HermeSpeech

Engineering Design Doc

github: @forrestpark

Goal of This Document

The goal of this design document is to describe the architecture, design, and implementation of HermeSpeech Recorder, a user-friendly and open-source web platform designed to record speech remotely in a HIPAA-compliant and efficient manner. HermeSpeech is particularly useful for collecting speech data that is difficult to crawl from publicly available resources, such as pathological and atypical speech.

This document will provide a detailed overview of the following:

- The overall system architecture of HermeSpeech Recorder, including the different components and how they interact with each other.
- The design of each component, including its purpose, functionality, and interfaces.
- The implementation details of each component, including the programming languages and frameworks used.

This document is intended for software engineers who will be responsible for developing, maintaining, and extending HermeSpeech Recorder. It may also be useful for other stakeholders, such as product managers and system administrators, who need to understand how the platform works.

Background and Motivation

Speech recording platforms are essential tools for collecting voice samples that can be used in speech and language technologies, including speech and speaker recognition, spoken language understanding (SLU), or text-to-speech. These technologies are used in a wide range of applications, such as voice assistants, smart speakers, and customer service systems.

However, existing speech recording platforms have several limitations. First, many of them require in-person recording, which can be inconvenient and time-consuming for participants. Second, some platforms are not HIPAA-compliant, which makes them unsuitable for collecting sensitive data. Third, few platforms are designed to collect atypical and pathological speech, which is essential for developing inclusive speech technologies.

The motivation for developing HermeSpeech Recorder is to address the limitations of existing speech recording platforms and to make it easier to collect high-quality speech data, especially for atypical and pathological speech applications.

HermeSpeech Recorder is a web-based platform that allows participants to record their speech remotely. The platform is HIPAA-compliant and provides features for administrators to easily manage large cohorts of participants and speech tasks. HermeSpeech Recorder is also designed to be easy to use for both participants and administrators.

HermeSpeech Recorder has the potential to expand the pool of available speech data for research and development of speech technologies, particularly for atypical and pathological speech applications. This could lead to more inclusive and effective speech technologies that can benefit everyone.

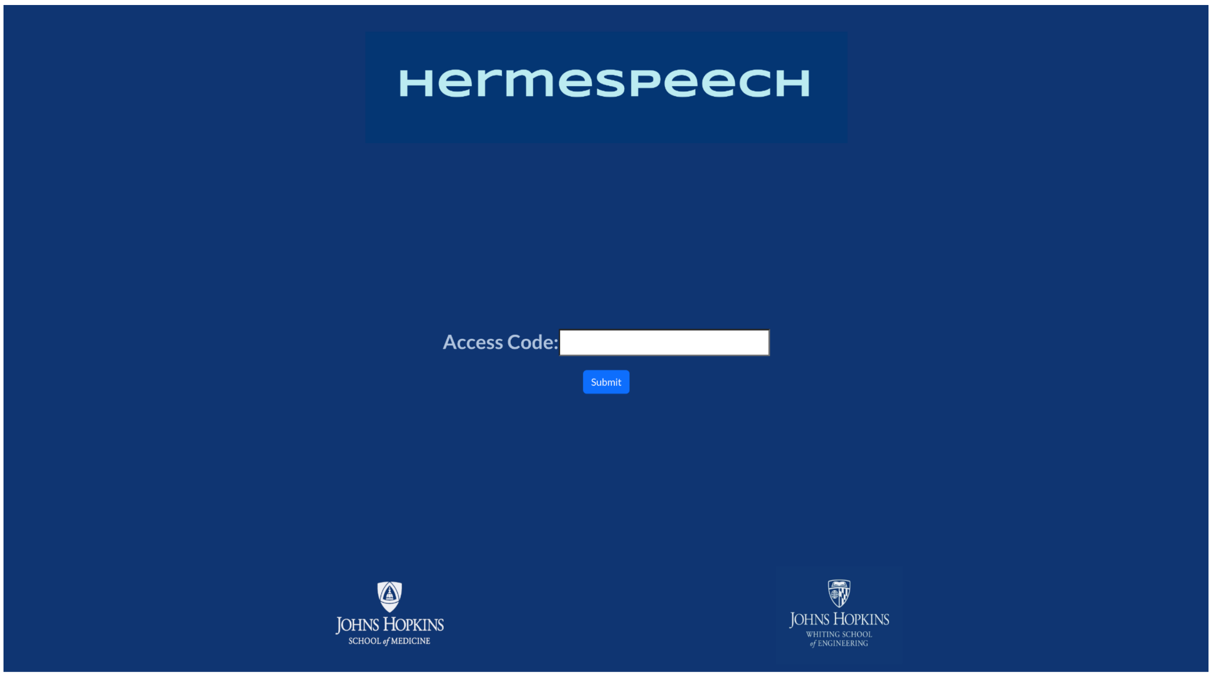
In addition to the above, here are some specific examples of how HermeSpeech Recorder can be used to collect speech data for atypical and pathological speech applications:

- To collect speech data from people with dysarthria, a speech disorder that affects muscle control and coordination.
- To collect speech data from people with aphasia, a language disorder that affects the ability to produce and understand language.
- To collect speech data from children with autism spectrum disorder (ASD), who often have difficulty with communication.
- To collect speech data from people with accents or dialects that are not well-represented in existing speech datasets.

By collecting more speech data from people with atypical and pathological speech, researchers and developers can develop speech technologies that are more accurate and accessible for everyone.

High-level Decomposition

User Login Page


HermeSpeech
Access Code:
Submit
Johns Hopkins School of Medicine
Johns Hopkins Whiting School of Engineering

The HermeSpeech login page is the gateway to the service. The page consists of two main elements: a trio of image assets that visually identify HermeSpeech and its supporters, and an access code input box that grants users entry to the platform.

The login page is a simple page with two main parts:

1. Image assets: The login page includes three image assets: the HermeSpeech service logo, the Whiting School of Engineering logo, and the School of Medicine logo at Johns Hopkins University. These image assets are styled uniformly to create a consistent visual experience for users.
2. Access code input box: The access code input box is the main element of the HermeSpeech login page. Users enter their access codes into this box to log in to the service. The access code input box is styled to be visually prominent and easy to use.

The login method for the HermeSpeech login page is simple:

1. Admins create access codes for users.
2. Users use their access codes to log in to the service.

Technical Details

The HermeSpeech login page is rendered based on a component file. This file contains the HTML, CSS, and JavaScript code for the page. The component file can be edited to make changes to the page, such as updating the text of the access code input box or changing the images that are used for the logos.

Links

- CSS Styling: client/src/App.css
- Page Rendering & Code: client/src/pages/Home.js
- Logo Images: client/src/images

Login Form

The LoginForm component is a React component that renders a form for users to enter their access code and login to the application. It takes the following props:

- `Login`: A function that is called when the user successfully logs in. This function should accept the user's access code as a parameter.
- `error`: An optional error message to display to the user.

The component renders a form with the following elements:

- An `h1` element with the text "Access Code:".
- An `input` element for the user to enter their access code.
- A `button` element for the user to submit the form and login.

Events

The LoginForm component emits the following event:

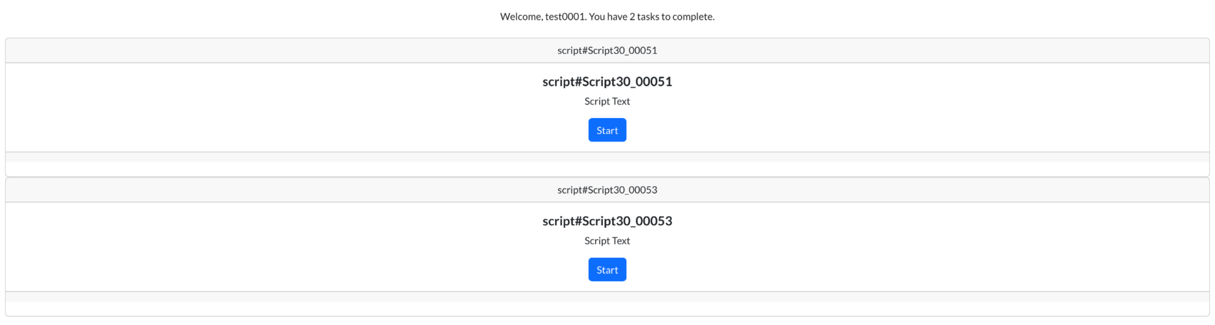
- `submit`: This event is emitted when the user submits the form. The event handler receives the following parameter:
    - `accessCode`: The user's access code.

Props

The LoginForm component accepts the following props:

- `Login`: A function that is called when the user successfully logs in. This function should accept the user's access code as a parameter.
- `error`: An optional error message to display to the user.

Dashboard



Welcome, test0001. You have 2 tasks to complete.

script#Script30_00051

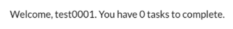
script#Script30_00051

Script Text

script#Script30_00053

script#Script30_00053

Script Text



Welcome, test0001. You have 0 tasks to complete.

Once users have successfully logged in to HermeSpeech, they are redirected to their script dashboard. This dashboard displays a horizontally long card for each script that the user is assigned to and has not yet completed recording. Each card displays the script number and/or name, and a start button.

To begin recording a script, the user simply clicks on the start button for the desired script. This will redirect them to the starting page of the script recording modal.

Here is a more detailed breakdown of the dashboard:

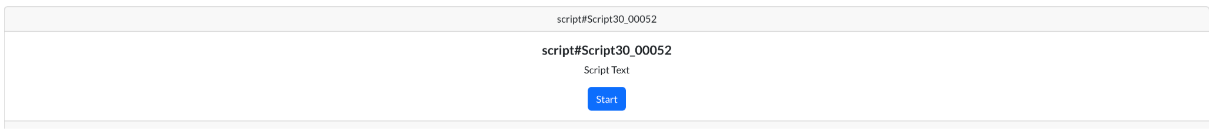
- Card format: Each script is rendered as a horizontally long card. This format allows users to see all of their assigned scripts at a glance.
- Card content: Each card displays the following information:
    - Script number and/or name: This helps users to identify the specific script that they want to record.
    - Start button: This button allows users to begin recording the script.
- Functionality: When a user clicks on the start button for a script, they are redirected to the starting page of the script recording modal.

The component uses the `useState` and `useEffect` hooks to manage its state and side effects.

- The `useState` hook is used to track the following state variables:
    - `currentState`: This object stores the user's assigned tasks and a list of DashboardCard components.
    - `isFetched`: This boolean variable indicates whether the data has been fetched from the backend API.
    - `isCardsCreated`: This boolean variable indicates whether the DashboardCard components have been created.
- The `useEffect` hook is used to fetch the user's assigned tasks from the backend API when the component mounts. It also uses the `useState` hook to update the `currentState` and `isFetched` state variables.

Once the data has been fetched, the component renders a list of DashboardCard components. Each DashboardCard component renders a single task and allows the user to complete the task.

DashboardCard



- `script_id`: The ID of the script or module.
- `accessCode`: The user's access code.
- `type`: The type of script or module (e.g., "script", "review", "revise").
- `line`: The line number of the script (optional).

The component renders a card with the following elements:

- A header with the script or module type and ID.
- A body with the script or module text and a button to start the recording or module.
- A footer with the current date and time.

Events

The DashboardCard component emits the following events:

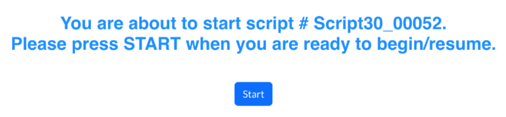
- `click`: This event is emitted when the user clicks the "Start" button. The event handler receives the following parameters:
  - `script_id`: The ID of the script or module.
  - `accessCode`: The user's access code.
  - `type`: The type of script or module (e.g., "script", "review", "revise").
  - `line`: The line number of the script (optional).

Props

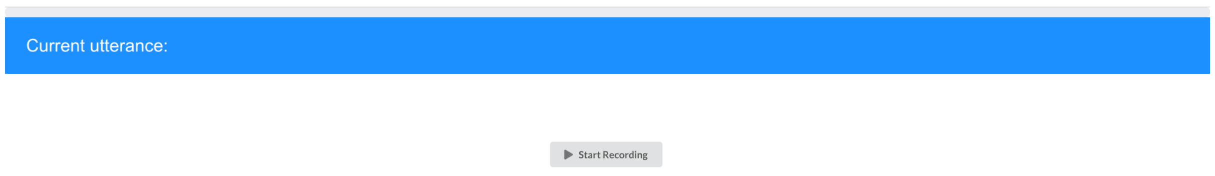
The DashboardCard component accepts the following props:

- `script_id`: The ID of the script or module.
- `accessCode`: The user's access code.
- `type`: The type of script or module (e.g., "script", "review", "revise").
- `line`: The line number of the script (optional).

Recording Page


You are about to start script # Script30_00052.
Please press START when you are ready to begin/resume.

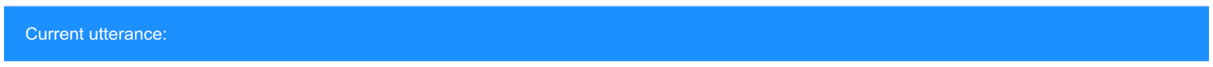
We’ve added a buffer introductory screen between selecting a script and actually starting recording to minimize any recordings made by mistake. During our development cycle, we have encountered multiple incidents where the research assistants and participants did not read the instructions and therefore did not use the platform in a way that we did not intend. This user behavior resulted in lower recording data quality. To prevent this, we have added buffers so that we can safeguard unintended user behavior as much as possible.

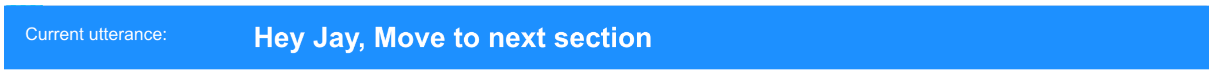


As a user presses the second start button above, the recording module screen renders. This page mainly has three components:

- Utterance (Text) Display
    - Source code: client/src/components/UtteranceDisplayer.js

UtteranceDisplay


Current utterance:


Current utterance: **Hey Jay, Move to next section**

Design Choices

- Sentences are only displayed after the recording is triggered by the user. This design choice was to minimize the destruction of data quality. During our beta testing period, our team observed that participants sometimes start speaking immediately after they press start, which sometimes doesn't necessarily mean the recording module is ready yet. This resulted in the first second or two not being recorded. By hiding the sentence until the recorder is ready to record, we were able to minimize this.
- By hiding the sentence until the recorder is ready, we also prevent participants from being influenced in any way. For example, many of the participants of our research experience atypical speech patterns, and providing them with the sentences from the very beginning, even before recording starts, in some cases provided them the time to practice the pronunciation in one way or another (either in their head, or actually vocally), and this should not happen for the purpose of the research and to maintain the data quality of our recording samples.

- To enhance the independence from any unwanted influence, we also have added a buffer period to take place between the recording module rendering is complete and the actual recording starts. This is to allow for a better guarantee of any issues with the timing in which the recording starts and also with the chained rendering of the entire recording page.
- One of the future feature requests could be to allow customization of the buffer length.

The UtteranceDisplayer component is a React component that renders the current utterance in a script recording session. It takes the following props:

- `user_id`: The ID of the user who is recording the script.
- `script_id`: The ID of the script that is being recorded.
- `line`: The array of utterances in the script.
- `current_line`: The index of the current utterance in the script.
- `currentRecordState`: A boolean value indicating whether the user is currently recording the script.

The component renders a div element with two child elements:

- A float-left-child element that displays the text "Current utterance:".
- A float-right-child element that displays the current utterance in the script, or the previous utterance if the user is currently recording.

The component also uses a `useEffect` hook to fade the current utterance in and out every second. This is done to help the user focus on the current utterance.

Implementation

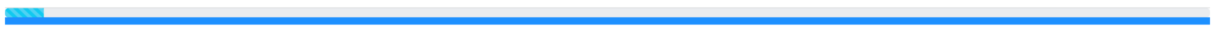
The UtteranceDisplayer component uses the following React features:

- `useState`: To manage the state of the component, including the previous utterance.
- `useEffect`: To fade the current utterance in and out every second.
- `conditional rendering`: To display the current utterance or the previous utterance depending on the `currentRecordState` prop.

Links

- Dashboard Page
    - Source code: client/src/components/Dashboard.js
- Dashboard Card Format
    - Source code: client/src/components/DashboardCard.js

ProgressBar



The `renderProgressBar` function is a React function that renders a progress bar. It takes the following props:

- `current`: The current progress, as a number between 0 and 1.
- `total`: The total progress, as a number.

The function returns a `ProgressBar` component from the `react-bootstrap` library. The `ProgressBar` component is visually hidden, but it can still be used to track progress and display it to screen reader users.

Technical Details

The `renderProgressBar` function uses the following React features:

- `props`: To receive the current and total progress as props.
- `ProgressBar` component from `react-bootstrap` to render the progress bar.

Admin

All code for the Admin pages are contained under this admin subdirectory

- https://github.com/maxzinkus/AtypicalSpeech/tree/master/client/src/components/admin

Component Structure

- State Management:
  - Utilizes `useState` for managing local component state.
  - Uses `notification` from Ant Design for displaying notifications.
- Notification Handling:
  - Defines a function `openNotificationWithIcon` to display notifications with Ant Design.
- Form Submission:
  - Defines a `handleSubmit` function to handle form submission.
  - Retrieves username and password from the form inputs.
  - Makes a POST request to '/api/auth/admin/login' using axios.
  - Displays an error notification if the login is unsuccessful.
  - Navigates to the admin page on successful login.
- JSX Structure:
  - Displays a welcome message, a form for entering username and password, and logos at the top and bottom of the page.
  - Uses Bootstrap's `Button` component for the form submission button.

AdminDashboard Component

The `AdminDashboard` component is a React functional component that represents the main dashboard for the admin interface. This dashboard includes a sidebar with navigation links and a content area where different tabs are displayed based on the selected navigation item. Below is a detailed technical documentation for the code:

- Layout Components: Destructuring Layout components from Ant Design.
- Sidebar Items: An array of objects representing the navigation items for the sidebar. Each object includes a key, an icon, and a label.
- Layout Structure:
  - Uses Ant Design's `Layout` component to structure the dashboard with a sidebar and a content area.
- Sidebar:
  - Contains a logo and a `Menu` component with navigation items.
  - Uses the `Sider` component for the sidebar.
- Content Area:
  - Displays different tabs based on the selected navigation item.
- Tabs:
  - Imports and renders custom tab components (`UserTab`, `ScriptTab`, `Statistic`, `Medias`) based on the selected navigation item.
- Footer:
  - Displays a footer with copyright information.

State Management

The component utilizes React's `useState` hook to maintain two state variables: `key` for managing the selected menu item and `navigate` for routing purposes.

User Authentication

An `useEffect` hook is employed to perform user authentication by making an API request to `/api/auth/admin`. If the user is not authenticated, the component redirects to the login page using `navigate`.

Layout and Menu

The component utilizes Ant Design's Layout component to structure the interface. The Sider section contains a vertical menu with icons representing different sections. The `onClick` handler for the menu items updates the `key` state variable, triggering the rendering of the corresponding content.

Dynamic Content Rendering

The Content section of the Layout renders different components based on the value of the `key` state variable. Currently, the component supports four sections: UserTab, ScriptTab, Statistic, and Medias.

AssignScriptMultipleUsersModal

Demo

add_scripts_demo.mov

The `AssignScriptMultipleUsersModal` component is a React functional component that represents a modal for assigning scripts to multiple users. It uses React Bootstrap components for styling and interacts with the server to fetch user data and perform script assignments. Below is a detailed technical documentation for the code:

- react-select: A flexible and customizable select component.
- makeAnimated: A utility for animating the `react-select` components.

Modal Structure

- Uses React Bootstrap `Modal` components for displaying the modal
- Header includes a close button and a title.
- Body contains a `Select` component from `react-select` for choosing users.
- Footer has buttons for closing the modal and confirming the script assignment.

User Selection

The component employs a react-select component to display a list of all users, allowing administrators to select multiple users for script assignment. The selected users are stored in the `selectedUsers` state variable.

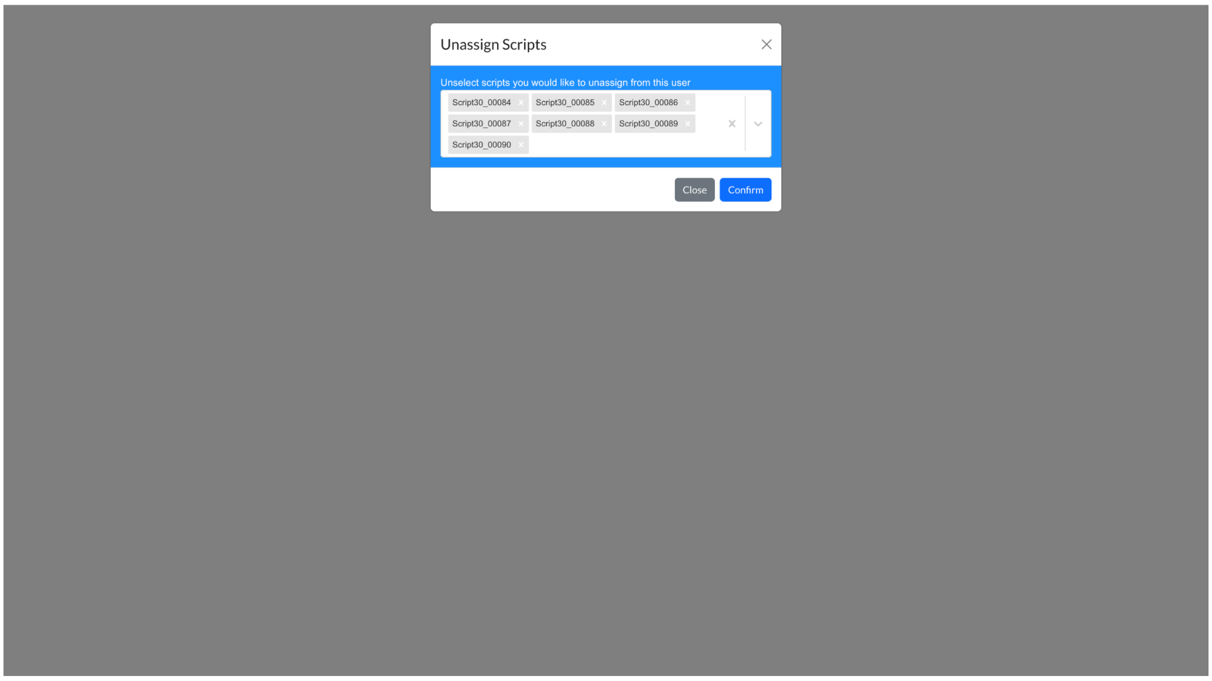
Script Assignment

When the "Confirm" button is clicked, the `handleAssignMultipleTasks` function is triggered. This function constructs an object containing the selected user IDs and the script ID, and sends a POST request to the API endpoint `/api/script/assign_task_to_multiple_users`.

Modal Handling

The component utilizes react-bootstrap's Modal component to manage the modal dialog. The `handleClose` function is responsible for closing the modal and redirecting to the admin page.

UnassignScriptSpecificUserModal


Unassign Scripts

Unselect scripts you would like to unassign from this user

Demo

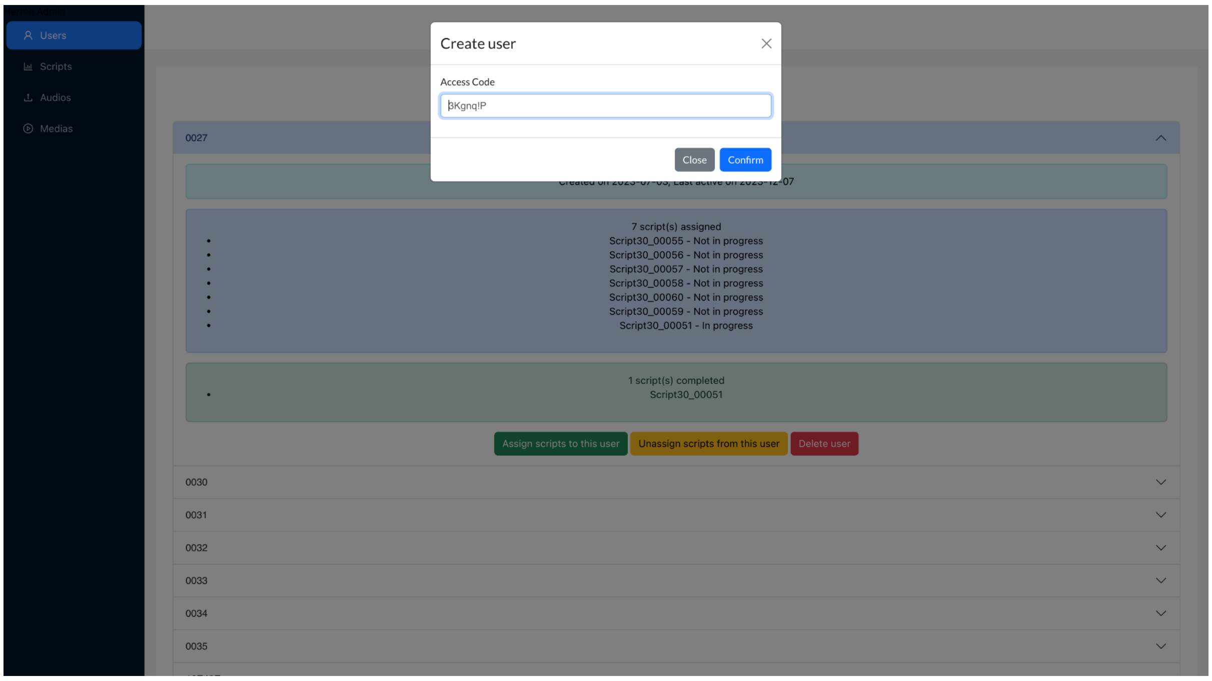
[unassign_scripts_demo.mov]

Functionality

1. Fetching Data
   - The component fetches all available scripts and assigned scripts for the specific user whose access code is provided in the location state.
   - `fetchData` function fetches available scripts and assigned scripts from the API.
   - `renderSelectModule` function creates a modal with a React Select component to allow selecting assigned scripts for unassignment.
2. Unassigning Scripts
   - The selected scripts for unassignment are stored in the `selectedScripts` state.
   - Clicking the "Confirm" button triggers the `handleUnAssignScripts` function.
   - This function sends a POST request to the `unassign_script_url` with the user ID and selected script IDs.
   - Upon successful unassignment, the modal is closed and the user is navigated back to the admin page.
3. Closing Modal

    - Clicking the "Close" button triggers the `handleClose` function.
    - This function navigates the user back to the admin page without performing any unassignments.
4. The component assumes access to the following APIs
    - `/api/script/get_all_script_ids`: Retrieves all available script IDs.
    - `/api/user/get_assigned_tasks`: Retrieves assigned scripts for a specific user.
    - `/api/script/update_assigned_task`: Unassigns scripts from a user.
5. The component assumes access to a backend API that handles script management and user authentication.

CreateUsersModal



Access Code

3Kgnq!P

0027

Created on 2023-07-03, Last active on 2023-12-07

7 script(s) assigned

- Script30_00055 - Not in progress
- Script30_00056 - Not in progress
- Script30_00057 - Not in progress
- Script30_00058 - Not in progress
- Script30_00060 - Not in progress
- Script30_00059 - Not in progress
- Script30_00051 - In progress

1 script(s) completed

- Script30_00051

0030

0031

0032

0033

0034

0035

Demo

create_users.mov

Purpose

This component allows users to create new users within the application. It is typically used in the admin panel for user management.

Functionality

1. User Input
   - The component displays a modal with an input field for entering the access code for the new user.
   - The entered access code is stored in the `currentState.accessCode` state.
2. Modal Control
   - Clicking the "Create user" button triggers the `handleShow` function, which opens the modal.
   - Clicking the "Close" button or the modal backdrop triggers the `handleClose` function, which closes the modal.
3. User Creation

    - Clicking the "Confirm" button triggers the `handleCreateUser` function.
    - This function sends a POST request to the API with the user's access code.
    - Upon successful user creation, the modal is closed.
4. State Management
    - The component utilizes the `useState` hook to manage the state of the modal (open/closed) and the user's access code.

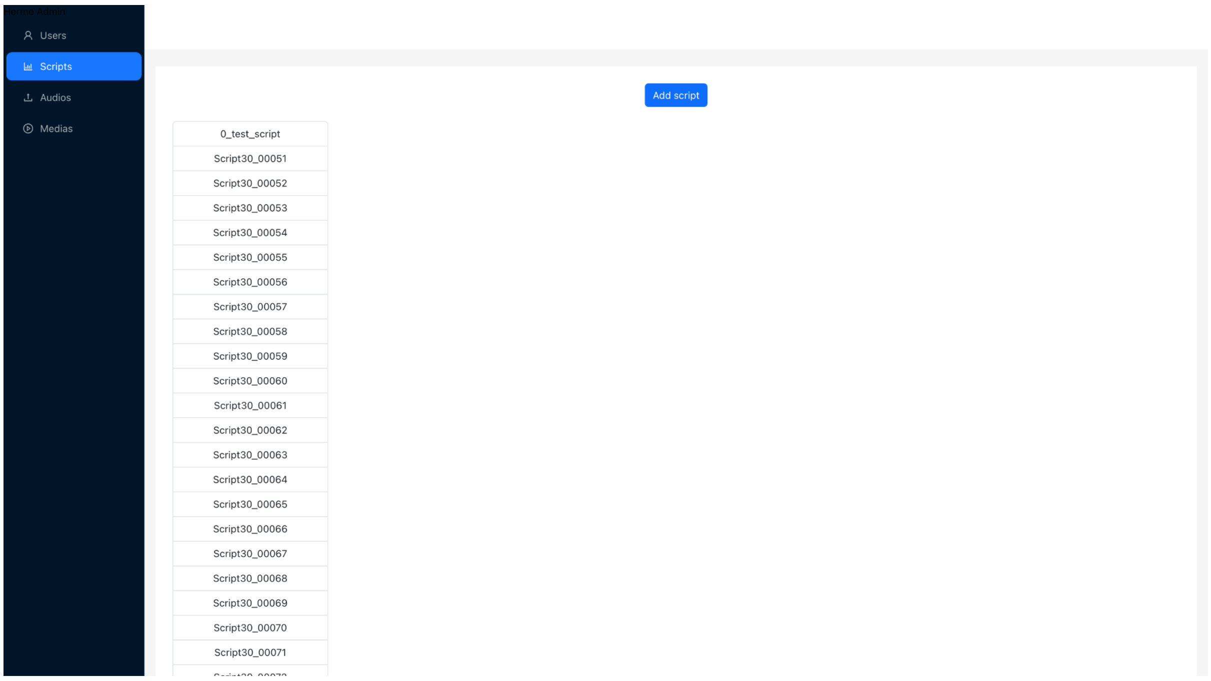
Implementation Details

- The component uses the following libraries and frameworks:
    - React
    - React Bootstrap (Button, Modal, Form)
    - axios (for API requests)
    - React Router (useNavigate)
- The component utilizes state variables to store the access code and the modal state.
- Various functions are used to handle user input, modal control, user creation, and state updates.

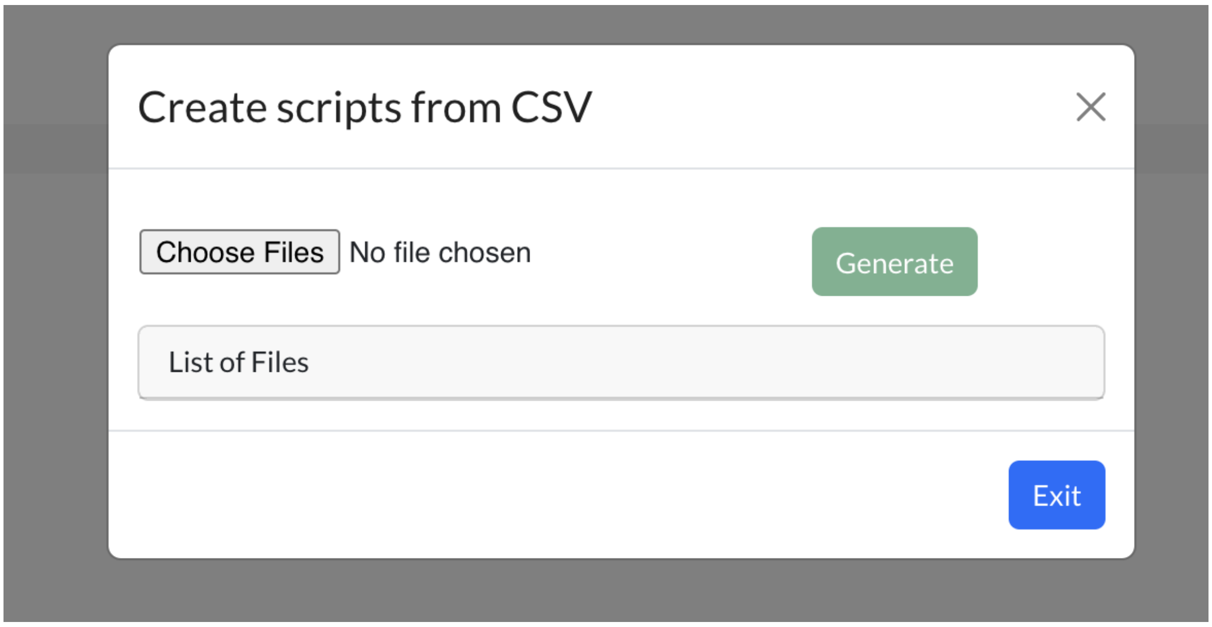
Assumptions

- The component assumes access to the following API:
    - `/api/user/create`: Creates a new user with the specified access code.
- The component assumes a backend API that handles user management and authentication.

AddScripts or CreateScripts



0_test_script
Script30_00051
Script30_00052
Script30_00053
Script30_00054
Script30_00055
Script30_00056
Script30_00057
Script30_00058
Script30_00059
Script30_00060
Script30_00061
Script30_00062
Script30_00063
Script30_00064
Script30_00065
Script30_00066
Script30_00067
Script30_00068
Script30_00069
Script30_00070
Script30_00071



Functionality

1. File Selection
   - Users can select multiple CSV files using an input field.
   - The selected files are stored in the `selectedFiles` state.
2. File Processing
   - Clicking the "Generate" button triggers the `uploadFiles` function.

    - This function iterates through the selected files and performs the following actions:
        - Reads the CSV data using Papa.parse.
        - Preprocesses the data by grouping utterances by script ID and creating details objects.
        - Sends the processed data to the API for script generation.
        - Updates the UI with progress information and success/failure messages.
3. Progress Information
    - The component displays the progress of each file upload using a progress bar.
    - The progress information is stored in the `progressInfos` state.
4. Message Display
    - The component displays success and failure messages related to the upload and processing of files.
    - The messages are stored in the `message` state.
5. File List
    - The component displays a list of uploaded files with links to download them.
    - The file information is retrieved from the API.

Implementation Details

- The component uses the following libraries and frameworks:
    - React
    - React Bootstrap (Button, Modal)
    - UploadService (custom service for file upload)
    - axios (for API requests)
    - Papa (for parsing CSV files)
- The component utilizes state variables to manage data and UI behavior:
    - `selectedFiles`: Stores the selected files.
    - `progressInfos`: Stores the progress information for each upload.
    - `message`: Stores success and failure messages.
    - `fileInfos`: Stores information about uploaded files.
    - `progressInfosRef`: Used to access progress information from within asynchronous functions.
    - `onetime`: Used to disable the button after files are uploaded once.
- Various functions are used to handle file selection, processing, progress updates, message display, and file list retrieval.

Routing traffic to HermeSpeech using NGINX

The HermeSpeech team has been using NGINX to route traffic to our HermeSpeech platform. NGINX is a reverse proxy server that can be used to load balance traffic across multiple servers, cache static content, and perform other tasks.

We have found that using NGINX to route traffic to HermeSpeech has provided us with a number of benefits. NGINX distributes traffic across multiple HermeSpeech servers, which improves the overall performance of our platform and can be used to implement security features such as SSL/TLS termination and access control.

Our NGINX configuration

Our NGINX configuration is relatively simple. We have configured NGINX to listen on port 80 and to route all traffic to our HermeSpeech servers. We have also configured NGINX to perform load balancing and caching.

This configuration will tell NGINX to listen on port 80 and to route all traffic to the `herme_speech` upstream. The `herme_speech` upstream is a group of servers that NGINX will load balance traffic across.

Deploying HermeSpeech using GitHub's deployment feature

The HermeSpeech team has been using GitHub's deployment feature to deploy code to production for the past few months. This feature has been a great addition to our workflow, as it has allowed us to automate the deployment process and make it more reliable.

Getting started

To get started with GitHub's deployment feature, we first created a workflow for each environment that we wanted to deploy to. We used the workflow templates provided by GitHub, but we also customized them to meet our specific needs.

Once we had created our workflows, we configured them to be triggered when we pushed a change to the `master` branch. We also specified which steps should be executed for each deployment.

Deploying to production

To deploy our code to production, we simply push a change to the `master` branch. GitHub will then automatically execute the workflow that we configured for the production environment.

Using environments

We also use environments to deploy our code to staging and testing environments. This allows us to test our code before deploying it to production.

To deploy our code to a staging or testing environment, we simply push a change to the corresponding branch. For example, to deploy our code to the staging environment, we would push a change to the `staging` branch.

Our workflows for the staging and testing environments are similar to our workflow for the production environment, but they may include additional steps, such as running unit tests or integration tests.

Viewing deployment history

Our deployment history can be viewed in the Deployments tab of our repository's Actions page. This shows us a list of all of the deployments that have been made to each environment.