


## Session 11 – Notes – React Props and State

### Using Components and Props

Let's start with Scoreboard\_Header

 scoreboard\_Header

You can see a simple Use of a Component

```
function Header() {
  return (
    <header>
      <h1>Scoreboard</h1>
      <span className="stats">Players: 1</span>
    </header>
  );
}

ReactDOM.render(
  <Header />,
  document.getElementById('root')
);
```

You can use arrow function too:

```
const Header= () => {
  return (
    <header>
      :
    )
}
```

Also, if you only return JSX you can use inclusive return by deleting the return keyword:

```
const Header= () => (
  <header>
    :
  )
```


Or even we can remove parenthesis as they were optional in the first place anyway:

```
const Header= () =>
  <header>
    :
```

You can choose any style as you wish.

Now let's create the Player component:

Please see:

 scoreboard\_Player


```
const Player = () => {
  return (
    <div className="player">
      <span className="player-name">
        Guil
      </span>

      <div className="counter">
        <button className="counter-action decrement"> - </button>
        <span className="counter-score">35</span>
        <button className="counter-action increment"> + </button>
      </div>
    </div>
  );
}

ReactDOM.render(
  <Player />,
  document.getElementById('root')
);
```

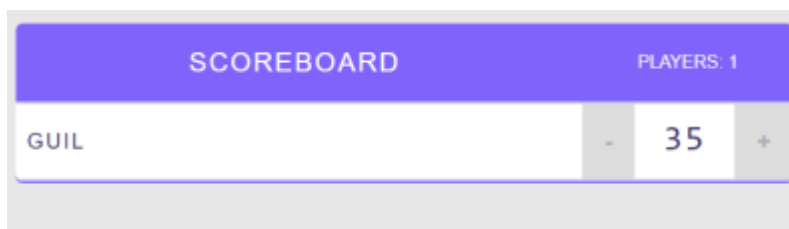
### Quick Challenge:

- 1- Create a new Component for Counter,  
Separate it from Header,  
Then add the counter component to the header,
- 2- Then Add the Header with the Player as components into the app.
- 3- think about its pros and cons.

You will see the answer in  scoreboard\_Header\_Player\_Answer

Google React Developer Tools is a nice extension that can help to see your components in React. It is a useful tool, as React doesn't change DOM directly and we need to see the virtual DOM to check our components.

The result of the last app is:



However, it is static. Talking about components being formulas or classes that return JSX, we like to see them act like formulas or classes, meaning we'd like to pass data and manipulate properties.

## Setting and Using Props

Up to this point, we have created and nested components, and now we are going to learn how to have data passed and shared between components.

Every React component and element can receive a list of attributes called properties or props. Props (short for properties), are the most basic and common way to pass data between components. React elements are technically objects and have properties attached to them.

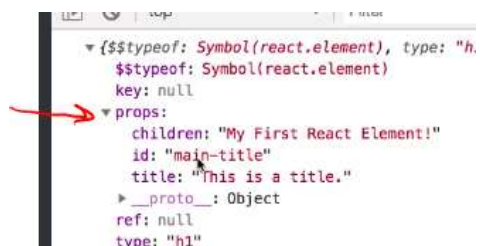
In React, we use properties, or props, to customize our components and pass dynamic information into them, specially attributes of HTML elements. Like an element's attributes, every React component and element can receive a list of attributes just like HTML elements.

Imagine if we can modify our components with a set of arguments. For example see below:

```
<Component propName="propValue" />
```

Obviously, we will have to be able to work with these values in the function/classes that declare the components. Pay attention that in React the functions and values that defines a component are passed to its children too.

Every React element and component has a props object containing the list of props given to it. Let's look at a React's object:



We can manipulate props.

Let's open [1 - Setting and Using Props](#)

We will pass some data from the `<Header />` component ... assuming some imaginary data, we should have something like this:

```
<Header {pass some data} />
```

However, in practice the following is the way we identify the data we want to pass.

```
<Header title="Scoreboard" totalPlayers={1} />
```

In effect, we can add as much data, and they all pass through `(props)` in the function.

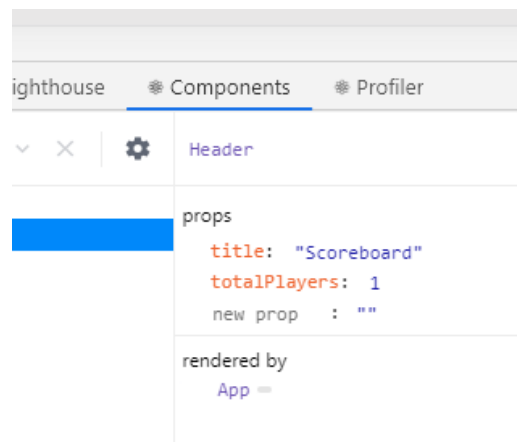
See below, there are two parts to look at:

```
32 const App = () => {
33   return (
34     <div className="scoreboard">
35       <Header
36         title="Scoreboard"
37         totalPlayers={1}
38       />
39
40       {/* Players list */}
41       <Player />
42     </div>
43   );
44 }
```

```
board_final > JS app.js > ...
1 const Header = (props) => {
2   return (
3     <header>
4       <h1>{ props.title }</h1>
5       <span className="stats">Players: { props.totalPlayers }</span>
6     </header>
7   );
8 }
```

Take a look at the final version and go live.

We can see Header has props in React Dev Tools



Let's change the title: and totalPlayers, and see the effect on the page. Also, pay attention that `props.title`

is javascript, and that is why we use curly braces. In fact, you can also add any javascript objects to the braces too. Let's try:

```
<h1>{ props.title }</h1>
<span className="stats">Players: { props.totalPlayers * 3 }</span>
```

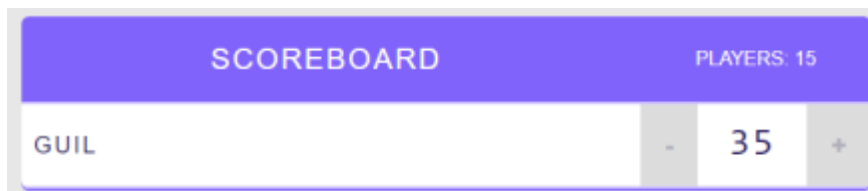
We can even have a call back function in the function Header(), so we put a function inside the component <Header />

Try:

```
<Header
  title="Scoreboard"
  totalPlayers={n => n+10}
/>
```

And

```
<header>
  <h1>{ props.title }</h1>
  <span className="stats">Players: { props.totalPlayers(5) }</span>
</header>
);
```



## ONE WAY DATA FLOW THROUGH COMPONENTS

Historically, JavaScript applications have had a two way flow of data. This makes more sense with an MVC or MV\* architecture where different views or controllers had to have a live record of data in other views or controllers.

React has a different model. React does not have two way data binding. It has a one way flow of data.

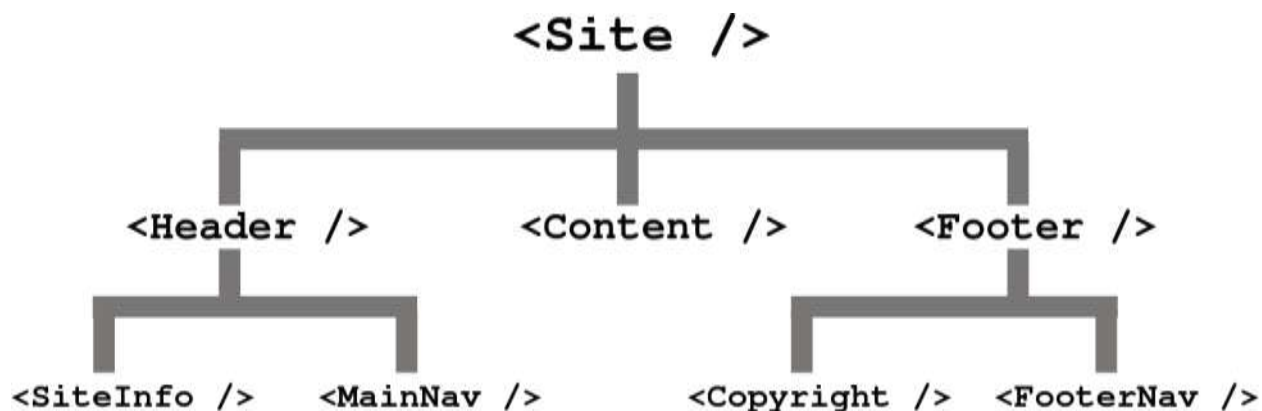
Data in React apps only flows down through an app, from parent components to children components. If we have worked with two way data binding or MVC architecture in the past, this can take a while to get used to. However, the React one way data flow model is actually beautifully elegant and removes much of the complexity and places where something can go wrong with two way data binding.

What this means at a practical level is that once data is passed down from a parent to a child, that prop value should not be changed. This reinforces the practice of immutability for all data passed through our app as props. If we need to change prop values, there is a way to do it using something called State that we will look at in an upcoming chapter.

## SETTING PROPS AT THE HIGHEST COMPONENT LEVEL NECESSARY

Since we cannot pass data up the component hierarchy, from children to parent components, we want to start passing props at a level in the component hierarchy where that data will reach all necessary children components that need the data.

Let's take a look at a simple site built using React as an example:



Let's imagine now that the `<SiteInfo />` component and the `<Copyright />` component both need to have access to the site name.

To get the site name, we might write a function called `getSiteName()` that would pull the site name from wherever it is stored. However, the question arises of where should we write that function?

We could write the function twice, once in the `<SiteInfo />` component and again in the `<Copyright />` component, but obviously duplicating our code is a bad idea. We could also write a helper library and import that into our React code and call the function in each of the `<SiteInfo />` and `<Copyright />` components. While possible, this is not the normal practice in React, as we are still duplicating that function call. Plus, in most cases, we want to keep all our necessary functions within actual components, not in separate libraries for easier tracking and troubleshooting.

So the most common practice would be to write the `getSiteName()` function inside of the `<Site />` component. Then we want to pass the data from that component down through the `<Header />` and `<Footer />` components as props. Finally, we want to pass it into the `<SiteInfo />` and `<Copyright />` components. As we can see, the `<Site />` component is the first parent component that shares both `<SiteInfo />` and `<Copyright />` as children.

Here is a compatible code:

```

// Setup the main site component
const Site = () => {
  function getSiteName() {
    // Do something to get siteName return siteName;
  }
  return (
    <Fragment>

```

```

    <Header siteName={getSiteName()} />
    <Content />
    <Footer siteName={getSiteName()} />
  </Fragment>
);
};

// Display the Header component
const Header = (props) => {
  return (
    <header>
      <SiteInfo siteName={props.siteName} />
      <MainNav />
    </header>
  );
};

// Display the SiteInfo component
const SiteInfo = (props) => {
  return <div className="site-info">{props.siteName}</div>;
};

```

Another important detail to remember about props is that they are "read only" (or immutable), which means that a component can only read the props given to it, never change them. The (parent) component higher in the tree owns and controls the property values.

For example, if you try to set or change a prop value like this:

```

const Header = (props) => {
  return (
    <header>
      <h1>{ props.title = 'Fun Board' }</h1>
    </header>
  );
}

```

React will throw the error: // Cannot assign to read only property 'title' of object.

When a component has more than one prop, you'll often see them written on separate lines and indented, like so:

```

<Header
  title="My Scoreboard"
  totalPlayers={5}
  isFun={true}
/>;

```

You can omit the value of a prop when it's explicitly true:

```

<Header
  title="My Scoreboard"
  totalPlayers={5}
  isFun
/>;

```

## Use Props to Create Reusable Components

You learned that Props pass data from a parent component down to a child component. For instance, in the app component, below, we're passing data to the header component via the title and total Players props. Which are then consumed by header and displayed as content. We still need to write the props for the player and counter components. The player component should display a player's name and the counter component needs to display their score alongside the buttons.

```

32 const App = () => {
33   return (
34     <div className="scoreboard">
35       <Header
36         title="Scoreboard"
37         totalPlayers={1}
38       />
39
40       { /* Players list */ }
41       <Player />
42     </div>
43   );
44 }
45
46 ReactDOM.render(
47   <App />,

```

```

1  const Header = (props) => {
2    return (
3      <header>
4        <h1>{ props.title }</h1>
5        <span className="stats">Players: { props.totalPlayers }</span>
6      </header>
7    );
8  }
9
10 const Player = () => {
11   return (
12     <div className="player">
13       <span className="player-name">
14         Guil
15       </span>

```

Now, assume, the player component should get two props. One for the name and another for the score. And player passes the counter component, a prop for the score. Since player is the parent of counter, it's going to define the props for both a player's name and score. So in the app component, we pass two props to the player component and call them name and score. Set name to the string, Guil, and Score to the number 50.

```

40   { /* Players list */ }
41   <Player name="Guil" score={50} />
42 </div>
43 );

```

The Player component need to use these props. So, pass the player function. The parameter props and replace the text in the span with curly braces and props.name.



```

10 const Player = (props) => {
11   return (
12     <div className="player">
13       <span className="player-name">
14         {props.name}
15       </span>
16
17       <Counter score={props.score} />
18     </div>
19   );
20 }

```

Well the same way we passed props to the header in player components, we pass to the Counter just one more level deep.

```
<Counter score={props.score} />
```

We see that counter has a score prop with a value of 50. So now use the prop inside counter by passing the counter function, the parameter props and replacing this static score with curly braces and props.score.

```

22 const Counter = (props) => {
23   return (
24     <div className="counter">
25       <button className="counter-action decrement"> - </button>
26       <span className="counter-score">{ props.score }</span>
27       <button className="counter-action increment"> + </button>
28     </div>
29   );
30 }

```

Now we can use multiple player tags inside the app component to add players and scores to the scoreboard.

The full code is in `scoreboard_Components`

We are just passing our components static values as props. Normally this information could be coming in dynamically from a database or API for example. Or passed down from the main application state.

## Iterating and Rendering with map()

For this part, let's check the code in:

`scoreboard_Looping_map`

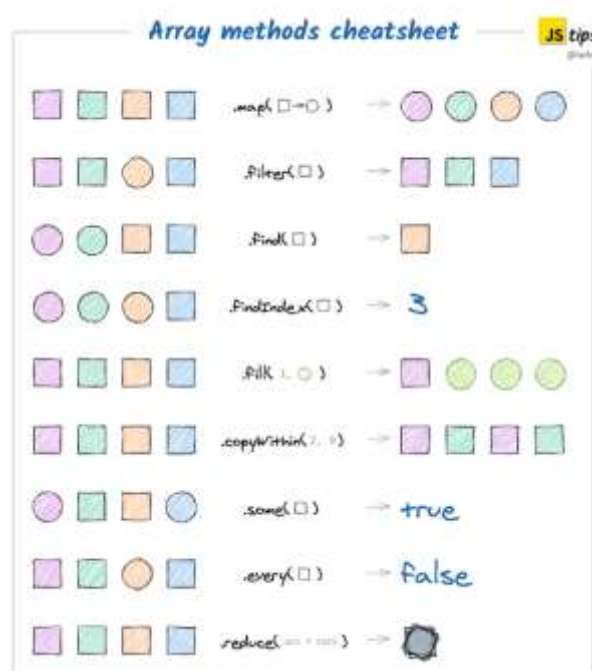
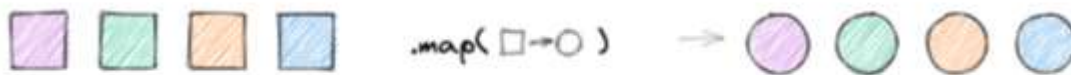
The most important part is illustrated below:

```

51  const App = (props) => {
52    return (
53      <div className="scoreboard">
54        <Header
55          title="Scoreboard"
56          totalPlayers={props.initialPlayers.length}
57        />
58
59        {/* Players list */}
60        {props.initialPlayers.map( player =>
61          <Player
62            name={player.name}
63            score={player.score}
64          />
65        )}
66      </div>
67    );
68  }
69
70  ReactDOM.render(
71    <App initialPlayers={players} />,
72    document.getElementById('root')
73  );

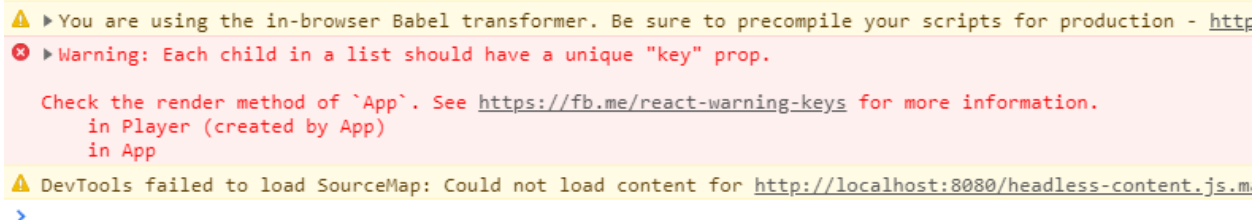
```

To understand array functions, see the following illustration:



In most apps, you will have a list of sibling elements that get updated, removed, added and reordered on the page. For example, items in a to-do list app, a list of guests in an RSVP app, or in our case, a list of players. React manages what gets rendered to the DOM and displayed on the screen. In order for this process to be fast and efficient, React needs a way to quickly know which items were changed, added, or removed. For this, React gives elements a special built-in prop named key.


Without key, React gives a warning, we can check in Console.



The screenshot shows a browser console with several messages. The first is a yellow warning: 'You are using the in-browser Babel transformer. Be sure to precompile your scripts for production - http://babeljs.io/docs/en/next/external-transformer.html'. The second is a red warning: 'Warning: Each child in a list should have a unique "key" prop. Check the render method of `App`. See https://fb.me/react-warning-keys for more information. in Player (created by App) in App'. The third is a yellow error: 'DevTools failed to load SourceMap: Could not load content for http://localhost:8080/headless-content.js.map'. A blue arrow cursor points to the bottom of the console.

A key is a unique identifier that gives React a way to quickly and reliably identify an element in the list.

For the use of keys, check:

 `scoreboard_Use Keys`

You need to specify a key prop on each player component to keep track of it and differentiate each component from its sibling. In the map function where we create players, we pass player a new prop named Key, has the value pass it an ID with player.id. React docs recommend that we use a string as the key value, so to convert our ID into a string, using `toString()` method on `player.id`.

You're not always going to create keys manually like this. If your data comes from an API, for example, there usually will be a unique ID for each item that you could use for key values. Since our data comes from the static list we're creating the ID property manually. Not all React elements need a key prop. Pass a key prop when creating elements by iterating over an array of items that will be rearranged, added or deleted in your UI. The key will help React identify which items were changed, added or removed from the DOM.

The player's display never changes because we're passing a static list of players to our app, and the counter doesn't change a player's score. In order to add dynamic behavior and interactions, our app needs data that can change.

A web application's UI is normally a function of the data you put into it. In other words, changes in your data result in changes to the UI, and when changes are made to the UI your data changes in response. State is used to store information about a component that can change over time.

## Understanding State

There are two ways that data gets handled than react, props and state. So far we built our application using functions that take props and return react elements. But props are read-only or immutable and we may sometimes need to change our components dynamically. That is where State comes in. In react, state is the data you want to track in your app.

So far, our player app didn't have functionalities, but if we could change score by clicking on plus/minus signs or add new players or remove some then we need state to store the data.

State is only available to components that are class components. So in the next video, we'll begin by converting a function component to a class component.

As discussed before, we can create components using Class instead of functions. Class components offer a more powerful way to create components, because you can use state with class as well. So, let's see how Class works.

Let's start by converting Counter function to a class.

```

45  const Counter = (props) => {
46    return (
47      <div className="counter">
48        <button className="counter-action decrement"> - </button>
49        <span className="counter-score">{ props.score }</span>
50        <button className="counter-action increment"> + </button>
51      </div>
52    );
53  }

```

We only need to add the Class declaration and add only a render() method and adding this to the props.score.

```

45  class Counter extends React.Component {
46    render() {
47      return (
48        <div className="counter">
49          <button className="counter-action decrement"> - </button>
50          <span className="counter-score">{ this.props.score }</span>
51          <button className="counter-action increment"> + </button>
52        </div>
53      );
54    }
55  }

```

In this case, because every time a button is clicked score changes, then “score” is our “state”. So we need to add a constructor as shown below:

```

45 class Counter extends React.Component {
46
47   constructor() {
48     super()
49     this.state = {
50       score: 0
51     };
52   }
53
54   render() {
55     return (
56       <div className="counter">

```

Now we can remove score prompt from “Counter” component as we have access to it through our Class.

```

    </span>
    <Counter score={props.score} />
  </div>
);

```

Changed to ||  
V

```

38   </span>
39
40   <Counter />
41 </div>
42 );

```

We can also delete the score from the player list:

```

73   { /* Players list */ }
74   { props.initialPlayers.map( player =>
75     <Player
76       name={player.name}
77       score={player.score}
78       key={player.id.toString()}
79     />
80   ) }

```

```

73   { /* Players list */ }
74   { props.initialPlayers.map( player =>
75     <Player
76       name={player.name}
77       key={player.id.toString()}
78     />
79   ) }

```

and we access the score through state so we change “this.props.score” to “this.state.score”

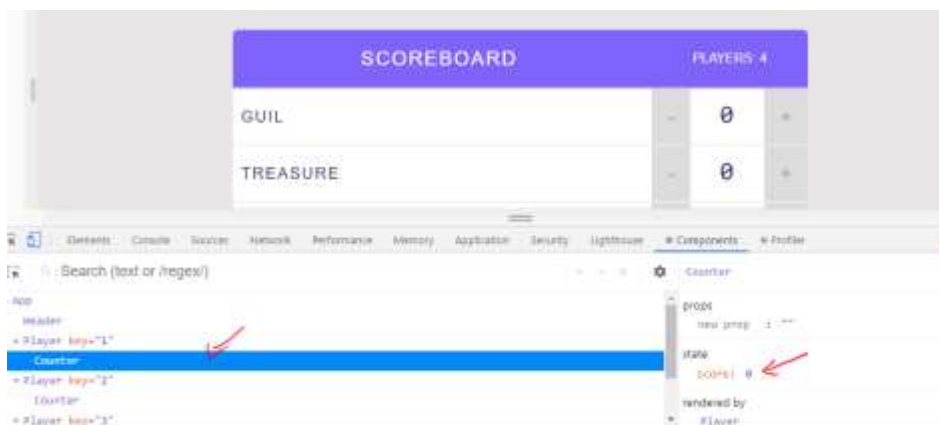
```
54   render() {
55     return (
56       <div className="counter">
57         <button className="counter-action decrement"> - </button>
58         <span className="counter-score">{ this.props.score }</span>
59         <button className="counter-action increment"> + </button>
60       </div>

```

```
53   render() {
54     return (
55       <div className="counter">
56         <button className="counter-action decrement"> - </button>
57         <span className="counter-score">{ this.state.score }</span>
58         <button className="counter-action increment"> + </button>
59       </div>
60     );

```

Now everytime props or state changes, render () will pass it to state so it is updated. We can check state in the RDT.



To simplify, we can replace constructor with a Class property for State as below:

```
45   class Counter extends React.Component {
46     state = {
47       score: 0
48     };
49   }
50
51   render() {
52     return (JSX attribute) className: string
53     <div className="counter">


```

Now, in order to manipulate score, we create a function (Method in Class) and link it to an even handler to the plus/minus signs. Let's call the method “incrementScore” .

```
58 incrementScore = () => {  
59   this.setState({  
60     score: this.state.score + 1  
61   });  
62 }  
63  
64 render() {  
65   return (  
66     <div className="counter">  
67       <button className="counter-action decrement">-</button>  
68       <span className="counter-score">{ this.state.score }</span>  
69       <button className="counter-action increment" onClick={this.incrementScore}>+</button>  
70     </div>  
71   );  
72 }
```

Pay attention, we need to have an arrow function in the incrementScore and also we define the event handler through React core handles (onClick).

Check

 scoreboard\_Handling Events

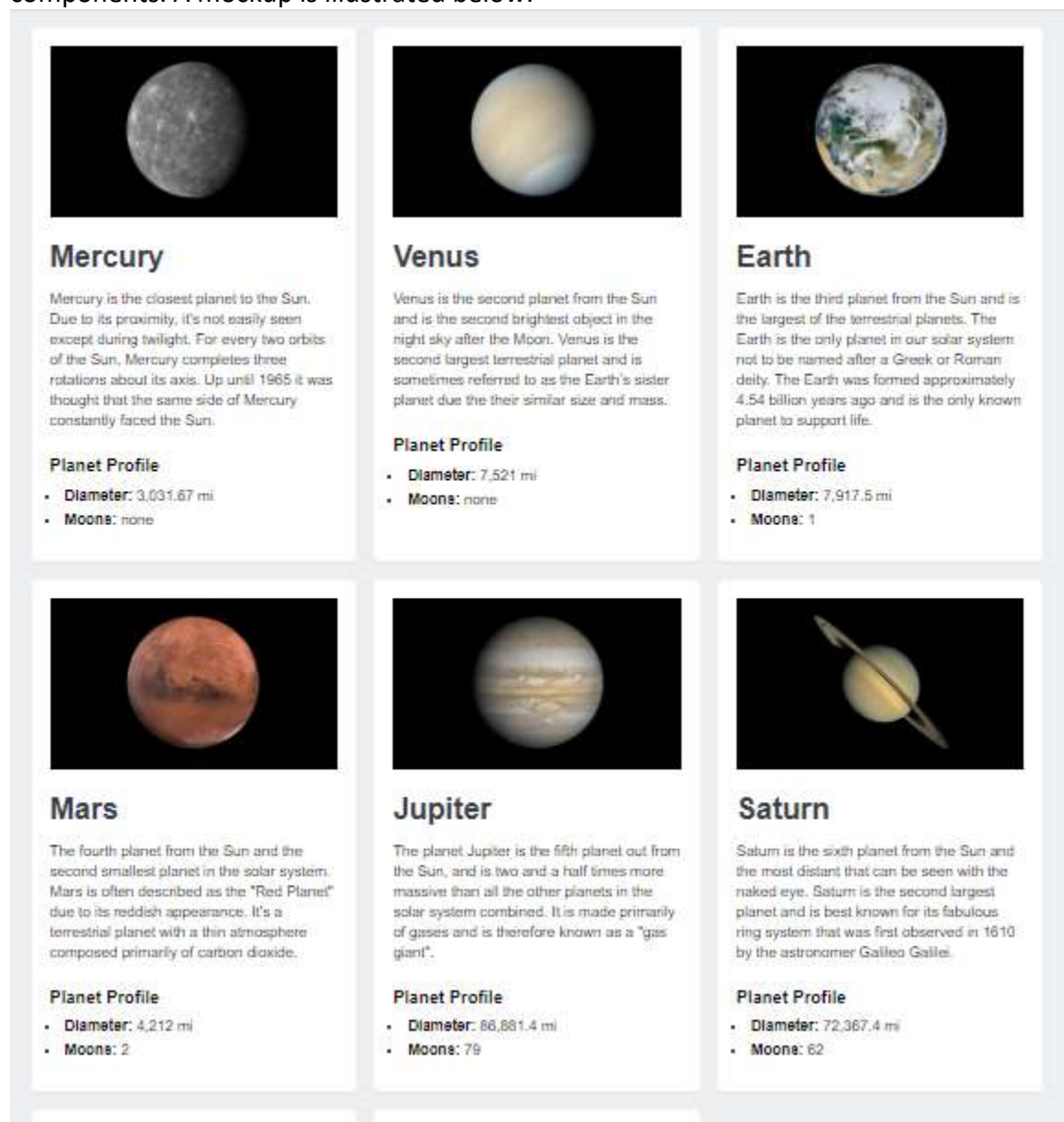
### Class Challenge:

Your task is to modify the code to add decrement function for the minus sign.




## Class Lab (10 marks) Due by This Saturday Night

In this Class Exercise you're going to sharpen your React skills by practicing JSX, creating and rendering components, passing down props, as well as iterating over data and more. You are going to use React to build a Planet Card Site by breaking it up into smaller components. A mockup is illustrated below:



To get started launch the download the project files from:

 **Class Lab - Building Planet Card - start**

HTML, CSS, images and a JavaScript file for this exercise is in the folder. In index.html, links to React, and ReactDOM, and the Babel transpiler are provided at the bottom of the file.

You're going to write your React code inside the file app.js.



**Your Tasks:**

App.js currently contains an array of objects assigned to the constant `planets`. Each object has properties that describe a planet, like name, diameter, moons, description. And it has a URL property that points to an image located in the image folder. In `index.html`, in the comments you can see an example of the markup you'll need to use to create a planet card.

Just below the planet array in `app.js`, you will create two components, a Planet component that renders a planet card, and a container component, that iterates over the planet's array and renders a planet component for each object in the array.

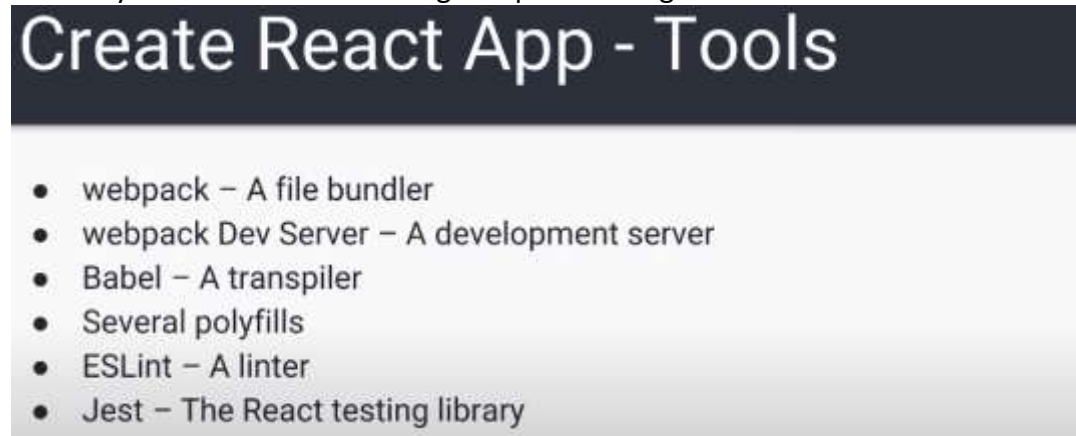
You will need to pass the planet's data to the main container, then pass that data down to the planet card using props. You should use the commented markup in `index.html` as a reference for how to display the data.

The only text that should not be dynamic is the `h3` with the text, Planet Profile, and the text between the `strong` tags. Everything else needs to be displayed with props.

Finally, to display the planet cards, you will need to render the main container component to the DOM.

## Create React App

Create-react-app is a tool built by developers at Facebook to help you build React applications. It saves you from time-consuming setup and configuration.



Go to your folder and open terminal (Node Terminal) and enter:

➤ `npm install -g create-react-app`

when installed then you start with

➤ `create-react-app [project Name]`

*(or alternatively, instead of the above steps: > `npx create-react-app [project Name]`)*

This takes some time. Once complete, you see a success message:

```
Success! Created firstapp at D:\junk\junk react code\firstapp
Inside that directory, you can run several commands:
```

```
npm start
```

```
Starts the development server.
```

```
npm run build
```

```
Bundles the app into static files for production.
```

```
npm test
```

```
Starts the test runner.
```

```
npm run eject
```

```
Removes this tool and copies build dependencies, configuration files
and scripts into the app directory. If you do this, you can't go back!
```

Go to the created folder for your app and enter: `npm start`

This automatically, start your development server on localhost:3000

Index.js imports and enables a service worker created for you here in the file registerServiceWorker.js. This file caches your static assets and serves them from local cache in production. Ensuring that your web app is reliably fast, even on a slower unreliable network. Another key feature in progressive web apps is the web app manifest. The manifest is a JSON file containing metadata associated with your app, like the app's name, author, description, and more.

To customize our own app, we mainly manipulate App.js. This is what we need to keep:

```

Project
└─ search-app
   └─ node_modules
   └─ public
   └─ src
      └─ App.js
      └─ App.test.js
      └─ index.css
      └─ index.js
      └─ registerServiceWorker.js
      └─ .gitignore
      └─ package-lock.json
      └─ package.json
      └─ README.md

App.js
1  import React, { Component } from 'react';
2
3  class App extends Component {
4    render() {
5      return (
6        <div>
7
8        </div>
9      );
10   }
11 }
12
13 export default App;
14

```

With the above changes, you can delete app.css and logo.svg also.

Below is some helpful hints for using C-R-A

## USING CREATE REACT APP ON AN EXISTING PROJECT

So far we have been approaching Create React App as if we were starting a React project from scratch. However, there are instances where we want to fork or work on someone else's project who started it with Create React App.

If this is the case, we'll know they are using Create React App in one of two ways. First, they will hopefully tell us in the README.md file of the project. Second, we can open the package.json file and see if they have a setup like the following:

```

"scripts": {
  "start": "react-scripts
start", "build": "react-
scripts build",
  "test": "react-scripts test --
env=jsdom", "eject": "react-scripts
eject"
}

```

If the following commands exist in the package.json, it is more than likely they are using

Create React App. So everything we have said so far applies, and everything we will continue to cover in this chapter applies.

However, before we get rolling, we have to run `npm install`. This will pull down all the dependencies we need to our `node_modules` folder. Since Create React App does this for us when we first create a project with it, we will have to do this step manually when working with a project that has already been created.

## RUNNING TESTS WITH CREATE REACT APP

Testing with JavaScript and React is a little outside the scope of this book. However, Create React App does ship with the Jest testing library, the preferred testing library for React apps.

If we ever need to run any of our tests, `npm run test` will kick off that process. We are not going to explore those options in this book, but once you are comfortable with testing, it should all make sense and be a helpful integration.

## EJECTING FROM CREATE REACT APP

As we have seen, Create React App offers a number of built in tools behind the scenes. With some projects, we may want to customize the configuration files for the built in tools.

When this happens for a project, we will need to run a one time `npm run eject` command. This command migrates all of the configuration files from being hidden and makes them all available to edit.

We cannot undo the `npm run eject` process.

The only reason to run `eject` is if we know the tool we want to customize and feel comfortable making and maintaining changes to it along with the other tools. We are not going to look into running `eject` in this book, as it will not be necessary for our needs, and for likely most of our React projects.

Remember, we do not have to eject from Create React App to build or launch our app, just if we want to customize the underlying tools settings in a way that is not possible.

## HOW YOU WILL LIKELY USE CREATE REACT APP

In most cases when we start building a new React project, we will run `npx create-react-app project-name` to kick off the project.


After that first time though, `npm start` will be the command we run from the project directory to start watching our files for changes, and for starting up the development server.

When our project reaches points where we are ready to ship to staging or production, run `npm run build`, and send the `build` directory where it needs to go.

If testing is part of our workflow, we will likely run `npm run test` regularly. Calling `npm run eject`, however, should rarely be necessary and a command that means the end to working with Create React App on that project.

## C-R-A Challenge:

The exercises for this challenge are in

 [Class Challenge -create-react-app](#)

The practices are as follows:

### ## PRACTICE WITH CREATE REACT APP

Now that we have a basic understanding of what Create React Does and how to use it, let's do some practice.

For these practice exercises you will need a generic practice folder. Then for each exercise you will spin up a new Create React App instance from that practice folder.

#### PRACTICE EXERCISE #1

The exercise below will help you establish comfort setting up a new project with Create React App and moving into it with the command line.

Inside of a practice folder, call `npx create-react-app exercise-1` from the command line. Then navigate into the new folder using `cd exercise-1` and run `ls`.

You should see the list of default React files outlined in the section, "Setting Up Create React App," from the last chapter.

#### PRACTICE EXERCISE #2

This next exercise will help you gain confidence starting the Create React App development server and seeing the changes to your code reflected in the browser.

Inside of a practice folder, call `npx create-react-app exercise-2` from the command line.

Open up the `exercise-2` directory in your code editor.

Run the command `npm start` from inside the `exercise-2` directory.

Open the URL it gives you for the development server in your browser. To stop the development server, type `Ctrl + C` in the command line.

Then, in your code editor, change the text of the `p` tag in the `/src/App.js` file from “Edit `src/App.js` and save to reload” to something else. On save, you should see the browser refresh with your new value.

### PRACTICE EXERCISE #3

This practice exercise will help reinforce the skill of setting up React apps. It will also get you comfortable adding new component files to an app and having them show up working in the browser.

Inside of a practice folder, call `npx create-react-app exercise-3` from the command line. Open up the `exercise-3` directory in your code editor.

Run the command `npm start` from inside the `exercise-3` directory.

Open the URL it gives you for the development server in your browser. To stop the development server, type `Ctrl + C` in the command line.

Then, in your `src` directory, add a new file named `Hello.js` with the following code:

```
import React, { Component } from
"react"; class Hello extends
Component {
  render() {
    return <p className="Hello">Hello!</p>;
  }
}
export default Hello;
```

Then open your `src/index.js` file and change the references to `App` on line 4 and `<App />` in line 7 to the following:

```
import React from 'react';
import ReactDOM from
'react-dom'; import
'./index.css';
import Hello from './Hello';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <Hello />,
  document.getElementById(
    'root')
);
```

When you save the changes to the `src/index.js` file, you should see the changes reflected in the browser.

#### PRACTICE EXERCISE #4

After you complete this fourth exercise, you should feel more comfortable running the build process for getting your app ready for production.

Inside of a practice folder, call `npx create-react-app exercise-4` from the command line. Open up the exercise-4 directory in your code editor.

Then change the text in the `p` tag in the `/src/App.js` file from “Edit `<code>src/App.js</code>` and save to reload” to something else.

Then run `npm run build` in your project directory. It should create a build folder. As suggested, then run the following two commands:  
`yarn global add serve`  
`serve -s build`

This should in turn give you a link to open up the built version of the site on a server of its own, different from the development server.

You can now also preview this built version using its own little server, although doing that is completely optional.

#### PRACTICE EXERCISE #5

Inside of a practice folder, call `npx create-react-app exercise-5` from the command line. Then `cd` into the exercise-5 directory from the command line.

Run the following command to eject your configuration file settings and leave Create React App:

`npm run eject`

You will have to confirm “y” that you want to eject from Create React App. If you are using git, you will have to make sure you have no untracked changes before doing this.

Once the eject command has run, you should see a `config` and `scripts` directory in your exercise-5 folder.

Now run `npm run start` just to make sure everything is still working. You should see the development server start, as it did in Practice #2. To stop the server, type `Ctrl + C` in the command line.

There is no undoing the eject command, so only do it when you know what you are doing or have support from someone who does. However, practicing this exercise on a practice project is a good idea to see how it works without breaking an existing project.