tylermcginnis / **react-fundamentals**

Branch: **master** ▾ **react-fundamentals** / **Notes.md** Find file Copy path

edreeseg Corrected typo in notes.md 2877ba7 on 25 Nov 2018

2 contributors

386 lines (242 sloc) 31.5 KB Raw Blame History

React Fundamentals (tutorial on tylermcginnis.com)

Intro to the React Ecosystem

The following properties of react are what make it loved and popular amongst adopters and advocates:

- **Composition**
  - Everything in the react DOM is a component
  - Components are analogous to a container that wrap a bunch of functionality within themselves, much like functions do
  - Components can be used to compose other components much like functional composition
  - Well defined components can be used between different projects (hence the advent of component libraries)
- **Declarative**
  - A declarative solution/program abstracts the implementation/control flow handled by an imperative program with the help of functions.
  - Hence while writing a declarative solution one focusses on the WHAT rather than the HOW of the problem and uses the api that abstracts the how to do so.
  - React is declarative with everything except with changing the state of a component which anyways results in it being re-rendered
- **Unidirectional Dataflow**
  - In react the state is stored in a component as opposed to the DOM (which is how it is with JQuery)
  - Hence the state is explicitly changed and that causes the DOM to re-render
  - The data flows from the state to the DOM and not the other way around
  - Also parent components pass data to children components with the help of props (not sure if this should come here but this is true)
- **Explicit Mutations**
  - Changing the state has to be done explicitly in React
  - Since changing the state of a component with this.setState renders it to the DOM
    - there is no need of adding event listeners or dirty checking
- **Just JavaScript**
  - React takes advantage of the JavaScript programming language’s functionality, api and capabilities (also functional style)

This course will cover the following tools from the React Ecosystem:

- **React Router**
  - A component that allows us to map a url path to a component
  - Within the Router we declare the routes to be rendered
  - Each route has a path property and a component property that allows us to map a path to a component
  - The router is powerful and enables routes to have different children and makes active link states, et cetera extremely easy
- **Webpack**
  - The module bundler most popular within the react community

  - It helps us perform code transformations to the code with write to make it ready for the development server or for production
  - We specify a entry files which we work on, the output directory and filename webpack should produce and the loaders through which the specified files should be transformed
- **Babel**
  - Code transpiler used to transform JSX to JS and ES6 to ES5, et cetera
  - It is a loader in webpack
- **Axios**
  - Api used to send HTTP requests to different apis (in our case the GitHub api)

Setting up first React component with npm, webpack and babel

NPM

In writing software, it is an accepted idea that modularity is a great idea to for reusing code and increase the abilities and efficiencies of programmers. NPM is a node package manager that aims to solve the problem of enabling efficient collaboration and sharing of different modules.

The main problem with CDNs is maintaining updated versions of dependencies once their number starts going out of hand and sharing the same dependency files with team members working on the project.

npm init initialises a new npm project in current directory. It gives the directory a package.json file which contains information about the project’s dependencies with versions and also allows to run node scripts that perform different tasks (like testing ava, et cetera).

NPM allows us to install modules globally and locally only for the project. NPM has commands for saving individual modules globally or locally, to run tests mentioned in the scripts object and to install all dependencies mentioned in package.json.

`npm install —save` adds the module being installed to package.json as a dependency rather than a devDependency, which is what happens when we use `npm install —save-dev`

Webpack

Webpack is a module bundler that helps us make production and development transformations to the code we write. The reason for it to exist is that web developers shouldn’t have to transform code every time they want to test or deploy it. These transformations include but are not limited to: bundling js, bundling css, minification and uglification, jsx to js, sass/less to css, et cetera.

Webpack needs the entry point for the main javascript files. It has a modules property that we use to specify all loaders (rules) that need to make a transformation on the code. It has a plugins property which specifies plugins like html-webpack-config’s object in its array. It needs the template and filename in the output distribution folder.

Webpack loaders allow us to preprocess files (like css, et cetera) as we `require` them into the root js file.

The code that pieces the above together is fairly simple.

While using webpack-dev-server to run a web server locally, any changes we make do not cause webpack to compile our bundle to the dist folder. However, it dynamically updates quickly because webpack saves the changes in a cache that is meant to refresh quickly rather than compiling a build bundle every time a change is made.

Babel

Babel is a popular javascript transpiler. It enables us to write future versions / variations / abstractions of javascript and babel compiles them to javascript which is compatible with most browsers.

We need to install each babel preset package using npm as a dev dependency for the project we’re using it with.

In web pack’s config object’s module property’s rules array we need to use `babel-loader` on .js files.

In order to configure babel, we need to explicitly mention the `presets` corresponding to the transformations we wish to make in a babel config object.

We can either include this object in a file name `.babelrc` or we can refer it to a property babel in `package.json` .

React

React relies on a component based architecture in which the app is a component made up of components. Each component can be thought of as a collection of HTML, CSS, JS and some internal data specific to that component.

The way parent/child component relationship works is that the state of a component lives in its parent and has to be passed to the child with an attribute (called props/properties). This makes managing data simple because we know exactly where our data lives and we shouldn't manipulate it elsewhere.

A React Component may be composed of the following:

- [internal data]
- ui
- [lifecycle event]

Every component is supposed to have a render method. The reason is that the render method returns the template for that component and it is necessary for a component to have a ui.

We need to tell ReactDOM to which element the components should be rendered to. You usually have to use ReactDOM.render only once in your applications because rendering the most parent element will render all the children as well.

JSX is converted to React.createElement methods which describes what you see on the screen (notice only describes, doesn't mean that it is what we see). React.createElement returns an object representation of the DOM node. It is also called virtual DOM node.

React interprets JSX and transforms it into lightweight javascript objects which are used to create a virtual DOM. Changes in the virtual dom are tracked on only the necessary updates are rendered to the DOM.

`React.createElement` takes 3 arguments:

- element type: `div` , `span` , component
- properties object
- children (multiple) When React encounters a component in any of the above arguments, it replaces that with what the components React.createElement returns. Hence when rendering the most parent component using ReactDOM, the entire virtual DOM is created.

This invocation of React.createElement to create a virtual DOM node only happens while using ReactDOM.render and while changing state using setState.

The process looks something like this,

Signal to notify our app some data has changed -> re-render virtual dom -> diff previous virtual dom with new virtual dom -> only update real dom with necessary changes. This gives react performance ups.

Lecture video

npm install —-save-dev means add this to the object referenced by the devDependencies property of package.json.

While npm install —-save means add this to the object referenced by the dependencies property of package.json.

One can add scripts to package.json that can be run with the script name as argument to npm. These scripts can range from webpack productiont, webpack-dev-server to ava test scripts.

From the example mentioned in NPM text section of this lesson:

```
"scripts": {
  "test": "ava 'app/**/*.test.js' --verbose --require ./other/setup-ava-tests.js"
}
```

Dataflow with props

We already know that props are the primary way to pass data into a component (when it is created a.k.a. invoked within a parent component or the ReactDOM.render method). In this lesson we got hands on practice with passing props like username, name and img url to a component from its creation in the ReactDOM.render method and its invocation in a parent component. We also replaced multiple props in a parent component with a single property which takes a javascript object.

In JSX a javascript expression can be written within single curly braces {*javascript goes here*}. The primary way to build list uis in React is to simply pass in an array or other iterator and use javascript's functionality of map and filter, et cetera.

When rendering list items in jsx (`<li> {something} </li>`), React gives us a warning saying that we should add a unique key prop value pair to each list item. This allows React to keep track of changes in the list and reflect them on the ui in an efficient manner. Particularly for items that have been added or removed from the list. The value of the key prop should be a unique identifier for that item in the list (be careful using the index argument in filter and maps callback function, since the index for each item keep changing depending on the number of items in the iterative).

Pure functions. `f(d)=v` . Props and nesting components in React.

Nested components and props

Props are to components what arguments are to functions. Props are the way in which a parent components passes data to its child component. When specifying a component in JSX (invoking a component), a property value can be passed in the component in a way that looks similar to an attribute="value" pair in HTML tags.

Building UIs with pure functions and function compositions in React

In react, components can be thought of as pure functions, often composed with other pure functions. In a pure function, we pass arguments and get a value in return.

In react, components are like pure functions that take arguments and return a UI view. A neat expression to summarise this would be `f(d) = V` . (where d is data and V is the view).

Composing function with helper functions is good practice to modularise code and similarly the same goes for react components. So much so, that React recently introduced stateless functional components which allow us to define components as normal JS functions that return JSX dom, these functions take the props as their argument (unlike the traditional component class with a render method).

Tyler goes on to say that learning React the right way is like an introduction to the world of functional programming where composing pure functions is the go to way to do something. Hence we should treat the props passed to a component as immutable.

React components can easily be composed of other components. What this means is that the JSX that is rendered by a component could be invoking other components while passing the prop types it need as property attributes.

Intro to PropTypes

PropTypes is a way to do type checking on props in React. It has recently been separated from React to its own library.

PropTypes allow us to declare the type of each prop being passed to a component. Then, if the prop passed isn't of the declared type, we get a warning in the console.

PropTypes can be string, func, bool, object (of a particular shape with each properties type specified), et cetera.

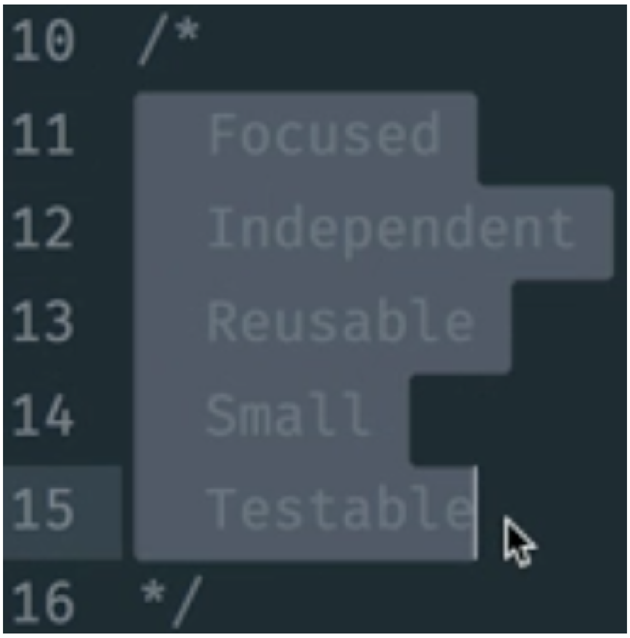
Video

The reliance on particular state is a very large contributor to a system's complexity. What this means is that when using a function with side effects, the system has to worry about and keep track of the state that it depends on and modifies, whereas with pure functions the need to keep track of particular state disappears. Hence simplifying the system.

Hence components are easy to think about, they're easy to test (because we don't have to asses the state before testing components), they're easy to reuse and they're easy to reproduce the same results. So we can conclude that: react components do not change any state outside they're scope and cause any mutations/ async requests, et cetera.

Important thing to note: The JSX dom representation that a react component returns should be wrapped in a single DOM container element (which can be any html element objects or a react component). In short, don't return multiple DOM or react components without wrapping them in a single container.

Any react component we want to make, should be:



```
/*
  Focused
  Independent
  Reusable
  Small
  Testable
*/
```

Quiz

The above purity means that we should treat props as immutable from within components (or their render method).

The `this` keyword + Managing and Updating State The this keyword

When invoking a method, the keyword this within its body refers to the object the method is a property of (the method will be invoked on an instance of that object, which will be on the left of the method invocation before the dot notation). In different situations the value of the this keyword is not very obvious and requires a thorough understanding of the different ways JavaScript binds the this keyword (which acts as context in functions/methods).

- **Implicit Binding**: object has a property which equals to a function and within that functions body one can refer to other properties of that object with this.dotNotation
- **Explicit Binding**: binding using functions' in-built functionality .call, .apply, .bind.
- **New/Window Binding**: when we use new to create an instance of an object, it binds the this keyword to itself within any of its methods/properties.

Managing & Updating State

We specify the state of a component within its constructor (which first calls super with the props as argument). The state has to be explicitly changed using the component's `setState` method which takes in a function as an argument; the function having to return the new `state` object.

We go on to create the popular tabs on the github-battle project and set its state. We defined a method that changes the components state when a particular item in the component was clicked. We link the component's method to that list item using the `onClick` attribute.

React advocates love how React needs users to explicitly change the state. When the state is changed the necessary changes reflect on the DOM immediately.

Stateless Functional Components

While using React, we're gonna have a lot of components that only have a render method and optionally take in some props (they have no other method or state). These components can be defined as pure functions instead, removing the unnecessary component class abstraction.

It is a great idea to separate container components from presentational components in React.

Stateless functional components will aid us in doing so. It is a good idea to use these because this is a good way to separate container and presentational components. The functional component takes props as its first argument.

Tyler encourages us to create a stateless functional component whenever we feel that the components render method is getting long enough to understand with a glance.

Stateless functional components should not be called functional components because in future, React will have functional components to which we can add state.

Private Components If a components render method is getting long, we can compose it out of multiple stateless functional components. If such a component is composing a container component and is present in the same module from which only the container component is being exported, it is called a private component (like private functions).

Another advantage of using stateless functional components is that the chances of calling the container component's methods in a wrong context reduce heavily, because of the absence of references to `this`.

Life Cycle Events + Ajax

React Life Cycle Events

In the beginning of the course Tyler explained that a component has certain life cycle events associated with it; the chart on the bottom illustrates each one's time of invocation with respect to the life-cycle of a component.

React's life cycle events can be broken into two categories:

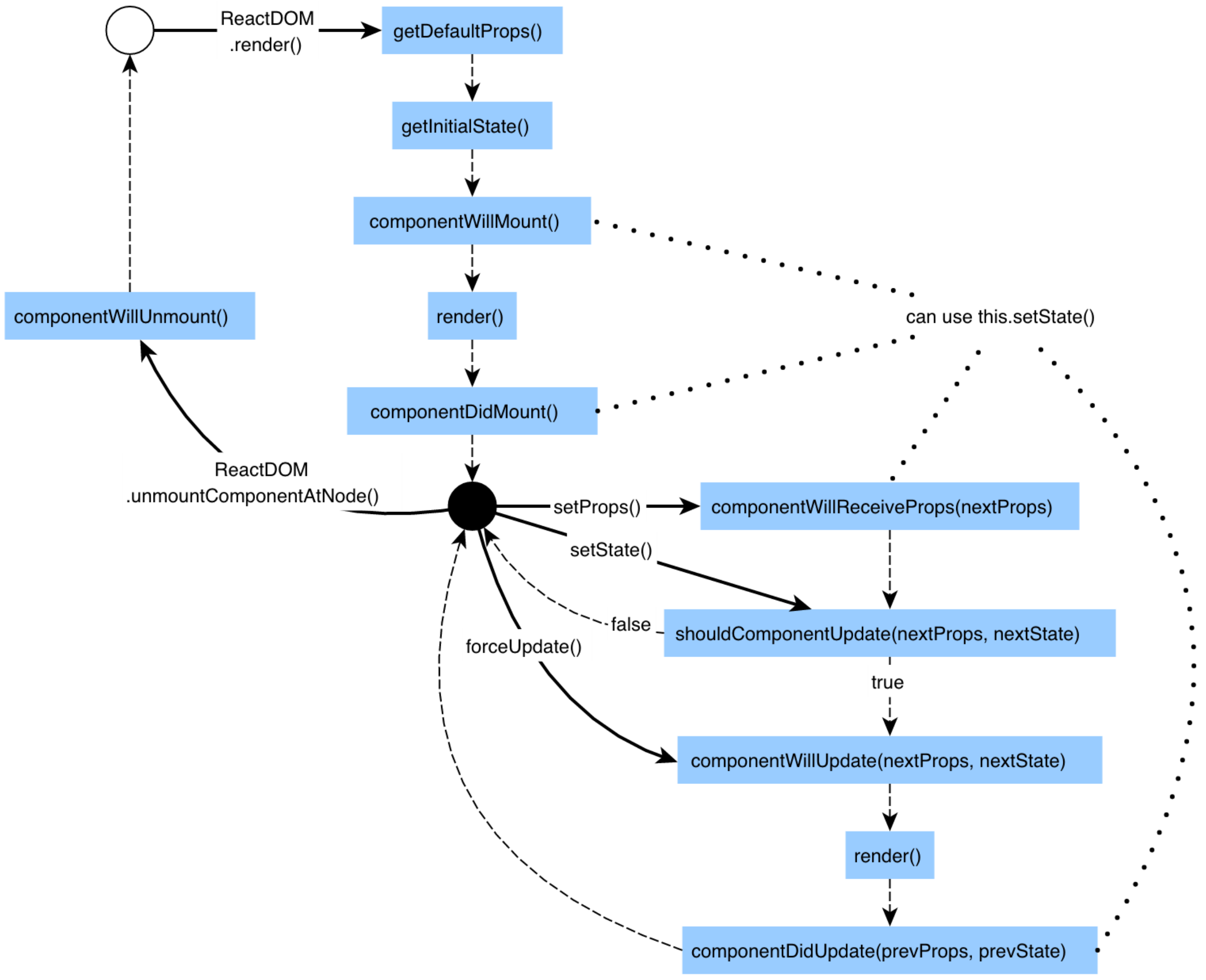
1. When a component gets mounted to the DOM or unmounted from it.
2. When a component receives new data.

For the first category, the component may need to perform some/all of the following tasks during that time in the life cycle:

- Default props - if props aren't given to a component, they default to the specified Component.defaultProps object.
- Initial State - initial state is retrieved from within the constructor of the component (like how we do it)
- Make an Ajax request to fetch some data - common use case, the request can be made in the body of componentDidMount method (which is self explanatory)
- Set up or remove any listeners or web-sockets (like a firebase ref listener) - start listener within componentDidMount and stop listener in componentWillUnmount (events are self-explanatory)

Now coming to the life cycle events that triggered when the component receives new data from its parent component.

- `componentWillReceiveProps` - for times when you want to execute some code whenever your component receives props
- `shouldComponentUpdate` - allows us to add another condition which should return true for react to re-render that component (and obviously its children)



Making Ajax Requests

An Http client library can be used to make Http requests to the server side (or any api we wish to use). *Axios* is a promise based Http library that we used. It requires the `encodedURI` as an argument and returns a promise.

In our web application, we need to make ajax requests when the popular component mounts and when its state changes. So we make some changing in the method `updateLanguage` , to add a `repos=null` property on the state object temporarily, till the axios promise returns its response—after which the state is updated again to `repos=repos` .

Loved the way Tyler structured the files and method names to make the Ajax request, abstracted the entire api calls in a module with contextual readable method names. Created a subdirectory `utils` in `App` , and in that a file named `Api.js` . We required axios within that file and exported an object with all necessary methods required for our app to interact with the api. This method was then invoked in the new implementation of the `updateLanguage` method of the Popular component.

React Router v4

React Router is a component based routing mechanism for React apps (and native). It is the most popular routing package for react and is built by the reacttraining.com team itself.

It's very simple to use. We used the following components present in the react-router-dom module:

- BrowserRouter as Router - is the component between which all the routes are nested. Our most parent component.
- Route - the component that helps specify views for each component, has many small nuances one needs to be aware of
- Link - just a link with prop to.
- NavLink - has activeClassName prop, to apply a css class when active
- Switch - used to make sure only one of the Routes is active at any moment (useful for showing a route without a path for paths that are not handled aka 404)
    - instead of rendering all of the routes that are active, switch is gonna let only 1 route be active at a time

Route

The main routing component which links url paths to components. We used the following props that Route accepts:

- path - prop takes string for the url path at which component should be active in the view
  - Note: if a route has path='/first', it's component will be active for all paths that start with /first
  - to avoid the above, we need to prepend prop path with keyword exact
- component - takes a javascript expression referring to the component that one wants to link to the path
- render - this prop is useful when you don't want to link the route to a component, but instead specify the JSX right there. This prop takes a value of a function that returns the JSX you wish this route to return. This is used mostly for the 404 pages I'm assuming, in a route that's the last child of the switch component (and has no path prop value obviously).
- exact prop makes sure the route is rendered/active only when the path matches exactly and not just partly

Link

This component is the basic anchor tag in react, except obviously it knows what component it originates in.

- to - prop takes the path to route to when the user clicks the link

NavLink

Composed of the component Link, except with additional functionality to make Links active when their path matches the current path.

- `activeClassName` - is the class that is applied to the NavLink component when the current path matches with the link's path
- `exact` - this prop makes sure the activeClassName is applied only when the path has an exact match with the link's path (not just partly, similar to the exact prop in routes)

`BrowserRouter` and `Switch` don't have much more than what's outlined in the brief points above.

Forms and Encapsulation in React

In this lesson we start-off using the React developer tools where we can see the tree like structure of our app, we can see the props passed to each component and the state living in each component.

Input fields

In a typical web application, what we're used to doing is to grab the value in the input field when a user fills it in and clicks submit. However, in React, we have two ways in which we could handle input fields in forms:

- Controlled Component
  - The controlled way is when we bind the value of the input field to the state of that component
  - So when the user types in the value, the state updates and then changes the value of the input field (counterintuitive but it is like that and not the other way around)
  - We can see the state change in real time as the user types in the React developer tool
  - This is the way we're going to be following in this course and is how the React docs typically recommend that we deal with forms
  - This is called a controlled component because React is controlling the value of the specific input field
- Uncontrolled Component
  - The uncontrolled way is a little more traditional, where the user fills the input field
  - and the state doesn't change till he presses submit (or a similar event)

Idea of Encapsulation

This idea is such that the form we're working with is a component which has child components for individual input fields.

- **From -> component, Input -> child controlled components**
- The input field components have their values bound to a specific property on their state
- As the value of this state updates, the state of the parent form component remains the same (sees no change)
- The parent component passes a function (like onSubmit handler) to these controlled components and the value of the parent's state updates only when this function is invoked
  - and the parent's state has values which are the result of passing the child components state through that function
  - this is a little hard to understand without context, but I'm on the right page So we can encapsulate the complexity of a specific state in a child component and pass in a function which helps its parent grab the value at a certain event

invocation.

Dynamic Rendering & Query Parameters

Dynamic Rendering

This seems to be very natural in React. We know that state changes are reflected in the view immediately (obviously when there is a change). So in the render method of a component—if we return certain a component/html element conditionally, based on the value of a property of the state, the element/component will be rendered dynamically based on whether that property of the state has a value or not. The same conditional rendering can be used to make elements/components disappear, replace each other, show up dynamically.

In our example project, we use this for the following

- make the PlayerInput disappear
- Get the PlayerPreview to show up
- Get the Battle results link to show up

Query Params

The `to` prop of the `Link` component of React Router accepts url path strings or an object with props:

- pathname: which takes a string for the link's path
- search: takes a string beginning with '?' for query params followed by the query param name value string

React Router's Route component passes a few props to the component it's linked with. One of the prop is `match` which is an object with a property `url` which contains the current url's path. The `to` value of a `Link` that routes to a sub-path of the current url should be composed of the `this.props.match.url` property, so that the path can be changed later without affecting the link.

Axios, Promises and the Github Api

Promises, is a helpful JS library (not a part of js) that helps us write easy to digest async code.

If a certain function returns a `promise` instead of a *value*, the function invocation needs to be chained with a method `.then` which takes a function as its argument; this function is given the `data` that the promise ultimately returns as an argument and returns desired values that can be derived from that `data` .

The axios library is a promise based http client. We use its method `.get` to request the GitHub api for data and chain it with `.then` to return the data in the format desired. Axios also has a useful method .all that returns a promise which is an aggregation of an array of promises which it takes as an argument.

Any function that returns an invocation of `axios.get` (chained with a `.then` method) is returning a promise and when that function is invoked it needs to be chained with a `.then` invocation to be able to receive the value axios intended to return.

The function composition used to interact with the GitHub api and finally expose a single simple method in api object is a great way to compose functions. Get inspired for your own api objects you compose in the future.

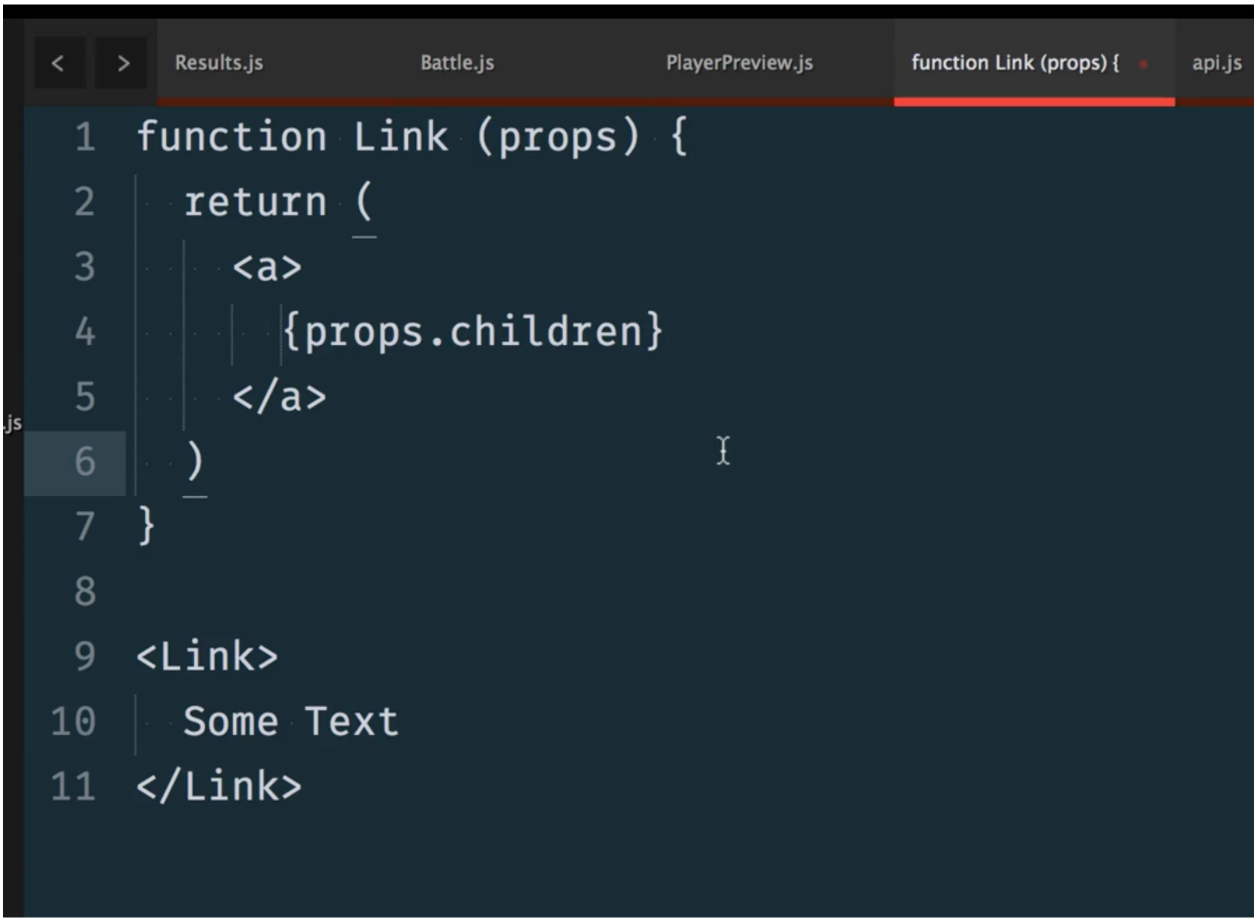
Composition vs. this.props.children

`this.props.children` in React

A lotta times in while working with HTML pages, we need access to the specific data between the opening and closing tag of an element. React gives us an easy way to do this for components. `this.props.children` will give us the specific data between the opening and closing tag of the component referred to here by `this` .

In case of there being more components inside the opening and closing tag of a component, `this.props.children` will return an array composed of the components contained within the component's opening and closing tag.

So simply put, props.children is whatever is between the `<Opening>` and closing `</Opening>` blocks of a component.



```
function Link (props) {
  return (
    <a>
      {props.children}
    </a>
  )
}

<Link>
  Some Text
</Link>
```

This picture simply explains how this is useful to us while creating our components in JSX. So components that have an opening and closing tag—include `props.children` in the ui they return to keep a section of their ui dynamic. So while creating the component, we can pass in any relevant ui as its child.

Building a Highly Reusable React Component

Defaulting props with defaultProps

An important advantage of using the React library is reusable components that can even be used between different projects. Being able to specify `defaultProps` for a component makes designing reusable components a lot simpler. It enables different users to use the component as it is or with customized `props` suited to their use within their project.

We make the `<Loading />` component more reusable by specifying `defaultProps` for it so that different users can either use it as it is or by changing `props` as needed.

Video

The `Loading` component has two `props` : `text` and `speed` for which we provide default values and specify prop types. Users of the `Loading` component can either default on the text "Loading" with dots added to it every 300 miliseconds or change the text and the speed how they like.

For the dot animation effect, we updated the component's state continuosly between an interval specified by the `speed` prop; this happens in the life-cycle event `componentDidMount` . If we don't clear this interval when the `componentWillUnmount` the state of this component will keep updating till the app's instance shuts down. So the interval is referenced to a property `interval` of the component which is then used to clear the interval when the component unmounts.

Building for Production

When building our project for production, we're gonna need the JavaScript in `index_bundle.js` to be minified and we're gonna need React to be compiled for production (without warning and unnecessary console logs) rather than development.

In order to achieve this, we modify the webpack configuration object exported from `webpack.config.js` . * We conditionally add a couple items to the `plugins` property of the `config` object if the process' `NODE_ENV` property says `'production'` . * 1st item: we define a `new` plugin using `webpack.DefinePlugin` which makes the `NODE_ENV` property of `process.env` equal to `'production'` . **This `NODE_ENV='production'` is for the code that webpack is compiling.** Whereas, the one in the condition, was for the code that compiles our project (i.e. the code in `webpack.config.js`). * 2nd item: an instance of `webpack.optimize.UglifyJsPlugin()` for webpack to minify and uglify all the JavaScript in `index_bundle.js` . This makes it impossible for other devs to imitate our application using dev tools.

Lastly, we add a script `"build"` to the npm `"scripts"` in `package.json` ; it says: `NODE_ENV='production' webpack -p`

Now when we run `npm run build` from our project directory, webpack will build our project in the `dist` subdirectory of our project. The `NODE_ENV='production'` is essential to pass the condition specified in `webpack.config.js` to add production plugins to the `config` object. `webpack -p` simply tells webpack to build for production.

Installing modules locally rather than globally

When installing `firebase-tools` , Tyler gives an awesome suggestion to avoid installing npm modules globally and to refer their commands from within scripts in `package.json` instead. Since when the commands run from there, they refer to the locally installed node modules and don't require us to install them globally. Another example of this is how we use webpack in this project. If we want to refer to webpack from our command line, npm will require us to install it globally.

There's nothing wrong with installling modules globally, except that our project will be forced to use the most updated version of that module. Which will create problems when incompatible updates are made to the module.