Final Project Presentation

Created by Rehumile Sekoto

Introduction

For our final project from the Dynamic Web Applications Module, we had to create a Podcast Web App. The Podcast App will allows us to browse through various genres of podcasts, play some episodes and even track your favourites episodes.

In the next page, you will be presented with the user stories we were given to consider this project as successful.

Following that I will take you through how I built my own very Podcast app.

Then lastly, I will share some of my thoughts while building this project.

User stories

- Project is deployed to a custom Netlify URL
- All views in the app display correctly on the smallest mobile device available, “iPhone SE”. This can be emulated in Chrome Dev tools.
- All favicon information has been created an added correctly via https://realfavicongenerator.net/ (you are welcome to use any free PNG image you find on https://www.flaticon.com/)
- All metatag information has been created and added via https://metatags.io/ (You are welcome to use any free image on https://unsplash.com/). Be mindful to manually replace all URL values (especially image URL) to absolute Netlify URL values (you will need to deploy to Netlify first)
- All show data loaded via a `fetch` call from the `https://podcast-api.netlify.app/shows`
- When viewing a specific show, data is loaded via `fetch` from individual show endpoint
- There is a loading state while initial data is being loaded
- There is a loading state while new data is being loaded

User stories

- User can view the details of a show broken down into seasons, sorted by number
- User can listen to any episode in a season of a show
- User can see a view where only episodes for a specifically selected season are shown
- User can toggle between different seasons for the same show
- User can see the name of all available shows on the platform
- User sees preview image of shows when browsing
- User sees the amount of seasons per show as a number when browsing
- User sees a human-readable date as to when a show was last updated
- User sees what genres (as genre titles) a show is associated with when browsing
- User sees a preview image of seasons for a specific show
- User sees the amount of episodes in a season as a number
- User can go back to a show view from a season-specific view

User Stories

- User can mark specific episodes as favourites to find them again
- User can visit a view where they see all their favourites
- User can see the show and season of any episode in their favourites list
- Episodes related by season/show are grouped in favourites
- User is able to remove episodes from their favourites
- User can arrange the list of shows by title from A-Z
- User can arrange the list of shows by title from Z-A
- User can arrange the list of shows by date updated in ascending order
- User can arrange the list of shows by date updated in descending order
- User can filter shows by title through a text input
- User can find shows based on fuzzy matching of strings (you can use something like https://fusejs.io/)

User Stories

- User sees the date and time that an episode was added to their favourites list
- User can arrange favourites by show titles from A-Z
- User can arrange favourites by show titles from Z-A
- User can arrange favourites by date updated in ascending order
- User can arrange favourites by date updated in descending order
- Audio player shows current progress and episode length as timestamps
- Audio player is always visible, so the user can listen to episodes while they browse
- User is prompted to confirm they want to close the page when audio is playing
- App remembers which show and episode the user listened to last when returning to the platform
- App remembers which shows and episodes the user listened to all the way through
- App remembers the timestamp where the user stopped listening within a 10-second accuracy period of closing
- App remembers and shows the timestamp progress of any episode the user has started listening to
- User can "reset" all their progress, effectively removing their listening history

User Stories

- User is presented with a sliding carousel of possible shows they might be interested in on the landing page
- User can log in via https://app.supabase.com authentication
- User favourites are stored in the https://app.supabase.com database
- User favourites are automatically synced when logged in, ensuring that they share favourites between devices
- Users can share their favourites as a publicly accessible URL

Data Loading and Fetch Calls

To demonstrate how I fetched the shows data from the specified API calls. Here is an example below. Below you will see I used asynchronous functions which uses fetch request which is where the URL is placed. The fetch request will be converted in JSON to be stored in a variable and set to a state (singleShow or PodcastShows). The async functions will be called in a useEffect Hook so that is rendered when the page has loaded

```
const fetchPodcasts = async () => {
  try {
    const data = await fetch(`https://podcast-api.netlify.app/shows`);
    const result = await data.json();
    setPodcastShows(result);
    setLoadingPodcasts(false);
  } catch (error) {
    console.log(`ERROR ${error}`);
    setIsError(true);
    setLoadingPodcasts(false);
  }
};

useEffect(() => {
  fetchPodcasts();
}, []);
```

```
// set state for the single show info
const [singleShow, setSingleShow] = useState(null);

// set state for loading
const [loadingDetails, setLoadingDetails] = useState(true);

// set state for when there is an error in fetching single podcast show
const [isError, setIsError] = useState(false);

//set state for selected genre
const [selectedSeason, setSelectedSeason] = useState(1);
const [selectedSeasonData, setSelectedSeasonData] = useState(null);

//fetch the data for single podcast
useEffect(() => {
  const fetchSinglePodcast = async () => {
    try {
      const data = await fetch(
        `https://podcast-api.netlify.app/id/${podcastId}`
      );
      const result = await data.json();

      setSingleShow(result);
      setLoadingDetails(false);
    } catch (error) {
      console.log(`ERROR ${error}`);
      // setIsError(true);
    }
  };

  fetchSinglePodcast();
}, [podcastId]);
```

Loading State for initial Data

While the initial data for fetching shows or individual shows, I have added a state to set the loading process to be false once the data is fetched and true while it is still being fetched. While the loading state is true, the page will render a react spinner to show user that it is in loading state

```
if (loadingDetails) {
  return (
    <div className="loading--icon">
      <ColorRing
        visible={true}
        height="150"
        width="150"
        ariaLabel="blocks-loading"
        wrapperStyle={{}}
        wrapperClass="blocks-wrapper"
        colors={["#003EAB", "#008033", "#EEF3F6", "#003EAB", "#008033"]}
      />
    </div>
  );
}
```

```
// set state for the single show info
const [singleShow, setSingleShow] = useState(null);

// set state for loading
const [loadingDetails, setLoadingDetails] = useState(true);

// set state for when there is an error in fetching single podcast show
const [isError, setIsError] = useState(false);

//set state for selected genre
const [selectedSeason, setSelectedSeason] = useState(1);
const [selectedSeasonData, setSelectedSeasonData] = useState(null);

//fetch the data for single podcast
useEffect(() => {
  const fetchSinglePodcast = async () => {
    try {
      const data = await fetch(
        `https://podcast-api.netlify.app/id/${podcastId}`
      );
      const result = await data.json();

      setSingleShow(result);
      setLoadingDetails(false);
    } catch (error) {
      console.log(`ERROR ${error}`);
      // setIsError(true);
    }
  };

  fetchSinglePodcast();
}, [podcastId]);
```

User Interface and Navigation

```
{singleShow && (
  <div className="show" key={singleShow.id}>
    <img className="show--image" src={singleShow.image} />
    <div className="details--container">
      <h1 className="show--Title">{singleShow.title}</h1>
      <p className="show--seasons">
        <p>
          {singleShow.seasons
            ? `${singleShow.seasons.length} seasons`
            : "Seasons data not available"}
        </p>
      </p>

      <p className="show--descrip">
        {shortenDescription(singleShow.description)}
      </p>
      <span className="bold">Show More</span>
    </div>
  </div>
)}
<div className="season--form">
  {
    <form>
      <select
        onChange={handleSelectSeason}
        value={selectedSeason}
        name="Seasons"
      >
        {singleShow &&
          singleShow.seasons.map((season) => (
            <option key={season.season} value={season.season}>
              Season {season.season}
            </option>
          ))}
```

For a user to view details of a show and show seasons of that show, in a separate component file `SinglePodcastDetails`. The specific show will be fetched from a fetch request and stored in state variable called `singleShow`. Thereafter I add conditional rendering that if singleShow is truthy then it will display the properties from the shows object. The seasons of a show will then be mapped over and the season number will be stored in a drop down menu for a user to click on. Once a user has selected a season the episodes will render which shows the episode number, title and description. Below are two buttons, a heart button which a user is able to toggle to add to favourites or remove and then there is a play button which once it is clicked on it will render the audio player API.

Podcast Previews

Once data of all the shows has been fetched and saved to a state variable it will be used to render Podcast details to the page namely the title, description, last update and genres. I have create a separate `Card component` which will take props from the Podcast Preview Component to render information about the Podcast. The Card component will be mapped over and also a function that takes the genreId and return the corresponding genre from the genres arrays is passed to props to render the genres

```
 *
 */
const concatenatedGenres = props.genre.reduce((accumulator, currentGenre) => {
  if (accumulator === "") {
    return currentGenre;
  } else {
    return accumulator + ", " + currentGenre;
  }
}, "");

return (
  <>
    <Link
      style={{ textDecoration: "none", color: "#000" }}
      to={`podcast/${props.item.id}`}
    >
      <div
        onClick={() => props.handleClick(props.item.id)}
        key={props.item.id}
        className="podcast"
      >
        <img src={props.item.image} className="podcast--image" />
        <h3 className="podcast--title">{props.item.title}</h3>
        <div className="info--container">
          <p className="podcast--seasons">Seasons: {props.item.seasons}</p>
          <p className="podcast--date">
            Last Update: {changeDateFormat(props.item.updated)}
          </p>
          <p className="podcast--genres">{concatenatedGenres}</p>
        </div>
      </div>
    </Link>
```

Favourites

A separate `favourites` page has been create which a user can access by clicking the navigation bar. Using react router dom, it will use `useNavigate` to navigate to the favourites pages. The favourite episode composite key will be passed down as props from the `App` which I can then use to fetch an individual show. Below you see it renders information about the favourite episode including the image, title and season. The same button used in a single podcast episode to add to favourites is also used in favourites toggle off and remove the favourite episode

```
 *
 */
const concatenatedGenres = props.genre.reduce((accumulator, currentGenre) => {
  if (accumulator === "") {
    return currentGenre;
  } else {
    return accumulator + ", " + currentGenre;
  }
}, "");

return (
  <>
    <Link
      style={{ textDecoration: "none", color: "#000" }}
      to={`podcast/${props.item.id}`}
    >
      <div
        onClick={() => props.handleClick(props.item.id)}
        key={props.item.id}
        className="podcast"
      >
        <img src={props.item.image} className="podcast--image" />
        <h3 className="podcast--title">{props.item.title}</h3>
        <div className="info--container">
          <p className="podcast--seasons">Seasons: {props.item.seasons}</p>
          <p className="podcast--date">
            Last Update: {changeDateFormat(props.item.updated)}
          </p>
          <p className="podcast--genres">{concatenatedGenres}</p>
        </div>
      </div>
    </Link>
```

```
          Share Episode
        </button>
        {sharedURLs[episode.episode.episode] && (
          <div className="Url">
            <p>{sharedURLs[episode.episode.episode]}</p>
          </div>
        )}
        <div className="favourite--buttons">
          <div
            onClick={() => playSelectedEpisode(episode.episode)}
            className="play--button">
            <IconButton
              aria-label="playbutton"
              size="large"
              sx={{ color: "#008033", fontSize: "3rem" }}>
              <SmartDisplayOutlinedIcon fontSize="inherit" />
            </IconButton>
          </div>
          <div
            onClick={() =>
              toggleFavourite(episode.show, episode.season, episode.episode)}>
            <IconButton
              sx={{
                color: "red",
                fontSize: "3rem",
              }}
              aria-label="favoourite"
            >
              <FavoriteIcon fontSize="inherit" />
            </IconButton>
          </div>
        </div>
```

Filtering Functionality

For the user to be able to arrange the shows from most recent to least recent and from ascending to descending in alphabetical order, I have implemented a component `GenreSortFilter` which will render a button group and have the names of the sorting options. These buttons will have a onclick function which will pass a `sortPodcast` function taking in a string as argument. Using the Array.sort function it will then sort the podcast Accordingly and store them in a state variable

```jsx
import { Button, ButtonGroup } from "@mui/material"
import '../SortFilterComponent/SortFilter.css'

export default function SortFilter({ sortPodcast}) {


    return (
        <div className="filter--sort" >

        <ButtonGroup
        size="small"
        color="primary"
        variant="outlined"
        aria-label="outlined button group"
        sx={{ width: '18rem'}}
        >

              <Button onClick={()=> sortPodcast("mostRecent")}>Most Recent</Button>
              <Button onClick={()=> sortPodcast("leastRecent")}>Least Recent</Button>
              <Button onClick={()=> sortPodcast("titleAZ")}>a - z</Button>
              <Button onClick={()=> sortPodcast("titleZA")}>z - a</Button>
            </ButtonGroup>
        </div>
    )
```

```jsx
const sortPodcast = (order) => {
  setSortedPodcasts(order);

  const orderedShows = [...podcastShows];

  switch (order) {
    case "mostRecent":
      orderedShows.sort((a, b) => new Date(b.updated) - new Date(a.updated));
      break;
    case "leastRecent":
      orderedShows.sort((a, b) => new Date(a.updated) - new Date(b.updated));
      break;
    case "titleAZ":
      orderedShows.sort((a, b) => a.title.localeCompare(b.title));
      break;
    case "titleZA":
      orderedShows.sort((a, b) => b.title.localeCompare(a.title));
      break;
  }
  setPodcastShows(orderedShows);
};
```

Search

For users to be able to search for podcast. I have added a search component which filter podcast shows based on the titles. I have used `Fuse.Js` to perform a fuzzy searching which means that the search results include matches even if the user's input is not an exact match. The search results will be updated in realtime from the use of `useState` and `useEffect`

```
export default function Search({ podcastShows, setSearchResults }) {
  const [query, setQuery] = useState("");

  const handleSearch = (query) => {
    const fuse = new Fuse(podcastShows, {
      keys: ["title"],
    });

    const results = query
      ? fuse.search(query).map((result) => result.item)
      : podcastShows;
    setSearchResults(results);
  };

  useState(() => {
    if (query !== "") {
      handleSearch(query);
    }
  }, [query, podcastShows]);

  return (
    <>
      <form className="filter--search">
        <label>Search</label>
        <input
          type="text"
          value={query}
          onChange={(e) => {
            setQuery(e.target.value);
            handleSearch(e.target.value);
          }}
        />
      </form>
```

Genre Filter

For users to be able to filter shows by genre, I have created a component which renders a dropdown menu to the page allowing the user to select from the various genres. Then when the users selects it, it will update the state variable `selected genre` which will then be used to filter using the Array.filter function to filter the shows that contains that genre

```
export default function GenreFilter({selectedGenre, setSelectedGenre}) {

  const handleChange = (event) => {
      setSelectedGenre(event.target.value);
    };
  return (
      <div>
    <FormControl  sx={{width: '17rem', borderRadius: '5px'}}>
      <InputLabel >Genres</InputLabel>
      <Select

        labelId="demo-simple-select-label"
        id="demo-simple-select"
        value={selectedGenre}
        label="Genre"
        onChange={handleChange}
        name="genre"
        autoWidth
      >
        <MenuItem value=''>All Genres</MenuItem>
        <MenuItem value='Personal Growth'>Personal Growth</MenuItem>
        <MenuItem value='True Crime and Investigative Journalism'>True Crime
        <MenuItem value='History'>History</MenuItem>
        <MenuItem value='Comedy'>Comedy</MenuItem>
        <MenuItem value='Entertainment'>Entertainment</MenuItem>
        <MenuItem value='Business'>Business</MenuItem>
        <MenuItem value='Fiction'>Fiction</MenuItem>
        <MenuItem value='News'>News</MenuItem>
        <MenuItem value='Kids and Family'>Kids and Family</MenuItem>

      </Select>
    </FormControl>
</div>
```

Carousel Shows

For a user to be presented with sliding carousel of shows they might be interested in, I have imported a carousel package `react-slick` to use. It also imports `carouselSettings` which is an object that will configure the behaviour of the carousel. The carousel is responsive meaning it shows 3 slides at a time on larger screens and 2 screens of a medium sized screen

```
export default function Carousel({ podcastShows, handleClick }) {
  return (
    <Slider {...carouselSettings} className="show--carousel">
      {podcastShows.slice(0, 8).map((show) => (
        <div
          key={show.id}
          className="carousel--slide"
          onClick={() => handleClick(show.id)}
        >
          <img className="carousel--image" src={show.image} alt={show.
          <div className="show-details">
            <h3 className="show-title">{show.title}</h3>
            <p className="show-seasons">Seasons: {show.seasons}</p>
          </div>
        </div>
      ))}
    </Slider>
  );
}
```

Log in via Supabase

For a user to be authenticated, I used supabase. I have created 2 components, one for logging in and another for signing up. Using supabase methods I was able to allow user to log in using the `supabase.auth.signInWithPassword` and sign in using the `supabase.auth.signUp`. Once a user has signed it will redirect them to the email account to confirm their email.If a user is logged they will be redirected to the podcast preview page

```
return (
  <>
    <div className="auth--form">
      <div className="form--info">
        <p className="podcast--title">Podcast Hub</p>
        <p className="text">Want to Log in?</p>
      </div>
      <form onSubmit={handleSubmit}>
        <div className="inputBox">
          <input name="email" onChange={handleChange} />
          <span>Email</span>
        </div>
        <div className="inputBox">
          <input type="password" name="password" onChange={handleChange} />
          <span>Password</span>
        </div>

        <button className="submit--button" type="submit">
          Submit
        </button>
      </form>
      <p>
        Do not have an account?{" "}
        <Link to="/signup">
          <span className="Link">Sign Up</span>
        </Link>
      </p>
    </div>
  </>
);
```

Supabase favourites Management

To manage a user's favourites in a supabase database, I have create 3 asynchronous functions which will add, remove and fetch data from the favourites database. The `addToFavouritesDatabase` function will add an episode to the supabase database. It will take the `favourite_id` as a parameter. Thereafter it constructs a favourite object containing the necessary data. Then it will use the supabase object to insert the `favouriteEpisode` data into the `favourites` database. Once successful, a message is logged and then it calls the `fetchFavouriteEpisodesFromDatabase` to update it. This function is similar to the `removeFavouritesFromDatabase` and `fetchFavouriteEpisodeFromDatabase`but instead it uses the supabase object to fetch all episodes and delete a episode where the id matches the user id.

Supabase Favourites Management

```
const addToFavouritesDatabase = async (favouriteEpisodeId) => {
  if (!session) {
    alert("User not authenticated")
    return;
  }
  try {
    const favouriteEpisode = {
      favourite_id: favouriteEpisodeId,
      id: session.user.id,
      date_added: new Date(),

    }
    const {data, error} = await supabase
    .from('favourites')
    .insert([favouriteEpisode])
    if (error){
      console.error('error saving episode to favourites.' ,error)
    } else {
      console.log('episode is added to favourites', data)
      fetchFavouriteEpisodesFromDatabase()
    }
  } catch (error) {
    console.error('error saving to favourites', error.message)
  }
}
```

```
const removeFavouriteFromDatabase = async (favouriteEpisodeId) => {
  try {
    const {data, error} = await supabase
    .from('favourites')
    .delete()
    .eq('id', session.user.id)
    .eq('favourite_id', favouriteEpisodeId)
    if (error) {
      console.error('Error removing from favorites:', error);
    } else {
      console.log('Episode removed from favorites:', data);
  fetchFavouriteEpisodesFromDatabase() // Refresh the favorite episodes after removal
    }
  } catch (error) {
    console.error('Error removing from favorites:', error.message);
  }
}

const fetchFavouriteEpisodesFromDatabase = async () => {
  if (!session) return; // Exit if the user is not authenticated

    const { data, error } = await supabase
      .from('favourites')
      .select('*')
      .eq('id', session.user.id); // Filter favorite episodes based on user ID

    if (error) {
      console.log('error fetching episodes', error)
    } else {
      setFavourites(data)
    }
};
```

Audio Player

For a user to listen to an episode from the shows, I have added a functional audio player component which will be rendered when the user clicks on the episodes `play button`. In the component there is state variables which will change the audio's behavior which includes tracking whether the audio is playing or not, the current time of the audio, total duration, tracking episode progress to resume playback from the last played position and to track whether a episode has listened to episode all the way.

```
<div onClick={handleCloseAudioPlayer}>
  <IconButton  sx={{fontSize: '0.5rem',color:'white'}}>
    <CloseIcon/>
  </IconButton>
  </div>
      <audio
      id="audioPlayer"
      ref={audioRef}
      onLoadedMetadata={onLoadedMetadata}
      onTimeUpdate={onPlaying}
      currentTime={episodeProgress[currentEpisode?.id] || 0}
      />

   <div className="audio--info">
      <p className="episode--title"> Episode {currentEpisode.episode} - {currentEpisode.title}</p>
   </div>

      <div className="progress">


    <div className="navigation--wrapper" ref={progressBarRef}>
    <div className='seek-bar'style={{width: `${(timeProgress/ duration) * 100}%`}}
     onClick={clickSeekBar}></div>
    </div>
    <div className="time--progress">
        <span className="time current">{formatTime(timeProgress)}</span>
    <span className="time">{formatTime(duration)}</span>
      </div>

</div>

 <div className="controls--wrapper">
    <div onClick={togglePlayPause}>
   <IconButton  sx={{fontSize: '2rem',color:'white'}}>
      {isPlaying ?<PlayArrowIcon fontSize="inherit" /> : <PauseIcon fontSize="inherit"/> }
```

Audio Player

To prompt a user to confirm if they want to close the page while audio is playing I have made use of `useEffect` which adds an event listener to the window and calls the `handleBeforeUnload` This function will check if the audio is still playing. If it is it will then prompt user confirm that they want to close to page

```
useEffect(()=> {
  const audioElement = audioRef.current
  const handleBeforeUnload = (event) => {
    if (!audioElement.paused) {
      event.preventDefault()
      event.returnValue =''
      return ''
    }
  };



  window.addEventListener('beforeunload', handleBeforeUnload);
  return () => {
    window.addEventListener('beforeunload', handleBeforeUnload)
  };
}, []);
```

Audio Player

For the app to remember the show and episode a user played last I created a `useEffect` that plays the audio player and set `isPlaying` to true. Thereafter I set the current episode in local storage. And then I have a separate function which will get the episode and the set state to that episode. When the user returns to the platform, the audio player will already be present for the user to continue listening to that episode.

```
useEffect(() => {
    if (currentEpisode) {
      audioRef.current.src = currentEpisode.file;
      audioRef.current.play();
      setIsPlaying(true)
     // save the last played episode to local storage
    localStorage.setItem('lastPlayedEpisode', JSON.stringify(currentEpisode))
    }
  }, [currentEpisode]);
```

Audio Player

For the app to remember the show and episodes the user listened to all the way through. I am storing a list of completed episodes in local storage and updating it when the user closes the audio player. This functionality is implement when the user clicks on the close button.

```
const handleCloseAudioPlayer = () => {
  audioRef.current.pause()
  setIsPlaying(false)
  if (currentEpisode) {
    updateProgress(currentEpisode.episode, audioRef.current.currentTime);
    localStorage.setItem('lastPlayedEpisode', JSON.stringify(currentEpisode));

    if (!completedEpisodes.includes(currentEpisode.title)) {
      setCompletedEpisodes([...completedEpisodes, currentEpisode.title])
      localStorage.setItem('completedEpisodes', JSON.stringify(completedEpisodes))
    }
  }

  saveLastPlayedProgress()
}
```

Then to reset a user's progress, I created a function which will remove episodes from local storage. This function is implemented when user closes the audio

```
const resetProgress = () => {
  localStorage.removeItem('lastPlayedEpisode');
  localStorage.removeItem('completedEpisodes');
  localStorage.removeItem('lastPlayedProgress');
  setCompletedEpisodes([]);
  setEpisodeProgress({});
};
```

Overall thoughts

The project was great challenge. Lots of user stories were implemented and that was a difficult challenge in itself. There is still a lot of functionality that needs to be added and a few things that need to be refined. Although it was challenging it allowed me to learn a lot more about using the React framework and all the other technologies I have implemented such as the React router, FuseJs, Material UI.

Thank you.