React Context Injection

MAY 24, 2021 BY ROBIN WIERUCH - EDIT THIS POST

Quite recently, in a React freelance project of mine, I came across a React pattern where I had been using the **same UI components** repetitively across multiple pages **tied to specific page information**. Let's walk through this thing that I name Context Injection in React.

Let's say we have an App component which splits its top-level navigation into three pages:

```
const App = () => {
  return (
    <Router>
      <div>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
```

```
            <Link to="/portfolio">Portfolio</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
        </ul>

        <hr />

        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/portfolio">
            <Portfolio />
          </Route>
          <Route path="/">
            <h2>Home</h2>
          </Route>
        </Switch>
      </div>
    </Router>
  );
};
```

Two of the three pages are of relevance here -- Portfolio and About -- because they are the ones which will get equipped with context injection, because they share the same underlying UI component. Let's check out Portfolio first:

```
const Portfolio = () => {
  const match = useRouteMatch();

  return (
    <div>
      <h1>Portfolio</h1>

      <ul>
        <li>
          <Link to={`${match.url}`}>Summary</Link>
        </li>
        <li>
          <Link to={`${match.url}/events`}>Events</Link>
        </li>
      </ul>

      <hr />

      <Switch>
        <Route path={`${match.path}/events`}>
          <EventList />
        </Route>
```

```
        <Route path={match.path}>
          <Summary />
        </Route>
      </Switch>
    </div>
  );
};
```

The Portfolio page renders two inner routes (or inner pages, e.g. via tabs): summary and events. Here the events (EventList) are of interested, because if we look into the Portfolio page's sibling About page, it renders the EventList component as well:

```
const About = () => {
  const match = useRouteMatch();

  return (
    <div>
      <h1>About</h1>

      <ul>
        <li>
          <Link to={`${match.url}`}>Summary</Link>
        </li>
        <li>
          <Link to={`${match.url}/events`}>Events</Link>
        </li>
      </ul>

      <hr />

      <Switch>
        <Route path={`${match.path}/events`}>
          <EventList />
        </Route>
        <Route path={match.path}>
          <Summary />
        </Route>
      </Switch>
    </div>
  );
};

export { About };
```

By seeing all three components -- the parent App component and its child components Portfolio and About --, it becomes easier to define the problem space: We want to reuse the EventList components on multiple pages, but each EventList should know about its specific context where it's rendered.

The solution by just passing props seems straightforward for most React developers:

```
const About = () => {
  const match = useRouteMatch();

  return (
    <div>

      ...

      <hr />

      <Switch>
        <Route path={`${match.path}/events`}>
          <EventList whereAmI="about" />
        </Route>
        <Route path={match.path}>
          <Summary />
        </Route>
      </Switch>
    </div>
  );
};
```

We can pass page specific information from the pages to the shared UI component. In this case, the page components About and Portfolio would become mediator components. Now when using the EventList component, we have access to this page specific property and can act upon it:

```
const EventList = ({ whereAmI }) => {
  const match = useRouteMatch();

  return (
    <div>
      <h2>Events</h2>

      <ul>
        <li>
          <Link to={`${match.url}/1`}>Event 1</Link>
        </li>
        <li>
          <Link to={`${match.url}/2`}>Event 2</Link>
        </li>
      </ul>

      <Switch>
        <Route path={`${match.path}/:eventId`}>
          <EventItem />
        </Route>
```

```
          <Route path={match.path}>
            <p>
              Please select an event in <strong>{whereAmI}</strong>.
            </p>
          </Route>
        </Switch>
      </div>
    );
  };
```

In this example of the an list component, we are rendering two hardcoded items. Whether we render this component in the Portfolio or About page does not change the data at the moment.

However, now by having the page specific information (`whereAmI`) at our disposal, we could fetch these items from a remote API based on this knowledge. For instance, the request for the data fetching could change the API endpoint conditionally:

```
const url = `api.mydomain.com/events/${whereAmI}`;
```

That's the solution to our problem, isn't it? We can reuse a generic component in our Portfolio and About components and **inject the information via React props** which gives the UI component knowledge about where it's used and based upon this knowledge it can perform its actions. For smaller React projects, where there are not many components, this solution is absolutely fine and you should go with it.

But let's see how this plays out for large React projects.

. . .

In a recent project of mine, each page had a nesting of at least 10 component levels deep, where the 10th component isn't necessarily a leaf component yet, but it still had to know where it's rendered to perform page specific API requests.

Again, a straightforward solution would be just passing the **page specific information via props** (`whereAmI`) down to all the components. While this is a solution which works, it becomes more tedious with every nested component (see vertical props drilling) and every additional prop that needs to be passed (see horizontal props drilling).

Instead of using props, an alternative solution to this would be **acting upon the URL**. In this case, the EventList component would act upon the route `/portfolio` or `/about`. However, with lots of nested routes (and params in between), decomposing the URL to something useful isn't as straightforward. Entering context injections

useful isn't as straightforward. Entering context injections ...

. . .

For the following, you can find a working demo over here. By using React Context and its useContext Hook, we can pass page specific information to all nested components without the pain of props drilling. Let's see how this looks like for our use case. First, we initialize a new context:

```
import React from 'react';

const PageContext = React.createContext();

export { PageContext };
```

Then we use this context's Provider component as wrapper component for our page components; which render themselves the underlying common UI component that needs to know about this information:

```
const App = () => {
  return (
    <Router>
      <div>
        ...

        <Switch>
          <Route path="/about">
            <PageContext.Provider value="about">
              <About />
            </PageContext.Provider>
          </Route>
          <Route path="/portfolio">
            <PageContext.Provider value="portfolio">
              <Portfolio />
            </PageContext.Provider>
          </Route>
          <Route path="/">
            <h2>Home</h2>
          </Route>
        </Switch>
      </div>
    </Router>
  );
};
```

Usually when using React Context, in most cases you will only use one Provider component at the very top-level of your React component hierarchy. In addition, the value of the context will mostly change over time (e.g. via React State). However, in this case it's quite the

will mostly change over time (e.g. via React State). However, in this case it's quite the opposite: We are using a Provider component at two distinct locations with a static value.

This way, each underlying component tree receives its page specific information (here `value`).

Now in our EventList component -- or next to it in a new *context.js* file (see how to create a scaling React folder structure) -- we can create a new context consumer hook which consumes this new Context and selects the respective page specific information from a dictionary:

```
const DICTIONARY = {
  portfolio: {
    whereAmI: 'Portfolio yoo',
  },
  about: {
    whereAmI: 'About yay',
  },
};

const usePage = () => {
  const page = React.useContext(PageContext);

  return DICTIONARY[page];
};
```

When using this hook in our UI component, we can render the respective context based string:

```
const EventList = () => {
  const match = useRouteMatch();

  const { whereAmI } = usePage();

  return (
    <div>
      <h2>Events</h2>

      <ul>
        <li>
          <Link to={`${match.url}/1`}>Event 1</Link>
        </li>
        <li>
          <Link to={`${match.url}/2`}>Event 2</Link>
        </li>
      </ul>

      <Switch>
```

```
          <Route path={`${match.path}/:eventId`}>
            <EventItem />
          </Route>
          <Route path={match.path}>
            <p>
              Please select an event in <strong>{whereAmI}</strong>.
            </p>
          </Route>
        </Switch>
      </div>
    );
  };
```

Now the information is not coming from React props anymore, but from React context.

In this example, this might seem like a super overkill, because the shared UI component is rendered as child component right below the page specific components. However, as mentioned earlier, imagine this scenario when this shared UI component is rendered 10 levels below of the page specific component and you want to avoid props drilling.

There is more to context injection: If EventList needs more context aware information, a React developer can just extend the dictionary next to the component and use this additional information in the component:

```
const DICTIONARY = {
  portfolio: {
    whereAmI: 'Portfolio yoo',
    api: '/portfolio/events',
  },
  about: {
    whereAmI: 'About yay',
    api: '/about/events',
  },
};
```

What if a child component of EventList, let's say EventItem, needs to access context aware page information too? One way would be extending the dictionary in the EventList component and just passing this information down to EventItem as props.

However, if EventItem is another 5 components below of EventList component, you want to avoid props drilling again. Fortunately, you have the context available across the whole component hierarchy now, so you could just create a new context consuming hook with a dictionary next to the EventItem component:

```
const DICTIONARY = {
```

```
  portfolio: {
    operationName: 'multiply eventId with itself',

    operation: (id) => Number(id) * Number(id),
  },
  about: {
    operationName: 'add eventId to itself',
    operation: (id) => Number(id) + Number(id),
  },
};

const usePage = () => {
  const page = React.useContext(PageContext);

  return DICTIONARY[page];
};
```

Then EventItem can use this hook and display information based on this context:



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

import { usePage } from './context';

const EventItem = () => {
  const { eventId } = useParams();

  const { operationName, operation } = usePage();

  return (
    <table>
      <tr>
        <th>Value</th>
        <th>Source</th>
      </tr>
      <tr>
        <td>{eventId}</td>
        <td>eventId URL via React Router</td>
      </tr>
      <tr>
        <td>{operationName}</td>
        <td>constant variable via PageContext</td>
      </tr>
      <tr>
        <td>{operation(eventId)}</td>
        <td>computed variable via PageContext</td>
      </tr>
    </table>
  );
};
```

As you can see, the context aware information doesn't need to be just JavaScript primitives, but can also be functions which are dynamically executed with a flexible input.

In conclusion, this method of context injection by using a provider component for sibling components helped us to distinguish page specific tree hierarchies from one another while avoiding props drilling. Each shared UI component can act based on its page its rendered on. This way, we were able to execute different API requests, include page specific component behavior -- which isn't the same for every page --, and change styling as well.

In the end, I hope this pattern helps React teams who are working on larger React applications the same way it helped us in our large React project.

Show Comments

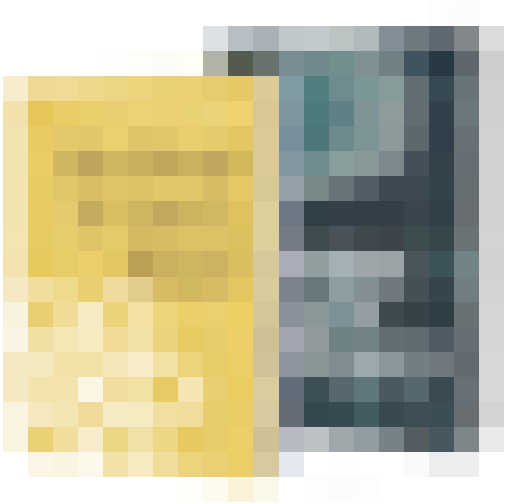
KEEP READING ABOUT JAVASCRIPT›

CONST VS LET, AND VAR IN JAVASCRIPT

There are three different ways to declare a variable in JavaScript: const, let and var. Historically, var has been the only way of declaring a JavaScript variable : An addition to JavaScript -- to be…

HOW TO USECONTEXT IN REACT?

React's Function Components come with React Hooks these days. Not only can React Hooks be used for State in React but also for using React's Context in a more convenient way. This tutorial shows…



THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development
- ✔ Learn JavaScript

✔ Access Tutorials, eBooks and Courses

✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.



PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms