React Router 6 Features

What are we going over today?

- Challenges with FE architecture
- React Router 6 feature review
- Loaders
  - Examples
  - Conventions
- Actions
  - Examples
  - Conventions
- Async routes
  - Examples
  - Conventions
- Migration
- Structure
- Questions

Challenges with our FE architecture

Data loading... need I say more?

- Currently data loading is done both from pages and components
- API access is done with an in house pattern and is showing its cracks
- This pattern introduces a ton of boilerplate
- This proposal does not include the library we'll use for data access

Intro to React Router 6

Mostly review

- Responsible for rendering components depending on the route
- Dynamic routes
- Consolidated error handling
- Utilities for reading and writing to the URL and history
- New strategies for loading and submitting data
- New strategies for pending / optimistic UI's
- Focused on web standards

Loaders

A new way to fetch data!

- Loaders are about *when* to fetch data
- A loader is a function attached to a route
- A loader is executed when the route is navigated to, an action runs, or the URL (path or query) of the current route changes
- A loader has access to the route params and `request` object
- By default, loaders return *before* the component is rendered
- Loaders can return any type of data, including Responses and Promises.
- Errors thrown in loaders bubble up to the nearest `errorElement` defined in the router
- We can prevent a loader from re-running using the `shouldRevalidate` prop
- We will have conventions to follow for loaders

Examples

- Defining loaders
- Accessing data
- Creating routes
- Catching errors
- Pending loaders
- Loader nuances

Defining a loader

Loaders are how we fetch data for a route. We'll take a look at a bare-bones loader. It uses a global GraphQL client to make a query and return the result. It does not include error handling or data transformation.

```
const loader = async ({ params }) => {
  return await client.query(readEngagementSettings, {
    engagementId: params.engagementId,
  });
};
```

Defining a loader

- The barebones example lacked
  - Error handling
  - Data transformation
- We want to follow best practices with our loaders
  - That includes passing dependencies to them instead of relying on globals

A note on fetch

- If using `fetch`, React Router can infer when we return a `Response` and will automatically call `.json()` for us
    - It's unlikely we'll be using the fetch API directly
    - If we do, prefer calling `.json()` directly to be more explicit

A better loader

Let's take a look at a better loader. This one will be passed dependencies and will perform error handling and transformation.

```javascript
const settingsLoaderFactory = ({ client }) => {
  return async function settingsLoader({ params }) {
    const { accountId, engagementId } = params;

    // If the accountId and engagementId params aren't found throw a 404. This
    // could happen if the loader is moved to a route without those params.
    if (!accountId || !engagementId) {
      throw new Response("Not Found", { status: 404 });
    }

    // Get engagement settings
    const result = await client.query(readEngagementSettings, {
      engagementId,
      accountId,
    });

    // If an unexpected error is encountered throw a 500.
    if (result.error) {
      throw new Response(result.error.message, { status: 500 });
    }

    // If no engagement was found throw a 404.
    if (!result.data || !result.data.engagement) {
      throw new Response("Not Found", { status: 404 });
    }

    return {
      // Return the engagement data
      engagement: result.data.engagement,
    };
  };
};
```

Accessing data

To access data from a loader we use the `useLoaderData` hook. Remember that a loader has finished execution *before* this route has rendered. So the data is guranteed to be there.

```
function Settings() {
  const { engagement } = useLoaderData();

  return (
    // ... settings rendering
  );
}
```

Accessing data

- `useLoaderData` returns an object with a type of `unknown`.
- We help the compiler by defining a return type for our loader and using type casting where we call the hook.

```
type SettingsLoaderData = {
  engagement: Engagement;
};

const settingsLoaderFactory = ({ client }) => {
  return async function settingsLoader({
    params,
  }): Promise<SettingsLoaderData> {};
};

function Settings() {
  const { engagement } = useLoaderData() as SettingsLoaderData;
}
```

Creating a route

Creating a route is the same, with the addition of a `loader` prop. However, now that our loaders are passed their dependencies we need to ensure those dependencies are passed to the router.

```
// Routes becomes a function
function routes({ client }) {
  return createBrowserRouter(
    createRoutesFromElements(
      <Route index element={Root}>
        <Route
          path=":engagementId"
          element={Settings}
          {/*
            Our factory is called with dependencies and we get an
            initialized loader
          */}
          loader={settingsLoaderFactory({ client })}
        />
      </Route>
    )
  );
}
```

Creating a route

- One important thing piece lacking is an `errorElement` to catch failures in loaders, actions, or renders.

Catching errors

Errors happen. They may be errors we throw, or they may come from outside our control. In either case we need to handle them. To catch these errors, we define an `errorElement` on our routes.

```jsx
function ErrorPage() {
  // Get access to the thrown error
  const error = useRouteError();

  switch (error.status) {
    case 404:
      return <NotFound />;
    case 401:
      return <NotAuthorized />;
    case 500:
      return <InternalError message={error.message} />;
    default:
      return <InternalError />;
  }
}

<Route index element={Root} errorElement={ErrorPage}>
  <Route
    path=":engagementId"
    element={Settings}
    loader={settingsLoaderFactory({ client })}
  />
</Route>;
```

Catching errors

- Errors bubble up. We can define granular error handling at the individual route level, or define generic error handling higher in the tree.

Pending loaders

Sometimes we need to render UI before our network calls have finished. In that case, we can use the `deferred` utility to return pending promises from a loader, and the `Await` component to display fallback UI as those promises resolve.

```tsx
type SlowLoaderResult = {
  data: Promise<any>;
};

const slowLoaderFactory = ({ client }) => {
  return async function slowLoader(): Promise<SlowLoaderResult> {
    // Notice no await
    const longRequest = client.query(longRunningQuery);

    return deferred({
      data: longRequest,
    });
  };
};

function MyPage() {
  // We access the data exactly the same. In this case, data is a pending
  // pending promise that will resolve with a response from our API.
  const { data } = useLoaderData() as SlowLoaderResult;

  return (
    <React.Suspense fallback={<p>Loading...</p>}>
      <Await
        resolve={data}
        {/*
          We can provide an error element at this level, to granularly handle
          a section of the page not getting its data. If we don't provide an
          errorElement here and the promise rejects the error will be bubbled
          to the next errorElement in the router tree.
        */}
        errorElement={<ErrorPage />}
        {/*
          I'm omitting it here, but we still want to follow best practices and
          map the data from the result before passing it to a component.
        */}
        children={(data) => <SomeComponent {...data} />}
      />
    </React.Suspense>
  );
}
```

Loader nuances

To review some of the nuances

- Loaders are called from navigation events, actions, or the URL query changing
- By default loaders return data before the route is rendered
- We can prevent loaders from re-running using the `shouldRevalidate` prop
- Loaders do not have access to react hooks

Conventions

- Define loaders in their own files.
- Prefer returning resolved promises from loaders, use `deferred` when performance is impacting user experience.
- Leverage the factory pattern to pass dependencies from the app, to the router, and into loaders.
- Transform your responses, don't return GraphQL implementation details to pages.
- Collapse multiple queries into one.
- Keep react router implementation details in pages and out of components

Actions

The data submission pair to loaders!

- Many of the same patterns and conventions we talked about still apply
- Actions are triggered for any non GET request to the route
- Most typically, actions are triggered from form submissions, or using a hook
- Actions have access to the `request` that triggered them
- Actions use `FormData` by default
- Once an action has executed, the loader for the route is re-executed and fresh data is available from the `useLoaderData` hook. This allows us to submit data to the API, retrieve fresh data, and display that data without reloading the page
- We can prevent a loader from re-running by leveraging `shouldRevalidate` on the route and getting action data from the `useActionData` or `useFetcher` hook

Examples

- Defining actions
- Accessing data
- Submitting data

Defining actions

Actions are how we accept data from the UI and handle it asynchronously. We'll take a look at a bare-action loader. It uses a global GraphQL client to execute a mutation and return the result. It does not include error handling or data transformation. It calls a single mutation.

```
const settingsAction = async ({ request }) => {
  const engagement = await request.formData();
  await client.mutation(updateEngagement, engagement);
  return;
};
```

Defining actions

- The barebones example lacked
    - Error handling
    - Data transformation
    - The ability to run more than one mutation
- We want to follow best practices with our actions
    - That includes passing dependencies to them instead of relying on globals

A better action

Let's take a look at a better action. This one will be passed dependencies and will perform error handling and transformation. We'll also use the convention of passing `intent` when we submit the form, to allow us to select which mutation to perform on the backend.

```
const settingsActionFactory = ({ client }) => {
  return async function settingsAction({ request }) {
    const formData = await request.formData();
    const intent = formData.get("intent");

    switch (intent) {
      case "update-engagement-destinations":
        const result = await client.mutation(
          updateEngagementDestinations,
          destinations: formData
        );
        // If an unexpected error is encountered throw a 500.
        if (result.error) {
          throw new Response(result.error.message, { status: 500 });
        }
        return {
          intent: "update-engagement-destinations",
          destinations: result.data.destinations,
        };
      default:
        break;
    }
  };
};
```

Accessing data

To access data from an action we can use the `useActionData` hook. However, the need for this is rare. In most cases we should rely on the loader being re-executed and fresh data made available through the `useLoaderData` hook.

```
function Settings() {
  const actionData = useActionData();
  const { intent } = actionData;
  if (intent === 'update-engagement-destinations') {
    const { destinations } = actionData;
  }


  return (
    // ... settings rendering
  );
}
```

Submitting data

To submit data we can rely on standard HTML forms, or we can trigger submissions imperatively. For the most part we'll be doing the later. Imperative form submission. We have two hooks we can use.

`useSubmit` returns a function that can be used to call the action for the current route. Data can be passed into the function and will be encoded into `FormData`.

`useFetcher` returns an object with several properties including a `.load()` method for triggering a loader, a `.submit()` method for triggering an action, and a `.state` property that tells us if the action is `submitting` or the loader is `loading`. We can use these properties to disable parts of the UI while we wait for results.

```
function Settings() {
  // Full page data from initial load. We used `shouldRevalidate={false}` on the
  // route to prevent this loader from re-running.
  const { engagement } = useLoaderData();

  // Submit functions for forms
  const {
    submit,
    data,
    state: { submitting },
  } = useFetcher();

  // Place our destinations in state, using the original loader data to
  // initialize the state.
  const [destinations, setDestinations] = useState(engagement.destinations);

  // Run an effect after any form submission.
  useEffect(() => {
    // If the form is no longer submitting.
    if (!submitting) {
      // Look at the intent returned (which mutation did we call).
      switch (data.intent) {
        case "update-engagement-destinations":
          // Update the destinations state with the data returned from the API.
          const { destinations } = data;
          setDestinations(destinations);
          break;
      }
    }
  }, [data, submitting]);

  // Create a handler that is called when the destinations form is valid and
  // submitted.
  const onSubmitDestinations = (data) =>
    submit(
      { intent: "update-engagement-destinations", ...data },
      { method: "post" }
    );

  // Pass simple props and callbacks to DestinationsForm, keeping it completely
  // unaware of React Router.
  return (
    <>
      <DestinationsForm
        onSubmit={onSubmitDestinations}
        disabled={submitting || loading}
        destinations={destinations}
      />
    </>
  );
}
```

Submitting data

- We want to keep react router implementation details out of our components. Our `DestinationsForm` takes a callback to execute when it's submitted. Keeping our action logic in the page.
- Using the `useFetcher` hook we can call actions and loaders ad-hoc, and inspect the state of both. This allows us to build more complex, reactive UI's such as the Engagement Settings page.
- Using `shouldRevalidate` we can prevent the loader from re-running, and instead fetch only the data that we submitted to the server. This can be especially useful for large pages with large queries and multiple mutations.

Conventions

- Define actions in their own file
- Keep react router implementation details out of components
- Leverage callbacks to take data from forms and make it available to pages
- Consider defining these callbacks in their own handlers file
- Use react-hook-form for validating forms and prevent a callback from being executed unless the form is valid
- For pages that require multiple mutations, leverage an `intent` field to inform the action which mutation needs to be called
- Prefer the regular request response lifecycle by triggering an action and letting the loader run in response
- Step outside of the default request response lifecycle only if neccesary
- Optimizations like `shouldRevalidate` should always be opt-in and should rarely be a first step

Async Routes

Loading pages on demand!

- Async routes let us defer loading a JS bundle until that route is hit
- This functionality helps performance (initial loads) especially on weaker networks
- The host downloads a single module from every remote on init
- This module contains the route definitions for the remote
- Each route is configured to lazily load its page, so loading the bundle is deferred until the user navigates to the route
- We can optimize further by statically including the loader and action for each route
- This executes the loader or action for a route while fetching the page bundle

Examples

- Remote route configuration
- Host route configuration

Remote route configuration

```jsx
function routes({ client }) {
  return (
    <>
      <Route
        index
        // Our loader has been statically imported at the top of the file so
        // react router executes it as soon as this route is visited.
        loader={listLoaderFactory({ client })}
        // Our page is dynamically imported when the route is visited.
        lazy={async () => {
          // This syntax - await import("foo") - tells bundlers like webpack
          // and rollup that we want to use code splitting and split List out
          // into a separate bundle.
          const { default: Component } = await import("./pages/List");
          // We return an object with a property called Component which is the
          // component we want rendered. In our case it's our page.
          return { Component };
        }}
      />
      <Route
        path=":engagementId"
        loader={settingsLoaderFactory({ client })}
        action={settingsActionFactory({ client })}
        lazy={async () => {
          const { default: Component } = await import("./pages/Settings");
          return { Component };
        }}
      />
    </>
  );
}
```

Host route configuration

```jsx
// Import a single module from each remote, the module is a function that takes
// in common dependencies and returns the routes for that remote. The routes
// lazy load their pages so the application bundle still stays small.
import engageRoutes from "remote_engagements/Routes";
import digitalTwinRoutes from "remote_digital_twins/Routes";
import analyticsRoutes from "remote_analytics/Routes";

function routes({ client }) {
  return (
    <Route path=":accountId">
      <Route path="engage">{engageRoutes({ client })}</Route>
      <Route path="digital-twins">{digitalTwinRoutes({ client })}</Route>
      <Route path="analytics">{analyticsRoutes({ client })}</Route>
    </Route>
  );
}
```

Conventions

- Prefer static loader and action imports, and lazy component imports
- Export your routes as a function from your remote
- If in doubt follow the conventions you see

Structure

How are we planning on structuring this functionality?

```
.
└── src
    ├── components                                    # Reusable components
    │   └── SearchBar                                 # Reusable search bar
    │       ├── index.ts
    │       ├── SearchBar.tsx                         # Component
    │       ├── SearchBar.stories.tsx                 # Stories
    │       └── SearchBar.spec.tsx                    # Unit Tests
    └── pages                                         # Pages corresponding to routes
        └── Settings
            ├── index.ts
            ├── Settings.tsx                          # Page
            ├── Settings.stories.tsx                  # Stories
            ├── Settings.spec.tsx                     # Unit tests
            ├── loader.ts                             # Loader for data access
            ├── loader.spec.ts                        # Loader unit tests
            ├── action.ts                             # Action for data submission
            ├── action.spec.ts                        # Action unit tests
            ├── handlers.ts                           # Event handlers passed to children
            ├── handlers.spec.ts                      # Handler unit tests
            └── DestinationsForm                      # Page components
                ├── index.ts
                ├── DestinationsForm.tsx              # Page component
                ├── DestinationsForm.stories.tsx      # Stories
                └── DestinationsForm.spec.tsx         # Unit tests
```

Migration

How do we migrate existing pages to these new patterns?

- We'll create a /v2 folder under pages, and a `/v2` route collection
- We'll select a page and gradually migrate its functionality and components to the new patterns
- This will include moving components that belong to a page, to a folder under that page
- During this process we will leave the legacy page alone
- Once we have completed the migration process we'll A/B test both pages to ensure we have no regressions
- After testing is complete we'll deprecate the old page and move the new one out of the `/v2` routes and folder

Questions / comments / feedback?