IMY 220 Project 2024

Playlist Sharing Website

Deliverable 1

Instructions

- This deliverable must be **completed by 07:30 on 13 September** as indicated on ClickUP.
- You must **demo** your deliverable submission in the demo sessions in order to be awarded marks.
- **No late / email submissions will be accepted**.

We recommend you constantly refer to the overall spec as well as your individual deliverable specs to make sure your project meets all of the requirements by the end.

Make sure you also continuously test your project as you add new functionality to make sure everything keeps working as expected.

The focus of this deliverable is to start working on the **basic structure** of your project. This includes setting up a basic React project using webpack and babel as described in the 'Lecture 19' videos, separating the different project functionality into components and pages, as well as linking these components and pages together.

- By the end of this deliverable, all of the **frontend React code** should be completed. That is, **all of the basic HTML** for each page should be present, as well as some **basic functionality** for specific components (see below). Additionally, **routing** between different pages should be implemented.

- At this stage, all of your data may be **dummy data**, i.e., you do not need to fetch data from any backend yet. However, even though your data may be dummy data, **your data must be dynamic**, i.e., passed down to your components. In other words, your components may **not** be hard-coded with values, they must receive data through their props from their parents.

- You also **don't need any functionality that requires a backend**, i.e., login, etc. and **no styling** for this deliverable (everything can be basic HTML).

This deliverable aims to guide you through everything. Please see below for further explanation / details.

Part 1: Basic React Project

This deliverable requires you to set up a **basic React project using webpack and babel** as taught in the 'Lecture 19' videos (under *NodeJs and React* on Clickup). **You may not set up your project using a builder like Create-react-app, or Vite, etc**. You must do it *manually*, the way that is described in the lecture videos.

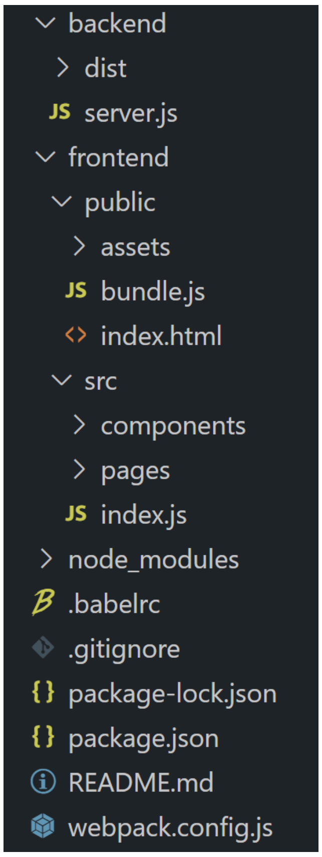
You do not need to wait until the point in the semester where you learn this setup before starting this deliverable. You can continue with the following sections in the meantime, as you know enough about React components that you can start developing and testing them, and then copy them over into your React project once you learn how to set it up. Alternatively, you can watch ahead to see how to complete this step.

After this basic setup is **complete**, follow the steps below to change your file structure slightly to better organise your files for this project.

1. In the **root** of your project, create a folder called **frontend**. Place your **public** and **src** folders (with everything currently inside them) in this folder. This is where your *frontend* (React) code will live.
2. In the **root** of your project, create a folder called **backend**. Place your **server.js** in this folder. This is where your *backend* (Express and MongoDB) code will live.
3. Inside your **src** folder (that is now in frontend), create two sub folders called **components** and **pages** for your different React components and pages respectively. This will help you better organise your React code. **Your main index.js file should still remain in the src folder**, not in any sub folders.
4. Inside your **public** folder (that is now in frontend), create a folder called **assets**, and another called **images** inside of **assets**. This is where your website images (e.g., background images, etc.) should live.

From now on, when you create different files, you should place them in the correct folder based on this structure. You are welcome to create more subfolders inside this structure to suit your needs, but you should follow the overall hierarchy above. This will help you keep track of where all your files are when the number of files increases.

Your final folder structure should look something like this:



```
backend
  dist
  server.js
frontend
  public
    assets
    bundle.js
    index.html
  src
    components
    pages
    index.js
node_modules
.babelrc
.gitignore
package-lock.json
package.json
README.md
webpack.config.js
```

You will now notice that your project will **no longer transpile** because our scripts in our ***package.json* and *webpack.config.js*** files cannot find the files we have moved around. Change the ***package.json* and *webpack.config.js*** files as follows to point to where our files have moved.

In package.json:

1. Change your main entry point to include the frontend folder: `"main": "frontend/src/index.js"`
2. Change the `"build"` script to include the *backend* folder: `"webpack && babel backend/server.js -d backend/dist"`.

   This will tell webpack to transpile server.js (that is now in the *backend* folder) and put the *dist* folder, with the transpiled *script.js*, also in the *backend* folder.
3. Change the `“start”` script to include the *backend* folder: `"npm run build && node backend/dist/server.js"`.

   This will tell webpack where the correct *server.js* file (in dist) is to run.

In webpack.config.js:

1. Change the `entry` point to include the *frontend* folder: `entry: path.join(__dirname, 'frontend', 'src', 'index.js'),` OR `entry: "./frontend/src/index.js"`.

   This will help webpack find your *index.js* file (that is now in the *frontend* folder)

2. Change the **`output path`** to output to the *frontend* folder: `path: path.resolve(__dirname, 'frontend', 'public')`

Some other general tips:

- You will need to change *webpack.config.js* to include and make use of other babel-loaders (besides just babel-loader) in the module.rules to **properly transpile and load css files, images, etc.** These will have to be installed as dependencies. Have a look at style-loader, css-loader and file-loader.
- If you are going to use **.jsx files**, you will need to change your test condition in *webpack.config.js*. Find the module.rules and the rule specifying how you test for .js files, change this to include .jsx.

Part 2: Frontend Components with Dummy Data

At the core of every React Application are components. They are reusable pieces of code that you can use and reuse throughout the different pages of your project, and they aim to encapsulate one 'object' / function. You can think about React components like classes in an object-oriented paradigm. They build on one another, but are self-contained enough that they are able to be reused anywhere without much reliance on other components (ideally).

This deliverable will guide you on how to divide your different pages into components based on the functionality you need. In a real-world scenario, you would need to consider the needs of your application and decide how YOU would split up the required functionality into pieces (like you might in an OOP module like COS 110 or COS 214). Since you may not have the 'sense' of how to do this yet, this deliverable gives you a starting point on how to go about doing this.

You are welcome to create additional components based on how you visualise your application, but you should have **at least** the following components for each page:

Note*: this list is* ***not*** *a list of the required content on each page, but rather what content must be encapsulated in a component form rather than just being on the page. You may add extra components if you wish, this is to get you started.*

Your HTML should make use of semantic HTML elements (nav, main, aside, article, etc.) where appropriate. This is good practice.

General

- **Song** Component (contains all required information on a single song)
- Component to **add a song to the website** (contains all the form information to let a user add a song)
- Component to **add a song to a playlist** (the context menu that contains all the information to let a user add a song to a playlist - the adding functionality does not need to be implemented yet)
- **Playlist Preview** Component (contains all basic information on a single playlist that is viewed in a list, search results, or in a feed - i.e., not the entire playlist, but a small preview of it)
- **Profile Preview** Component (displays all basic information on a profile, that is viewed in follower / following lists, search results, etc. - i.e., not the entire profile, but a small preview of it)
- **Header** / Navigation Component (contains your page **routing** - the 'navbar' and must be present on **all** pages besides the Splash page)

Splash Page

- **Login** Form (contains all the information / inputs required to log in, the login functionality does not need to be implemented yet)
- **Sign up** Form (contains all the information required / inputs to sign up, the sign up functionality does not need to be implemented yet)

Home Page (with Feed)

- **Feed** Component (displays the song / playlist preview components - this can be one component that handles the display of both feeds, or a separate component for each)
- **Search Input** Component (handles search inputs - the search functionality does not need to be implemented yet, you only need to be able to enter a search term based on your planned functionality)

Profile Page

- **Profile** Component (contains basic profile information)
- Component to **edit** a profile (contains all the form information for editing the user's profile)
- Components to **list** playlists / songs on the users' profile (you could make use of the Feed component for this if done correctly, however it is up to you on how you manage this functionality).

- **Follower** / **Following** Components (displays all users that are following / followed by the current user (hint: profile preview components) - this can be one component that handles the display of both, or a separate component for each)
- Component to **create** a playlist (contains all the form information for adding a playlist)

Playlist Page

- **Playlist** Component (contains all required information on a single playlist)
    - You would need to be able to **list all of the songs** (hint: song component) that belong to a playlist. This could be done in the playlist component itself, or in a separate component, depending on how you manage this functionality.
- Component to **edit** a playlist (contains all the form information for editing a playlist - this could use the same component as the one to add a playlist, depending on how you manage this functionality)
- **Comment** Component (contains all required information on a single comment)
- Component to **list** all of the comment components
- Component to **add** a comment (contains all required information to add a comment)

Part 3: Basic Functionality

All of the components listed above **do not need to be functioning yet**. For example, your login form doesn't have to have any functionality to actually log in the user, only the basic HTML elements that would allow a user to do so if this was implemented. Your Feed component can take in dummy data in order to render Song / Playlist preview components, it does not need to do any sort of API call yet.

However, some components should have **basic <u>client-side</u> functionality**. This is to get you started on implementing some of the functionality you need before you are required to implement backend code in Deliverable 2.

Choose two of your form components you have created (i.e., any components that make use of HTML inputs) and implement **basic <u>client-side</u> validation for them**.

- For example, checking if a value filled in is the correct number of characters or has a required symbol (e.g., @), checking (if applicable) if password and repeated passwords match, checking if input fields are not empty, etc.

- **You may pick any of your form components**, e.g., components with inputs to add / edit a piece of content, a login form, etc.

- You do **not need to do any validation that requires a backend**. However, you **do need to make use of component state** to make sure your validation functions correctly (i.e., the form is not able to submit until all required fields are valid).

Part 4: Routing

Finally, set up **basic page routing** to access each of your main pages via the URL.

You may choose how you would like to go about this, either via Express routes, or using *react-router-dom* package like in the lecture videos.

- What is important is that a user is able to **navigate** to the **Splash** page, **login / sign up** forms (if they are not on the splash page), **Home** page, **Profile** page and **Playlist** page via the URL (i.e., "/", "/home", etc.). You may add any additional routing to any other pages you have if you wish to do so.

- Your **Profile** page and **Playlist** page should make use of **dynamic routes**. Eventually, you will need to be able to access a specific playlist or profile **through its id** in order to render a specific profile using the same component template. This is done by **passing an id through the URL**. For now, you can render the same profile regardless of which id is passed through the URL, but you should be able to handle passing any id (e.g., "/profile/23532").

Submission Instructions

Submit the following to ClickUP before the deadline:

Place these in a folder named with your `Position_Surname_D1`.

- Upload all your files, **excluding node_modules**, and including all relevant files to build and run your Docker container.
- Include a **README** with the commands that you use to build and run your Docker image.
- Include a .txt file with a link to your GitHub repository. At this point you should have **two extra commits** (you can commit these to your main branch)

Zip the **folder** and upload this to ClickUP in the relevant submission slot before the deadline.