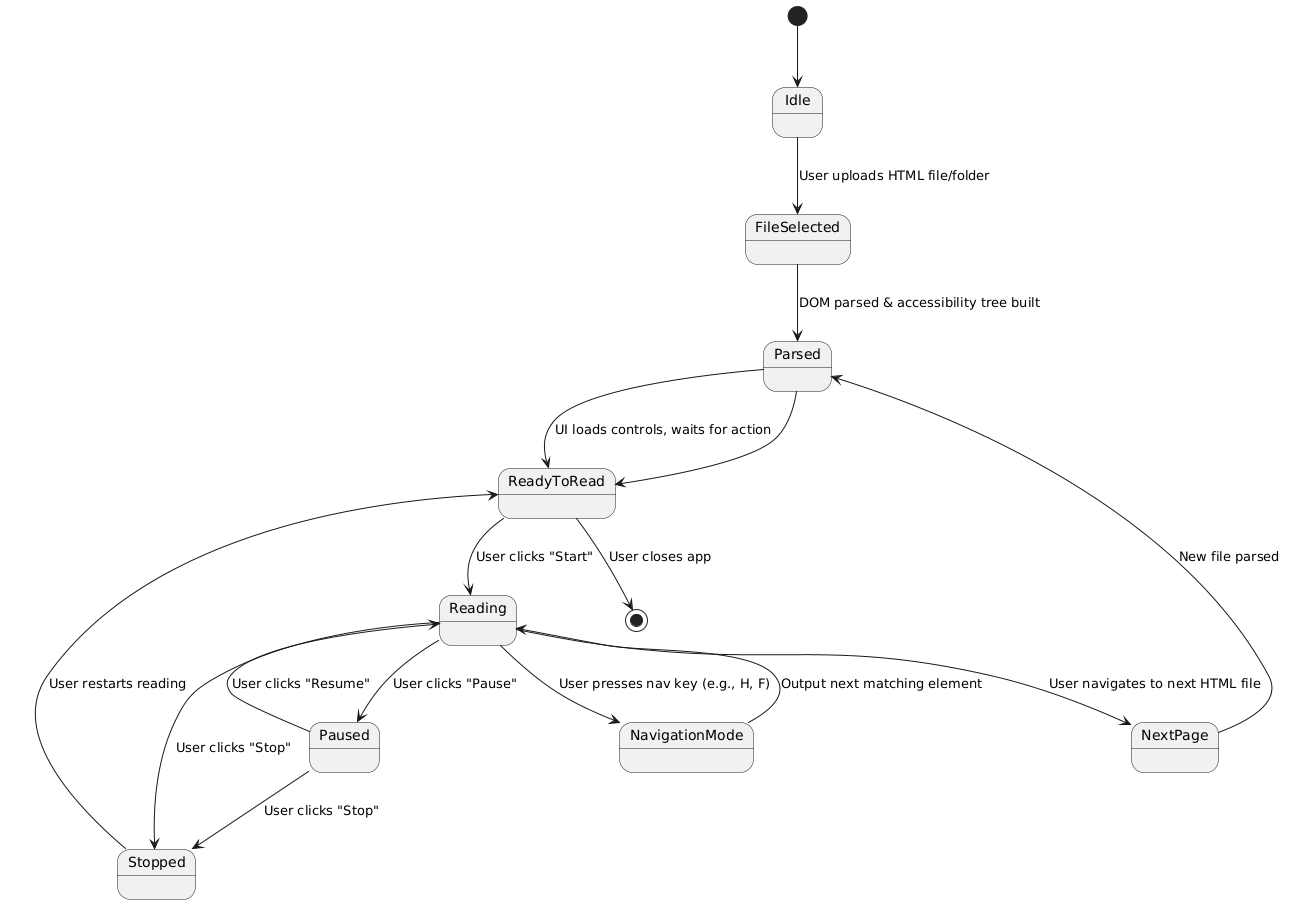
* Parser/ - DOM and ARIA parsing
* Reader/ - TTS logic, focus tracking
* UI/ - Tailwind components, visual panels (html/css)
* Utils/ - Helper functions (file handling, TTS configurations)



(example structure, subject to change)

domParser.ts will ensure that DOM will dynamically access and manipulate the document's content, structure, and style. This will ensure that it will access the websites elements, ARIA parsing will also make sure that it can extract contents that have accessibility, such as reading abbreviations and skipping over unnecessary words, and will read to the user the ARIA-friendly words like a screen reader would. The next file ttsEngine.ts will handle all the TTS logic and will focus tracking, so the actual reading of the words and highlighting the colour of the selected word, it will handle logic functions, edge cases, and handle errors as well. The Html file screenStructure will hold the actual structure of the web page with all the elements, file upload buttons etc. and tailwind.css will be used to make a dynamic user-friendly web page for easy navigation and to make the design look more professional and not empty. For the actual reader panel there can be another file called visualReaderPanel.ts which will directly handle the reading panel that extracts the content of the html files on the screen, and will handle which word is highlighted etc. The last file fileUtils.ts is a helper file, and will have all helper functions for additional support, like handling of many different files or folders being extracted all at once, or even additional features beneficial for users. It can also be used to change the settings of the colour on their screen.

State Machine Diagram:



Explanation:

Black Dot (Start): Initial entry point when the app loads. Nothing has been uploaded yet.

Idle: The app is idle, waiting for the user to upload an HTML file or folder.

FileSelected: The user has selected one or more HTML files. The app is preparing to parse the content.

Parsed: The app has successfully parsed the HTML and built a simulated accessibility tree (based on ARIA roles, headings, forms, etc.).

ReadyToRead: The UI is ready for interaction, user can now start playback, navigate, or control output.

Reading: The simulator is actively "reading" the content (either via on-screen output or text-to-speech).

Paused: Reading is temporarily paused; user can resume without restarting.

Stopped: Reading has been fully stopped. The user can restart or load new content.

NavigationMode: The user is using keyboard shortcuts (e.g. H for next heading) to navigate accessible elements.

NextPage: If multiple HTML files were uploaded via folder upload, this state handles moving to the next file.

(End): User exits the application or closes the browser.

Transitions:

From Idle → FileSelected: User uploads one or more HTML files.

From FileSelected → Parsed: App parses DOM and ARIA attributes.

From Parsed → ReadyToRead: Controls become active for reading.

From ReadyToRead → Reading: User starts playback.

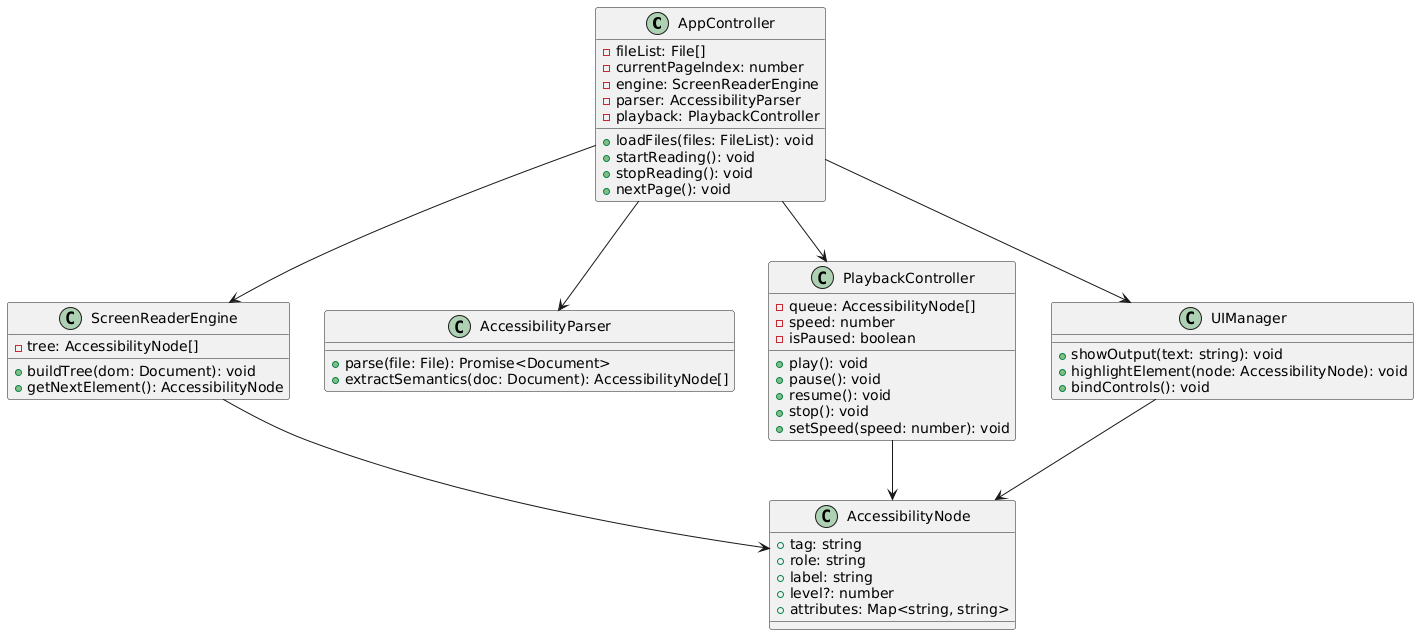
From Reading → Paused/Stopped: User controls playback.

From Reading → NavigationMode: User uses keyboard to jump through structure.

From Reading → NextPage → Parsed: User moves to another HTML file from folder.

From Any State → [\*]: User closes or exits app.

UML Class Diagram:



AppController – Main Orchestrator

Coordinates everything: file upload, parsing, reading, and UI updates.

Properties:

fileList: File[] – List of uploaded HTML files

currentPageIndex: number – Tracks the currently selected file/page

engine: ScreenReaderEngine – Accessibility tree handler

parser: AccessibilityParser – HTML parsing module

playback: PlaybackController – Controls play/pause/resume

UIManager – Handles UI updates and controls

Methods:

loadFiles(files: FileList): void – Accepts user-uploaded files or folders

startReading(): void – Begins reading current page

stopReading(): void – Stops reading

nextPage(): void – Loads next file in the list and resets state

ScreenReaderEngine – Accessibility Tree Manager

Builds and provides the reading order from parsed DOM content.

Properties:

tree: AccessibilityNode[] – Ordered list of accessible elements

Methods:

buildTree(dom: Document): void – Generates the tree from DOM

getNextElement(): AccessibilityNode – Retrieves the next readable node

AccessibilityParser – HTML File Processor

Processes raw HTML files into parsed documents and extracts semantic structure.

Methods:

parse(file: File): Promise<Document> – Parses uploaded file using FileReader

extractSemantics(doc: Document): AccessibilityNode[] – Converts DOM to accessible nodes (based on ARIA and semantics)

PlaybackController – Reading Logic & Controls

Manages playback speed, pause state, and reading queue.

Properties:

queue: AccessibilityNode[] – What will be read aloud (or shown)

speed: number – Current playback speed

isPaused: boolean – Indicates if reading is paused

Methods:

play(): void – Start reading from beginning

pause(): void – Temporarily stop reading

resume(): void – Resume reading from paused state

stop(): void – Completely stop reading

setSpeed(speed: number): void – Adjust playback speed

UIManager – Visual & Interaction Layer

Handles what the user sees and interacts with on the page.

Methods:

showOutput(text: string): void – Displays current reading text in output panel

highlightElement(node: AccessibilityNode): void – Visually highlights the active DOM node

bindControls(): void – Sets up event listeners for buttons and keyboard shortcuts

AccessibilityNode – Simulated Screen Reader Element

Represents one accessible item to be "read" (heading, button, link).

Properties:

tag: string – HTML tag (h1, a, button)

role: string – ARIA role (heading, link)

label: string – Text to be read (from aria-label, alt, or content)

level?: number – Optional heading level (h1 = 1, etc.)

attributes: Map<string, string> – Other relevant attributes

Class Relationships (Dependencies):

AppController → ScreenReaderEngine

AppController → AccessibilityParser

AppController → PlaybackController

AppController → UIManager

ScreenReaderEngine → AccessibilityNode

PlaybackController → AccessibilityNode

UIManager → AccessibilityNode

These relationships show how the controller delegates tasks to the other modules and how each component interacts with AccessibilityNode objects to simulate reading behavior.

Summary:

This architecture cleanly separates responsibilities:

AppController: Orchestration

Parser & Engine: HTML to logical structure

PlaybackController: Reading logic

UIManager: User interaction and display

AccessibilityNode: Core data mode

Example Image:

A screenshot of a computer

AI-generated content may be incorrect.

(Subject to change)

In the example image, the user can use the upload button to either upload the html file or folder of html files at once, then after the play button their html file contents will be parsed into the screen reader below there a TTS will read the words out loud like how a normal screen reader would ignoring Aria labels and abbr tags and reciting everything like. The user will be able to highlight/hover over the abbreviations and play them separately like in the image where NRST is highlighted in blue, the red text shows the reader where they are in the html file TTS, the skip forward button will allow the TTS to be recited 5x faster and other key inputs will be used like H for next heading etc. or another button like tab for the next abbreviation (it will say NRST then the next abbreviation after for an example if its LTT for land transfer tax), the user will also be able to pause and resume the screen reader as it recites all the words. The functionality isn’t meant for the user to bore themselves and listen to the entire html files, they will be able to click and skip to wherever they want and ensure the work they are putting out is 100% accessible.