Assignment: Forms in React

The purpose of this assignment is to learn how to create a more complex React application with a few additional features, particularly forms. Actually, not only will we create a front-end React application, but also a back-end API for it to connect to!

As for the React part of the application, here are a few things we'll do in this assignment:

- Implement forms the React way;
- Submit data to an API;
- Update the application to reflect data we've submitted;
- Have components clean up after themselves when they've unmounted;
- Explore different ways of organizing the application and reusing code;

The purpose of the application will be to find hilarious jokes and save them in playlists. The jokes will come from this joke API: https://karljoke.herokuapp.com

(As far as I can see, the jokes from the above API are clean and inoffensive; however, I haven't read them all, so if you see something problematic, please let me know. I don't take responsibility for the content, but l can find a different API for us to use.)

For the playlists, (each of which will contain a list of jokes chosen from the joke API) you will create your own, very basic API running as an Express application on the back end.

As usual, a **demo** video, showing how the final product should work, has been included in the starter files.

Note that this assignment will require that you write both a back end application and front end application.

An important note about the joke API: they have a usage limit and will block your IP if you make too many requests within a short period of time. As you're writing your application, try not to do this! But if you do happen to get blocked, they'll unblock you automatically after 15 minutes.

Another note: this assignment is somewhat complicated, so for simplification, we're not going to worry about things like form validation, loading indicators, etc. We're not even going to use a real database on the backend (although you can if you want to.) Instead, just use an array, and be aware that it will be cleared every time the server-side application restarts.

One more note: I've provided a **style.css** file that you can use - in order for it to work properly, each component will need to be stored inside a wrapper div with **className="name-of-component'.** This styling is optional.

Task 1: React Setup and Random Joke List

Your first task is to set up a basic React application, packaged with Webpack from source files in a folder called **src** and served from a static folder called **public.** The application should have a top-level component called **App** with a child component called **RandomJokes.** This child component should do the following:

- Use an effect hook to fetch, once upon mounting, **six** random jokes from the "programming" category of the API. (Use the API documentation to figure out how to do this.)
- Render the jokes in a list;
    - Each joke should be rendered using a **Joke** component, which you should create.
    - Each **Joke** component should simply display the **setup** and **punchline** as separate paragraphs or divs.
- **Warning:** make sure you configure the effect hook to only run once upon mounting. Otherwise, you may inadvertently surpass your usage limit for the API and be blocked for 15 minutes.

Although it would be nice to have a button to refresh the list of 6 jokes with 6 new ones, don't worry about that, as we're trying to keep the application simple.

Task 2: Create Playlist

Next, create a component called **CreatePlaylist** and render it in **App.** The purpose of this component is to provide a form by which the user can create a new playlist, which is a list of saved jokes, presumably the most hilarious. Since the Joke API doesn't have any functionality for saving favorites, you'll have to create your own API for this.

In your server side application, create a POST route with endpoint **/playlists**. For now, it shouldn't do anything other than send back a response message that says "success". Eventually, this route will be used to create a new playlist and save it on the back end.

In your React application, have the **CreatePlaylist** component render a form with a **text input** for the new name. (Remember to use a <label> element for accessibility.) Also create a submit button. The form should be implement using **controlled components,** which means the **value** attribute of the text input should be stored in the **state** of its parent component and updated each time its changed, using an **onChange** event handler.

When the form is submitted, your React application (rather than the browser) should handle the submission, using an **onSubmit** event handler, and send a POST request to your API to create the new playlist.

Task 3: API

Spend a few minutes now fleshing out your server-side API; in particular, you'll need route handlers for **GET '/playlists'** and **POST '/playlists'**, which get an array of playlists and create a new playlist, respectively. Each playlist should be represented as an object with a **name** and **array of jokes.**

Ideally, one would use a database for this; however, since this assignment is already quite long, feel free to use a local variable in **app.locals** to store the array of playlists. This is not a persistent solution, and all lists will be cleared when the application is restarted.

Task 4: List of Playlists

Now, create a component called **Playlists** and render it in **App**. This component should display a list of all playlists, fetched from the API, that have been saved so far. None of them will have any jokes in them yet, but you should be able to see a list of names.

Note that every time a new playlist is created (using the **CreatePlaylist** component you created earlier) the list displayed in the **Playlists** component should be updated automatically without a page refresh. Figure out how to make this happen. (hint: since **CreatePlaylist** will need to trigger a reload of all playlists, and **Playlists** will need to render the most up-to-date set of playlists, and these components are siblings, some of the program logic will have to be moved to their common parent, which is **App.**)

You've probably noticed that there's a way to update the array of playlists on the client side without making a call to the API; however, let's assume that other people could use the same application and add their own playlists, which means we would have to make an API call to get all updates.

We'll need a way to add each joke to a chosen playlists, but we'll implement that later.

Task 5: Featured Joke

Create a new component called **FeaturedJoke** and render it in **App**. The purpose of this component is to load a new random joke from the API every 8 seconds and render it. However, don't worry about the loading of the joke yet, or the 8 second interval. For now, just have it render a heading that says **"Featured Joke".**

Now, in **App**, create a button that toggles the **FeaturedJoke** component, either hiding or showing it when clicked. However, don't simply use CSS to hide or show the component; the component should be **mounted** or **unmounted** each time the button is clicked. Hint: store the

shown/hidden status of the **FeaturedJoke** component in the **state** of **App**, and conditionally render **FeaturedJoke** based on this state. (I'm not saying this method is necessarily the best, but it will allow us to practice a particular technique later.)

Now, to implement the behavior of loading a new joke every 8 seconds: use an effect hook to set up—once, when **FeaturedJoke** mounts—an **interval** function that is called every 8 seconds and loads one new joke from the API. Use the **setInterval** function for this: https://www.w3schools.com/jsref/met_win_setinterval.asp

Check the API documentation to figure out how to load a single joke from the "programming" category.

However, when we unmount the **FeaturedJoke** component (using the button discussed earlier) the interval should be **cleared** so that the jokes stop loading. We don't want the application to continue loading new jokes when we can't even see them! To do this, we need to "clean up" after the **FeaturedJoke** component when it unmounts. You can do this by adding a **return** to the effect hook function that sets up the interval. Have the effect hook function return ***another function***. This function will be run automatically when the component unmounts. In this function, **clear the interval**. https://www.w3schools.com/jsref/met_win_clearinterval.asp

Here's a reference on cleaning up after unmount: https://reactjs.org/docs/hooks-effect.html#example-using-hooks-1

One more thing: make sure the **FeaturedJoke** component uses the **Joke** component that you created earlier. We want to reuse code as much as possible.

Task 5: Adding Jokes to a Playlist

Now we'll get back to the playlist idea. Each **Joke** component displayed in the application (within **RandomJokes** or **FeaturedJoke**) should have a **form** that allows one to choose an existing playlist from a dropdown and add the joke to that playlist. The **Playlists** component should then re-render itself to show the newly-added joke in the appropriate playlist.

First, within the **Joke** component, go ahead and implement a controlled form that allows one to choose a playlist from a **select** input, using a **<select>** tag. (It looks like a dropdown list.) Note that you'll have to pass an array of playlists down to each **Joke** component from wherever its being stored in the state. (Probably in **App,** since we established earlier that the siblings **CreatePlaylist** and **Playlists** will both need access to it.) Don't worry about what happens when the form is submitted just yet.

When implementing the select list, use the following code:

```
<select onChange={handleChangePlaylist} defaultValue='default'>
        <option disabled value='default'> -- choose a playlist --</option>
        {/* TODO: other <option> elements rendered here */}
</select>
```

Basically, the above code will give the select list a default value.

Then, add a submit button that submits the joke to the chosen playlist. Of course, this means you'll have to switch back to the server side of things and modify your API to handle requests to update a given playlist by adding a new joke. Requests to add a joke to a playlist should be handled using a **PATCH** request. (Patch requests are used for updates that modify a resource without replacing it.) The endpoint should be **'/playlists/:listName',** where listName is the name of the playlist you're adding a joke to.

Normally, the route handler for this endpoint would accept a set of instructions for *how* to patch the document. However, to keep things simple, let's assume that all patch requests to this endpoint are always meant to add a new joke to the list. (There's no way to update the name of the playlist or anything like that.)

So, in short, submitting the form for adding a joke to a playlist should do the following:

- Submit a **patch** request to the API to add a joke to a specified list;
- Reload all playlists from the API;
- Propagate the new playlists to the **Joke** components and **Playlists** components.

Task 5: Reorganizing the Code

Back on the front end, you can see that there's a complication with this method of adding jokes to lists: each **Joke** component now needs to receive, as a prop, a full array of all playlists from **App**. (for the select list.) Similarly, the event handler for submitting a new joke to the API is going to have to be received as a prop from **App**. There's nothing wrong with this in principle, but it means we'll have to pass props down through multiple levels of hierarchy to the components at the bottom that need them. In this case, you'll have to pass props from **App** down to **RandomJokes**, and then from **RandomJokes** down to **Joke**. This is unfortunate because **RandomJokes** doesn't need these props and won't use them at all; it's simply the intermediary between **App** and **Joke**. This is a common problem in React, sometimes called "prop drilling."

One way to handle this would be to use a React feature called **Context** to give every component in the hierarchy access to the same state without having to pass props, but React recommends against doing this except in special cases. Instead, we're going to use **render props** to package up a bunch of things into a single prop, thereby making it easier to pass down. This will also help us reuse the same code in **RandomJokes** and **FeaturedJoke**. A render prop is basically a prop that is used for rendering. (as the name implies!) Note that there are other strategies for

mitigating the unaesthetic effects of prop drilling, such component composition: https://reactjs.org/docs/composition-vs-inheritance.html

In the **App** component, add the following function:

```
const renderJoke = joke=><Joke joke={joke} playlists={playlists}
handleAddToPlaylist={handleAddToPlaylist}/>
```

A brief explanation of the above code:

- **joke** is a joke object obtained from the Joke API. (note that this is distinct from **Joke**, which is the component used to render the setup, punchline, and add-to-playlist widget.)
- **playlists** is a state variable containing an array of all playlists that have been created so far. If you called this variable something different, make sure you change the name in your code.
- **handleAddToPlaylist** is an event handler function used to add a chosen joke to a chosen playlist.

As you can see, the function **renderJoke** takes a **joke** object (from the API) as a parameter, and returns a **Joke** component. **Figure out how to use this function to render Joke components from withing RandomJokes and FeaturedJoke.**

This pattern may seem a bit abstract at first, and you may have to experiment a bit to get it to work.

Note: you may have noticed that the **Playlists** component also renders jokes, but hasn't been included in the discussion about reusing code. I actually haven't yet thought of way for **Playlists** to share code with **RandomJokes** and **FeaturedJoke**, but I believe it's a good idea, and I encourage you to figure out a way to do this.

Hand In

When you're done, delete the node_modules folder, make sure the project folder is named **firstname-lastname**, zip the folder, and hand it in to Brightspace.

Checklist

- **[2 marks]** RandomJokes component is fetching and rendering 6 random jokes (using a Joke component for each)
- **[2 marks]** CreatePlaylist component for creating a new playlist
- **[1 mark]** Playlists component is updated each time a new playlist is created
- **[1 mark]** Your API updated to accept POST requests to create a new playlist
- **[1 mark]** FeaturedJoke component that can be toggled on or off
- **[1 mark]** FeaturedJoke component loads a new joke every 8 seconds
- **[1 mark]** Interval is cleared when FeaturedJoke is unmounted
- **[1 mark]** Joke component updated to include a form for adding jokes to playlists;
- **[1 mark]** Your API updated to handle PATCH requests to add a joke to a given playlist
- **[1 mark]** Playlists component should be updated when new jokes are added
- **[1 mark]** Render prop used to render Joke components

Total: 13

As usual, up to 25% may be deducted for improper hand in, poor organization, etc. Please ask me if in doubt.