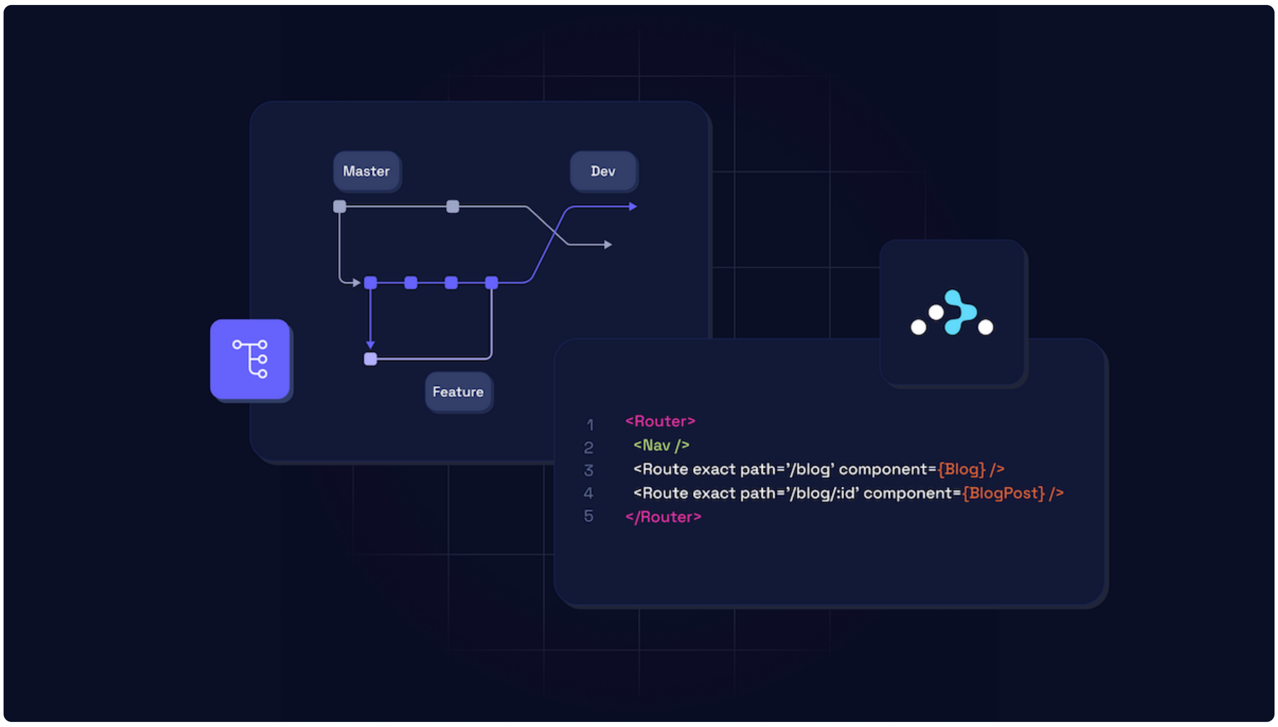
A Complete Guide to Routing in React

hygraph.com/blog/routing-in-react

Examples and Tutorials

In this guide, we will learn how to perform routing in React using React router, as well as the various aspects of routing and how React router handles them, such as dynamic routing, programmatic navigation, no matching route, and much more.

Joel Olawanle
Sep 6, 2022



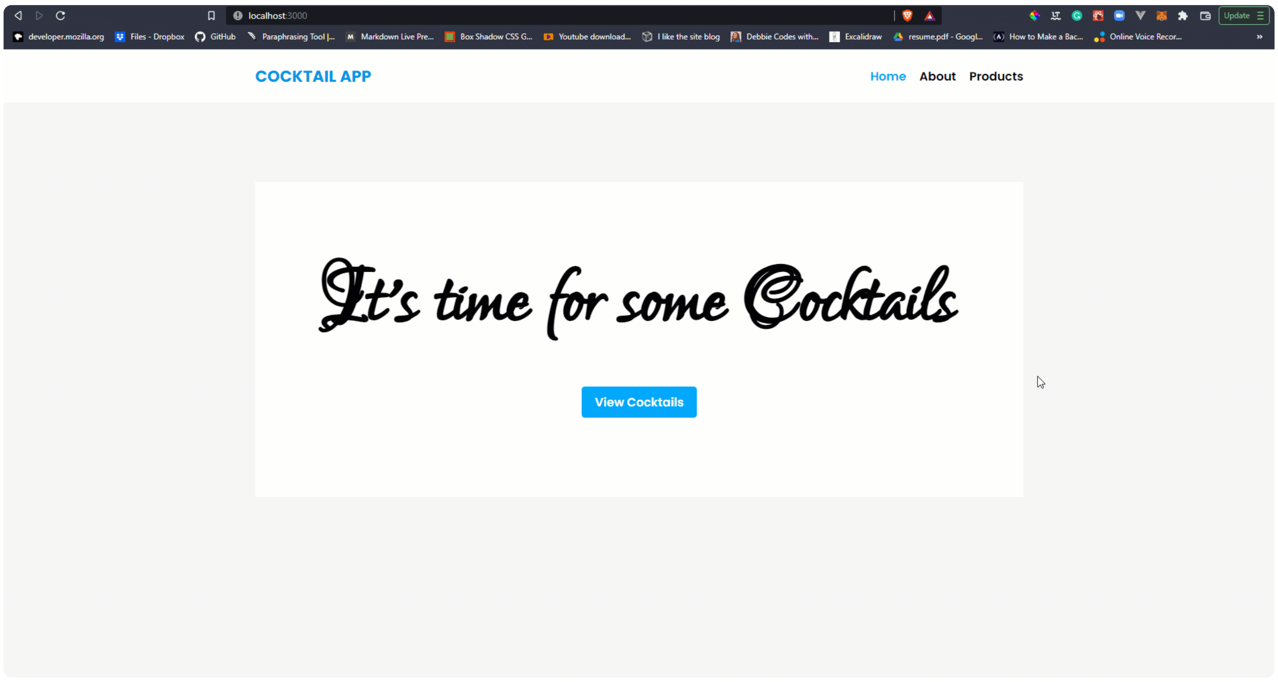
```
<Router>
  <Nav />
  <Route exact path='/blog' component={Blog} />
  <Route exact path='/blog/:id' component={BlogPost} />
</Router>
```

React is an open-source front-end JavaScript library that allows developers to create user interfaces using UI components and single-page applications. One of the most important features we always want to implement when developing these applications is routing.

Routing is the process of redirecting a user to different pages based on their action or request. In React routing, you'll be using an external library called React router, which can be difficult to configure if you don't understand how it works.

Getting Started

To fully comprehend and follow this guide, we would create an application that properly illustrates almost all aspects of navigation with appropriate use cases. We would create/use a cocktails app that retrieves her data from Hygraph via GraphQL. All aspects of routing covered in this guide are used in this application, which can be accessed via this live link.



Note: This guide only covers routing; however, the aspect of creating a schema on Hygraph and how we consumed the data will not be covered; notwithstanding, here is a link to the source code on GitHub, and I have also included a picture of what the schema looks like in this article.

Prerequisite

You should have the following to follow along with this guide and code:

- A fundamental understanding of HTML, CSS, and JavaScript
- Some experience or knowledge of React
- Node and npm or yarn installed on your machine

- A fundamental understanding of how the terminal works

How to Install React Router

As previously stated, React makes use of an external library to handle routing; however, before we can implement routing with that library, we must first install it in our project, which is accomplished by running the following command in your terminal (within your project directory):

```
npm install react-router-dom@6
```

After successfully installing the package, we can now set up and configure react-router within our project.

How to Setup React Router

To configure React router, navigate to the `index.js` file, which is the root file, and import `BrowserRouter` from the `react-router-dom` package that we installed, wrapping it around our App component as follows:

```
// index.js

import React from 'react';

import ReactDOM from 'react-dom/client';

import { BrowserRouter } from 'react-router-dom';

import App from './App';


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

 <React.StrictMode>

    <BrowserRouter>

       <App />

    </BrowserRouter>

 </React.StrictMode>

);
```

How to Configure Routes In React

We have now successfully installed and imported React router into our project; the next step is to use React router to implement routing. The first step is to configure all of our routes (that is, all the pages/components to which we want to navigate).

We would first create those components, in our case three pages: the Home page, the About Page, and the Products Page. This GitHub repository contains the content for these pages. Once those pages are properly configured, we can now set up and configure our routes in the `App.js` file, which serves as the foundation for our React application:

```
// App.js

import { Routes, Route } from 'react-router-dom';

import Home from './Pages/Home';

import About from './Pages/About';

import Products from './Pages/Products';


const App = () => {

 return (

    <>

       <Routes>

          <Route path="/" element={<Home />} />

          <Route path="/products" element={<Products />} />

          <Route path="/about" element={<About />} />

       </Routes>

    </>

 );

};


export default App;
```

We can see in the above code that we imported `Routes` and `Route` components from `react-router-dom` and then used them to declare the routes we want. All Routes are wrapped in the `Routes` tag, and these Routes have two major properties:

- `path` : As the name implies, this identifies the path we want users to take to reach the set component. When we set the `path` to `/about` , for example, when the user adds `/about` to the URL link, it navigates to that page.

- `element`: This contains the component that we want the set path to load. This is simple to understand, but remember to import any components we are using here, or else an error will occur.

  Note: We created a folder (Pages) to keep all page components separate from actual components.

  When we go to our browser and try to navigate via the URL, it will load whatever content we have on such pages.

How to Access Configured Routes with Links

We just saw how we could manually access these routes via the URL, but that shouldn't be the case for our users. Let's now look at how we can create links within our application so that users can easily navigate, such as a navbar.

This is accomplished with the Link tag, just as it is with the `<a>` tag in HTML. We made a NavBar component and placed it at the top of our routes in the App.js file so that it appears regardless of the route:

```
// App.js

import NavBar from './Components/NavBar';


const App = () => {

 return (

    <>

       <NavBar />

       <Routes>

          // ...

       </Routes>

    </>

 );

};

export default App;
```

Let’s now add Links to the NavBar this way:

```
// Components/NavBar.js
import { Link } from 'react-router-dom';


const NavBar = () => {
 return (
 <nav>
       <ul>
          <li>
             <Link to="/">Home</Link>
          </li>
          <li>
             <Link to="/about">About</Link>
          </li>
          <li>
             <Link to="/products">Products</Link>
          </li>
       </ul>
 </nav>
 );
};


export default NavBar;
```

In the preceding code, we first imported `Link` from `react-router-dom` and then added the `to` property based on the `path` we specified while configuring our routes. This is how simple it is to add links to our React application, allowing us to access configured routes.

How to Implement Active Links

When adding links to a navbar, we want users to be able to identify the exact link they are currently navigating to by giving it a different color than other nav links, adding extra styles like underline, and so on.

It is easier to handle and implement the active link feature in React by using the `NavLink` component rather than the `Link` component, which knows whether a user is currently navigated to by adding a `class` of `active` to such link.

The `NavLink` component will now replace the `Link` component in the `NavBar`:

```
// NavBar.js

import { NavLink } from 'react-router-dom';


const NavBar = () => {

 return (

    <nav>

       <ul>

          <li>

             <NavLink to="/">Home</NavLink>

          </li>

          <li>

             <NavLink to="/about">About</NavLink>

          </li>

          <li>

             <NavLink to="/products">Products</NavLink>

          </li>

       </ul>

    </nav>

 );

};


export default NavBar;
```

This alone does not make a significant difference in the user interface until we style the active class added, which can be done however you wish, but here is an example of the styling I used:

```
/* index.css */

ul li a {

 color: #000;

}


ul li a:hover {

 color: #00a8ff;

}


ul li a.active {

 color: #00a8ff;

}
```

How to fix No Routes Found Error

When routing, a situation may cause a user to access an unconfigured route or a route that does not exist; when this occurs, React does not display anything on the screen except a warning with the message "No routes matched location."

This can be fixed by configuring a new route to return a specific component when a user navigates to an unconfigured route as follows:

```
// App.js

import { Routes, Route } from 'react-router-dom';

import NoMatch from './Components/NoMatch';


const App = () => {

 return (

    <>

       <Routes>

          // ...

          <Route path="*" element={<NoMatch />} />

       </Routes>

    </>

 );

};


export default App;
```

In the preceding code, we created a route with the path `*` to get all non-configured paths and assign them to the attached component.

Note: We created a component called `NoMatch.js`, but you can name yours whatever you want to display 404, page not found, on the screen, so users know they are on the wrong page. We can also add a button that takes the user to another page or back, which leads us to programmatic navigation.

How to Navigate Programmatically in React

Programmatic navigation is the process of navigating/redirecting a user as a result of an action on a route, such as a login or a signup action, order success, or when he clicks on a back button.

Let's first look at how we can redirect to a page when an action occurs, such as when a button is clicked. We accomplish this by adding an `onClick` event, but first, we must create the route in our `App.js` file. After that, we can import the `useNavigate` hook from the `react-router-dom` and use it to navigate programmatically as follows:

```
// Products.js
import { useNavigate } from 'react-router-dom';

const Products = () => {
 const navigate = useNavigate();
 return (
    <div className="container">
       <div className="title">
          <h1>Order Product CockTails</h1>
       </div>
       <button className="btn" onClick={() => navigate('order-summary')}>
          Place Order
       </button>
    </div>
 );
};

export default Products;
```

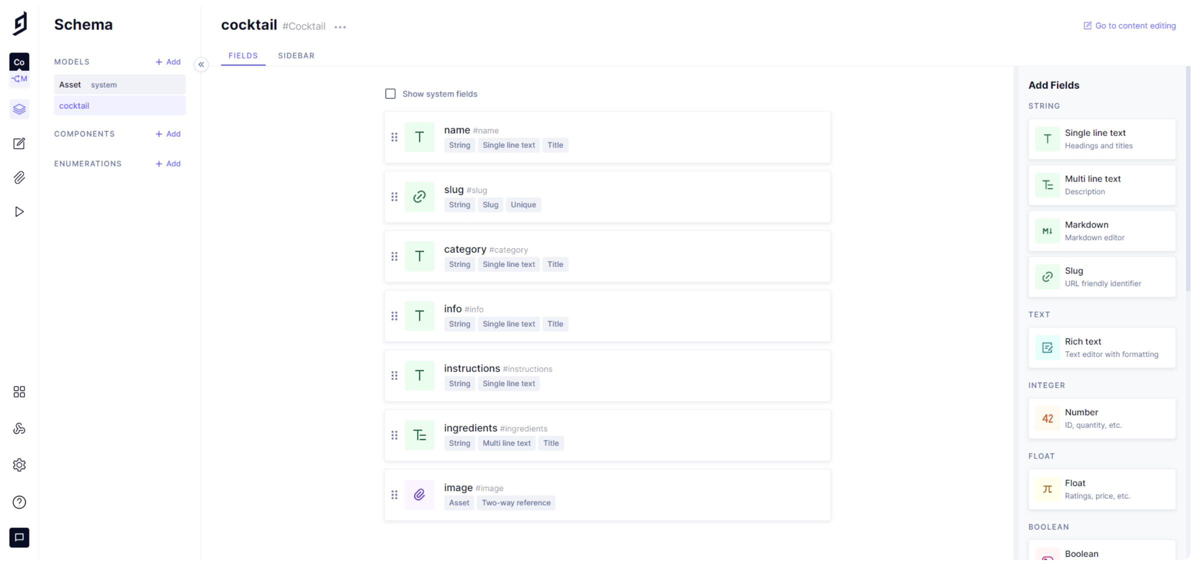
Note: We already created a route with the path `order-summary`, so when this button is clicked, the user is automatically navigated to the `orderSummary` component attached to this route. We can also use this hook to handle the back button in the following manner:

```
<button className="btn" onClick={() => navigate(-1)}>
 Go Back
</button>
```

Ensure you already have the hook imported and instantiated as we did earlier else this won't work.

How to Implement Dynamic Routing with React Router

We created three files in our pages folder earlier to implement routing, one of which was the products component, which we will populate with Hygraph content. We created a schema in Hygraph to receive cocktail details, and this is how it looks:


Schema
MODELS + Add
Asset system
cocktail
COMPONENTS + Add
ENUMERATIONS + Add
FIELDS
SIDEBAR
Go to content editing
Add Fields
STRING
Single line text Headings and titles
Multi line text Description
Markdown Markdown editor
Slug URL friendly identifier
TEXT
Rich text Text editor with formatting
INTEGER
Number ID, quantity, etc.
FLOAT
Float Ratings, price, etc.
BOOLEAN
Boolean

cocktail #Cocktail

Show system fields

name #name — String, Single line text, Title
slug #slug — String, Slug, Unique
category #category — String, Single line text, Title
info #info — String, Single line text, Title
instructions #instructions — String, Single line text
ingredients #ingredients — String, Multi line text, Title
image #image — Asset, Two-way reference

We then filled it in with cocktail specifics. We will now use GraphQL to retrieve these data so that we can consume them in our React project. This is how the products page appears:

```
// Products.js

import { useState, useEffect } from 'react';

import { useNavigate, Link } from 'react-router-dom';

import { request } from 'graphql-request';


const Products = () => {

 const [products, setProducts] = useState([]);

 const navigate = useNavigate();

 useEffect(() => {

    const fetchProducts = async () => {

       const { cocktails } = await request(

          'https://api-us-east-
1.hygraph.com/v2/cl4ji8xe34tjp01yrexjifxnw/master',

          `

       {

          cocktails {

             id

             name

             slug

             info

             ingredients

             instructions

             image {

             url

             }

             category

          }

       }

    `

       );
```

```
      setProducts(cocktails);
    };
    fetchProducts();
  }, []);

  return (
    <div className="container">
      <button className="btn" onClick={() => navigate(-1)}>
        Go Back
      </button>
      <div className="title">
        <h1>CockTails</h1>
      </div>
      <div className="cocktails-container">
        {products.map((product) => (
          <div key={product.id} className="cocktail-card">
            <img src={product.image.url} alt="" className="cocktail-
img" />
            <div className="cocktail-info">
              <div className="content-text">
                <h2 className="cocktail-name">{product.name}</h2>
                <span className="info">{product.info}</span>
              </div>
              <Link to={`/products/${product.slug}`}>
                <div className="btn">View Details</div>
              </Link>
            </div>
          </div>
        ))}
      </div>
      </div>
   );
  };

  export default Products;
```

Note: You can learn more about React and Hygraph here.

We fetched our content from Hygraph in the preceding code; if you already created your own schema, you can simply change the Endpoint URL and possibly the schema name if you gave it a different name.

Note: We added a button on each cocktail card so that a user can click it to view more details about each cocktail, but this would be done dynamically because we can create different components for each cocktail, which would be stressful if we had more than 5 different cocktails. Dynamic routing comes into play here.

We added a `LInk` and used string interpolation to dynamically attach the `slug` of each product to the path, so we can get the `slug` and use it to get the data to show.

Let us now put dynamic routing into action.

The first step would be to create the component that we want to render dynamically, and for that we would create a `ProductDetials.js` file where we would dynamically fetch details of each product based on the `slug` passed through the URL, but for now we can just place dummy data into the component like this:

```
  // ProductDetails.js
  const ProductDetails = () => {
   return (
      <div className="container">
         <h1>Products Details Page</h1>
      </div>
   );
  };

  export default ProductDetails;
```

We can now proceed to create a route to handle dynamic routing in our `App.js` file this way:

```
// App.js

import { Routes, Route } from 'react-router-dom';

// ...

import ProductDetails from './Pages/ProductDetails';


const App = () => {

 return (

    <>

       <Routes>

          // ...

          <Route path="/products/:slug" element={<ProductDetails />} />

       </Routes>

    </>

 );

};


export default App;
```

Note: We used `slug`, which can be anything, but this route will match any value and display the component as long as the pattern is the same, for example, `http://localhost:3000/products/cocktail` will show the `ProductDetails` component.

So far, we've dealt with the first part of dynamic routing. We must now obtain the parameter passed through the URL in order to dynamically query the data for the specific cocktail. This will be accomplished through the use of `urlParams`.

How to Use URL Params to handle Dynamic Routing

We will import the `useParams` hook into the `ProductDetails` component so that we can use it to get the URL parameter and then use that parameter to query our data from Hygraph via GraphQL.

For the time being, this is how to use the `useParams` hook, and we decided to destructure it so that we can directly access the `slug` parameter and use it within our App:

```
// ProductDetails.js

import { useParams } from 'react-router-dom';


const ProductDetails = () => {

 const { slug } = useParams();

 return (

    <div className="container">

       <h1>Products Details Page - {slug}</h1>

    </div>

 );

};


export default ProductDetails;
```

At this point, we have successfully been able to get the URL param passed, let’s now make use of this slug to fetch data from Hygraph using GraphQL:

```
// ProductDetails.js

import { useState, useEffect } from 'react';

import { useNavigate, useParams } from 'react-router-dom';

import { request } from 'graphql-request';


const ProductDetails = () => {

 const [product, setProduct] = useState([]);

 const navigate = useNavigate();

 const { slug } = useParams();

 useEffect(() => {

    const fetchProduct = async () => {

       const { cocktail } = await request(

          'https://api-us-east-
1.hygraph.com/v2/cl4ji8xe34tjp01yrexjifxnw/master',

          `

       {

          cocktail(where: {slug: "${slug}"}) {

             category

             info

             ingredients

             instructions

             image {

                url

             }

             name


          }

       }

    `

       );
```

```
        setProduct(cocktail);
      };
      fetchProduct();
    }, [slug]);

    return (
      <div className="container">
        <button className="btn" onClick={() => navigate(-1)}>
          Go Back
        </button>
        <div>
          <div className="title">
            <h1>{product.name}</h1>
          </div>
          <div className="flex-container">
            {product.image && (
              <img src={product.image.url} alt="" className="cocktail-
img" />
            )}
            <div className="cocktail-infos">
              <div className="row">
                <h3 className="label">Name: </h3>
                <p className="text">{product.name}</p>
              </div>
              <div className="row">
                <h3 className="label">Category: </h3>
                <p className="text">{product.category}</p>
              </div>
              <div className="row">
                <h3 className="label">Info: </h3>
```

```
                    <p className="text">{product.info}</p>

                 </div>

                 <div className="row">

                    <h3 className="label">Instructions: </h3>

                    <p className="text">{product.instructions}</p>

                 </div>

                 <div className="row">

                    <h3 className="label">Ingredients: </h3>

                    <p className="text">{product.ingredients}</p>

                 </div>

              </div>

           </div>

        </div>

     </div>

   );

  };


  export default ProductDetails;
```

At this point, we have successfully implemented dynamic routing.

How to Implement Lazy Loading with React Router

We've already seen how to create routes and implement routing with React router; now let's look at how to lazy load routes with React router.

Lazy loading is a technique in which components that are not required on the home page are not loaded until a user navigates to that page, allowing our application to load faster than having to wait for the entire app to load at once. This contributes to improved performance, which leads to a positive user experience.

To implement lazy loading, simply go to `App.js` and wrap our routes with the `Suspense` component, along with a `fallback` props that are rendered on the screen until the component loads:

```
// App.js

import { lazy, Suspense } from 'react';

import { Routes, Route } from 'react-router-dom';


import NavBar from './Components/NavBar';

const Home = lazy(() => import('./Pages/Home'));

const About = lazy(() => import('./Pages/About'));

const Products = lazy(() => import('./Pages/Products'));

const ProductDetails = lazy(() => import('./Pages/ProductDetails'));

const NoMatch = lazy(() => import('./Components/NoMatch'));


const App = () => {

 return (

    <>

       <NavBar />

       <Suspense fallback={<div className="container">Loading...</div>}>

          <Routes>

             <Route path="/" element={<Home />} />

             <Route path="/about" element={<About />} />

             <Route path="/products" element={<Products />} />

             <Route path="/products/:slug" element={<ProductDetails />}
/>

             <Route path="*" element={<NoMatch />} />

          </Routes>

       </Suspense>

    </>

 );

};


export default App;
```

Note: We wrapped the routes with the Suspense component, and it's important for you to know that the fallback props can hold a component.

Conclusion

We learned about routing and how to implement it in our React application in this guide. It is critical to understand that the React router is what allows us to perform single-page routing without reloading the application.