30 DAYS OF REACT

AN INTRODUCTION TO REACT IN 30 BITE-SIZE MORSELS

FULLSTACK.io

What is React?

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-01/post.md)

Today, we're starting out at the beginning. Let's look at what React is and what makes it tick. We'll discuss why we want to use it.

Over the next 30 days, you'll get a good feel for the various parts of the React (https://facebook.github.io/react/) web framework and its ecosystem.

Each day in our 30 day adventure will build upon the previous day's materials, so by the end of the series, you'll not only know the terms, concepts, and underpinnings of how the framework works, but be able to use React in your next web application.

Let's get started. We'll start at the very beginning (https://www.youtube.com/watch?v=1RW3nDRmu6k) as it's a very good place to start.

What is React?

React (https://facebook.github.io/react/) is a JavaScript library for building user interfaces. It is the view layer for web applications.

At the heart of all React applications are **components**. A component is a self-contained module that renders some output. We can write interface elements like a button or an input field as a React component. Components are *composable*. A component might include one or more other components in its output.

Broadly speaking, to write React apps we write React components that correspond to various interface elements. We then organize these components inside higher-level components which define the structure of our application.

For example, take a form. A form might consist of many interface elements, like input fields, labels, or buttons. Each element inside the form can be written as a React component. We'd then write a higher-level component, the form component itself. The form component would specify the structure of the form and include each of these interface elements inside of it.

Importantly, each component in a React app abides by strict data management principles. Complex, interactive user interfaces often involve complex data and application state. The surface area of React is limited and aimed at giving us the tools to be able to anticipate how our application will look with a given set of circumstances. We dig into these principles later in the course.

Okay, so how do we use it?

React is a JavaScript framework. Using the framework is as simple as including a JavaScript file in our HTML and using the `React` exports in our application's JavaScript.

For instance, the *Hello world* example of a React website can be as simple as:

```html
<html>
<head>
  <meta charset="utf-8">
  <title>Hello world</title>
  <!-- Script tags including React -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react.min.js"
></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react-
dom.min.js"></script>
  <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
</head>
<body>
  <div id="app"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Hello world</h1>,
      document.querySelector('#app')
    );
  </script>
</body>
</html>
```

Although it might look a little scary, the JavaScript code is a single line that dynamically adds *Hello world* to the page. Note that we only needed to include a handful of JavaScript files to get everything working.

How does it work?

Unlike many of its predecessors, React operates not directly on the browser's Document Object Model (DOM) immediately, but on a **virtual DOM**. That is, rather than manipulating the `document` in a browser after changes to our data (which can be quite slow) it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React intelligently determines what changes to make to the actual browser's DOM.

The React Virtual DOM (https://facebook.github.io/react/docs/dom-differences.html) exists entirely in-memory and is a representation of the web browser's DOM. Because of this, when we write a React component, we're not writing directly to the DOM, but we're writing a virtual component that React will turn into the DOM.

In the next article, we'll look at what this means for us as we build our React components and jump into JSX and writing our first real components.

What is JSX?

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-02/post.md)

Now that we know what React is, let's take a look at a few terms and concepts that will come up throughout the rest of the series.

In our previous article, we looked at what React (https://facebook.github.io/react/) is and discussed at a high-level how it works. In this article, we're going to look at one part of the React ecosystem: ES6 and JSX.

JSX/ES5/ES6 WTF??!

In any cursory search on the Internet looking for React material, no doubt you have already run into the terms `JSX`, ES5, and ES6. These opaque acronyms can get confusing quickly.

ES5 (the `ES` stands for ECMAScript) is basically "regular JavaScript." The 5th update to JavaScript, ES5 was finalized in 2009. It has been supported by all major browsers for several years. Therefore, if you've written or seen any JavaScript in the recent past, chances are it was ES5.

ES6 is a new version of JavaScript that adds some nice syntactical and functional additions. It was finalized in 2015. ES6 is almost fully supported (http://kangax.github.io/compat-table/es6/) by all major browsers. But it will be some time until older versions of web browsers are phased out of use. For instance, Internet Explorer 11 does not support ES6, but has about 12% of the browser market share.

In order to reap the benefits of ES6 today, we have to do a few things to get it to work in as many browsers as we can:

1. We have to *transpile* our code so that a wider range of browsers understand our JavaScript. This means converting ES6 JavaScript into ES5 JavaScript.
2. We have to include a *shim* or *polyfill* that provides additional functionality added in ES6 that a browser may or may not have.

We'll see how we do this a bit later in the series.

> Most of the code we'll write in this series will be easily translatable to ES5. In cases where we use ES6, we'll introduce the feature at first and then walk through it.

As we'll see, all of our React components have a `render` function that specifies what the HTML output of our React component will be. **JavaScript eXtension**, or more commonly **JSX**, is a React extension that allows us to write JavaScript that *looks like* HTML.

> Although in previous paradigms it was viewed as a bad habit to include JavaScript and markup in the same place, it turns out that combining the view with the functionality makes reasoning about the view straight-forward.

To see what this means, imagine we had a React component that renders an `h1` HTML tag. JSX allows us to declare this element in a manner that closely resembles HTML:

```
class HelloWorld extends React.Component {
  render() {
    return (
      <h1 className='large'>Hello World</h1>
    );
  }
}
```

Hello World

The `render()` function in the `HelloWorld` component looks like it's returning HTML, but this is actually JSX. The JSX is *translated* to regular JavaScript at runtime. That component, after translation, looks like this:

```
class HelloWorld extends React.Component {
  render() {
    return (
      React.createElement(
        'h1',
        {className: 'large'},
        'Hello World'
      )
    );
  }
}
```

While JSX looks like HTML, it is actually just a terser way to write a `React.createElement()` declaration. When a component renders, it outputs a tree of React elements or a **virtual representation** of the HTML elements this component outputs. React will then determine what changes to make to the actual DOM based on this React element representation. In the case of the `HelloWorld` component, the HTML that React writes to the DOM will look like this:

```
<h1 class='large'>Hello World</h1>
```

> The `class extends` syntax we used in our first React component is ES6 syntax. It allows us to write objects using a familiar Object-Oriented style. In ES5, the `class` syntax might be translated as:
>
> ```
> var HelloWorld = function() {}
> Object.extends(HelloWorld, React.Component)
> HelloWorld.prototype.render = function() {}
> ```

Because JSX is JavaScript, we can't use JavaScript reserved words. This includes words like `class` and `for`.

React gives us the attribute `className`. We use it in `HelloWorld` to set the `large` class on our `h1` tag. There are a few other attributes, such as the `for` attribute on a label that React translates into `htmlFor` as `for` is also a reserved word. We'll look at these when we start using them.

If we want to write pure JavaScript instead of rely on a JSX compiler, we can just write the `React.createElement()` function and not worry about the layer of abstraction. But we like JSX. It's especially more readable with complex components. Consider the following JSX:

```
<div>
  <img src="profile.jpg" alt="Profile photo" />
  <h1>Welcome back Ari</h1>
</div>
```

The JavaScript delivered to the browser will look like this:

```
React.createElement("div", null,
  React.createElement("img", {src: "profile.jpg", alt: "Profile
photo"}),
  React.createElement("h1", null, "Welcome back Ari")
);
```

Again, while you can skip JSX and write the latter directly, the JSX syntax is well-suited for representing nested HTML elements.

Now that we understand JSX, we can start writing our first React components. Join us tomorrow when we jump into our first React app.

Our First Components

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-03/post.md)

The first two articles in this series were heavy on discussion. In today's session, let's dive into some code and write our first React app.

Let's revisit the "Hello world" app we introduced on day one. Here it is again, written slightly differently:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello world</title>
  <!-- Script tags including React -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react.min.js"
></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react-
dom.min.js"></script>
  <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
</head>
<body>
  <div id="app"></div>
  <script type="text/babel">
    var app = <h1>Hello world</h1>
    var mountComponent = document.querySelector('#app');
    ReactDOM.render(app, mountComponent);
  </script>
</body>
</html>
```

Hello world

Loading the React library

We've included the source of React as a `<script>` tag inside the `<head>` element of our page. It's important to place our `<script>` loading tags *before* we start writing our React application otherwise the `React` and `ReactDOM` variables won't be defined in time for us to use them.

Also inside `head` is a `script` tag that includes a library, `babel-core`. But what is `babel-core`?

Babel

Yesterday, we talked about ES5 and ES6. We mentioned that support for ES6 is still spotty. In order to use ES6, it's best if we transpile our ES6 JavaScript into ES5 JavaScript to support more browsers.

Babel is a library for transpiling ES6 to ES5.

Inside `body`, we have a `script` body. Inside of `script`, we define our first React application. Note that the `script` tag has a `type` of `text/babel`:

```
<script type="text/babel">
```

This signals to Babel that we would like it to handle the execution of the JavaScript inside this `script` body, this way we can write our React app using ES6 JavaScript and be assured that Babel will live-transpile its execution in browsers that only support ES5.

> # Warning in the console?
>
> When using the `babel-standalone` package, we'll get a warning in the console. This is fine and expected. We'll switch to a precompilation step in a few days.
>
> We've included the `<script />` tag here for ease of use.

The React app

Inside the Babel `script` body, we've defined our first React application. Our application consists of a single element, the `<h1>Hello world</h1>`. The call to `ReactDOM.render()` actually places our tiny React application on the page. Without the call to `ReactDOM.render()`, nothing would render in the DOM. The first argument to `ReactDOM.render()` is *what* to render and the second is *where*:

```
ReactDOM.render(<what>, <where>)
```

We've written a React application. Our "app" is a React element which represents an `h1` tag. But this isn't very interesting. Rich web applications accept user input, change their shape based on user interaction, and communicate with web servers. Let's begin touching on this power by building our first React component.

Components and more

We mentioned at the beginning of this series that at the heart of all React applications are *components*. The best way to understand React components is to write them. We'll write our React components as ES6 classes.

Let's look at a component we'll call `App`. Like all other React components, this ES6 class will extend the `React.Component` class from the React package:

```
class App extends React.Component {
  render() {
    return <h1>Hello from our app</h1>
  }
}
```

> All React components require at least a `render()` function. This `render()` function is expected to return a virtual DOM representation of the browser DOM element(s).

In our `index.html`, let's replace our JavaScript from before with our new `App` component.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello world</title>
  <!-- Script tags including React -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react.min.js"
></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.1/react-
dom.min.js"></script>
  <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
</head>
<body>
  <div id="app"></div>
  <script type="text/babel">
    class App extends React.Component {
      render() {
        return <h1>Hello from our app</h1>
      }
    }
  </script>
</body>
</html>
```

However, nothing is going to render on the screen. Do you remember why?

We haven't told React we want to render anything on the screen or *where* to render it. We need to use the `ReactDOM.render()` function again to express to React what we want rendered and where.

Adding the `ReactDOM.render()` function will render our application on screen:

```
var mount = document.querySelector('#app');
ReactDOM.render(<App />, mount);
```

Hello from our app

Notice that we can render our React app using the `App` class as though it is a built-in DOM component type (like the `<h1 />` and `<div />` tags). Here we're using it as though it's an element with the angle brackets: `<App />`.

The idea that our React components act just like any other element on our page allows us to build a component tree **just as if we were creating a native browser tree**.

While we're rendering a React component now, our app still lacks richness or interactivity. Soon, we'll see how to make React components data-driven and dynamic.

But first, in the next installment of this series, we'll explore how we can layer components. Nested components are the foundation of a rich React web application.

Complex Components

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-04/post.md)

Awesome, we've built our first component. Now let's get a bit fancier and start building a more complex interface.

In the previous section of 30 *Days of React*, we started building our first React component. In this section, we'll continue our work with our `App` component and start building a more complex UI.

A common web element we might see is a user timeline. For instance, we might have an application that shows a history of events happening such as applications like Facebook and Twitter.

Styles

As we're not focusing on CSS (https://www.w3.org/standards/webdesign/htmlcss) in this course, we're not covering the CSS specific to build the timeline as you see it on the screen.

However, we want to make sure the timeline you build looks similar to ours. If you include the following CSS as a `<link />` tag in your code, your timeline will look similar and will be using the same styling ours is using:

```
<link
  href="https://cdn.jsdelivr.net/gh/fullstackreact/30-days-of-react@master/day-04/src/components/Timeline/Timeline.css"
  rel="stylesheet"
  type="text/css"
/>
```

And make sure to surround your code in a component with the class of `demo` (we left it this way purposefully as it's the *exact* same code we use in all the demos here). Check out the https://jsfiddle.net/auser/zwomnfwk/ (https://jsfiddle.net/auser/zwomnfwk/) for a working example.

The entire compiled CSS can be found on the github repository at https://github.com/fullstackreact/30-days-of-react/blob/master/day-04/src/components/Timeline/Timeline.css (https://github.com/fullstackreact/30-days-of-react/blob/master/day-04/src/components/Timeline/Timeline.css).

In addition, in order to make the timeline look *exactly* like the way ours does on the site, you'll need to include font-awesome (http://fontawesome.io/) in your web application. There are multiple ways to handle this. The simplest way is to include the link styles:

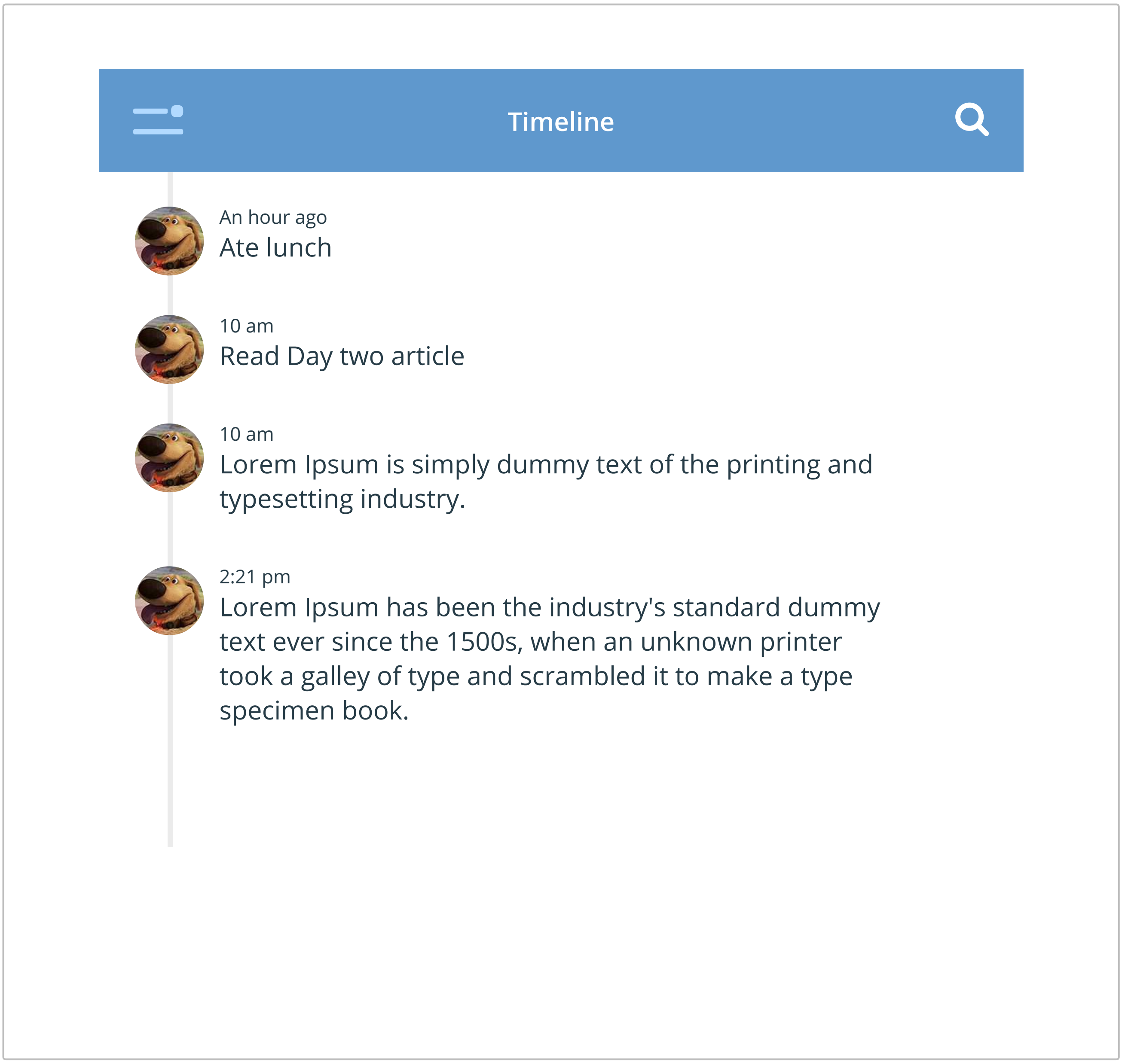
```html
<link
  href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"
  rel="stylesheet"
  type="text/css"
/>
```

All the code for the examples on the page is available at the github repo (at https://github.com/fullstackreact/30-days-of-react) (https://github.com/fullstackreact/30-days-of-react).

We *could* build this entire UI in a single component. However, building an entire application in a single component is not a great idea as it can grow huge, complex, and difficult to test.

```
class Timeline extends React.Component {
  render() {
    return (
      <div className="notificationsFrame">
        <div className="panel">
          <div className="header">

            <div className="menuIcon">
              <div className="dashTop"></div>
              <div className="dashBottom"></div>
              <div className="circle"></div>
            </div>

            <span className="title">Timeline</span>

            <input
              type="text"
              className="searchInput"
              placeholder="Search ..." />

            <div className="fa fa-search searchIcon"></div>
          </div>
          <div className="content">
            <div className="line"></div>
            <div className="item">

              <div className="avatar">
                <img
                alt='doug'
                src="http://www.croop.cl/UI/twitter/images/doug.jpg"
/>
              </div>

              <span className="time">
                An hour ago
              </span>
              <p>Ate lunch</p>
            </div>

            <div className="item">
              <div className="avatar">
                <img
```

```
                  alt='doug'
src="http://www.croop.cl/UI/twitter/images/doug.jpg" />
              </div>

              <span className="time">10 am</span>
              <p>Read Day two article</p>
            </div>

            <div className="item">
              <div className="avatar">
                <img
                  alt='doug'
src="http://www.croop.cl/UI/twitter/images/doug.jpg" />
              </div>

              <span className="time">10 am</span>
              <p>Lorem Ipsum is simply dummy text of the printing and
typesetting industry.</p>
            </div>

            <div className="item">
              <div className="avatar">
                <img
                  alt='doug'
src="http://www.croop.cl/UI/twitter/images/doug.jpg" />
              </div>

              <span className="time">2:21 pm</span>
              <p>Lorem Ipsum has been the industry's standard dummy
text ever since the 1500s, when an unknown printer took a galley of
type and scrambled it to make a type specimen book.</p>
            </div>

          </div>
        </div>
      </div>
    )
  }
}
```

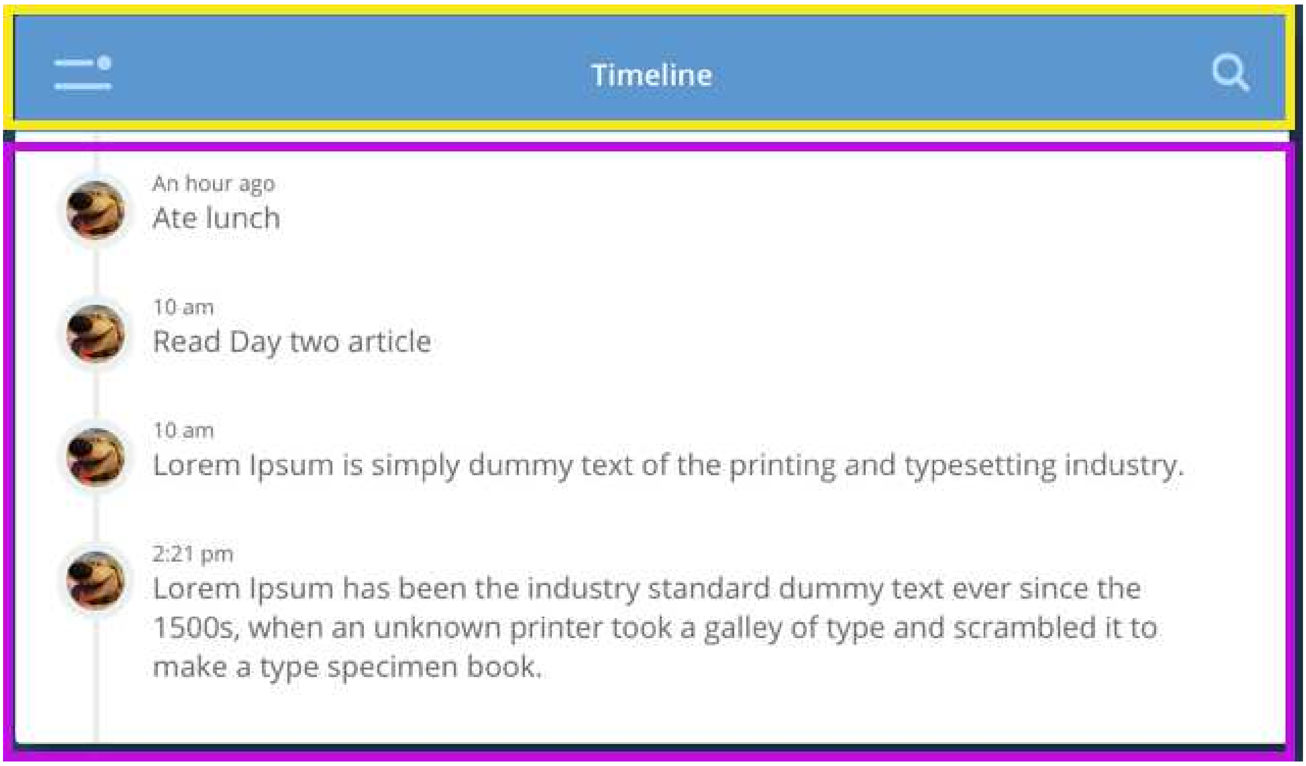


Breaking it down

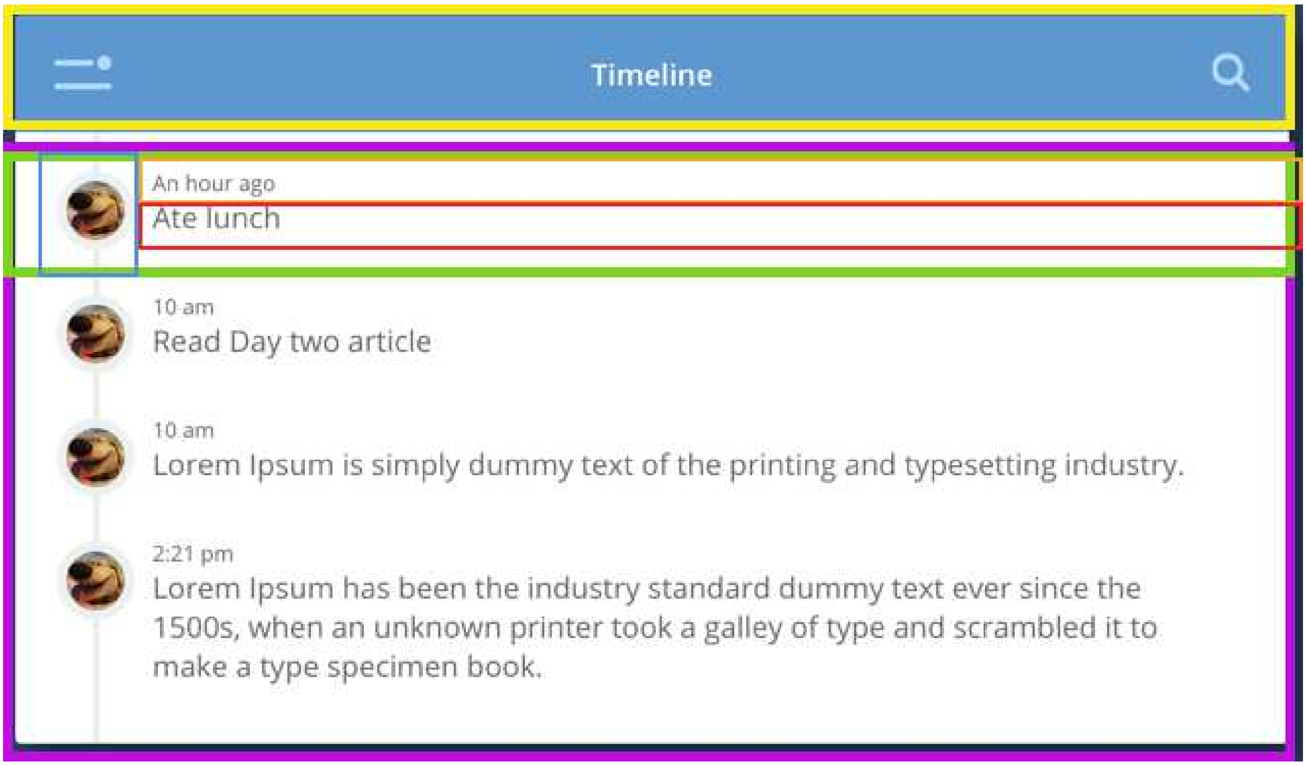
Rather than build this in a single component, let's break it down into multiple components.

Looking at this component, there are 2 separate parts to the larger component as a whole:

1. The title bar
2. The content



We can chop up the content part of the component into individual places of concern. There are 3 different *item* components inside the content part.



> If we wanted to go one step further, we could even break down the title bar into 3 component parts, the *menu* button, the *title*, and the *search* icon. We could dive even further into each one of those if we needed to.
>
> Deciding how deep to split your components is more of an art than a science and is a skill you'll develop with experience.

In any case, it's usually a good idea to start looking at applications using the idea of *components*. By breaking our app down into components it becomes easier to test and easier to keep track of what functionality goes where.

The container component

To build our notifications app, let's start by building the container to hold the entire app. Our container is simply going to be a wrapper for the other two components.

None of these components will require special functionality (yet), so they will look similar to our `HelloWorld` component in that it's just a component with a single render function.

Let's build a wrapper component we'll call `App` that might look similar to this:

```
class App extends React.Component {
  render() {
    return (
      <div className="notificationsFrame">
        <div className="panel">{/* content goes here */}</div>
      </div>
    );
  }
}
```

> Notice that we use the attribute called `className` in React instead of the HTML version of `class`. Remember that we're not writing to the DOM directly and thus not writing HTML, but JSX (which is just JavaScript).
>
> The reason we use `className` is because `class` is a reserved word in JavaScript. If we use `class`, we'll get an error in our console.

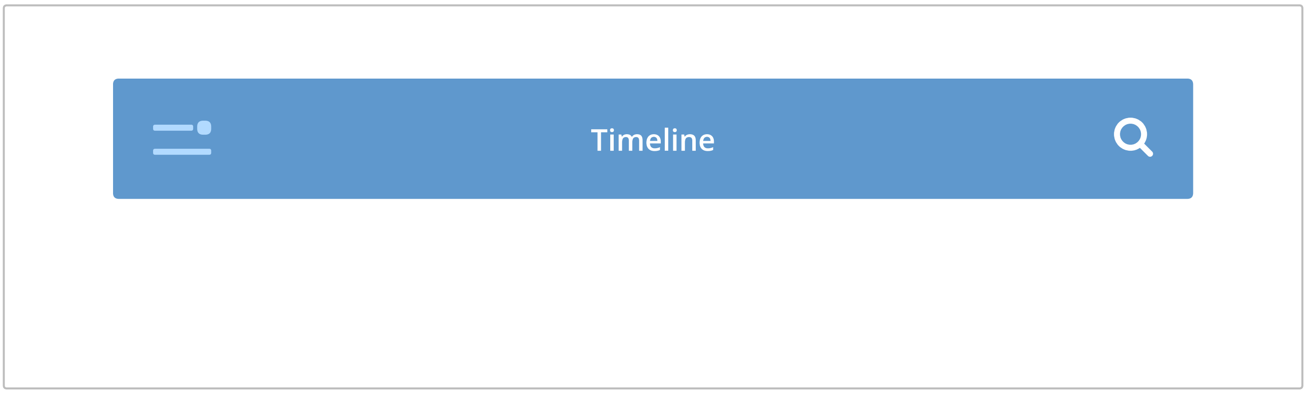
Child components

When a component is nested inside another component, it's called a *child* component. A component can have multiple children components. The component that uses a child component is then called it's *parent* component.

With the wrapper component defined, we can build our `title` and `content` components by, essentially, grabbing the source from our original design and putting the source file into each component.

For instance, the header component looks like this, with a container element `<div className="header">`, the menu icon, a title, and the search bar:

```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>
        <span className="title">Timeline</span>

        <input type="text" className="searchInput" placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    );
  }
}
```



And finally, we can write the `Content` component with timeline items. Each timeline item is wrapped in a single component, has an avatar associated with it, a timestamp, and some text.

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        <div className="line"></div>

        {/* Timeline item */}
        <div className="item">
          <div className="avatar">
            <img
              alt="Doug"
              src="http://www.croop.cl/UI/twitter/images/doug.jpg"
            />
            Doug
          </div>

          <span className="time">An hour ago</span>
          <p>Ate lunch</p>
          <div className="commentCount">2</div>
        </div>

        {/* ... */}
      </div>
    );
  }
}
```

In order to write a comment in a React component, we have to place it in the brackets as a multi-line comment in JavaScript.

Unlike the HTML comment that looks like this:

```
<!-- this is a comment in HTML -->
```

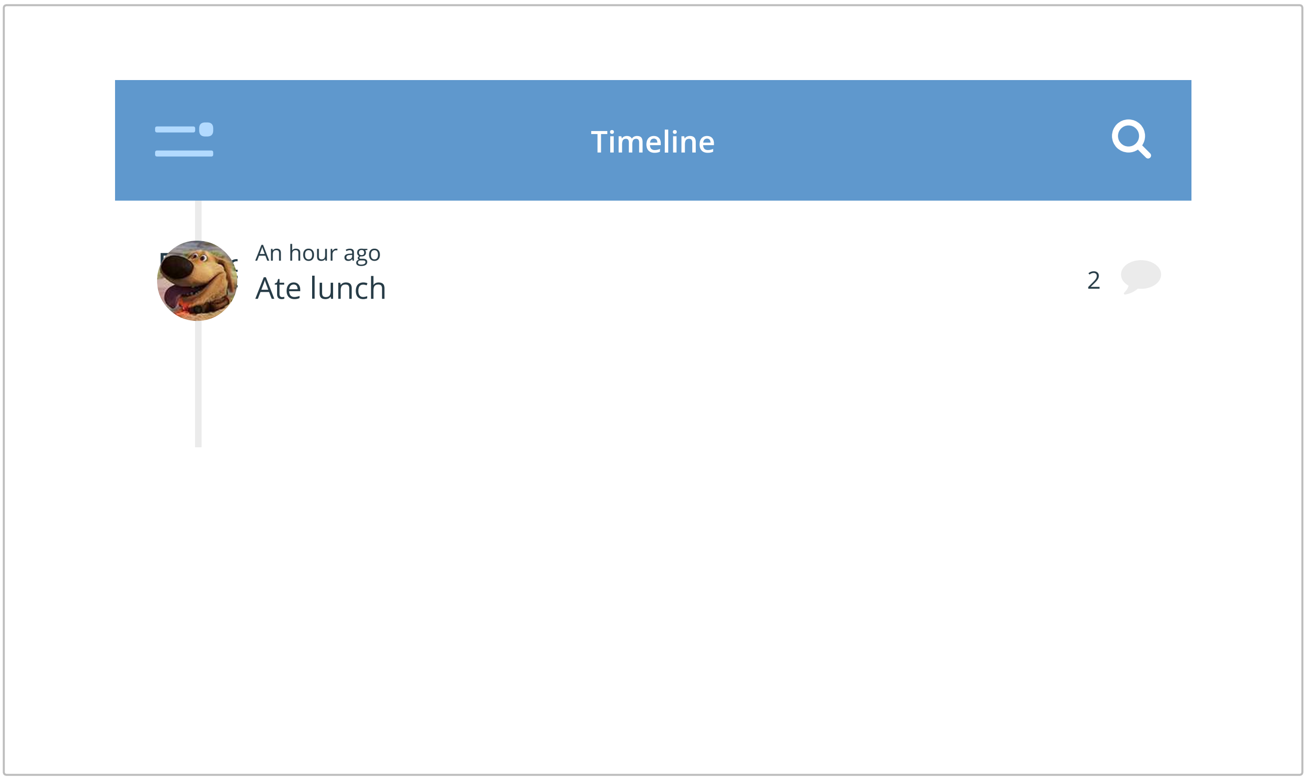
the React version of the comment must be in brackets:

```
{/* This is a comment in React */}
```

Putting it all together

Now that we have our two *children* components, we can set the `Header` and the `Content` components to be *children* of the `App` component. Our `App` component can then use these components *as if they are HTML elements built-in to the browser*. Our new `App` component, with a header and content now looks like:

```
class App extends React.Component {
  render() {
    return (
      <div className="notificationsFrame">
        <div className="panel">
          <Header />
          <Content />
        </div>
      </div>
    );
  }
}
```



Note!

Don't forget to call `ReactDOM.render` to place your app on the page

```
var mountComponent = document.querySelector("#app");
ReactDOM.render(<App />, mountComponent);
```

With this knowledge, we now have the ability to write multiple components and we can start to build more complex applications.

However, you may notice that this app does not have any user interaction nor custom data. In fact, as it stands right now our React application isn't that much easier to build than straight, no-frills HTML.

In the next section, we'll look how to make our component more dynamic and become *data-driven* with React.

Data-Driven

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-05/post.md)

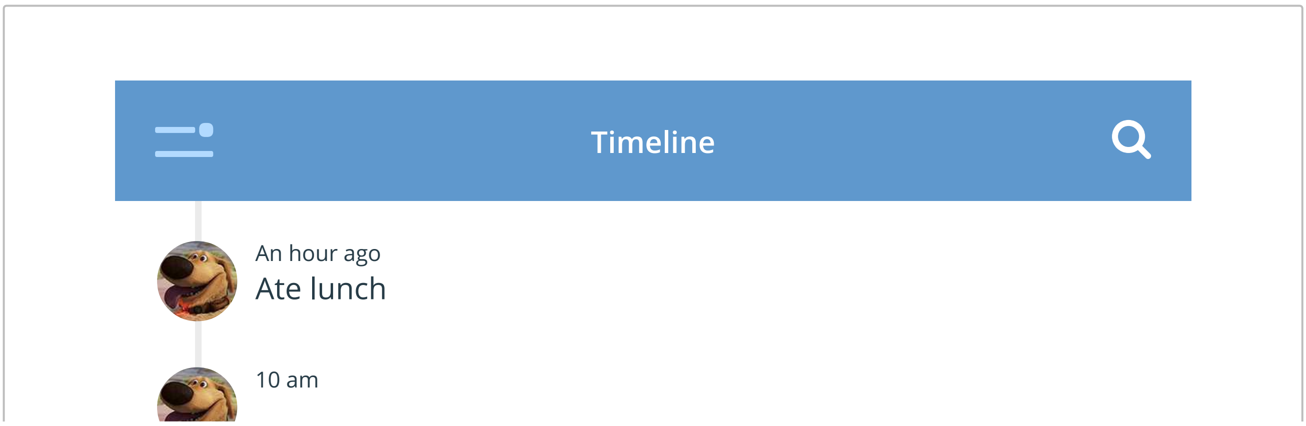
Hard-coding data in our applications isn't exactly ideal. Today, we'll set up our components to be driven by data to them access to external data.

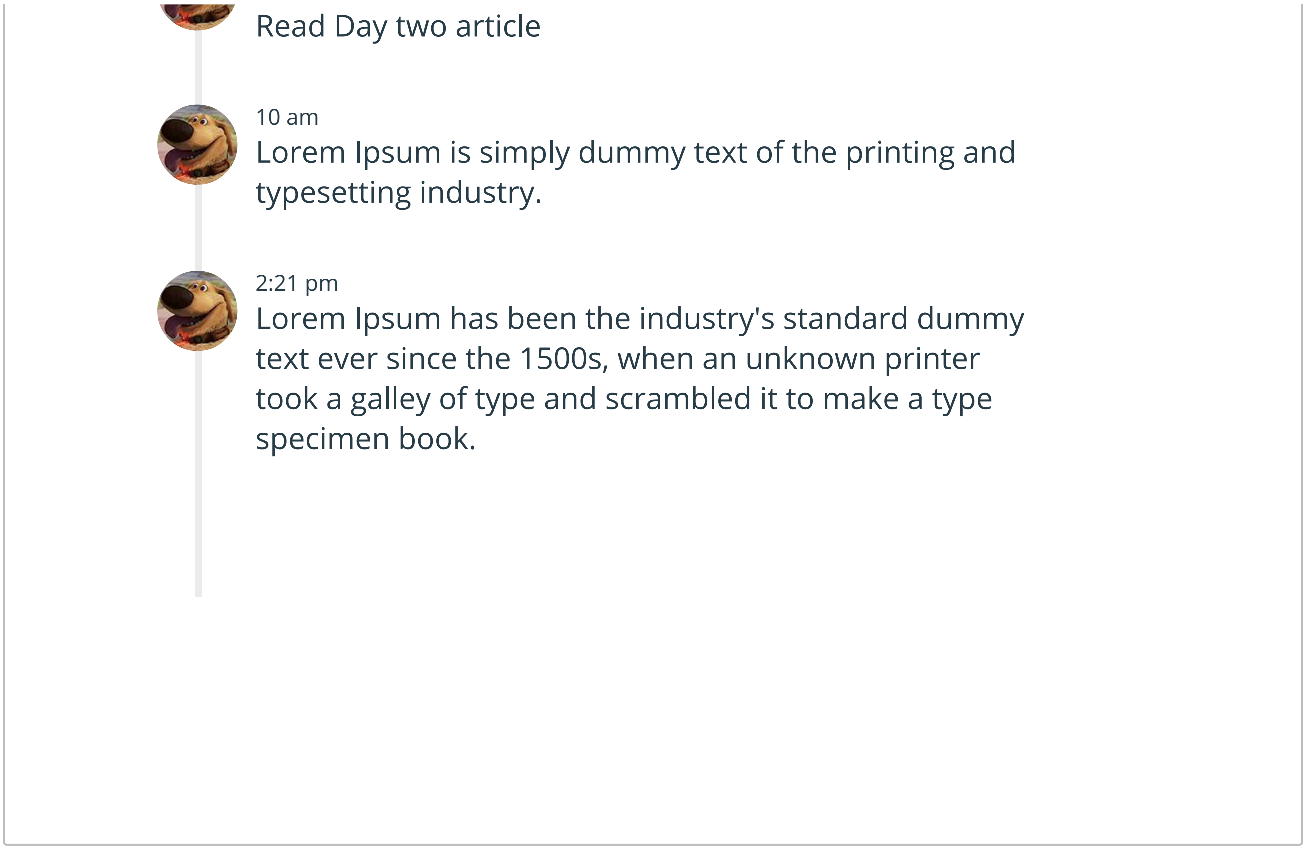
Through this point, we've written our first components and set them up in a child/parent relationship. However, we haven't yet tied any data to our React components. Although it's a more pleasant experience (in our opinion) writing a website in React, we haven't taken advantage of the power of React to display any dynamic data.

Let's change that today.

Going data-driven

Recall, yesterday we built the beginning of our timeline component that includes a header and an activity list:





We broke down our demo into components and ended up building three separate components with static JSX templates. It's not very convenient to have to update our component's template everytime we have a change in our website's data.

Instead, let's give the components data to use to display. Let's start with the `<Header />` component. As it stands right now, the `<Header />` component only shows the title of the element as `Timeline`. It's a nice element and it would be nice to be able to reuse it in other parts of our page, but the title of `Timeline` doesn't make sense for every use.

Let's tell React that we want to be able to set the title to something else.

Introducing props

React allows us to send data to a component in the same syntax as HTML, using attributes or *properties* on a component. This is akin to passing the `src` attribute to an image tag. We can think about the property of the `<img />` tag as a `prop` we're setting on a component called `img`.

We can access these properties inside a component as `this.props`. Let's see `props` in action.

Recall, we defined the `<Header />` component as:

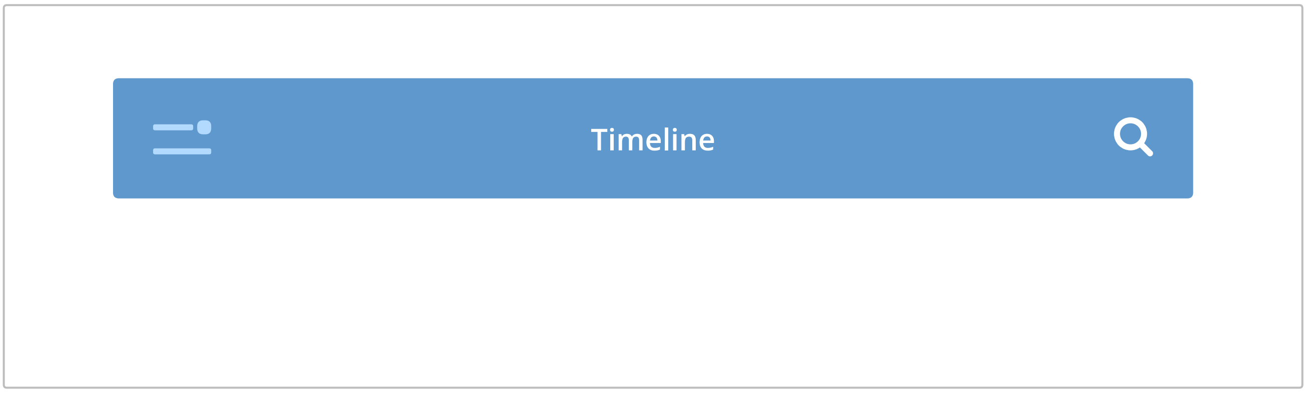
```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>
        <span className="title">Timeline</span>

        <input
          type="text"
          className="searchInput"
          placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

When we use the `<Header />` component, we placed it in our `<App />` component as like so:

```
<Header />
```

We can pass in our `title` as a prop as an attribute on the `<Header />` by updating the usage of the component setting the attribute called `title` to some string, like so:

```
<Header title="Timeline" />
```



Inside of our component, we can access this `title` prop from the `this.props` property in the `Header` class. Instead of setting the title statically as `Timeline` in the template, we can replace it with the property passed in.

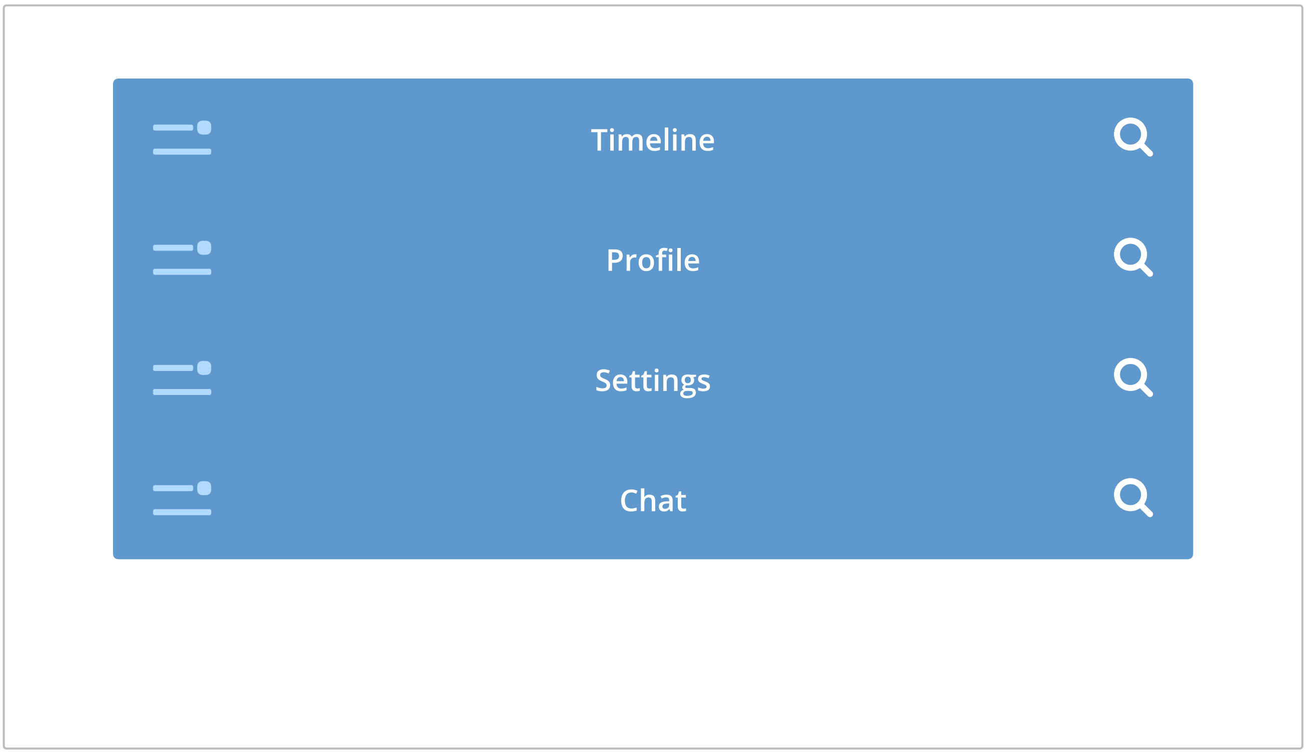
```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className="searchInput"
          placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

> We've also updated the code slightly to get closer to what our final `<Header />` code will look like, including adding a `searchIcon` and a few elements to style the `menuIcon`.

Now our `<Header />` component will display the string we pass in as the `title` when we call the component. For instance, calling our `<Header />` component four times like so:

```
<Header title="Timeline" />
<Header title="Profile" />
<Header title="Settings" />
<Header title="Chat" />
```

Results in four `<Header />` components to mount like so:



Pretty nifty, ey? Now we can reuse the `<Header />` component with a dynamic `title` property.

We can pass in more than just strings in a component. We can pass in numbers, strings, all sorts of objects, and even functions! We'll talk more about how to define these different properties so we can build a component api later.

Instead of statically setting the content and date let's take the `Content` component and set the timeline content by a data variable instead of by text. Just like we can do with HTML components, we can pass multiple `props` into a component.

Recall, yesterday we defined our `Content` container like this:

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        <div className="line"></div>

      {/* Timeline item */}
        <div className="item">
          <div className="avatar">
            <img src="http://www.croop.cl/UI/twitter/images/doug.jpg"
/>
            Doug
          </div>

          <span className="time">
            An hour ago
          </span>
          <p>Ate lunch</p>
          <div className="commentCount">
            2
          </div>
        </div>

        {/* ... */}

      </div>
    )
  }
}
```

As we did with `title`, let's look at what `props` our `Content` component needs:

- A user's avatar image
- A timestamp of the activity
- Text of the activity item
- Number of comments

Let's say that we have a JavaScript object that represents an activity item. We will have a few fields, such as a string field (text) and a date object. We might have some nested objects, like a `user` and `comments`. For instance:

```
{
  timestamp: new Date().getTime(),
  text: "Ate lunch",
  user: {
    id: 1,
    name: 'Nate',
    avatar: "http://www.croop.cl/UI/twitter/images/doug.jpg"
  },
  comments: [
    { from: 'Ari', text: 'Me too!' }
  ]
}
```

Just like we passed in a string title to the `<Header />` component, we can take this activity object and pass it right into the `Content` component. Let's convert our component to display the details from this activity inside it's template.

In order to pass a dynamic variable's value into a template, we have to use the template syntax to render it in our template. For instance:

```
class Content extends React.Component {
  render() {
    const {activity} = this.props; // ES6 destructuring

    return (
      <div className="content">
        <div className="line"></div>

        {/* Timeline item */}
        <div className="item">
          <div className="avatar">
            <img
              alt={activity.text}
              src={activity.user.avatar} />
            {activity.user.name}
          </div>

          <span className="time">
            {activity.timestamp}
          </span>
          <p>{activity.text}</p>
          <div className="commentCount">
            {activity.comments.length}
          </div>
        </div>
      </div>
    )
  }
}
```

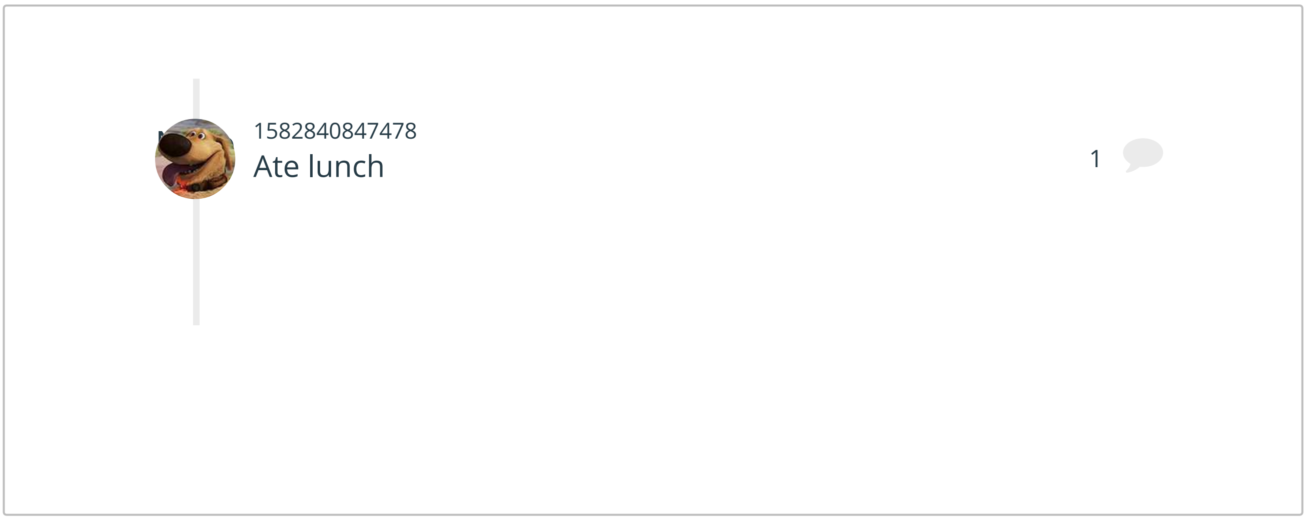
We've use a little bit of ES6 in our class definition on the first line of the `render()` function called *destructuring*. The two following lines are functionally equivalent:

```
// these lines do the same thing
const activity = this.props.activity;
const {activity} = this.props;
```

Destructuring allows us to save on typing and define variables in a shorter, more compact way.

We can then *use* this new content by passing in an object as a prop instead of a hard-coded string. For instance:

```
<Content activity={moment1} />
```



Fantastic, now we have our activity item driven by an object. However, you might have noticed that we would have to implement this multiple times with different comments. Instead, we could pass an array of objects into a component.

Let's say we have an object that contains multiple activity items:

```
const activities = [
  {
    timestamp: new Date().getTime(),
    text: "Ate lunch",
    user: {
      id: 1, name: 'Nate',
      avatar: "http://www.croop.cl/UI/twitter/images/doug.jpg"
    },
    comments: [{ from: 'Ari', text: 'Me too!' }]
  },
  {
    timestamp: new Date().getTime(),
    text: "Woke up early for a beautiful run",
    user: {
      id: 2, name: 'Ari',
      avatar: "http://www.croop.cl/UI/twitter/images/doug.jpg"
    },
    comments: [{ from: 'Nate', text: 'I am so jealous' }]
  },
]
```

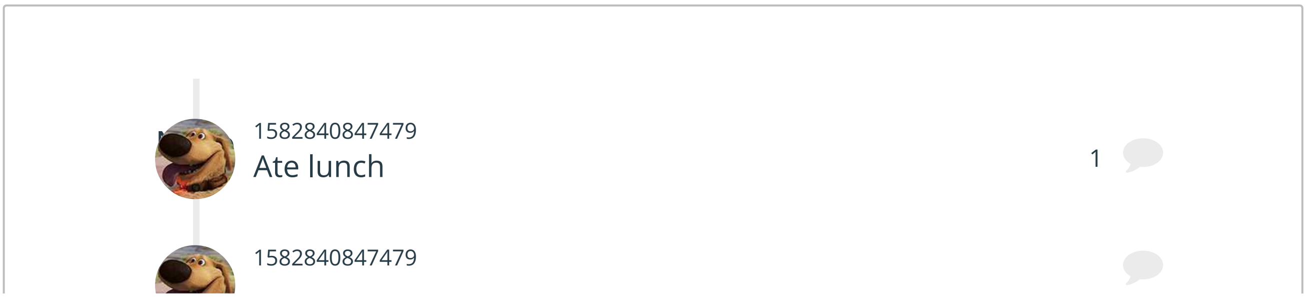
We can rearticulate our usage of `<Content />` by passing in multiple activities instead of just one:

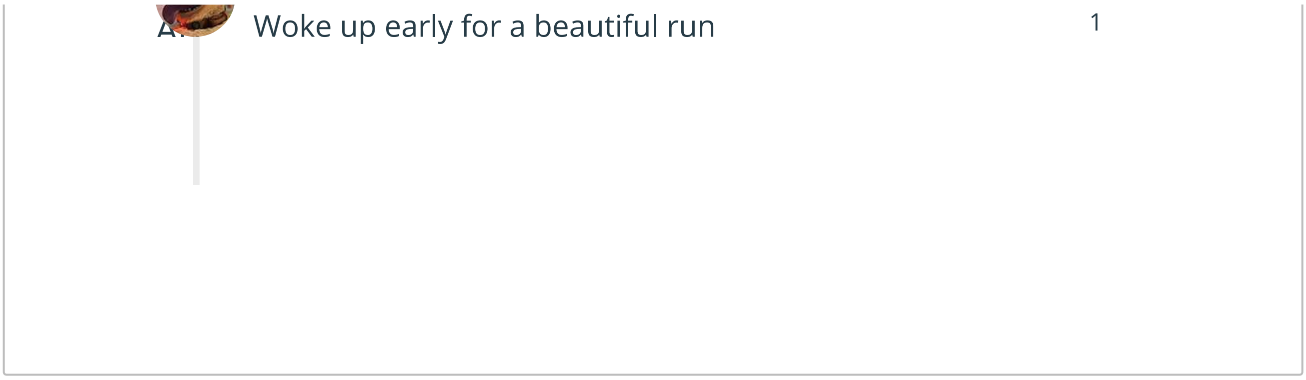
```
<Content activities={activities} />
```

However, if we refresh the view nothing will show up! We need to first update our `Content` component to accept multiple activities. As we learned about previously, JSX is really *just* JavaScript executed by the browser. We can execute JavaScript functions inside the JSX content as it will just get run by the browser like the rest of our JavaScript.

Let's move our activity item JSX inside of the function of the `map` function that we'll run over for every item.

```
class Content extends React.Component {
  render() {
    const {activities} = this.props; // ES6 destructuring

    return (
      <div className="content">
        <div className="line"></div>

        {/* Timeline item */}
        {activities.map((activity) => {
          return (
            <div className="item">
              <div className="avatar">
                <img
                  alt={activity.text}
                  src={activity.user.avatar} />
                {activity.user.name}
              </div>

              <span className="time">
                {activity.timestamp}
              </span>
              <p>{activity.text}</p>
              <div className="commentCount">
                {activity.comments.length}
              </div>
            </div>
          );
        })}

      </div>
    )
  }
}
```





Now we can pass any number of activities to our array and the `Content` component will handle it, however if we leave the component right now, then we'll have a relatively complex component handling both containing and displaying a list of activities. Leaving it like this really isn't the React way.

ActivityItem

Here is where it makes sense to write one more component to contain displaying a single activity item and then rather than building a complex `Content` component, we can move the responsibility. This will also make it easier to test, add functionality, etc.

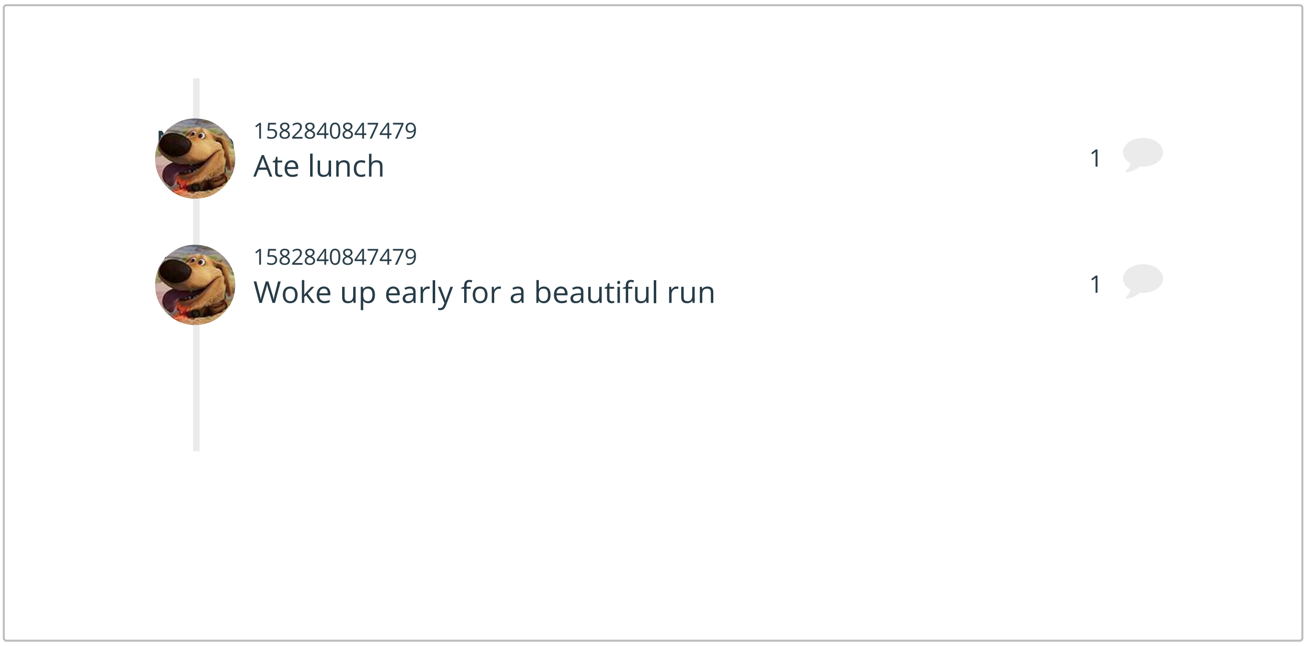
Let's update our `Content` component to display a list of `ActivityItem` components (we'll create this next).

```
class Content extends React.Component {
  render() {
    const {activities} = this.props; // ES6 destructuring

    return (
      <div className="content">
        <div className="line"></div>

        {/* Timeline item */}
        {activities.map((activity) => (
          <ActivityItem
            activity={activity} />
        ))}

      </div>
    )
  }
}
```

Not only is this much simpler and easier to understand, but it makes testing both components easier.

With our freshly-minted `Content` component, let's create the `ActivityItem` component. Since we already have the view created for the `ActivityItem`, all we need to do is copy it from what was our `Content` component's template as it's own module.

```
class ActivityItem extends React.Component {
  render() {
    const {activity} = this.props; // ES6 destructuring

    return (
      <div className="item">
        <div className="avatar">
          <img
            alt={activity.text}
            src={activity.user.avatar} />
          {activity.user.name}
        </div>

        <span className="time">
          {activity.timestamp}
        </span>
        <p>{activity.text}</p>
        <div className="commentCount">
          {activity.comments.length}
        </div>
      </div>
    )
  }
}
```



This week we updated our components to be driven by data by using the React `props` concept. In the next section, we'll dive into stateful components.

State

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-06/post.md)

Today we're getting started on how stateful components work in React and look at when and why we'll use state.

We've almost made it through the first week of getting up and running on React. We have worked through JSX, building our first components, setting up parent-child relationships, and driving our component properties with React. We have one more major idea we have yet to discuss about React, the idea of *state*.

The `state` of things

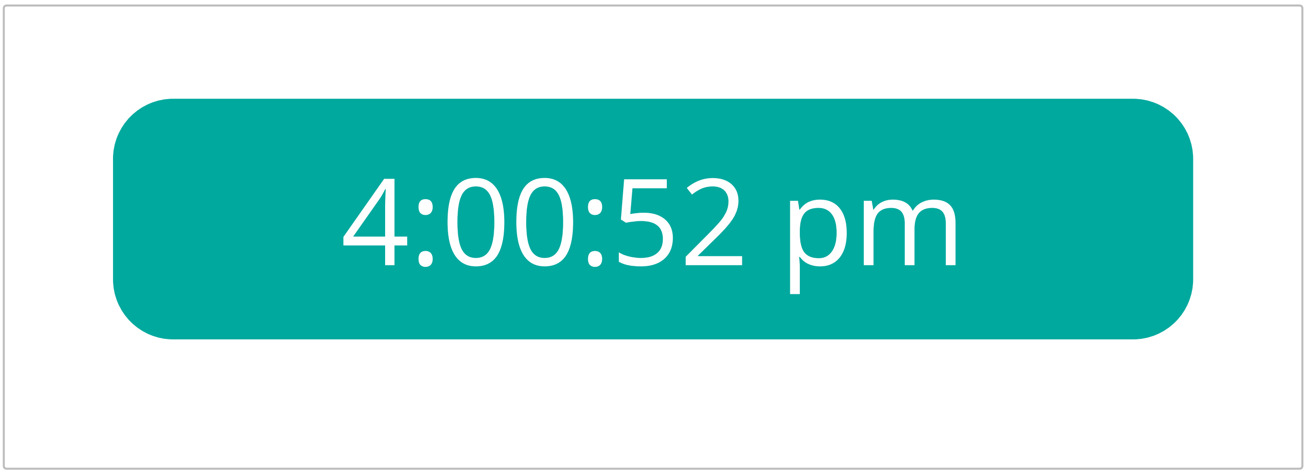
React does *not* allow us to modify `this.props` on our components for good reason. Imagine if we passed in the `title` prop to the `Header` component and the `Header` component was able to modify it. How do we know what the `title` is of the `Header` component? We set ourselves up for race-conditions, confusing data state, and it would be an all-around bad idea to modify a variable passed to a child component by a parent component.

However, sometimes a component needs to be able to update its own state. For example, setting an `active` flag or updating a timer on a stopwatch, for instance.

While it's preferable to use `props` as much as we can, sometimes we need to hold on to the state of a component. To handle this, React gives us the ability to hold *state* in our components.

`state` in a component is intended to be completely internal to the Component and its children (i.e. accessed by the component and any children it used). Similar to how we access `props` in a component, the state can be accessed via `this.state` in a component. Whenever the state changes (via the `this.setState()` function), the component will rerender.

For instance, let's say we have a simple clock component that shows the current time:



Even though this is a simple clock component, it does retain state in that it needs to know what the current time is to display. Without using `state`, we could set a timer and rerender the entire React component, but other components on the page may not need rerendering... this would become a headache and slow when we integrate it into a more complex application.

Instead, we can set a timer to call rerender *inside* the component and change just the *internal* state of this component.

Let's take a stab at building this component. First, we'll create the component we'll call `Clock`.

Before we get into the state, let's build the component and create the `render()` function. We'll need to take into account the number and prepend a zero (`0`) to the number if the numbers are smaller than 10 and set the `am/pm` appropriately. The end result of the `render()` function might look something like this:

```
class Clock extends React.Component {
  render() {
    const currentTime = new Date(),
          hours = currentTime.getHours(),
          minutes = currentTime.getMinutes(),
          seconds = currentTime.getSeconds(),
          ampm = hours >= 12 ? 'pm' : 'am';

    return (
      <div className="clock">
        {
          hours == 0 ? 12 :
            (hours > 12) ?
              hours - 12 : hours
        }:{
          minutes > 9 ? minutes : `0${minutes}`
        }:{
          seconds > 9 ? seconds : `0${seconds}`
        } {ampm}
      </div>
    )
  }
}
```

Alternative padding technique

Alternatively, we could use the short snippet to handle padding the clock time:

```
("00" + minutes).slice(-2)
```

But we've opted to be more clear with the previous code.

If we render our new `Clock` component, we will only get a time rendered everytime the component itself rerenders. It's not a very useful clock (yet). In order to convert our static time display `Clock` component into a clock that displays the time, we'll need to update the time every second.

In order to do that, we'll need to track the *current* time in the state of the component. To do this, we'll need to set an initial state value.

To do so, we'll first create a `getTime()` function that returns a javascript object containing `hours`, `minutes`, `seconds` and `ampm` values. We will call this function to set our state.

```
class Clock extends React.Component {
  //...
  getTime() {
    const currentTime = new Date();
    return {
      hours: currentTime.getHours(),
      minutes: currentTime.getMinutes(),
      seconds: currentTime.getSeconds(),
      ampm: currentTime.getHours() >= 12 ? 'pm' : 'am'
    }
  }
  // ...
}
```

In the ES6 class style, we can set the initial state of the component in the `constructor()` by setting `this.state` to a value (the return value of our `getTime()` function).

```
  constructor(props) {
    super(props);
    this.state = this.getTime();
  }
```

`this.state` will now look like the following object

```
{
  hours: 11,
  minutes: 8,
  seconds: 11,
  ampm: "am"
}
```

> The first line of the constructor should *always* call `super(props)`. If you forget this, the component won't like you very much (i.e. there will be errors).

Now that we have a `this.state` defined in our `Clock` component, we can reference it in the `render()` function using the `this.state`. Let's update our `render()` function to grab the values from `this.state`:

```
class Clock extends React.Component {
  // ...
  render() {
    const {hours, minutes, seconds, ampm} = this.state;
    return (
      <div className="clock">
        {
          hours === 0 ? 12 :
            (hours > 12) ?
              hours - 12 : hours
        }:{
          minutes > 9 ? minutes : `0${minutes}`
        }:{
          seconds > 9 ? seconds : `0${seconds}`
        } {ampm}
      </div>
    )
  }
}
```

Instead of working directly with data values, we can now update the `state` of the component and separate the `render()` function from the data management.

In order to update the state, we'll use a special function called: `setState()`, which will trigger the component to rerender.

> We need to call `setState()` on the `this` value of the component as it's a part of the `React.Component` class we are subclassing.

In our `Clock` component, let's use the native `setTimeout()` JavaScript function to create a timer to update the `this.state` object in 1000 milliseconds. We'll place this functionality in a function as we'll want to call this again.

```
class Clock extends React.Component {
  // ...
  constructor(props) {
    super(props);
    this.state = this.getTime();
  }
  // ...
  componentDidMount() {
    this.setTimer();
  }
  // ...
  setTimer() {
    clearTimeout(this.timeout);
    this.timeout = setTimeout(this.updateClock.bind(this), 1000);
  }
  // ...
  updateClock() {
    this.setState(this.getTime, this.setTimer);
  }
  // ...
}
```

> To start updating the timer immediately after the our component has been rendered, we call `this.setTimer()` in a React component lifecycle method called `componentDidMount`.We will get into the lifecycle hooks in the next section.

In the `updateClock()` function we'll want to update the state with the new time. We can now update the state in the `updateClock()` function:

```
class Clock extends React.Component {
  // ...
  updateClock() {
    this.setState(this.getTime, this.setTimer);
  }
  // ...
}
```

The component will be mounted on the page and will update the time every second (approximately every 1000 milliseconds)

Now the component itself might rerender slower than the timeout function gets called again, which would cause a rerendering bottleneck and needlessly using up precious battery on mobile devices. Instead of calling the `setTimer()` function after we call `this.setState()`, we can pass a second argument to the `this.setState()` function which will be guaranteed to be called *after* the state has been updated.

```
class Clock extends React.Component {
  // ...
  updateClock() {
    const currentTime = new Date();
    this.setState({
      currentTime: currentTime
    }, this.setTimer);
  }
  // ...
}
```

Here is our full `Clock` component code.

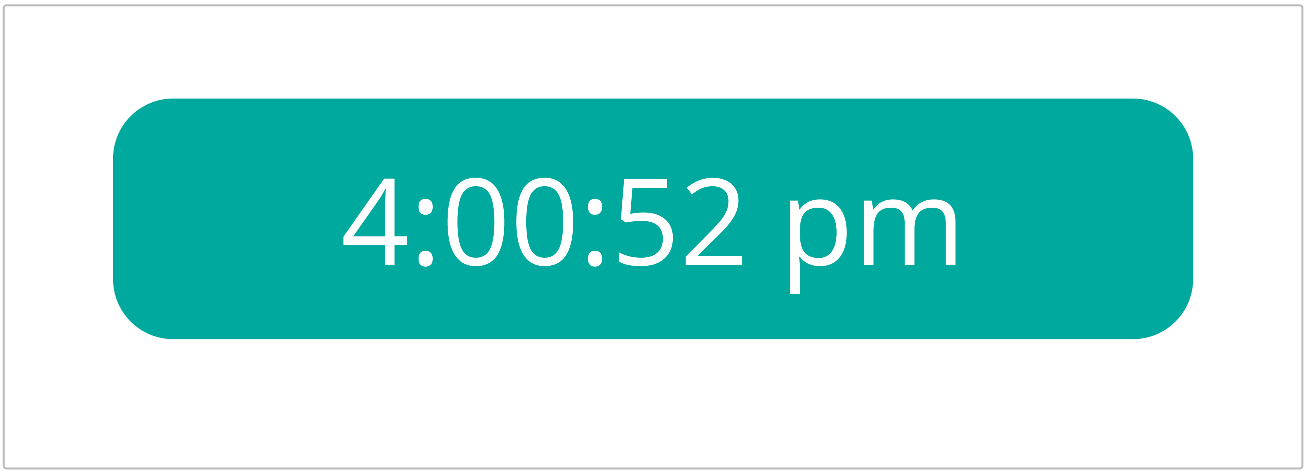
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = this.getTime();
  }

  componentDidMount() {
    this.setTimer();
  }

  setTimer() {
    clearTimeout(this.timeout);
    this.timeout = setTimeout(this.updateClock.bind(this), 1000);
  }

  updateClock() {
    this.setState(this.getTime, this.setTimer);
  }

  getTime() {
    const currentTime = new Date();
    return {
      hours: currentTime.getHours(),
      minutes: currentTime.getMinutes(),
      seconds: currentTime.getSeconds(),
      ampm: currentTime.getHours() >= 12 ? 'pm' : 'am'
    }
  }

  render() {
    const {hours, minutes, seconds, ampm} = this.state;
    return (
      <div className="clock">
        {hours == 0 ? 12 : hours > 12 ? hours - 12 : hours}:
        {minutes > 9 ? minutes : `0${minutes}`}:
        {seconds > 9 ? seconds : `0${seconds}`} {ampm}
      </div>
    );
  }
}
```

Styles

As we're not focusing on CSS (https://www.w3.org/standards/webdesign/htmlcss) in this course, we're not covering the CSS specific to build the clock as you see it on the screen.

However, we want to make sure the clock you build looks similar to ours. If you include the following CSS as a `<link />` tag in your code, your clock will look similar and will be using the same styling ours is using:

```
<link href="https://cdn.jsdelivr.net/gh/fullstackreact/30-days-of-react@master/day-06/public/Clock.css" rel="stylesheet" type="text/css" />
```



Some things to keep in mind

- When we call `this.setState()` with an object argument, it will perform a *shallow merge* of the data into the object available via `this.state` and then will rerender the component.
- We generally only want to keep values in our state that we'll use in the `render()` function. From the example above with our clock, notice that we stored the `hours`, `minutes`, and `seconds` in our state. It's usually a

> bad idea to store objects or calculations in the state that we don't plan on using in the `render` function as it can cause unnecessary rendering and wasteful CPU cycles.

As we noted at the top of this section, it's preferred to use `props` when available not only for performance reasons, but because stateful components are more difficult to test.

Today, we've updated our components to be stateful and now have a handle on how to make a component stateful when necessary. Tomorrow we'll dive into the lifecycle of a component and when/how to interact with the page.

Lifecycle Hooks

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-07/post.md)

> **NOTE**: This post is about *classic* React Lifecycle hooks. If you're looking to learn about the **new Hooks API** then click here (https://www.fullstackreact.com/articles/an-introduction-to-hooks-in-react/)

Today, we'll look through a few of the most common lifecycle hooks we can use with React components and we'll discuss why they are useful and when we should each one.

Congrats! We've made it to the end of the first week on React and we've already covered so much ground. We just finished working with stateful components to keep track of a component's internal state. Today, we're going to pause on implementation and talk a bit about how a component *lives* in an application. That is, we'll talk about the component's lifecycle.

As React mounts our application, it gives us some hooks where we can insert our own functionality at different times in the component's lifecycle. In order to *hook into* the lifecycle, we'll need to define functions on our component which React calls at the appropriate time for each hook. Let's dive into the first lifecycle hook:

`componentWillMount()` / `componentDidMount()`

When a component is defined on a page in our application, we can't depend upon it being available in the DOM immediately as we're defining virtual nodes. Instead, we have to wait until the component itself has actually *mounted* in the browser. For functionality that we need to run when it has been mounted, we get two different *hooks* (or functions) we can define. One that is called just before the component is due to be mounted on the page and one that is called just after the component has been mounted.

What does `mounting` mean?

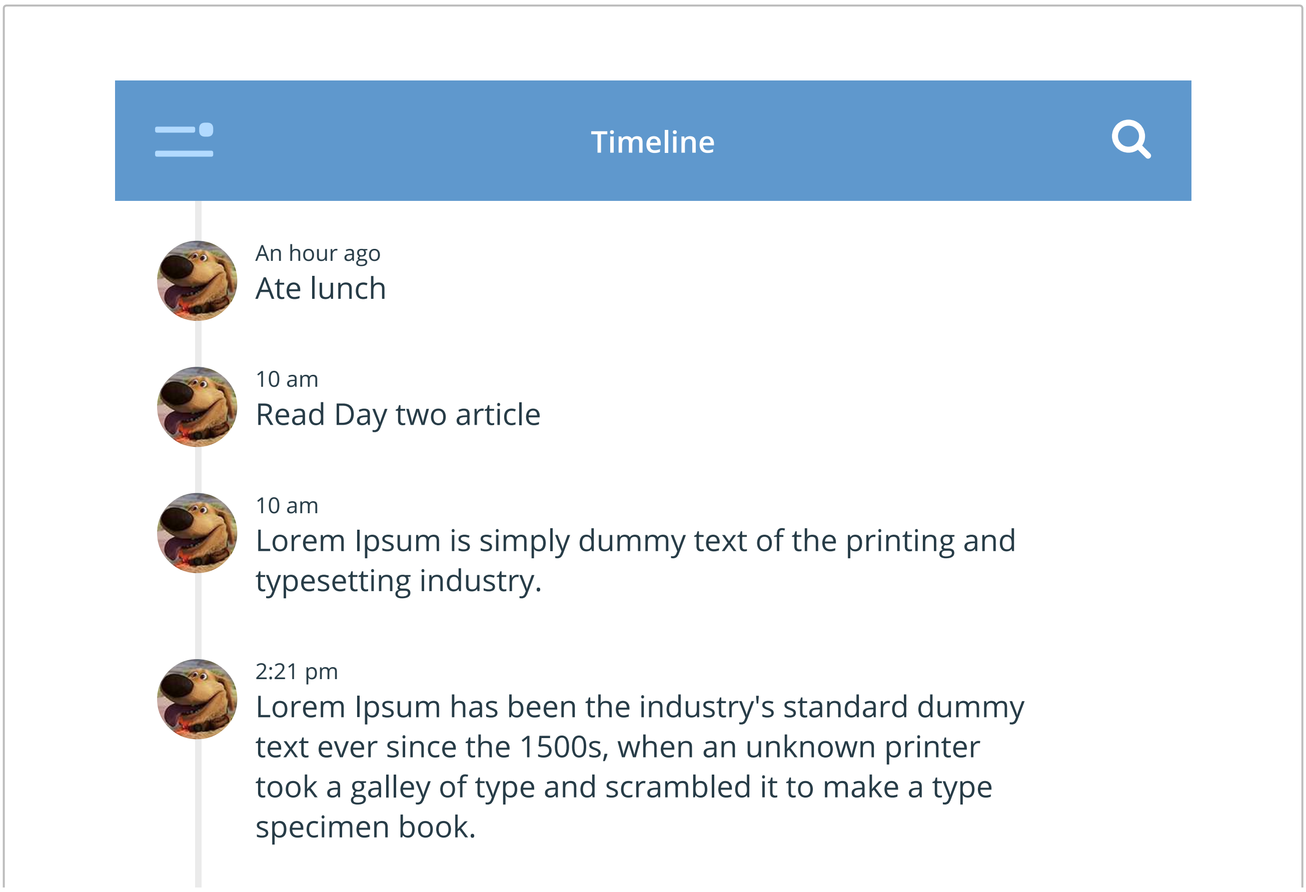
Since we're defining *virtual representations* of nodes in our DOM tree with React, we're not actually defining DOM nodes. Instead, we're building up an in-memory view that React maintains and manages for us. When we talk about *mounting*, we're talking about the process of converting the virtual components into actual DOM elements that are placed in the DOM by React.

This is useful for things such as fetching data to populate the component. For instance, let's say that we want to use our activity tracker to display github events, for example. We will want to load these events only when the data itself is going to be rendered.

Recall we defined our `Content` component in our activity list:

```
class Content extends React.Component {
  render() {
    const { activities } = this.props; // ES6 destructuring

    return (
      <div className="content">
        <div className="line" />

        {/* Timeline item */}
        {activities.map(activity => (
          <ActivityItem activity={activity} />
        ))}
      </div>
    );
  }
}
```

Let's update the `Content` component to make a request to the github.com events api (https://developer.github.com/v3/activity/events/) and use the response to display the activities. As such, we'll need to update the `state` of the object.



As we did yesterday, let's update our component to be stateful by setting `this.state` to an object in the constructor

```
class Content extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      activities: []
    };
  }

  // ...
}
```

Now, we'll want to make an HTTP request when the component itself is getting ready to be mounted (or just after it mounts). By defining the function `componentWillMount()` (or `componentDidMount()`) in our component, React runs the method just before it mounts in the DOM. This is a perfect spot for us to add a `GET` request.

Let's update the `Content` component with the request to the github api. Since we'll only want to display a small list, let's take the latest four events.

We've stored a static JSON file of github data that we'll load directly from source here (we'll get back to making AJAX requests in a few days) using promises. For now, let's focus on how we'll implement updating our component with new data:

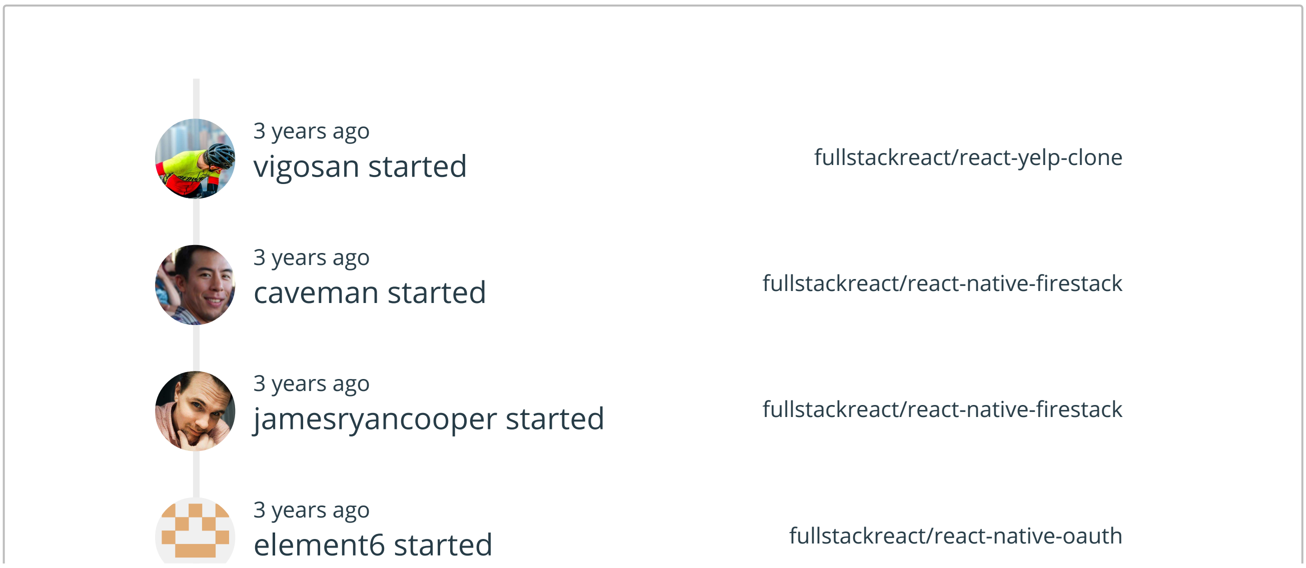
```
class Content extends React.Component {
  // ...
  componentWillMount() {
    this.setState({ activities: data });
  }
  // ...
}
```

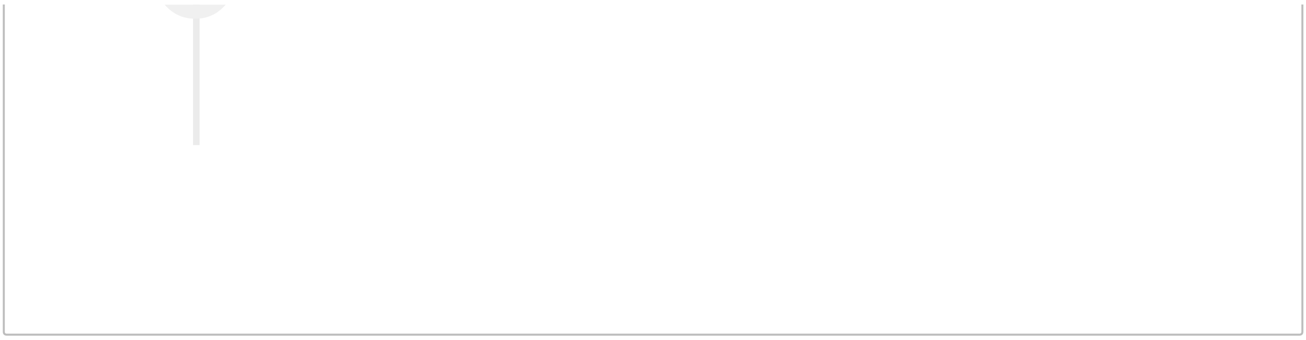
Let's also update our `ActivityItem` component slightly to reflect our new `activity` object structure. We're also using [Moment.js (https://momentjs.com/)](https://momentjs.com/) library to format the dates into a human friendly string e.g `30 min ago` To include it in your file, add the following `script` tag to your document

```
<script
src="https://unpkg.com/moment@2.24.0/min/moment.min.js">
</script>
```

```
class ActivityItem extends React.Component {
  render() {
    const { activity } = this.props;

    return (
      <div className='item'>
        <div className={'avatar'}>
          <img
            alt='avatar'
            src={activity.actor.avatar_url} />
        </div>

        <span className={'time'}>
          {moment(activity.created_at).fromNow()}
        </span>

        <p>{activity.actor.display_login} {activity.payload.action}
</p>
        <div className={'right'}>
          {activity.repo.name}
        </div>
      </div>
    )
  }
}
```

Notice that we didn't change anything else from our `Content` component and it just works.





`componentWillUpdate()` / `componentDidUpdate()`

Sometimes we'll want to update some data of our component before or after we change the actual rendering. For instance, let's say we want to call a function to set up the rendering or call a function set when a component's props are changed. The `componentWillUpdate()` method is a reasonable hook to handle preparing our component for a change (as long as we don't call `this.setState()` to handle it as it will cause an infinite loop).

Since we won't really need to handle this in-depth, we won't worry about setting up an example here, but it's good to know it exists. A more common lifecycle hook we'll use is the `componentWillReceiveProps()` hook.

`componentWillReceiveProps()`

React will call a method when the component is about to receive new `props`. This is the first method that will be called when a component is going to receive a new set of props. Defining this method is a good time to look for updates to specific `props` as it gives us an opportunity to calculate changes and update our component's internal state.

This is the time when we can update our state based on new props.

> One thing to keep in mind here is that even though the `componentWillReceiveProps()` method gets called, the value of the `props` may not have changed. It's *always* a good idea to check for changes in the prop values.

For instance, let's add a *refresh* button to our activity list so our users can request a rerequest of the github events api.

We'll use the `componentWillReceiveProps()` hook to ask the component to reload it's data. As our component is stateful, we'll want to refresh this state with new data, so we can't simply update the `props` in a component. We can use the `componentWillReceiveProps()` method to *tell* the component we want a refresh.

Let's add a button on our containing element that passes a `requestRefresh` boolean prop to tell the `Content` component to refresh.

```
class Container extends React.Component {
  constructor(props) {
    super(props);

    this.state = { refreshing: false };
  }

  // Bound to the refresh button
  refresh() {
    this.setState({ refreshing: true });
  }

  // Callback from the `Content` component
  onComponentRefresh() {
    this.setState({ refreshing: false });
  }

  render() {
    const { refreshing } = this.state;
    return (
      <div className="notificationsFrame">
        <div className="panel">
          <Header title="Github activity" />
          {/* refreshing is the component's state */}
          <Content
            onComponentRefresh={this.onComponentRefresh.bind(this)}
            requestRefresh={refreshing}
            fetchData={fetchEvents}
          />
          {/* A container for styling */}
          <Footer>
            <button onClick={this.refresh.bind(this)}>
              <i className="fa fa-refresh" />
              Refresh
            </button>
          </Footer>
        </div>
      </div>
    );
  }
}
```

`<Footer />`

Notice that we have a new element here that displays the children of the element. This is a pattern which allows us to add a CSS class around some content.

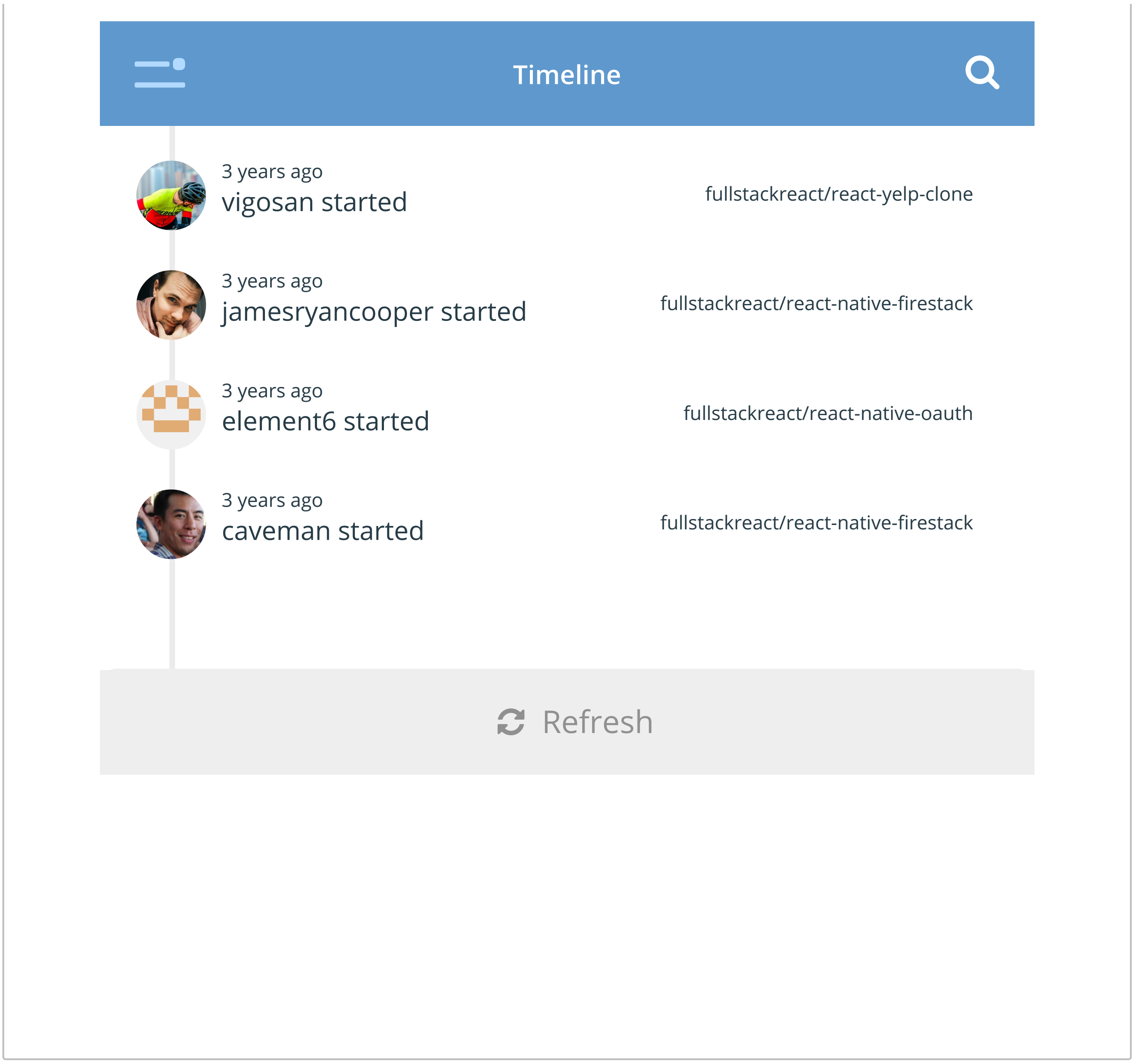
```
class Footer extends React.Component {
  render() {
    return <div className="footer">{this.props.children}</div>;
  }
}
```

Using this new `prop` (the `requestRefresh` prop), we can update the `activities` from our `state` object when it changes value.

```
class Content extends React.Component {
  constructor {
    this.state = {
      activities: [],
      loading: false // <~ set loading to false
    };
  }
  // ...
  updateData() {
    this.setState(
      {
        loading: false,
        activities: data.sort(() => 0.5 - Math.random()).slice(0, 4)
      },
      this.props.onComponentRefresh
    );
  }

  componentWillReceiveProps(nextProps) {
    // Check to see if the requestRefresh prop has changed
    if (nextProps.requestRefresh === true) {
      this.setState({ loading: true }, this.updateData);
    }
  }
  // ...
}
```

Let's also update our `componentWillMount` method to call `this.updateData()` instead of `this.setState`

```
class Content extends React.Component {
  // ...
  componentDidMount() {
    this.updateData();
  }
  // ...
}
```



> This demo is using static data from a JSON file and randomly picking four elements when we refresh. This is set up to *simulate* a refresh.

`componentWillUnmount()`

Before the component is unmounted, React will call out to the `componentWillUnmount()` callback. This is the time to handle any clean-up events we might need, such as clearing timeouts, clearing data, disconnecting websockets, etc.

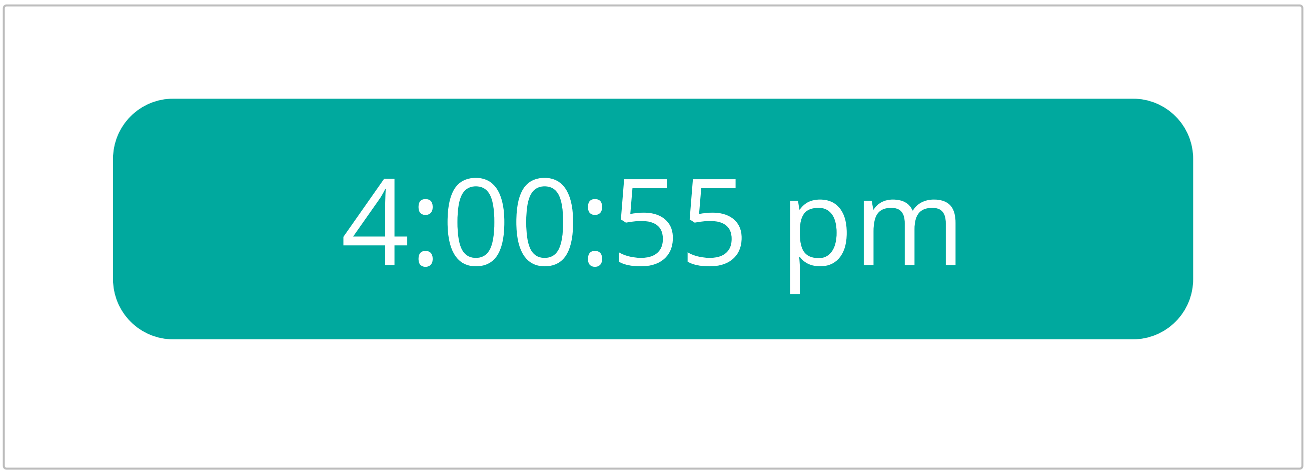
For instance, with our clock component we worked on last time, we set a timeout to be called every second. When the component is ready to unmount, we want to make sure we clear this timeout so our JavaScript doesn't continue running a timeout for components that don't actually exist.

Recall that our `timer` component we built looks like this:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = this.getTime();
  }

  componentDidMount() {
    this.setTimer();
  }

  setTimer() {
    this.timeout = setTimeout(this.updateClock.bind(this), 1000);
  }

  updateClock() {
    this.setState(this.getTime, this.setTimer);
  }

  getTime() {
    const currentTime = new Date();
    return {
      hours: currentTime.getHours(),
      minutes: currentTime.getMinutes(),
      seconds: currentTime.getSeconds(),
      ampm: currentTime.getHours() >= 12 ? "pm" : "am"
    };
  }

  // ...
  render() {}
}
```

When our clock is going to be unmounted, we'll want to clear the timeout we create in the `setTimer()` function on the component. Adding the `componentWillUnmount()` function takes care of this necessary cleanup.

```
class Clock extends React.Component {
  // ...
  componentWillUnmount() {
    if (this.timeout) {
      clearTimeout(this.timeout);
    }
  }
  // ...
}
```



These are a few of the lifecycle hooks we can interact with in the React framework. We'll be using these a lot as we build our react apps, so it's a good idea to be familiar with them, that they exist, and how to hook into the life of a component.

We did introduce one new concept in this post which we glossed over: we added a callback on a component to be called from the child to it's parent component. In the next section, we're going to look at how to define and document the `prop` API of a component for usage when sharing a component across teams and an application in general.

Packaging and PropTypes

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-08/post.md)

We're looking at how to make reusable React components today so we can share our components across apps and teams.

Phew! We made it to week two (relatively unscathed)! Through this point, we've talked through most of the basic features of React (`props` , `state` , life-cycle hooks, JSX, etc.).

In this section, we're going to look a bit at annotating and packaging our components.

PropTypes

You may have noticed we use `props` quite a bit in our components. For the most part, we'll expect these to be a particular type or set of types (aka an `object` or a `string`). React provides a method for defining and validating these types that allow us to easily expose a component API.

Not only is this a good practice for documentation purposes, it's great for building reusable react components (https://reactjs.org/docs/components-and-props.html).

The `prop-types` object exports a bunch of different types which we can use to define what type a component's prop should be. We can define these using the `propTypes` method in the ES6 class-style React prop:

```
class Clock extends React.Component {
  // ...
}

Clock.propTypes = {
  // key is the name of the prop and
  // value is the PropType
}
```

From within this `prop`, we can define an object which has the key of a prop as the name of the prop we are defining and a value defines the type (or types) it should be defined as.

For instance, the `Header` component we built a few days ago accepts a a prop called `title` and we expect it to be a string. We can define it's type to be a string as such:

> First, we'll need to `import` the `PropTypes` object from the `prop-types` package using the `import` keyword again:
>
> ```
> import PropTypes from 'prop-types'
> ```
>
> You can also use the `PropTypes` object directly in your browser by adding the following `script` tag in your page
>
> ```
> <script src="https://unpkg.com/prop-types@15.6/prop-
> types.min.js (https://unpkg.com/prop-types@15.6/prop-
> types.min.js)"></script>
> ```

```
import PropTypes from 'prop-types'

class Header extends React.Component {
  // ...
}

Header.propTypes = {
  title: PropTypes.string
}
```

React has a lot of types to choose from, exported on the `PropTypes` object and even allows for us to define a custom object type. Let's look at an overall list of available types:

Basic types

React exposes a few basic types we can use out of the box.

type	example	class
String	'hello'	`PropTypes.string`
Number	10, 0.1	`PropTypes.number`
Boolean	true / false	`PropTypes.bool`
Function	`const say => (msg) => console.log("Hello world")`	`PropTypes.func`
Symbol	Symbol("msg")	`PropTypes.symbol`
Object	`{name: 'Ari'}`	`PropTypes.object`
Anything	'whatever', 10, `{}`	

It's possible to tell React we want it to pass through *anything* that can be rendered by using `PropTypes.node`:

type	example	class
A rendererable	10, 'hello'	`PropTypes.node`

```
Clock.propTypes = {
  title: PropTypes.string,
  count: PropTypes.number,
  isOn: PropTypes.bool,
  onDisplay: PropTypes.func,
  symbol: PropTypes.symbol,
  user: PropTypes.object,

  name: PropTypes.node
}
```

We've already looked at how to communicate from a parent component to a child component using `props`. We can communicate from a child component to a parent component using a function. We'll use this pattern quite often when we want to manipulate a parent component from a child.

Collection types

We can pass through iterable collections in our `props`. We've already seen how we can do this when we passed through an array with our activities. To declare a component's proptype as an array, we can use the `PropTypes.array` annotation.

We can also require that an array holds only objects of a certain type using `PropTypes.arrayOf([])`.

type	example	class
Array	[]	`PropTypes.array`
Array of numbers	[1, 2, 3]	`PropTypes.arrayOf([type])`
Enum	['Red', 'Blue']	`PropTypes.oneOf([arr])`

It's possible to describe an object that can be one of a few different types as well using `PropTypes.oneOfType([types])`.

```
Clock.propTypes = {
  counts: PropTypes.array,
  users: PropTypes.arrayOf(PropTypes.object),
  alarmColor: PropTypes.oneOf(['red', 'blue']),
  description: PropTypes.oneOfType([
      PropTypes.string,
      PropTypes.instanceOf(Title)
    ]),
}
```

Object types

It's possible to define types that need to be of a certain shape or instance of a certain class.

type	example	class
Object	`{name: 'Ari'}`	`PropTypes.object`
Number object	`{count: 42}`	`PropTypes.objectOf()`
Instance	`new Message()`	`PropTypes.objectOf()`
Object shape	`{name: 'Ari'}`	`PropTypes.shape()`

```
Clock.propTypes = {
  basicObject: PropTypes.object,

  numbers: PropTypes
    .objectOf(PropTypes.numbers),

  messages: PropTypes
    .instanceOf(Message),

  contactList: PropTypes.shape({
    name: PropTypes.string,
    phone: PropTypes.string,
  })
}
```

React types

We can also pass through React elements from a parent to a child. This is incredibly useful for building templates and providing customization with the templates.

type	example	class
Element	`<Title />`	`PropTypes.element`

```
Clock.propTypes = {
  displayEle: PropTypes.element
}
```

When we use *element*, React expects that we'll be able to accept a single child component. That is, we won't be able to pass multiple elements.

```
// Invalid for elements
<Clock displayElement={
  <div>Name</div>
  <div>Age</div>
}></Clock>
// Valid
<Clock displayElement={
  <div>
    <div>Name</div>
    <div>Age</div>
  </div>
}></Clock>
```

Requiring types

It's possible to require a prop to be passed to a component by appending *any* of the proptype descriptions with `.isRequired`:

```
Clock.propTypes = {
  title: PropTypes.name.isRequired,
}
```

Setting a `prop` as required is very useful for times when the component is dependent upon a `prop` to be passed in by it's parent component and won't work without it.

Custom types

Finally, it's also possible to pass a function to define custom types. We can do this for a single prop or to validate arrays. The one requirement for the custom function is that if the validation does *not* pass, it expects we'll return an `Error` object:

type	example	class
Custom	'something_crazy'	`function(props, propName, componentName) {}`
CustomArray	['something', 'crazy']	`PropTypes.arrayOf(function(props, propName, componentName) {})`

```
UserLink.propTypes = {
  userWithName: (props, propName, componentName) => {
    if (!props[propName] || !props[propName].name) {
      return new Error(
        "Invalid " + propName + ": No name property defined for
component " + componentName
      )
    }
  }
}
```

Default props

Sometimes we want to be able to set a default value for a prop. For instance, our `<Header />` component, we built yesterday might not require a title to be passed. If it's not, we'll still want a title to be rendered, so we can define a common title instead by setting it's default prop value.

To set a default prop value, we can use the `defaultProps` object key on the component.

```
Header.defaultProps = {
  title: 'Github activity'
}
```

Phew, today we went through a lot of documentation. It's *always* a good idea to build our resuable components using the `propTypes` and `defaultProps` attributes of components. Not only will it make it easier to communicate across developers, it'll be much easier when we return to our components after leaving them for a few days.

Next, we'll get back to code and start integrating some style into our components.

Styles

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-09/post.md)

No application is complete without style. We'll look at the different methods we can use to style our components, from traditional CSS to inline styling.

Through this point, we haven't touched the styling of our components beyond attaching Cascading StyleSheet (CSS) class names to components.

Today, we'll spend time working through a few ways how to style our React components to make them look great, yet still keeping our sanity. We'll even work through making working with CSS a bit easier too!

Let's look at a few of the different ways we can style a component.

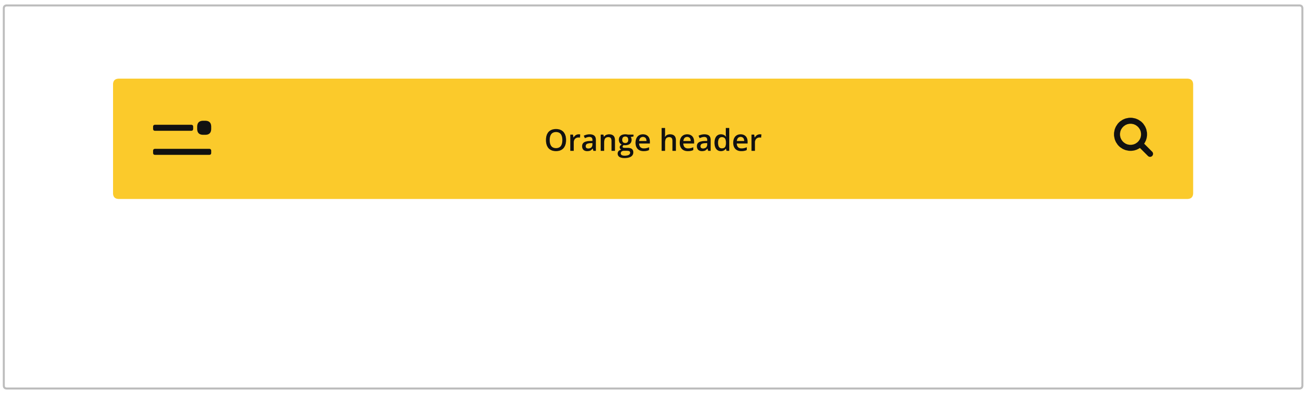
1. Cascasding StyleSheets (CSS)
2. Inline styles
3. Styling libraries

CSS

Using CSS to style our web applications is a practice we're already familiar with and is nothing new. If you've ever written a web application before, you most likely have used/written CSS. In short, CSS is a way for us to add style to a DOM component outside of the actual markup itself.

Using CSS alongside React isn't novel. We'll use CSS in React just like we use CSS when *not* using React. We'll assign ids/classes to components and use CSS selectors to target those elements on the page and let the browser handle the styling.

As an example, let's style our `Header` component we've been working with a bit.



Let's say we wanted to turn the header color orange using CSS. We can easily handle this by adding a stylesheet to our page and targeting the CSS class of `.header` in a CSS class.

Recall, the render function of our `Header` component currently looks like this:

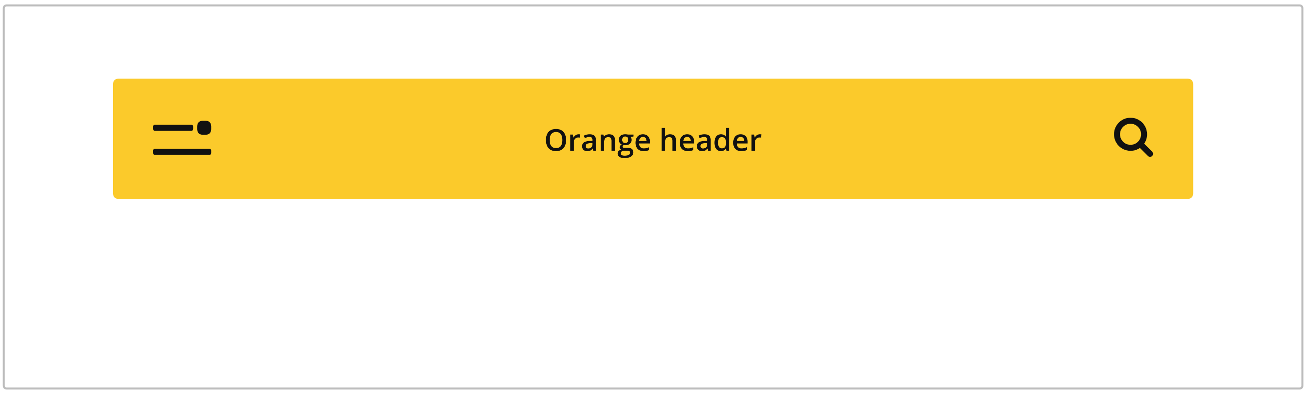
```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className="searchInput"
          placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

We can target the `header` by defining the styles for a `.header` class in a regular css file. As per-usual, we'll need to make sure we use a `<link />` tag to include the CSS class in our HTML page. Supposing we define our styles in a file called `styles.css` in the same directory as the `index.html` file, this `<link />` tag will look like the following:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

Let's fill in the styles for the `Header` class names:

```
.demo .notificationsFrame .header {
  background: rgba(251, 202, 43, 1);
}
.demo .notificationsFrame .header .searchIcon,
.demo .notificationsFrame .header .title {
  color: #333333;
}

.demo .notificationsFrame .header .menuIcon .dashTop,
.demo .notificationsFrame .header .menuIcon .dashBottom,
.demo .notificationsFrame .header .menuIcon .circle {
  background-color: #333333;
}
```



One of the most common complaints about CSS in the first place is the cascading feature itself. The way CSS works is that it *cascades* (hence the name) parent styles to it's children. This is often a cause for bugs as classes often have common names and it's easy to overwrite class styles for non-specific classes.

Using our example, the class name of `.header` isn't very specific. Not only could the page itself have a header, but content boxes on the page might, articles, even ads we place on the page might have a class name of `.header`.

> One way we can avoid this problem is to use something like [css modules (https://glenmaddern.com/articles/css-modules)](https://glenmaddern.com/articles/css-modules) to define custom, very unique CSS class names for us. There is nothing magical about CSS modules other than it forces our build-tool to define custom CSS class names for us so we can work with less unique names. We'll look into using CSS modules a bit later in our workflow.

React provides a not-so-new method for avoiding this problem entirely by allowing us to define styles inline along with our JSX.

Inline styles

Adding styles to our actual components not only allow us to define the styles inside our components, but allow us to dynamically define styles based upon different states of the app.

React gives us a way to define styles using a JavaScript object rather than a separate CSS file. Let's take our `Header` component one more time and instead of using css classes to define the style, let's move it to inline styles.

Defining styles inside a component is easy using the `style` prop. All DOM elements inside React accept a `style` property, which is expected to be an object with camel-cased keys defining a style name and values which map to their value.

For example, to add a `color` style to a `<div />` element in JSX, this might look like:

```
<div style={{ color: 'blue' }}>
  This text will have the color blue
</div>
```

This text will have the color blue

> Notice that we defined the styles with two braces surrounding it. As we are passing a JavaScript object within a template tag, the inner brace is the JS object and the outer is the template tag.
>
> Another example to possibly make this clearer would be to pass a JavaScript object defined outside of the JSX, i.e.
>
> ```
> render() {
>   const divStyle = { color: 'blue' }
>   return (
>     This text will have the color blue
>   );
> }
> ```

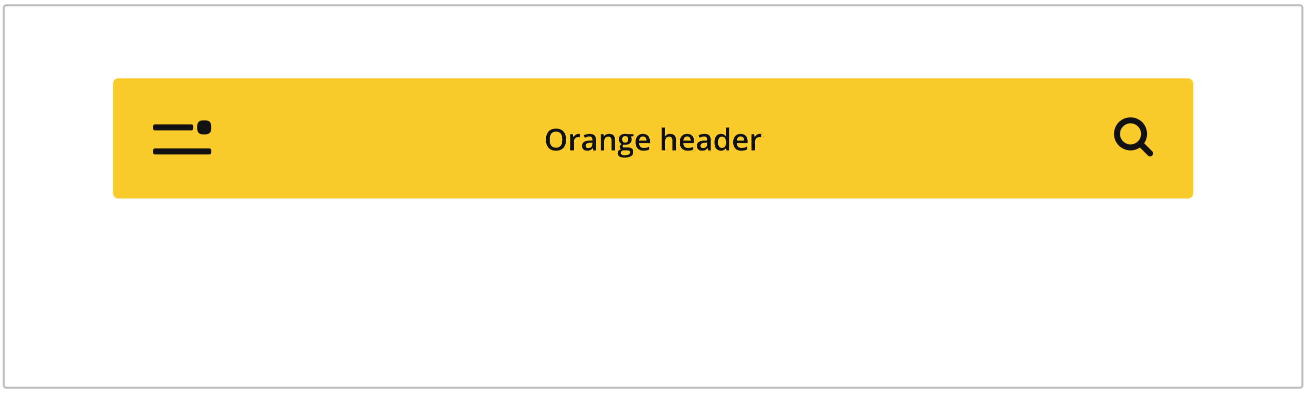
In any case, as these are JS-defined styles, so we can't use just any ole' css style name (as `background-color` would be invalid in JavaScript). Instead, React requires us to camel-case the style name.

> camelCase (https://en.wikipedia.org/wiki/CamelCase) is writing compound words using a capital letter for every word with a capital letter except for the first word, like `backgroundColor` and `linearGradient`.

To update our header component to use these styles instead of depending on a CSS class definition, we can add the `className` prop along with a `style` prop:

```
class Header extends React.Component {
  render() {
    const wrapperStyle = {
      backgroundColor: "rgba(251, 202, 43, 1)"
    };

    const titleStyle = {
      color: "#111111"
    };

    const menuColor = {
      backgroundColor: "#111111"
    };

    return (
      <div style={wrapperStyle} className="header">
        <div className="menuIcon">
          <div className="dashTop" style={menuColor}></div>
          <div className="dashBottom" style={menuColor}></div>
          <div className="circle" style={menuColor}></div>
        </div>

        <span style={titleStyle} className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className="searchInput"
          placeholder="Search ..."
        />

        <div style={titleStyle} className="fa fa-search searchIcon">
</div>
      </div>
    );
  }
}
```

Our header will be orange again.



Styling libraries

The React community is a pretty vibrant place (which is one of the reasons it is a fantastic library to work with). There are a lot of styling libraries we can use to help us build our styles, such as Radium (https://formidable.com/open-source/radium/) by Formidable labs.

Most of these libraries are based upon workflows defined by React developers through working with React.

Radium allows us to define common styles outside of the React component itself, it auto-vendor prefixes, supports media queries (like `:hover` and `:active`), simplifies inline styling, and kind of a lot more.

We won't dive into Radium in this post as it's more outside the scope of this series, but knowing other libraries are good to be aware of, especially if you're looking to extend the definitions of your inline styles.

Now that we know how to style our components, we can make some good looking ones without too much trouble. In the next section, we'll get right back to adding user interactivity to our components.

Interactivity

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-10/post.md)

Today we'll walk through how to add interactivity to our applications to make them engaging and dynamic.

Through this point, we've built our few handful of components without adding much user interaction. Today, we're going to change that.

User interaction

The browser is an event-driven application. Everything that a user does in the browser fires an event, from clicking buttons to even just moving the mouse. In plain JavaScript, we can listen for these events and attach a JavaScript function to interact with them.

For instance, we can attach a function to the `mousemove` browser event with the JS:

```
const ele = document.getElementById('mousemove');
ele.innerHTML = 'Move your mouse over this text';
ele.addEventListener('mousemove', function(evt) {
  const { screenX, screenY } = evt;
  ele.innerHTML = '<div>Mouse is at: X: ' +
        screenX + ', Y: ' + screenY +
                  '</div>';
})
```

This results in the following functionality:

Move your mouse over this text

In React, however we don't have to interact with the browser's event loop in raw JavaScript as React provides a way for us to handle events using `props`.

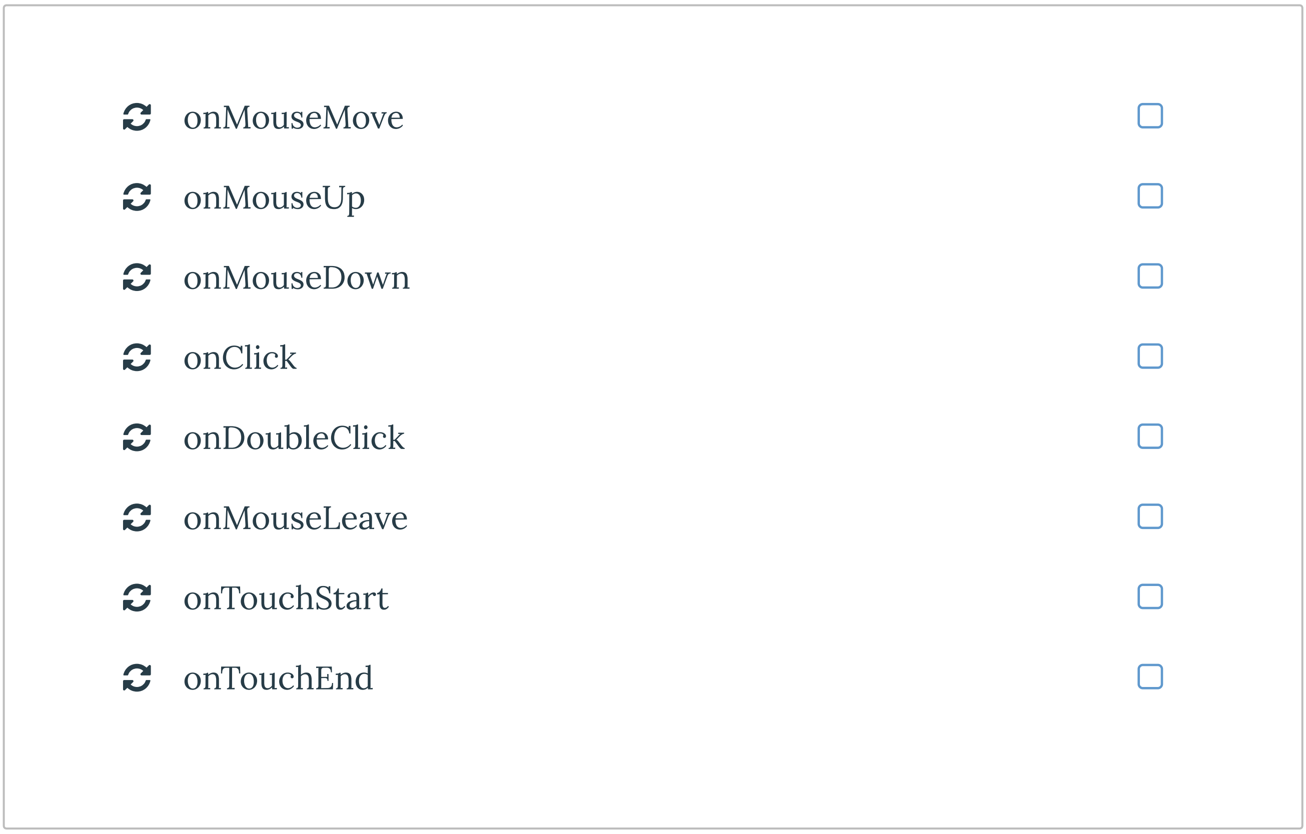
For instance, to listen for the `mousemove` event from the (rather unimpressive) demo above in React, we'll set the prop `onMouseMove` (notice the camelcasing of the event name).

```
class MouseMover extends React.Component {
  state = {
    x: 0,
    y: 0
  };

  handleMouseMove = e => {
    this.setState({
      x: e.clientX,
      y: e.clientY
    });
  };

  render() {
    return (
      <div onMouseMove={this.handleMouseMove}>
        {this.state.x || this.state.y
          ? "The mouse is at x: " + this.state.x + ", y: " +
this.state.y
          : "Move the mouse over this box"}
      </div>
    );
  }
}
```

React provides a lot of `props` we can set to listen for different browser events, such as click, touch, drag, scroll, selection events, and many more (see the events (https://facebook.github.io/react/docs/events.html) documentation for a list of all of them).

The mouse is at x: undefined, y: undefined

To see some of these in action, the following is a small demo of some of the `props` we can pass on our elements. Each text element in the list set the prop it lists. Try playing around with the list and seeing how the events are called and handled within the element (all events are set on the text, not the list item):



We'll be using the `onClick` prop quite a bit all throughout our apps quite a bit, so it's a good idea to be familiar with it. In our activity list header, we have a search icon that we haven't hooked up yet to show a search box.

The interaction we *want* is to show a search `<input />` when our users click on the search icon. Recall that our `Header` component is implemented like this:

```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className="searchInput"
          placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

Let's update it a bit so that we can pass dynamic `className` prop to the `<input />` element

```
class Header extends React.Component {
  render() {
    // Classes to add to the <input /> element
    let searchInputClasses = ["searchInput"];
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className={searchInputClasses.join(' ')}
          placeholder="Search ..." />

        <div className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

When the user clicks on the `<div className="fa fa-search searchIcon"></div>` element, we'll want to run a function to update the state of the component so the `searchInputClasses` object gets updated. Using the `onClick` handler, this is pretty simple.

Let's make this component stateful (it needs to track if the search field should be showing or not). We can convert our component to be stateful using the `constructor()` function:

```
class Header extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      searchVisible: false
    }
  }
  // ...
}
```

What is a `constructor` function?

In JavaScript, the `constructor` function is a function that runs when an object is created. It returns a reference to the Object function that created the instance's `prototype`.

In plain English, a constructor function is the function that runs when the JavaScript runtime creates a new object. We'll use the constructor method to set up instance variables on the object as it runs right when the object is created.

When using the `ES6` class syntax to create an object, we have to call the `super()` method before any other method. Calling the `super()` function calls the parent class's `constructor()` function. We'll call it with the *same arguments* as the `constructor()` function of our class is called with.

When the user clicks on the button, we'll want to update the state to say that the `searchVisible` flag gets updated. Since we'll want the user to be able to close/hide the `<input />` field after clicking on the search icon for a second time, we'll *toggle* the state rather than just set it to true.

Let's create this method to bind our click event:

```
class Header extends React.Component {
  // ...
  showSearch() {
    this.setState({
      searchVisible: !this.state.searchVisible
    })
  }
  // ...
}
```

Let's add an if statement to update `searchInputClasses` if `this.state.searchVisible` is `true`

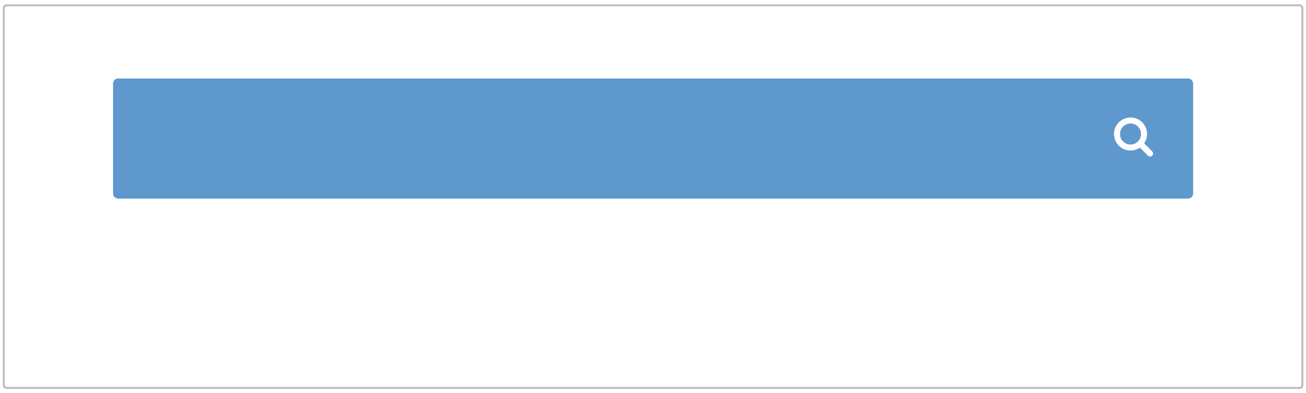
```
class Header extends React.Component {
  // ...
  render() {

    // ...
    // Update the class array if the state is visible
    if (this.state.searchVisible) {
      searchInputClasses.push("active");
    }
    // ...
  }
}
```

Finally, we can attach a click handler (using the `onClick` prop) on the icon element to call our new `showSearch()` method. The entire updated source for our `Header` component looks like this:

```
class Header extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      searchVisible: false
    }
  }

  // toggle visibility when run on the state
  showSearch() {
    this.setState({
      searchVisible: !this.state.searchVisible
    })
  }

  render() {
    // Classes to add to the <input /> element
    let searchInputClasses = ["searchInput"];

    // Update the class array if the state is visible
    if (this.state.searchVisible) {
      searchInputClasses.push("active");
    }

    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">
          {this.props.title}
        </span>

        <input
          type="text"
          className={searchInputClasses.join(' ')}
          placeholder="Search ..." />
```

```
        {/* Adding an onClick handler to call the showSearch button
*/}
        <div
          onClick={this.showSearch.bind(this)}
          className="fa fa-search searchIcon"></div>
      </div>
    )
  }
}
```

Try clicking on the search icon and watch the input field appear and disappear (the animation effect is handled by CSS animations).



Input events

Whenever we build a form in React, we'll use the input events offered by React. Most notably, we'll use the `onSubmit()` and `onChange()` props most often.

Let's update our search box demo to capture the text inside the search field when it updates. Whenever an `<input />` field has the `onChange()` prop set, it will call the function *every time the field changes*. When we click on it and start typing, the function will be called.

Using this prop, we can capture the value of the field in our state.

Rather than updating our `<Header />` component, let's create a new child component to contain a `<form />` element. By moving the form-handling responsibilities to it's own form, we can simplify the `<Header />` code and we can call up to the parent of the header when our user submits the form (this is a usual React pattern).

Let's create a new component we'll call `SearchForm`. This new component is a stateful component as we'll need to hold on to the value of the search input (track it as it changes):

```
class SearchForm extends React.Component {
  // ...
  constructor(props) {
    super(props);

    this.state = {
      searchText: ''
    }
  }
  // ...
}
```

Now, we already have the HTML for the form written in the `<Header />` component, so let's grab that from our `Header` component and return it from our `SearchForm.render()` function:

```
class SearchForm extends React.Component {
  // ...
  render() {
    const { searchVisible } = this.props;
    let searchClasses = ["searchInput"];
    if (searchVisible) {
      searchClasses.push("active");
    }

    return (
      <form>
        <input
          type="search"
          className={searchClasses.join(" ")}
          placeholder="Search ..."
        />
      </form>
    );
  }
}
```

Now that we've moved some code from the `Header` component to the `SearchForm`, let's update its `render` method to incorporate the `SearchForm`

```
class Header extends React.Component {
  // ...
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">{this.props.title}</span>

        <SearchForm />

        {/* Adding an onClick handler to call the showSearch button
*/}
        <div
          onClick={this.showSearch.bind(this)}
          className="fa fa-search searchIcon"
        ></div>
      </div>
    );
  }
}
```

Notice that we lost the styles on our `<input />` field. Since we no longer have the `searchVisible` state in our new component, we can't use it to style the `<input />` any longer. *However*, we can pass a prop from our `Header` component that tells the `SearchForm` to render the input as visible.

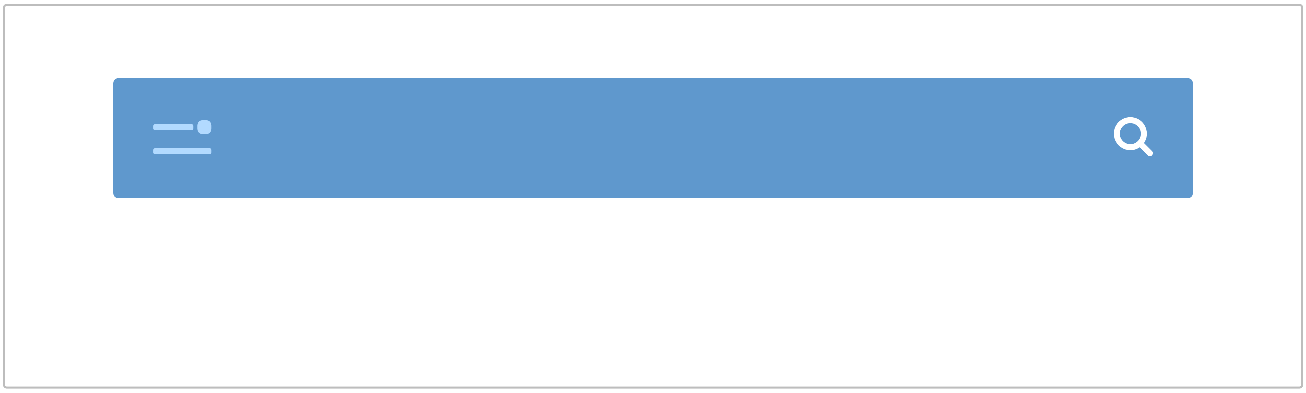
Let's define the `searchVisible` prop (using `PropTypes`, of course) and update the `render` function to use the new prop value to show (or hide) the search `<input />`. We'll also set a default value for the visibility of the field to be false (since our `Header` shows/hides it nicely):

```
class SearchForm extends React.Component {
  // ...
}

SearchForm.propTypes = {
  searchVisible: PropTypes.bool
}

SearchForm.defaultProps = {
  searchVisible: false
};
```

> In case you forgot to include `PropTypes` package in your page just add the following `script` tag in your page
>
> ```
> <script src="https://unpkg.com/prop-types@15.6/prop-types.min.js"></script>
> ```

Finally, let's pass the `searchVisible` state value from `Header` as a prop to `SearchForm`

```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">{this.props.title}</span>

        <SearchForm searchVisible={this.state.searchVisible} />

        {/* Adding an onClick handler to call the showSearch button
*/}
        <div
          onClick={this.showSearch.bind(this)}
          className="fa fa-search searchIcon"
        ></div>
      </div>
    );
  }
}
```



Now we have our styles back on the `<input />` element, let's add the functionality for when the user types in the search box, we'll want to capture the value of the search field. We can achieve this workflow by attaching the `onChange` prop to the `<input />` element and passing it a function to call every time the `<input />` element is changed.

```
class SearchForm extends React.Component {
  // ...
  updateSearchInput(e) {
    const val = e.target.value;
    this.setState({
      searchText: val
    });
  }
  // ...
  render() {
    const { searchVisible } = this.state;
    let searchClasses = ['searchInput']
    if (searchVisible) {
      searchClasses.push('active')
    }

    return (
      <form>
        <input
          type="search"
          className={searchClasses.join(" ")}
          onChange={this.updateSearchInput.bind(this)}
          placeholder="Search ..."
        />
      </form>
    );
  }
}
```

When we type in the field, the `updateSearchInput()` function will be called. We'll keep track of the value of the form by updating the state. In the `updateSearchInput()` function, we can call directly to `this.setState()` to update the state of the component.

> The value is held on the `event` object's target as `event.target.value`.

```
class SearchForm extends React.Component {
  // ...
  updateSearchInput(e) {
    const val = e.target.value;
    this.setState({
      searchText: val
    });
  }
  // ...
}
```

Controlled vs. uncontrolled

We're creating what's known as an **uncontrolled** component as we're not setting the value of the `<input />` element. We can't provide any validation or post-processing on the input text value as it stands right now.

If we want to validate the field or manipulate the value of the `<input />` component, we'll have to create what is called a **controlled** component, which really just means that we pass it a value using the `value` prop. A controlled component version's `render()` function would look like:

```
class SearchForm extends React.Component {
  render() {
    return (
      <input
        type="search"
        value={this.state.searchText}
        className={searchInputClasses}
        onChange={this.updateSearchInput.bind(this)}
        placeholder="Search ..." />
    );
  }
}
```

As of now, we have no way to actually submit the form, so our user's can't really search. Let's change this. We can capture the form submission by using the `onSubmit` prop on the `<form />` element.

Let's update the `render()` function to reflect this change.

```
class SearchForm extends React.Component {
  // ...
  submitForm(event) {
    event.preventDefault();
  }
  // ...
  render() {
    const { searchVisible } = this.props;
    let searchClasses = ['searchInput']
    if (searchVisible) {
      searchClasses.push('active')
    }

    return (
      <form onSubmit={this.submitForm.bind(this)}>
        <input
          type="search"
          className={searchClasses.join(' ')}
          onChange={this.updateSearchInput.bind(this)}
          placeholder="Search ..." />
      </form>
    );
  }
}
```

> We immediately call `event.preventDefault()` on the `submitForm()` function. This stops the browser from bubbling the event up which would causes the default behavior of the entire page to reload (the default function when a browser submits a form).

Now when we type into the `<input />` field and press enter, the `submitForm()` function gets called with the event object.

So... great, we can submit the form and stuff, but when do we actually do the searching? For demonstration purposes right now, we'll pass the search text up the parent-child component chain so the `Header` can decide *what* to search.

> The `SearchForm` component certainly doesn't know what it's searching, so we'll have to pass the responsibility up the chain. We'll use this callback strategy quite a bit.

In order to pass the search functionality up the chain, our `SearchForm` will need to accept a prop function to call when the form is submitted. Let's define a prop we'll call `onSubmit` that we can pass to our `SearchForm` component. Being good developers, we'll also add a default `prop` value and a `propType` for this `onSubmit` function. Since we'll want to make sure the `onSubmit()` is defined, we'll set the `onSubmit` prop to be a required prop:

```
class SearchForm extends React.Component {
  // ...
}

SearchForm.propTypes = {
  onSubmit: PropTypes.func.isRequired,
  searchVisible: PropTypes.bool
}

SearchForm.defaultProps = {
  onSubmit: () => {},
  searchVisible: false
}
```

When the form is submitted, we can call this function directly from the `props`. Since we're keeping track of the search text in our state, we can call the function with the `searchText` value in the state so the `onSubmit()` function only gets the value and doesn't need to deal with an event.

```
class SearchForm extends React.Component {
  // ...
  submitForm(event) {
    // prevent the form from reloading the entire page
    event.preventDefault();
    // call the callback with the search value
    this.props.onSubmit(this.state.searchText);
  }
}
```

Now, when the user presses enter we can call this `onSubmit()` function passed in the `props` by our `Header` component.

Let's add the `onSubmit` prop to the `SearchForm` in the `Header` component:

```
class Header extends React.Component {
  // ...
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <span className="title">{this.props.title}</span>

        <SearchForm searchVisible={this.state.searchVisible} onSubmit=
{this.props.onSearch}/>

        {/* Adding an onClick handler to call the showSearch button
*/}
        <div
          onClick={this.showSearch.bind(this)}
          className="fa fa-search searchIcon"
        ></div>
      </div>
    );
  }
}
```

Now we have a search form component we can use and reuse across our app. Of course, we're not actually searching anything yet. Let's fix that and implement search.

Implementing search

To implement search in our component, we'll want to pass up the search responsibility one more level from our `Header` component to a container component we'll call `Panel`.

First things first, let's implement the same pattern of passing a callback to a parent component from within a child component from the `Panel` to the `Header` component.

On the `Header` component, let's update the `propTypes` for a prop we'll define as a prop called `onSearch`:

```
class Header extends React.Component {
  // ...
}
Header.propTypes = {
  onSearch: PropTypes.func
}
```

Here's our `Panel` component:

```
class Content extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activities: data,
    };
  }

  render() {
    const { activities } = this.state; // ES6 destructuring

    return (
      <div>
        <Header
          title="Github activity" />
        <div className="content">
          <div className="line" />
          {/* Timeline item */}
          {activities.map(activity => (
            <ActivityItem key={activity.id} activity={activity} />
          ))}
        </div>
      </div>
    );
  }
}
```

In any case, our `Panel` component is essentially a copy of our `Content` component we previously built on day 7. Make sure to include the `ActivityItem` component in your page. Also don't forget to include `Moment.js` in your file as it's used by `ActivityItem` to format dates. Add the following `script` tag in your page

```
<script
src="https://unpkg.com/moment@2.24.0/min/moment.min.js">
</script>
```

Notice that our virtual tree looks like this:

```
<Panel>
  <Header>
    <SearchForm></SearchForm>
  </Header>
</Panel>
```

When the `<SearchForm />` is updated, it will pass along it's awareness of the search input's change to it's parent, the `<Header />`, when it will pass along upwards to the `<Panel />` component. This method is *very common* in React apps and provides a good set of functional isolation for our components.

Back in our `Panel` component, we'll pass a function to the `Header` as the `onSearch()` prop on the `Header`. What we're saying here is that when the search form has been submitted, we want the search form to call back to the header component which will then call to the `Panel` component to handle the search.

Since the `Header` component doesn't control the content listing, the `Panel` component does, we *have* to pass the responsibility one more level up, as we're defining here.

In order to actually handle the searching, we'll need to pass an `onSearch()` function to our `Header` component. Let's define an `onSearch()` function in our `Panel` component and pass it off to the `Header` props in the `render()` function:

```
class Panel extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activities: data,
    };
  }

  handleSearch(val) {
    // handle search here
  }

  render() {
    const { activities } = this.state; // ES6 destructuring
    return (
      <div>
        <Header
          title="Github activity"
          onSearch={this.handleSearch.bind(this)}
        />
        <div className="content">
          <div className="line" />
          {/* Timeline item */}
          {activities.map(activity => (
            <ActivityItem key={activity.id} activity={activity} />
          ))}
        </div>
      </div>
    );
  }
}
```

All we did here was add a `handleSearch()` function and pass it to the header. Now when the user types in the search box, the `handleSearch()` function on our `Panel` component will be called.

Let's update our `handleSearch` method to actually do the searching:

```
class Panel extends React.Component {
  // ...
  handleSearch(val) {
    // resets the data if the search value is empty
    if (val === "") {
      this.setState({
        activities: data
      });
    } else {
      const { activities } = this.state;
      const filtered = activities.filter(
        a => a.actor && a.actor.login.match(val)
      );
      this.setState({
        activities: filtered
      });
    }
  }
  // ...
}
```

All the `activities.filter()` function does is run the function passed in for every element and it filters *out* the values that return falsy values, keeping the ones that return truthy ones. Our search function simply looks for a match on the Github activity's `actor.login` (the Github user) to see if it regexp-matches the `val` value.

With the `handleSearch()` function updated, our search is complete.

Try searching for `auser`.

Now we have a 3-layer app component that handles search from a nested child component. We jumped from beginner to intermediate with this post. Pat yourself on the back. This was some hefty material. Make sure you understand this because we'll use these concepts we covered today quite often.

In the next section, we'll jump out and look at building *pure* components.

Pure Components

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-11/post.md)

React offers several different methods for creating components. Today we'll talk about the final method of creating components, the function stateless pure component.

We've looked at a few different ways to build react components. One method we left out up through this point is the stateless component/functional method of building React components.

As we've seen up through this point, we've only worked through building components using the `React.Component` and `React.createClass()` methods. For more performance and simplicity, React *also* allows us to create pure, stateless components using a normal JavaScript function.

A *Pure* component can replace a component that only has a `render` function. Instead of making a full-blown component just to render some content to the screen, we can create a *pure* one instead.

Pure components are the simplest, fastest components we can write. They are easy to write, simple to reason about, and the quickest component we can write. Before we dive into *why* these are better, let's write one, or heck a couple!

```
// The simplest one
const HelloWorld = () => (<div>Hello world</div>);

// A Notification component
const Notification = (props) => {
  const {level, message} = props;
  const classNames = ['alert', 'alert-' + level]
  return (
    <div className={classNames}>
      {message}
    </div>
  )
};

// In ES5
var ListItem = function(props) {
  var handleClick = function(event) {
    props.onClick(event);
  };

  return (
    <div className="list">
      <a
        href="#"
        onClick={handleClick}>
          {props.children}
      </a>
    </div>
  )
}
```

So they are just functions, right? Yep! Since they are just functions, it's really easy to test using pure JavaScript. The idea is that if React knows the `props` that are sent into a component, it can be deterministic in knowing if it has to rerender or not. The same props in equal the same output virtual DOM.

In React, functional components are called with an argument of `props` (similar to the `React.Component` constructor class), which are the `props` it's called with as well as with the current `context` of the component tree.

For instance, let's say we want to rewrite our original `Timer` component using functional components as we want to give our users a dynamic way to set their own clock styles (24 hour clock vs. 12, different separators, maybe they don't want to display the seconds, etc).

We can break up our clock into multiple components where we can use each block of time as an individual component. We might break them up like so:

```
const Hour    = (props) => {
  let {hours} = props;
  if (hours === 0) { hours = 12; }
  if (props.twelveHours) { hours -= 12; }
  return (<span>{hours}</span>)
}
```

```
const Minute  = ({minutes}) => (<span>{minutes<10 && '0'}{minutes}
</span>)
```

```
const Second  = ({seconds}) => (<span>{seconds<10 && '0'}{seconds}
</span>)
```

```
const Separator = ({separator}) => (<span>{separator || ':'}</span>)
```

```
const Ampm = ({hours}) => (<span>{hours >= 12 ? 'pm' : 'am'}</span>)
```

With these, we can place individual components as through they are full-blown React components (they are):

```
<div>Minute: <Minute minutes={12} /></div>
<div>Second: <Second seconds={51} /></div>
```

Minute: 12
Second: 51

We can refactor our clock component to accept a `format` string and break up this string selecting only the components our user is interested in showing. There are multiple ways we can handle this, like forcing the logic into the `Clock` component *or* we can create another stateless component that accepts a format string. Let's do that (easier to test):

```
const Formatter = (props) => {
  let children = props.format.split('').map((e, idx) => {
    if (e === 'h') {
      return <Hour key={idx} {...props} />
    } else if (e === 'm') {
      return <Minute key={idx} {...props} />
    } else if (e === 's') {
      return <Second key={idx} {...props} />
    } else if (e === 'p') {
      return <Ampm key={idx} {...props} />
    } else if (e === ' ') {
      return <span key={idx}> </span>;
    } else {
      return <Separator key={idx} {...props} />
    }
  });

  return <span>{children}</span>;
}
```

This is a little ugly with the `key` and `{...props}` thingie in there. React gives us some helpers for mapping over children and taking care of handling the unique `key` for each child through the `React.Children` object.

The `render()` function of our `Clock` component can be greatly simplified thanks to the `Formatter` component into this:

```
class Clock extends React.Component {
  state = { currentTime: new Date() }
  componentDidMount() {
    this.setState({
      currentTime: new Date()
    }, this.updateTime);
  }
  componentWillUnmount() {
    if (this.timerId) {
      clearTimeout(this.timerId)
    }
  }

  updateTime = e => {
    this.timerId = setTimeout(() => {
      this.setState({
        currentTime: new Date()
      }, this.updateTime);
    })
  }

  render() {
    const { currentTime } = this.state
    const hour = currentTime.getHours();
    const minute = currentTime.getMinutes();
    const second = currentTime.getSeconds();

    return (
      <div className='clock'>
        <Formatter
          {...this.props}
          state={this.state}
          hours={hour}
          minutes={minute}
          seconds={second}
        />
      </div>
    )
  }
}
```

We can now render the clock in a custom format:

```
ReactDOM.render(<Clock format="h:m:s p" />,
document.querySelector("#app"));
```

Not only is our `Clock` component *much* simpler, but it's so much easier to test. It *also* will help us transition to using a data state tree, like Flux/Redux frameworks, but more on those later.

16:01:11 pm

Uhh... so why care?

Advantages to using functional components in React are:

- We can do away with the heavy lifting of components, no constructor, state, life-cycle madness, etc.
- There is no `this` keyword (i.e. no need to bind)
- Presentational components (also called dumb components) emphasize UI over business logic (i.e. no state manipulation in the component)
- Encourages building smaller, self-contained components
- Highlights badly written code (for better refactoring)
- FAST FAST FAST FAST FAST
- They are *easy* to reuse

You might say why not use a functional component? Well, some of the disadvantage of using a functional component are some of the advantages:

- No life-cycle callback hooks
- Limited functionality
- There is no `this` keyword

Overall, it's a really good idea to try to prefer using functional components over their heavier `React.Component` cousins. When we get to talking about data management in React, we'll see how we can use these presentational components with data as pure `props`.

Nice work today. We've successfully achieved React rank after today. We now know the *three* ways to make a React Component.

Tomorrow, we'll get set up using/building React apps with the package management tool shipped by the React team: `create-react-app`.

create-react-app

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-12/post.md)

Today, we're going to add a build process to store common build actions so we can easily develop and deploy our applications.

The React team noticed that there is a lot of configuration required (and the community helped bloat -- us included) to run a React app. Luckily, some smart folks in the React team/community got together and built/released an official generator app that makes it much easier to get up and running quickly.

Packaging

So far in this course, we've only been working with writing our components in a single script. Although it's great for simplicity, it can be difficult to share components amongst multiple developers. A single file is also pretty difficult to write complex applications.

Instead, we'll set up a build tool for our applications using a very popular packaging tool called create-react-app (https://github.com/facebookincubator/create-react-app). The tool provides a great place to start out developing our applications without needing to spend too much time working on setting up our build tooling.

In order to use it, we'll need to start out by installing it. We can use `npm` or `yarn` to install `create-react-app`:

create-react-app

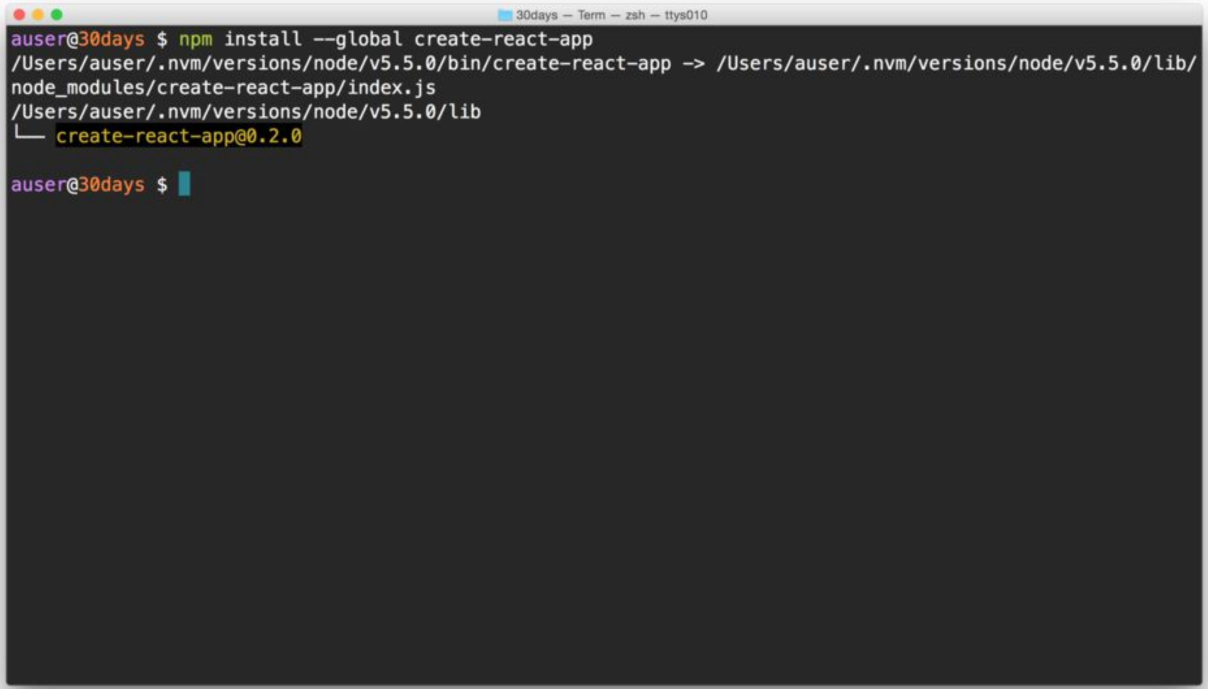
The create-react-app (https://github.com/facebookincubator/create-react-app) project is released through Facebook helps us get up and running quickly with a React app on our system with no custom configuring required on our part.

The package is released as a Node (https://nodejs.org/) Package (https://www.npmjs.com/package/create-react-app) and can be installed using `npm`.

> ## A plug for `nvm` and `n`
>
> The Node (https://nodejs.org) homepage has simple documentation on how to install node, if you don't already have it installed on your system.
>
> We recommend using the nvm (https://github.com/creationix/nvm) or the n (https://github.com/tj/n) version management tools. These tools make it incredibly easy to install/use multiple versions of node on your system at any point.

With `node` installed on our system, we can install the `create-react-app` package:

```
npm install --global create-react-app
```



With `create-react-app` installed globally, we'll be able to use the `create-react-app` command anywhere in our terminal.

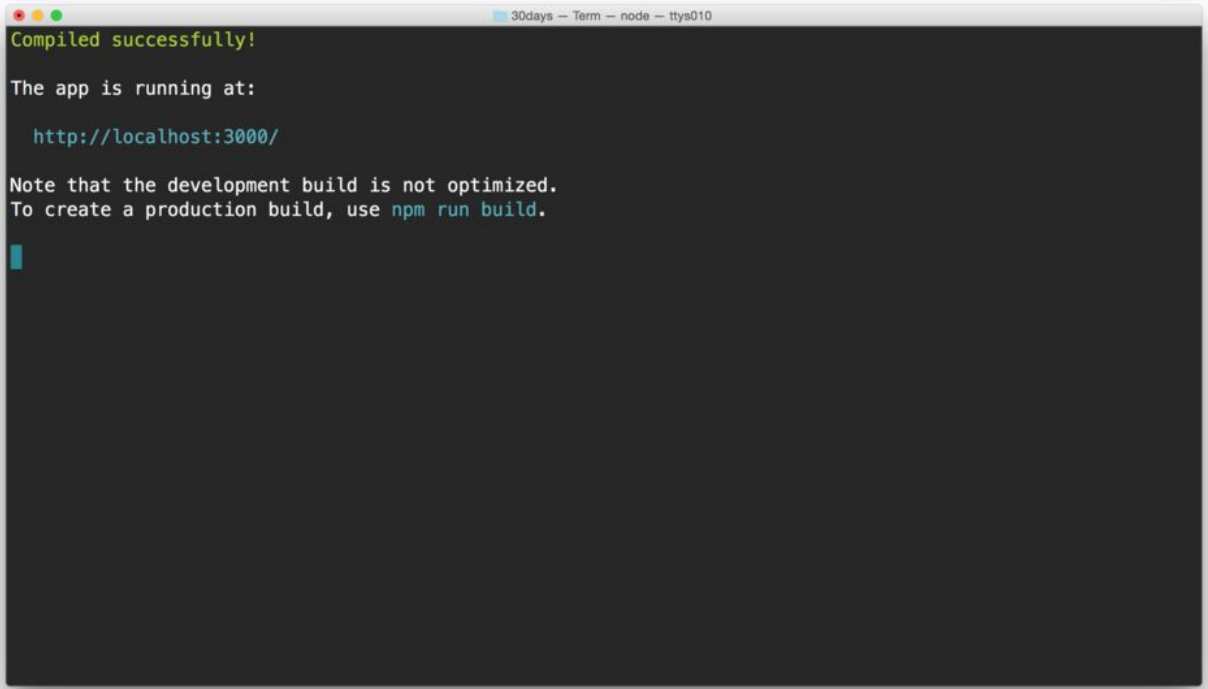
Let's create a new app we'll call 30days using the `create-react-app` command we just installed. Open a Terminal window in a directory where you want to create your app.

In terminal, we can create a new React application using the command and adding a name to the app we want to create.

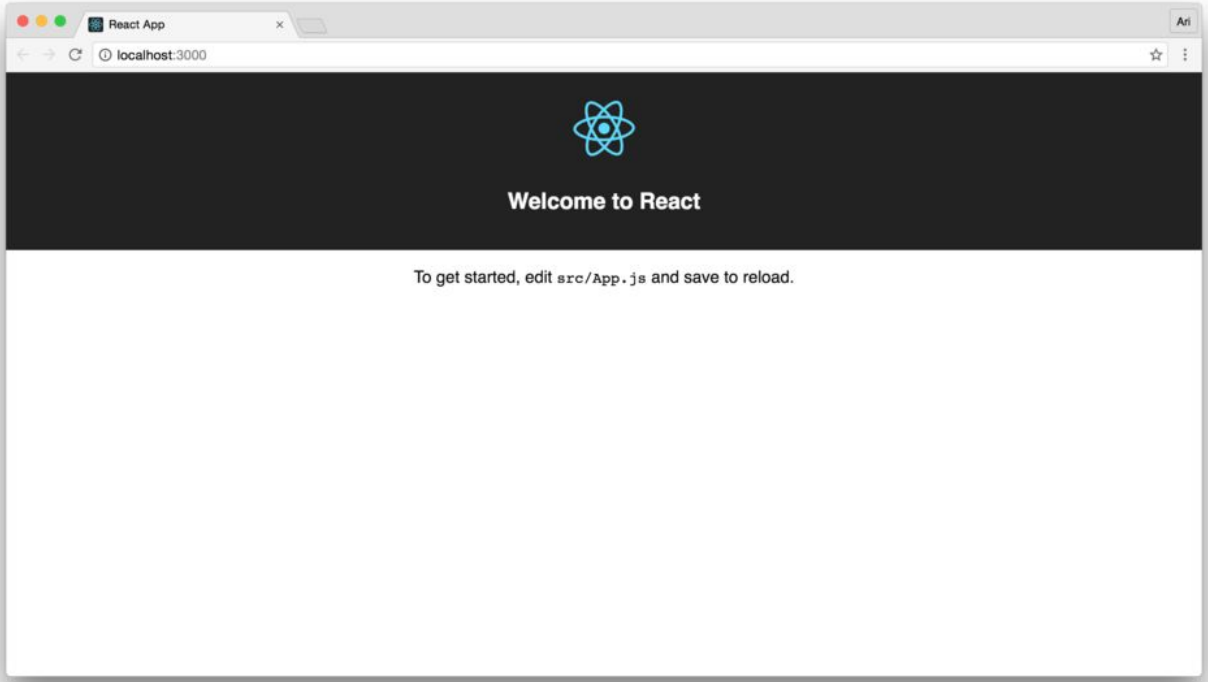
```
create-react-app 30days && cd 30days
```

Let's start our app in the browser. The `create-react-app` package comes with a few built-in scripts it created for us (in the `package.json` file). We can *start* editing our app using the built-in webserver using the `npm start` command:

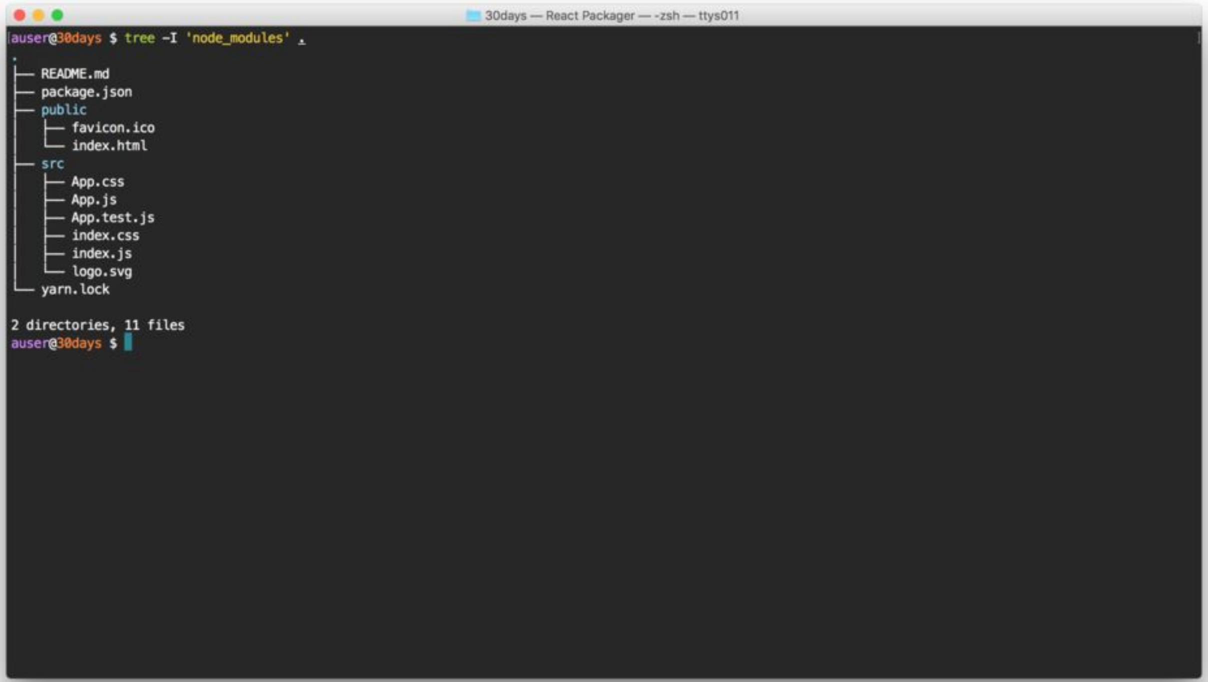
```
npm start
```



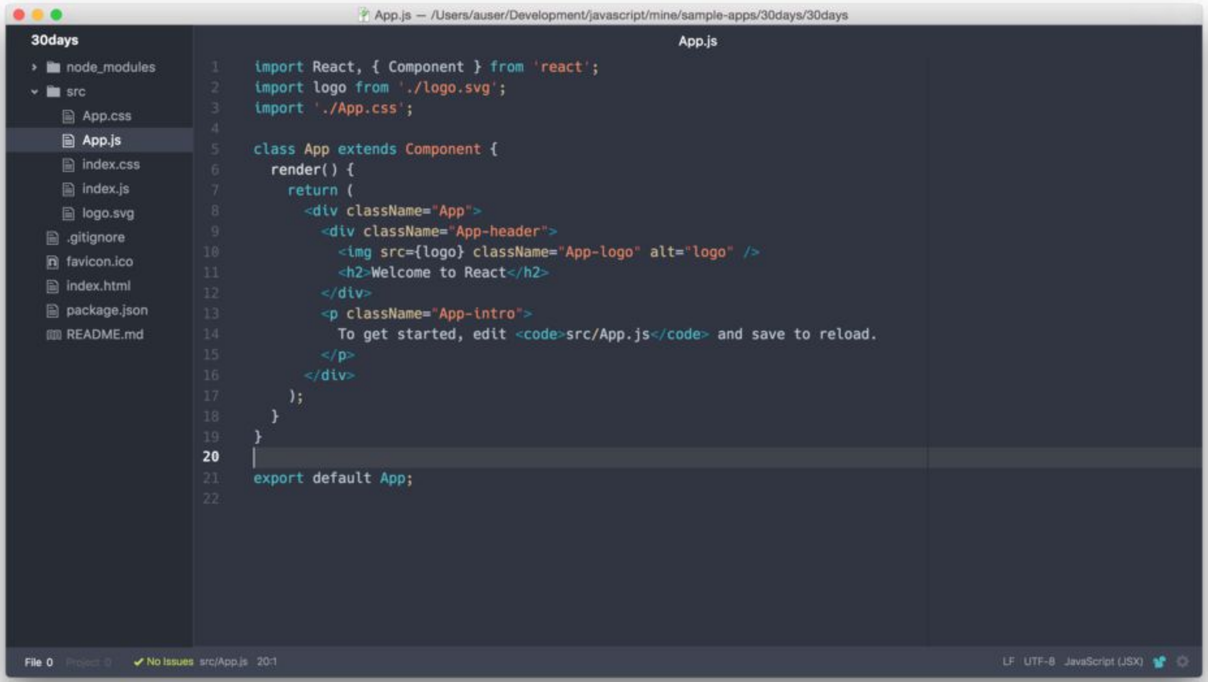
This command will open a window in Chrome to the default app it created for us running at the url: http://localhost:3000/ (http://localhost:3000/).



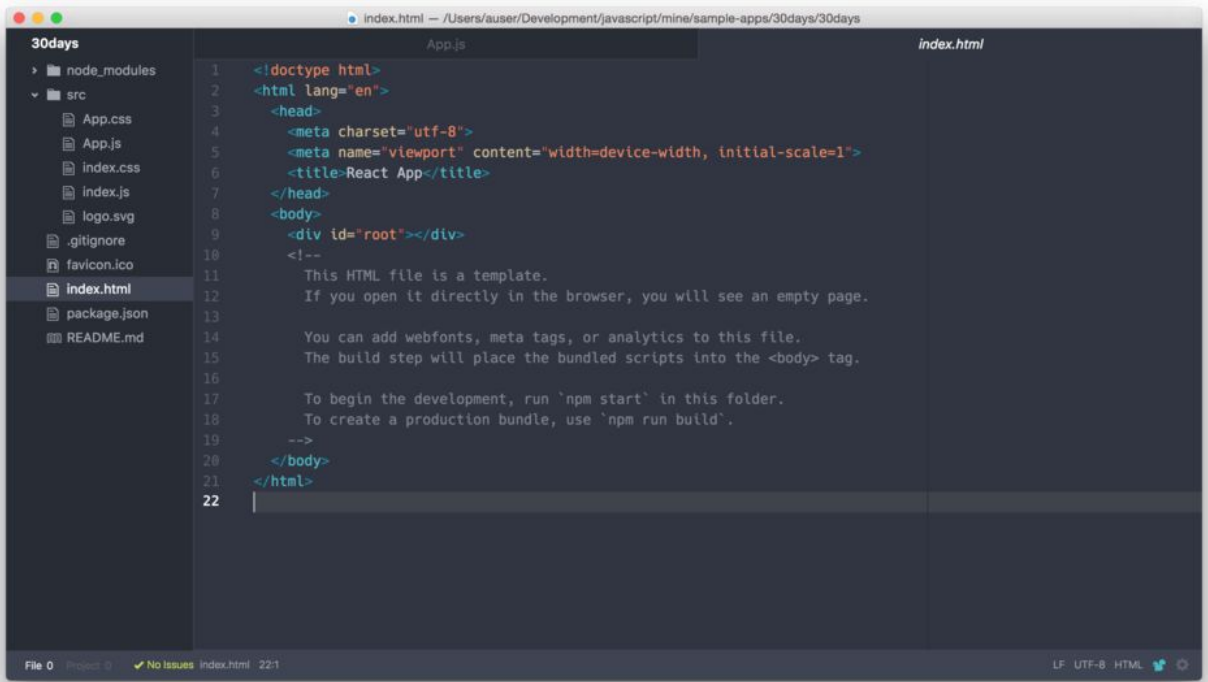
Let's edit the newly created app. Looking at the directory structure it created, we'll see we have a basic node app running with a `public/index.html` and a few files in the `src/` directory that comprise our running app.



Let's open up the `src/App.js` file and we'll see we have a very basic component that should all look familiar. It has a simple render function which returns the result we see in the Chrome window.



The `index.html` file has a single `<div />` node with the id of `#root`, where the app itself will be mounted for us automatically (this is handled in the `src/index.js` file). Anytime we want to add webfonts, style tags, etc. we can load them in the `index.html` file.



Let's look at a few of the features `create-react-app` enables for us.

We've used multiple components in the past. Let's pull in the example we walked through on day-4 with a header and content (slightly simplified -- changing the className from `notificationsFrame` to `App` and removing the inner component):

```
import React from "react";

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <Header />
        <Content />
      </div>
    );
  }
}
```

We *could* define the `Header` and the `Content` component in the same file, but as we discussed, that becomes pretty cumbersome. Instead, let's create a directory called `components/` in the `src/` directory (`src/components/`) and create two files called `Header.js` and `Content.js` in there:

```
# in my-app/
mkdir src/components
touch src/components/{Header,Content}.js
```

Now, let's write the two components in their respective file. First, the `Header` components in `src/components/Header.js`:

```
import React from "react";

class Header extends React.Component {
  render() {
    return (
      <div id="header">
        <h1>Header</h1>
      </div>
    );
  }
}
```

And now let's write the `Content` component in the `src/components/Content.js` file:

```
import React from "react";

class Content extends React.Component {
  render() {
    return <p className="App-intro">Content goes here</p>;
  }
}
```

By making a small update to these two component definitions, we can then `import` them into our `App` component (in `src/App.js`).

We'll use the `export` keyword before the `class` definition:

Let's update the `Header` component slightly:

```
export class Header extends React.Component {
  // ...
}
```

and the `Content` component:

```
export class Content extends React.Component {
  // ...
}
```

Now we can `import` these two component from our `src/App.js` file. Let's update our `App.js` by adding these two `import` statements:

```
import React from "react";

import { Header } from "./components/Header";
import { Content } from "./components/Content";

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <Header />
        <Content />
      </div>
    );
  }
}
```

Here, we're using *named* exports to pull in the two components from their respective files in `src/components/`.

By convention, if we only have a single export from these files, we can use the `export default` syntax so we can remove the `{}` surrounding the named export. Let's update each of these respective files to include an extra line at the end to enable the default import:

```
export class Header extends React.Component {
  // ...
}

export default Header;
```

and the `Content` component:

```
export class Content extends React.Component {
  // ...
}

export default Content;
```

Now we can update our import of the two components like so:

```
import React from "react";

import Header from "./components/Header";
import Content from "./components/Content";

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <Header />
        <Content />
      </div>
    );
  }
}
```

Using this knowledge, we can now also update our components by importing the named `Component` class and simplify our definition of the class file again. Let's take the `Content` component in `src/components/Content.js`:

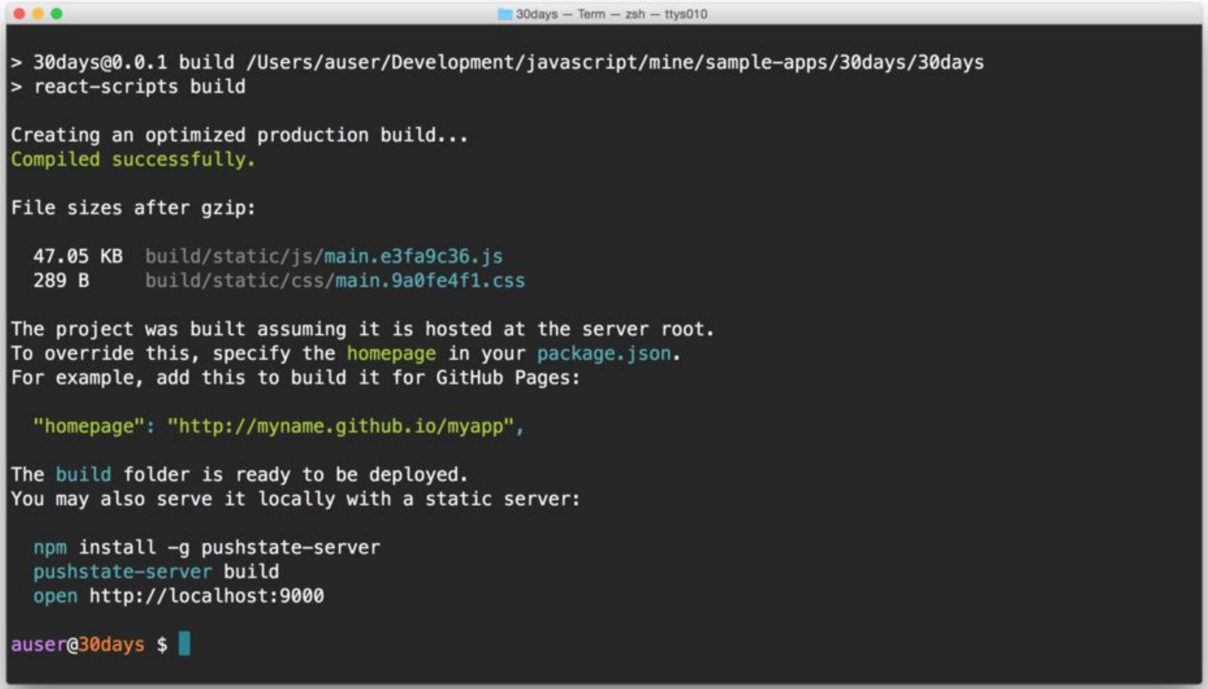
```
import React, {Component} from 'react'; // This is the change

export class Content extends Component { // and this allows us
                                         // to not call
React.Component
                                         // but instead use just
                                         // the Component class

  render() {
    return <p className="App-intro">Content goes here</p>;
  }
}

export default Content;
```

Shipping

We'll get to deployment in a few weeks, but for the time being know that the generator created a build command so we can create minified, optimize versions of our app that we can upload to a server.

We can build our app using the `npm run build` command in the root of our project:

```
npm run build
```



With that, we now have a live-reloading single-page app (SPA) ready for development. Tomorrow, we'll use this new app we built diving into rendering multiple components at run-time.

Repeating Elements

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-13/post.md)

Today we're going to work through how to display multiple components in preparation for pulling in external data into our app.

Up through this point, we've been building a basic application without any external data. Before we get there (we'll start on this functionality tomorrow), let's look over something we glossed over in the previous two weeks:

Repeating elements

We've already seen this before where we've iterated over a list of objects and render multiple components on screen. Before we add too much complexity in our app with loading external data, today we'll take a quick peek at how to repeat components/elements in our app.

Since JSX is seen as plain JavaScript by the browser, we can use any ole' JavaScript inside the template tags in JSX. We've already seen this in action. As a quick demo:

```
const a = 10;
const ShowA = () => <div>{a}</div>;
const MultipleA = () => <div>{a * a}</div>;

const App = props => {
  return (
    <div className="app">
      <ShowA />
      <MultipleA />
    </div>
  );
};
```

10
100

Notice the things inside of the template tags `{}` look like simple JavaScript. That's because it is *just* JavaScript. This feature allows us to use (most) native features of JavaScript inside our template tags **including** native iterators, such as `map` and `forEach`.

Let's see what we mean here. Let's convert the previous example's `a` value from a single integer to a list of integers:

```
const a = [1, 10, 100, 1000, 10000];
```

We can map over the `a` variable here inside our components and return a list of React components that will build the virtual DOM for us.

```
const a = [1, 10, 100, 1000, 10000];
const Repeater = () => {
  return (
    <ul>
      {a.map(i => {
        return <li>{i}</li>;
      })}
    </ul>
  );
};
```

What is the `map()` function?

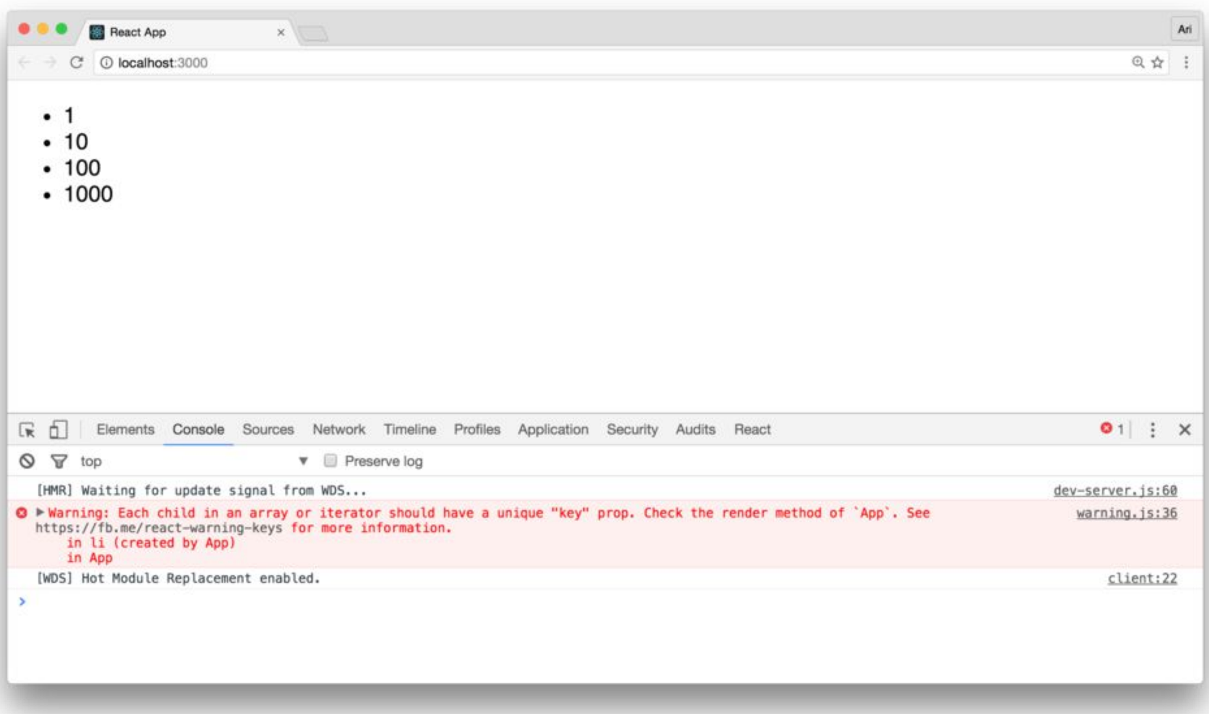
The `map` function is a native JavaScript built-in function on the array. It accepts a function to be run on each element of the array, so the function above will be run four times with the value of `i` starting as `1` and then it will run it again for the second value where `i` will be set as `10` and so on and so forth.

- 1
- 10
- 100
- 1000

Let's update the app we created on day 12 with our `App` component here. Let's open up our `src/App.js` file and replace the content of the `App` component with this source. Cleaning up a few unused variables and your `src/App.js` should look similar to this:

```
import React from "react";

const a = [1, 10, 100, 1000, 10000];
const App = props => {
  return (
    <ul>
      {a.map(i => {
        return <li>{i}</li>;
      })}
    </ul>
  );
};

export default App;
```

Starting the app again with the command generated by the `create-react-app` command: `npm start`, we can see the app is working in the browser!



However, if we open the developer console, we'll see we have an error printed out. This error is caused by the fact that React doesn't know how to keep track of the individual components in our list as each one just looks like a `<li />` component.

For performance reasons, React uses the virtual DOM to attempt to limit the number of DOM elements that need to be updated when it rerenders the view. That is if nothing has changed, React won't make the browser update anything to save on work.

This feature is really fantastic for building web applications, but sometimes we have to help React out by providing unique identifiers for nodes. Mapping over a list and rendering components in the map is one of those times.

React expects us to **uniquely** identify components by using a special prop: the `key` prop for each element of the list. The `key` prop can be anything we want, but it **must be unique** for that element. In our example, we can use the `i` variable in the map as no other element in the array has the same value.

Let's update our mapping to set the key:

```
const App = props => {
  return (
    <ul>
      {a.map(i => {
        return <li key={i}>{i}</li>;
      })}
    </ul>
  );
};
```

Children

We talked about building a parent-child relationship a bit earlier this week, but let's dive a bit more into detail about how we get access to the children inside a parent component and how we can render them.

On day 11, we built a `<Formatter />` component to handle date formatting within the Clock component to give our users flexibility with their own custom clock rendering. Recall that the implementation we created is actually pretty ugly and relatively complex.

```
const Formatter = props => {
  let children = props.format.split("").map((e, idx) => {
    if (e === "h") {
      return <Hour key={idx} {...props} />;
    } else if (e === "m") {
      return <Minute key={idx} {...props} />;
    } else if (e === "s") {
      return <Second key={idx} {...props} />;
    } else if (e === "p") {
      return <Ampm key={idx} {...props} />;
    } else if (e === " ") {
      return <span key={idx}> </span>;
    } else {
      return <Separator key={idx} {...props} />;
    }
  });

  return <span>{children}</span>;
};
```

We can use the `React.Children` object to map over a list of React objects and let React do this heavy-lifting. The result of this is a much cleaner `Formatter` component (not perfect, but functional):

```
const Formatter = props => {
  let children = props.format.split("").map(e => {
    if (e == "h") {
      return <Hour />;
    } else if (e == "m") {
      return <Minute />;
    } else if (e == "s") {
      return <Second />;
    } else if (e == "p") {
      return <Ampm />;
    } else if (e == " ") {
      return <span> </span>;
    } else {
      return <Separator />;
    }
  });
  return (
    <span>
      {React.Children.map(children, c => React.cloneElement(c,
props))}
    </span>
  );
};
```

React.cloneElement

We have yet to talk about the `React.cloneElement()` function, so let's look at it briefly here. Remember WWWWWAAAAAYYYYY back on day 2 we looked at how the browser *sees* JSX? It turns it into JavaScript that looks similar to:

```
React.createElement("div", null,
 React.createElement("img", {src: "profile.jpg", alt: "Profile
photo"}),
 React.createElement("h1", null, "Welcome back Ari")
);
```

Rather than creating a new component instance (if we already have one), sometimes we'll want to copy it or add custom props/children to the component so we can retain the same props it was created with. We can use `React.cloneElement()` to handle this for us.

The `React.cloneElement()` has the same API as the `React.createElement()` function where the arguments are:

1. The ReactElement we want to clone
2. Any `props` we want to add to the instance
3. Any `children` we want it to have.

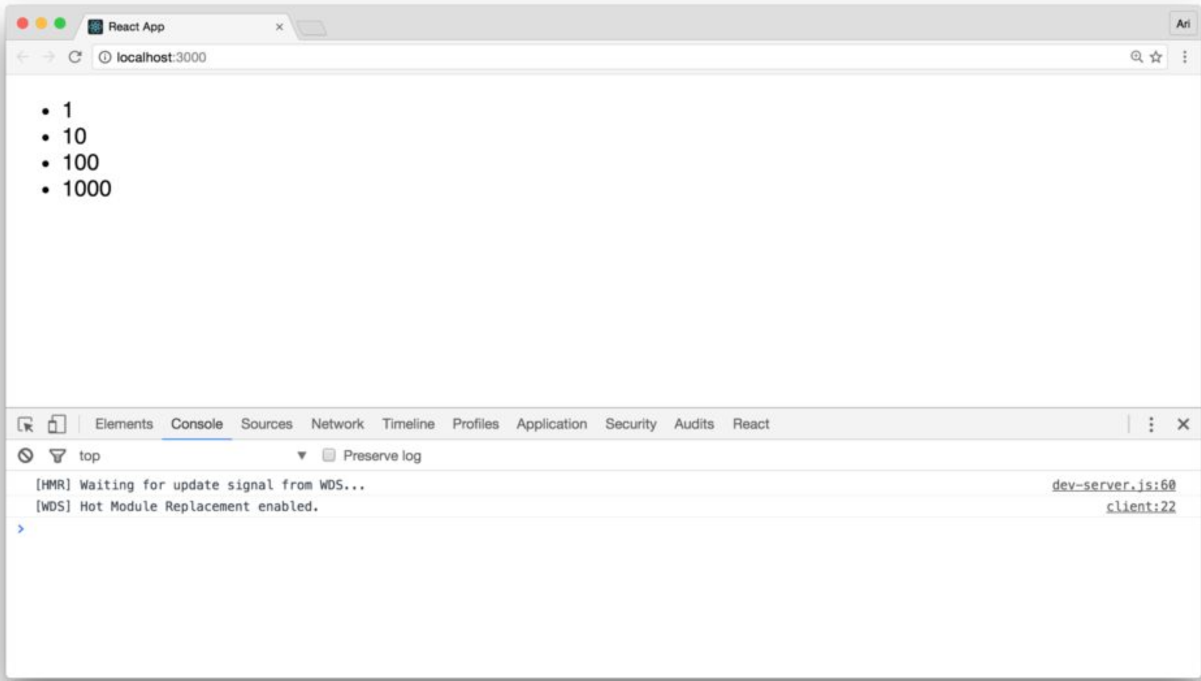
In our `Formatter` example, we're creating a copy of all the children in the list (the `<Hour />`, `<Minute />`, etc. components) and passing them the `props` object as their props.

The `React.Children` object provides some nice utility functions for dealing with children. Our `Formatter` example above uses the `map` function to iterate through the children and clone each one in the list. It creates a `key` (if necessary) for each one, freeing us from having to manage the uniqueness ourselves.

Let's use the `React.Children.map()` function to update our App component:

```
const App = props => {
  return (
    <ul>
      {React.Children.map(a, i => (
        <li>{i}</li>
      ))}
    </ul>
  );
};
```

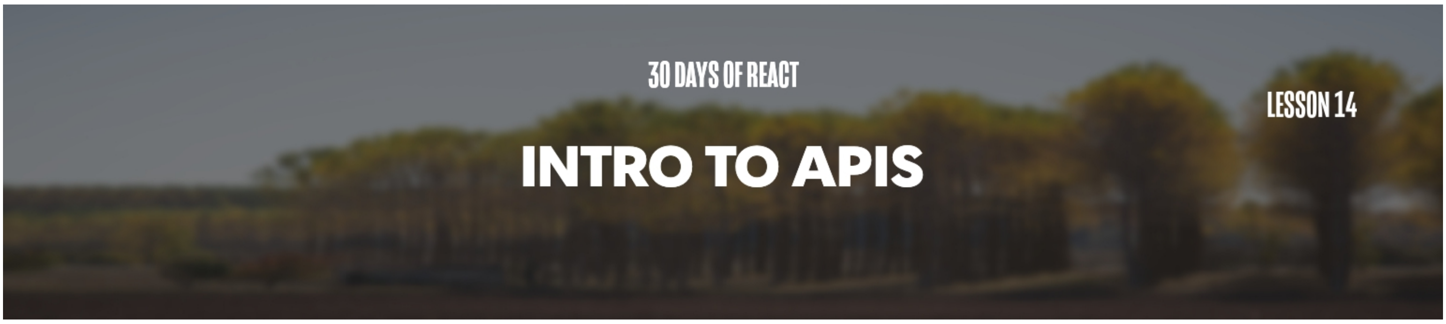
Back in the browser, everything still works.



There are several other really useful methods in the `React.Children` object available to us. We'll mostly use the `React.Children.map()` function, but it's good to know about the other ones available (https://facebook.github.io/react/docs/top-level-api.html#react.children) to us. Check out the documentation (https://facebook.github.io/react/docs/top-level-api.html#react.children) for a longer list.

Up through this point, we've only dealt with local data, not really focusing on remote data (although we *did* briefly mention it when building our activity feed component). Tomorrow we're going to get into interacting with a server so we can use it in our React apps.

Great work today!



Fetching Remote Data

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-14/post.md)

We're ready to make an external request to fetch data! Today we're looking at the first step of making a call to an external API.

Our apps, until this point have largely been static. Even the data we displayed from Github was static data included in our project. Our apps are really only as interesting as the data we use, so let's make our apps more interesting.

Querying for remote data

The normal browser workflow is actually a synchronous one. When the browser receives html, it parses the string of html content and converts it into a tree object (this is what we often refer to as the DOM object/DOM tree).

When the browser parses the DOM tree, as it encounters remote files (such as `<link />` and `<script />` tags), the browser will request these files (in parallel), but will execute them synchronously (so as to maintain their order they are listed in the source).

What if we want to get some data from off-site? We'll make requests for data that's not available at launch time to populate data in our app. However, it's not necessarily *that* easy to do because of the asynchronous nature of external API requests.

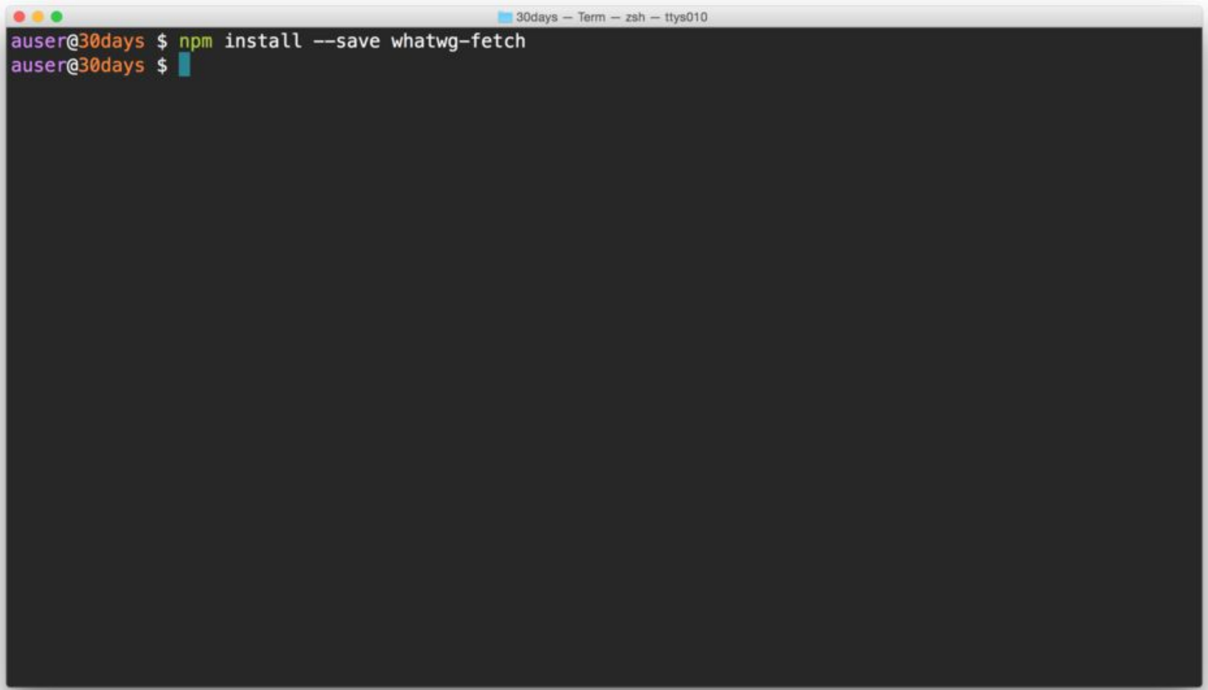
Essentially, what this means is that we'll have to handle with JavaScript code after an unknown period of time as well actually make an HTTP request. Luckily for us, other people have dealt with this problem for a long time and we now have some pretty nice ways of handling it.

Starting with handling how we'll be making an HTTP request, we'll use a library (called `fetch`, which is also a web standard (https://fetch.spec.whatwg.org/), hopefully) to make the http requesting easier to deal with.

Fetch

In order to use fetch, we'll need to install the library in our app we previously created. Let's open up a terminal window again and use `npm` to install the `whatwg-fetch` library (an implementation of `fetch`). In the same directory where we created our application, let's call:

```
npm install --save whatwg-fetch
```



With the library installed, we can make a request to an off-site server. In order to get access to the `fetch` library, we'll need to `import` the package in our script. Let's update the top few lines of our `src/App.js` file adding the second line:

```
import React, { Component } from "react";
import "whatwg-fetch";
// ...
```

> The `whatwg-fetch` object is unique in that it is one of the few libraries that we'll use which attaches it's export on the `global` object (in the case of the browser, this object is `window`). Unlike the `react` library, we don't need to get a handle on it's export as the library makes it available on the global object.

With the `whatwg-fetch` library included in our project, we can make a request using the `fetch()` api. However, before we can actually start using the `fetch()` api, we'll need to understand what Promises are and how they work to deal with the asynchronous we discussed in the introduction.

We'll pick up with `promises` tomorrow. Good job getting through week two and see you tomorrow!

Introduction to Promises

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-15/post.md)

Today, we're going to look at what we need to know to understand Promises from a high-level, so we can build our applications using this incredibly useful concept.

Yesterday (/articles/30-days-of-react-day-14/) we installed the `fetch` library into our `create-react-app` project we started on day 12 (/articles/30-days-of-react-day-12/). Today we'll pick up from yesterday discussing the concept and the *art* of Promises (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).

What is a promise

As defined by the Mozilla, a `Promise` object is used for handling asynchronous computations which has some important guarantees that are difficult to handle with the callback method (the more old-school method of handling asynchronous code).

A `Promise` object is simply a wrapper around a value that may or may not be known when the object is instantiated and provides a method for handling the value *after* it is known (also known as `resolved`) or is unavailable for a failure reason (we'll refer to this as `rejected`).

Using a `Promise` object gives us the opportunity to associate functionality for an asynchronous operation's eventual success or failure (for whatever reason). It also allows us to treat these complex scenarios by using synchronous-like code.

For instance, consider the following synchronous code where we print out the current time in the JavaScript console:

```
var currentTime = new Date();
console.log('The current time is: ' + currentTime);
```

This is pretty straight-forward and works as the `new Date()` object represents the time the browser knows about. Now consider that we're using a different clock on some other remote machine. For instance, if we're making a Happy New Years clock, it would be great to be able to synchronize the user's browser with everyone elses using a single time value for everyone so no-one misses the ball dropping ceremony.

Suppose we have a method that handles getting the current time for the clock called `getCurrentTime()` that fetches the current time from a remote server. We'll represent this now with a `setTimeout()` that returns the time (like it's making a request to a slow API):

```
function getCurrentTime() {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    return new Date();
  }, 2000);
}
var currentTime = getCurrentTime()
console.log('The current time is: ' + currentTime);
```

Our `console.log()` log value will return the timeout handler id, which is definitely *not* the current time. Traditionally, we can update the code using a callback to get called when the time is available:

```
function getCurrentTime(callback) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    var currentTime = new Date();
    callback(currentTime);
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
});
```

What if there is an error with the rest? How do we catch the error and define a retry or error state?

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
    } else {
      onFail('Unknown error');
    }
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
}, function(error) {
  console.log('There was an error fetching the time');
});
```

Now, what if we want to make a request based upon the first request's value? As a short example, let's reuse the `getCurrentTime()` function inside again (as though it were a second method, but allows us to avoid adding another complex-looking function):

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    console.log(didSucceed);
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
    } else {
      onFail('Unknown error');
    }
  }, 2000);
}
getCurrentTime(function(currentTime) {
  getCurrentTime(function(newCurrentTime) {
    console.log('The real current time is: ' + currentTime);
  }, function(nestedError) {
    console.log('There was an error fetching the second time');
  })
}, function(error) {
  console.log('There was an error fetching the time');
});
```

Dealing with asynchronousity in this way can get complex quickly. In addition, we could be fetching values from a previous function call, what if we only want to get one... there are a lot of tricky cases to deal with when dealing with values that are not yet available when our app starts.

Enter Promises

Using promises, on the other hand helps us avoid a lot of this complexity (although is not a silver bullet solution). The previous code, which could be called spaghetti code can be turned into a neater, more synchronous-looking version:

```
function getCurrentTime() {
  // Get the current 'global' time from an API using Promise
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      var didSucceed = Math.random() >= 0.5;
      didSucceed ? resolve(new Date()) : reject('Error');
    }, 2000);
  })
}
getCurrentTime()
  .then(currentTime => getCurrentTime())
  .then(currentTime => {
    console.log('The current time is: ' + currentTime);
    return true;
  })
  .catch(err => console.log('There was an error:' + err))
```

This previous source example is a bit cleaner and clear as to what's going on and avoids a lot of tricky error handling/catching.

To catch the value on success, we'll use the `then()` function available on the `Promise` instance object. The `then()` function is called with whatever the return value is of the promise itself. For instance, in the example above, the `getCurrentTime()` function resolves with the `currentTime()` value (on successful completion) and calls the `then()` function on the return value (which is another promise) and so on and so forth.

To catch an error that occurs anywhere in the promise chain, we can use the `catch()` method.

We're using a promise chain in the above example to create a *chain* of actions to be called one after another. A promise chain sounds complex, but it's fundamentally simple. Essentially, we can "synchronize" a call to multiple asynchronous operations in succession. Each call to `then()` is called with the previous `then()` function's return value.

For instance, if we wanted to manipulate the value of the `getCurrentTime()` call, we can add a link in the chain, like so:

```
getCurrentTime()
  .then(currentTime => getCurrentTime())
  .then(currentTime => {
    return 'It is now: ' + currentTime;
  })
  // this logs: "It is now: [current time]"
  .then(currentTimeMessage => console.log(currentTimeMessage))
  .catch(err => console.log('There was an error:' + err))
```

Single-use guarantee

A promise only ever has one of three states at any given time:

- pending
- fulfilled (resolved)
- rejected (error)

A *pending* promise can only ever lead to either a fulfilled state or a rejected state *once and only once*, which can avoid some pretty complex error scenarios. This means that we can only ever return a promise once. If we want to rerun a function that uses promises, we need to create a *new* one.

Creating a promise

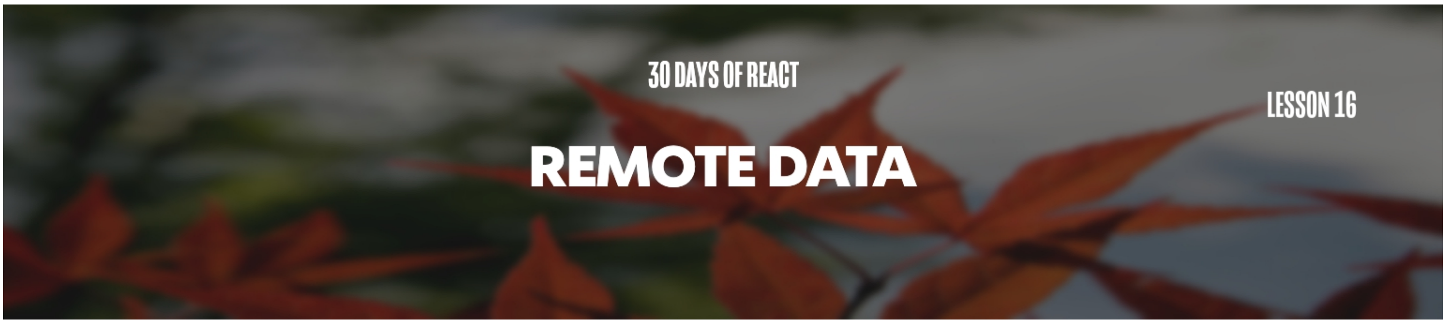
We can create new promises (as the example shows above) using the `Promise` constructor. It accepts a function that will get run with two parameters:

- The `onSuccess` (or `resolve`) function to be called on success resolution
- The `onFail` (or `reject`) function to be called on failure rejection

Recalling our function from above, we can see that we call the `resolve()` function if the request succeeded and call the `reject()` function if the method returns an error condition.

```
var promise = new Promise(function(resolve, reject) {
  // call resolve if the method succeeds
  resolve(true);
})
promise.then(bool => console.log('Bool is true'))
```

Now that we know what promises are, how to use, and how to create them, we can actually get down to using the `fetch()` library we installed yesterday. dd



Displaying Remote Data

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-16/post.md)

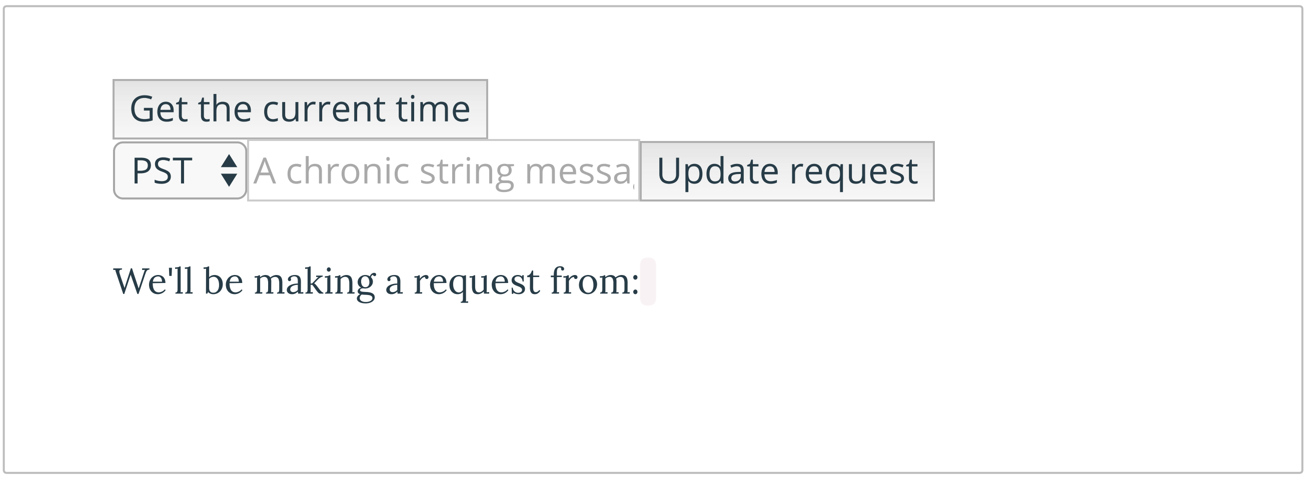
Our front-end applications are only as interesting as the data we display in them. Today, let's actually start making a request for data and get it integrated into our app.

As of today, we've worked through promises, built our app using the `npm` packager, installed our remote object fetching library (`whatwg-fetch`) and we're finally ready to integrate remote data into our application.

Fetching data

Let's get into using the `fetch` library we installed on day 14 (/articles/30-days-of-react/14-ajax).

For simplicity purposes, let's break out our demo from yesterday where we fetched the current time from an API server:



This demo react component makes a request to the API server and reports back the current time from it's clock. Before we add the call to fetch, let's create a few stateful components we'll use to display the time and update the time request.

> ### Walls of code warning
>
> We realize the next few lines are *walls of code*, which we generally try to avoid, especially without discussing how they work. However, since we're not talking about how to create a component in detail here, yet we still want to fill out a complete component, we've made an exception.
>
> Please leave us feedback (links at the bottom) if you prefer us to change this approach for today.

First, the basis of the wrapper component which will show and fetch the current time looks like the following. Let's copy and paste this code into our app at `src/App.js`:

```
import React from 'react';
import 'whatwg-fetch';
import './App.css';
import TimeForm from './TimeForm';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.fetchCurrentTime = this.fetchCurrentTime.bind(this);
    this.handleFormSubmit = this.handleFormSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);

    this.state = {
      currentTime: null, msg: 'now'
    }
  }

  // methods we'll fill in shortly
  fetchCurrentTime() {}
  getApiUrl() {}
  handleFormSubmit(evt) {}
  handleChange(newState) {}

  render() {
    const {currentTime, tz} = this.state;
    const apiUrl = this.getApiUrl();

    return (
      <div>
        {!currentTime &&
          <button onClick={this.fetchCurrentTime}>
            Get the current time
          </button>}
        {currentTime && <div>The current time is: {currentTime}</div>}
        <TimeForm
          onFormSubmit={this.handleFormSubmit}
          onFormChange={this.handleChange}
          tz={tz}
          msg={'now'}
        />
        <p>We'll be making a request from: <code>{apiUrl}</code></p>
```

```
      </div>
    )
  }
}

export default App;
```

The previous component is a basic stateful React component as we've created. Since we'll want to show a form, we've included the intended usage of the `TimeForm` let's create next.

Let's create this component in our react app using `create-react-app`. Add the file `src/TimeForm.js` into our project:

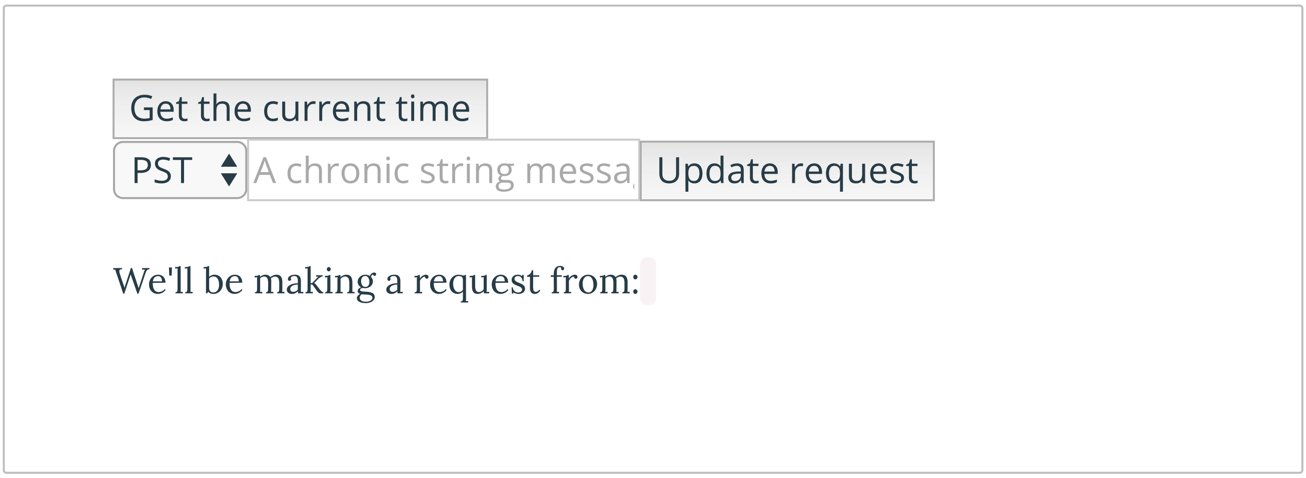
```
touch src/TimeForm.js
```

Now let's add content. We'll want our `TimeForm` to take the role of allowing the user to switch between timezones in their browser. We can handle this by creating a *stateful* component we'll call the `TimeForm`. Our `TimeForm` component might look like the following:

```
import React from 'react'
const timezones = ['PST', 'MST', 'MDT', 'EST', 'UTC']

export class TimeForm extends React.Component {
  constructor(props) {
    super(props);

    this._changeTimezone = this._changeTimezone.bind(this);
    this._handleFormSubmit = this._handleFormSubmit.bind(this);
    this._handleChange = this._handleChange.bind(this);
    this._changeMsg = this._changeMsg.bind(this);

    const {tz, msg} = this.props;
    this.state = {tz, msg};
  }

  _handleChange(evt) {
    typeof this.props.onFormChange === 'function' &&
      this.props.onFormChange(this.state);
  }

  _changeTimezone(evt) {
    const tz = evt.target.value;
    this.setState({tz}, this._handleChange);
  }

  _changeMsg(evt) {
    const msg =
      encodeURIComponent(evt.target.value).replace(/%20/g, '+');
    this.setState({msg}, this._handleChange);
  }

  _handleFormSubmit(evt) {
    evt.preventDefault();
    typeof this.props.onFormSubmit === 'function' &&
      this.props.onFormSubmit(this.state);
  }

  render() {
    const {tz} = this.state;

    return (
```

```
      <form onSubmit={this._handleFormSubmit}>
        <select
          onChange={this._changeTimezone}
          defaultValue={tz}>
          {timezones.map(t => {
            return (<option key={t} value={t}>{t}</option>)
          })}
        </select>
        <input
          type="text"
          placeholder="A chronic string message (such as 7 hours from now)"
          onChange={this._changeMsg}
        />
        <input
          type="submit"
          value="Update request"
        />
      </form>
    )
  }
}

export default TimeForm;
```

With these Components created, let's load up our app in the browser after running it with `npm start` and we'll see our form (albeit not incredibly beautiful yet). Of course, at this point, we won't have a running component as we haven't implemented our data fetching. Let's get to that now.



Fetching data

As we said yesterday, we'll use the `fetch()` API with promise support. When we call the `fetch()` method, it will return us a promise, where we can handle the request however we want. We're going to make a request to our now-based API server (so start-up might be slow if it hasn't been run in a while).

We're going to be building up the URL we'll request as it represents the time query we'll request on the server.

I've already defined the method `getApiUrl()` in the `App` component, so let's fill that function in.

The chronic api server accepts a few variables that we'll customize in the form. It will take the timezone to along with a chronic message. We'll start simply and ask the chronic library for the `pst` timezone and the current time (`now`):

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentTime: null, msg: 'now', tz: 'PST'
    }
  }
  // ...
  getApiUrl() {
    const {tz, msg} = this.state;
    const host = 'https://andthetimeis.com';
    return host + '/' + tz + '/' + msg + '.json';
  }
  // ...
export default App;
```

Now, when we call `getApiUrl()`, the URL of the next request will be returned for us. Now, finally, let's implement our `fetch()` function. The `fetch()` function accepts a few arguments that can help us customize our requests.

The most basic `GET` request can just take a single URL endpoint. The return value on `fetch()` is a promise object, that we explored in-depth yesterday.

Let's update our `fetchCurrentTime()` method to fetch the current time from the remote server. We'll use the `.json()` method on the response object to turn the body of the response from a JSON object into JavaScript object and then update our component by setting the response value of the `dateString` as the `currentTime` in the component state:

```
class App extends React.Component {
  // ...
  fetchCurrentTime() {
    fetch(this.getApiUrl())
      .then(resp => resp.json())
      .then(resp => {
        const currentTime = resp.dateString;
        this.setState({currentTime})
      })
  }
  // ...
}
```

The final piece of our project today is getting the data back from the form to update the parent component. That is, when the user updates the values from the `TimeForm` component, we'll want to be able to access the data in the `App` component. The `TimeForm` component already handles this process for us, so we just need to implement our form functions.

When a piece of state changes on the form component, it will call a prop called `onFormChange`. By defining this method in our `App` component, we can get access to the latest version of the form.

In fact, we'll just call `setState()` to keep track of the options the form allows the user to manipulate:

```
class App extends React.Component {

// ...
  handleChange(newState) {
    this.setState(newState);
  }

// ...
}
```

Finally, when the user submits the form (clicks on the button *or* presses enter in the input field), we'll want to make another request for the time. This means we can define our `handleFormSubmit` prop to just call the `fetchCurrentTime()` method:

```
class App extends React.Component {

// ...
  handleFormSubmit(evt) {
    this.fetchCurrentTime();
  }

// ...
}
```

We'll be making a request from: `https://andthetimeis.com/PST/now.json`

Try playing around with the demo and passing in different chronic options. It's actually quite fun.

In any case, today we worked on quite a bit to get remote data into our app. However, at this point, we only have a single page in our single page app. What if we want to show a different page in our app? Tomorrow, we're going to start adding multiple pages in our app so we can feature different views.

Client-side Routing

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-17/post.md)

Most, if not all of our applications will have multiple views in our single-page application. Let's dive right into creating multiple views for our applications using React Router.

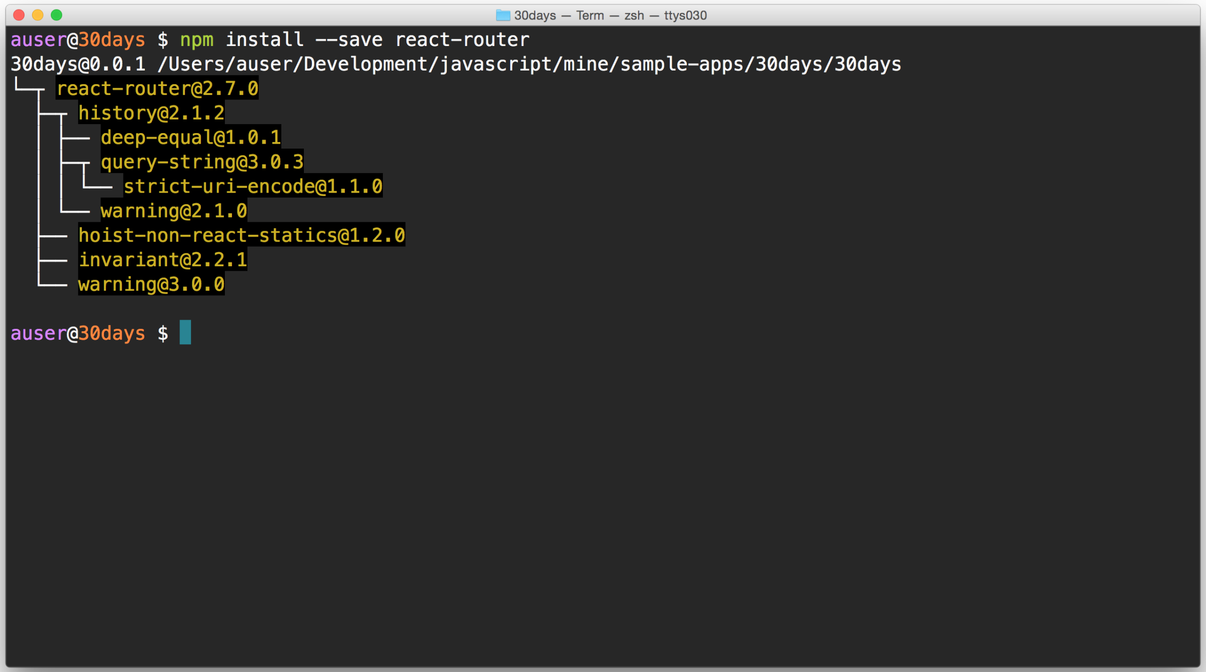
We've made it through 16 days already! Pat yourself on the back... but not for too long... there is still a lot more.

Right now, our app is limited to a single page. It's pretty rare to find any complex application that shows a single view. For instance, an application might have a login view where a user can log in or a search results page that shows a user a list of their search results. These are two different views with two different page structures.

Let's see how we can change that with our app today.

We'll use the very popular react-router (https://github.com/reactjs/react-router) library for handling different links. In order to use the `react-router` library, we'll need to install it using the `npm` package manager:

```
npm install --save react-router-dom
```



With `react-router` installed, we'll import a few packages from the library and update our app architecture. Before we make those updates, let's take a step back and from a high level look at *how* and *why* we architect our application this way.

Conceptually with React, we've seen how we can create tree structures using components and nested components. Using this perspective with a single page app with routes, we can think of the different parts of a page as children. Routing in a single page app from this perspective is the idea that we can take a part of a subtree and switch it out with another subtree. We can then *dynamically* switch out the different trees in the browser.

In *other* words, we'll define a React component that acts as a *root* component of the routable elements. We can then tell React to change a view, which can just swap out an entire React component for another one as though it's a completely different page rendered by a server.

We'll take our `App` component and define all of the different routes we can make in our app in this `App` component. We'll need to pull some components from the `react-router` package. These components we'll use to set up this structure are as follows:

`<BrowserRouter />` / `<Router />`

This is the component we'll use to define the *root* or the routing tree. The `<BrowserRouter />` component is the component where React will replace it's children on a per-route basis.

`<Route />`

We'll use the `<Route />` component to create a route available at a specific location available at a url. The `<Route />` component is mounted at page URLs that match a particular route set up in the route's configuration `props`.

One older, compatible way of handling client-side navigation is to use the `#` (hash) mark denoting the application endpoint. We'll use this method. We'll need this object imported to tell the browser this is how we want to handle our navigation.

From the app we created a few days ago's root directory, let's update our `src/App.js` to import these modules. We'll import the `BrowserRouter` using a different name syntax via ES6:

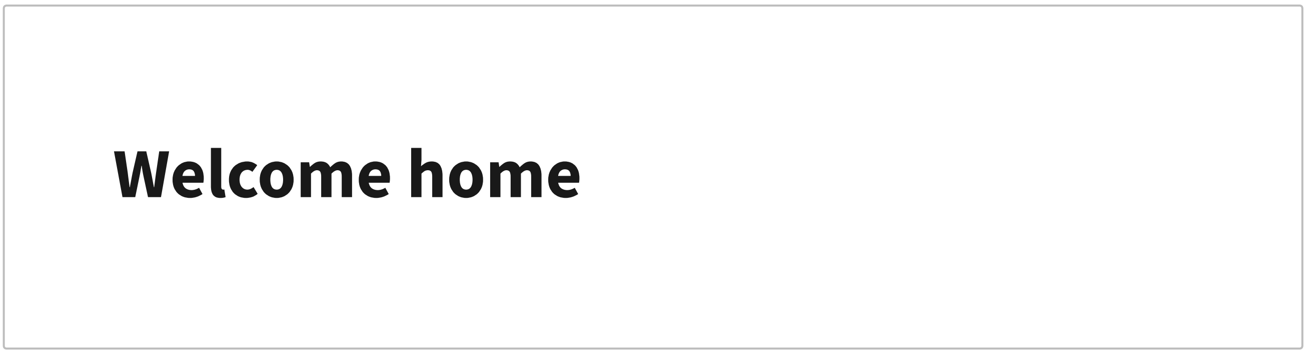
```
import React from "react";

import { BrowserRouter as Router, Route } from "react-router-dom";

export class App extends React.Component {
  render() {
    <Router>{/* routes will go here */}</Router>;
  }
}
```

Now let's define our first route. To define a route, we'll use the `<Route />` component export from `react-router` and pass it a few props:

- `path` - The path for the route to be active
- `component` - The component that defines the view of the route

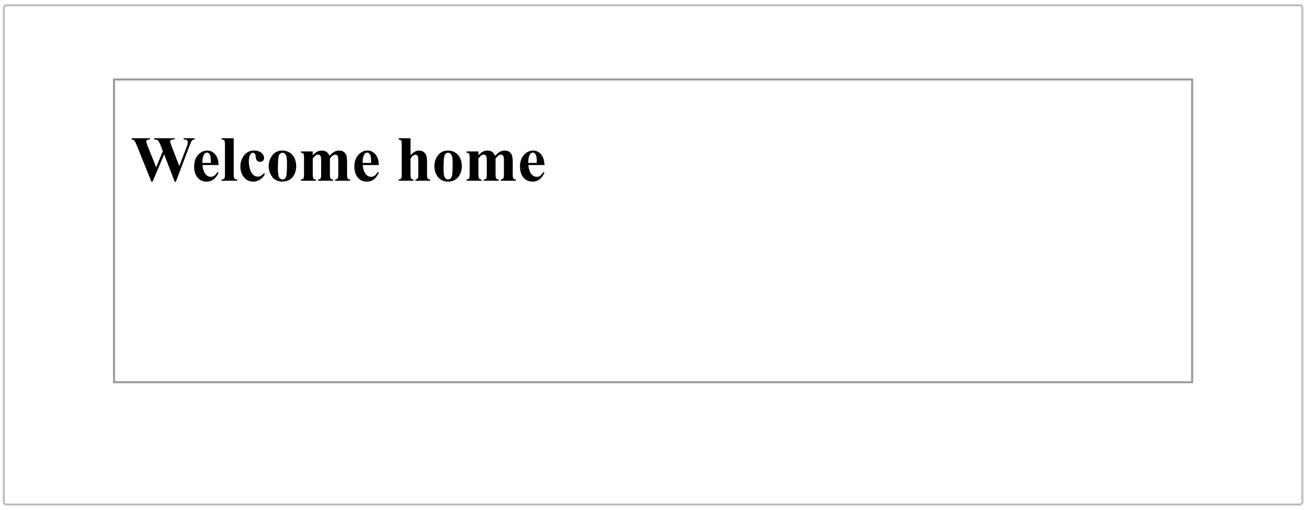
Let's define the a route at the root path `/` with a stateless component that just displays some static content:

```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <Route path="/" component={Home} />
      </Router>
    );
  }
}
```



Loading this page in the browser, we can see we get our single route at the root url. Not very exciting. Let's add a second route that shows an about page at the `/about` URL.

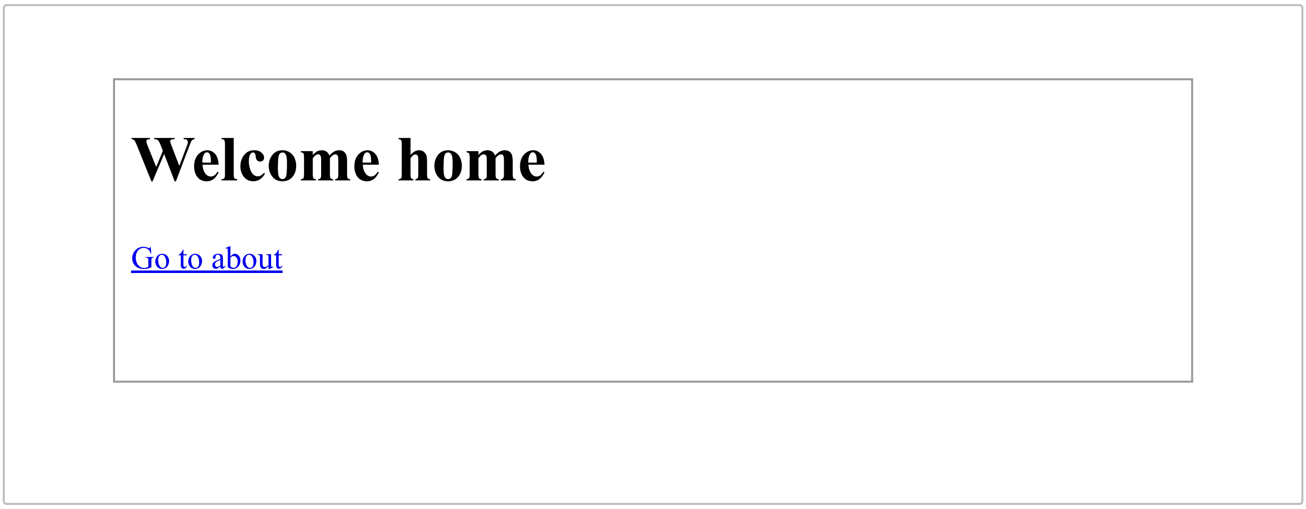
```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <div>
          <Route path="/" component={Home} />
          <Route path="/about" component={About} />
        </div>
      </Router>
    );
  }
}
```



In our view we'll need to add a link (or an anchor tag -- `<a />`) to enable our users to travel freely between the two different routes. However, using the `<a />` tag will tell the browser to treat the route like it's a server-side route. Instead, we'll need to use a different component (surprise) called: `<Link />`.

The `<Link />` component requires a prop called `to` to point to the client-side route where we want to render. Let's update our `Home` and `About` components to use the `Link`:

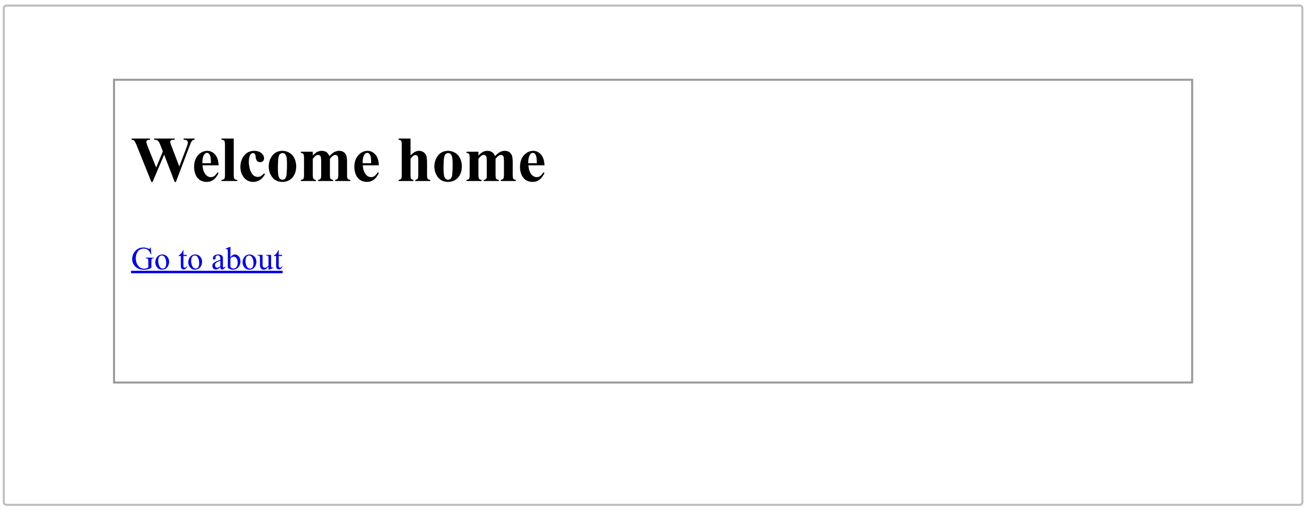
```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
const About = () => (
  <div>
    <h1>About</h1>
    <Link to="/">Go home</Link>
  </div>
);
// ...
```



Wait a minute... we don't quite want *both* routes to show up... This happens because the react router will render *all* content that matches the path (unless otherwise specified). For this case, react router supplies us with the `Switch` component.

The `<Switch />` component will *only render the first matching route* it finds. Let's update our component to use the `Switch` component. As react router will try to render *both* components, we'll need to specify that we only want an `exact` match on the root component.

```
import { BrowserRouter as Router, Route, Link, Switch } from "react-
router-dom";
// ...
const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route path="/about" component={About} />
          <Route path="/" component={Home} />
        </Switch>
      </Router>
    );
  }
}
```



Showing views

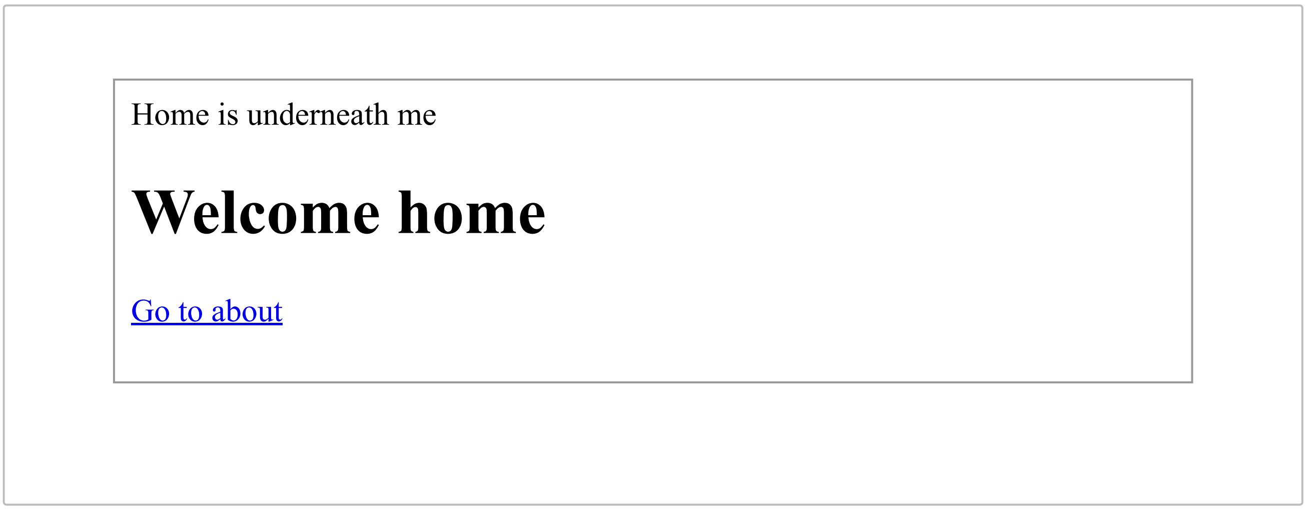
Although this is a limited introduction, we could not leave the discussion of dealing with react router without talking about the different ways we can get subcomponents to render.

We've already seen the simplest way possible, using the `component` prop, however there is a more powerful method using a prop called `render`. The `render` prop is expected to be a function that will be called with the `match` object along with the `location` and route configuration.

The `render` prop allows us to render *whatever* we want in a subroute, which includes rendering other routes. Nifty, ey? Let's see this in action:

```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
const About = ({ name }) => (
  <div>
    <h1>About {name}</h1>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route
            path="/about"
            render={renderProps => (
              <div>
                <Link to="/about/ari">Ari</Link>
                <Link to="/about/nate">Nate</Link>
                <Route
                  path="/about/:name"
                  render={renderProps => (
                    <div>
                      <About name={renderProps.match.params.name} />
                      <Link to="/">Go home</Link>
                    </div>
                  )}
                />
              </div>
            )}
          />
          <Route
            path="/"
            render={renderProps => (
              <div>
                Home is underneath me
                <Home {...this.props} {...renderProps} />
              </div>
```

```
              )}
            />
          </Switch>
        </Router>
      );
    }
  }
```



Now we have multiple pages in our application. We've looked at how we can render these routes through nested components with just a few of the exports from `react-router`.

`react-router` provides so much more functionality that we don't have time to cover in our brisk intro to routing. More information is available at:

- https://github.com/reactjs/react-router/tree/master/docs (https://github.com/reactjs/react-router/tree/master/docs)
- fullstack react routing (https://fullstackreact.com)

Tomorrow, we're going to be starting integration with Redux. Here's where we start integrating more complex data handling.

Introduction to Flux

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-18/post.md)

Handling data inside a client-side application is a complex task. Today we're looking at a one method of handling complex data proposed by Facebook called the Flux Architecture.

As our applications get bigger and more complex, we'll need a better data handling approach. With more data, we'll have more to keep track of.

Our code is required to handle more data and application state with new features. From asynchronous server responses to locally-generated, unsynchronized data, we have to not only keep track of this data, but also tie it to the view in a sane way.

Recognizing this need for data management, the Facebook team released a pattern for dealing with data called Flux (https://facebook.github.io/flux/docs/overview.html).

Today, we're going to take a look at the Flux architecture, what it is and why it exists.

What is flux

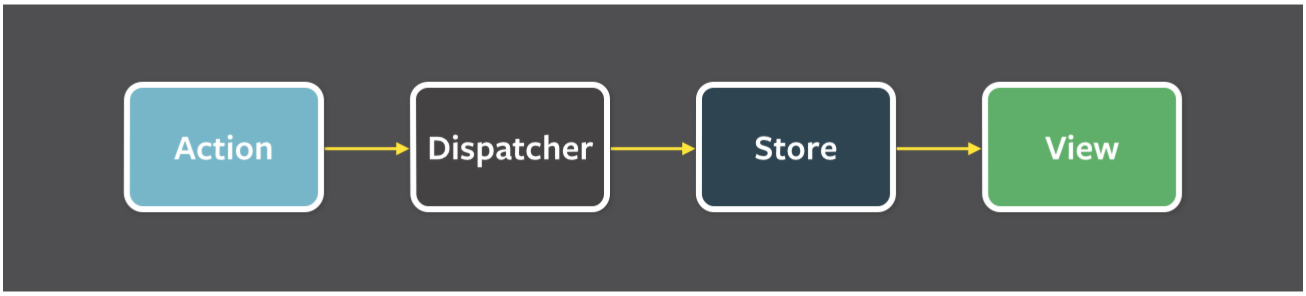
Flux is a pattern for managing how data flows through a React application. As we've seen, the preferred method of working with React components is through passing data from one parent component to it's children components. The Flux pattern makes this model the default method for handling data.

There are three distinct roles for dealing with data in the flux methodology:

- Dispatcher
- Stores
- Views (our components)

The major idea behind Flux is that there is a single-source of truth (the stores) and they can only be updated by triggering *actions*. The actions are responsible for calling the dispatcher, which the stores can *subscribe* for changes and update their own data accordingly.

When a dispatch has been triggered, and the store updates, it will emit a change event which the views can rerender accordingly.



This may seem unnecessarily complex, but the structure makes it incredibly easy to reason about where our data is coming from, what causes it's changes, how it changes, and lets us track specific user flows, etc.

The key idea behind *Flux* is:

Data flows in one direction and kept entirely in the stores.

Implementations

Although we can create our own flux implementation, many have already created some fantastic libraries we can pick from.

- Facebook's flux (https://github.com/facebook/flux)

- alt (http://alt.js.org/)
- nuclear-js (https://optimizely.github.io/nuclear-js/)
- Fluxible (http://fluxible.io/)
- reflux (https://github.com/reflux/refluxjs)
- Fluxxor (http://fluxxor.com/)
- flux-react (https://github.com/christianalfoni/flux-react)
- And more... many many more

> # Plug for fullstackreact
>
> We discuss this material in-depth about Flux, using libraries, and even implementing our own version of flux that suits us best. Check it out at fullstackreact.com (https://fullstackreact.com)

It can be pretty intense trying to pick the *right* choice for our applications. Each has their own features and are great for different reasons. However, to a large extent, the React community has focused in on using another flux tool called Redux (http://redux.js.org/).

Redux (http://redux.js.org/)

Redux is a small-ish library that takes it's design inspiration from the Flux pattern, but is not itself a pure flux implementation. It provides the same general principles around how to update the data in our application, but in slightly different way.

Unlike Flux, Redux does not use a dispatcher, but instead it uses pure functions to define data mutating functions. It still uses stores and actions, which can be tied directly to React components.

The 3 major principles (http://redux.js.org/docs/introduction/ThreePrinciples.html) of Redux we'll keep in mind as we implement Redux in our app are:

- Updates are made with pure functions (in reducers)
- `state` is a read-only property
- `state` is the single source of truth (there is only one `store` in a Redux app)

One big difference with Redux and Flux is the concept of middleware. Redux added the idea of middleware that we can use to manipulate actions as we receive them, both coming in and heading out of our application. We'll discuss them in further detail in a few days.

In any case, this is a lot of introduction to the flux pattern. Tomorrow we'll actually start moving our data to use Redux.

Data Management with Redux

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-19/post.md)

With the knowledge of flux and Redux, let's integrate Redux in our application and walk through connected applications.

Yesterday, we discussed (in light detail) the reason for the Flux pattern, what it is, the different options we have available to us, as well as introduced Redux (http://redux.js.org/).

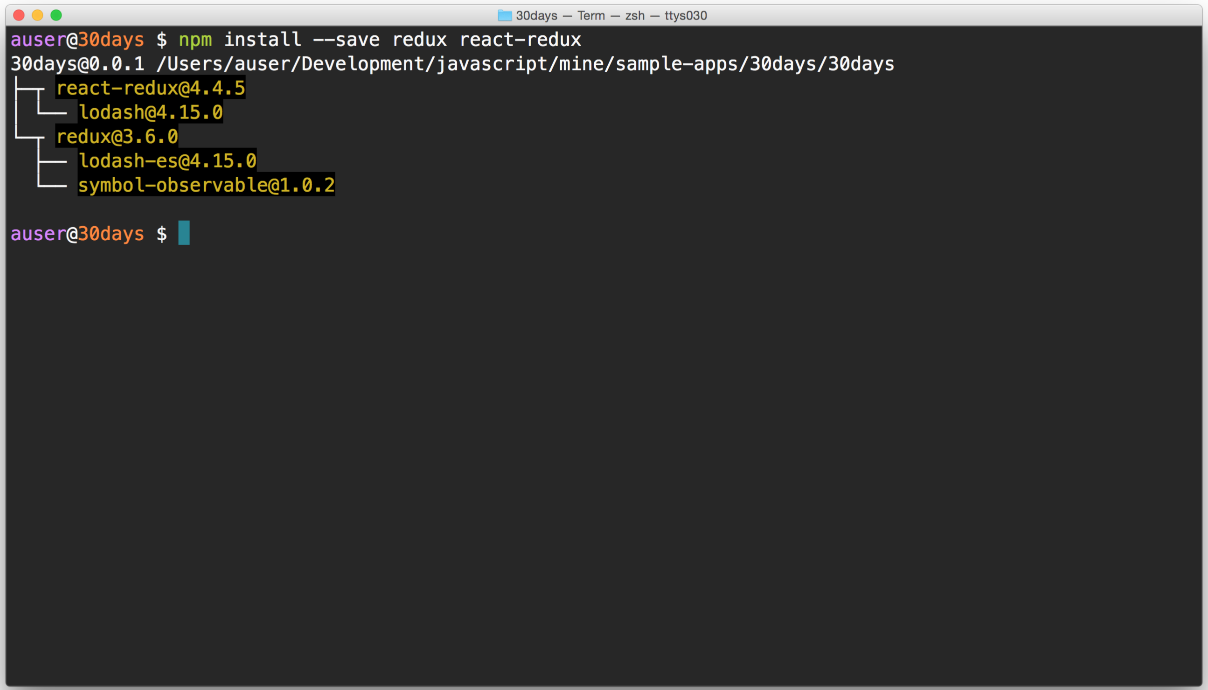
Today, we are going to get back to code and on to adding Redux in our app. The app we're building with it right now is bare-bones simple, which will just show us the last time the page fetched the current time. For simplicity for now, we won't call out to a remote server, just using the JavaScript `Date` object.

The first thing we'll have to do to use Redux is install the library. We can use the `npm` package manager to install `redux`. In the root directory of our app we previously built, let's run the `npm install` command to install redux:

```
npm install --save redux
```

We'll also need to install another package that we'll use with redux, the `react-redux` that will help us tie together `react` and `redux`:

```
npm install --save react-redux
```



Configuration and setup

The next bit of work we need to do is to set up Redux inside of our app. We'll need to do the following to get it set up:

1. Define reducers
2. Create a store
3. Create action creators
4. Tie the store to our React views
5. Profit

> No promises on step 5, but it would be nice, eh?

Precursor

We'll talk terminology as we go, so take this setup discussion lightly (implementing is more important to get our fingers moving). We'll restructure our app just slightly (annoying, I know... but this is the last time) so we can create a wrapper component to provide data down through our app.

When we're complete, our app tree will have the following shape:

```
[Root] -> [App] -> [Router/Routes] -> [Component]
```

Without delaying any longer, let's move our `src/App.js` into the `src/containers` directory and we'll need to update some of the paths from our imports at the same time. We'll be using the react router material we discussed a few days ago.

We'll include a few routes with the `<Switch />` statement to ensure only one shows up at a time.

```
import React from "react";

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

// We'll load our views from the `src/views`
// directory
import Home from "./views/Home/Home";
import About from "./views/About/About";

const App = props => {
  return (
    <Router>
      <Switch>
        <Route path="/about" component={About} />
        <Route path="*" component={Home} />
      </Switch>
    </Router>
  );
};

export default App;
```

In addition, we'll need to create a new container we'll call `Root` which will wrap our entire `<App />` component and make the store available to the rest of the app. Let's create the `src/containers/Root.js` file:

```
touch src/containers/Root.js
```

For the time being, we'll use a placeholder component here, but we'll replace this content as we talk about the store. For now, let's export *something*:

```
import React from "react";
import App from "./App";

const Root = props => {
  return <App />;
};

export default Root;
```

Finally, let's update the route that we render our app in the `src/index.js` file to use our new `Root` container instead of the `App` it previously used.

```
import React from "react";
import ReactDOM from "react-dom";
import Root from "./containers/Root";
import "./index.css";

ReactDOM.render(<Root />, document.getElementById("root"));
```

Adding in Redux

Now with a solid app structure in place, we can start to add in Redux. The steps we'll take to tie in some Redux structure are generally all the same for most every application we'll build. We'll need to:

1. Write a root reducer
2. Write actionCreators

3. Configure the store with the rootReducer, the store, and the app
4. Connect the views to the actionCreators

We'll purposefully be keeping this high-level introduction a tad short, so hang tight if that's a mouthful, it will all make more sense shortly.

Let's setup the structure to allow us to add redux. We'll do almost all of our work in a `src/redux` directory. Let's create that directory.

```
mkdir -p src/redux
touch src/redux/configureStore.js
touch src/redux/reducers.js
```

Let's start by creating our reducer first. Although it sounds complex, a reducer is actually pretty straight-forward with some experience. A reducer is *literally* only a function. It's sole responsibility is to return a representation of the *next* state.

In the Redux pattern, unlike flux we are only handling *one* global store for the *entire* application. This makes things much easier to deal with as there's a single place for the data of our application to live.

The *root* reducer function is responsible to return a representation of the current global state of the application. When we dispatch an action on the store, this reducer function will be called with the current state of the application and the action that causes the state to update.

Let's build our root reducer in a file at `src/redux/reducers.js`.

```
// Initial (starting) state
export const initialState = {
  currentTime: new Date().toString()
};

// Our root reducer starts with the initial state
// and must return a representation of the next state
export const rootReducer = (state = initialState, action) => {
  return state;
};
```

In the function, we're defining the first argument to start out as the initial state (the first time it runs, the `rootReducer` is called with no arguments, so it will always return the `initialState` on the first run).

That's the rootReducer for now. As it stands right now, the state always will be the same value as the initialState. In our case, this means our data tree has a single key of `currentTime`.

What is an action?

The second argument here is the action that gets dispatched from the store. We'll come back to what that means exactly shortly. For now, let's look at the action.

At the very minimum, an action *must* include a `type` key. The `type` key can be any value we want, but it must be present. For instance, in our application, we'll occassionally dispatch an action that we want to tell the store to get the *new* current time. We might call this action a string value of `FETCH_NEW_TIME`.

The action we might dispatch from our store to handle this update looks like:

```
{
  type: "FETCH_NEW_TIME"
}
```

As we'll by typing this string a lot and we want to avoid a possible mispelling somewhere, it's common to create a `types.js` file that exports the action types as constants. Let's follow this convention and create a `src/redux/types.js` file:

```
export const FETCH_NEW_TIME = "FETCH_NEW_TIME";
```

Instead of calling the action with the hard-coded string of 'FETCH_NEW_TIME', we'll reference it from the `types.js` file:

```
import * as types from './types';

{
  type: types.FETCH_NEW_TIME,
}
```

When we want to send data along with our action, we can add any keys we want to our action. We'll commonly see this called `payload`, but it can be called anything. It's a convention to call additional information the `payload`.

Our `FETCH_NEW_TIME` action will send a payload with the new current time. Since we want to send a *serializable* value with our actions, we'll send the string value of the new current time.

```
{
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString() // Any serializable value
}
```

Back in our reducer, we can check for the action type and take the appropriate steps to create the next state. In our case, we'll just store the `payload`. If the `type` of the action is `FETCH_NEW_TIME`, we'll return the new currentTime (from our action payload) and the rest of the state (using the ES6 spread syntax):

```
export const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case types.FETCH_NEW_TIME:
      return { ...state, currentTime: action.payload };
    default:
      return state;
  }
};
```

Remember, the reducers *must* return a state, so in the default case, make sure to return the current state *at the very minimum*.

> ## Keep it light
>
> Since the reducer functions run everytime an action is dispatched, we want to make sure these functions are as simple and fast as possible. We don't want them to cause any side-effects or have much delay at all.
>
> We'll handle our side-effects *outside* of the reducer in the action creators.

Before we look at action creators (and why we call them action creators), let's hook up our store to our application.

We'll be using the `react-redux` package to connect our views to our redux store. Let's make sure to install this package using `npm`:

```
npm install --save react-redux
```

Hooking up the store to the view

The `react-redux` package exports a component called `Provider`. The `Provider` component makes the store available to all of our container components in our application without needing for us to need to pass it in manually every time.

The `Provider` component expects a `store` prop that it expects to be a valid redux store, so we'll need to complete a `configureStore` function before our app will run without error. For now, let's hook up the `Provider` component in our app. We'll do this by updating our wrapper `Root` component we previously created to use the `Provider` component.

```
import { Provider } from "react-redux";
// ...
const Root = props => {
  // ...

  return (
    <Provider store={store}>
      <App />
    </Provider>
  );
};
```

Notice we're sending in the `store` value to our `Provider` component... but we haven't created the store yet! Let's fix that now.

Configuring the store

In order to create a store, we'll use the new `src/redux/configureStore.js` to export a function which will be responsible for creating the store.

How do we create a store?

The `redux` package exports a function called `createStore` which will create the actual store for us, so let's open up the `src/redux/configureStore.js` file and export a function (we'll define shortly) called `configureStore()` and import the `createStore` helper:

```
import { createStore } from "redux";
// ...
export const configureStore = () => {
  // ...
};
// ...
export default configureStore;
```

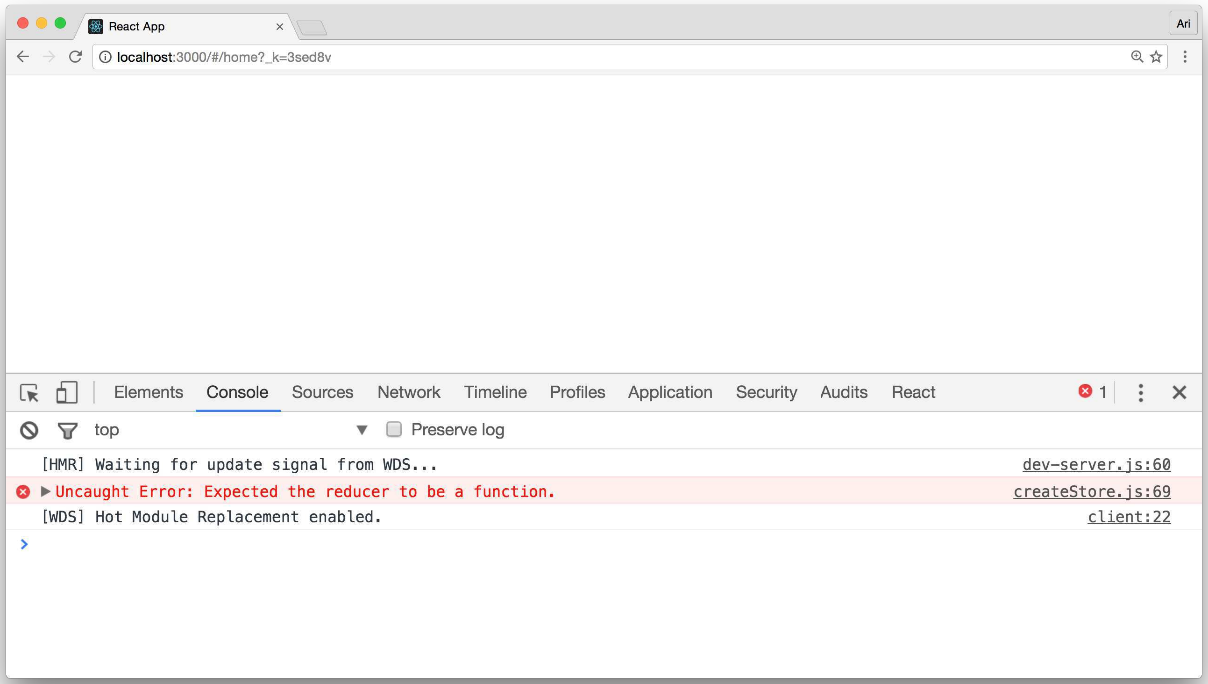
We don't actually return anything in our store quite yet, so let's actually create the `redux` store using the `createStore` function we imported from redux:

```
import { createStore } from "redux";

export const configureStore = () => {
  const store = createStore();

  return store;
};

export default configureStore;
```

Now let's update our `Root.js` file with an instance of the `store` created by calling the `configureStore()` function.

```
// ...
import configureStore from "../redux/configureStore";
const Root = props => {
  const store = configureStore();

  return (
    <Provider store={store}>
      <App />
    </Provider>
  );
};
```

If we load our page in the browser, we'll see we have one giant error and no page gets rendered.



The error redux is giving us is telling us that we don't have a reducer inside our store. Without a reducer, it won't know what to do with actions or how to create the state, etc. In order to move beyond this error, we'll need to reference our rootReducer we created.

The `createStore` function expects us to pass the rootReducer in as the first argument. It'll also expect the initial state to be passed in as the second argument. We'll import both of these values from the `reducers.js` file we created.

```
import { rootReducer, initialState } from "./reducers";
// ...
export const configureStore = () => {
  const store = createStore(
    rootReducer, // root reducer
    initialState // our initialState
  );

  return store;
};
```

Connecting the view (cont'd)

Everything in our app is set-up to use Redux without *too* much overhead. One more convenience that `redux` offers is a way to *bind* pieces of the state tree to different components using the `connect()` function exported by the `react-redux` package.

The `connect()` function returns a function that expects the 1st argument to be that of a component. This is often called a higher-order component.

The `connect()` function expects us to pass in at least one argument to the function (but often we'll pass in two). The first argument it expects is a function that will get called with the `state` and expects an object in return that connects data to the view. Let's see if we can demystify this behavior in code.

We'll call this function the `mapStateToProps` function. Since it's responsibility is to map the state to an object which is merged with the component's original `props`.

Let's create the Home view in `src/views/Home.js` and use this `connect()` function to bind the value of `currentTime` in our state tree.

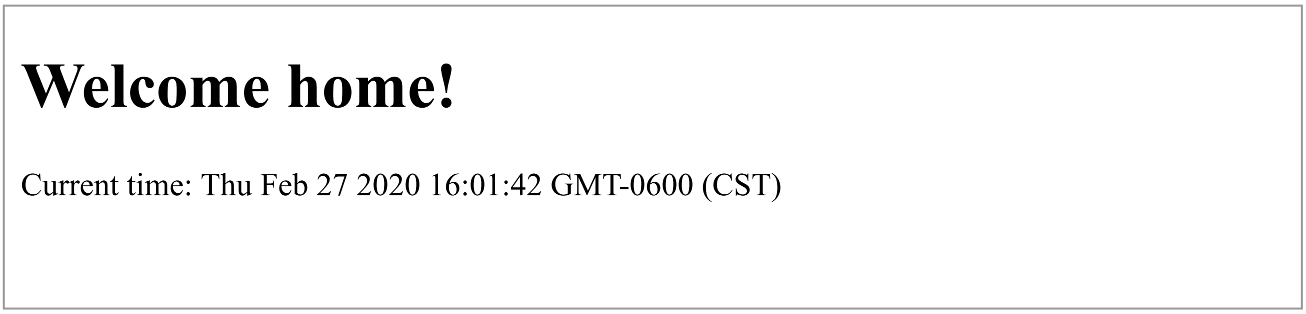
```
import { connect } from "react-redux";
// ...
const mapStateToProps = state => {
  return {
    currentTime: state.currentTime
  };
};
export default connect(mapStateToProps)(Home);
```

This `connect()` function *automatically* passes any of the keys in the function's first argument as `props` to the `Home` component.

In our demo's case, the `currentTime` prop in the `Home` component will be mapped to the state tree key at `currentTime`. Let's update the `Home` component to show the value in the `currentTime`:

```
const Home = props => {
  return (
    <div className="home">
      <h1>Welcome home!</h1>
      <p>Current time: {props.currentTime}</p>
    </div>
  );
};
```

Although this demo isn't very interesting, it shows we have our `Redux` app set up with our `data` committed to the global state and our view components mapping the data.



Tomorrow we're going to start triggering updates into our global state through action creators as well as work through combining multiple redux modules together.

Redux actions

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-20/post.md)

With Redux in place, let's talk about how we actually modify the Redux state from within our applications.

Yesterday we went through the difficult part of integrating our React app with Redux. From here on out, we'll be defining functionality with our Redux setup.

As it stands now, we have our demo application showing the current time. But there currently isn't any way to update to the new time. Let's modify this now.

Triggering updates

Recall that the only way we can change data in Redux is through an action creator. We created a redux store yesterday, but we haven't created a way for us to update the store.

Welcome home!

Current time: Thu Feb 27 2020 16:01:45 GMT-0600 (CST)

What we *want* is the ability for our users to update the time by clicking on a button. In order to add this functionality, we'll have to take a few steps:

1. Create an actionCreator to *dispatch* the action on our store
2. Call the actionCreator `onClick` of an element
3. Handle the action in the reducer

We already implemented the third step, so we only have two things to do to get this functionality working as we expect.

Yesterday, we discussed what actions are, but not really why we are using this thing called actionCreators or what they are.

As a refresher, an action is a simple object that *must* include a `type` value. We created a `types.js` file that holds on to action type constants, so we can use these values as the `type` property.

```
export const FETCH_NEW_TIME = 'FETCH_NEW_TIME';
export const LOGIN = 'USER_LOGIN';
export const LOGOUT = 'USER_LOGOUT';
```

As a quick review, our actions can be any object value that has the `type` key. We can send data along with our action (conventionally, we'll pass extra data along as the `payload` of an action).

```
{
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString()
}
```

Now we need to *dispatch* this along our `store`. One way we could do that is by calling the `store.dispatch()` function.

```
store.dispatch({
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString()
})
```

However, this is pretty poor practice. Rather than dispatch the action directly, we'll use a function to return an action... the function will *create* the action (hence the name: actionCreator). This provides us with a better testing story (easy to test), reusability, documentation, and encapsulation of logic.

Let's create our first `actionCreator` in a file called `redux/actionCreators.js`. We'll export a function who's entire responsibility is to return an appropriate action to dispatch on our store.

```
import * as types from './types';

export const fetchNewTime = () => ({
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString(),
})
```

Now if we call this function, *nothing* will happen except an action object is returned. How do we get this action to dispatch on the store?

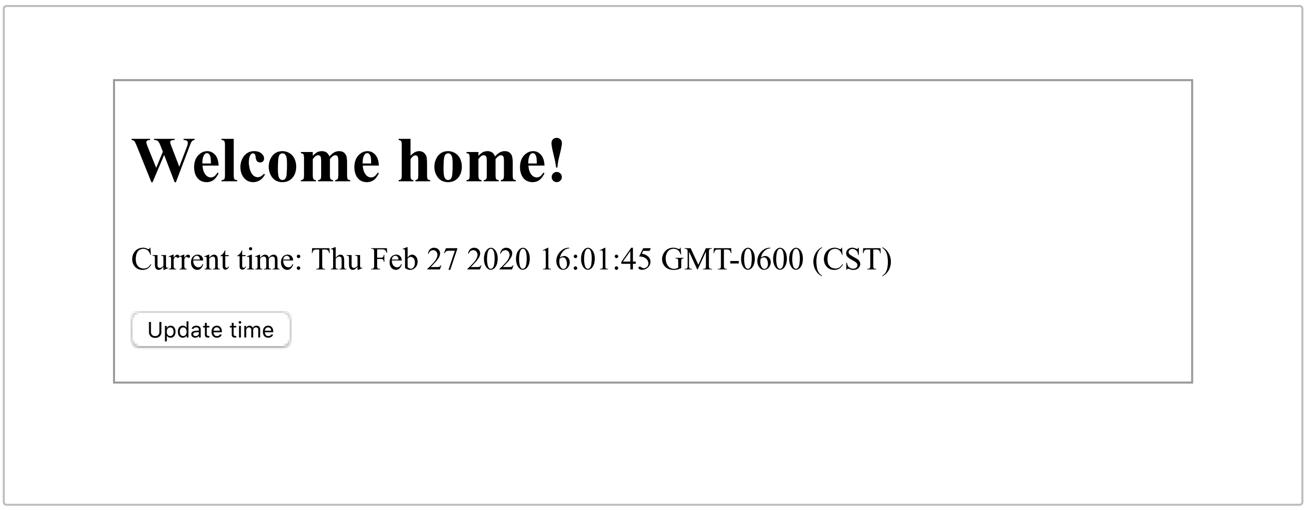
Recall we used the `connect()` function export from `react-redux` yesterday? The first argument is called `mapStateToProps`, which maps the state to a prop object. The `connect()` function accepts a second argument which allows us to map functions to props as well. It gets called with the `dispatch` function, so here we can *bind* the function to call `dispatch()` on the store.

Let's see this in action. In our `src/views/Home/Home.js` file, let's update our call to connect by providing a second function to use the actionCreator we just created. We'll call this function `mapDispatchToProps`.

```
import { fetchNewTime } from '../../../redux/actionCreators';
  // ...
const mapDispatchToProps = dispatch => ({
  updateTime: () => dispatch(fetchNewTime())
})
  // ...
export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(Home);
```

Now the `updateTime()` function will be passed in as a prop and will call `dispatch()` when we fire the action. Let's update our `<Home />` component so the user can press a button to update the time.

```
const Home = (props) => {
  return (
    <div className="home">
      <h1>Welcome home!</h1>
      <p>Current time: {props.currentTime}</p>
      <button onClick={props.updateTime}>
        Update time
      </button>
    </div>
  );
}
```



Although this example isn't that exciting, it does showcase the features of redux pretty well. Imagine if the button makes a fetch to get new tweets or we have a socket driving the update to our redux store. This basic example demonstrates the full functionality of redux.

Multi-reducers

As it stands now, we have a single reducer for our application. This works for now as we only have a small amount of simple data and (presumably) only one person working on this app. Just imagine the headache it would be to develop with one gigantic switch statement for *every single piece of data* in our apps...

Ahhhhhhhhhhhh...

Redux to the rescue! Redux has a way for us to split up our redux reducers into multiple reducers, each responsible for only a leaf of the state tree.

We can use the `combineReducers()` export from `redux` to compose an object of reducer functions. For every action that gets triggered, each of these functions will be called with the corresponding action. Let's see this in action.

Let's say that we (perhaps more realistically) want to keep track of the current user. Let's create a `currentUser` redux module in... you guessed it: `src/redux/currentUser.js`:

```
touch src/redux/currentUser.js
```

We'll export the same four values we exported from the `currentTime` module... of course, this time it is specific to the currentUser. We've added a basic structure here for handling a current user:

```
import * as types from './types'

export const initialState = {
  user: {},
  loggedIn: false
}

export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case types.LOGIN:
      return {
        ...state, user: action.payload, loggedIn: true};
    case types.LOGOUT:
      return {
        ...state, user: {}, loggedIn: false};
    default:
      return state;
  }
}
```

Let's update our `configureStore()` function to take these branches into account, using the `combineReducers` to separate out the two branches

```
import { createStore, combineReducers } from 'redux';

import { rootReducer, initialState } from './reducers'
import { reducer, initialState as userInitialState } from
'./currentUser'

export const configureStore = () => {
  const store = createStore(
    combineReducers({
      time: rootReducer,
      user: reducer
    }), // root reducer
    {
      time: initialState,
      user: userInitialState
    }, // our initialState
  );

  return store;
}

export default configureStore;
```

Let's also update our `Home` component `mapStateToProps` function to read it's value from the `time` reducer

```
// ...
const mapStateToProps = state => {
  // our redux store has `time` and `user` states
  return {
    currentTime: state.time.currentTime
  };
};
// ...
```

Now we can create the `login()` and `logout()` action creators to send along the action on our store.

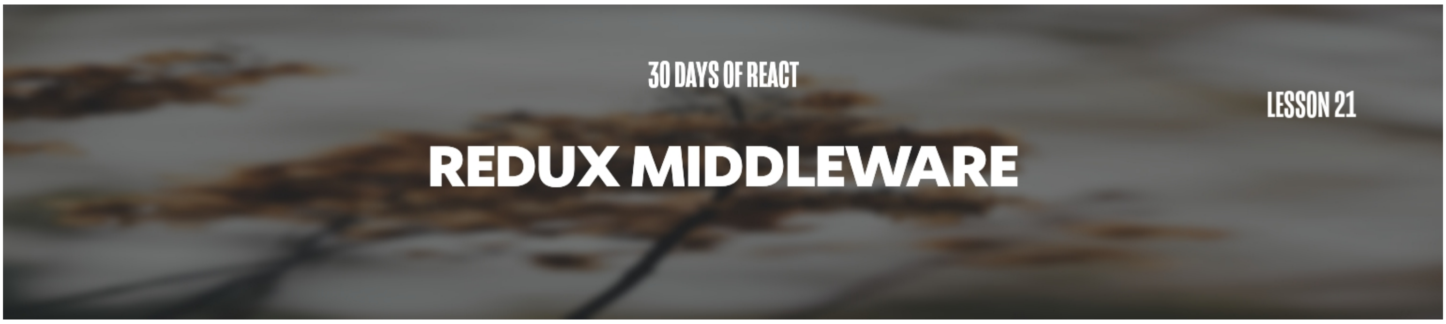
```
export const login = (user) => ({
  type: types.LOGIN,
  payload: user
})
  // ...
export const logout = () => ({
  type: types.LOGOUT,
})
```

Now we can use the actionCreators to call `login` and `logout` just like the `updateTime()` action creator.

Phew! This was another hefty day of Redux code. Today, we completed the circle between data updating and storing data in the global Redux state. In addition, we learned how to extend Redux to use multiple reducers and actions as well as multiple connected components.

However, we have yet to make an asynchronous call for off-site data. Tomorrow we'll get into how to use middleware with Redux, which will give us the ability to handle fetching remote data from within our app and still use the power of Redux to keep our data.

Good job today and see you tomorrow!



Redux Middleware

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-21/post.md)

Today, we're looking at the Redux method of managing complex state changes in our code using Redux middleware.

Yesterday we connected the dots with Redux, from working through reducers, updating action creators, and connecting Redux to React components. **Redux middleware** unlocks even more power which we'll touch on today.

Redux middleware

Middleware generally refers to software services that "glue together" separate features in existing software. For Redux, middleware provides a third-party extension point between dispatching an action and handing the action off to the reducer:

[Action] <-> [Middleware] <-> [Dispatcher]

Examples of middleware include logging, crash reporting, routing, handling asynchronous requests, etc.

Let's take the case of handling asynchronous requests, like an HTTP call to a server. Middleware is a great spot to do this.

Our API middleware

We'll implement some middleware that will handle making asynchronous requests on our behalf.

Middleware sits between the action and the reducer. It can listen for all dispatches and execute code with the details of the actions and the current states. Middleware provides a powerful abstraction. Let's see exactly how we can use it to manage our own.

Continuing with our `currentTime` redux work from yesterday, let's build our middleware to fetch the current time from the server we used a few days ago to actually GET the time from the API service.

Before we get too much further, let's pull out the `currentTime` work from the `rootReducer` in the `reducers.js` file out to it's own file. We left the root reducer in a state where we kept the `currentTime` work in the root reducer. More conventionally, we'll move these in their own files and use the `rootReducer.js` file (which we called `reducers.js`) to hold just the main combination reducer.

First, let's pull the work into it's own file in `redux/currentTime.js`. We'll export two objects from here (and each reducer):

- `initialState` - the initial state for this branch of the state tree
- `reducer` - this branch's reducer

```
import * as types from './types';

export const initialState = {
  currentTime: new Date().toString(),
}

export const reducer = (state = initialState, action) => {
  switch(action.type) {
    case types.FETCH_NEW_TIME:
      return { ...state, currentTime: action.payload}
    default:
      return state;
  }
}

export default reducer
```

With our `currentTime` out of the root reducer, we'll need to update the `reducers.js` file to accept the new file into the root reducer. Luckily, this is pretty easy:

```
import { combineReducers } from 'redux';

import * as currentUser from './currentUser';
import * as currentTime from './currentTime';

export const rootReducer = combineReducers({
  currentTime: currentTime.reducer,
  currentUser: currentUser.reducer,
})

export const initialState = {
  currentTime: currentTime.initialState,
  currentUser: currentUser.initialState,
}

export default rootReducer
```

Lastly, let's update the `configureStore` function to pull the rootReducer and initial state from the file:

```
import { rootReducer, initialState } from './reducers'
// ...
export const configureStore = () => {
  const store = createStore(
    rootReducer,
    initialState,
  );

  return store;
}
```

Back to middleware

Middleware is basically a function that accepts the `store`, which is expected to return a function that accepts the `next` function, which is expected to return a function which accepts an action. Confusing? Let's look at what this means.

The simplest middleware possible

Let's build the smallest middleware we possibly can to understand exactly what's happening and how to add it to our stack.

Let's create our first middleware.

Now the signature of middleware looks like this:

```
// src/redux/loggingMiddleware.js
const loggingMiddleware = (store) => (next) => (action) => {
  // Our middleware
}
export default loggingMiddleware;
```

Befuddled about this middleware thing? Don't worry, we all are the first time we see it. Let's peel it back a little bit and destructure what's going on. That `loggingMiddleware` description above could be rewritten like the following:

```
const loggingMiddleware = function(store) {
  // Called when calling applyMiddleware so
  // our middleware can have access to the store

  return function(next) {
    // next is the following action to be run
    // after this middleware

    return function(action) {
      // finally, this is where our logic lives for
      // our middleware.
    }
  }
}
```

We don't need to worry about *how* this gets called, just that it does get called in that order. Let's enhance our `loggingMiddleware` so that we do actually log out the action that gets called:

```
const loggingMiddleware = (store) => (next) => (action) => {
  // Our middleware
  console.log(`Redux Log:`, action)
  // call the next function
  next(action);
}
```

Our middleware causes our store to, when every time an action is called, we'll get a `console.log` with the details of the action.

In order to apply middleware to our stack, we'll use this aptly named `applyMiddleware` function as the third argument to the `createStore()` method.

```
import { createStore, applyMiddleware } from 'redux';
```

To *apply* middleware, we can call this `applyMiddleware()` function in the `createStore()` method. In our `src/redux/configureStore.js` file, let's update the store creation by adding a call to `applyMiddleware()`:

```
// ...
import loggingMiddleware from "./loggingMiddleware";
// ...
const store = createStore(
  rootReducer,
  initialState,
  applyMiddleware(
    loggingMiddleware,
  )
);
```

Now our middleware is in place. Open up the console in your browser to see all the actions that are being called for this demo. Try clicking on the `Update` button with the console open...

As we've seen, middleware gives us the ability to insert a function in our Redux action call chain. Inside that function, we have access to the action, state, and we can dispatch other actions.

We want to write a middleware function that can handle API requests. We can write a middleware function that listens only to actions corresponding to API requests. Our middleware can "watch" for actions that have a special marker. For instance, we can have a `meta` object on the action with a `type` of `'api'`. We can use this to ensure our middleware does not handle any actions that are not related to API requests:

```
// src/redux/apiMiddleware.js
const apiMiddleware = store => next => action => {
  if (!action.meta || action.meta.type !== 'api') {
    return next(action);
  }

  // This is an api request
}
export default apiMiddleware
```

If an action does have a meta object with a type of `'api'`, we'll pick up the request in the `apiMiddleware`.

Let's convert our `fetchNewTime()` actionCreator to include these properties into an API request. Let's open up the `actionCreators` redux module we've been working with (in `src/redux/actionCreators.js`) and find the `fetchNewTime()` function definition.

Let's pass in the URL to our `meta` object for this request. We can even accept parameters from inside the call to the action creator:

```
const host = 'https://andthetimeis.com'
export const fetchNewTime = (timezone = 'pst', str='now') => ({
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString(),
  meta: {
    type: 'api',
    url: host + '/' + timezone + '/' + str + '.json'
  }
})
```

When we press the button to update the time, our `apiMiddleware` will catch this before it ends up in the reducer. For any calls that we catch in the middleware, we can pick apart the meta object and make requests using these options. Alternatively, we can just pass the entire sanitized `meta` object through the `fetch()` API as-is.

The steps our API middleware will have to take:

1. Find the request URL and compose request options from meta
2. Make the request
3. Convert the request to a JavaScript object
4. Respond back to Redux/user

Let's take this step-by-step. First, to pull off the `URL` and create the `fetchOptions` to pass to `fetch()`. We'll put these steps in the comments in the code below:

```
const apiMiddleware = store => next => action => {
  if (!action.meta || action.meta.type !== 'api') {
    return next(action);
  }
  // This is an api request

  // Find the request URL and compose request options from meta
  const {url} = action.meta;
  const fetchOptions = Object.assign({}, action.meta);

  // Make the request
  fetch(url, fetchOptions)
    // convert the response to json
    .then(resp => resp.json())
    .then(json => {
      // respond back to the user
      // by dispatching the original action without
      // the meta object
      let newAction = Object.assign({}, action, {
        payload: json.dateString
      });
      delete newAction.meta;
      store.dispatch(newAction);
    })
}

export default apiMiddleware
```

We have several options for how we respond back to the user in the Redux chain. Personally, we prefer to respond with the same type the request was fired off without the `meta` tag and placing the response body as the `payload` of the new action.

In this way, we don't have to change our redux reducer to manage the response any differently than if we weren't making a request.

We're also not limited to a single response either. Let's say that our user passed in an `onSuccess` callback to be called when the request was complete. We could call that `onSuccess` callback and then dispatch back up the chain:

```
const apiMiddleware = store => next => action => {
  if (!action.meta || action.meta.type !== 'api') {
    return next(action);
  }
  // This is an api request

  // Find the request URL and compose request options from meta
  const {url} = action.meta;
  const fetchOptions = Object.assign({}, action.meta);

  // Make the request
  fetch(url, fetchOptions)
    // convert the response to json
    .then(resp => resp.json())
    .then(json => {
      if (typeof action.meta.onSuccess === 'function') {
        action.meta.onSuccess(json);
      }
      return json; // For the next promise in the chain
    })
    .then(json => {
      // respond back to the user
      // by dispatching the original action without
      // the meta object
      let newAction = Object.assign({}, action, {
        payload: json.dateString
      });
      delete newAction.meta;
      store.dispatch(newAction);
    })
}
```

The possibilities here are virtually endless. Let's add the `apiMiddleware` to our chain by updating it in the `configureStore()` function:

```
import { createStore, applyMiddleware } from 'redux';
import { rootReducer, initialState } from './reducers'

import loggingMiddleware from './loggingMiddleware';
import apiMiddleware from './apiMiddleware';

export const configureStore = () => {
  const store = createStore(
    rootReducer,
    initialState,
    applyMiddleware(
      apiMiddleware,
      loggingMiddleware,
    )
  );

  return store;
}

export default configureStore;
```

Welcome home!

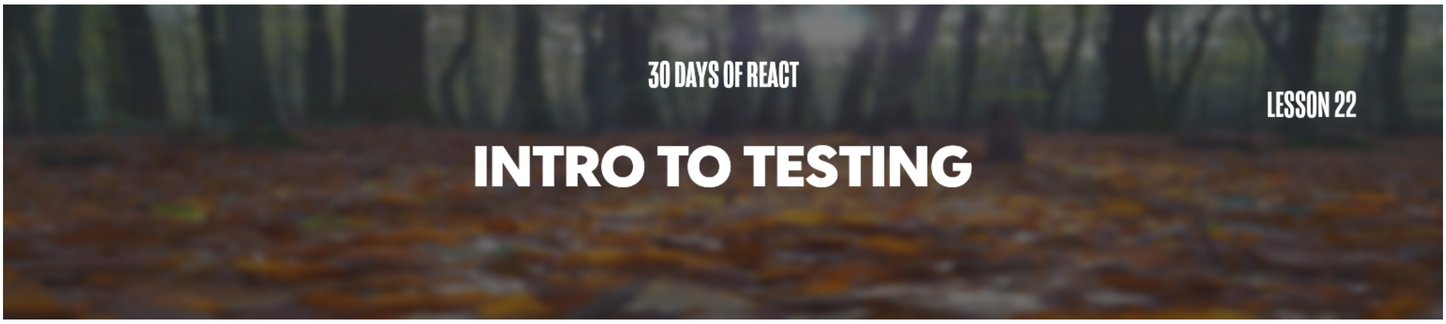
Current time: Thu Feb 27 2020 16:01:49 GMT-0600 (CST)

Update time

Notice that we didn't have to change *any* of our view code to update how the data was populated in the state tree. Pretty nifty, eh?

This middleware is pretty simplistic, but it's a good solid basis for building it out. Can you think of how you might implement a caching service, so that we don't need to make a request for data we already have? How about one to keep track of pending requests, so we can show a spinner for requests that are outstanding?

Awesome! Now we really are Redux ninjas. We've conquered the Redux mountain and are ready to move on to the next step. Before we head there, however... pat yourself on the back. We've made it through week 3!



Introduction to Testing

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-22/post.md)

Test suites are an upfront investment that pay dividends over the lifetime of a system. Today we'll introduce the topic of testing and discuss the different types of tests we can write.

Okay, close your eyes for a second... wait, don't... it's hard to read with your eyes closed, but imagine for a moment your application is getting close to your first deployment.

It's getting close and it gets tiring to constantly run through the features in your browser... and so inefficient.

There must be a better way...

Testing

When we talk about testing, we're talking about the process of automating the process of setting up and measuring our assumptions against assertions of functionality about our application.

When we talk about front-end testing in React, we're referring to the process of making assertions about what our React app renders and how it responds to user interaction.

We'll discuss three different software testing paradigms: unit testing, functional testing, and integration testing.

Unit tests

Unit testing refers to testing individual pieces (or units, hence the name) of our our code so we can be confident these specific pieces of code work as we expect.

For example, we have a few reducers already in our application. These reducers comprise a single function that we can make assertions on under different scenarios.

In React, Unit tests typically do not require a browser, can run incredibly quickly (no writing to the DOM required), and the assertions themselves are usually simple and terse.

We'll mostly concentrate on answering the question: with a given set of inputs (state and props), does the output match our expectations of what *should* be in the virtual dom. In this case, we're testing the rendering output.

Functional testing

With functional testing, we're focused on testing the behavior of our component. For instance, if we have a navigation bar with a user login/logout button, we can test our expectations that:

- Given a logged in user, the navbar renders a button with the text *Logout*
- Given no logged in user, the navbar renders a button with the text *Login*

Functional tests usually run in isolation (i.e. testing the component functionality without the rest of the application).

Integration testing

Finally, the last type of testing we'll look at is integration testing. This type of testing tests the entire service of our application and attempts to replicate the experience an end-user would experience when using our application.

On the order of speed and efficiency, integration testing is incredibly slow as it needs to run expectations against a live, running browser, where as unit and functional tests can run quite a bit faster (especially in React where the functional test is testing against the in-memory virtual dom rather than an actual browser render).

When testing React components, we will test both our expectations of what is contained in the virtual dom as well as what is reflected in the actual dom.

The tools

We're going to use a testing library called called jasmine (http://jasmine.github.io) to provide a readable testing language and assertions.

As far as test running, there is a general debate around which test runner is the easiest/most efficient to work with, largely between mocha (https://mochajs.org) and jest (https://facebook.github.io/jest).

We're going to use Jest in our adventure in testing with React as it's the *official* (take this with a grain of salt) test runner. Most of the code we'll be writing will be in Jasmine, so feel free to use mocha, if it's your test library of choice.

Finally, we'll use a library we cannot live without called Enzyme (https://github.com/airbnb/enzyme) which puts the fun back in FUNctional testing. Enzyme provides some pretty nice React testing utility functions that make writing our assertions a cinch.

Tomorrow, we'll get our application set up with the testing tooling in place so that we can start testing our application and be confident it works as we expect. See you tomorrow!

Implementing Tests

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-23/post.md)

Yesterday we examined the different types of tests that we write in React. Today we'll see it in action. We'll install the dependencies required to set up tests as well as write our first assertions.

Let's get our application set up to be tested. Since we're going to be using a few different libraries, we'll need to install them before we can use them (obviously).

Dependencies

We're going to use the following `npm` libraries:

jest/jest-cli

Jest (https://facebook.github.io/jest/) is the official testing framework released by Facebook and is a fantastic testing framework for testing React applications. It is incredibly fast, provides sandboxed testing environments, support for snapshot testing, and more.

babel-jest/babel-preset-stage-0

We'll write our tests using the stage 0 (or ES6-edge functionality), so we'll want to make sure our test framework can read and process our ES6 in our tests and source files.

sinon

Sinon is a test utility library which provides a way for us to write spies, stubs, and mocks. We'll discuss what these are when we need them, but we'll install the library for now.

react-addons-test-utils/enzyme

The `react-addons-test-utils` package contains testing utilities provided by the React team.

Enzyme (http://airbnb.io/enzyme/), a JavaScript testing library built/maintained by Airbnb is a bit easier to work with and provides really nice methods for traversing/manipulating React's virtual DOM output. While we'll start with `react-addons-test-utils`, we'll transition to using Enzyme as we prefer using it in our tests.

react-test-renderer

The `react-test-renderer` library allows us to use the snapshot feature from the jest library. Snapshots are a way for Jest to serialize the rendered output from the virtual DOM into a file which we can automate comparisons from one test to the next.

redux-mock-store

The redux-mock-store (https://github.com/arnaudbenard/redux-mock-store) library allows us to easily make a redux store for testing. We'll use it to test our action creators, middleware, and our reducers.

To install all of these libraries, we'll use the following `npm` command in the terminal while in the root directory of our projects:

```
yarn add --dev babel-jest babel-preset-stage-0 enzyme enzyme-adapter-react-16 jest-cli react-addons-test-utils react-test-renderer redux-mock-store sinon
```

Configuration

We'll also need to configure our setup. First, let's add an npm script that will allow us to run our tests using the `npm test` command. In our `package.json` file in the root of our project, let's add the `test` script. Find the scripts key in the `package.json` file and add the `test` command, like so:

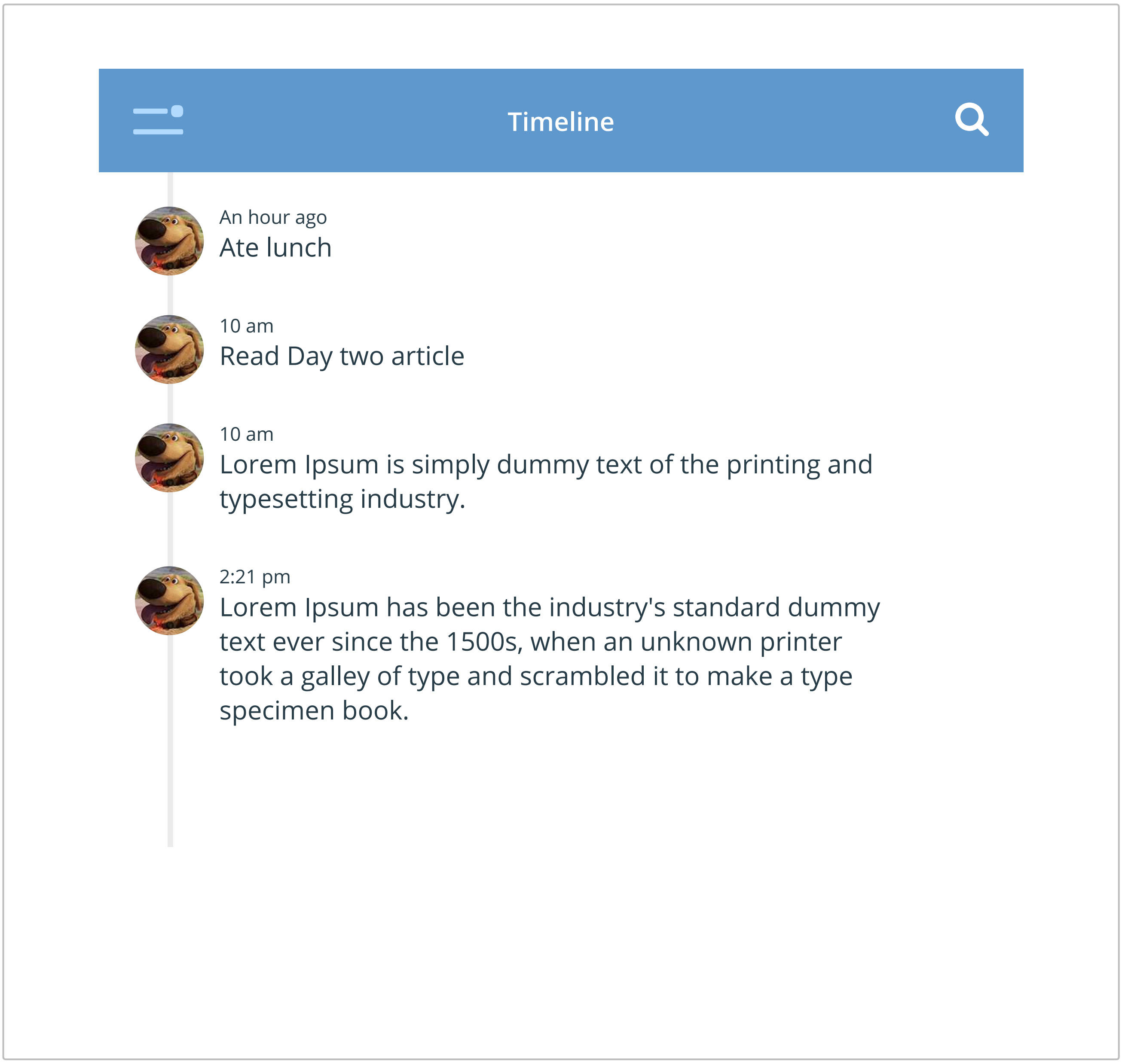
```
{
  // ...
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "eject": "react-scripts eject",
    "test": "react-scripts test --env=jsdom"
  },
}
```

Writing tests

Let's confirm that our test setup is working properly. Jest will automatically look for test files in the entire tree in a directory called `__tests__` (yes, with the underscores). Let's create our first `__tests__` directory in our `src/components/Timeline` directory and create our first test file:

```
mkdir src/components/Timeline/__tests__
touch src/components/Timeline/__tests__/Timeline-test.js
```

The `Timeline-test.js` file will include all the tests for our `Timeline` component (as indicated by the filename). Let's create our first test for the Timeline component.



We'll write our tests using the Jasmine (http://jasmine.github.io) framework. Jasmine provides a few methods we'll use quite a bit. Both of the following methods accept two arguments, the first being a description string and the second a function to execute:

- `describe()`
- `it()`

The `describe()` function provides a way for us to group our tests together in logical bundles. Since we're writing a bunch of tests for our `Timeline`, we'll use the `describe()` function in our test to indicate we're testing the Timeline.

In the `src/components/Timeline/__tests__/Timeline-test.js` file, let's add the describe block:

```
describe("Timeline", () => {});
```

We can add our first test using the `it()` function. The `it()` function is where we will set our expectations. Let's set up our tests with our first expectations, one passing and one failing so we can see the difference in output.

In the same file, let's add two tests:

```
describe("Timeline", () => {
  it("passing test", () => {
    expect(true).toBeTruthy();
  });

  it("failing test", () => {
    expect(false).toBeTruthy();
  });
});
```

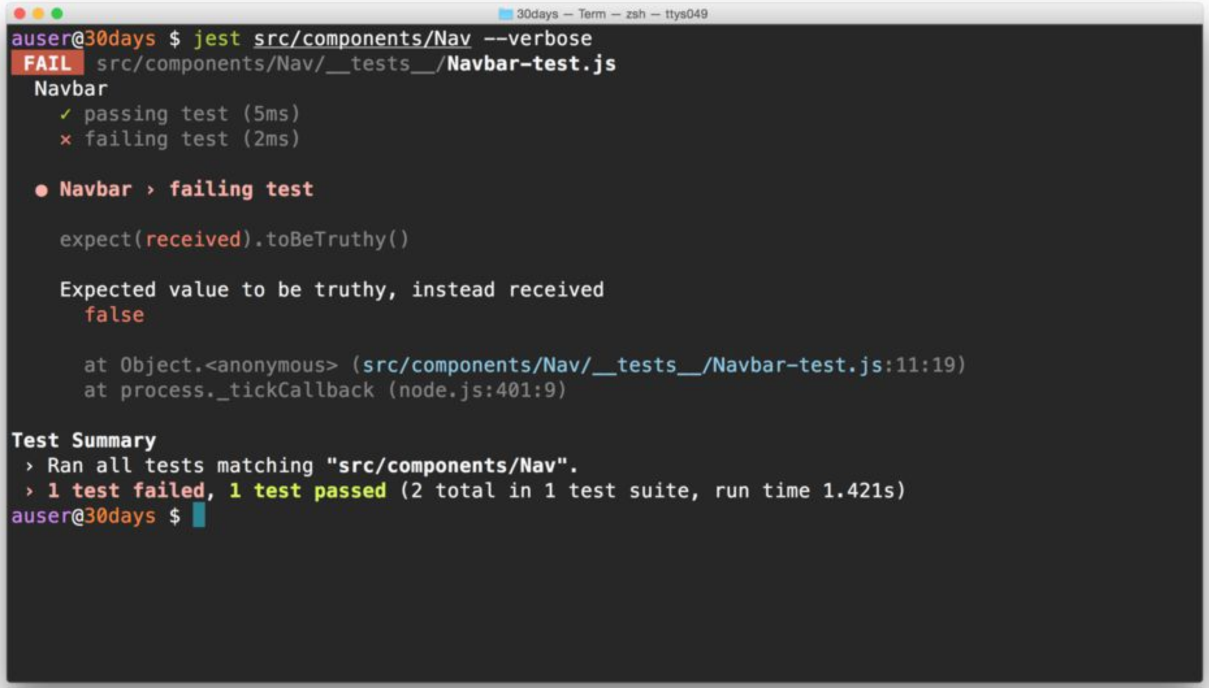
We'll look at the possible expectations we can set in a moment. First, let's run our tests.

Executing tests

The `create-react-app` package sets up a quality testing environment using Jest automatically for us. We can execute our tests by using the `yarn test` or `npm test` script.

In the terminal, let's execute our tests:

```
yarn test
```



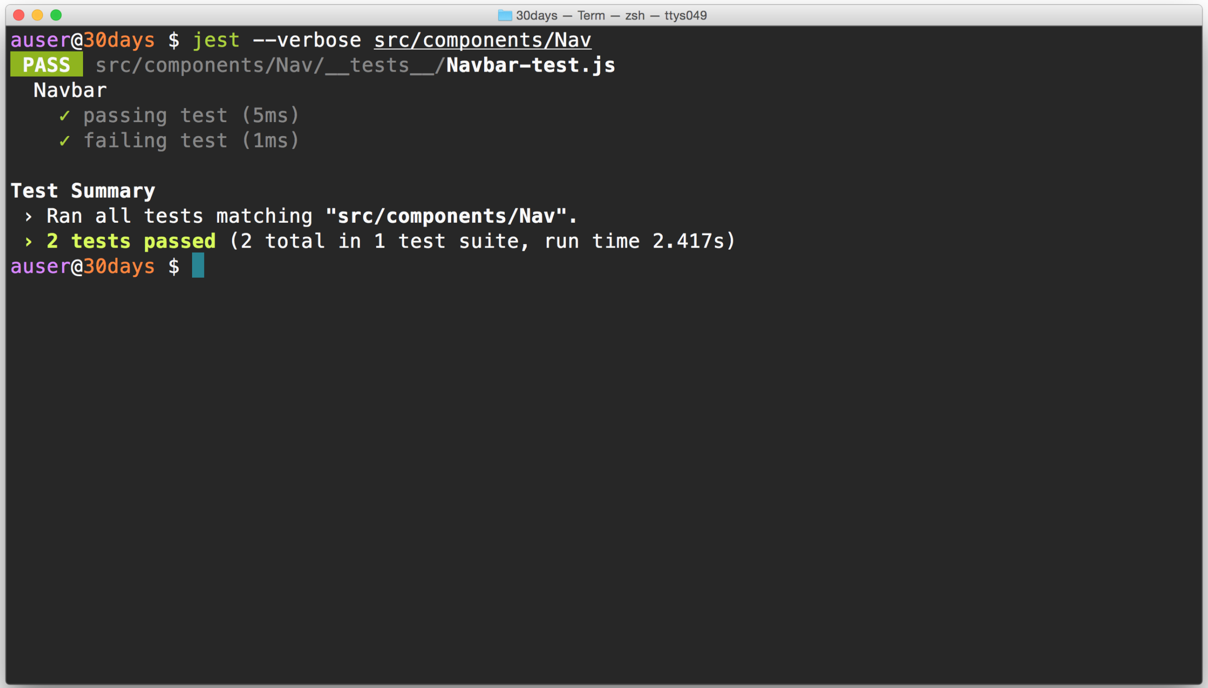
From this output, we can see the two tests with one passing test (with a green checkmark) and one failing test (with the red x and a description of the failure).

Let's update the second test to make it pass by changing the expectation to `toBeFalsy()`:

```
describe("Timeline", () => {
  it("passing test", () => {
    expect(true).toBeTruthy();
  });

  it("failing test", () => {
    expect(false).toBeFalsy();
  });
});
```

Re-running the test, we can see we have two passing tests

```
yarn test
```



Expectations

Jest provides a few global commands in our tests by default (i.e. things you don't need to require). One of those is the `expect()` command. The `expect()` command has a few expectations which we can call on it, including the two we've used already:

- `toBeTruthy()`
- `toBeFalsy()`
- `toBe()`
- `toEqual()`
- `toBeDefined()`
- `toBeCalled()`
- etc.

The entire suite of expectations is available on the jest documentation page at: https://facebook.github.io/jest/docs/api.html#writing-assertions-with-expect (https://facebook.github.io/jest/docs/api.html#writing-assertions-with-expect).

The `expect()` function takes a single argument: the value or function that returns a value to be tested. For instance, our two tests we've already writen pass the boolean values of `true` and `false`.

Now that we've written our first tests and confirmed our setup, we'll actually get down to testing our Timeline component tomorrow. Great job today and see you tomorrow!

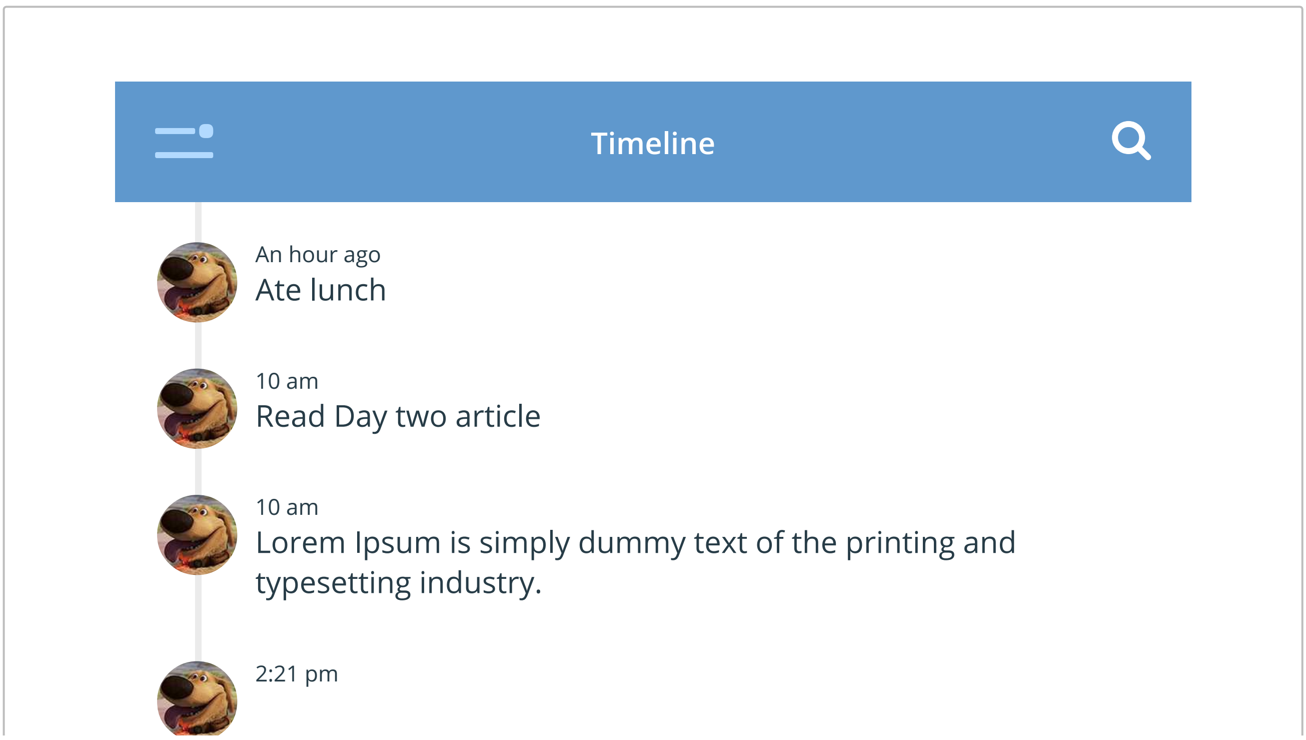
Testing the App

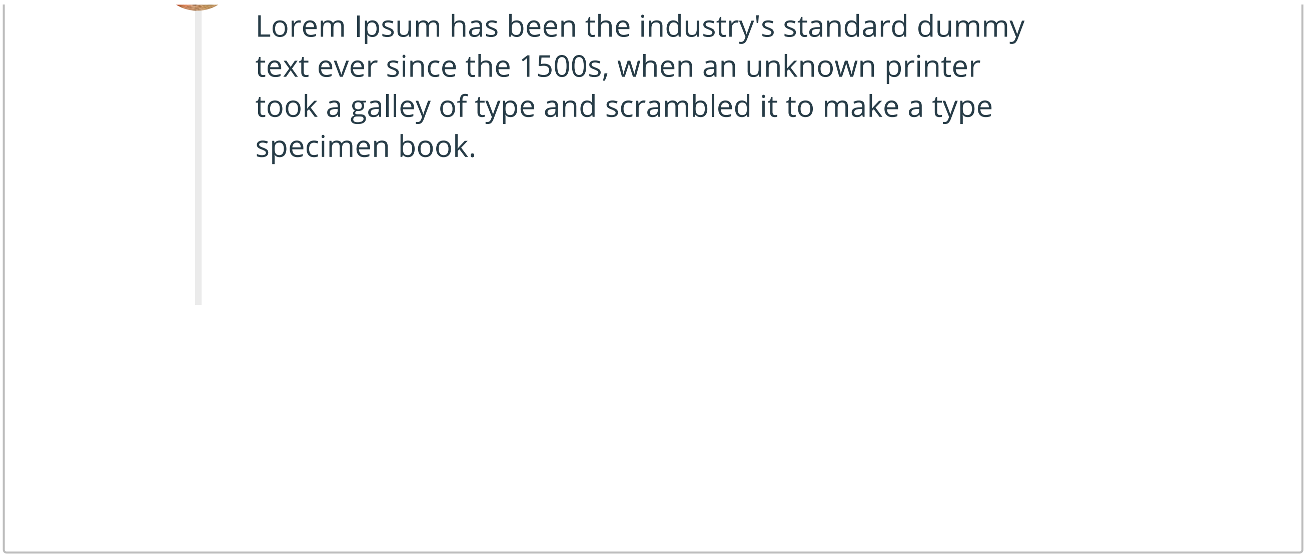
Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-24/post.md)

Let's start by looking at one feature of our application and thinking about where the edge cases are and what we assume will happen with the component.

Let's start with the `Timeline` component as it's the most complex in our current app.

The `Timeline` component dispays a list of statuses with a header with a dynamic title. We'll want to test any dynamic logic we have in our components. The simplest bit of logic we have to start out with our tests are around the dynamic title presented on the timeline.





We like to start out testing by listing our assumptions about a component and under what circumstances these assumptions are true. For instance, a list of assumptions we can make about our Timeline component might include the following:

- Under all circumstances, the Timeline will be contained within a `<div />` with the class of `.notificationsFrame`
- Under all circumstances, we can assume there will be a title
- Under all circumstances, we assume the search button will start out as hidden
- There is a list of at least four status updates

These *assumptions* will translate into our tests.

Testing

Let's open the file `src/components/Timeline/__tests__/Timeline-test.js`. We left off with some dummy tests in this file, so let's clear those off and start with a fresh describe block:

```
describe("Timeline", () => {
  // Tests go here
});
```

For every test that we write against React, we'll want to import react into our test file. We'll also want to bring in the react test utilities:

```
import React from "react";
import TestUtils from "react-dom/test-utils";

describe("Timeline", () => {
  // Tests go here
});
```

Since we're testing the `Timeline` component here, we'll also want to bring that into our workspace:

```
import React from "react";
import TestUtils from "react-dom/test-utils";

import Timeline from "../Timeline";

describe("Timeline", () => {
  // Tests go here
});
```

Let's write our first test. Our first assumption is pretty simple to test. We're testing to make sure the element is wrapped in a `.notificationsFrame` class. With every test we'll write, we'll need to render our application into the working test document. The `react-dom/test-utils` library provides a function to do just this called `renderIntoDocument()`:

```
import React from "react";
import TestUtils from "react-dom/test-utils";

import Timeline from "../Timeline";

describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    const wrapper = TestUtils.renderIntoDocument(<Timeline />);
  });
});
```

If we run this test (even though we're not setting any expectations yet), we'll see that we have a problem with the testing code. React thinks we're trying to render an undefined component:

Let's find the element we expect to be in the DOM using another `TestUtils` function called `findRenderedDOMComponentWithClass()`.

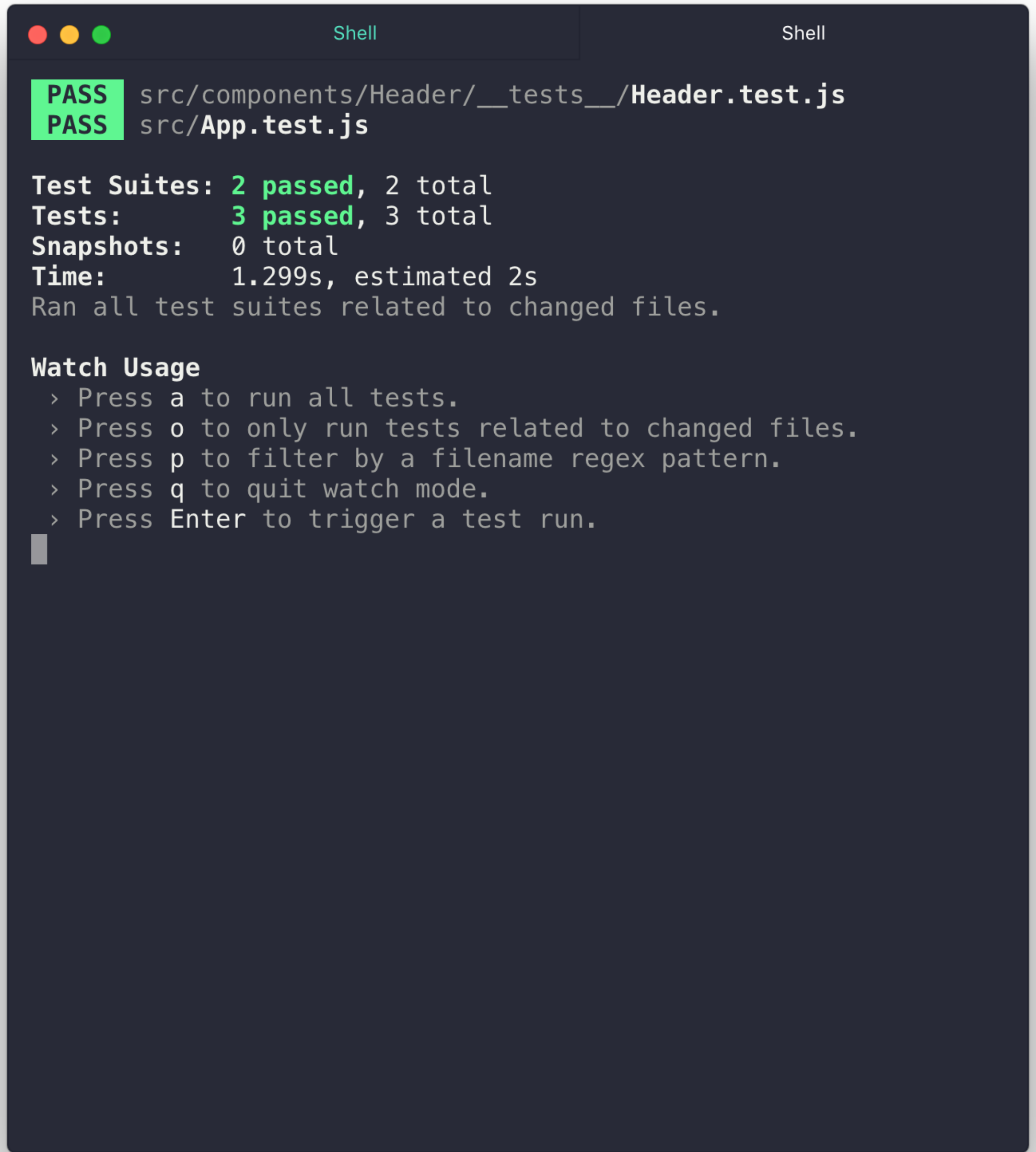
The `findRenderedDOMComponentWithClass()` function accepts *two* arguments. The first is the render tree (our `wrapper` object) and the second is the CSS class name we want it to look for:

```
import React from "react";
import TestUtils from "react-dom/test-utils";

import Timeline from "../Timeline";

describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    const wrapper = TestUtils.renderIntoDocument(<Timeline />);
    const node = TestUtils.findRenderedDOMComponentWithClass(
      wrapper,
      "notificationsFrame"
    );
  });
});
```

With that, our tests will pass (believe it or not). The TestUtils sets up an expectation that it can find the component with the `.notificationsFrame` class. If it doesn't find one, it will throw an error and our tests will fail.

As a reminder, we can run our tests using either the `npm test` command or the `yarn test` command. We'll use the `yarn test` command for now since we're testing one component:

```
yarn test
```



With our one passing test, we've confirmed our test setup is working.

Unfortunately, the interface for `TestUtils` is a little complex and low-level. The `enzyme` library wraps `TestUtils`, providing an easier and higher-level interface for asserting against a React component under test. We'll discuss enzyme in detail tomorrow.

Great job today and see you tomorrow!

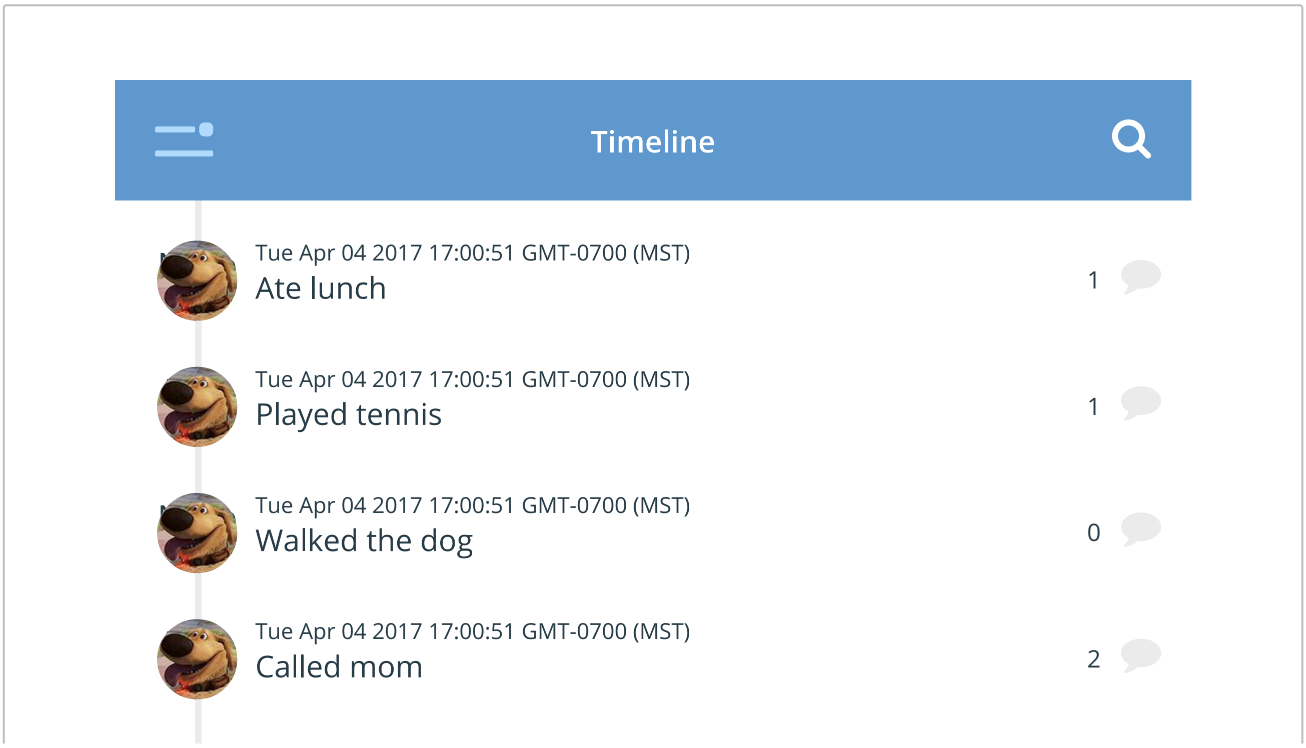
Better Testing with Enzyme

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-25/post.md)

Today, we'll look at an open-source library maintained by Airbnb called Enzyme that makes testing fun and easy.

Yesterday we used the `react-dom/test-utils` library to write our first test against the `Timeline` component. However, this library is fairly low-level and can be a bit cumbersome to use. Enzyme (http://airbnb.io/enzyme/) is a testing utility library released and maintained by the AirBnb (http://airbnb.io) team and it offers a nicer, higher-level API for dealing with React components under test.

We're testing against our `<Timeline />` component:



Using Enzyme

We'll use Enzyme to make these tests easier to write and more readable.

Yesterday, we wrote our first test as the following:

```
import React from "react";
import TestUtils from "react-dom/test-utils";

import Timeline from "../Timeline";

describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    const wrapper = TestUtils.renderIntoDocument(<Timeline />);
    TestUtils.findRenderedDOMComponentWithClass(wrapper,
"notificationsFrame");
  });
});
```

Although this works, it's not quite the easiest test in the world to read. Let's see what this test looks like when we rewrite it with Enzyme.

Rather than testing the complete component tree with Enzyme, we can test just the output of the component. Any of the component's children will not be rendered. This is called *shallow* rendering.

Enzyme makes shallow rendering super easy. We'll use the `shallow` function exported by Enzyme to mount our component.

Let's first configure `enzyme` use the adapter that makes it compatible with React version 16. Create `src/setupTests.js` and add the following:

```
import { configure } from "enzyme";
import Adapter from "enzyme-adapter-react-16";

configure({ adapter: new Adapter() });
```

Let's update the `src/components/Timeline/__tests__/Timeline-test.js` file to include the `shallow` function from `enzyme`:

```
import React from "react";
import { shallow } from "enzyme";

describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    // our tests
  });
});
```

> Shallow rendering is supported by `react-dom/test-utils` as well. In fact, Enzyme just wraps this functionality. While we didn't use shallow rendering yesterday, if we were to use it would look like this:
>
> ```
> const renderer = ReactTestUtils.createRenderer();
> renderer.render(<Timeline />);
> const result = renderer.getRenderOutput();
> ```

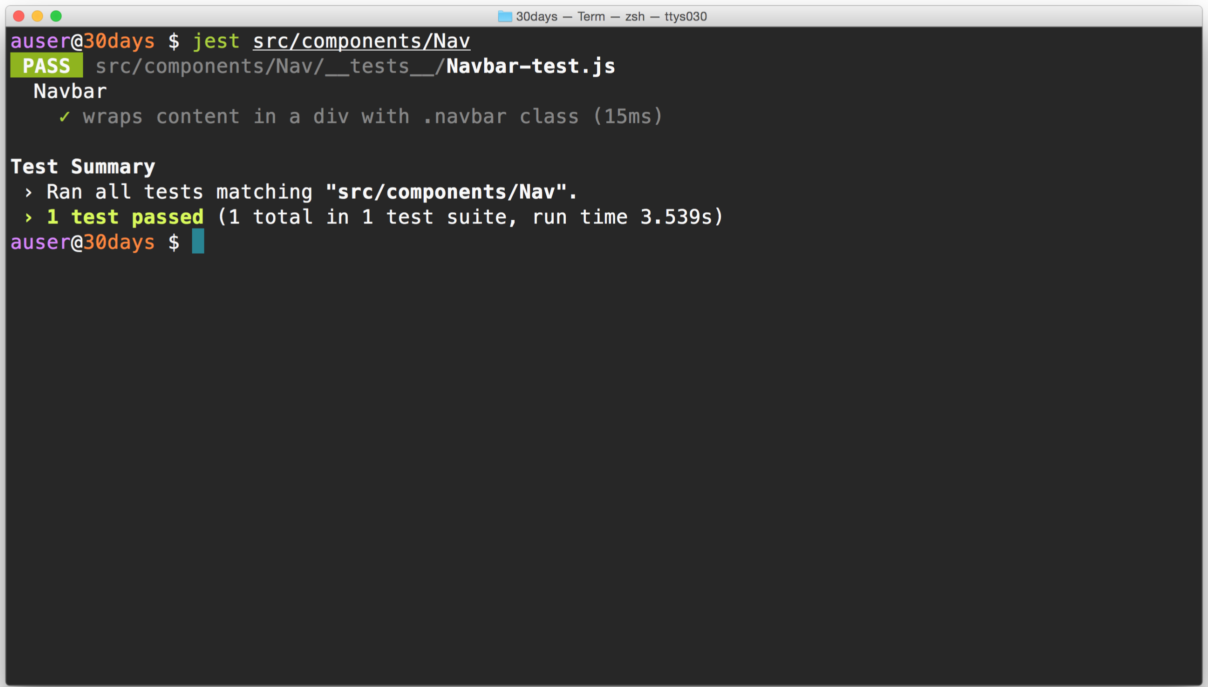
Now to render our component, we can use the `shallow` method and store the result in a variable. Then, we'll *query* the rendered component for different React elements (HTML or child components) that are rendered inside its virtual dom.

The entire assertion comprises two lines:

```
import React from "react";
import { shallow, mount } from "enzyme";

import Timeline from "../Timeline";

describe("Timeline", () => {
  let wrapper;

  it("wraps content in a div with .notificationsFrame class", () => {
    wrapper = shallow(<Timeline />);
    expect(wrapper.find(".notificationsFrame").length).toEqual(1);
  });
});
```

We can run our tests in the same manner as we did before using the `yarn test` command (or the `npm test` command):

```
yarn test
```



Our test passes and is more readable and maintainable.

Let's continue writing assertions, pulling from the list of assumptions that we made at the beginning of yesterday. We'll structure the rest of our test suite first by writing out our `describe` and `it` blocks. We'll fill out the specs with assertions after:

```
import React from "react";
import { shallow } from "enzyme";

import Timeline from "../Timeline";

describe("Timeline", () => {
  let wrapper;

  it("wraps content in a div with .notificationsFrame class", () => {
    wrapper = shallow(<Timeline />);
    expect(wrapper.find(".notificationsFrame").length).toEqual(1);
  });

  it("has a title of Timeline");

  describe("search button", () => {
    it("starts out hidden");
    it("becomes visible after being clicked on");
  });

  describe("status updates", () => {
    it("has 4 status updates at minimum");
  });
});
```

> If we were following Test Driven Development (or TDD for short), we would write these assumptions first and then build the component to pass these tests.

Let's fill in these tests so that they pass against our existing `Timeline` component.

Our title test is relatively simple. We'll look for the title element and confirm the title is `Timeline`.

We expect the title to be available under a class of `.title`. So, to use the `.title` class in a spec, we can just grab the component using the `find` function exposed by Enzyme.

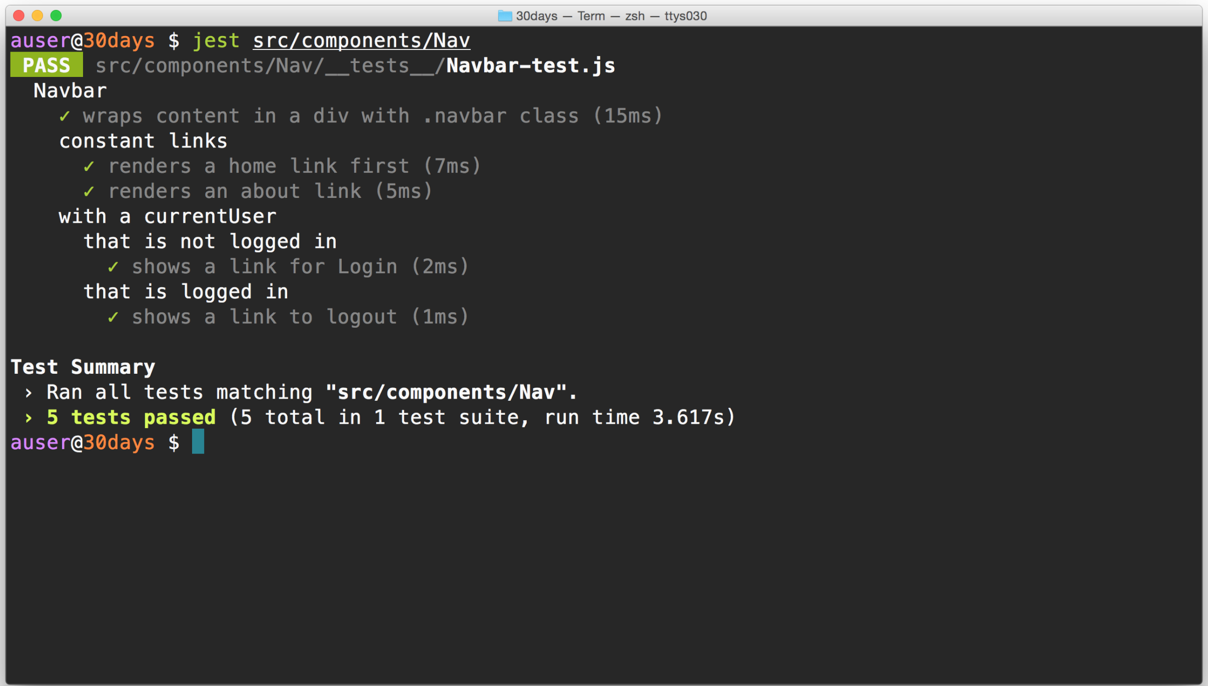
Since our `Header` component is a child component of our `Timeline` component, we can't use the `shallow()` method. Instead we have to use the `mount()` method provided by Enzyme.

> # Shallow? Mount?
>
> The `shallow()` rendering function only renders the component we're testing specifically and it won't render child elements. Instead we'll have to `mount()` the component as the child `Header` won't be available in the jsdom otherwise.
>
> We'll look at more Enzyme functions at the end of this article.

Let's fill out the title spec now:

```
import React from "react";
import { shallow, mount } from "enzyme";

import Timeline from "../Timeline";

describe("Timeline", () => {
  let wrapper;

  it("wraps content in a div with .notificationsFrame class", () => {
    wrapper = shallow(<Timeline />);
    expect(wrapper.find(".notificationsFrame").length).toEqual(1);
  });

  it("has a title of Timeline", () => {
    wrapper = mount(<Timeline />); // notice the `mount`
    expect(wrapper.find(".title").text()).toBe("Timeline");
  });
});
```

Running our tests, we'll see these two expectations pass:



Next, let's update our search button tests. We have two tests here, where one requires us to test an interaction. Enzyme provides a very clean interface for handling interactions. Let's see how we can write a test against the search icon.

Again, since we're testing against a child element in our Timeline, we'll have to `mount()` the element. Since we're going to write two tests in a nested `describe()` block, we can write a before helper to create the `mount()` anew for each test so they are pure.

In addition, we're going to use the `input.searchInput` element for both tests, so let's write the `.find()` for that element in the before helper too.

```
describe("Timeline", () => {
  let wrapper;
  // ...
  describe("search button", () => {
    beforeEach(() => (wrapper = mount(<Timeline />)));
    // ...
  });
});
```

To test if the search input is hidden, we'll just have to know if the `active` class is applied or not. Enzyme provides a way for us to detect if a component has a class or not using the `hasClass()` method. Let's fill out the first test to expect the search input doens't have the active class:

```
describe("Timeline", () => {
  let wrapper;
  // ...
  describe("search button", () => {
    beforeEach(() => (wrapper = mount(<Timeline />)));

    it("starts out hidden", () => {

expect(wrapper.find("input.searchInput").hasClass("active")).toBeFalsy
();
    });
    it("becomes visible after being clicked on");
    // ...
  });
});
```

The tricky part about the second test is that we need to click on the icon element. Before we look at how to do that, let's find it first. We can target it by it's `.searchIcon` class on the wrapper:

```
it("becomes visible after being clicked on", () => {
  const icon = wrapper.find(".searchIcon");
});
```

Now that we have the icon we want to simulate a click on the element. Recall that the `onClick()` method is really just a facade for browser events. That is, a click on an element is just an event getting bubbled through the component. Rather than controlling a mouse or calling `click` on the element, we'll simulate an event occurring on it. For us, this will be the `click` event.

We'll use the `simulate()` method on the `icon` to create this event:

```
it("becomes visible after being clicked on", () => {
  const icon = wrapper.find(".searchIcon");
  icon.simulate("click");
});
```

Now we can set an expectation that the `search` component has the `active` class.

```
it("becomes visible after being clicked on", () => {
  const icon = wrapper.find(".searchIcon");
  icon.simulate("click");

expect(wrapper.find("input.searchInput").hasClass("active")).toBeTruthy();
});
```

Our last expectation for the `Timeline` component is that we have at least four status updates. As we are laying these elements on the `Timeline` component, we can `shallow` render the component. In addition, since each of the elements are of a custom component, we can search for the list of specific components of type 'ActivityItem'.

```
describe("status updates", () => {
  it("has 4 status updates at minimum", () => {
    wrapper = shallow(<Timeline />);
    // ...
  });
});
```

Now we can test for the length of a list of `ActivityItem` components. We'll set our expectation that the list if at least of length 4.

```
describe("status updates", () => {
  it("has 4 status updates at minimum", () => {
    wrapper = shallow(<Timeline />);
    expect(wrapper.find("ActivityItem").length).toBeGreaterThan(3);
  });
});
```

The entire test suite that we have now is the following:

```
import React from "react";
import { shallow, mount } from "enzyme";

import Timeline from "../Timeline";

describe("Timeline", () => {
  let wrapper;

  it("wraps content in a div with .notificationsFrame class", () => {
    wrapper = shallow(<Timeline />);
    expect(wrapper.find(".notificationsFrame").length).toEqual(1);
  });

  it("has a title of Timeline", () => {
    wrapper = mount(<Timeline />);
    expect(wrapper.find(".title").text()).toBe("Timeline");
  });

  describe("search button", () => {
    beforeEach(() => (wrapper = mount(<Timeline />)));

    it("starts out hidden", () => {

expect(wrapper.find("input.searchInput").hasClass("active")).toBeFalsy
();
    });
    it("becomes visible after being clicked on", () => {
      const icon = wrapper.find(".searchIcon");
      icon.simulate("click");

expect(wrapper.find("input.searchInput").hasClass("active")).toBeTruth
y();
    });
  });

  describe("status updates", () => {
    it("has 4 status updates at minimum", () => {
      wrapper = shallow(<Timeline />);
      expect(wrapper.find("ActivityItem").length).toBeGreaterThan(3);
    });
  });
});
```

What's the deal with `find()`?

Before we close out for today, we should look at the interface of an Enzyme shallow-rendered component (in our tests, the `wrapper` object). The Enzyme documentation (http://airbnb.io/enzyme/docs/api/shallow.html) is fantastic, so we'll keep this short.

Basically, when we use the `find()` function, we'll pass it a selector and it will return a `ShallowWrapper` instance that wraps the found nodes. The `find()` function can take a string, function, or an object.

When we pass strings into the `find()` function, we can pass CSS selectors or the *displayName* of a component. For instance:

```
wrapper.find("div.link");
wrapper.find("Link");
```

We can also pass it the component constructor, for instance:

```
import { Link } from "react-router";
// ...
wrapper.find(Link);
```

Finally, we can also pass it an object property selector object, which selects elements by their key and values. For instance:

```
wrapper.find({ to: "/login" });
```

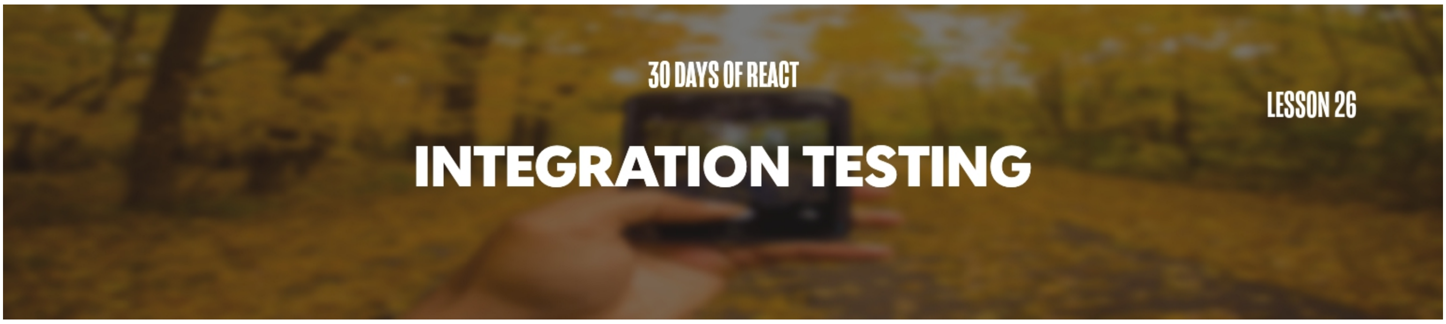
The return value is a `ShallowWrapper`, which is a type of `Wrapper` (we can have rendered wrappers and shallow wrappers). These `Wrapper` instances have a bunch of functions we can use to target different child components, ways to look into the `props` and the `state`, as well as other attributes of a rendered component, such as `html()` and `text()`. What's more, we can chain these calls together.

Take the case of the `<Link />` component. If we wanted to find the HTML of the link class based on all the links available, we can write a test like this:

```
// ...
it("displays a link tag with the Login text", () => {
  link = wrapper.find("Link").find({ to: "/login" });

  expect(link.html()).toBe('<a class="link">Login</a>');
});
```

Phew! That's a lot of new information today, but look how quickly we wrote our follow-up tests with Enzyme. It's much quicker to read and makes it easier to discern what's actually happening.

Tomorrow we'll continue with our testing journey and walk through integration testing our application.

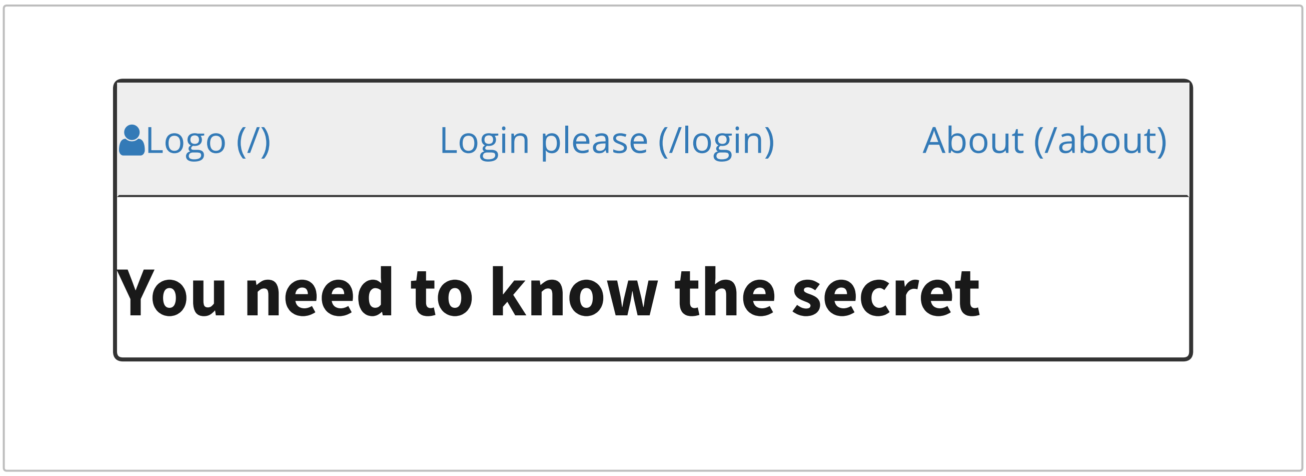


Integration Testing

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-26/post.md)

Today we'll write tests to simulate how users interact with our application and will test the entire flow of our app in a live browser.

We've reached the final part of our introduction to testing. We're going to wrap up our testing section with integration testing. As a reminder of what Integration testing is, it's the process of automating the experience that our actual users experience as they use our application.



Integration testing

As we're integration testing, we'll need to have our app actually running as we're going to have a browser launch and execute our application. We'll be using an automation server called selenium (http://www.seleniumhq.org), so we'll need to download it as well as a really nifty node automated testing framework called Nightwatch (http://nightwatchjs.org).

Install

The easiest way to install selenium (http://docs.seleniumhq.org/download/) is to download it through the the selenium website at: http://docs.seleniumhq.org/download/ (http://docs.seleniumhq.org/download/).

> If you're on a mac, you can use Homebrew (http://brew.sh) with the `brew` command:
>
> ```
> brew install selenium-server-standalone
> ```

We'll also need to install the `nightwatch` command, which we can do with the `npm` package manager. Let's install `nightwatch` globally using the `--global` flag:

```
npm install --global nightwatch
```

This command gives us the `nightwatch` command available anywhere in our terminal. We'll need to add a configuration file in the root directory called `nighwatch.json` (or `nighwatch.conf.js`). We'll use the default configuration file at `nighwatch.json`

Let's create the file in our root directory:

```
touch nightwatch.json
```

Now add the following content in the new `nightwatch.json`:

```
{
  "src_folders" : ["tests"],
  "output_folder" : "reports",

  "selenium" : {
    "start_process" : false,
    "server_path" : "",
    "log_path" : "",
    "host" : "127.0.0.1",
    "port" : 4444,
    "cli_args" : {
      "webdriver.chrome.driver" : "",
      "webdriver.ie.driver" : ""
    }
  },

  "test_settings" : {
    "default" : {
      "launch_url" : "http://localhost:3000",
      "selenium_port"  : 4444,
      "selenium_host"  : "localhost",
      "silent": true,
      "screenshots" : {
        "enabled" : false,
        "path" : ""
      },
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    },

    "chrome" : {
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    }
  }
}
```

Nightwatch gives us a lot of configuration options available, so we won't cover all the possible ways to configure it. For our purposes, we'll just use the base configuration above as it's more than enough for getting integration testing going.

Writing tests

We'll write our nightwatch tests in a `tests/` directory. Let's start by writing a test for handling the auth workflow. Let's write our test in a `tests/` directory (which matches the `src_folders`) that we'll call `tests/auth-flow.js`.

```
mkdir tests
touch tests/auth-flow.js
```

The nightwatch tests can be set as an object of exports, where the key is the description of the test and the value is a function with a reference to the client browser. For instance, we'll set up four tests for our `tests/auth-flow.js` test.

Updating our `tests/auth-flow.js` file with these four test functions look like the following:

```
module.exports = {
  "get to login page": browser => {},
  "logging in": browser => {},
  "logging out": browser => {},
  close: browser => {}
};
```

Each of the functions in our object exports will receive a `browser` instance which serves as the interface between our test and the selenium webdriver. We have all sorts of available options we can run on this `browser` variable.

Let's write our first test to demonstrate this function. We're going to set up nightwatch so that it launches the page, and clicks on the Login link in the navbar. We'll take the following steps to do this:

1. We'll first call the `url()` function on browser to ask it to load a URL on the page.
2. We'll wait for the page to load for a certain amount of time.
3. We'll find the Login link and click on it.

And we'll set up assertions along the way. Let's get busy! We'll ask the `browser` to load the URL we set in our configuration file (for us, it's `http://localhost:3000`)

```
module.exports = {
  "get to login page": browser => {
    browser
      // Load the page at the launch URL
      .url(browser.launchUrl)
      // wait for page to load
      .waitForElementVisible(".navbar", 1000)
      // click on the login link
      .click('a[href="/login"]');

    browser.assert.urlContains("login");
  },
  "logging in": browser => {},
  "logging out": browser => {},
  close: browser => {}
};
```

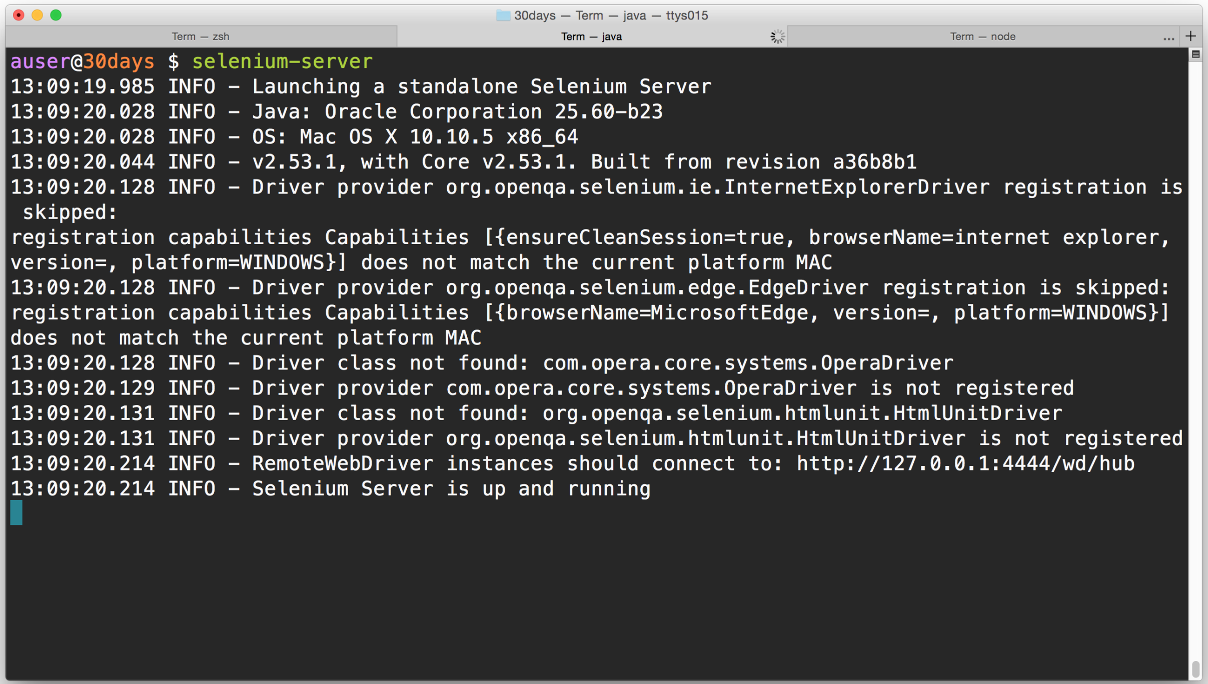
Thats it. Before we get too far ahead, let's run this test to make sure our test setup works. We'll need to open 3 terminal windows here.

In the first terminal window, let's launch selenium. If you downloaded the `.jar` file, you can start this with the command:

```
java -jar selenium-server-standalone-{VERSION}.jar
```

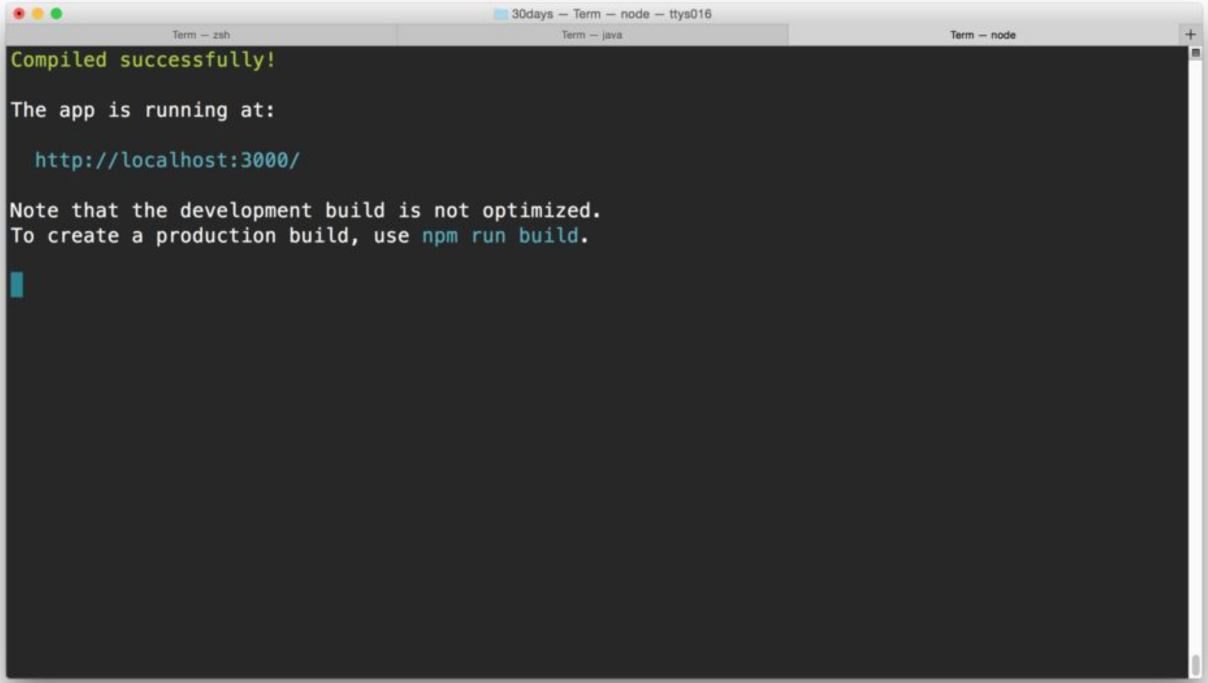
If you downloaded it through homebrew, you can use the `selenium-server` command:

```
selenium-server
```



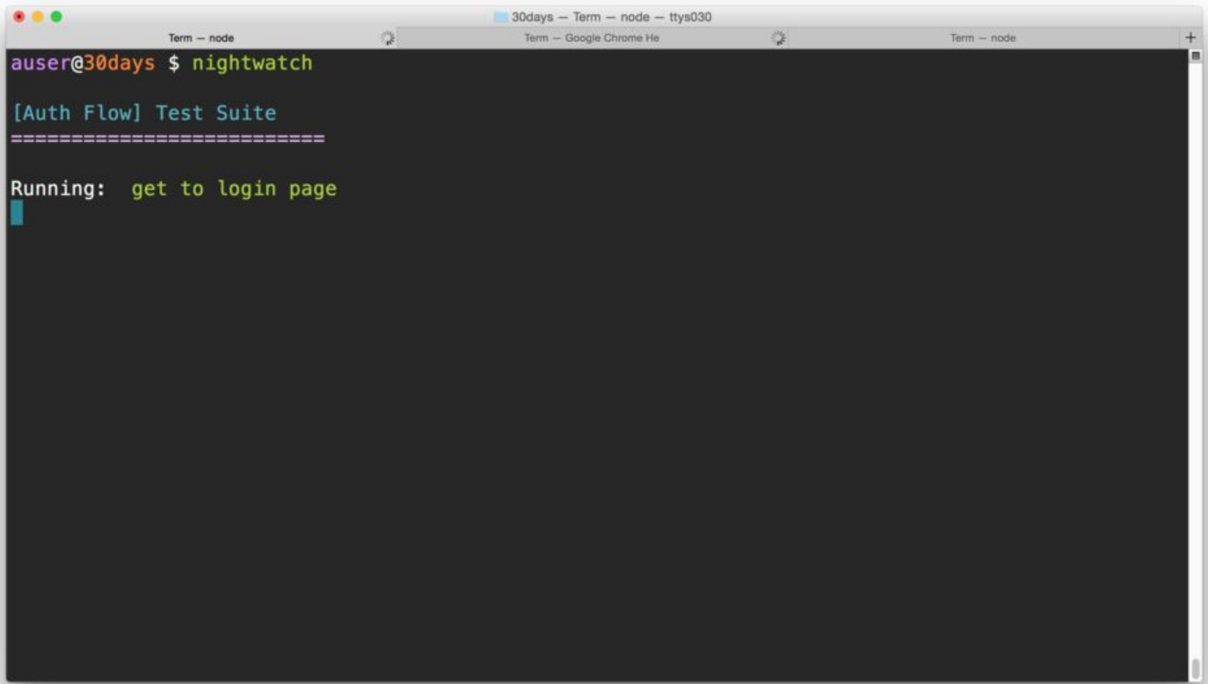
In the second window, we'll need to launch our app. Remember, the browser we're going to launch will *actually* hit our site, so we need an instance of it running. We can start our app up with the `npm start` comamnd:

```
npm start
```



Finally, in the third and final terminal window, we'll run our tests using the `nightwatch` command.

```
nightwatch
```



When we run the `nightwatch` command, we'll see a chrome window open up, visit the site, and click on the login link automatically... (pretty cool, right?).

All of our tests pass at this point. Let's actually tell the browser to log a user in.

Since the first step will run, the browser will already be on the login page. In the second key of our tests, we'll want to take the following steps:

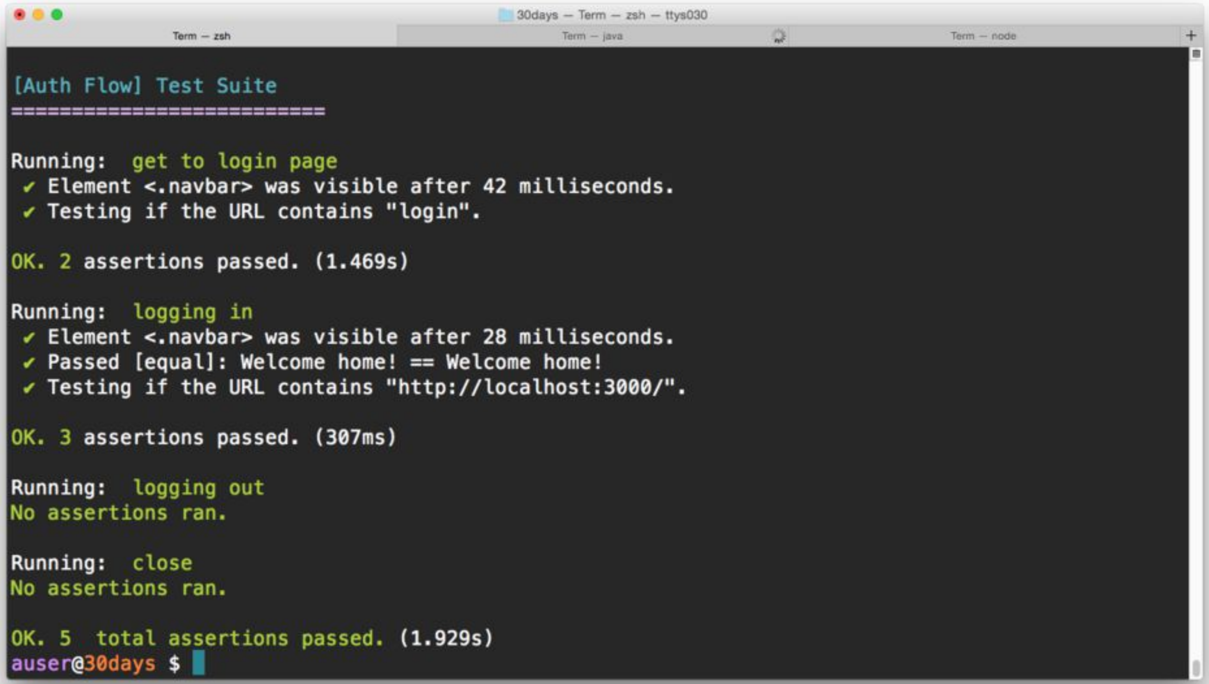
1. We'll want to `find the input for he user's email` and set a value to a valid email.
2. We'll want to `click` the submit/login button
3. We'll `wait` for the page to load (similar to how we did previously)
4. We'll want to `assert` that the text of the page is equal to what we expect it to be.
5. We'll set an assertion to make sure the URL is what we think it is.

Writing this up in code is straight-forward too. Just like we did previously, let's write the code with comments inline:

```
module.exports = {
  "get to login page": browser => {
    browser
      // Load the page at the launch URL
      .url(browser.launchUrl)
      // wait for page to load
      .waitForElementVisible(".navbar", 1000)
      // click on the login link
      .click('a[href="/login"]');

    browser.assert.urlContains("login");
  },
  "logging in": browser => {
    browser
      // set the input email to a valid username / password
      .setValue("input[type=text]", "admin")
      .setValue("input[type=password]", "secret")
      // submit the form
      .click("input[type=submit]")
      // wait for the page to load
      .waitForElementVisible(".navbar", 1000)
      // Get the text of the h1 tag
      .getText(".home h1", function(comp) {
        this.assert.equal(comp.value, "Welcome home!");
      });

    browser.assert.urlContains(browser.launchUrl);
  },
  "logging out": browser => {},
  close: browser => {}
};
```

Running these tests again (in the third terminal window):

```
nightwatch
```



We can do a similar thing with the `logging out` step from our browser. To get a user to log out, we will:

1. `Find and click` on the logout link
2. We'll want to `wait for the content to load again
3. We'll `assert` that t`he h1 tag contains the value we expect it to have
4. And we'll make sure the page shows the Login button

Let's implement this with comments inline:

```
module.exports = {
  "get to login page": browser => {
    browser
      // Load the page at the launch URL
      .url(browser.launchUrl)
      // wait for page to load
      .waitForElementVisible(".navbar", 1000)
      // click on the login link
      .click('a[href="/login"]');

    browser.assert.urlContains("login");
  },
  "logging in": browser => {
    browser
      // set the input email to a valid username / password
      .setValue("input[type=text]", "admin")
      .setValue("input[type=password]", "secret")
      // submit the form
      .click("input[type=submit]")
      // wait for the page to load
      .waitForElementVisible(".navbar", 1000)
      // Get the text of the h1 tag
      .getText(".home h1", function(comp) {
        this.assert.equal(comp.value, "Welcome home!");
      });

    browser.assert.urlContains(browser.launchUrl);
  },
  "logging out": browser => {
    browser
      // Find and click on the logout link
      .click(".logout")
      // We'll wait for the next content to load
      .waitForElementVisible("h1", 1000)
      // Get the text of the h1 tag
      .getText("h1", function(res) {
        this.assert.equal(res.value, "You need to know the secret");
      })
      // Make sure the Login button shows now
      .waitForElementVisible('a[href="/login"]', 1000);
  },
```

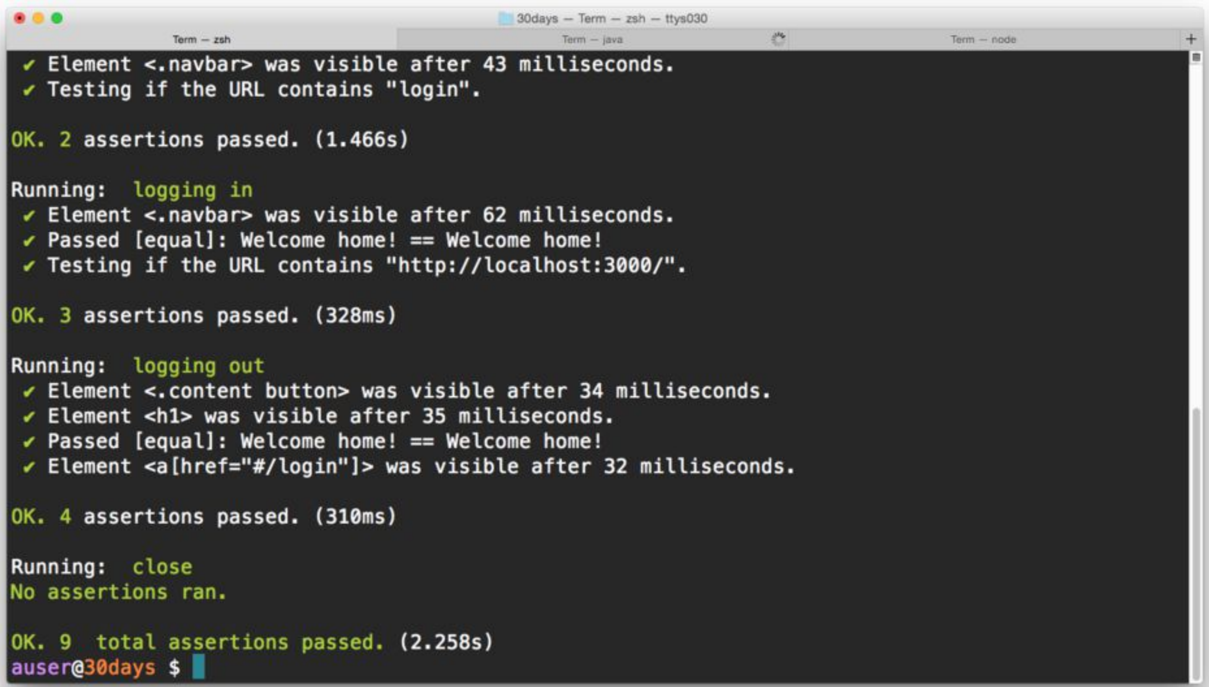
```
  close: browser => {}
};
```

As of now, you may have noticed that your chrome browsers haven't been closing when the tests have completed. This is because we haven't told selenium that we want the session to be complete. We can use the `end()` command on the `browser` object to close the connection. This is why we have the last and final step called `close`.

```
{
  // ...
  'close': (browser) => browser.end()
}
```

Now let's run the entire suite and make sure it passes again using the `nightwatch` command:

```
nightwatch
```



One final note, if you're interested in a deeper set of selenium tutorials, check out the free tutorials from guru99.com at https://www.guru99.com/selenium-tutorial.html (https://www.guru99.com/selenium-tutorial.html). They are pretty in-depth and well done (in our opinion).

That's it! We've made it and have covered 3 types of testing entirely, from low-level up through faking a real browser instance. Now we have the tools to ensure our applications are ready for full deployment.

But wait, we don't actually have deployment figured out yet, do we? Stay tuned for tomorrow when we start getting our application deployed into the cloud.

Deployment Introduction

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-27/post.md)

Today, we'll explore the different pieces involved in deploying our application so the world can use our application out in the wild.

With our app all tested up through this point, it's time to get it up and live for the world to see. The rest of this course will be dedicated to deploying our application into production.

Production deployment

When talking about deployment, we have a lot of different options:

- Hosting
- Deployment environment configuration
- Continuous Integration (CI, for short)
- Cost cycles, network bandwidth cost
- Bundle size
- and more

We'll look at the different hosting options we have for deploying our react app tomorrow and look at a few different methods we have for deploying our application up. Today we're going to focus on getting our app ready for deployment.

Ejection (from `create-react-app`)

First things first... we're going to need to handle some customization in our web application, so we'll need to run the `npm run eject` command in the root of our directory. This is a permanent action, which just means we'll be responsible for handling customizations to our app structure for now on (without the help of our handy `create-react-app`).

> This is where I *always* say make a backup copy of your application. We cannot go back from `ejecting`, but we can revert to old code.

We can *eject* from the `create-react-app` structure by running the eject command provided by the generator:

```
npm run eject
```

After *ejecting* from the `create-react-app` structure, we'll see we get a lot of new files in our application root in the `config/` and `scripts/` directories. The `npm run eject` command created all of the files it uses internally and wrote them all out for us in our application.

The key method of the `create-react-app` generator is called webpack (https://webpack.github.io), which is a module bundler/builder.

Webpack basics

Webpack is a module bundler with a ginormous community of users, tons of plugins, is under active development, has a clever plugin system, is incredibly fast, supports hot-code reloading, and much much more.

Although we didn't really call it out before, we've been using webpack this entire time (under the guise of `npm start`). Without webpack, we wouldn't have have been able to just write `import` and expect our code to load. It works like that because webpack "sees" the `import` keyword and knows we need to have the code at the path accessible when the app is running.

Webpack takes care of hot-reloading for us, nearly automatically, can load and pack many types of files into bundles, and it can split code in a logical manner so as to support lazy-loading and shrink the initial download size for the user.

This is meaningful for us as our apps grow larger and more complex, it's important to know how to manipulate our build tools.

For example, when we want to deploy to different environments... which we'll get to shortly. First, a tiny introduction to webpack, what it is and how it works.

What it does with `bundle.js`

Looking into the generated files when we ran `npm start` before we ejected the app, we can see that it serves the browser two or more files. The first is the `index.html` and the `bundle.js`. The webpack server takes care of injecting the `bundle.js` into the `index.html`, even if we don't load our app in the `index.html` file.

The `bundle.js` file is a giant file that contains *all* the JavaScript code our app needs to run, including dependencies and our own files alike. Webpack has it's own method of packing files together, so it'll look kinda funny when looking at the raw source.

Webpack has performed some transformation on all the included JavaScript. Notably, it used Babel to transpile our ES6 code to an ES5-compatible format.

If you look at the comment header for `app.js`, it has a number, `254`:

```
/* 254 */
/*!********************!*\
  !*** ./src/app.js ***!
  \********************/
```

The module itself is encapsulated inside of a function that looks like this:

```
function(module, exports, __webpack_require__) {
  // The chaotic `app.js` code here
}
```

Each module of our web app is encapsulated inside of a function with this signature. Webpack has given each of our app's modules this function container as well as a module ID (in the case of `app.js`, 254).

But "module" here is not limited to ES6 modules.

Remember how we "imported" the `makeRoutes()` function in `app.js`, like this:

```
import makeRoutes from './routes'
```

Here's what the variable declaration of `makeRoutes` looks like inside the chaos of the `app.js` Webpack module:

```
var _logo = __webpack_require__(/*! ./src/routes.js */ 255);
```

This looks quite strange, mostly due to the in-line comment that Webpack provides for debugging purposes. Removing that comment:

```
var _logo = __webpack_require__(255);
```

Instead of an `import` statement, we have plain old ES5 code.

Now, search for `./src/routes.js` in this file.

```
/* 255 */
/*!**********************!*\
  !*** ./src/routes.js ***!
  \**********************/
```

Note that its module ID is `255`, the same integer passed to `__webpack_require__` above.

Webpack treats *everything* as a module, including image assets like `logo.svg`. We can get an idea of what's going on by picking out a path in the mess of the `logo.svg` module. Your path might be different, but it will look like this:

```
static/media/logo.5d5d9eef.svg
```

If you open a new browser tab and plug in this address (your address will be different... matching the name of the file webpack generated for you):

```
http://localhost:3000/static/media/logo.5d5d9eef.svg
```

You should get the React logo:

So Webpack created a Webpack module for `logo.svg`, one that refers to the path to the SVG on the Webpack development server. Because of this modular paradigm, it was able to intelligently compile a statement like this:

```
import makeRoutes from './routes'
```

Into this ES5 statement:

```
var _makeRoutes = __webpack_require__(255);
```

What about our CSS assets? Yep, *everything* is a module in Webpack. Search for the string `./src/app.css`:

Webpack's `index.html` didn't include any references to CSS. That's because Webpack is including our CSS here via `bundle.js`. When our app loads, this cryptic Webpack module function dumps the contents of `app.css` into `style` tags on the page.

So we know *what* is happening: Webpack has rolled up every conceivable "module" for our app into `bundle.js`. You might be asking: Why?

The first motivation is universal to JavaScript bundlers. Webpack has converted all our ES6 modules into its own bespoke ES5-compatible module syntax. As we briefly touched on, it's wrapped all of our JavaScript modules in special functions. It provides a module ID system to enable one module to reference another.

Webpack, like other bundlers, consolidated all our JavaScript modules into a single file. It *could* keep JavaScript modules in separate files, but this requires some more configuration than `create-react-app` provides out of the box.

Webpack takes this module paradigm further than other bundlers, however. As we saw, it applies the same modular treatment to image assets, CSS, and npm packages (like React and ReactDOM). This modular paradigm unleashes a lot of power. We touch on aspects of that power throughout the rest of this chapter.

Complex, right?

It's okay if you don't understand that out of the box. Building and maintaining webpack is a complex project with lots of moving parts and it often takes even the most experienced developers a while to "get."

We'll walk through the different parts of our webpack configuration that we'll be working with. If it feels overwhelming, just stick with us on the basics here and the rest will follow.

With our newfound knowledge of the inner workings of Webpack, let's turn our attention back to our app. We'll make some modifications to our webpack build tool to support multiple environment configurations.

Environment configuration

When we're ready to deploy a new application, we have to think about a few things that we wouldn't have to focus on when developing our application.

For instance, let's say we are requesting data from an API server... when developing this application, it's likely that we are going to be running a development instance of the API server on our local machine (which would be accessible through `localhost`).

When we deploy our application, we'll want to be requesting data from an off-site host, most likely not in the same location from where the code is being sent, so `localhost` just won't do.

One way we can handle our configuration management is by using `.env` files. These `.env` files will contain different variables for our different environments, yet still provide a way for us to handle configuration in a sane way.

Usually, we'll keep one `.env` file in the root to contain a *global* config that can be overridden by configuration files on a per-environment basis.

Let's install an `npm` package to help us with this configuration setup called `dotenv`:

```
npm install --save-dev dotenv
```

The dotenv (https://github.com/motdotla/dotenv) library helps us load environment variables into the `ENV` of our app in our environments.

> It's usually a good idea to add `.env` to our `.gitignore` file, so we don't check in these settings.
>
> Conventionally, it's a good idea to create an example version of the `.env` file and check that into the repository. For instance, for our application we can create a copy of the `.env` file called `.env.example` with the required variables.
>
> Later, another developer (or us, months from now) can use the `.env.example` file as a template for what the `.env` file should look like.

These `.env` files can include variables as though they are unix-style variables. Let's create our global one with the `APP_NAME` set to 30days:

```
touch .env
echo "APP_NAME=30days" > .env
```

Let's navigate to the exploded `config/` directory where we'll see all of our build tool written out for us. We won't look at all of these files, but to get an understanding of *what* are doing, we'll start looking in `config/webpack.config.dev.js`.

This file shows all of the webpack configuration used to build our app. It includes loaders, plugins, entry points, etc. For our current task, the line to look for is in the `plugins` list where we define the `DefinePlugin()`:

```
module.exports = {
  // ...
  plugins: [
    // ...
    // Makes some environment variables available to the JS code, for
example:
    // if (process.env.NODE_ENV === 'production') { ... }. See
`./env.js`.
    // It is absolutely essential that NODE_ENV is set to production
    // during a production build.
    // Otherwise React will be compiled in the very slow development
mode.
    new webpack.DefinePlugin(env.stringified),
    // ...
  ]
}
```

The `webpack.DefinePlugin` plugin takes an object with `keys` and `values` and finds all the places in our code where we use the key and it replaces it with the value.

For instance, if the `env` object there looks like:

```
{
  '__NODE_ENV__': 'development'
}
```

We can use the variable `__NODE_ENV__` in our source and it will be replaced with 'development', i.e.:

```
class SomeComponent extends React.Component {
  render() {
    return (
      <div>Hello from {__NODE_ENV__}</div>
    )
  }
}
```

The result of the `render()` function would say "Hello from development".

To add our own variables to our app, we're going to use this `env` object and add our own definitions to it. Scrolling back up to the top of the file, we'll see that it's currently created and exported from the `config/env.js` file.

Looking at the `config/env.js` file, we can see that it takes all the variables in our environment and adds the `NODE_ENV` to the environment as well as any variables prefixed by `REACT_APP_`.

```
// ...
// Grab NODE_ENV and REACT_APP_* environment variables and prepare
them to be
// injected into the application via DefinePlugin in Webpack
configuration.
const REACT_APP = /^REACT_APP_/i;
// ...
function getClientEnvironment(publicUrl) {
  const raw = Object.keys(process.env)
    .filter(key => REACT_APP.test(key))
    .reduce(
      (env, key) => {
        env[key] = process.env[key];
        return env;
      },
      {
        // Useful for determining whether we're running in production
mode.
        // Most importantly, it switches React into the correct mode.
        NODE_ENV: process.env.NODE_ENV || "development",
        // Useful for resolving the correct path to static assets in
`public`.
        // For example, <img src={process.env.PUBLIC_URL +
'/img/logo.png'} />.
        // This should only be used as an escape hatch. Normally you
would put
        // images into the `src` and `import` them in code to get
their paths.
        PUBLIC_URL: publicUrl,
      }
    );
  // Stringify all values so we can feed into Webpack DefinePlugin
  const stringified = {
    "process.env": Object.keys(raw).reduce((env, key) => {
      env[key] = JSON.stringify(raw[key]);
      return env;
    }, {})
  };

  return { raw, stringified };
}
```

```
module.exports = getClientEnvironment;
```

We can skip all the complex part of that operation as we'll only need to modify the second argument to the reduce function, in other words, we'll update the object:

```
{
  // Useful for determining whether we're running in production mode.
  // Most importantly, it switches React into the correct mode.
  NODE_ENV: process.env.NODE_ENV || "development",
  // Useful for resolving the correct path to static assets in
`public`.
  // For example, <img src={process.env.PUBLIC_URL + '/img/logo.png'}
/>.
  // This should only be used as an escape hatch. Normally you would
put
  // images into the `src` and `import` them in code to get their
paths.
  PUBLIC_URL: publicUrl,
}
```

> This object is the *initial* object of the reduce function. The `reduce` function merges all of the variables prefixed by `REACT_APP_` *into* this object, so we'll always have the `process.env.NODE_ENV` replaced in our source.

Essentially what we'll do is:

1. Load our default `.env` file
2. Load any environment `.env` file
3. Merge these two variables together as well as any default variables (such as the `NODE_ENV`)
4. We'll create a new object with all of our environment variables and sanitize each value.

5. Update the initial object for the existing environment creator.

Let's get busy. In order to load the `.env` file, we'll need to import the `dotenv` package. We'll also import the `path` library from the standard node library and set up a few variables for paths.

Let's update the `config/env.js` file

```
var REACT_APP = /^REACT_APP_/i;
var NODE_ENV = process.env.NODE_ENV || 'development';

const path = require('path'),
      resolve = path.resolve,
      join = path.join;

const currentDir = resolve(__dirname);
const rootDir = join(currentDir, '..');

const dotenv = require('dotenv');
```

To load the global environment, we'll use the `config()` function exposed by the `dotenv` library and pass it the path of the `.env` file loaded in the root directory. We'll also use the same function to look for a file in the `config/` directory with the name of `NODE_ENV.config.env`. Additionally, we don't want either one of these methods to error out, so we'll add the additional option of `silent: true` so that if the file is not found, no exception will be thrown.

```
// 1. Step one (loading the default .env file)
const globalDotEnv = dotenv.config({
  path: join(rootDir, '.env'),
  silent: true
});
// 2. Load the environment config
const envDotEnv = dotenv.config({
  path: join(currentDir, NODE_ENV + `.config.env`),
  silent: true
});
```

Next, let's concatenate all these variables together as well as include our `NODE_ENV` option in this object. The `Object.assign()` method creates a *new* object and merges each object from right to left. This way, the environment config variable

```
const allVars = Object.assign(
  {},
  {
    NODE_ENV: NODE_ENV
  },
  globalDotEnv.parsed,
  envDotEnv.parsed
);
```

With our current setup, the `allVars` variable will look like:

```
{
  'NODE_ENV': 'development',
  'APP_NAME': '30days'
}
```

Now we can add this `allVars` as an argument to the reduce function initial value called in the `raw` variable in the `getClientEnvironment` function. Let's update it to use this object:

```
function getClientEnvironment(publicUrl) {
  // ...
  const raw = Object.keys(process.env)
    .filter(key => REACT_APP.test(key))
    .reduce(
      (env, key) => {
        env[key] = process.env[key];
        return env;
      },
      {
        // Useful for determining whether we're running in production
mode.
        // Most importantly, it switches React into the correct mode.
        NODE_ENV: process.env.NODE_ENV || "development",
        // Useful for resolving the correct path to static assets in
`public`.
        // For example, <img src={process.env.PUBLIC_URL +
'/img/logo.png'} />.
        // This should only be used as an escape hatch. Normally you
would put
        // images into the `src` and `import` them in code to get
their paths.
        PUBLIC_URL: publicUrl,
        ...allVars
      }
    );
  // ...
}
```

Now, anywhere in our code we can use the variables we set in our `.env` files.

Since we are making a request to an off-site server in our app, let's use our new configuration options to update this host.

Let's say by default we want the TIME_SERVER to be set to `http://localhost:3001`, so that if we don't set the `TIME_SERVER` in an environment configuration, it will default to localhost. We can do this by adding the `TIME_SERVER` variable to the global `.env` file.

Let's update the `.env` file so that it includes this time server:

```
APP_NAME=30days
TIME_SERVER='http://localhost:3001'
```

Now, we've been developing in "development" with the server hosted on heroku. We can set our `config/development.config.env` file to set the `TIME_SERVER` variable, which will override the global one:

```
TIME_SERVER='https://fullstacktime.herokuapp.com'
```

Now, when we run `npm start`, any occurrences of `process.env.TIME_SERVER` will be replaced by which ever value takes precedence.

Let's update our `src/redux/actionCreators.js` module to use the new server, rather than the hardcoded one we used previously.

```
// ...
const host = process.env.TIME_SERVER;
export const fetchNewTime = (timezone = "pst", str = "now") => ({
  type: types.FETCH_NEW_TIME,
  payload: new Date().toString(),
  meta: {
    type: "api",
    url: host + "/" + timezone + "/" + str + ".json"
  }
});
```

Now, for our production deployment, we'll use the heroku app, so let's create a copy of the `development.config.env` file as `production.config.env` in the `config/` directory:

```
cp config/development.config.env config/production.config.env
```

Custom middleware per-configuration environment

We used our custom logging redux middleware in our application. This is fantastic for working on our site in development, but we don't really want it to be active while in a production environment.

Let's update our middleware configuration to only use the logging middleware when we are in development, rather than in all environments. In our project's `src/redux/configureStore.js` file, we loaded our middleware by a simple array:

```
let middleware = [
  loggingMiddleware,
  apiMiddleware
];

export const configureStore = () => {
  // ...
  const store = createStore(rootReducer, initialState,
applyMiddleware(...middleware));
  // ...
}
```

Now that we have the `process.env.NODE_ENV` available to us in our files, we can update the `middleware` array depending upon the environment we're running in. Let's update it to only add the logging if we are in the development environment:

```
// ...
let middleware = [apiMiddleware];
if ("development" === process.env.NODE_ENV) {
  middleware.unshift(loggingMiddleware);
}
// ...
```

Now when we run our application in development, we'll have the `loggingMiddleware` set, while in any other environment we've disabled it.

Today was a long one, but tomorrow is an exciting day as we'll get the app up and running on a remote server.

Great work today and see you tomorrow!

Deployment

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-28/post.md)

Today, we'll look through some ready-to-go options so we can get our site up and running. By the end of today, you'll be able to share a link to your running application.

We left off yesterday preparing for our first deployment of our application. We're ready to deploy our application. Now the question is *where* and *what* are we going to deploy?

Let's explore...

What

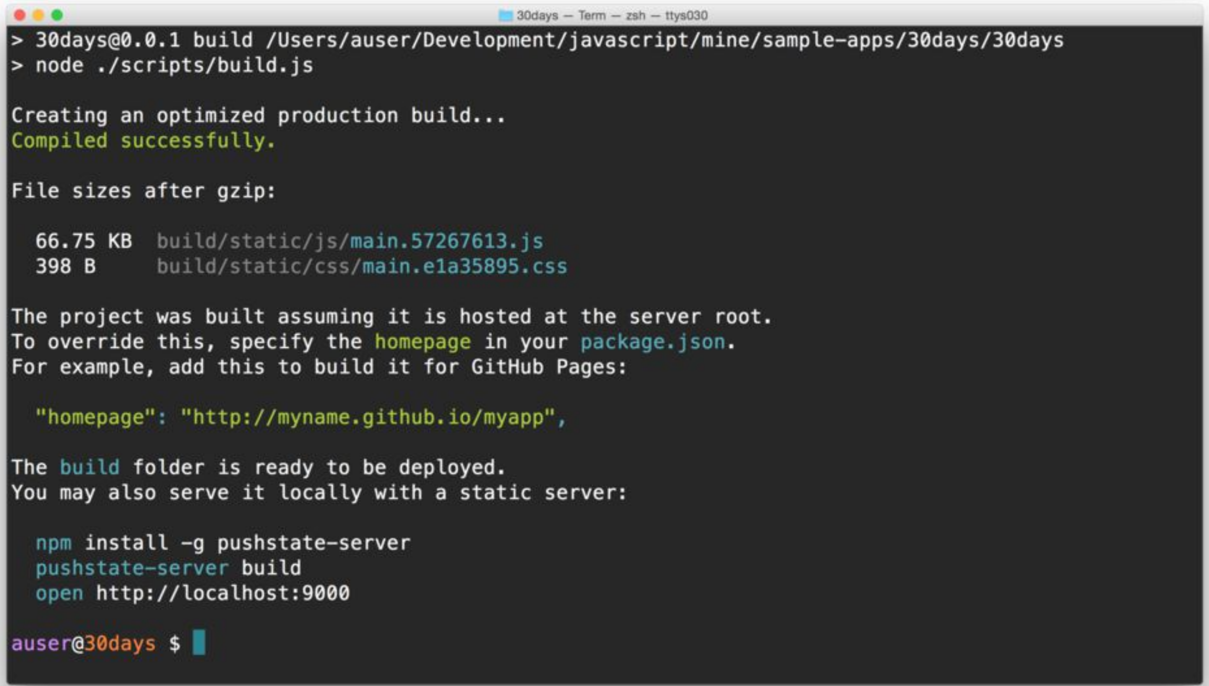
While deploying a single page application has it's own intricasies, it's similar to deploying a non-single page application. What the end-user's browser requests all need to be available for the browser to request. This means all javascript files, any custom fonts, images, svg, stylesheets, etc. that we use in our application need to be available on a publicly available server.

Webpack takes care of building and packaging our entire application for what we'll need to give the server to send to our clients. This *includes* any client-side tokens and our production configuration (which we put together yesterday).

This means we only need to send the contents of the distribution directory webpack put together. In our case, this directory is the `build/` directory. We don't need to send anything else in our project to a remote server.

Let's use our build system to generate a list of production files we'll want to host. We can run the `npm run build` command to generate these files in the `build/` directory:

```
npm run build
```



Where

These days we have many options for hosting client-side applications. We'll look at a few options for hosting our application today. Each of the following hosting services have their benefits and drawbacks, which we'll briefly discuss before we actually make a deployment.

There are two possible ways for us to deploy our application. If we are working with a back-end application, we can use the back-end server to host our public application files. For instance, if we were building a rails application, we can send the client-side application directly to the `public/` folder of the rails application.

This has the benefit of providing additional security in our application as we can verify a request from a client-side application made to the backend to have been generated by the server-side code. One drawback, however is that it can hog network bandwidth to send static files and potentially suck up resources from other clients.

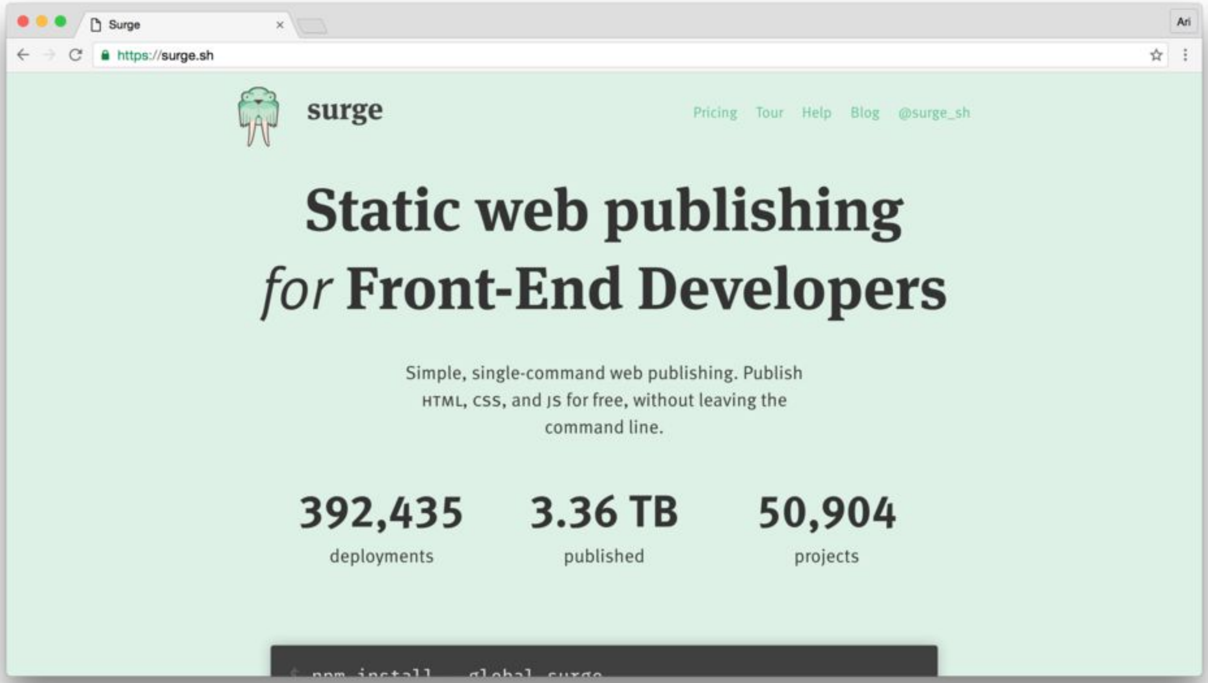
In this section, we'll work on hosting our client-side only application, which is the second way we can deploy our application. This means we can run/use a server which is specifically designed for hosting static files separate from the back-end server.

We'll focus on the second method where we are using other services to deploy our client-side application. That is, we'll skip building a back-end and upload our static files to one (or more) of the (non-exhaustive) list of hosting services.

- surge.sh (https://surge.sh/)
- github pages (https://pages.github.com/)
- heroku (https://www.heroku.com/)
- AWS S3 (https://aws.amazon.com/s3/)
- Forge (https://getforge.com/)
- BitBalloon (https://www.bitballoon.com/)
- Pancake (https://www.pancake.io/)
- ... More

We'll explore a few of these options together.

surge.sh



[surge.sh (https://surge.sh/)](https://surge.sh/) is arguably one of the easiest hosting providers to host our static site with. They provide a way for us to easily and repeatable methods for hosting our sites.

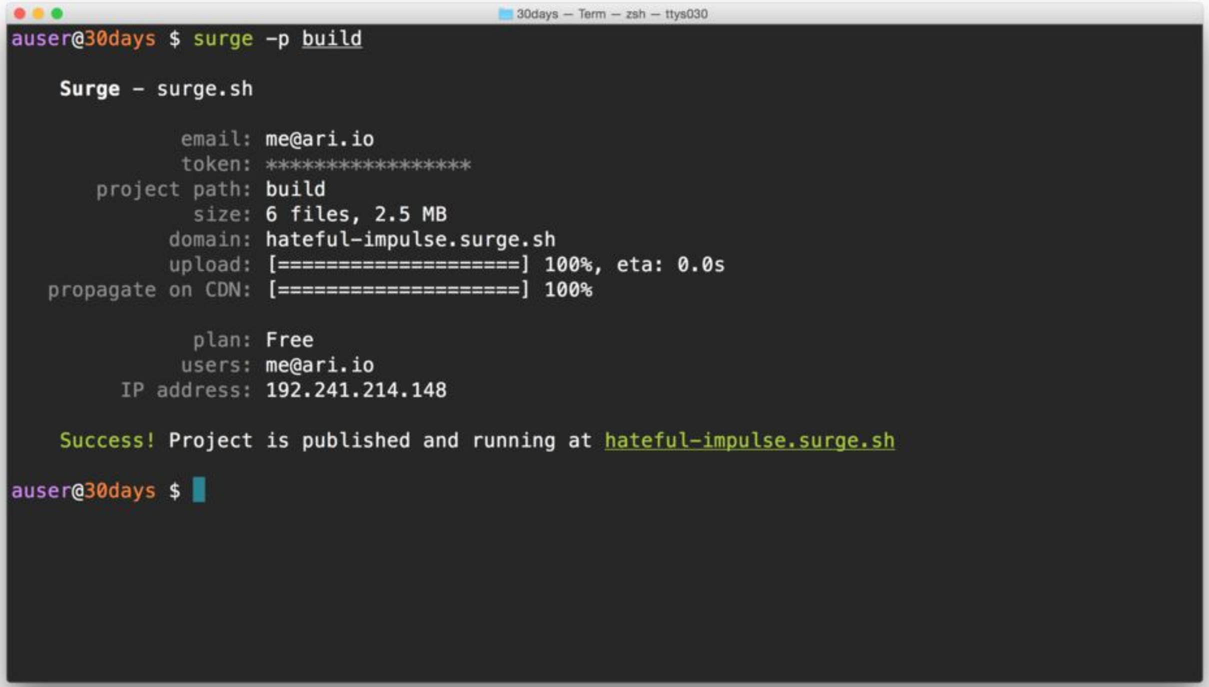
Let's deploy our application to surge. First, we'll need to install the `surge` command-line tool. We'll use `npm`, like so:

```
npm install --global surge
```

With the `surge` tool installed, we can run surge in our local directory and point it to the `build/` directory to tell it to upload the generated files in the `build/` directory.

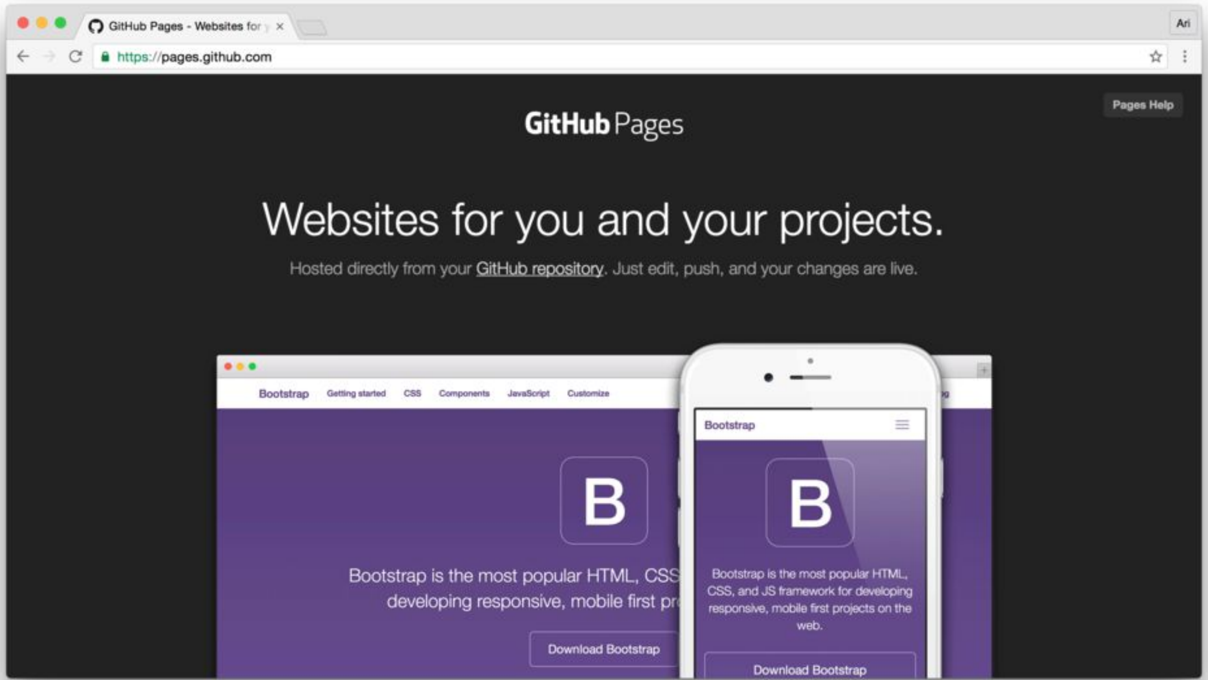
```
surge -p build
```

The surge tool will run and it will upload all of our files to a domain specified by the output of the tool. In the case of the previous run, this uploads our files to the url of [hateful-impulse.surge.sh (http://hateful-impulse.surge.sh/)](http://hateful-impulse.surge.sh/) (or the SSL version at [https://hateful-impulse.surge.sh/ (https://hateful-impulse.surge.sh/)](https://hateful-impulse.surge.sh/))



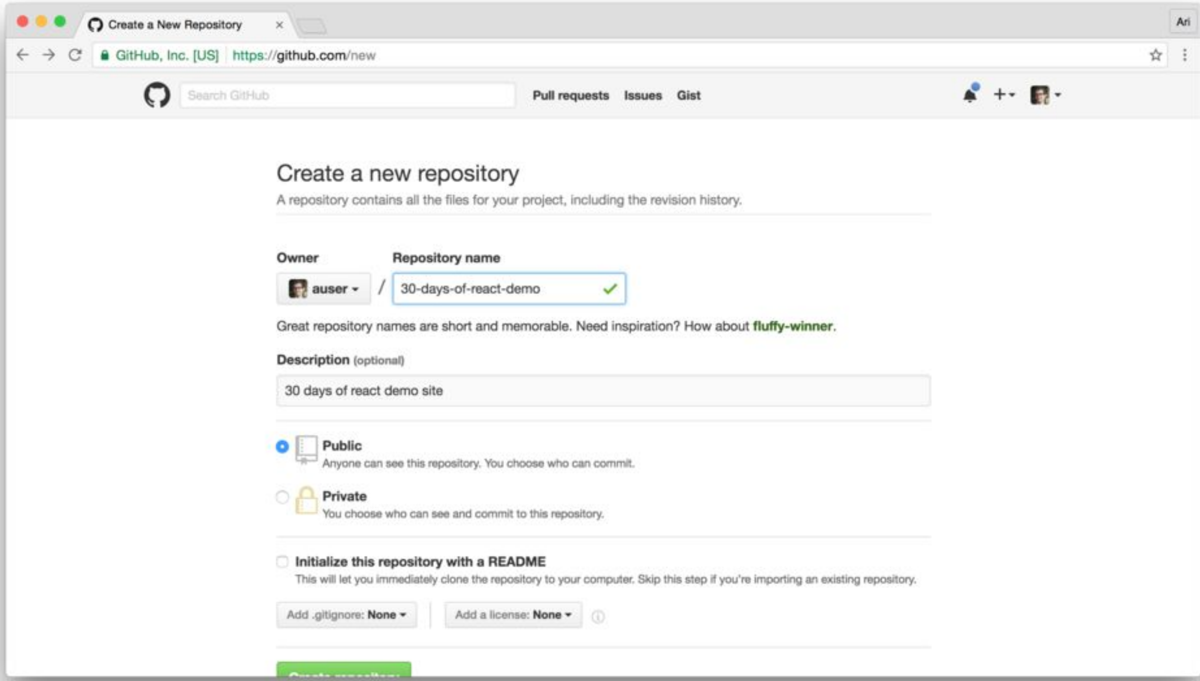
For more information on `surge`, check out their documentation at https://surge.sh/help/ (https://surge.sh/help/).

Github pages

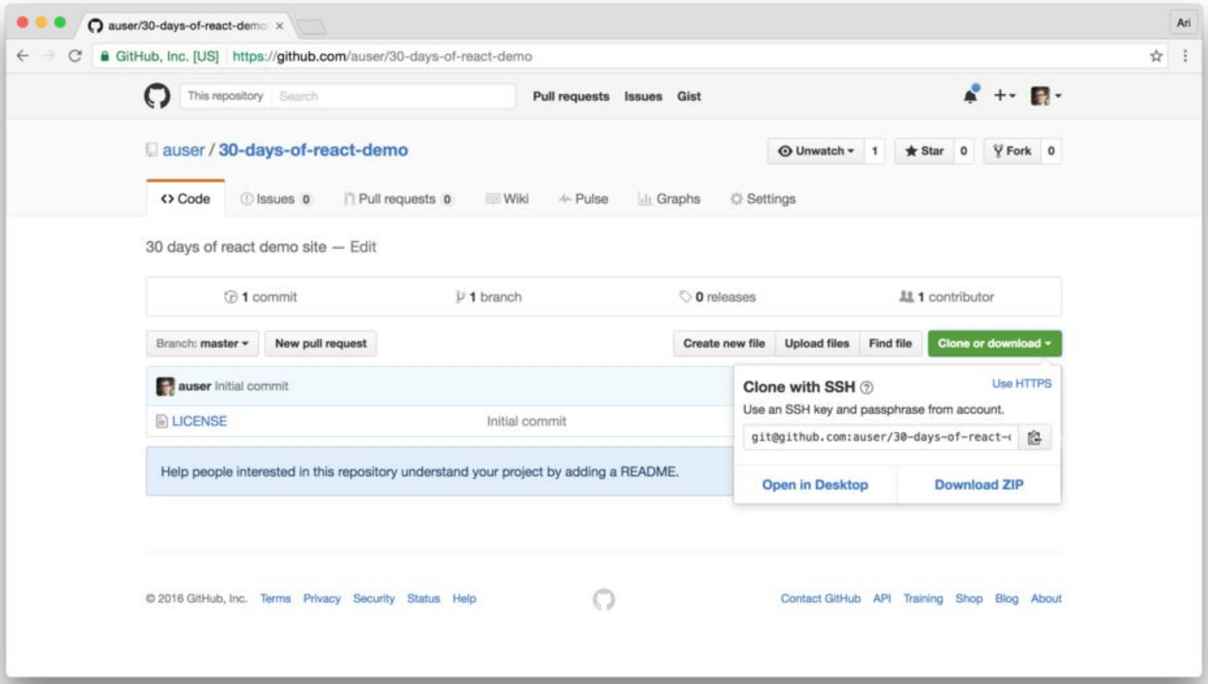


[github pages (https://pages.github.com/)](https://pages.github.com/) is another easy service to deploy our static files to. It's dependent upon using github to host our git files, but is another easy-to-use hosting environment for single page applications.

We'll need to start by creating our github pages repository on github. With an active account, we can visit the [github.com/new (https://github.com/new)](https://github.com/new) site and create a repository.



With this repo, it will redirect us to the repo url. Let's click on the `clone or download` button and find the github git url. Let's copy and paste this to our clipboard and head to our terminal.



In our terminal, let's add this as a remote origin for our git repo.

Since we haven't created this as a git repo yet, let's initialize the git repo:

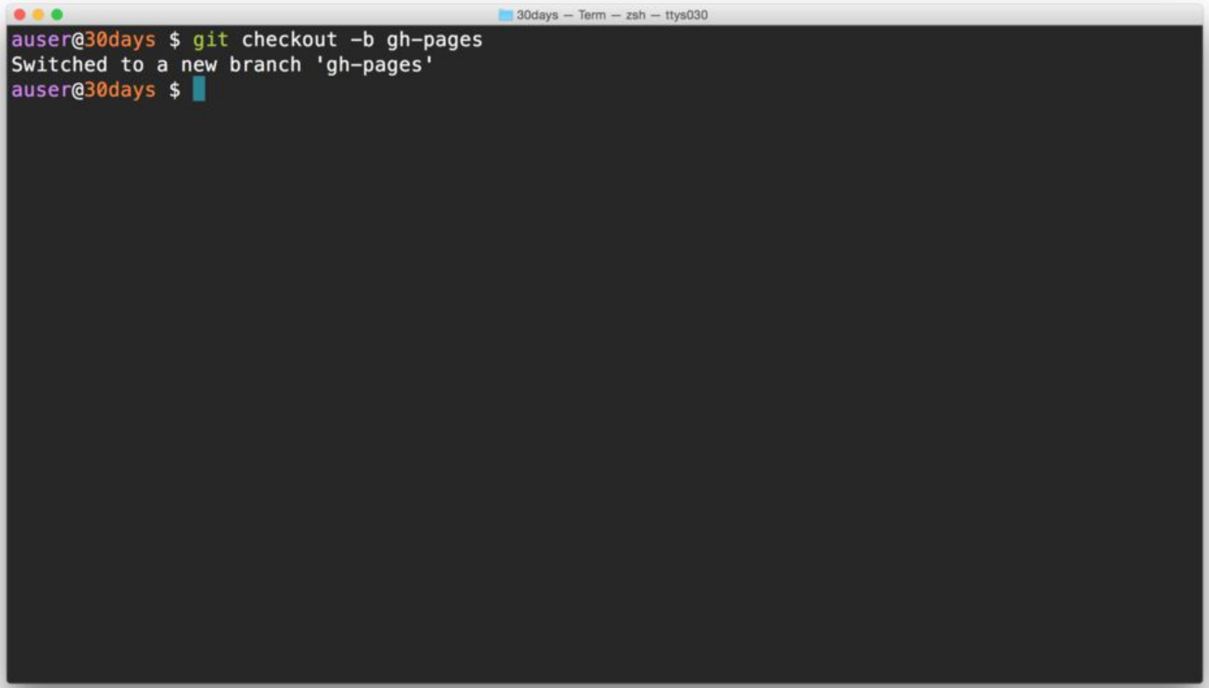
```
git init
git add -A .
git commit -am "Initial commit"
```

In the root directory of our application, let's add the remote with the following command:

```
git remote add github [your git url here]
# From the demo, this will be:
# git remote add origin git@github.com:auser/30-days-of-react-demo.git
```

Next, we'll need to move to a branch called `gh-pages` as github deploys from this branch. We can easily do this by checking out in a new branch using git. Let's also run the generator and tell git that the `build/` directory should be considered the root of our app:

```
npm run build
git checkout -B gh-pages
git add -f build
git commit -am "Rebuild website"
git filter-branch -f --prune-empty --subdirectory-filter build
git checkout -
```



Since github pages does not serve directly from the root, but instead the build folder, we'll need to add a configuration to our `package.json` by setting the `homepage` key to the `package.json` file with our github url. Let's open the `package.json` file and add the "homepage" key:

```
{
  "name": "30days",
  "version": "0.0.1",
  "private": true,
  "homepage": "http://auser.github.io/30-days-of-react-demo
(http://auser.github.io/30-days-of-react-demo)",
  // ...
}
```

Hint

We can modify json files by using the jq (https://stedolan.github.io/jq/) tool. If you don't have this installed, get it... get it now... It's invaluable

To change the `package.json` file from the command-line, we can use jq, like so:

```
jq '.homepage = \
    "http://auser.github.io/30-days-of-react-demo
(http://auser.github.io/30-days-of-react-demo)"' \
    > package.json
```

With our pages built, we can generate our application using `npm run build` and push to github from our local `build/` directory.

```
git push -f github gh-pages
```

Now we can visit our site at the repo pages url. For instance, the demo site is: https://auser.github.io/30-days-of-react-demo (https://auser.github.io/30-days-of-react-demo/#).

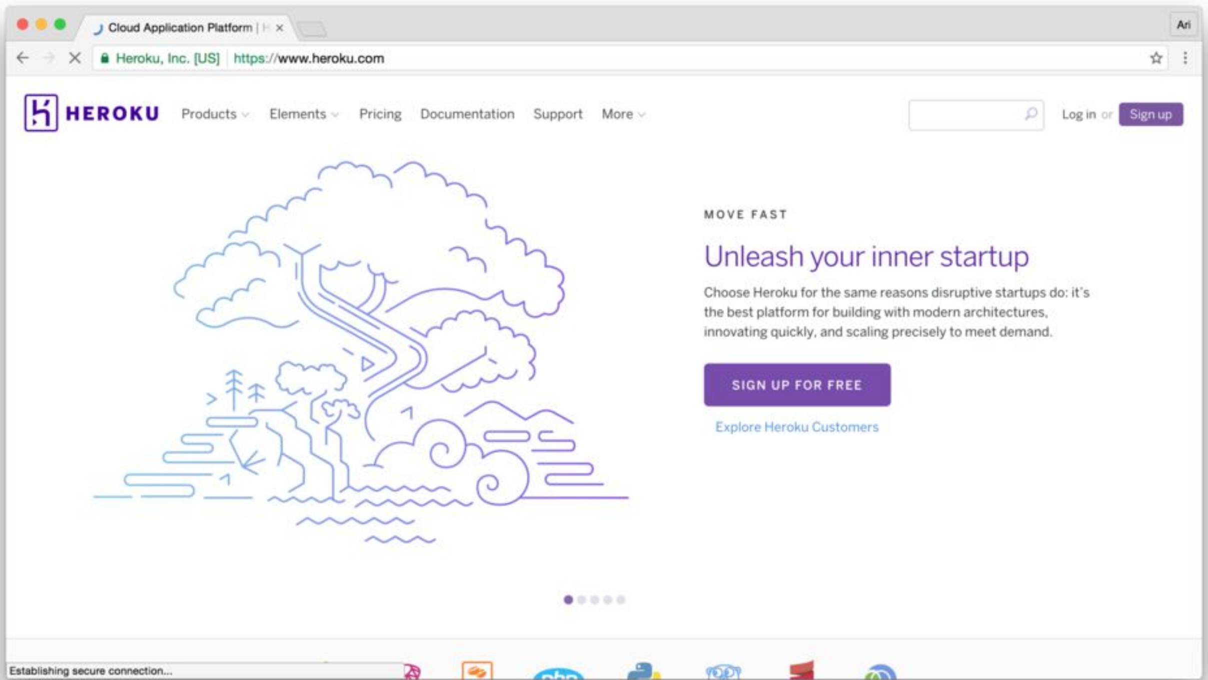
Future deployments

We'll need to add this work to a deployment script, so every time we want to release a new version of the site. We'll do more of this tomorrow. To release to github, we'll have to use the following script:

```
#!/usr/bin/env bash
git checkout -B gh-pages
git add -f build
git commit -am "Rebuild website"
git filter-branch -f --prune-empty --subdirectory-filter build
git push -f origin gh-pages
git checkout -
```

For more information on github pages, check out their documentation at https://help.github.com/categories/github-pages-basics/ (https://help.github.com/categories/github-pages-basics/).

Heroku



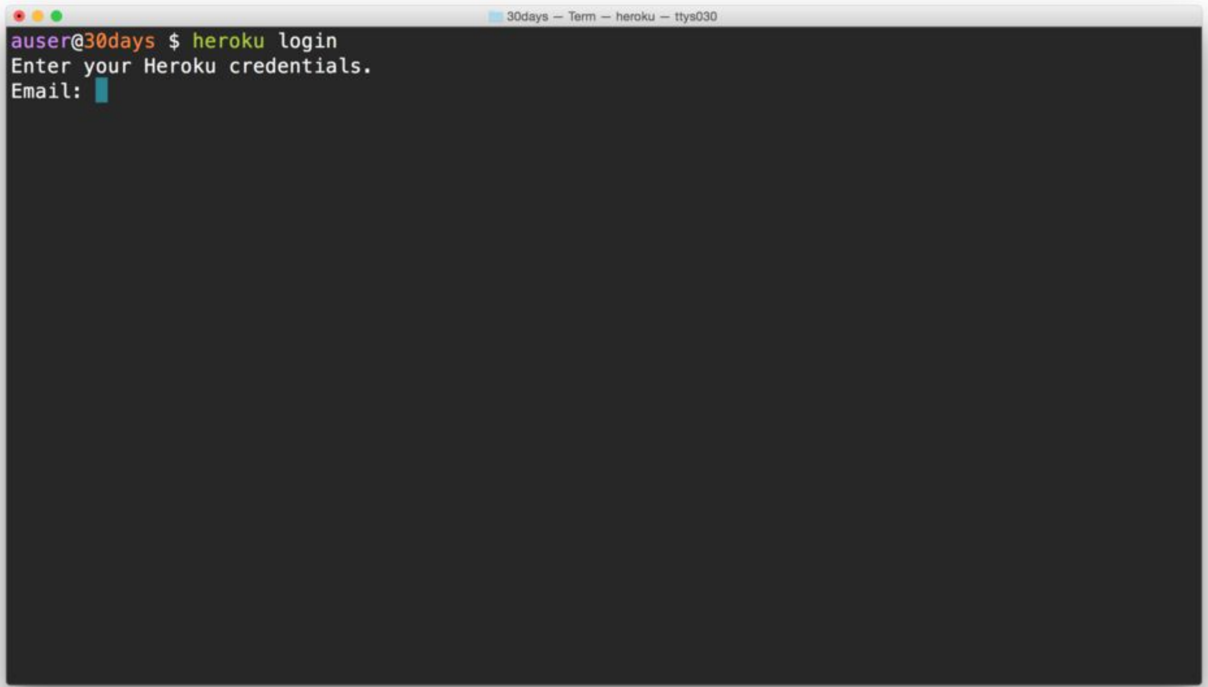
[Heroku (https://www.heroku.com/)](https://www.heroku.com/) is a very cool hosting service that allows us to host both static and non-static websites. We might want to deploy a static site to heroku as we may want to move to a dynamic backend at some point, are already comfortable with deploying to heroku, etc.

To deploy our site to heroku, we'll need an account. We can get one by visiting https://signup.heroku.com/ (https://signup.heroku.com/) to sign up for one.

We'll also need the heroku toolbet (https://devcenter.heroku.com/articles/heroku-command-line) as we'll be using the `heroku` command-line tool.

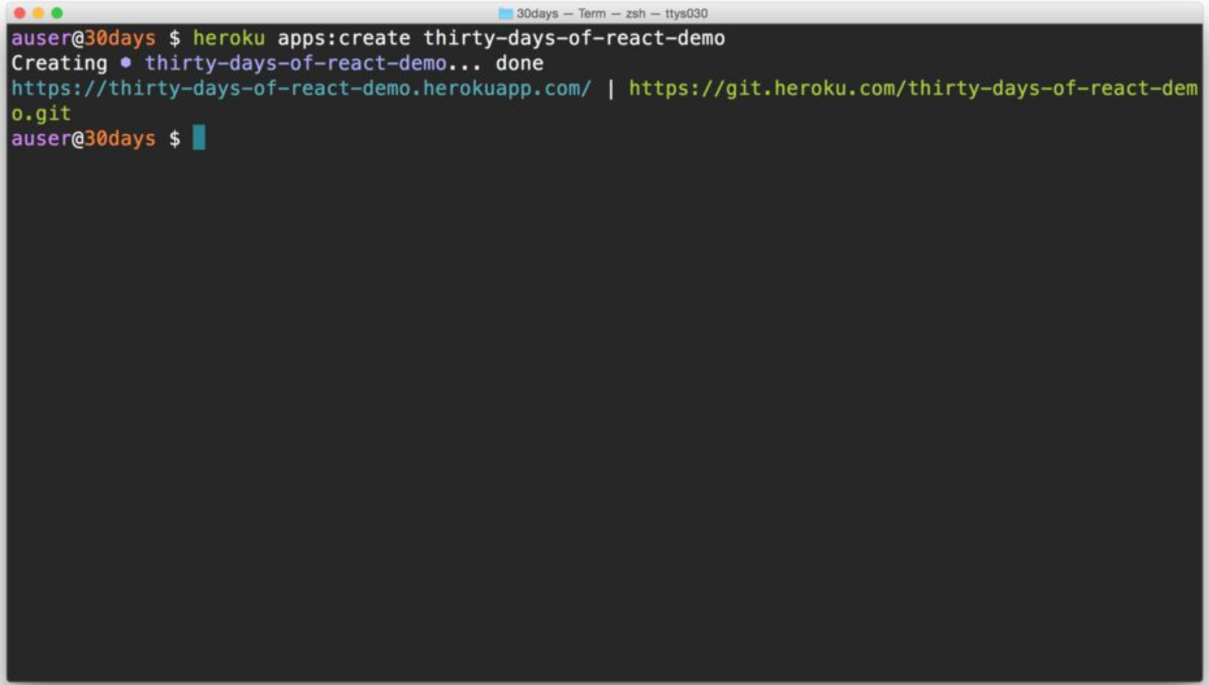
Finally, we'll need to run `heroku login` to set up credentials for our application:

```
heroku login
```



Next, we'll need to tell the `heroku` command-line that we have a heroku app. We can do this by calling `heroku apps:create` from the command-line in our project root:

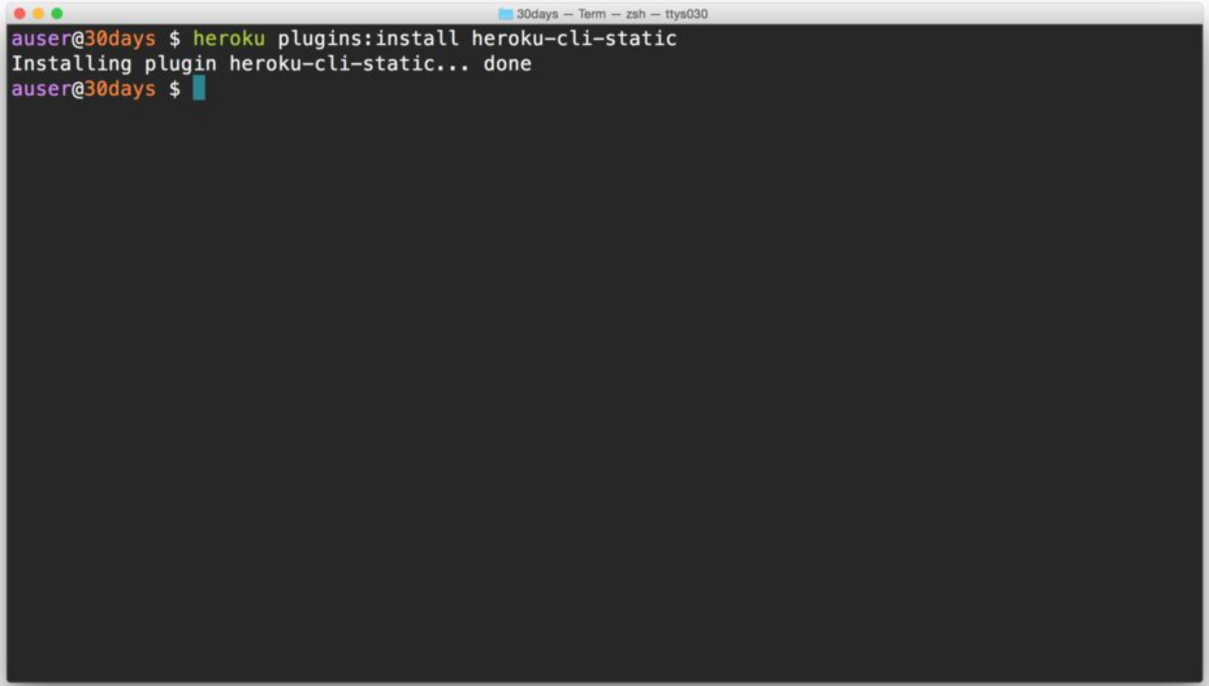
```
heroku apps:create
# or with a name
heroku apps:create thirty-days-of-react-demo
```



Heroku knows how to run our application thanks to buildpacks (https://devcenter.heroku.com/articles/buildpacks). We'll need to tell heroku we have a static-file buildpack so it knows to serve our application as a static file/spa.

We'll need to install the static-files plugin for heroku. This can be easiy install using the `heroku` tool:

```
heroku plugins:install heroku-cli-static
```

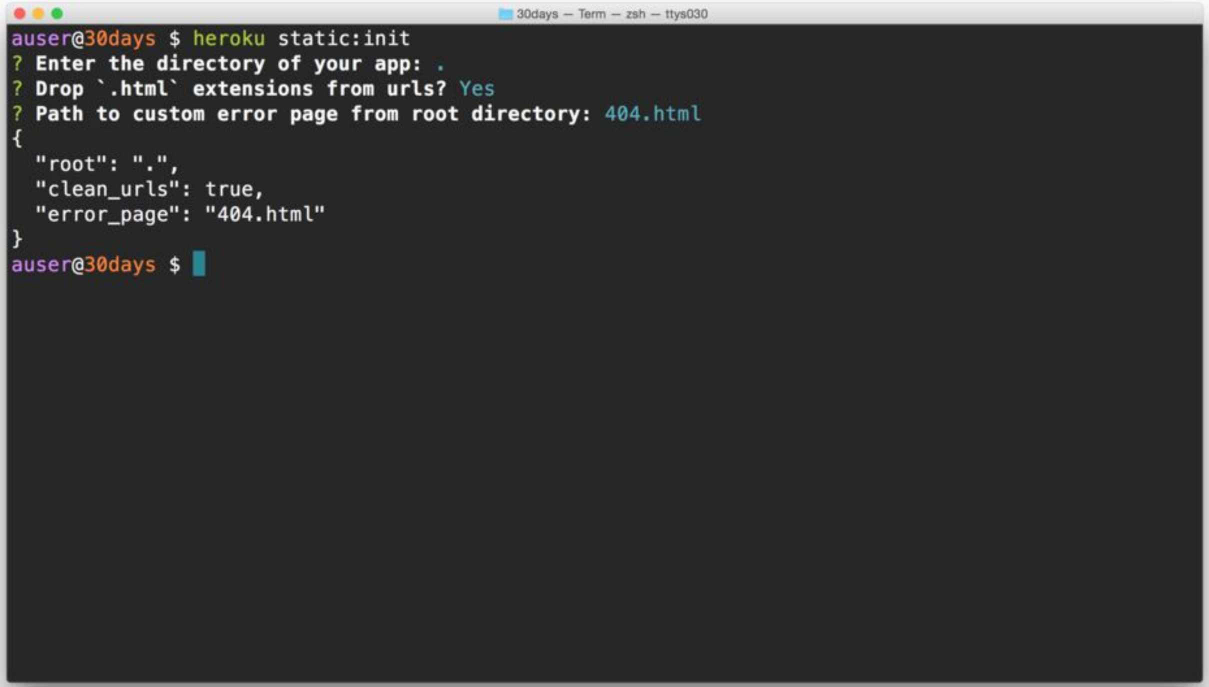


We can add the static file buildpack with the following command:

```
heroku buildpacks:set https://github.com/hone/heroku-buildpack-static
(https://github.com/hone/heroku-buildpack-static)
```

For any configuration updates, we'll need to run the `static:init` command from heroku to generate the necessary `static.json` file:

```
heroku static:init
```



Now we can deploy our static site to heroku using the `git` workflow:

```
git push heroku master
# or from a branch, such as the heroku branch
git push heroku heroku:master
```

We've deployed to only three of the hosting providers from the list above. There are many more options for deploying our application, however this is a pretty good start.

When we deploy our application, we will want to make sure everything is working before we actually send out the application to the world. Tomorrow we'll work on integrating a Continuous integration (CI, for short) server to run our tests before we deploy.

Continuous Integration

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-29/post.md)

Today we'll look through some continuous integration solutions available for us to run tests against as well as implement one to test our application in the cloud.

We've deployed our application to the "cloud", now we want to make sure everything runs as we expect it. We've started a test suite, but now we want to make sure it passes completely before we deploy.

We *could* set a step-by-step procedure for a developer to follow to make sure we run our tests before we deploy manually, but sometimes this can get in the way of deployment, especially under high-pressure deadlines in the middle of the night. There are better methods.

Testing then deployment

The core idea is that we want to deploy our application only *after* all of our tests have run and passed (sometimes known as "going green"). There are many ways we can do this. Mentioned above, we can handle it through humans, but that can become tedious and we're pretty good at forgetting things... what was I saying again?

Let's look at some better ways. One of the ways we can handle it is through a deployment script that only succeeds if all of our tests pass. This is the easiest, but needs to be replicated across a team of developers.

Another method is to push our code to a continuous integration server whose only responsibility is to run our tests and deploy our application if and only if the tests pass.

Just like hosting services, we have many options for running continuous integration servers. The following is a short list of some of the popular CI servers available:

- travis ci (https://travis-ci.org/)
- circle ci (https://circleci.com)
- codeship (https://codeship.io)
- jenkins (https://jenkins.io)
- AWS EC2 (https://aws.amazon.com/ec2/)

Let's look at a few ways of handling this process.

Custom build script

Without involving any extra servers, we can write a script to execute our tests before we deploy.

Let's create a script that actually does do the deployment process first. In our case, let's take the `surge.sh` example from yesterday. Let's add one more script we'll call `deploy.sh` in our `scripts/` directory:

```
touch scripts/deploy.sh
chmod u+x scripts/deploy.sh
```

In here, let's add the surge deploy script (changing the names to your domain name, of course):

```
#!/usr/bin/env bash
surge -p build --domain hateful-impulse.surge.sh
```

Let's write the release script next. To execute it, let's add the script to the `package.json` `scripts` object:

```
{
  // ...
  "scripts": {
    "start": "node ./scripts/start.js",
    "build": "node ./scripts/build.js",
    "release": "node ./scripts/release.js",
    "test": "node ./scripts/test.js"
  },
}
```

Now let's create the `scripts/release.js` file. From the root directory in our terminal, let's execute the following command:

```
touch scripts/release.js
```

Inside this file, we'll want to run a few command-line scripts, first our `build` step, then we'll want to run our tests, and finally we'll run the deploy script, if everything else succeeds first.

In a node file, we'll first set the `NODE_ENV` to be `test` for our build tooling. We'll also include a script to run a command from the command-line from within the node script and store *all* the output to an array.

```
process.env.NODE_ENV = "test";
process.env.CI = true;

var chalk = require("chalk");
const exec = require("child_process").exec;

var output = [];
function runCmd(cmd) {
  return new Promise((resolve, reject) => {
    const testProcess = exec(cmd, { stdio: [0, 1, 2] });

    testProcess.stdout.on("data", msg => output.push(msg));
    testProcess.stderr.on("data", msg => output.push(msg));
    testProcess.on("close", code => (code === 0 ? resolve() :
reject()));
  });
}
```

When called, the `runCmd()` function will return a promise that is resolved when the command exits successfully and will reject if there is an error.

Our release script will need to be able to do the following tasks:

1. build
2. test
3. deploy
4. report any errors

Mentally, we can think of this pipeline as:

```
build()
  .then(runTests)
  .then(deploy)
  .catch(error);
```

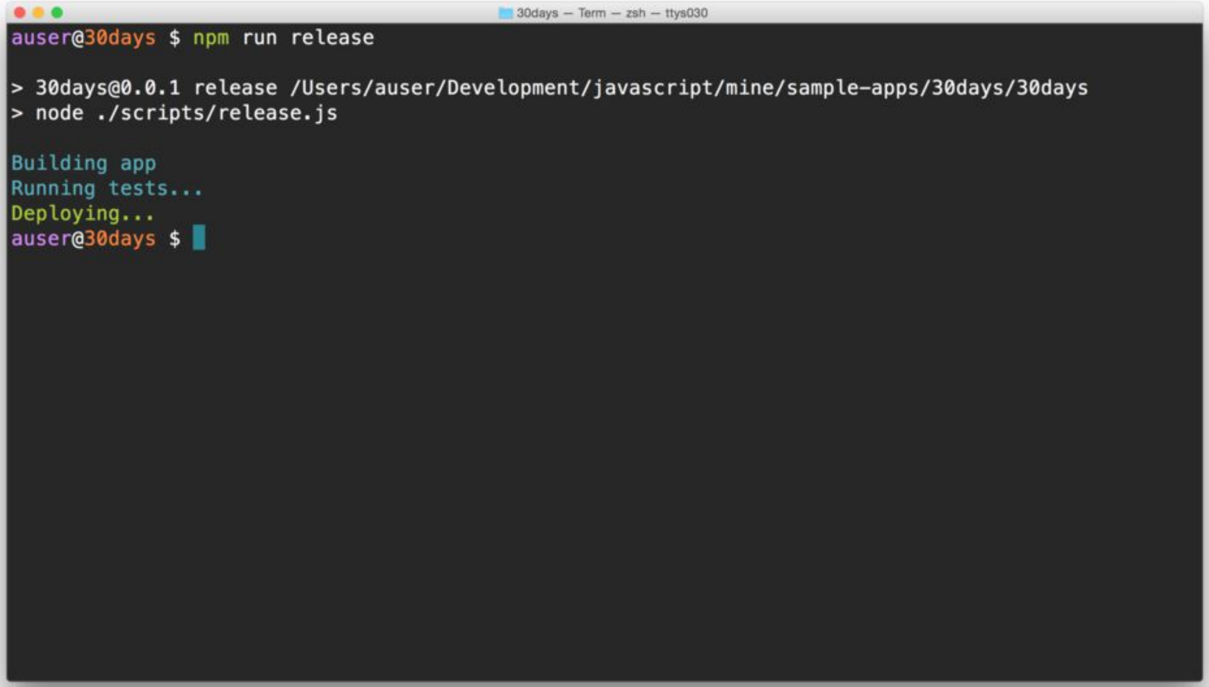
Let's build these functions which will use our `runCmd` function we wrote earlier:

```
function build() {
  console.log(chalk.cyan("Building app"));
  return runCmd("npm run build");
}

function runTests() {
  console.log(chalk.cyan("Running tests..."));
  return runCmd("npm test");
}

function deploy() {
  console.log(chalk.green("Deploying..."));
  return runCmd(`sh -c "${__dirname}/deploy.sh"`);
}

function error() {
  console.log(chalk.red("There was an error"));
  output.forEach(msg => process.stdout.write(msg));
}

build()
  .then(runTests)
  .then(deploy)
  .catch(error);
```

With our `scripts/release.js` file complete, let's execute our `npm run release` command to make sure it deploys:

```
npm run release
```

With all our tests passing, our updated application will be deployed successfully!



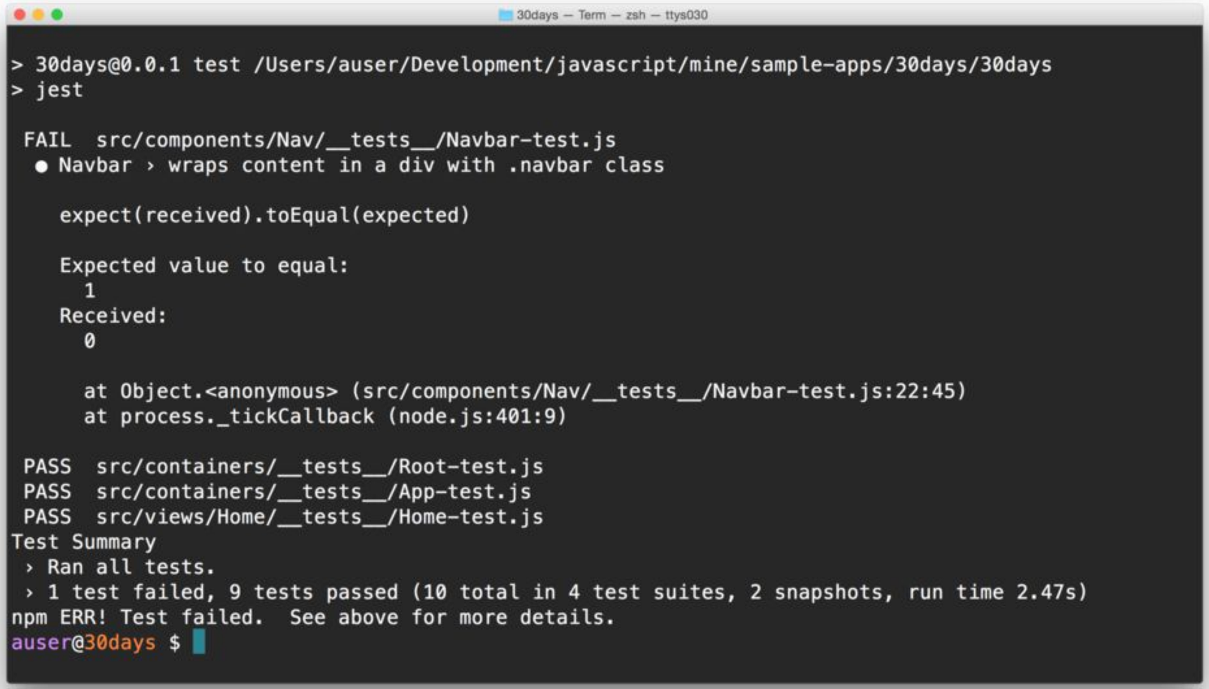
If any of our tests fail, we'll get all the output of our command, including the failure errors. Let's update one of our tests to make them fail purposefully to test the script.

I'll update the `src/components/Nav/__tests__/Navbar-test.js` file to change the first test to **fail**:

```
// ...
it("wraps content in a div with .navbar class", () => {
  wrapper = shallow(<Navbar />);
  expect(wrapper.find(".navbars").length).toEqual(1);
});
```

Let's rerun the `release` script and watch it fail and *not* run the deploy script:

```
npm run release
```

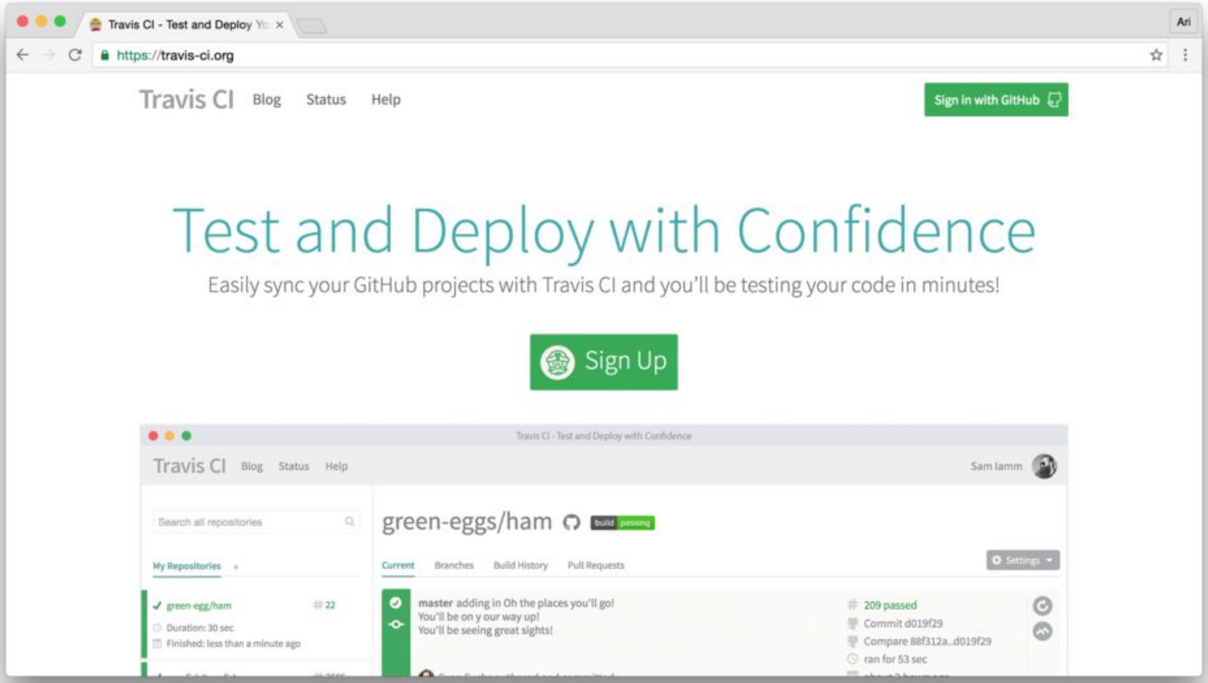


As we see, we'll get the output of the failing test in our logs, so we can fix the bug and then rerelease our application again by running the `npm run release` script again.

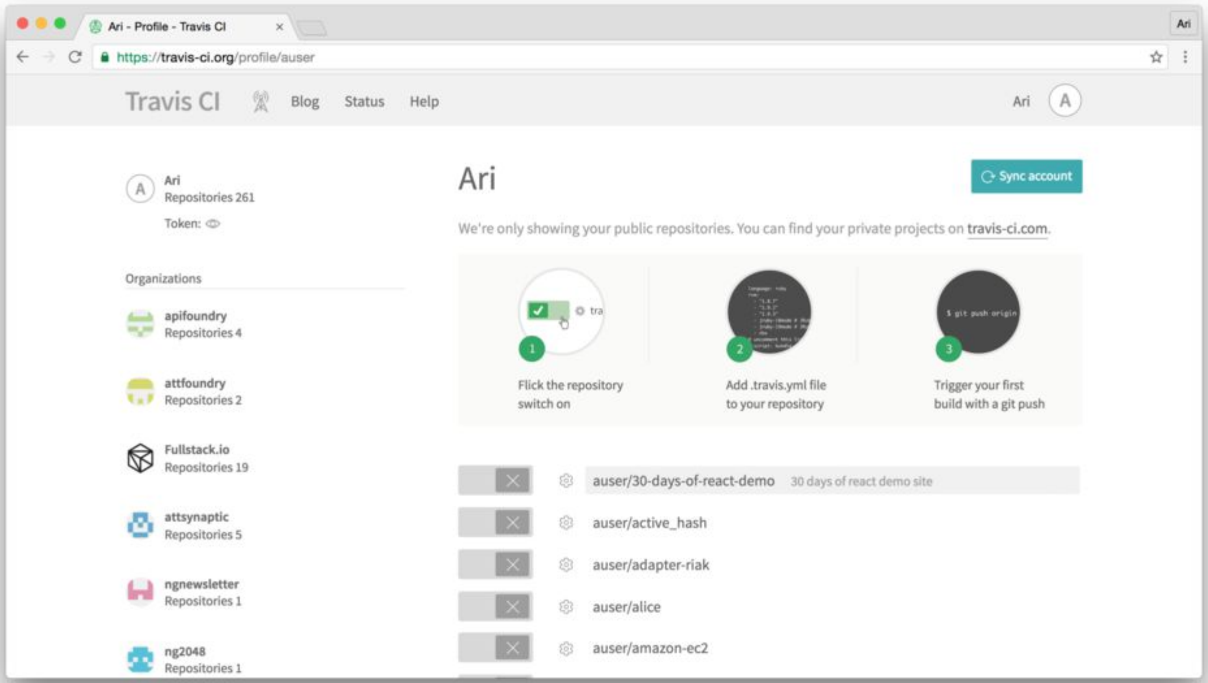
Travis CI

Travis ci (https://travis-ci.org/) is a hosted continuous integration environment and is pretty easy to set up. Since we've pushed our container to github, let's continue down this track and set up travis with our github account.

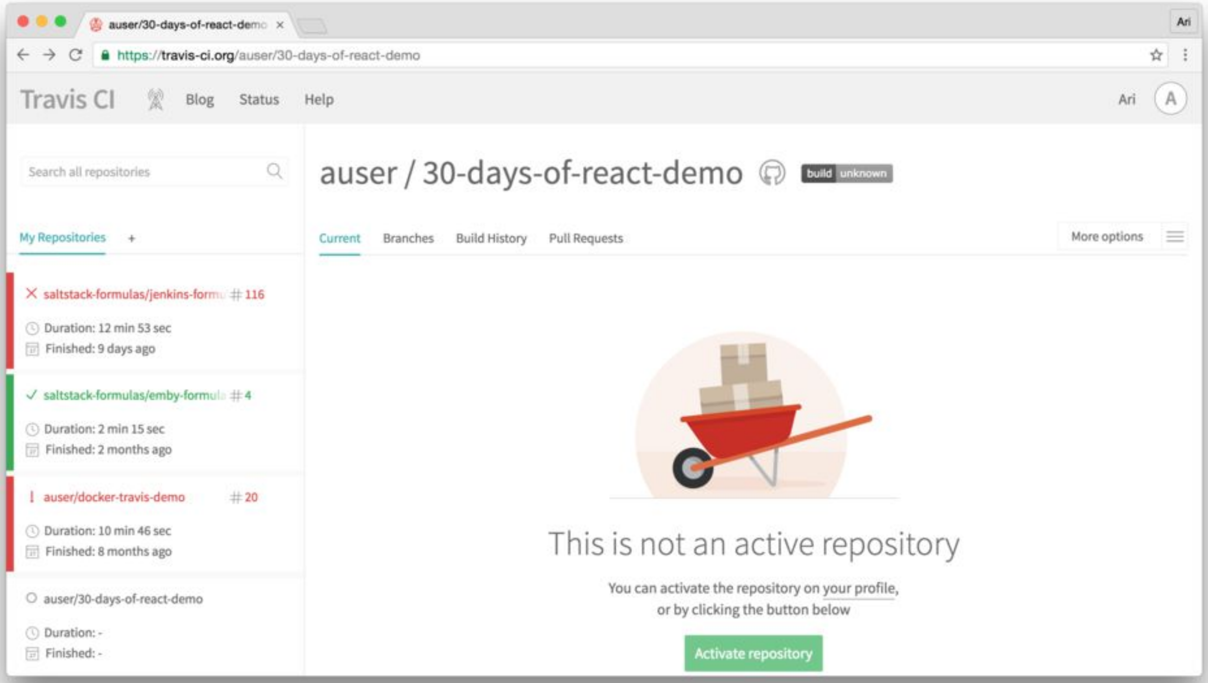
Head to travis-ci.org (https://travis-ci.org/) and sign up there.



Once you're signed up, click on the `+` button and find your repository:



From the project screen, click on the big 'activate repo' button.



To allow Travis CI to automatically log in for us during deployment, we need to add `SURGE_LOGIN` and `SURGE_TOKEN` environment variables. Open the *More Options* menu and click settings.

Under environment variables, create a variable called `SURGE_LOGIN` and set it to the email address you use with Surge. Next, add another variable called `SURGE_TOKEN` and set it to your Surge token.

> You can view your surge token by typing `surge token` in your terminal. Since we're using `surge` for depolyment, we should alsoadd it to our `devDependencies` in `package.json`. Run `npm install surge --save-dev` to add it

Now we need to configure travis to do what we want, which is run our test scripts and then deploy our app. To configure travis, we'll need to create a `.travis.yml` file in the root of our app.

```
touch .travis.yml
```

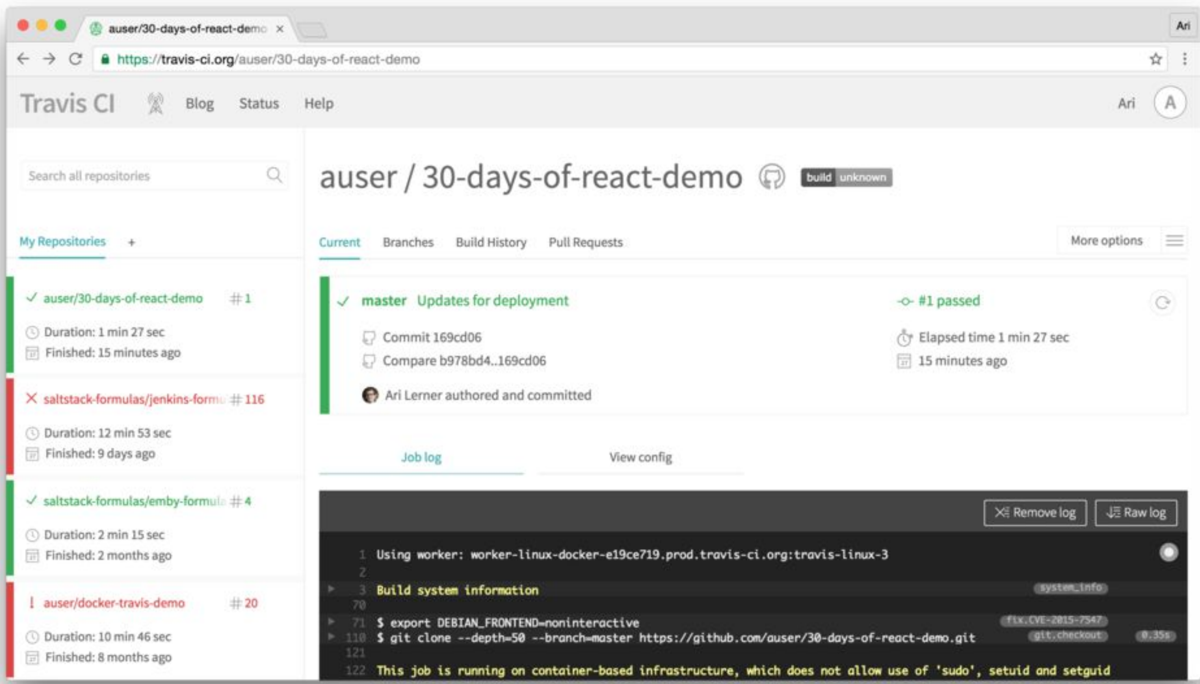
Let's add the following content to set the language to node with the node version of 10.15.0:

```
language: node_js
node_js:
  - "10.15.0"
```

Now all we need to do is add this file `.travis.yml` to git and push the repo changes to github.

```
git add .travis.yml
git commit -am "Added travis-ci configuration file"
git push github master
```

That's it. Now travis will execute our tests based on the default script of `npm test`.



Now, we'll want travis to actually deploy our app for us. Since we already have a `scripts/deploy.sh` script that will deploy our app, we can use this to deploy from travis.

To tell travis to run our `deploy.sh` script after we deploy, we will need to add the `deploy` key to our `.travis.yml` file. We also need to build our app before deploy, hence the `before_deploy`. Let's update the yml config to tell it to run our deploy script:

```
language: node_js
node_js:
  - "10.15.0"
before_deploy:
  - npm run build
deploy:
  provider: script
  skip_cleanup: true
  script: sh scripts/deploy.sh
  on:
    branch: master
```

The next time we push, travis will take over and push up to surge (or wherever the `scripts/deploy.sh` scripts will tell it to deploy).

Particulars for authentication. To deploy to github pages, we'll need to add a token to the script. The gist at https://gist.github.com/domenic/ec8b0fc8ab45f39403dd (https://gist.github.com/domenic/ec8b0fc8ab45f39403dd) is a great resource to follow for deploying to github pages.

Other methods

There are a lot of other options we have to run our tests before we deploy. This is just a getting started guide to get our application up.

> The Travis CI service is fantastic for open-source projects, however to use it in a private project, we'll need to create a billable account.
>
> An open-source CI service called Jenkins (https://jenkins.io) which can take a bit of work to setup (although it's getting a lot easier (https://jenkins.io/projects/blueocean/)).

Congrats! We have our application up and running, complete with testing and all.

See you tomorrow for our last day!

Wrap-up and More Resources

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-30/post.md)

We've made it! Day 30. Congrats! Now you have enough information to write some very complex applications, integrated with data, styled to perfection, tested and deployed.

Welcome to the final day! Congrats! You've made it!

The final component of our trip through React-land is a call to get involved. The React community is active, growing, and friendly.

Check out the community projects on Github at: https://github.com/reactjs (https://github.com/reactjs)

We've covered a lot of material in the past 30 days. The high-level topics we discussed in our first 30 days:

1. JSX and what it is, from the ground up.
2. Building components a. Static b. Data-driven components c. Stateful and stateless components d. Pure components e. The inherent tree-based structure of the virtual DOM
3. The React component lifecycle
4. How to build reusable and self-documenting components
5. How to make our components stylish using native React proptypes as well as third party libraries
6. Adding interaction to our components
7. How to use `create-react-app` to bootstrap our apps
8. How to integrate data from an API server, including a look at promises

9. We worked through the Flux architecture
10. Integrated Redux in our application, including how middleware works
11. We integrated testing strategies in our app a. Unit testing b. End-to-end testing c. Functional testing
12. We discussed deployment and extending our application to support multi-environment deployments
13. We added continuous integration in our deployment chain.
14. Client-side routing

There is so much more!

Although we covered a lot of topics in our first 30 days, there is so much more! How do we know? We wrote a book (https://www.fullstackreact.com)!

Interested in reading more and going deeper with React? Definitely check it out. Not only do we cover in-depth the topics we briefly introduced in our first 30 days, we go into a ton of other content, including (but not limited to):

- Using graphql and how to build a GraphQL server
- Relay and React
- How to use React to build a React Native application
- How to extend React Native to use our own custom modules
- An in-depth, much more comprehensive review of testing, from unit tests through view tests
- A deep look into components, from an internals perspective
- Advanced routing and dealing with production routing
- Forms forms forms! We cover form validations, from basic form inputs through validating and integrating with Redux
- And much much much more.

Just check out the book page at www.fullstackreact.com (https://www.fullstackreact.com) for more details.

Congrats on making it to day 30! Time to celebrate!