
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>
<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:

```
<!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:

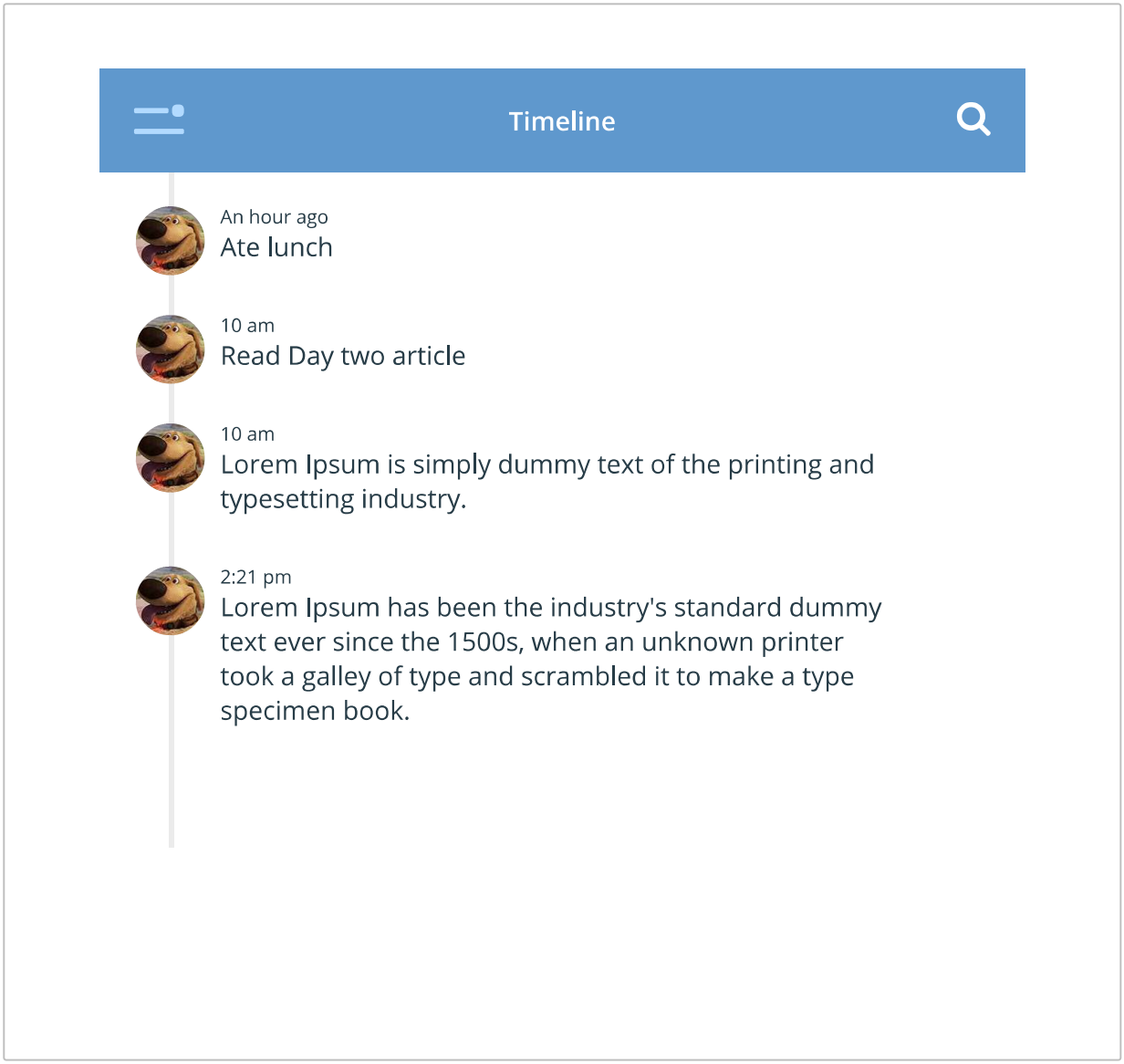
```
<link
  href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css"
  rel="stylesheet"
  type="text/css"
/>
```

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>
    )
  }
}
```

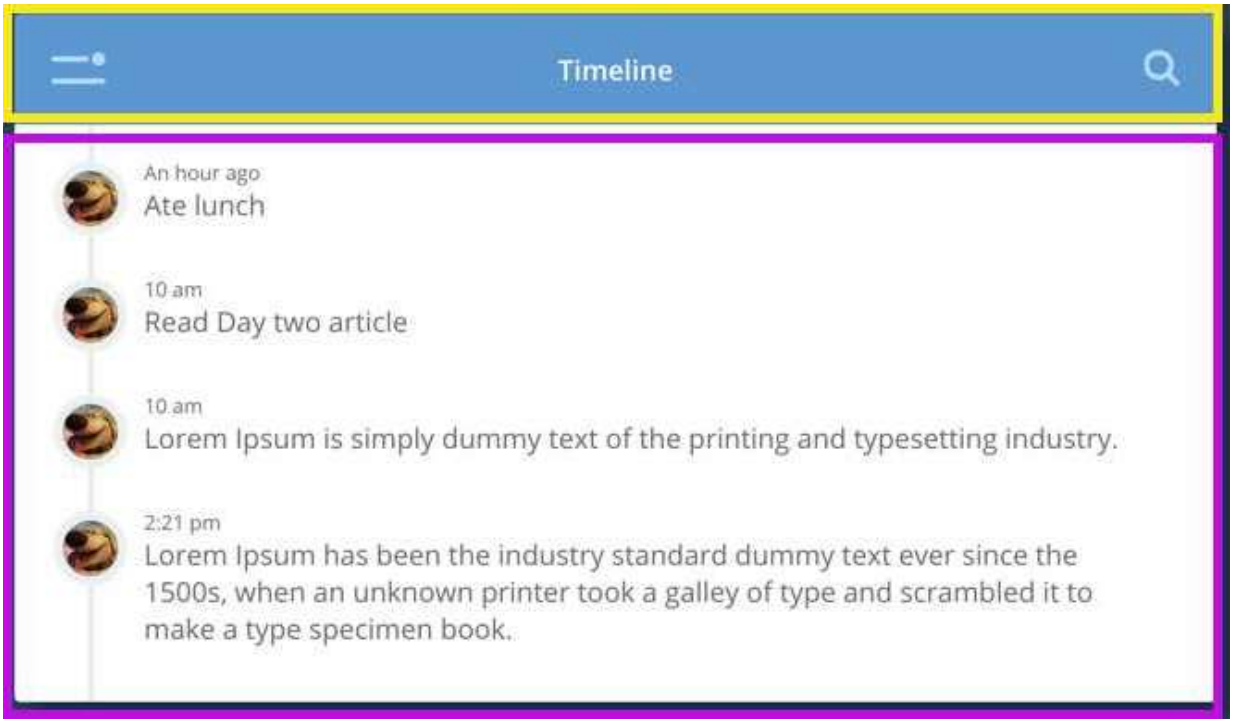


Breaking it down

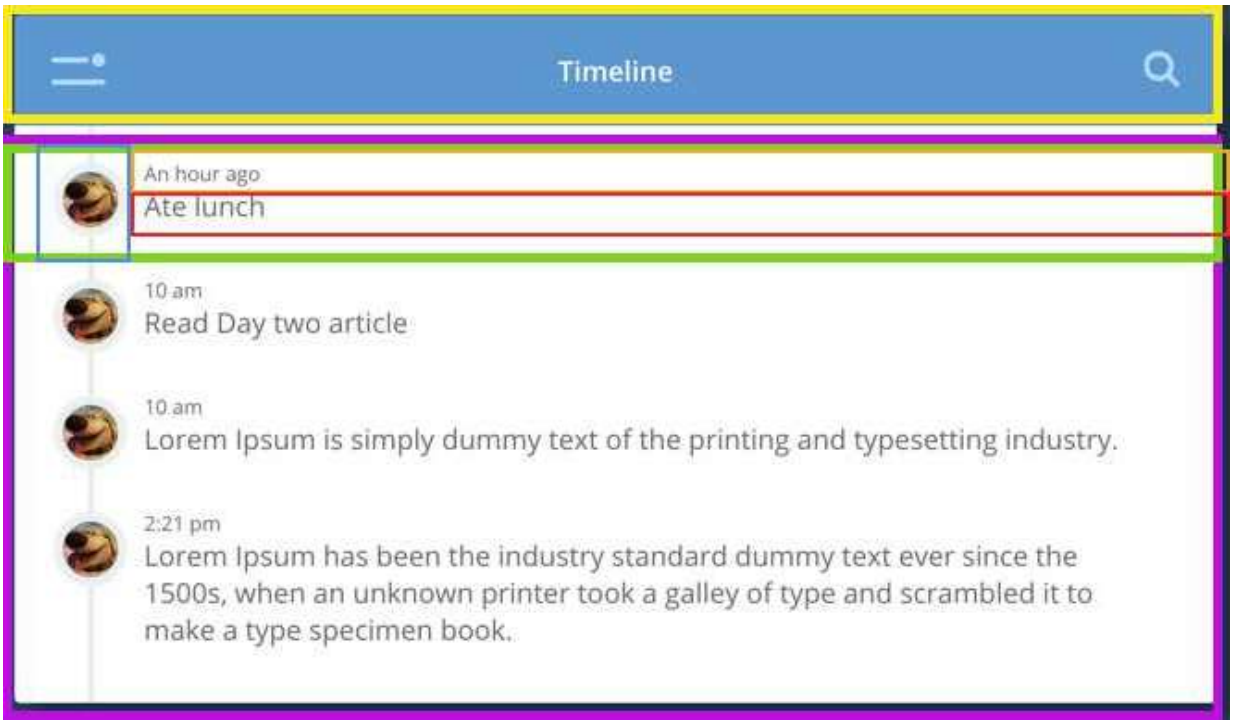
Rather than build this in a single component, let's break it down into multiple components.

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.

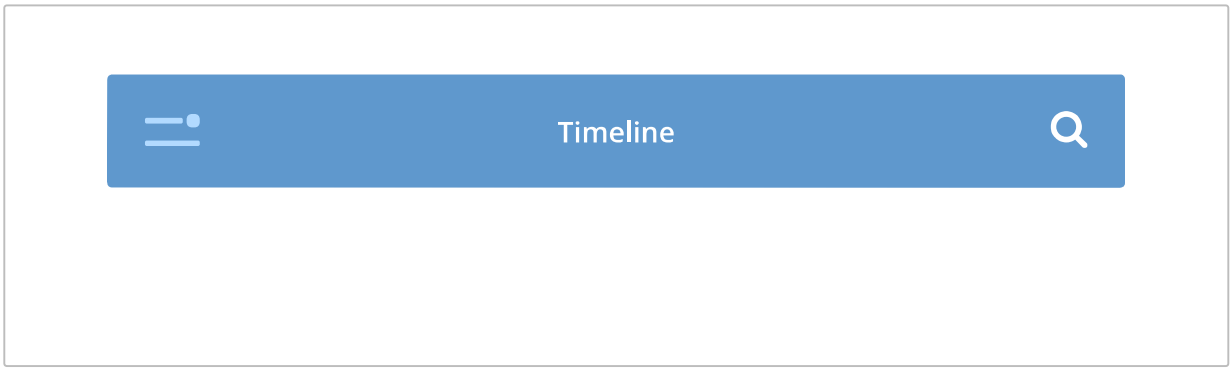
Child components

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 -->
```

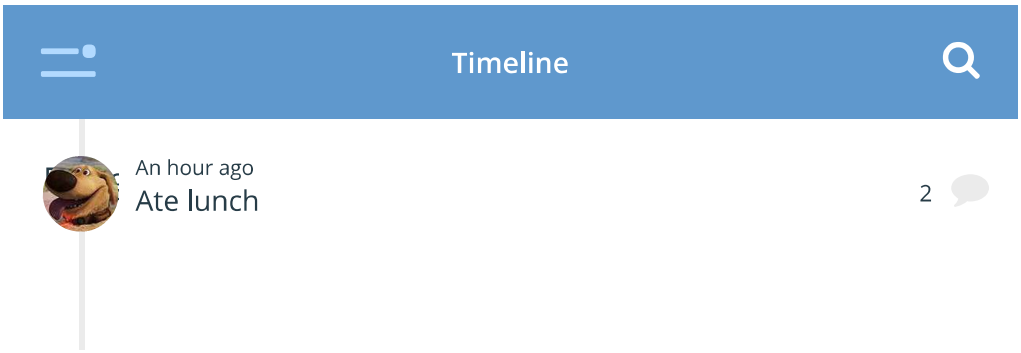
the React version of the comment must be in brackets:

```
{/* 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)

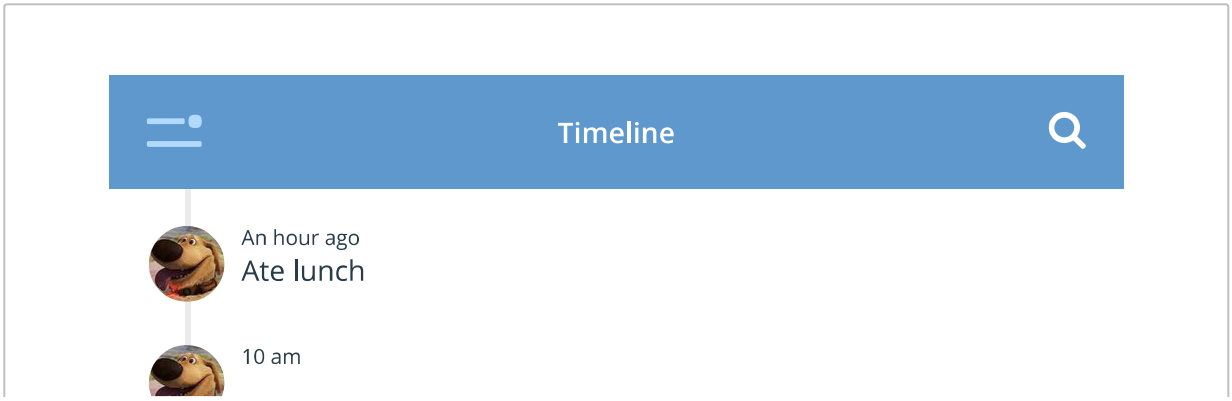
Hard-coding data in our applications isn't exactly ideal. Today, we'll set up our components to be driven by data to them access to external data.

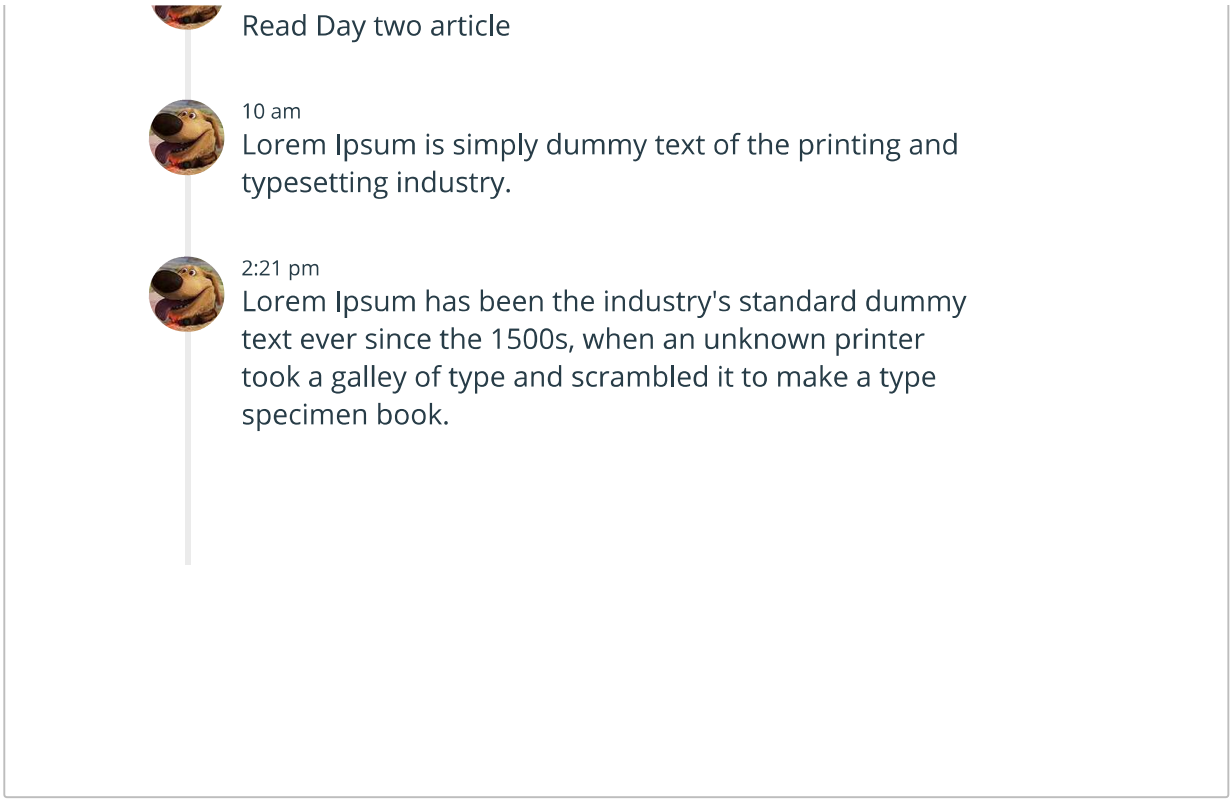
Through this point, we've written our first components and set them up in a child/parent relationship. However, we haven't yet tied any data to our React components. Although it's a more pleasant experience (in our opinion) writing a website in React, we haven't taken advantage of the power of React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

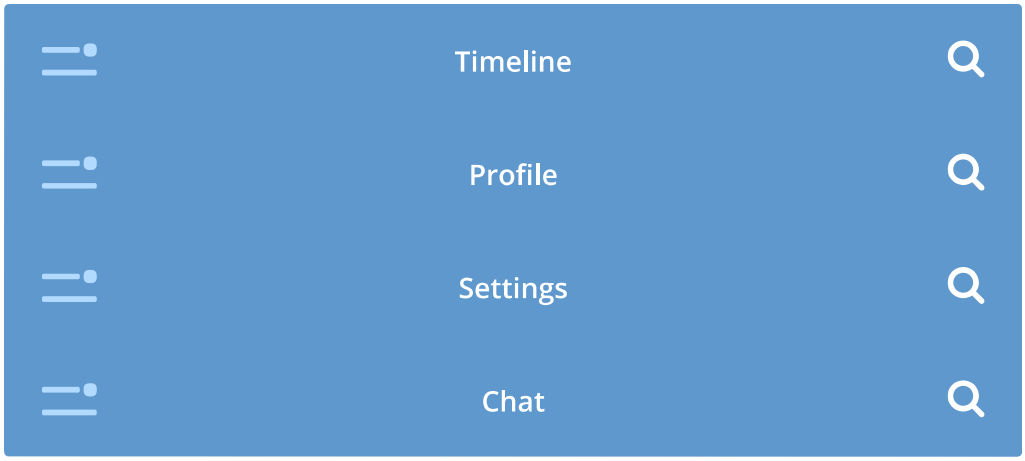
```
class Header extends React.Component {
  render() {
    return (
      <div className="header">
        <div className="menuIcon">
          <div className="dashTop"></div>
          <div className="dashBottom"></div>
          <div className="circle"></div>
        </div>

        <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' }]
  },
]
```

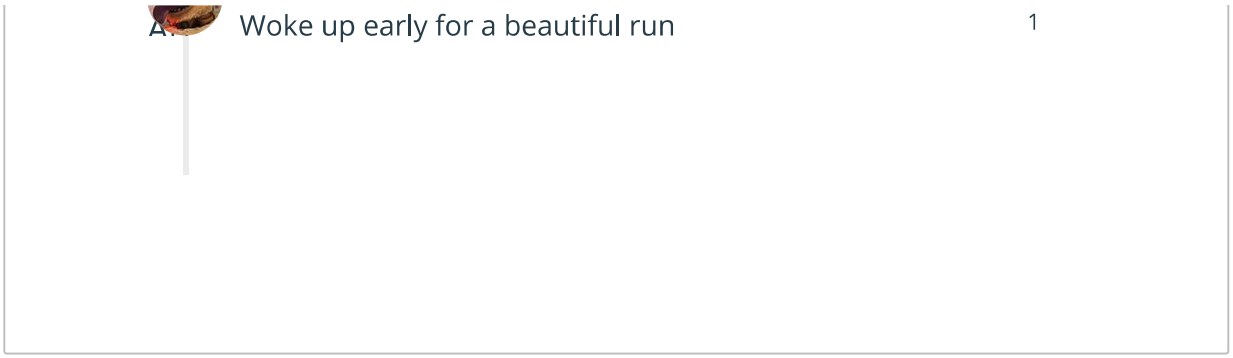
We can rearticulate our usage of `<Content />` by passing in multiple activities instead of just one:

```
<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.

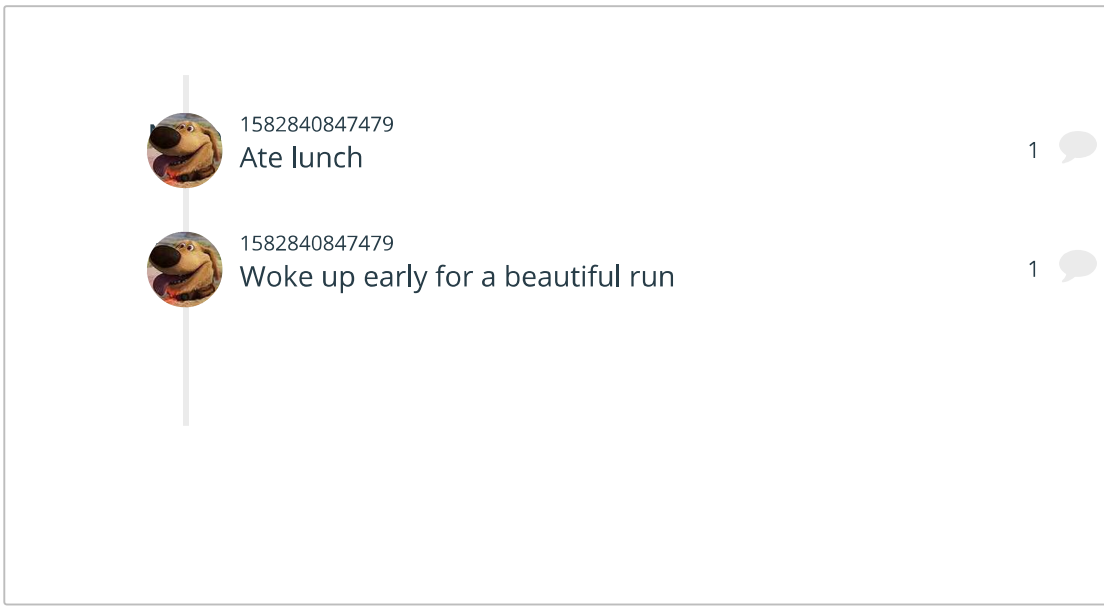
Let's update our `Content` component to display a list of `ActivityItem` components (we'll create this next).

```
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

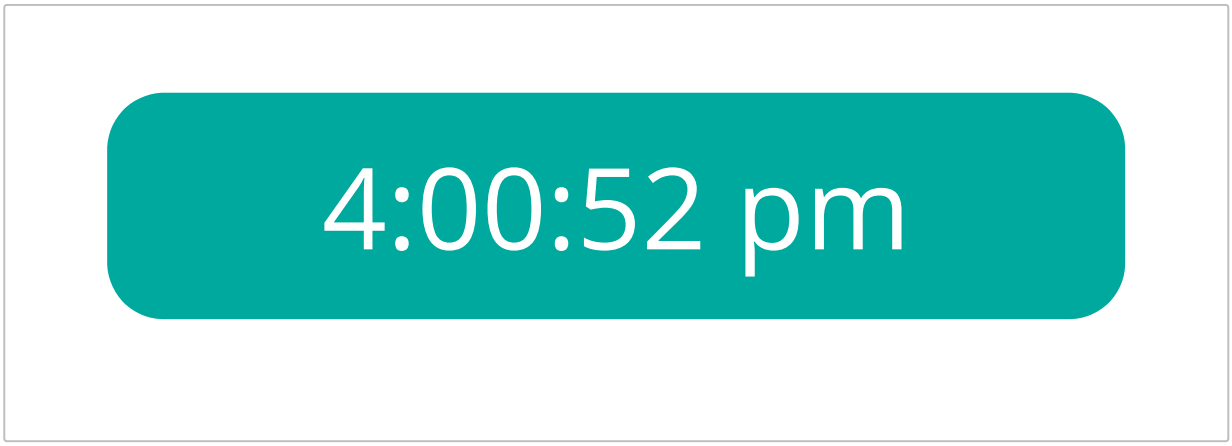
React does *not* allow us to modify `this.props` on our components for good reason. Imagine if we passed in the `title` prop to the `Header` component and the `Header` component was able to modify it. How do we know what the `title` is of the `Header` component? We set ourselves up for race-conditions, 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