CMPS 356 – Fall 2022
Web Applications Design and Development

Lab 06
React Suspense and Next Basics

Objective

1. Setting up and using React Query
2. Displaying fallback components using React Suspense
3. Handling errors using error boundaries
4. Static-site generation (SSG) using Next:
   4.1. Generating static properties and page content
   4.2. Generating static paths from dynamic routes

Prerequisites

1. React Query: https://tanstack.com/query/v4.
2. React Query with Suspense: https://tanstack.com/query/v4/docs/guides/suspense.
3. Next Tutorial: https://nextjs.org/learn/foundations/about-nextjs.

1. Suspense with React Query and Error Boundary (Experimental)

1. Create a new directory `01-suspense` and a React application under it.
2. Install React Query using "`npm install @tanstack/react-query`".
3. Create a query client and provide it in your `App` component:

```
import {
  QueryClient, QueryClientProvider
} from "@tanstack/react-query";

const queryClient = new QueryClient();

const App = () => {
  return (
    <div className="App">
      <QueryClientProvider client={queryClient}>
        ...
      </QueryClientProvider>
    </div>
  );
};
```

4. Create a `Facts` component that uses the query client to fetch country facts from https://restcountries.com/v3.1/all:

```
const fetchFacts = async () => {
  const res = await fetch("https://restcountries.com/v3.1/all");
  if (res.ok) {
    return await res.json();
  } else {
```

```
    throw new Error("Fetching failed!");
  }
};

const Facts = () => {
  const query = useQuery(["facts"], fetchFacts);
  ...
```

5. Use `query.isLoading`, `query.isError`, and `query.data` to conditionally render the `Facts` component and display the fetch request status and data.
6. Display a spinner when the content is loading. You can install and use the React Loader Spinner package: "`npm install react-loader-spinner`", for example:

```
import { InfinitySpin } from "react-loader-spinner";
const Spinner = () => <InfinitySpin width="200" color="#777" />;
```

7. Suspend your `Facts` component by wrapping it in a `Suspense` component and providing a spinner as a fallback to display when the content is loading:

```
<QueryClientProvider client={queryClient}>
  <Suspense fallback={<Spinner />}>
    <Facts />
  </Suspense>
</QueryClientProvider>
```

8. Update your query client configuration to work with Suspense:

```
const query = useQuery(["facts"], fetchFacts, {
  retry: false,
  suspense: true,
});
```

9. Error boundaries work like a JavaScript `catch {}` block, but for components. Wrap your `Suspense` component in an `ErrorBoundary` component to handle the errors that can occur and display their corresponding messages as fallback: "`npm install react-error-boundary`":

```
<ErrorBoundary
  fallbackRender={({ error }) => <div>Error: {error.message}</div>}
>
  <Suspense ...
</ErrorBoundary>
```

   Errors from React Query can be reset using the `QueryErrorResetBoundary` component or the `useQueryErrorResetBoundary()` hook.

2. Static-Site Generation (SSG) and Dynamic Routing

1. Create a new directory `02-next` and a Next application under it using: "`npx create-next-app . --use-npm`".
2. Run your application in development mode: "`npm run dev`". You can disable telemetry using: "`npx next telemetry disable`".
3. Recreate the country facts application from the previous lab using Next.
4. Define a main layout, `Layout`, and an associated CSS module, `Layout.module.css`.

5. Modify the index page to load the countries as a static property, `facts`, and use that property to display the list of countries in a dropdown list:

```
export async function getStaticProps() {
  const res = await fetch("https://restcountries.com/v3.1/all");
  let facts = null;
  if (res.ok) {
    facts = await res.json();
    facts = facts.map((a) => ({
      name: {
        common: a.name.common,
      },
      cca2: a.cca2.toLowerCase(),
    }));
    facts.sort((a, b) => (a.name.common > b.name.common ? 1 : -1));
  } else {
    throw new Error("Fetching facts failed!");
  }

  return {
    props: {
      facts,
    },
  };
}
```

6. Create a (dynamic) page, `[cca2].js`, with one path parameter, `cca2`, then generate the list of valid paths, `paths`, based on the list of CCA2 codes:

```
export async function getStaticPaths() {
  const res = await fetch("https://restcountries.com/v3.1/all");
  let facts = null;
  if (res.ok) {
    facts = await res.json();
    facts = facts.map((a) => ({
      cca2: a.cca2.toLowerCase(),
    }));
    facts.sort((a, b) => (a.cca2 > b.cca2 ? 1 : -1));
  } else {
    throw new Error("Fetching facts failed!");
  }

  const paths = facts.map((country) => ({
    params: { cca2: country.cca2 },
  }));

  return { paths, fallback: false };
}
```

7. Generate a static property, `country`, to load the corresponding country facts based on the `cca2` path parameter. This property will be used to display the facts for a given country:

```
export async function getStaticProps({ params }) {
  const res = await fetch(
```

```
      `https://restcountries.com/v3.1/alpha/${params.cca2}`
    );
    let country = null;
    if (res.ok) {
      country = await res.json();
      country = country[0];
    } else {
      throw new Error("Fetching country facts failed!");
    }

    return {
      props: {
        country,
      },
    };
  }
```

8. Update your index page and use Next's router to navigate to the corresponding country page when the selection is changed:

```
import { useRouter } from "next/router";

export default function Facts({ facts }) {
  const router = useRouter();
  ...
            <select
              value={"-"}
              onChange={(e) => {
                router.push(e.target.value);
              }}
            >
```

9. Modify your application to fetch the facts only once instead of multiple times for each country.
10. Test your statically generated application by building it then serving it: "`npm run build`" then "`npm run start`".