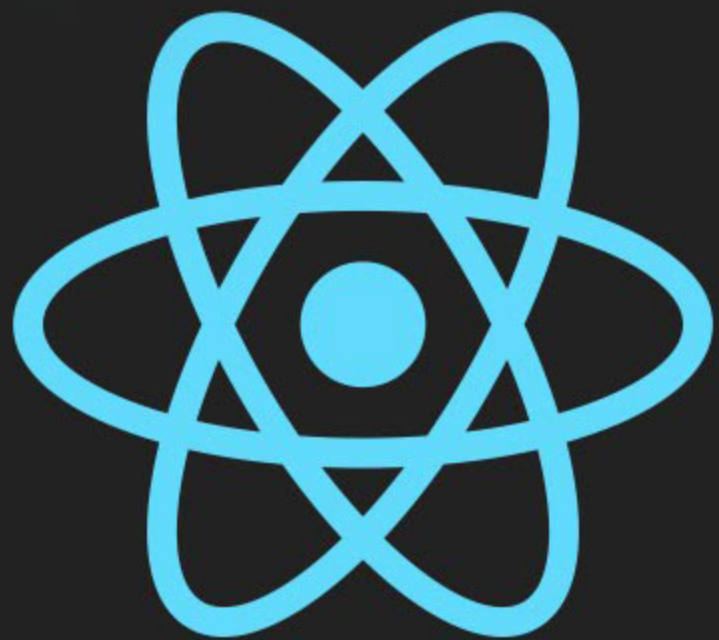


React And Friends



React Router

About react-router

react-router is developed and maintained by React Training and many amazing contributors. If you're interested in learning more about what React can do for your company, please get in touch!

LEARN ONCE, ROUTE ANYWHERE

REACT ROUTER

Components are the heart of React's powerful, declarative programming model. React Router is a collection of **navigational components** that compose declaratively with your application. Whether you want to have **bookmarkable URLs** for your web app or a composable way to navigate in **React Native**, React Router works wherever React is rendering--so take your pick!



A New API and A New Mental Model

Here it is in v3:

```
import { Router, Route, IndexRoute } from 'react-router'

const PrimaryLayout = props => (
  <div className="primary-layout">
    <header> Our React Router 3 App </header>
    <main> {props.children} </main>
  </div>
)

const HomePage = () => <div>Home Page</div>
const UsersPage = () => <div>Users Page</div>

const App = () => (
  <Router history={browserHistory}>
    <Route path="/" component={PrimaryLayout}>
      <IndexRoute component={HomePage} />
      <Route path="/users" component={UsersPage} />
    </Route>
  </Router>
)

render(<App />, document.getElementById('root'))
```

A New API and A New Mental Model

Here it is in v4:

New API Concept: Since our app is meant for the browser, we need to wrap it in `<BrowserRouter>` which comes from v4. Also notice we import from `react-router-dom` now (which means we npm install `react-router-dom` not `react-router`). Hint! It's called `react-router-dom` now because there's also a native version.

```
import { BrowserRouter, Route } from 'react-router-dom'

const PrimaryLayout = () => (
  <div className="primary-layout">
    <header> Our React Router 4 App</header>
    <main>
      <Route path="/" exact component={HomePage} />
      <Route path="/users" component={UsersPage} />
    </main>
  </div>
)

const HomePage = () => <div>Home Page</div>
const UsersPage = () => <div>Users Page</div>

const App = () => (
  <BrowserRouter>
    <PrimaryLayout />
  </BrowserRouter>
)

render(<App />, document.getElementById('root'))
```

Inclusive Routing

In the previous example, we're trying to render either the **HomePage** or the **UsersPage** depending on the path. If the **exact** prop were removed from the example, both the **HomePage** and **UsersPage** components would have rendered at the same time when visiting `/users` in the browser.

To demonstrate how inclusive routing is helpful, let's include a **UserMenu** in the header, but only if we're in the user's part of our application:

```
const PrimaryLayout = () => (  
  <div className="primary-layout">  
    <header> Our React Router 4 App  
      <Route path="/users" component={UsersMenu} />  
    </header>  
    <main>  
      <Route path="/" exact component={HomePage} />  
      <Route path="/users" component={UsersPage} />  
    </main>  
  </div>  
)
```

Exclusive Routing

Now, when the user visits `/users`, both components will render. Something like this was doable in v3 with certain patterns, but it was more difficult. Thanks to v4's inclusive routes, it's now a breeze.

If you need just one route to match in a group, use `<Switch>` to enable exclusive routing:

```
const PrimaryLayout = () => (  
  <div className="primary-layout">  
    <PrimaryHeader />  
    <main>  
      <Switch>  
        <Route path="/" exact component={HomePage} />  
        <Route path="/users/add" component={UserAddPage} />  
        <Route path="/users" component={UsersPage} />  
        <Redirect to="/" />  
      </Switch>  
    </main>  
  </div>  
)
```

Index Routes and Not Found

While there is no more `<IndexRoute>` in v4, using `<Route exact>` achieves the same thing. Or if no routes resolved, then use `<Switch>` with `<Redirect>` to redirect to a default page with a valid path (as I did with `HomePage` in the example), or even a not-found page.

Nested Layouts

You're probably starting to anticipate nested sub layouts and how you might achieve them. I didn't think I would struggle with this concept, but I did. React Router v4 gives us a lot of options, which makes it powerful. Options, though, means the freedom to choose strategies that are not ideal. On the surface, nested layouts are trivial, but depending on your choices you may experience friction because of the way you organized the router.

To demonstrate, let's imagine that we want to expand our users section so we have a "browse users" page and a "user profile" page. We also want similar pages for products. Users and products both need sub-layout that are special and unique to each respective section. For example, each might have different navigation tabs. There are a few approaches to solve this, some good and some bad. The first approach is not very good but I want to show you so you don't fall into this trap. The second approach is much better.

For the first, let's modify our `PrimaryLayout` to accommodate the browsing and profile pages for users and products:

Nested Layouts

```
const PrimaryLayout = props => {  
  return (  
    <div className="primary-layout">  
      <PrimaryHeader />  
      <main>  
        <Switch>  
          <Route path="/" exact component={HomePage} />  
          <Route path="/users" exact component={BrowseUsersPage} />  
          <Route path="/users/:userId" component={UserProfilePage} />  
          <Route path="/products" exact component={BrowseProductsPage} />  
          <Route path="/products/:productId" component={ProductProfilePage} />  
          <Redirect to="/" />  
        </Switch>  
      </main>  
    </div>  
  )  
}
```

While this does technically work, taking a closer look at the two user pages starts to reveal the problem:

```
const BrowseUsersPage = () => (  
  <div className="user-sub-layout">  
    <aside>  
      <UserNav />  
    </aside>  
    <div className="primary-content">  
      <BrowseUserTable />  
    </div>  
  </div>  
)  
  
const UserProfilePage = props => (  
  <div className="user-sub-layout">  
    <aside>  
      <UserNav />  
    </aside>  
    <div className="primary-content">  
      <UserProfile userId={props.match.params.userId} />  
    </div>  
  </div>  
)
```

Each user page not only renders its respective content but also has to be concerned with the sub layout itself (and the sub layout is repeated for each). While this example is small and might seem trivial, repeated code can be a problem in a real application. Not to mention, each time a **BrowseUsersPage** or **UserProfilePage** is rendered, it will create a new instance of **UserNav** which means all of its lifecycle methods start over. Had the navigation tabs required initial network traffic, this would cause unnecessary requests — all because of how we decided to use the router.

Here's a different approach which is better:

```
const PrimaryLayout = props => {  
  return (  
    <div className="primary-layout">  
      <PrimaryHeader />  
      <main>  
        <Switch>  
          <Route path="/" exact component={HomePage} />  
          <Route path="/users" component={UserSubLayout} />  
          <Route path="/products" component={ProductSubLayout} />  
          <Redirect to="/" />  
        </Switch>  
      </main>  
    </div>  
  )  
}
```

Instead of four routes corresponding to each of the user's and product's pages, we have two routes for each section's layout instead.

With this strategy, it becomes the task of the sub layouts to render additional routes. Here's what the UserSubLayout could look like:

```
const UserSubLayout = () => (  
  <div className="user-sub-layout">  
    <aside>  
      <UserNav />  
    </aside>  
    <div className="primary-content">  
      <Switch>  
        <Route path="/users" exact component={BrowseUsersPage} />  
        <Route path="/users/:userId" component={UserProfilePage} />  
      </Switch>  
    </div>  
  </div>  
)
```

The most obvious win in the new strategy is that the layout isn't repeated among all the user pages. It's a double win too because it won't have the same lifecycle problems as with the first example.

One thing to notice is that even though we're deeply nested in our layout structure, the routes still need to identify their full path in order to match. To save yourself the repetitive typing (and in case you decide to change the word "users" to something else), use `props.match.path` instead:

```
const UserSubLayout = props => (  
  <div className="user-sub-layout">  
    <aside>  
      <UserNav />  
    </aside>  
    <div className="primary-content">  
      <Switch>  
        <Route path={props.match.path} exact component={BrowseUsersPage} />  
        <Route path={` ${props.match.path} /:userId` } component={UserProfilePage} />  
      </Switch>  
    </div>  
  </div>  
)
```

Match

As we've seen so far, `props.match` is useful for knowing what `userId` the profile is rendering and also for writing our routes. The `match` object gives us several properties including `match.params`, `match.path`, `match.url` and several more.

`match.path` vs `match.url`

The differences between these two can seem unclear at first. Console logging them can sometimes reveal the same output making their differences even more unclear. For example, both these console logs will output the same value when the browser path is `/users`:

Match example

```
const UserSubLayout = ({ match }) => {
  console.log(match.url) // output: "/users"
  console.log(match.path) // output: "/users"
  return (
    <div className="user-sub-layout">
      <aside>
        <UserNav />
      </aside>
      <div className="primary-content">
        <Switch>
          <Route path={match.path} exact component={BrowseUsersPage} />
          <Route path={`/${match.path}/:userId`} component={UserProfilePage} />
        </Switch>
      </div>
    </div>
  )
}
```

While we can't see the difference yet, `match.url` is the actual path in the browser URL and `match.path` is the path written for the router. This is why they are the same, at least so far. However, if we did the same console logs one level deeper in `UserProfilePage` and visit `/users/5` in the browser, `match.url` would be `/users/5` and `match.path` would be `/users/:userId`.

Authorized Route

It's very common in applications to restrict the user's ability to visit certain routes depending on their login status. Also common is to have a "look-and-feel" for the unauthorized pages (like "log in" and "forgot password") vs the "look-and-feel" for the authorized ones (the main part of the application). To solve each of these needs, consider this main entry point to an application:

```
class App extends React.Component {  
  render() {  
    return (  
      <Provider store={store}>  
        <BrowserRouter>  
          <Switch>  
            <Route path="/auth" component={UnauthorizedLayout} />  
            <AuthorizedRoute path="/app" component={PrimaryLayout} />  
          </Switch>  
        </BrowserRouter>  
      </Provider>  
    )  
  }  
}
```


There are a few takeaways with this approach. The first being that I'm choosing between two top-level layouts depending on which section of the application we're in. Visiting paths like `/auth/login` or `/auth/forgot-password` will utilize the UnauthorizedLayout — one that looks appropriate for those contexts. When the user is logged in, we'll ensure all paths have an `/app` prefix which uses AuthorizedRoute to determine if the user is logged in or not. If the user tries to visit a page starting with `/app` and they aren't logged in, they will be redirected to the login page.

```
class AuthorizedRoute extends React.Component {
  componentWillMount() {
    getLoggedInUser()
  }
  render() {
    const { component: Component, pending, logged, ...rest } = this.props
    return (
      <Route {...rest} render={props => {
        if (pending) return <div>Loading...</div>
        return logged ? <Component {...this.props} />: <Redirect to="/auth/login" />
      }} />
    )
  }
}

const stateToProps = ({ loggedInUserState }) => ({
  pending: loggedInUserState.pending,
  logged: loggedInUserState.logged
})

export default connect(stateToProps)(AuthorizedRoute)
```

<Link> vs <NavLink>




In v4, there are two ways to integrate an anchor tag with the router: <Link> and <NavLink>

<NavLink> works the same as <Link> but gives you some extra styling abilities depending on if the <NavLink> matches the browser's URL. For instance, in the example application, there is a <PrimaryHeader> component that looks like this:

```
const PrimaryHeader = () => (  
  <header className="primary-header">  
    <h1>Welcome to our app!</h1>  
    <nav>  
      <NavLink to="/app" exact activeClassName="active">Home</NavLink>  
      <NavLink to="/app/users" activeClassName="active">Users</NavLink>  
      <NavLink to="/app/products" activeClassName="active">Products</NavLink>  
    </nav>  
  </header>  
)
```



REACT FINAL FORM

- ✓ Zero dependencies
- ✓ Only peer dependencies: React and  [Final Form](#)
- ✓ Opt-in subscriptions - only update on the state you need!
- ✓  2.9k gzipped 

Installation

```
npm install --save react-final-form final-form
```

Getting Started

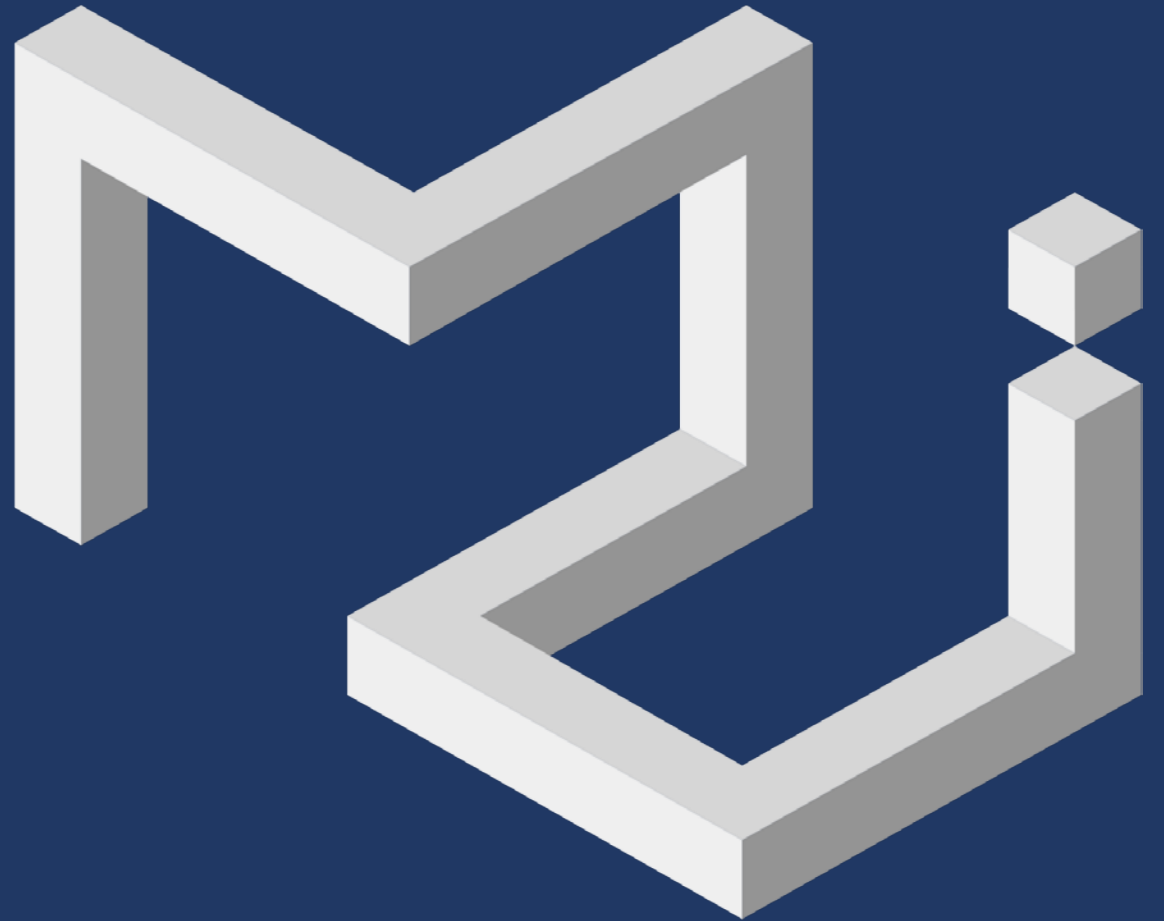
🚩 React Final Form is a thin React wrapper for 🚩 Final Form, which is a subscriptions-based form state management library that uses the [Observer pattern](#), so only the components that need updating are re-rendered as the form's state changes. By default, 🚩 React Final Form subscribes to *all* changes, but if you want to fine tune your form to optimized blazing-fast perfection, you may specify only the form state that you care about for rendering your gorgeous UI.

You can think of it a little like GraphQL's feature of only fetching the data your component needs to render, and nothing else.

Here's what it looks like in your code:

<https://github.com/final-form/react-final-form>

Material UI



Installation

```
npm install material-ui
```

Robot Font

Material-UI was designed with the Roboto font in mind. So be sure to include it in your project. Here are some instructions on how to do so.

ES Compiling

The examples [in this](#) documentation use the stage-1 features [of](#) the ECMAScript specification. Be sure [if](#) you are testing these examples [in](#) your own project that you have the proper plugins installed [in](#) your compiler. Here are some instructions on how to install the plugin for Babel.

Usage

Beginning with v0.15.0, Material-UI components require a theme to be provided. The quickest way to get up and running is by using the `MuiThemeProvider` to inject the theme into your application context. Following that, you can use any of the components as demonstrated in our documentation.

Here is a quick example to get you started:

```
//App.js
import React from 'react';
import ReactDOM from 'react-dom';
import MuiThemeProvider
from 'material-ui/styles/MuiThemeProvider';
import MyAwesomeReactComponent
from './MyAwesomeReactComponent';
```

```
const App = () => (
  <MuiThemeProvider>
    <MyAwesomeReactComponent />
  </MuiThemeProvider>
);
```

```
ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

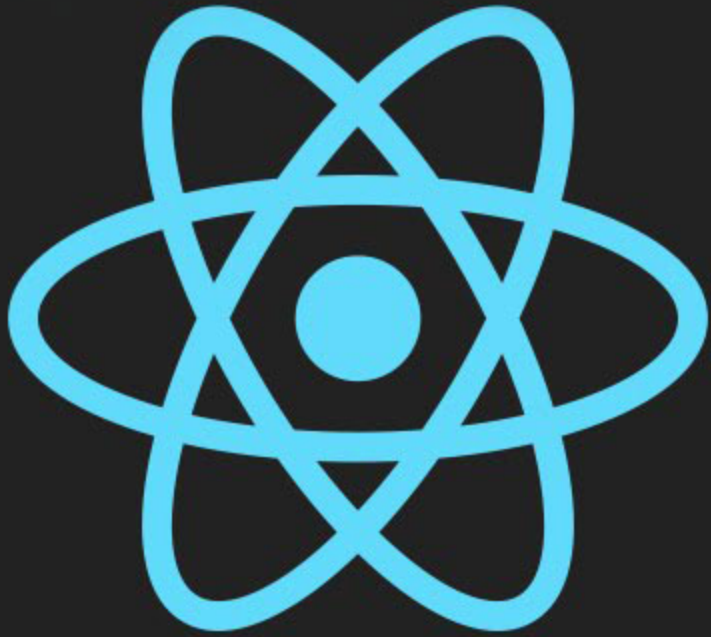
```
//MyAwesomeReactComponent.js
import React from 'react';
import RaisedButton from 'material-ui/RaisedButton';

const MyAwesomeReactComponent = () => (
  <RaisedButton label="Default" />
);

export default MyAwesomeReactComponent;
```

React

Recompose



Installation

npm install recompose --save

You can use Recompose to...

...lift state into functional wrappers

Helpers like `withState()` and `withReducer()` provide a nicer way to express state updates:

```
const enhance = withState('counter', 'setCounter', 0)
const Counter = enhance(({ counter, setCounter }) =>
  <div>
    Count: {counter}
    <button onClick={() => setCounter(n => n + 1)}>Increment</button>
    <button onClick={() => setCounter(n => n - 1)}>Decrement</button>
  </div>
)
```

withHandlers()

```
const enhance = compose(
  useState('counter', 'setCounter', 0),
  withHandlers({
    onAddValue: props => event => {
      const { setCounter } = props
      setCounter(n => n+1)
    },
    onSubValue: props => event => {
      const { setCounter } = props
      setCounter(n => n-1)
    }
  })
)
```

```
const CounterComponent = enhance(({ counter, setCounter, onAddValue, onSubValue }) =>
  <div>
    <button onClick={onAddValue}>Increment</button>
    Count: {counter}
    <button onClick={onSubValue}>Decrement</button>
  </div>
)
```

lifecycle()

```
const enhance = compose(
  useState('data', 'setData', null),
  lifecycle({
    componentDidMount() {
      const { setData } = this.props
      myCall
        .getResponse()
        .then(response => {
          setData(response)
        });
    }
  })
)

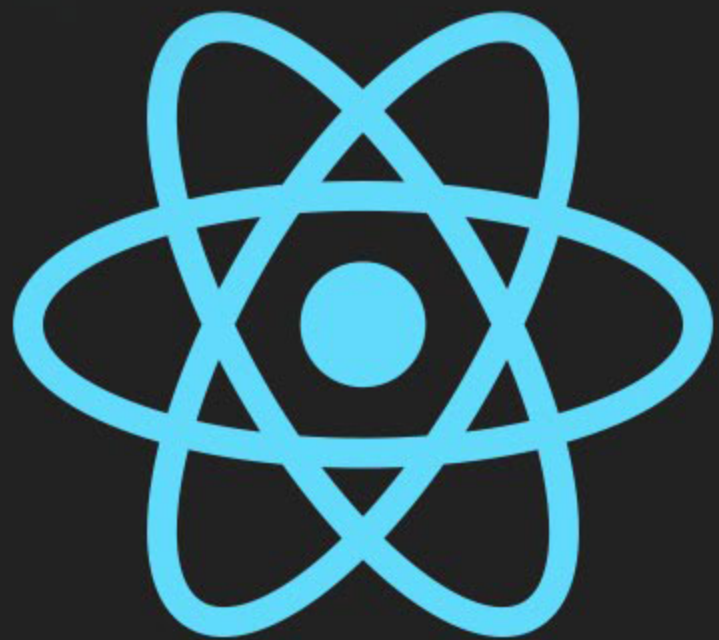
const DataComponent = enhance((props) =>
  <div>
    {props.data}
  </div>
)
```

branch()

```
const SomeCoolComponent = ({ isComponent1 }) => (  
  branch(isComponent1, <Component1 />, <Component2 />)  
)
```

```
const enhance = compose(  
  useState('page', 'setPage', 'index'),  
  branch(props => props.page === 'about', renderComponent(<About />)),  
  branch(props => props.page === 'contact', renderComponent(<Contact />))  
)
```

```
const Index = enhance((props) =>  
  <div>Index Page</div>  
)
```



React
INTL

Installation

`npm install react-intl --save`

The react-intl npm package distributes the following modules (links from unpkg):

- CommonJS: unbundled dependencies, "main" in package.json, warnings in dev.
- ES6: unbundled dependencies, "jsnext:main" and "module" in package.json, warnings in dev.
- UMD dev: bundled dependencies (except react), browser or Node, warnings.
- UMD prod: minified, bundled dependencies (except react), browser or Node, no warnings.
- UMD Locale Data: grouped by language, browser or Node, index.js contains all locales.

Note: React Intl's locale data is in a directory at the package's root. This allows the locale data to be import-ed or require-d relative to the package. For example:

`import englishLocaleData from 'react-intl/locale-data/en';`

Intl module

Whether you use the ES6, CommonJS, or UMD version of React Intl, they all provide the same named exports:

- addLocaleData
- intlShape
- injectIntl
- defineMessages
- IntlProvider
- FormattedDate
- FormattedTime
- FormattedRelative
- FormattedNumber
- FormattedPlural
- FormattedMessage
- FormattedHTMLMessage

Note: When using the UMD version of React Intl without a module system, it will expect react to exist on the global variable: React, and put the above named exports on the global variable: ReactIntl.

Loading Local Data

React Intl relies on locale data to support its plural and relative-time formatting features. This locale data is split out from the main library because it's 39KB gz, and instead grouped per language; e.g., `en.js`, `fr.js`, `zh.js`, etc.

If you are targeting browsers or Node versions which don't have the Intl APIs built-in, you'll need to polyfill the runtime using the Intl.js polyfill (See above for details.) This polyfill also has its locale data separated into files that are organized by locale tag; e.g., `en-US.js`, `fr.js`, `zh-Hant-TW.js`, etc.

Because of these differences in how the locale data is organized, you'll need to put extra attention on the locale data files available for the Intl.js polyfill and React Intl when loading locale data dynamically.

Local Data In Browser

When using React Intl in browsers, it will only contain locale data for basic English by default. This means you'll need to either bundle locale data with your app code, or dynamically load a locale data UMD module based on the current user's locale.

React Intl provides an `addLocaleData` API which can be passed the contents of a locale data module and will register it in its locale data registry.

If your app only supports a few languages, we recommend bundling React Intl's locale data for those languages with your app code as this approach is simpler. Here's an example of an app that supports English, French, and Spanish:

```
// app.js
import {addLocaleData} from 'react-intl';
import en from 'react-intl/locale-data/en';
import fr from 'react-intl/locale-data/fr';
import es from 'react-intl/locale-data/es';

addLocaleData([...en, ...fr, ...es]);
// ...
```

If your app supports many locales, you can also dynamically load the locale data needed for the current user's language. This would involve outputting a different HTML document per users which includes a `<script>` to the correct locale data file. When loading a locale data file in a runtime without a module system, it will be added to a global variable: `ReactIntlLocaleData`. Here's an example of loading React Intl and locale data for a French user:

```
<!-- Load React and ReactDOM if they're not already on the page. -->
<script src="https://unpkg.com/react@latest/dist/react.min.js"></script>
<script src="https://unpkg.com/react-dom@latest/dist/react-dom.min.js"></script>

<!-- Load ReactIntl and its locale data for French. -->
<script src="https://unpkg.com/react-intl@latest/dist/react-intl.min.js"></script>
<script src="https://unpkg.com/react-intl@latest/locale-data/fr.js"></script>
<script>
  ReactIntl.addLocaleData(ReactIntlLocaleData.fr);
</script>
```

Creating an I18n Context

Now with React Intl and its locale data loaded an i18n context can be created for your React app.

React Intl uses the provider pattern to scope an i18n context to a tree of components. This allows configuration like the current locale and set of translated strings/messages to be provided at the root of a component tree and made available to the `<Formatted*>` components. This is the same concept as what Flux frameworks like Redux use to provide access to a store within a component tree.

All apps using React Intl must use the `<IntlProvider>` component.

The most common usage is to wrap your root React component with `<IntlProvider>` and configure it with the user's current locale and the corresponding translated strings/messages:

```
ReactDOM.render(  
  <IntlProvider  
    locale={usersLocale}  
    messages={translationsForUsersLocale}  
  >  
    <App/>  
  </IntlProvider>,  
  document.getElementById('container')  
)
```

Formatting Data

React Intl has two ways to format data, through React components and its API. The components provide an idiomatic-React way of integrating internationalization into a React app, and the `<Formatted*>` components have benefits over always using the imperative API directly. The API should be used when your React component needs to format data to a string value where a React element is not suitable; e.g., a title or aria attribute, or for side-effect in `componentDidMount`.

React Intl's imperative API is accessed via `injectIntl`, a High-Order Component (HOC) factory. It will wrap the passed-in React component with another React component which provides the imperative formatting API into the wrapped component via its props. (This is similar to the connect-to-stores pattern found in many Flux implementations.)

Here's an example using `<IntlProvider>`, `<Formatted*>` components, and the imperative API to setup an i18n context and format data:

```

import React, {PropTypes} from 'react';
import ReactDOM from 'react-dom';
import {
  injectIntl,
  IntlProvider,
  FormattedRelative,
} from 'react-intl';

const PostDate = injectIntl(({date, intl}) => (
  <span title={intl.formatDate(date)}>
    <FormattedRelative value={date}/>
  </span>
));

const App = ({post}) => (
  <div>
    <h1>{post.title}</h1>
    <p><PostDate date={post.date}/></p>
    <div>{post.body}</div>
  </div>
);

```

```

ReactDOM.render(
  <IntlProvider locale={navigator.language}>
    <App
      post={{
        title: 'Hello, World!',
        date: new Date(1459913574887),
        body: 'Amazing content.',
      }}
    />
  </IntlProvider>,
  document.getElementById('container')
);

```

Assuming navigator.language is **"en-us"**:

```

<div>
  <h1>Hello, World!</h1>
  <p><span title="4/5/2016">yesterday</span></p>
  <div>
    Amazing content.
  </div>
</div>

```

Core Concepts

TODO: Add details for each **of** these:

- Locale data
- Formatters (Date, Number, Message, Relative)
- Provider and Injector
- API and Components
- Message Descriptor
- Message Syntax
- Defining **default** messages for extraction
- Custom, named formats

Homework

1. Create 1 Minimal Application with Friends of React.

Reference

- <https://css-tricks.com/react-router-4/>
- <http://www.material-ui.com/#/>
- <https://github.com/ReactTraining/react-router>
- <https://github.com/final-form/react-final-form>
- <https://github.com/acdlite/recompose>
- <https://github.com/yahoo/react-intl>
- **React: Functional Web Development with React and Redux 1st Edition, Kindle Edition**