CMPS 356 – Fall 2022
Web Applications Design and Development

Lab 05
React Hooks and Routing

Objective

1. Using outlet contexts (`useOutletContext`)
2. Navigating programmatically (`useNavigate`)
3. Setting protected routes
4. Using nested routes with multiple layouts
5. Reusing values and functions using memoization (`useMemo` and `useCallback`)
6. Prioritizing updates using transitions and deferred values (`useTransition` and `useDeferredValue`)

Prerequisites

1. React Router tutorial: https://reactrouter.com/en/main/getting-started/tutorial
2. React hooks API reference: https://reactjs.org/docs/hooks-reference.html
3. Debouncing and throttling: https://css-tricks.com/debouncing-throttling-explained-examples

1. Outlet Contexts and Programmatic Navigation

1. Create a new directory `01-routing-navigation` and copy the code of the last exercise from the previous lab under it.
2. Fetch the list of country facts and store it in the `App` component:

```
const App = () => {
  const [facts, setFacts] = useState(null);
  useEffect(() => {
    fetchFacts()
    ...
  }, []);
```

3. Rename your `Gallery` page to `Facts` and display the list of countries in a dropdown list instead of using multiple links.
4. Navigate to the corresponding country and display its facts when the selection changes:

```
const navigate = useNavigate();
```

5. Add an extra default disabled option for the index page when no country is selected:

```
<option value="-" disabled>-</option>
```

6. Use an outlet context to pass the list of country facts to the `Country` component to avoid the redundant requests used to fetch the facts for each country.
7. Navigate to the `Facts` page as fallback when the country provided is not part of the list. Use the path parameter to check whether the country code is valid or not, i.e., whether

the country code is part of the list of facts or not, then update the dropdown list selection accordingly.

8. How can the country selection be enhanced to work better on a mobile device?

2. Protected Routes and Multiple Layouts

1. Create a new directory `02-protected-routes` and reuse the code of the previous exercise.
2. Add a `Login` page that allows the user to authenticate by providing their name and email address. Store the user state and move your router and routes to the `App` component.
3. Use a different layout for the `Login` page that does not include a header nor a footer. This requires using nested routes and grouping your routes such that each group uses a different layout. Create a `Basic` component for the new layout.

```
<Route element={<Layout />}>
  ...
</Route>
<Route element={<Basic />}>
  ...
</Route>
```

4. Display the name and email in the header along with a button to logout.
5. Create a new `Photos` page that is only accessible after the user has authenticated. This requires a new route entry and component named `ProtectedRoute` to handle such protected pages. Update your navigation bar to link to this new page.
6. Create a protected `Profile` page that allows the user to update their name or email address. Add the corresponding route entry and navigation bar link.
7. Update your logout button such that it changes to a login button if the user is not authenticated and then allow the user to manually authenticate by clicking it.
8. Hide the protected links from the navigation bar when the user is not authenticated.

3. Memoization, Deferral, and Prioritization

1. Create a new directory `03-memoization-deferral` and an application under it.
2. Add a textbox to your `App` component and bind it to a state variable, `lorem`.
3. Create a `Playground` component with two textboxes. One textbox is bound to a state variable, `ipsum`, and the second textbox is read-only. `Playground` also receives `lorem` as a property through `value`. Add an instance of `Playground` to your application and pass `lorem` from `App` to it.

```
<Playground value={lorem} />
```

4. The content of the second textbox is the result of a function that returns the property `value` after actively waiting a certain duration in milliseconds. Use the following function to simulate a time-consuming computation:

```
const wait = (value, duration) => {
  const date = Date.now();
  while (Date.now() - date < duration);
  return value;
```

```
};
```

5. Try changing the text input, bound to `ipsum`, quickly and observe the responsiveness of the overall user interface when dealing with increasingly longer waiting duration for updating the content of the textbox using `wait()` based on the `value` property.
6. Use a hook to memoize the return value of `wait(value, duration)`, since it is independent of `ipsum`. This memoized value will now be reused whenever the component is rendered.
7. Instead of using the value of `value` directly in `Playground`, wrap it in a function, `fvalue`, that returns it instead. What do you observe? Memoize `fvalue` as a callback so that it is created only once.

```
const fvalue = () => value;
```

8. Add a slider that, when changed, returns its value through a function after busy waiting a certain duration in milliseconds. The value of the slider is bound to a state variable, `slider`, and the returned value is displayed in a read-only textbox and bound to a state variable, `dolor`.
9. Try moving the slider quickly. How responsive is the experience with increasingly longer waiting durations for setting `dolor` using `wait(slider, duration)`?
10. Use a hook to defer the value of `slider`. Is that sufficient to enhance the overall feel of the experience?
11. How can the user be kept updated about whether the value of `dolor` still being computed? Use a transition hook to provide a more informative experience.
12. Add another read-only textbox bound to a state variable, `amit`, that is updated to the value of `slider` asynchronously after a certain duration in milliseconds; we are trying to simulate an asynchronous request. Use the following function to simulate a request that resolves with a given value after a certain given duration:

```
const sleep = (value, duration) => {
  return new Promise((resolve) =>
    setTimeout(() => resolve(value), Math.random() * duration)
  );
};
```

13. Try moving the slider quickly. How responsive and linear is the experience with increasingly longer waiting durations for setting `amit` using `sleep(slider, duration)` as a side effect?
14. How can the user be kept updated about the status of the simulated request, i.e., the request that is initiated to set the value of `amit`? How to abort a pending request when a new one is generated? How can we make sure to have the latest value of slider reflected in the textbox content?

4. Highlights with Transitions

1. Create a new directory `04-highlight-transition` and an application under it.
2. Install the `Faker` package "`npm install @faker-js/faker`" and use it to generate 10,000 random names following its guide: https://fakerjs.dev/guide.

```
const records = Array.from(Array(10000), () => faker.name.fullName());
```

3. Create a `List` component that receives the names as a property and provides an input field that can be used as a query to filter the list of names. The text that matches with the search query should be highlighted in every name.

   ```
   const [query, setQuery] = useState("");
   ```

4. How responsive is the input when updating the query text? Create a state variable, highlight, to hold a copy of the query text and wrap the call to update its value, whenever the query is updated, in a transition callback. Display visual feedback when the update is still pending. You can use a spinner from the React Loader Spinner package: https://mhnpd.github.io/react-loader-spinner.

   ```
   const [highlight, setHighlight] = useState("");
   const [isPending, startTransition] = useTransition();

   const updateQuery = ({ target: { value } }) => {
     setQuery(value);
     startTransition(() => setHighlight(value));
   };
   ```