CMPS 356 – Fall 2022
Web Applications Design and Development

Lab 07
Rendering with Next

Objective

1. Using Next's SWR hook for data fetching
2. Rendering using client-side rendering (CSR)
3. Rendering using server-side rendering (SSR) with `getServerSideProps`
4. Rendering using static-site generation (SSG) with `getStaticProps` and `getStaticPaths`
5. Rendering using incremental-site regeneration (ISR)
6. Installing and using Material UI (MUI)

Prerequisites

1. Next tutorial: https://nextjs.org/learn/foundations/about-nextjs
2. Data fetching overview: https://nextjs.org/docs/basic-features/data-fetching
3. SWR hook: https://swr.vercel.app
4. MUI tutorial: https://mui.com/material-ui/getting-started/overview

1. Client-side Rendering (CSR)

1. Create a new directory `01-rendering` and a Next application under it.
2. Install MUI: "`npm install @mui/material @mui/icons-material @emotion/react @emotion/styled @emotion/server`".
3. Create a new `csr` directory and an `index.js` page under it, then use SWR to fetch and render the list of all pokémons from PokéAPI (https://pokeapi.co/docs/v2).
4. Use alerts to display the status of your request.

```
if (error) return <Alert severity="error">{error.message}</Alert>;
if (!data) return <Alert severity="info">Loading...</Alert>;
```

5. Use a card to display the name (clickable) of each pokémon. The cards should be organized using a responsive grid.

```
<Grid key={pokemon.name} item xs={6} sm={3}>
  <Link href={`/csr/pokemons/${pokemon.name}`}>
    <a>
      <Card>
        <CardContent>
          <Typography
            sx={{ fontSize: 14 }}
            color="text.secondary"
            gutterBottom
          >
            {pokemon.name}
          </Typography>
```

```
        </CardContent>
      </Card>
    </a>
  </Link>
</Grid>
```

6. Create a new `pokemons` directory under `csr` and a `[name].js` page under `pokemons`, then use SWR to fetch and render a selection of traits for each pokémon.

```
const traits = [
  { code: "id", name: "ID" },
  { code: "name", name: "Name" },
  { code: "base_experience", name: "Base Experience" },
  { code: "height", name: "Height" },
  { code: "order", name: "Order" },
  { code: "weight", name: "Weight" },
];
```

7. Use a list to display a pokémon's traits.

```
<ListItem key={trait.code}>
  <ListItemAvatar>
    <Avatar>
      ...
    </Avatar>
  </ListItemAvatar>
  <ListItemText primary={trait.name} secondary={data[trait.code]} />
</ListItem>
```

8. Update your implementation to use Suspense in your index page.

```
const { data, error } = useSWR(
  "https://pokeapi.co/api/v2/pokemon?limit=1154",
  fetcher,
  { suspense: true }
);
```

2. Server-side Rendering (SSR)

1. Create a new `ssr` directory and an `index.js` page under it, then use `getServerSideProps` to fetch and render the list of all pokémons from PokéAPI.

```
export async function getServerSideProps(context) {
  const data = await fetcher(
    "https://pokeapi.co/api/v2/pokemon?limit=1154"
  );
  return { props: { data } };
}
```

2. Create a new `pokemons` directory under `ssr` and a `[name].js` page under `pokemons`, then use `getServerSideProps` to fetch and render a selection of traits for each pokémon.

```
export async function getServerSideProps(context) {
  const { name } = context.params;
  let data = await fetcher(
    `https://pokeapi.co/api/v2/pokemon/${name}`
  );
```

```
  data = traits.reduce(
    (acc, field) => ({
      ...acc,
      [trait.code]: data[trait.code],
    }),
    {}
  );

  return { props: { data } };
}
```

3. Create a navigation bar that provides links to the two index pages, `csr` and `ssr`, that you have created so far. Use a common layout for your index pages.

3. Static-site Generation (SSG)

1. Create a new `ssg` directory and an `index.js` page under it, then use `getStaticProps` to fetch and render the list of pokémons from PokéAPI.
2. Create a new `pokemons` directory under `ssr` and a `[name].js` page under `pokemons`, then use `getStaticProps` and `getStaticPaths` to fetch and render a selection of traits for each pokémon.

```
export async function getStaticPaths() {
  const data = await fetcher(
    "https://pokeapi.co/api/v2/pokemon?limit=1154"
  );
  return {
    paths: data.results.map((pokemon) => ({
      params: { name: pokemon.name }
    })),
    fallback: false,
  };
}
```

3. Update your navigation bar with a link to the newly created page.

4. Incremental Static Regeneration (ISR)

1. Create a new `isr` directory and an `index.js` page under it, then regenerate your pages automatically at regular intervals. This can be achieved by adding a field "`revalidate: 60`" to update your content, at most once, every minute, for example. To test this behavior, use a subset of your data.

```
const data = await fetcher(
  "https://pokeapi.co/api/v2/pokemon?limit=1154"
);
const results = [...data.results]
  .sort(() => 0.5 - Math.random())
  .slice(0, 24);
return { props: { data: { ...data, results } }, revalidate: 60 };
```

2. Update your navigation bar with a link to the newly created page.
3. Regenerate your index page on-demand using an API endpoint: "`/api/revalidate`. Use a secret token stored in an environment variable to protect the API endpoint: create an `.env.local` file with your `TOKEN` and use `process.env.TOKEN` to access your token.

4. How can we use this endpoint to revalidate any given page in our application?