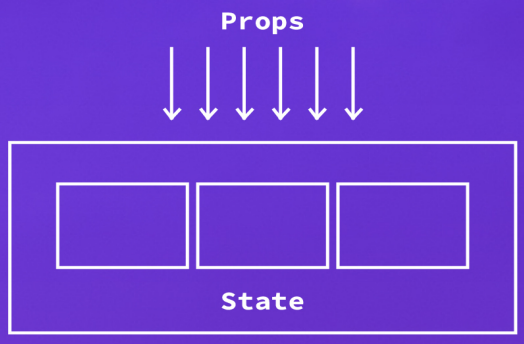
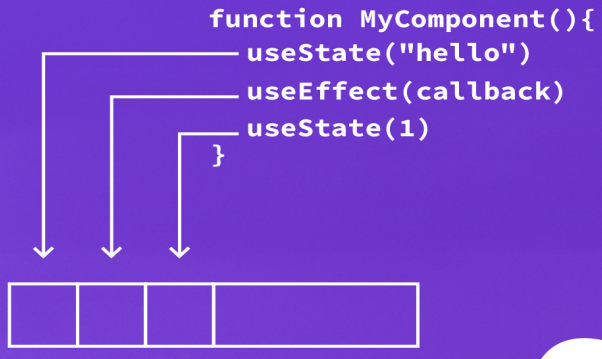
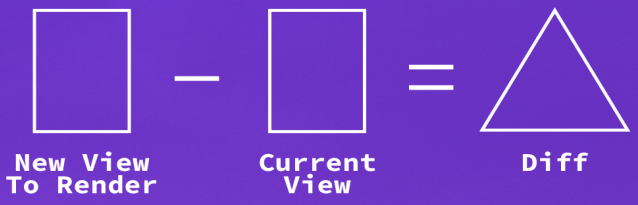
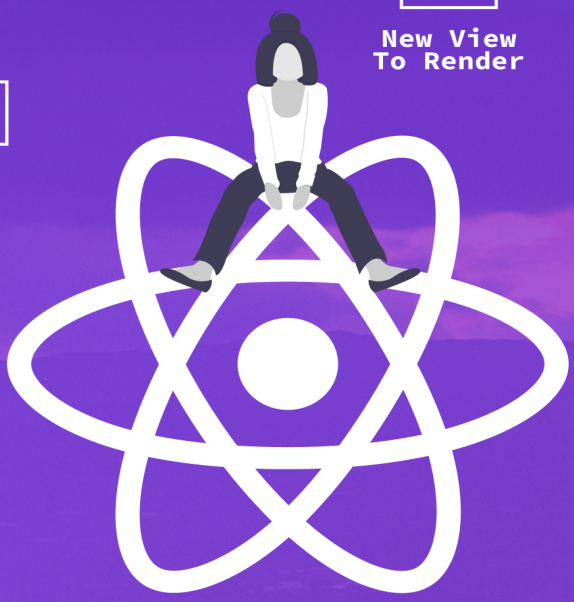
DELIGHTFUL REACT

Code and Sketches for React Beginners









BHARGAV PONNAPALLI

Delightful React

Code and Sketches for React Beginners

Bhargav Ponnapalli

This book is for sale at http://leanpub.com/delightful-react

This version was published on 2019-11-20

1. This Book is dedicated to

My wonderful family. I hope this book serves as a fun guide for readers beginning their journey in Web Development.

Special Mentions

- *Vijay Sharma*
- *Jayaa Bharadwaj*
- *Adil Virani*
- *Anil Choudary*
- *Sandeep & Subhashree*
- *Max, Evan and Phil*
- *Sunil Pai*
- *Dan Abramov*

Contents

2. Introduction to Reactjs

Reactjs is one of the most popular and in-demand javascript frameworks, used by many developers and companies all over the world. **Facebook** open-sourced Reactjs a few years ago, and ever since, it has grown immensely and has received an overwhelming adoption.

React has changed over the years, and ever since 2019 it looks more even refreshing and has a more concise approach to building components.

React's declarative approach is one of its strongest features and it influenced other libraries like Angular and Vue to adopt similar approaches to build meaningful User Interfaces.

The objective of this book is to introduce React in its new avatar, understand how it works, and build a Job Search App.

2.1 First steps

So let us get started by setting up a plain HTML project with React. Copy the snippet below and put it in a HTML file. And open HTML file in the browser.

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <title>Welcome</title>
 </head>
 <body>
  <div id="root"></div>
  <script src="https://unpkg.com/react@latest/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@latest/umd/react-dom.development.js">></\
script>
  <script>
   var domNode = document.getElementById("root");
   var reactElement = React.createElement("p", {
    children: "Hello World!"
   });
   ReactDOM.render(reactElement, domNode);
  </script>
 </body>
</html>
```

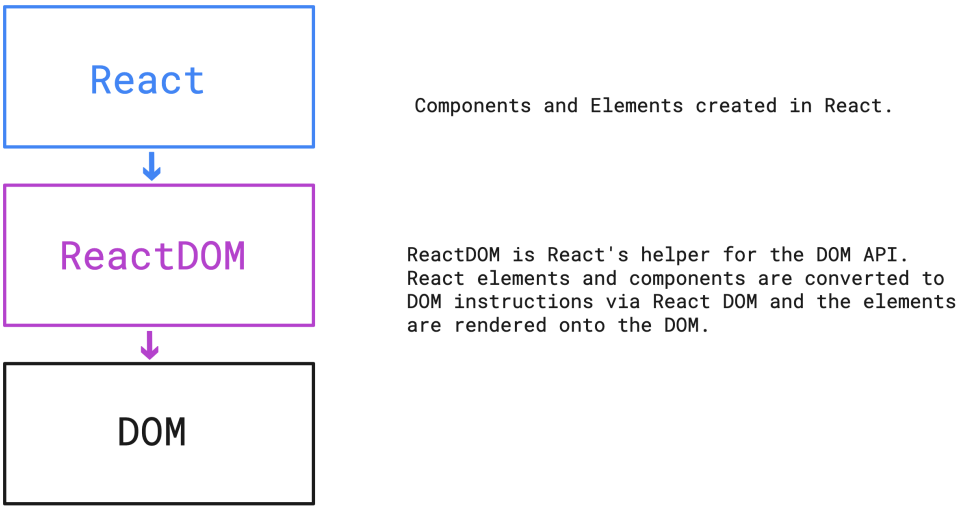
If we open this HTML file in a browser we should see the text `Hello world` on the screen.

React in Plain HTML

2.2 React and ReactDOM

- We import React and ReactDOM using the script tag.
- We identify the DOM node onto which we want to render our React code.
- Next, we create a React element of type `paragraph`, `h1`, or any valid HTML tag and put the text we need inside the `children` option.
- Finally, we `render` the React element onto the DOM node using ReactDOM's `render` method.

Note: We used the paragraph HTML element here but we can use any valid HTML element in the `React.createElement` function and create a corresponding React element.

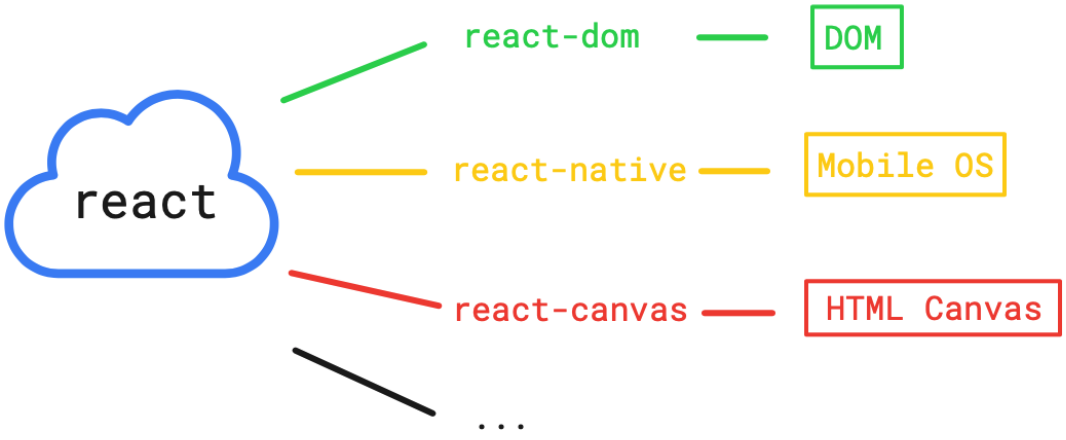


React, ReactDOM and the DOM

So React is where we create elements that correspond to physical DOM nodes in the browser. To render a paragraph DOM element, we create a corresponding React element, and render that using the ReactDOM package. ReactDOM is a glue for React to interact with the DOM.

2.3 React is the brain

React and ReactDOM are both needed to create DOM nodes. They are 2 separate packages because React's core philosophy of creating elements isn't limited to just DOM. React can be used to render content into other environments too. Hence it was envisoned that React would be one package and ReactDOM would be the adapter for React to render DOM nodes. Similar there are other adapters like - react-native, which renders React for native mobile apps - react-canvas, which renders react elements into HTML5 canvas



Adapter frameworks for React

And the best part is that you can build your own adapter for your environment. It is an advanced topic and is beyond the scope of the book, but, building a custom adapter framework for your a new environment might be an exciting side project for you after you have read this book!

2.4 create-react-app

The HTML Setup that we have created earlier is very primitive.

Let's use a more powerful setup tool called create-react-app. So now, let us bootstrap our project from a git repository that I created. So before we move forward, we need to make sure that node.js is installed in our computer because create-react-app is built on top of node.js tools. We need to make sure that node.js of 10 or greater is installed in our system.

- Make sure that node.js v10 or later by running this command.

  ```
  node -- version
  ```

- Please install it from the node.js official website if not installed yet.
- Next, clone the git repository by running

  ```
  git clone https://github.com/delightful-react/jobs-list-app.git
  ```

- Then, enter the root of the repository in the terminal and run *yarn install or npm install*

  ```
  cd jobs-list-app && npm install
  ```

This installs all the dependencies that create-react-app needs to run the project. And once that is done, run

```
npm start
```

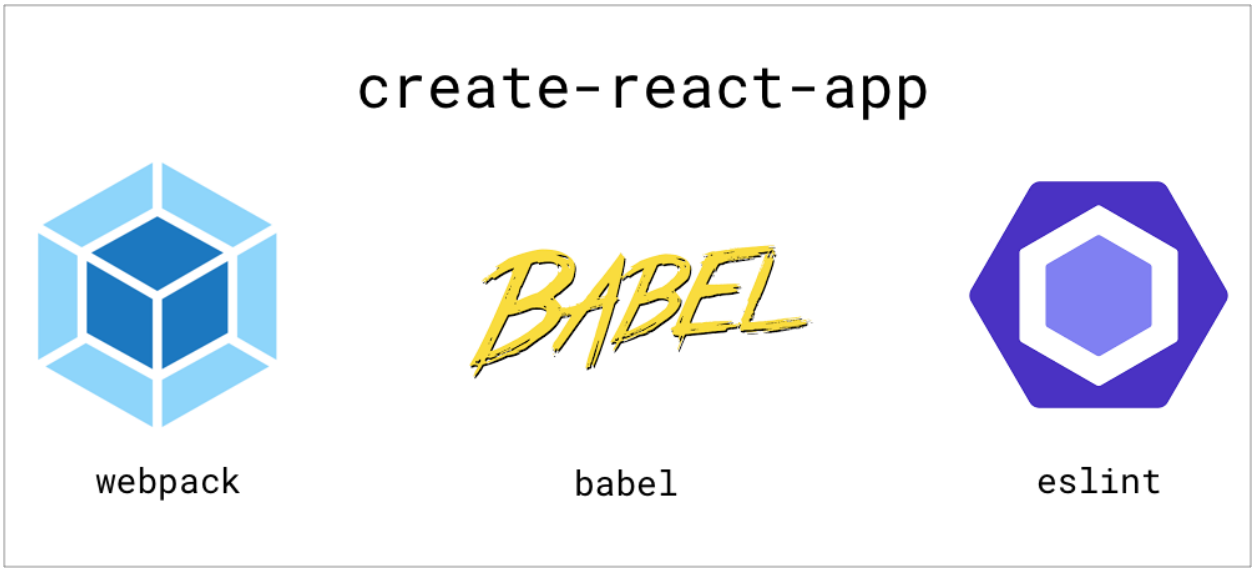
This command starts a project and runs it on port 3000. Now, open your browser and open http://localhost:3000, you should see our project running with this response.

Initial app screenshot

It is similar to what we created using our plain HTML setup, but this setup offers so much more. So create-react-app is capable of many cool developer experience related things.

Some of them include

- hot module replacement, which updates the browser without refreshing when the code changes.
- modern code support
- React JSX syntax support
- production build configuration
- environment variables
- automatically injects script tags with long term caching support into HTML
- and more.



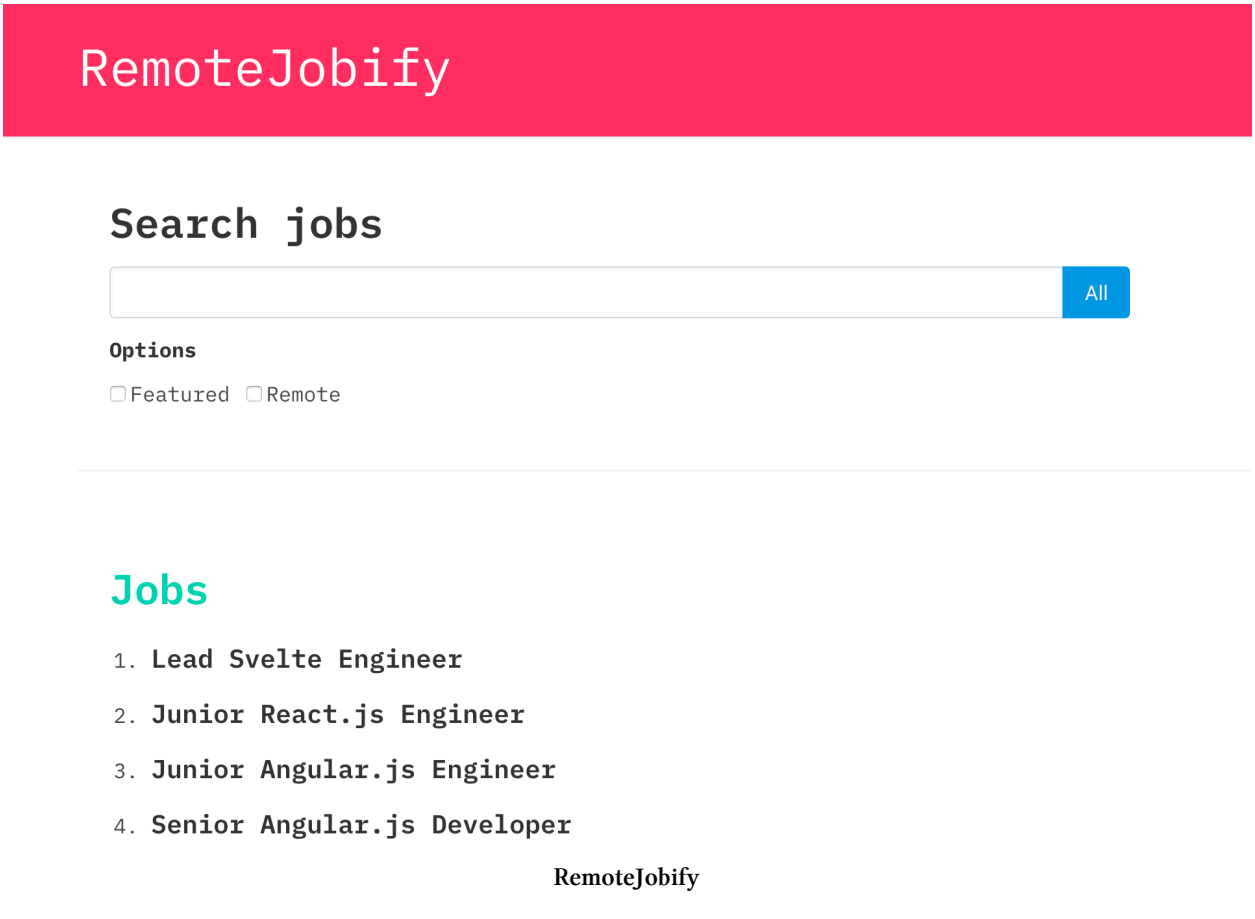
Create React App uses webpack and babel

In this book, we will learn how to create a job listing application while learning React.

By the time we finish all the chapters in the book we will

- be able to render a list of jobs
- be able to filter jobs using form elements
- be able to fetch the jobs list from a remote server.

Our finished project will look so.



RemoteJobify

Let us take a look at source files that we have in our project so far. The three main files that I want you to focus on are the package.json file, src/index.js file and public/index.html file.

- the package.json file is where our scripts are located
- the index.html file is similar to the HTML file that we created earlier. It contains the HTML essential HTML nodes, and it also contains the mount node into which react renders the elements. Notice how the index.html file does not have any script tags. create-react-app automatically compiles the javascript files and adds them as script tags.
- src/index.js file is where we have our first react element rendered onto the mount node.

2.5 The JSX Syntax

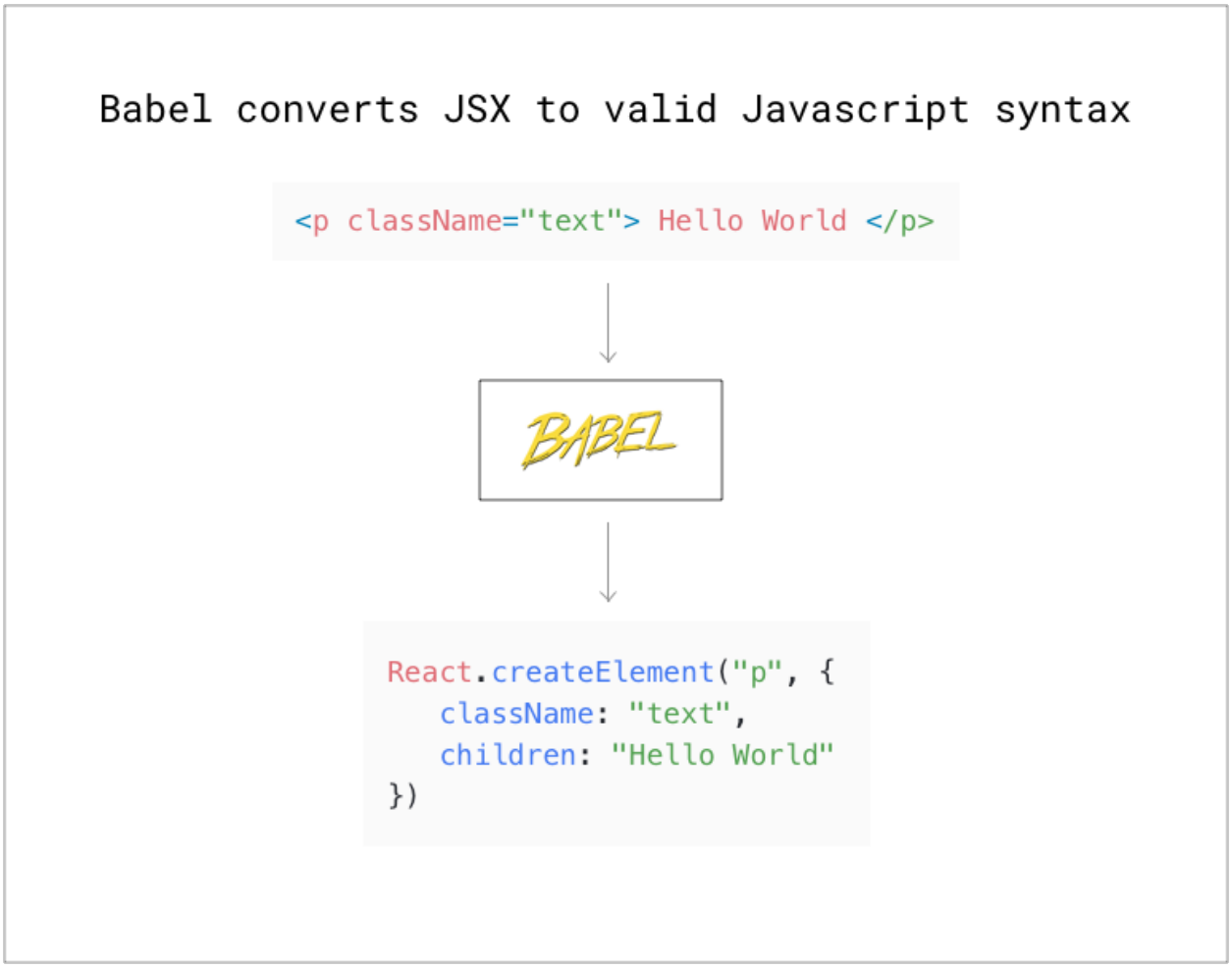
The React code written in plain javascript is verbose. Let us write this snippet.

```
//src/index.js
ReactDOM.render(
 React.createElement("p", {
  children: "Hello Create React App!"
 }),
 mountNode
);
```

And create a React paragraph element similar to a HTML element.

```
ReactDOM.render(<p>Hello Create React App!</p>, mountNode);
```

It works! This syntax is called JSX and it is created to be extremely similar to HTML to make creating and composing React Elements as easy as possible.



Babel converts JSX to Plain Javascript

create-react-app uses babel.js under the hood which transforms modern syntax (including JSX syntax) into browser ready javascript syntax.

Setup is complete

We are now ready with our project setup. Let us learn about React elements and components in the next couple of chapters!

3. Elements and Components

React elements are great. Because of our familiarity with HTML, creating new elements in JSX is straight forward.

Try creating different kinds of React elements.

```
ReactDOM.render(<span>Hello Create React App!</span>, mountNode);
```

Similar to HTML, JSX allows multiple elements to be put one inside another.

```
ReactDOM.render(
 <div>
  <p>Hello Create React App!</p>
  <span> 1 </span>
 </div>,
 mountNode
);
```

As a challenge, try the following elements:

- `h1` element
- a paragraph element
- a list of `li` elements using the `ol` or `ul` tags

Components

One of the strongest feature of React is its components feature. Multiple elements can be composed together to create components.

Let us create our first component like so.

```
function App() {
 return (
  <div>
   <p>Hello Create React App!</p>
   <span> 1 </span>
  </div>
 );
}
```

The component App can the be rendered like React elements like so

```
ReactDOM.render(<App />, mountNode);
```

The JSX expression returned in the App component instance will then be rendered by React.

Note: Component names need to start with a capital letter.

3.1 Composition

Composition is the process of combining multiple elements together. Components can also be composed together like so.

```
function Title() {
 return <h1>Hello Create React App!</h1>;
}

function Description() {
 return <p> A remote jobs app </p>;
}

function App() {
 return (
  <div>
   <Title />
   <Description />
  </div>
 );
}
```

Our project has two major components.

- A FilterJobs component which filters the jobs by using form components like inputs, check boxes and select menus
- A JobsList component which renders the filtered jobs

Let us create them like so.

```
function FilterJobs() {
 // When a component returns null
 // React does not render anything into the DOM
 return null;
}

function JobsList() {
 return (
  <div>
   <ol>
    <li> Lead React Engineer </li>
    <li> Junior Angular Engineer </li>
   </ol>
  </div>
 );
}

function App() {
 return (
  <div>
   <FilterJobs />
   <JobsList />
  </div>
 );
}

ReactDOM.render(<App />, mountNode);
```

3.2 Rendering variables in elements

We can also render variables inside our React elements.

```
function JobsList() {
 const jobTitle1 = "Lead React Engineer";
 const jobTitle2 = "Junior Angular Engineer";
 return (
  <div>
   <ol>
    <li>{jobTitle1}</li>
    <li>{jobTitle2}</li>
   </ol>
  </div>
 );
}
```

Since expressions within {} are evaluated like any other javascript expression, we can access values from any kind of variable in the scope of the component.

```
 const jobTitles = ["Lead React Engineer", "Junior Angular Engineer"];

 <li>{jobTitles[0]}</li>
 <li>{jobTitles[1]}</li>
```

3.3 Rendering a list of elements

React is capable of entering an array of elements as well. It requires a unique key prop to be specified for each of the indices.

So to render an array of 2 elements we can do something like this.

```
const jobTitles = ["Lead React Engineer", "Junior Angular Engineer"];

const elements = [<li key={jobTitles[0]}>{jobTitles[0]}</li>, <li key={jobTitles[1]}\
>{jobTitles[1]}</li>];

<ol>{
 jobTitleElements;
}</ol>
```

Transforming a list using Array.map

In the real world, we don't know how large an array is and for such scenarios we transform the array of value into an array of elements by using the Array.map function.

The map function takes in a transformation function as argument and it calls the transformation function for each element in the source array the computed return value becomes an entry at the corresponding location in the transformed array.

This way we can create an element for each value in the source array and render the resultant array within our React elements.

```
const elements = jobTitles.map(jobTitle => <li key={jobTitle}>{jobTitle}</li>);

<ol>{elements}</ol>;
```

Finally let us add a little structure to our elements like so and render them to the screen. The elements article and h4 etc are mainly present to make our elements look good when they are rendered to the screen. Styling for these elements and classes are already linked to our project.

```
function JobsList() {
 const jobTitles = ["Lead React Engineer", "Junior Angular Engineer"];

 const elements = jobTitles.map(jobTitle => (
  <li key={jobTitle}>
   <article className="media job">
    <div className="media-content">
     <h4>{jobTitle}</h4>
    </div>
   </article>
  </li>
 ));

 return (
  <div>
   <ol>{elements}</ol>
  </div>
 );
}
```

Awesome! Our app is now taking shape. It should now look like this.

Rendering a list of job titles

4. Components and Props

Components are like functions. Functions are used to combine reusable expressions. Components are used to combine reusable elements. In the last chapter, we added some markup to our list items to make it look better, but all of that markup can be associated with a single Job item.

4.1 The Job component

Let us create a job component that will render this markup for us.

Components and elements are capable of rendering dynamic values. But where can we get the job title from?

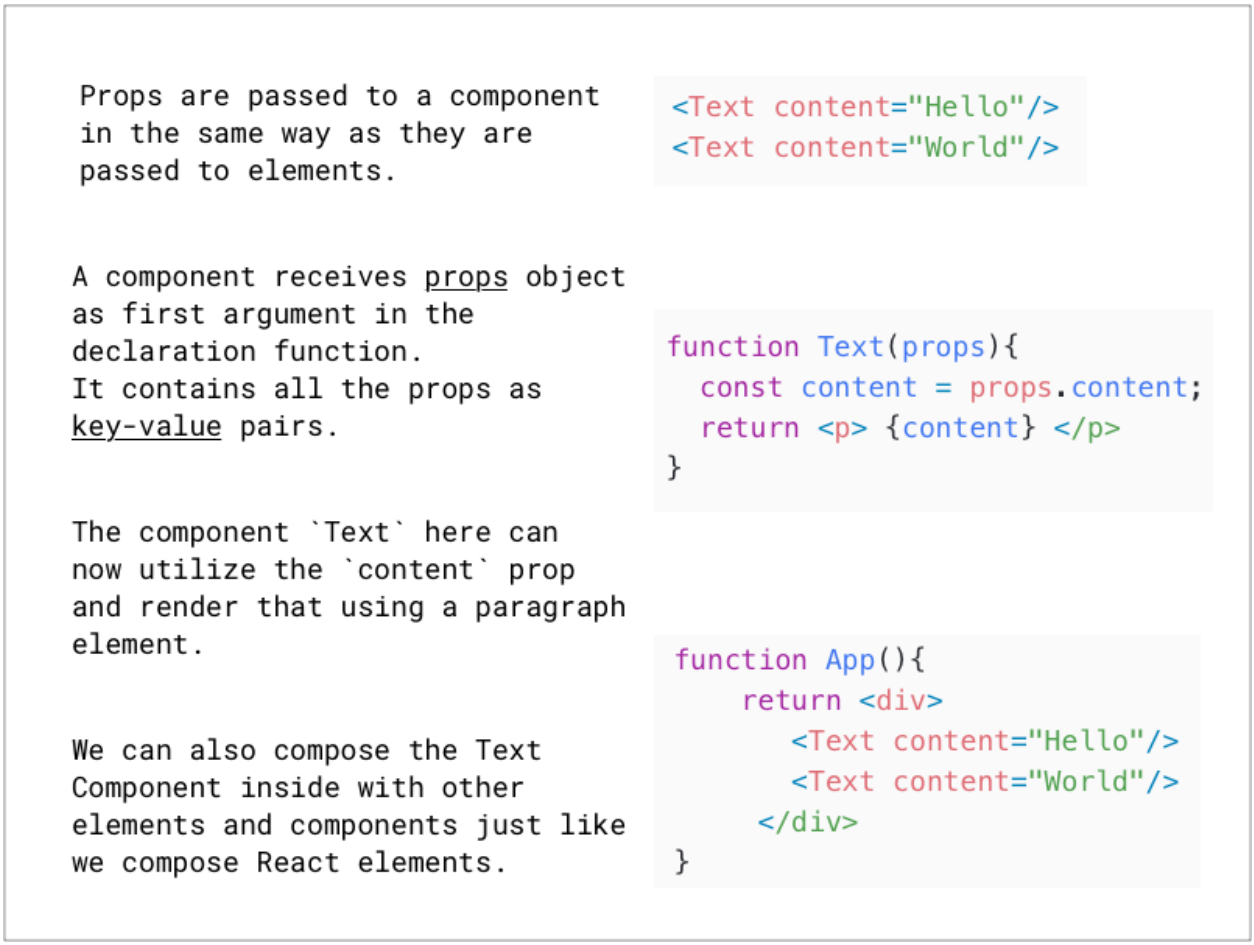
```
function Job() {
 const jobTitle = ???
 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4>{jobTitle}</h4>
    </div>
   </article>
  </li>
 );
}
```

We can get the job title from the arguments of the `Job` component. When the `Job` component is created and values are passed in this manner,

```
<Job title={jobTitle} />
```

The values passed to the component can be used inside the component definition like so.

```
function Job(props) {
 const jobTitle = props.title;
 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4>{jobTitle}</h4>
    </div>
   </article>
  </li>
 );
}
```



component-props

Finally, we want to render an array of Jobs, so we can render the `Job` component with the `key` prop and pass `jobTitle` as well.

```
const elements = jobTitles.map(jobTitle => (
 <Job key={jobTitle} title={jobTitle} />
));
```

4.2 Non-primitive values as props

Props don't need to primitive values like strings or numbers. They can also be non-primitive or reference type data like objects, functions and arrays.

Let us modify our jobs list to include descriptions as well.

```
const jobs = [{
   title : "Junior Angular Engineer",
   description: "
   Amet quo non reprehenderit aspernatur non ex tenetur debitis impedit. Dolor sed \
est. Dolorem assumenda molestiae vitae accusantium facilis incidunt rem soluta sint.\
 Velit tenetur quae quibusdam occaecati fuga itaque tenetur ut.
   ",
}, {
   title: "Junior Angular Engineer",
   description: "Aut commodi rem dolorem et ad aut error. Praesentium quis aspernat\
ur reprehenderit quibusdam est deleniti eos. Laborum quaerat omnis soluta nisi.",
}]
```

We can also pass multiple props to a component so we can pass title and description to our job component via props and we can render them like so.

```
const elements = jobs.map(job => (
 <Job key={jobTitle} title={job.title} description={job.description} />
));

function Job(props) {
 const jobTitle = props.title;
 const jobDescription = props.description;
 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4>{jobTitle}</h4>
     <p>{jobDescription}</p>
    </div>
```

```
    </article>
   </li>
  );
}
```

We can also pass the entire job object itself as props. Since props are just values, they can be any Javascript value. They can be primitive values like strings numbers or booleans. They can also be objects,functions or arrays.

```
function Job(props) {
 const { title, description } = props.job;
 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4>{title}</h4>
     <p>{description}</p>
    </div>
   </article>
  </li>
 );
}

const elements = jobs.map(job => <Job key={job.title} job={job} />);
```

4.3 Event handlers

React elements can handle events similar to how HTML elements have event attributes like onclick, onmousedown etc. So to pass an event handler to an element, we can do like so

```
function MyComponent() {
 function handleClick(event) {
  console.log(event);
 }
 return <p onClick={handleClick}> Hello </p>;
}
```

Try it out on any of our elements and see it in action.

Note that even handlers accept functions as arguments and whenever the event occurs, they are generally invoked with the event object as first argument, which allows you to react to the event with the data that the event contains. Here is an updated infographic on how events work within React

4.4

Event handlers make React elements extremely easy to work with. We will discuss more about events in subsequent chapters.

4.5 Webpack code reorganize

Let us take a step back and slightly reorganize our code better. So within our src folder, let us create two directories:

- src/data
- src/components

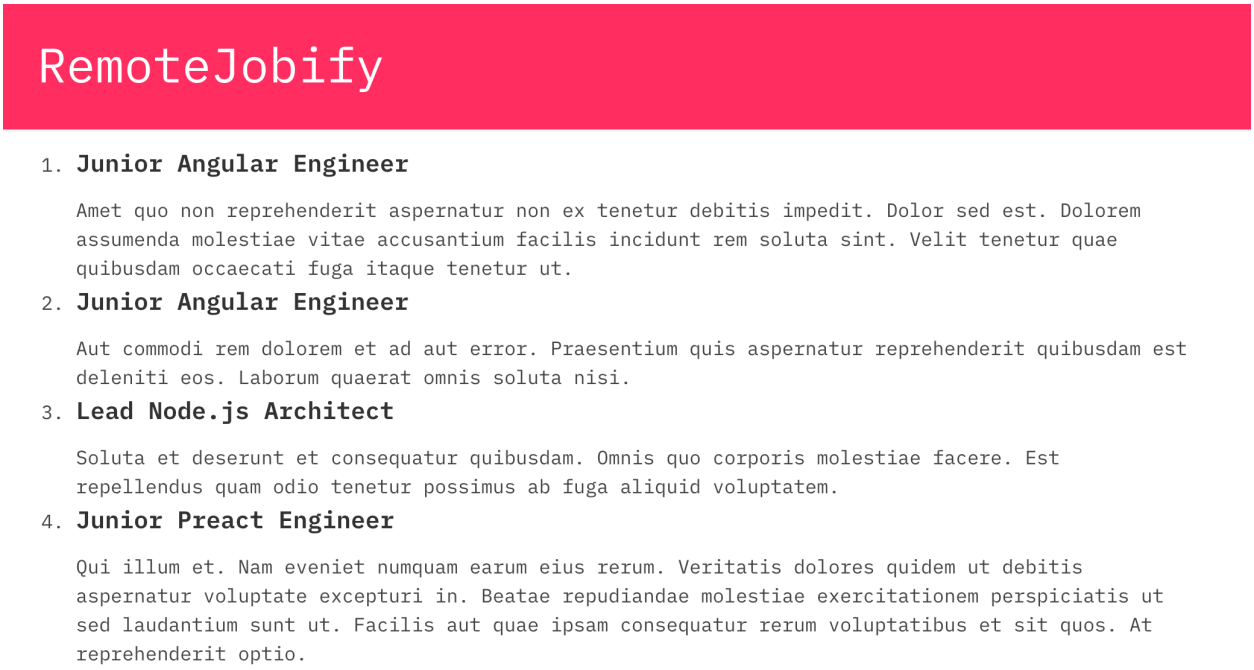
Let us move all our components into the components folder and move our jobs array into src/data/jobs.js.And now within our components we can import the values we need.

Here is how our files look like at the end of this chapter. Please take a look at the first comment in each snippet where the name and location of the file is mentioned for convenience.

```
// src/data/jobs.js

const jobs = [
 {
  title: "Junior Angular Engineer",
  description:
   "Amet quo non reprehenderit aspernatur non ex tenetur debitis impedit. Dolor s\
ed est. Dolorem assumenda molestiae vitae accusantium facilis incidunt rem soluta si\
nt. Velit tenetur quae quibusdam occaecati fuga itaque tenetur ut."
 },
 {
  title: "Mid level Angular Engineer",
  description:
   "Aut commodi rem dolorem et ad aut error. Praesentium quis aspernatur reprehen\
derit quibusdam est deleniti eos. Laborum quaerat omnis soluta nisi."
 },
 {
```

```
  title: "Lead Node.js Architect",
  description:
   "Soluta et deserunt et consequatur quibusdam. Omnis quo corporis molestiae fac\
ere. Est repellendus quam odio tenetur possimus ab fuga aliquid voluptatem."
 },
 {
  title: "Junior Preact Engineer",
  description:
   "Qui illum et. Nam eveniet numquam earum eius rerum. Veritatis dolores quidem \
ut debitis aspernatur voluptate excepturi in. Beatae repudiandae molestiae exercitat\
ionem perspiciatis ut sed laudantium sunt ut. Facilis aut quae ipsam consequatur rer\
um voluptatibus et sit quos. At reprehenderit optio."
 }
];

export default jobs;


// src/components/Job.js

import React from "react";

function Job(props) {
 const jobTitle = props.title;
 const jobDescription = props.description;
 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4>{jobTitle}</h4>
     <p>{jobDescription}</p>
    </div>
   </article>
  </li>
 );
}

export default Job;
```

```
// src/components/JobsList.js

import jobs from "../data/jobs";

function JobsList() {
 return (
  <div>
   <ol>
    {jobs.map(job => (
     <Job key={job.title} job={job} />
    ))}
   </ol>
  </div>
 );
}


// src/index.js

import JobsList from "./components/JobsList";
import FilterJobs from "./components/FilterJobs";

function App() {
 return (
  <div>
   <FilterJobs />
   <JobsList />
  </div>
 );
}

ReactDOM.render(<App />, mountNode);
```



Rendering an array of job objects

Props are incredibly powerful tools that help us write components that are reusable.And can work with different kinds of data in different environments. In the next chapter, we'll talk about another powerful React.js component behaviour, state.

5. Components and Hooks

The Component mode is React's core feature. Components allow us to compose multiple elements or even other components together to form reusable logic. But what makes components even more powerful are hooks. Hooks are supplements to components to solve common use cases.

React has a few built in hooks that allow us to to add behavior to a component. Let us talk about one such hoping this chapter it is called useState.

Modern Syntax: Array destructuring

Before we move onto useState, here is a small note about array destructuring.

Array destructuring is a new syntax within JavaScript. It is useful to access elements in an array in a more concise syntax. Consider the following examples which do the same thing.

Without destructuring

```
const fruits = ["mango", "banana"];
const a = fruits[0];
const b = fruits[1];
// a is 'mango'
// b is 'banana'
```

With destructuring

```
const fruits = ["mango", "banana"];
const [a, b] = fruits;
// a is 'mango'
// b is 'banana'
```

Array destructuring is a shorter syntax and we will use this with throughout our book for useState hook.

5.1 The useState hook

The useState hook allows a component to maintain a local state value. state variables are special values that are maintained across component rerenders. Local state allows a component to create values that are remembered by the component.

Here is how it works.

5.2

So to start using useState, we need to import useState from react.

```
import React, { useState } from "react";
```

We want to create a state variable inside our Job component to conditionally show the description when the state value is true.

- First, let us create a state variable called isDetail.

  ```
  // src/components/Job.js

  function Job(){
    const [isDetail, setIsDetail] = useState(false);
    ...
  }
  ```

- Next, let us create a click event handler on the article element.

  ```
  function handleClick() {
   console.log("clicked");
  }
  <article onClick={handleClick}>...</article>;
  ```

- Finally, let us link the setIsDetail updation function and call it within the event handler function handleClick.

  ```
  // src/components/Job.js

  import React, { useState } from "react";

  function Job(props) {
   const { job } = props;
   const { title, description } = job;
   const [isDetail, setIsDetail] = useState(false);

   function handleClick() {
    setIsDetail(true);
   }

   return (
    <li>
  ```

```
      <article onClick={handleClick} className="media job">
       <div className="media-content">
        <h4>{title}</h4>
        {isDetail ? <p>{description}</p> : null}
       </div>
      </article>
     </li>
    );
}

export default Job;
```

- Since the {} simply evaluate javascript expressions, we can render different elements by using the ternary if-else operator.
- Right now, our Job component only shows the description on click but doesn't toggle it back when clicked on again. Let us change that by doing so.

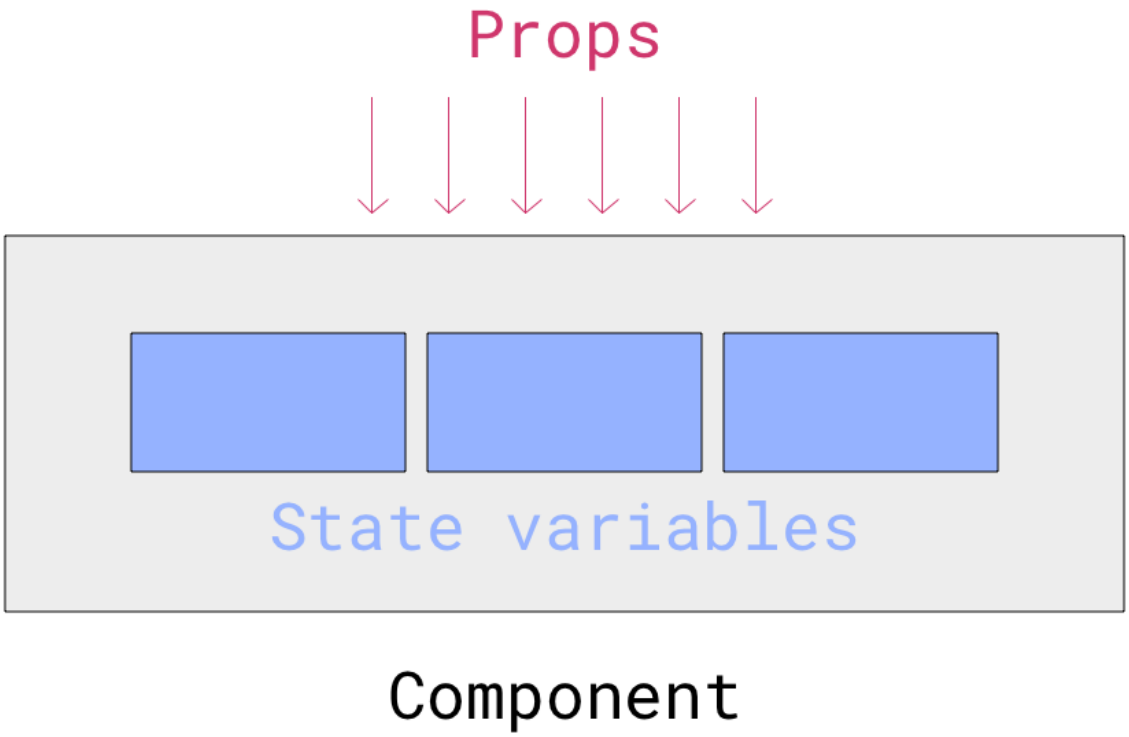
```
function handleClick() {
 setIsDetail(!isDetail);
 // if isDetail is false
 // setIsDetail(true) is called
 // and vice versa
}
```

Now, if we look at our browser, our Job is interactive. We are able to toggle the description of a Job by clicking on it. State adds interactivity to our components and allows components to create and maintain data locally.

***Note** : Each instance of a component creates it's own memory within which it's states and props are stored. Which is why clicking on one Job component doesn't show the description of another Job component.*

5.3 Props and State

Props and state are very essential for real world React applications. They often in fact used together. Let us take a look at the similarities and the differences between props and state with these two diagrams.



How Props and State look like in a component

Props	State
Passed to the component externally	Created internally by the component
Cannot be modified by the component	Can be modified by the component
Can be accessed as the first argument of the component definition function	Can be created using the useState hook
Can be given a default value as fallback	Can be given a default value as fallback
Component rerenders on update	Component rerenders on update

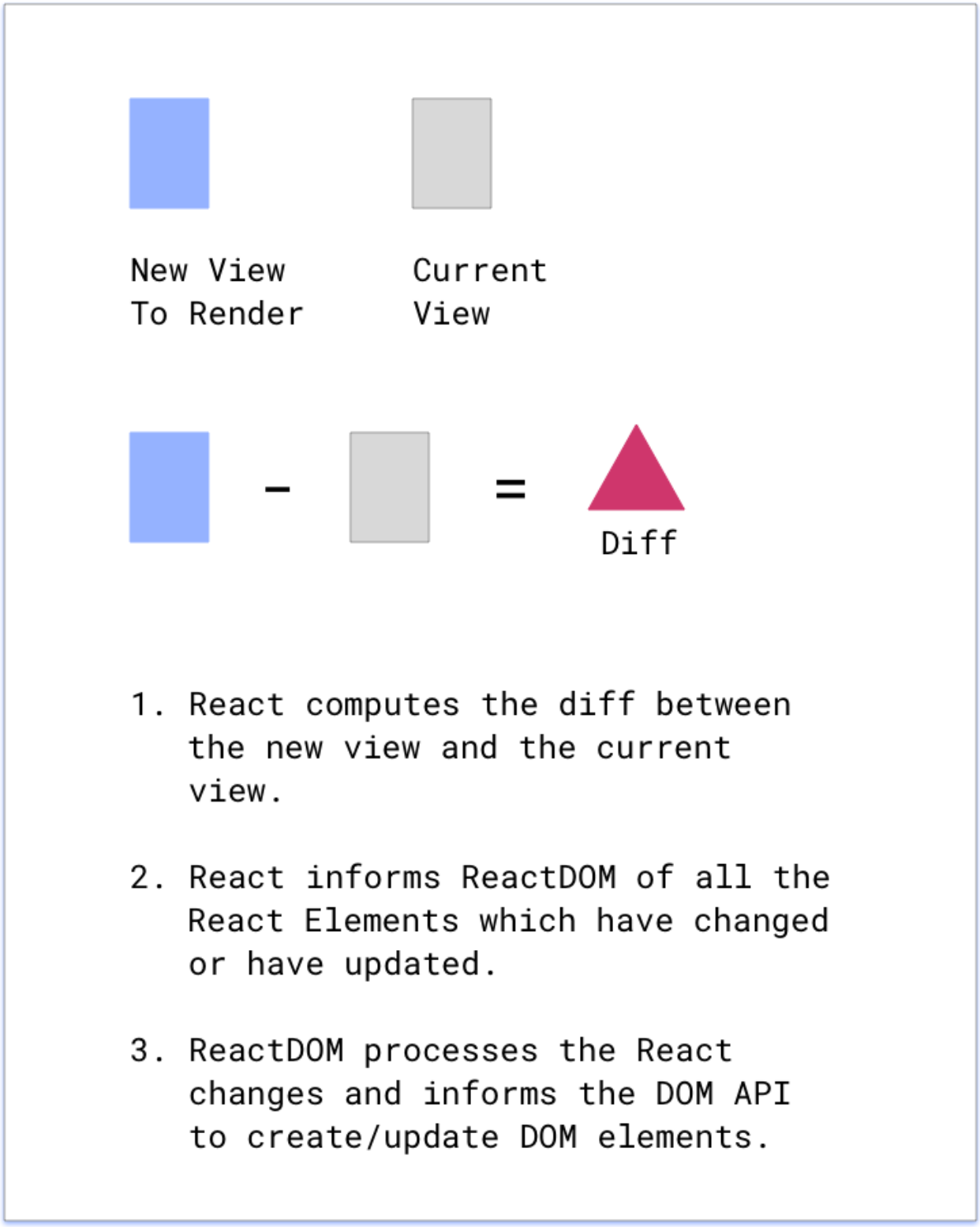
Props vs State

Props in Components are analogous to arguments for a function. Just like arguments to a function are passed from outside the function, props are sent to the component from outside. State in components is analgous to local variables in a function. They are created internal to the component. **Note**: However, state variables can hold values across multiple renders.

5.4 Is rerendering slow?

A good question to ask ourselves is that, "If React rerenders the entire component when state or props change, isn't that slow?"

Performance is extremely important in real-world apps and when it comes to whether rerendering is slow, the answer is **No**. React rerenders the component in memory using Virtual DOM which is an in memory implemtation of DOM and is really fast. Once it renders the component in the VDOM, it computes the difference between the existing and the new VDOM markup and performs only the actual DOM mutations corresponding to the diff. This is very performant and fast.



Virtual DOM diffing

We have understood how props and state are linked together and how components rerender when props or state changes. Next, let us create form elements using state for our `FilterJobs` component to filter the jobs list.

6. Forms

Let us start filtering our jobs by title. To interact with elements and webpage, we need Form elements. Form elements are interactive elements in the webpage, for eg: inputs, checkboxes, radio buttons, etc. They allow the user to enter values and interact with elements in a webpage.

So let us create a HTML input element inside our FilterJobs component. Let us also add some markup that gives our component a good look.

//src/components/FilterJobs.js

```
function FilterJobs(props) {
 return (
  <section className="section filter-jobs">
   <h1>Search Jobs</h1>
   <div className="field has-addons">
    <div className="control is-expanded">
     <input className="input" />
    </div>
   </div>
  </section>
 );
}
```

Now, if we type into the input field, it works fine except React doesn't really have a way of accessing that data within the component. It renders the input field, but after the rendering, every activity that the user does on input feed isn't sent back to react. Since React renders react elements to HTML, not the other way around, there is no easy way to access the text typed inside the input element.

6.1 Controlled Inputs

React solves this problem by using what is called Controlled Inputs. Controlled inputs are inputs whose values are managed completely by React.

How controlled inputs work

1. A React input element with a value prop is read-only. Typing into such an input does nothing. React doesn't allow a user to update its value. The input's value can only be updated by React.

```
<input />
// This is an uncontrolled input
// React cannot access the value entered by the user
// without accessing the DOM Node


<input value={inputValue} />
// This is a controlled input
// React doesn't allow the user to update
// the HTML element
```

1. The input element is given a state variable as value. Updating this state variable, triggers a rerender and React updates the input with the new value.
2. To update the state variable, the onChange React event is used on the input field. The event handler supplied as onChange prop is called with the event as argument. The event's event.target.value contains the new value that the user has typed in (but was not rendered yet).

```
function handleChange(event) {
 const newValue = event.target.value;
 // newValue is available here
}

<input className="input" value={value} onChange={handleChange} />;
```

1. Within the onChange handler, the new value is used and the state variable is updated. Since a state variable update causes a rerender, React rerenders and the new value is reflected in the input.

```
// If value is a state variable
// It can be updated in this manner
const [value, setValue] = useState("");

function handleChange(event) {
 const newValue = event.target.value;
 setValue(newValue);
}
<input className="input" value={value} onChange={handleChange} />;
```

After updating FilterJobs, it should look like this.

```
//src/components/FilterJobs

function FilterJobs() {
 const [searchText, setSearchText] = useState("");

 function handleChange(event) {
  setSearchText(event.target.value);
 }

 return (
  <section className="section filter-jobs">
   <h1>Search Jobs</h1>
   <div className="field has-addons">
    <div className="control is-expanded">
     <input className="input" value={searchText} onChange={handleChange} />
    </div>
   </div>
  </section>
 );
}
```

This makes our input work exactly like in the uncontrolled mode, except now the value of the input is fully available to React within the variable searchText. We can now use searchText string to check whether a job in the JobList contains it in it's title or not.

6.2 Inter-component communication

React components communicate with each other in the form of props.Props can only flow in a top-down way. A parent can send props to a child component but not vice versa. Sibling components cannot send props to each other. FilterJobs and JobsList components are siblings, so they cannot communicate the searchText variable.

However, they can do so if searchText was within a common parent and both FilterJobs and JobsList components received searchText and setSearchText as props.

So let us move our search text state variable into the common parent, which is the App component and let us pass the search text and set search text values as props to our FilterJobs component.

```
//src/index.js

function App() {
 const [searchText, setSearchText] = useState("");

 return (
  <div>
   <FilterJobs searchText={searchText} setSearchText={setSearchText} />
   <JobsList searchText={searchText} />
  </div>
 );
}


//src/components/FilterJobs.js

function FilterJobs(props) {
 const { searchText, setSearchText } = props;

 function handleChange(event) {
  setSearchText(event.target.value);
 }

 return (
  <section className="section filter-jobs">
   <h1>Search Jobs</h1>
   <div className="field has-addons">
    <div className="control is-expanded">
     <input className="input" value={searchText} onChange={handleChange} />
    </div>
   </div>
  </section>
 );
}
```

Now within a JobsList component, let us access our searchText and filter the jobs like so.

```
// src/components/JobsList.js

function JobsList(props) {
 const { searchText } = props;
 const filteredJobs = jobs.filter(job => {
  return job.title.toLowerCase().includes(value.toLowerCase());
 });
 return (
  <div>
   <ol>
    {filteredJobs.map(job => (
     <Job key={job.title} job={job} />
    ))}
   </ol>
  </div>
 );
}
```

Note we are using the filter function which is available to all JavaScript arrays. The filter function runs a function on all elements inside an array and applies a predicate function. If the predicate function returns true for that element, the element is retained in the filtered array else it is filtered out.

This way only jobs which contain the searchText substring in their title are rendered to the screen. Try it out!

6.3

6.4 Filtering with checkboxes

Before we move forward, let us update our jobs data and have each of the jobs to have two more values isFeatured and isRemote boolean values.

- isFeatured to indicate whether this job is featured in the list of items
- isRemote as a name suggests it tells whether the job is fully remote or not.

```
// src/data/jobs.js
 const jobs = [{
   id: 1,
   title: "Lead Svelte Engineer",
   description:
    "Amet quo non reprehenderit aspernatur non ex tenetur debitis impedit. Dolor s\
ed est. Dolorem assumenda molestiae vitae accusantium facilis incidunt rem soluta si\
nt. Velit tenetur quae quibusdam occaecati fuga itaque tenetur ut.",
   isFeatured: true,
   isRemote: false,
 },
 {
   id: 2,
   title: "Junior React.js Engineer",
   description:
    "Aut commodi rem dolorem et ad aut error. Praesentium quis aspernatur reprehen\
derit quibusdam est deleniti eos. Laborum quaerat omnis soluta nisi.",
   isFeatured: false,
   isRemote: true,
 },
 ...
}]
```

I created an updated list of jobs to put into the src/data/jobs.js file and it is available here. https://delightful-react-assets.imbhargav5.com/public/jobs-min.js. Paste the contents of that file into your src/data/jobs.js file and we are ready to filter our jobs even more.

Let us add a couple of checkboxes to allow a user to filter by these boolean values.

Similar to the input, let us add some state to our app component like so.

```
//src/index.js

function App() {
 const [searchText, setSearchText] = useState("");
 const [showOnlyFeaturedJobs, setShowOnlyFeaturedJobs] = useState(false);
 const [showOnlyRemoteJobs, setShowOnlyRemoteJobs] = useState(false);

 return (
  <div>
   <FilterJobs
    searchText={searchText}
    setSearchText={setSearchText}
    showOnlyFeaturedJobs={showOnlyFeaturedJobs}
```

```
      setShowOnlyFeaturedJobs={setShowOnlyFeaturedJobs}
      showOnlyRemoteJobs={showOnlyRemoteJobs}
      setShowOnlyRemoteJobs={setShowOnlyRemoteJobs}
     />
     <JobsList
      searchText={searchText}
      showOnlyFeaturedJobs={showOnlyFeaturedJobs}
      showOnlyRemoteJobs={showOnlyRemoteJobs}
     />
    </div>
  );
}
```

The only difference between the input and the checkboxes is that check boxes have received event.target.checked instead of event.target.value, so our updated code will look like this.

//src/components/FilterJobs.js

```
function FilterJobs(props) {
 const {
  searchText,
  setSearchText,
  showOnlyFeaturedJobs,
  setShowOnlyFeaturedJobs,
  showOnlyRemoteJobs,
  setShowOnlyRemoteJobs
 } = props;

 function handleChange(event) {
  setSearchText(event.target.value);
 }

 function handleShowFeaturedOnlyChange(event) {
  showOnlyFeaturedJobs(event.target.checked);
 }

 function handleShowRemoteOnlyChange(event) {
  setShowOnlyRemoteJobs(event.target.checked);
 }

 return (
  <section className="section filter-jobs">
   <h1>Search Jobs</h1>
```

```
    <div>
     <div className="field has-addons">
      <div className="control is-expanded">
       <input
        className="input"
        value={searchText}
        onChange={handleChange}
       />
      </div>
     </div>
     <div className="field">
      <label className="label">Options</label>
     </div>
     <div className="field is-grouped">
      <div className="control">
       <label className="checkbox" htmlFor="featured">
        <input
         id="featured"
         type="checkbox"
         checked={showOnlyFeaturedJobs}
         onChange={handleShowFeaturedOnlyChange}
        />
        Featured
       </label>
      </div>
      <div className="control">
       <label className="checkbox" htmlFor="remote">
        <input
         id="remote"
         type="checkbox"
         checked={showOnlyRemoteJobs}
         onChange={handleShowRemoteOnlyChange}
        />
        Remote
       </label>
      </div>
     </div>
    </div>
   </section>
  );
}
```

Finally, let us access the showOnlyFeaturedJobs and showOnlyRemoteJobs props and filter the jobs like

so.

```
// src/components/JobsList.js

function JobsList(props) {

 const { searchText, showOnlyFeaturedJobs, showOnlyRemoteJobs } = props;
 let filteredJobs = jobs.filter(job => {
  return job.title.toLowerCase().includes(searchText.toLowerCase());
 });

 if (showOnlyFeaturedJobs) {
  filteredJobs = filteredJobs.filter(job => job.isFeatured);
 }
 if (showOnlyRemoteJobs) {
  filteredJobs = filteredJobs.filter(job => job.isRemote);
 }

 return ...
}
```

Excellent! We now have an app that can filter jobs by multiple inputs seamlessly which we built using just the fundamental building blocks of React.

6.5

We used useState to accomplish a lot in the chapter. But are we using it the right way? Let us talk about the *Dos and Don'ts* of hooks in the next chapter.

7. The Magic of Hooks

We have seen how `useState` can be used to create state within a component and `useState` is a built-in React hook. Hooks are supplements for React components which allow us to add behaviour to them.

There are ï¿¼3 basic built-in hooks:

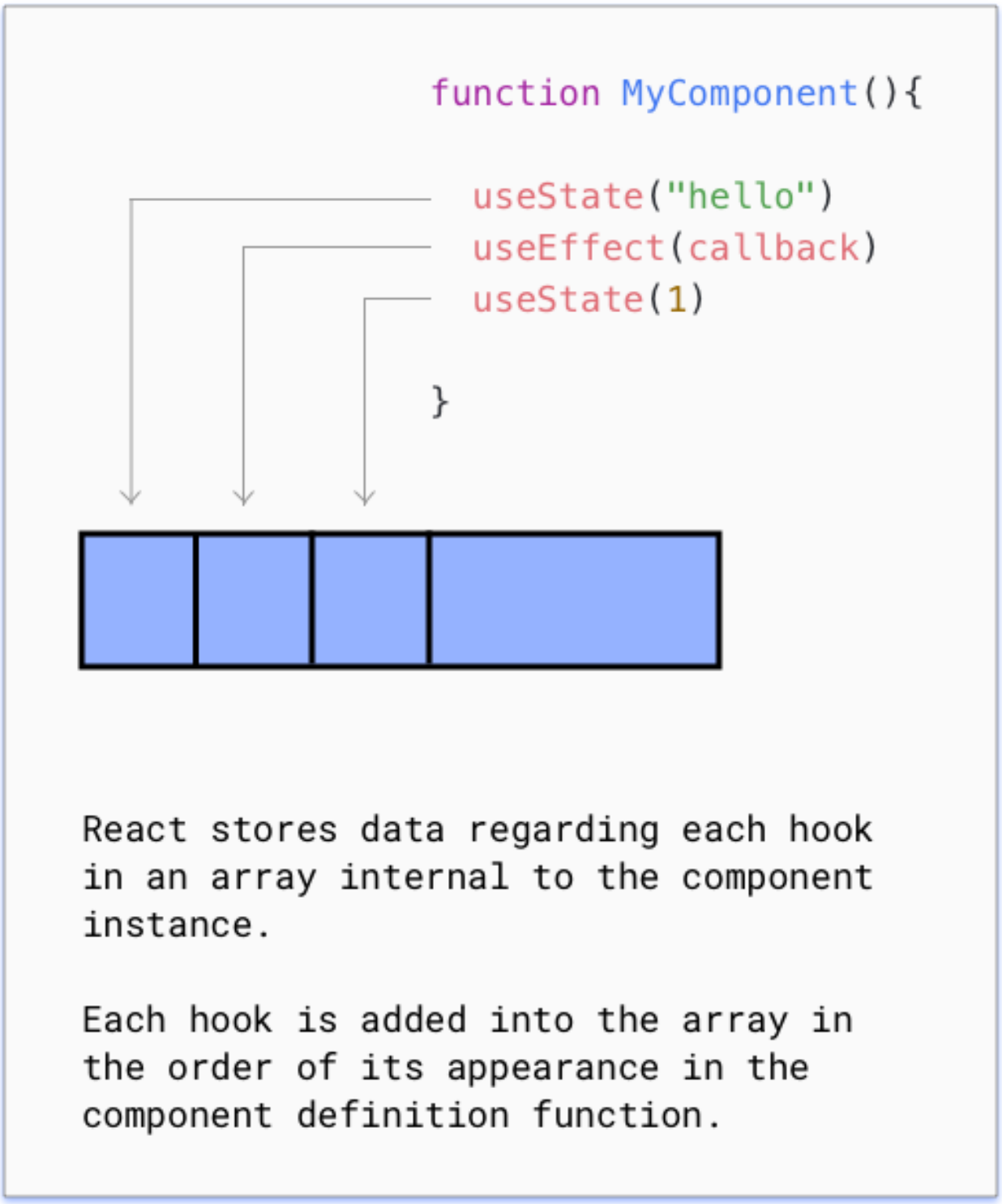
- `useState` - allows components to create and manage state.
- `useEffect` - allows components to run effects after a `render`.
- `useContext` - allows components to subscribe to `Context`.

There are also a few additional hooks with new ones in the works by the React team. We will discuss only the basic hooks within this book. To learn more about the additional built-in hooks, the official documentation at https://reactjs.org is a good place to start.

7.1 How do hooks work

Understanding the implementation details of hooks within the React library is crucial to understanding the best practices of using hooks.

- Hooks work within a component by using `Arrays`
- When a hook like `useState` is used in a component, its value is maintained across renders in a position inside the Array. That position can never change and that is crucial for hooks to work.
- When `n` hooks are used within a component an Array of `n` elements is created within the component's internals and the data for hooks is managed using that array.



How do hooks actually work?

7.2 How useState actually works

All hooks follow the same approach. Let us understand this implementation further by understanding how useState works internally. When a variable is created within a function, each time the function runs, the variable is created again and the value doesn't remain same across multiple function calls. The value is always reset to the instantiation.

```
function f() {
 let a = 1;
 console.log(a);
 a++;
 console.log(a);
}

f(); // prints 1 and 2
f(); // prints 1 and 2
// No matter how many times this function runs
// it always prints 1 and 2 because `a`
// always starts at 1.
```

However, we have seen that useState is capable of retaining the value across function invocation (which is component rerender) in this case. Other hooks can also maintain information across renders.

```
// React components are function
// However the value of `a` isn't reset when
// the `MyComponent` component is rendered
function MyComponent() {
 const [a] = useState(1);
 return <div>{a}</div>;
}
```

While the actual implementation of useState has more to it than just this explanation, the way useState works can be summarized in the following steps.

Let us consider this component.

```
function MyComponent() {
 const [a, setA] = useState(1);
 function handleClick() {
  setA(a++);
 }
 return <div onClick={handleClick}>{a}</div>;
}
```

- When the component is rendered for the first time, useState creates an element in the hooks manager Array and initializes with the initial value supplied to it in the hooks manager Array. (Note: It stores more information alongside the value, but let us not worry about that for now.) The initial value here is 1. It then reads that value from the hooks manager Array at the first position since it is the first hook in the component and returns that value.
- When the div is clicked and the state is updated, setA internally updates the value at the first index of the hooks manager array to 2 and requests a component rerender. When the component rerenders, useState runs again. However this time, it doesn't create a new value and initialize it with 1, but simply returns the value at the first position in the hooks manager array. That value is 2. This allows the variable a to be created with the last value that was updated to the hooks manager array.
- Even if the component rerenders for parent component updation reasons, useState simply reads the value from the hooks manager array.

If multiple useState hooks are used in the same component like in this component here.

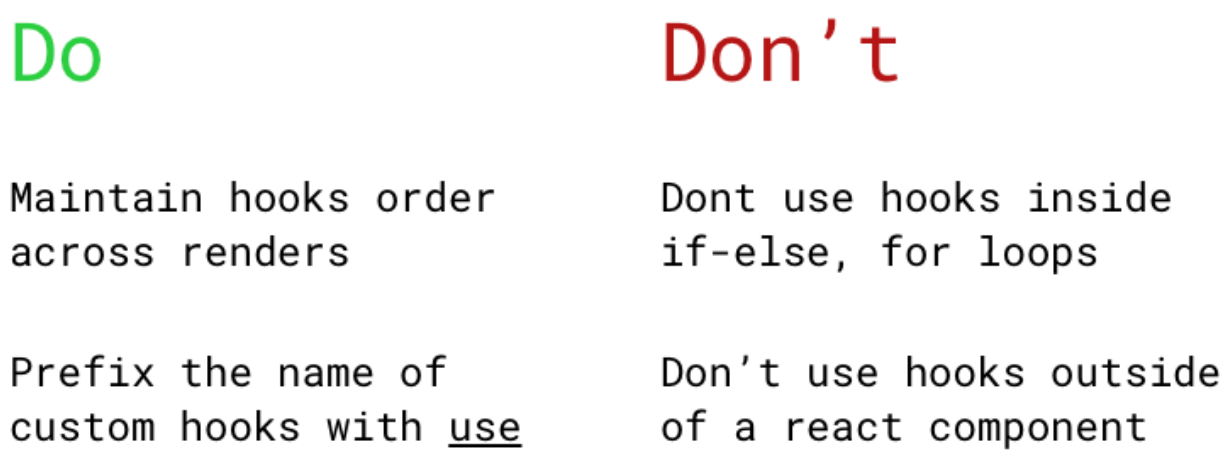
```
function MyComponent(){
 const [a,setA] = useState(1);
 const [b,setB] = useState(5);
 const [c,setC] = useState(9000);
 ...
}
```

- Each useState has a position in the Array and they are in the same order as they are used in the array. So the values 1, 5 and 9000 are present the first, second and third positions in the Array respectively.
- When either of these state values change and the component rerenders, each of the useStates simply read values from the respective positions of the hooks manager array and create the state variables in the new render.
- So now matter how many times the component rerenders and no matter how many times variables a, b and c are created, they always have the update state values within them and this way the component remembers state across rerenders.

When using other basic/advanced hooks, the concept is still the same. Internal information of all hook usages is maintained in a single array for that component instance.

7.3 Dos and Dont's of hooks?

Hooks internals are managed using arrays of a fixed size. If hooks are dynamically created a in a component or hooks are conditionally created in a component, their positions in the hooks manager Array are not consistent across renders and hooks will no longer work.



Rules of Hooks

Dynamically creating more hooks in any of the following ways goes against the usage of hooks. Let us look at the following rules more closely.

Use hooks in the top level

Always make sure that your hooks are used in the top level of the component like so.

```
function MyComponent(){
 // Top level
 // Not within an if-else block or loops
 // or other dynamic ways
 const [a, setA] = useState(1)
 ...
}
```

This ensures that the size of the hooks manager array is always the same in the component.

Do not use hooks within an if-else condition

Using a hook within in an if-else condition like so, the useState for b is only called when that condition matches and this makes the size of the hooks array to become 3 instead of 2. This distorts the hooks

data flow and hence it should be avoided.

```
const [a, setA] = useState(5);

function increment() {
 setA(a++);
}
if (myStateValue === 5) {
 // This runs when myStateValue is 5
 const [b, setB] = useState(6);
}

const [c, setC] = useState(7);
```

Do not use hooks within dynamic loops

Using hooks within loops also has the same disadvantages as above. If the number of times the loop runs is inconsistent the hooks data flow is broken.

Do not use one hook within another.

Some hooks like useEffect can run functions. Using a hook within that function like so, should also be avoided. This is because the number of times a function can run is dynamic and the size of the hooks array will keep changing.

```
useEffect(() => {
 const [a, setA] = useState();
});
```

Being safe with hooks

As long as hooks are initialized in a non-dynamic way in the top level within a component and not within blocks or loops, we are good to use them freely.

To sum up:

- Use hooks within the top level of the component
- You can use multiple hooks within the same component and there is no limit

Finally, let us talk about an incredibly useful feature of hooks, custom hooks.

7.4 Custom hooks

Hooks are simply javascript functions which hold relevance only within React components. However, hooks are still functions, which means that they can be composed together. Custom hooks are hooks built by combining one or more hooks together.

Consider this example where we are interested in computing the square value of a state variable.

```
const [num1, setNum1] = useState(5);
const num1Sq = num1 * num1;
```

It can be useful to create hook to manage square values in a state. A custom hook can be created like so.

```
// A custom hook that manages the square of a value
function useSquareValue(initialValue) {
 const [value, setValue] = useState(initialValue);
 return [value * value, setValue];
}
const [num1Sq, setNum1] = useSquareValue(5);
const [num2Sq, setNum2] = useSquareValue(7);
// Calling setNum1(6) will make squareValue 36 because it
// is still a state modifier function and it triggers a rerender
```

Note: All custom hooks should start with the word use to make it easier for linters to catch mistakes in usage. eslint with hooks linting plugins catch incorrect usages easily when this pattern is followed.

Custom hooks are an amazing of modularizing hooks logic and to reuse them. They have all the benefits of functions while they are used within React components. When used carefully, they are extremely powerful tools. Let us apply what we learned so far to create custom hooks in our app!

7.5 useInputState

We are using an input element in our FilterJobs component. Let us create a hook the can manage the state of an input element. Let us organize all our hooks in the src/hooks folder. Create a file src/hooks/useInputState and add the following code to it.

```
// src/hooks/useInputState.js
export default function useInputState(initialValue) {
 const [value, setValue] = useState(initialValue);

 function onChange(event) {
  setValue(event.target.value);
 }

 return {
  value,
  onChange
 };
}
```

A lot of the code above was used in our FilterJobs and App component. We have organized it into a custom hook called useInputState. This hook returns an object containing value and the onChange function directly instead of the setValue function to make it easy to work with input components.

This can now be used with any input field like so.

```
const inputState = useInputState("");
return <input {...inputState} />;
// This <element {...object}/> syntax is a short way of doing
// <element a={object.a} b={object.b} .../>
// inputs require a `value` and `onChange`. Since `inputState` contains
// them, it can simply be used like so
// <input {...inputState}/>
```

Let us go to our components and start refactoring them. FilterJobs requires both the value and onChange values for the input element it is rendering, hence we pass the entire searchTextInputState. JobsList component only requires the searchText so we just pass searchTextInputState.value.

```
// src/index.js
import useInputState from './hooks/useInputState';

function App() {
 const searchTextInputState = useInputState("");
 return ...
  <FilterJobs
    searchTextInputState={searchTextInputState}
    ...
  />
  <JobsList searchText={searchTextInputState.value} .../>
```

```
  ...;
}
```

Finally, within our FitlerJobs component, we access searchTextInputState and spread it on the input element like so.

//src/components/FilterJobs.js

```
function FilterJobs(props) {
 const {
  searchTextInputState,
  showOnlyFeaturedJobs,
  setShowOnlyFeaturedJobs,
  showOnlyRemoteJobs,
  setShowOnlyRemoteJobs
 } = props;

 ...
  <input className="input" {...searchTextInputState} />;
 ...
}
```

Thats it! That was our first custom hook! It is time to make more. Let us create a useCheckboxState hook similarly and update our code like so.

```
// src/hooks/useCheckboxState.js
export default function useCheckboxState(initialValue) {
 const [checked, setChecked] = useState(initialValue);

 function onChange(event) {
  setChecked(event.target.checked);
 }

 return {
  checked,
  onChange
 };
}
```

```
//src/index.js
import useInputState from "./hooks/useInputState";
import useCheckboxState from "./hooks/useCheckboxState";
function App() {
 const searchTextInputState = useInputState("");
 const showOnlyFeaturedCheckboxState = useCheckboxState(false);
 const showOnlyRemoteCheckboxState = useCheckboxState(false);

 return (
  <div>
   <FilterJobs
    searchTextInputState={searchTextInputState}
    showOnlyFeaturedCheckboxState={showOnlyFeaturedCheckboxState}
    showOnlyRemoteCheckboxState={showOnlyRemoteCheckboxState}
   />
   <JobsList
    searchText={searchTextInputState.value}
    showOnlyFeaturedJobs={showOnlyFeaturedCheckboxState.checked}
    showOnlyRemoteJobs={showOnlyRemoteCheckboxState.checked}
   />
  </div>
 );
}
```

//src/components/FilterJobs.js

```
function FilterJobs(props) {
 const {
  searchTextInputState,
  showOnlyFeaturedCheckboxState,
  showOnlyRemoteCheckboxState
 } = props;

  ...
  <input className="input" {...searchTextInputState} />;
  ...
  <input
    id="featured"
    type="checkbox"
    {...showOnlyFeaturedCheckboxState}
  />
  ...
```

```
  <input
    id="remote"
    type="checkbox"
    {...showOnlyRemoteCheckboxState}
  />
  ...
}
```

Custom hooks are a joy to work with because now we can package multiple hooks together to create all sorts of behaviours and package them on to npm or any component registry and anyone can simply install our hook to start using them without worrying about implementation details. Many custom hooks like useList, useMap, useLocalstorage, useWindowResize etc are already very popular on npm because they help our components get supercharged with very minimal code.

Great going! In the next chapter, let's talk about the second basic hook, useEffect.

8. useEffect

useEffect, like the name suggests, is a hook that can run a function(called an effect) each time after the component has rendered.

Some example effects are

- interacting with the DOM environment like updating the document title,
- adding event listeners,
- updating the localStorage or
- fetching data using fetch.

An effect is still a function, so state variables can also be updated within it.

8.1 Usage

Like useState, useEffect can be imported from React and an example usage looks like so.

```
import React, { useEffect } from "react";
```

```
useEffect(callback)
```

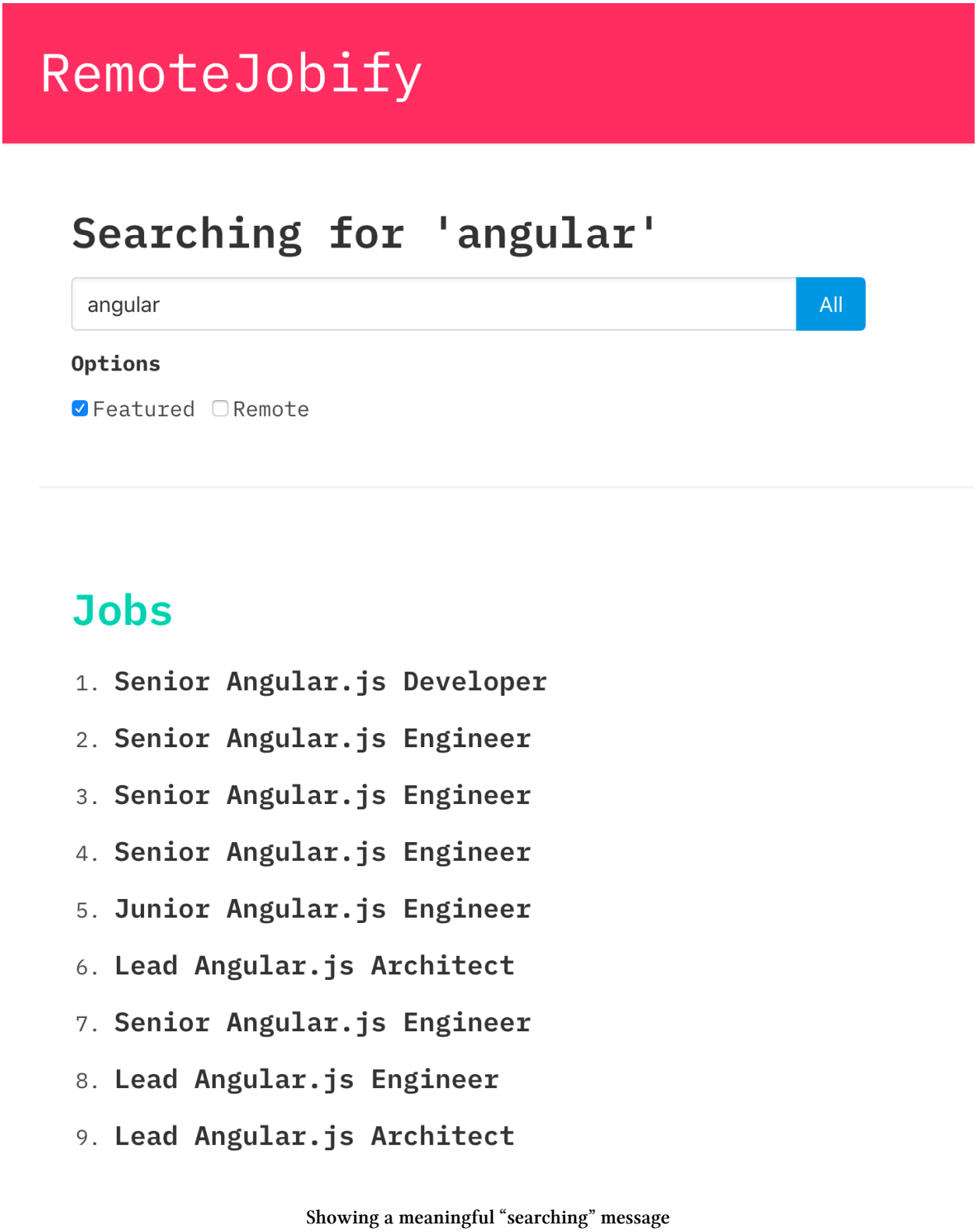
useEffect-1-arg

8.2 Updating document title

We want our FilterJobscomponent to show a meaningful search message, instead of simply saying "Search Jobs", so let us replace our heading with searchMessage variable that shows the search string in case there is a search string or the default message "Search Jobs".

And we can use the useEffect hook to update the document title after each render and we can set that to searchMessage.

```
//src/components/FilterJobs

function FilterJobs(props) {
 const {
  searchTextInputState,
  showOnlyFeaturedCheckboxState,
  showOnlyRemoteCheckboxState
 } = props;

 const searchMessage = searchTextInputState.value
  ? "Searching for " + searchTextInputState.value
  : "Search Jobs";

 useEffect(() => {
  document.title = searchMessage;
 });

 return (
  <section className="section filter-jobs">
   <h1>{searchMessage}</h1>
   <div>...</div>
  </section>
 );
}
```



Showing a meaningful “searching” message

8.3 Running effects selectively

useEffect is a versatile hook that can not only run after each render but it can selectively run an effect. The second argument of useEffect can be an array of values. useEffect is will track these values and only run the effect callback when one or all of these values have changed.

```
useEffect(callback, [value])
```

useEffect-2-arg-2

Our document title only depends on the searchMessage value. So let us specify that as our dependency and only run the effect when needed.

Note: useEvent hooks always run after the first render under irrespective of whether the dependencies are specified or not.

```
//src/components/FilterJobs

useEffect(() => {
 document.title = searchMessage;
}, [searchMessage]);
```

8.4 useDocumentTitle custom hook

Let us continue our exercise of creating as many custom hooks as possible. So let us create a useDocumentTitle hook that takes a text variable and renders that to the document title.

```
// src/hooks/useDocumentTitle
function useDocumentTitle(text) {
 useEffect(() => {
  document.title = text;
 }, [text]);
}
```

We can use this custom hook inside our filter FilterJobs component.

9. Building a SelectInput component

We're going to speed things up a little in this chapter. So let us use the knowledge we have gained in the last few chapters and build a SelectInput component.

The responsibilities of this component are

- to work as an input component. So it should accept a value and onChange property
- to display list of options and select an option when clicked and call onChange
- to automatically close itself when it is open and a click happens outside the SelectInput component.

1. Let us create a state variable and a toggle function for it's dropdown open and close behaviour.

```
const [areOptionsVisible, setAreOptionsVisible] = useState(false);

function toggleAreOptionsVisible() {
 if (areOptionsVisible) {
  setAreOptionsVisible(false);
 } else {
  setAreOptionsVisible(true);
 }
}
```

1. Let us create a changeOption function that calls onChange function with the clicked option value. We can use a data property like data-option on the paragraph element which renders the option and within the click event, we can grab the option value using event.target.dataset.option.

```
function changeOption(event) {
 onChange(event.target.dataset.option);
 setAreOptionsVisible(false);
}

const optionsMenu = (
 <div className="dropdown-menu" id="dropdown-menu" role="menu">
  <div className="dropdown-content">
   {options.map(option => (
    <p
     data-option={option}
     className="dropdown-item"
     onClick={changeOption}
     key={option}
    >
     {option}
    </p>
   ))}
  </div>
 </div>
);
```

9.1 The useOutsideClickRef hook

1. Detecting a click outside an element is not very trivial with React. We can use click handlers on react elements. But to handle a click event on all elements except an element and all its children is not very straightforward with JSX. So the solution is to use a useEffect and attach event handlers within the useEffect and compare the event target node with the DOM node of our element.

React has a handy hook called useRef which gives us the reference to the actual DOM node that is rendered for a react element. We can use it like this.

```
const ref = useRef()
// After component renders the first time
// ref.current will contain the actual DOM
// node that this element was rendered in

<p ref={ref}> </p>
```

9.2 More about useEffect

We know that useEffect we can interact with the document and the other APIs in the environment. It means that we can also add event listeners on the document object dynamically.

And because useEffect can take conditions, we can add this event listener conditionally. The final piece of the puzzle is that useEffect also has a cleanup phase.

```
useEffect(() => {
 // do something
 return () => {
  // cleanup
  // useful for cleaning up event handlers
 };
}, [conditions]);
```

If the callback function of useEffect returns a function, that function will be called as a cleanup mechanism. It runs before a new effect runs allowing us to remove event handlers.

Now we can use this to add an event listener when the dropdown is open and remove it when the dropdown is closed.

```
useEffect(() => {
 function handler(event) {}
 // add event handler
 document.addEventListener("click", handler);
 return () => {
  // cleanup => remove event handlers
  document.removeEventListener("click", handler);
 };
}, [...]);
```

Keep callbacks fresh with useRef

The useRef returns an object that is persisted across multiple renders. This object's current property can be modified without rerenders and is good for bookkeeping and to ensure that the newest callback is always maintained in the ref so we can use this. Within a useEffect the newest callback can be assigned to savedCallback.current.

```
// create a ref object with callback
const savedCallback = useRef(callback);
// after each render update the reference
// to the callback
useEffect(() => {
 savedCallback.current = callback;
});
```

A handler function can send be as event listener because the handler function was created when the savedCallback was also in scope, it means we can access the savedCallback object and because savedCallback.current is always updated the latest callback is always called.

We want the useOutsideClickRef effect to run only when a condition is true (for eg: look for clicks on document,when the dropdown is open). So let us call that variable as when. And within our use effect, if when is true, then we will add an event listener. And pass in the handler, which will track if the node contains the event target.

Recap

So while this appears to be very complicated is actually quite simple if you break it down into parts,

- so we have a ref to track the latest callback function and not allow it to go stale
- we also have to track the DOM node outside of which, click events need to be tracked
- we have an effect that adds an event listener without creating harmful closure and checks if the event target is within the node or not.

So now let us save this in src/hooks/useOutsideClickRef.js and wrap up our SelectInput component.

```
//src/hooks/useOutsideClickRef.js

import { useEffect, useRef } from "react";

function useOutsideClickRef(callback, when) {
 const savedCallback = useRef(callback);
 const ref = useRef();

 useEffect(() => {
  savedCallback.current = callback;
 });

 useEffect(() => {
  if (when) {
```

```
    function handler(event) {
     if (ref.current && !ref.current.contains(event.target)) {
      // if node doesn't contain event target
      savedCallback.current();
     }
    }
    document.addEventListener("click", handler);
    return () => {
     document.removeEventListener("click", handler);
    };
   }
 }, [when]);

 return ref;
}

export default useOutsideClickRef;
```

Finally, let's bring all of this together and create our SelectInput component like so. We can import our useOutsideClickRef and pass a callback that will close the options in the callback.

```
//src/components/SelectInput.js
import React, { useState, useEffect } from "react";

export default function SelectInput(props) {
 const { value, options, onChange } = props;

 // Options visibility management
 const [areOptionsVisible, setAreOptionsVisible] = useState(false);

 function toggleAreOptionsVisible() {
  if (areOptionsVisible) {
   setAreOptionsVisible(false);
  } else {
   setAreOptionsVisible(true);
  }
 }

 function changeOption(event) {
  onChange(event.target.dataset.option);
```

```
  setAreOptionsVisible(false);
 }

 const ref = useOutsideClickRef(() => {
  setAreOptionsVisible(false);
 }, areOptionsVisible);

 return (
  <div className="dropdown is-active">
   <div className="dropdown-trigger " onClick={toggleAreOptionsVisible}>
    <button
     className="button is-info"
     aria-haspopup="true"
     aria-controls="dropdown-menu"
    >
     <span>{value}</span>
    </button>
   </div>
   {areOptionsVisible ? (
    <div className="dropdown-menu" ref={ref} id="dropdown-menu" role="menu">
     <div className="dropdown-content">
      {options.map(option => (
       <p
        data-option={option}
        className="dropdown-item"
        onClick={changeOption}
        key={option}
       >
        {option}
       </p>
      ))}
     </div>
    </div>
   ) : null}
  </div>
 );
}
```

9.3 Updating components

Now, let's edit our components to filter verticals.

1. Add vertical state to App.js and pass necessary props to FilterJobs and JobsList.

```
//src/index.js
function App(){
  ...
  const [vertical, setVertical] = useState("All");

  return ...
      <FilterJobs
        vertical={vertical}
        setVertical={setVertical}
        ...
      />
      <JobsList vertical={vertical} .../>
      ...
}
```

1. Access vertical and setVertical props and render the SelectInput component in FilterJobs with valid props.

```
//src/components/FilterJobs

import SelectInput from './SelectInput'


// access vertical and setVertical among other props
const {vertical,setVertical,...} = props

<div className="field has-addons">
 <div className="control is-expanded">
  <input
   className="input"
   value={value}
   onChange={e => {
    const newValue = e.target.value;
```

```
      setValue(newValue);
    }}
   />
  </div>
  <div className="control is-expanded">
   <label>
    <SelectInput
     value={vertical}
     options={["All", "Frontend", "Backend"]}
     onChange={setVertical}
    />
   </label>
  </div>
</div>
```

1. Filter out jobs not matching a vertical and display filtered jobs to the screen.

```
//src/components/JobsList

function JobsList({
 value,
 showOnlyFeaturedJobs,
 showOnlyRemoteJobs,
 vertical
}) {
 let filteredJobs = jobs.filter(job => {
  return job.title.toLowerCase().includes(value.toLowerCase());
 });
 if (showOnlyFeaturedJobs) {
  filteredJobs = filteredJobs.filter(job => job.isFeatured);
 }
 if (showOnlyRemoteJobs) {
  filteredJobs = filteredJobs.filter(job => job.isRemote);
 }
 if (vertical && vertical !== "All") {
  filteredJobs = filteredJobs.filter(job => job.verticals.includes(vertical));
 }
 return <section>...</section>;
}
```

Save this and look at the screen and we should now see that our select input component is working flawlessly.

SelectInput component

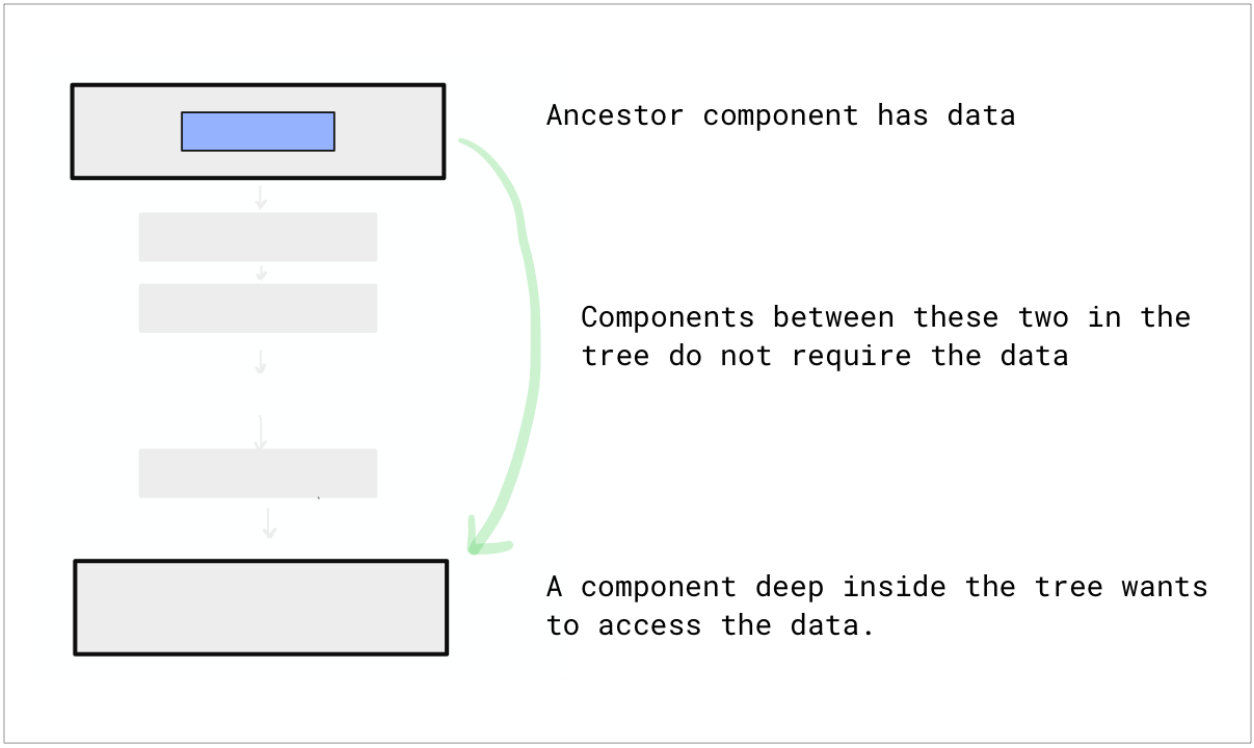
10. Context

In our project, we want to highlight the job that is clicked and also display information about the job in a JobDrawerComponent to the right. This component will be used in the `App` component so sharing the `job` data with it might be difficult for a `Job` component instance using just props.

Sometimes it so happens that a component is deep inside hierarchy and we want to share its data with a component elements much further away from it.

Consider components in these following scenarios:

- An ancestor-descendant pair of components when there are far too many components between the two components
- Components that are in different subtrees.



React Context

10.1 Why Context

Props can be extremely difficult to use in such scenarios because:

- Components that are between the two components in question don't require the data as props. But sending them props needlessly and forwarding them from within the component declaration adds noise to the code and makes all the components tightly bound to each other.
- Sometimes the data to be shared might be created very close to the sender component and moving this data to the top might be difficult.

In such scenarios, Context comes to our rescue. React context is a subtle way of sharing data between one component to all the components in the sub tree.

Let's create a context object to hold a job.

```
//src/SelectedJobContext.js

import { createContext } from "react";

const SelectedJobContext = createContext(null);

export default SelectedJobContext;
```

The context object has a key called Provider, which is a component and this component can be rendered like any other component with a value property. And this value will now be available for all the components in the sub tree.

```
//src/index.js
...
import SelectedJobContext from "./SelectedJobContext";

function App(){
  ...
  const [selectedJob, setSelectedJob] = useState(null);
  ...
  const selectedJobContextValue = {
    selectedJob,
    setSelectedJob
  }
  return <SelectedJobContext.Provider value={selectedJobContextValue}>
    <div>
   <FilterJobs
    vertical={vertical}
    setVertical={setVertical}
    searchTextInputState={searchTextInputState}
    showOnlyFeaturedCheckboxState={showOnlyFeaturedCheckboxState}
    showOnlyRemoteCheckboxState={showOnlyRemoteCheckboxState}
```

```
    />
    <JobsList
     vertical={vertical}
     searchText={searchTextInputState.value}
     showOnlyFeaturedJobs={showOnlyFeaturedCheckboxState.checked}
     showOnlyRemoteJobs={showOnlyRemoteCheckboxState.checked}
    />
   </div>
  </SelectedJobContext.Provider>
}
```

A component that requires this context can simply subscribe to this context using the useContext hook. Let's use this hook in our Job component.

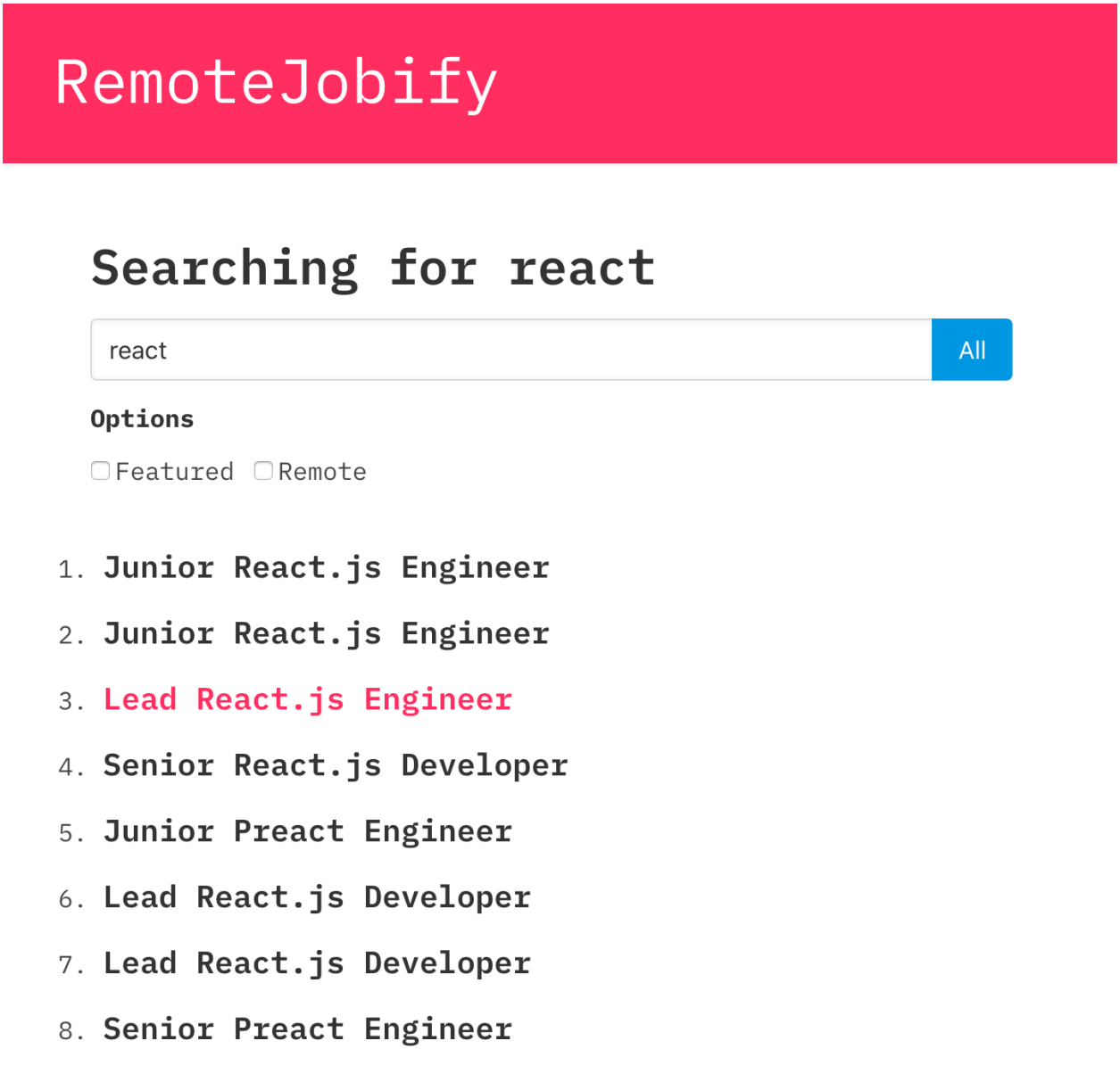
Finally within the job component we can request the context by using the useContext hook. And we can now access the selectedJobContextValue inside the job component. Finally when let's add a click handler and set the selectedJob value.

Let's also add a className to the selected job to visually identify it by comparing the job the component instance has in its props to the job it received from context.

Now if you preview this in the screen, we should see that whenever a job is clicked it is highlighted to red.

```
// src/components/Job.js

import React, { useContext, useState } from "react";

function Job(props) {
 const { job } = props;
 const { title, description } = job;
 const [isDetail, setIsDetail] = useState(false);
 const selectedJobContextValue = useContext(SelectedJobContext);

 function handleClick() {
  if (selectedJobContextValue.selectedJob) {
   selectedJobContextValue.setSelectedJob(null);
  } else {
   selectedJobContextValue.setSelectedJob(job);
  }
 }

 const isJobSelected =
  selectedJobContextValue.selectedJob &&
```

```
  selectedJobContextValue.selectedJob.id === job.id;

 return (
  <li>
   <article className="media job">
    <div className="media-content">
     <h4
      onClick={handleClick}
      className={isJobSelected ? "has-text-danger" : ""}
     >
      {title}
     </h4>
    </div>
   </article>
  </li>
 );
}

export default Job;
```



Selecting a Job item

10.2 JobDrawer component

Finally let us create a `JobDrawer` component that also requires the same context. Its responsibilities are

- to automatically open when the `SelectedJobContext` value is changes.

- to automatically close when it is open and a click happens outside it

We can use a state variable to track if the JobDrawer instance is open and we can use useOutsideClickRef to automatically close if clicked outside.

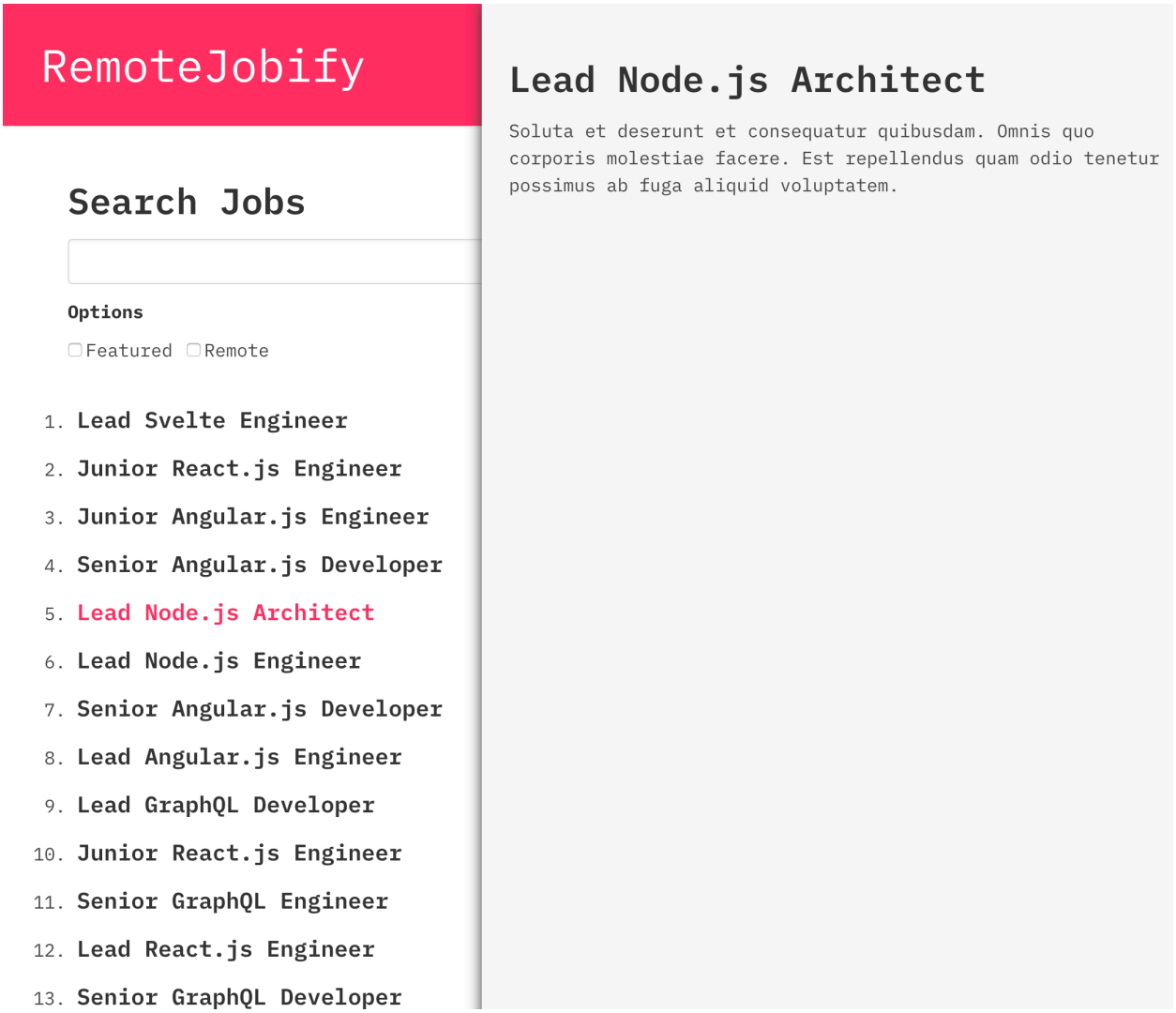
And let us create a useEffect which will automatically opens the JobDrawer when the selectedJobContextValue.selectedJob changes.

```
//src/components/JobDrawer

import React, { useContext, useRef, useState, useEffect } from "react";
import SelectedJobContext from "../SelectedJobContext";
import useOutsideClickRef from "../hooks/useOutsideClickRef";

export default function JobDrawer(props) {
 const selectedJobContextValue = useContext(SelectedJobContext);
 const [isOpen, setIsOpen] = useState(false);

 const ref = useOutsideClickRef(() => {
  setIsOpen(false);
 }, isOpen);

 useEffect(() => {
  if (selectedJobContextValue.selectedJob) {
   setIsOpen(true);
  }
 }, [selectedJobContextValue.selectedJob]);

 const { selectedJob } = selectedJobContextValue;

 return (
  <div
   className={
    isOpen
     ? "drawer has-background-light has-shadow open"
     : "drawer has-background-light has-shadow "
   }
   ref={ref}
  >
   <section className="section">
    <div className="content">
     <h1>{selectedJob ? selectedJob.title : ""}</h1>
     <p>{selectedJob ? selectedJob.description : ""}</p>
```

```
      </div>
    </section>
  </div>
 );
}
```

Now finally let's add our JobDrawer component to our App component and let us take a look at the screen.

```
//src/index.js
...
import SelectedJobContext from "./SelectedJobContext";
import JobDrawer from "./components/JobDrawer";

function App(){
  ...
  const [selectedJob, setSelectedJob] = useState(null);
  ...
  const selectedJobContextValue = {
    selectedJob,
    setSelectedJob
  }
  return <SelectedJobContext.Provider value={selectedJobContextValue}>
    <div>
   <FilterJobs
    vertical={vertical}
    setVertical={setVertical}
    searchTextInputState={searchTextInputState}
    showOnlyFeaturedCheckboxState={showOnlyFeaturedCheckboxState}
    showOnlyRemoteCheckboxState={showOnlyRemoteCheckboxState}
   />
   <JobsList
    vertical={vertical}
    searchText={searchTextInputState.value}
    showOnlyFeaturedJobs={showOnlyFeaturedCheckboxState.checked}
    showOnlyRemoteJobs={showOnlyRemoteCheckboxState.checked}
   />
   <JobDrawer />
  </div>
  </SelectedJobContext.Provider>
}
```



JobDrawer component

11. Asynchronous data fetching using hooks

When we are building real-world applications the data that we need for our application is generally present in a server. To access the data from that server we need to make an asynchronous API request to the server and fetch the data.

fetch is a modern approach of fetching data from the server. Let us see how we can create a simple use effect to fetch the data from the server.

Let us create an AppWrapper component, that will conditionally render the app component only when the data from the server has been fetched. To denote whether to store the data fetched from the server, let us create the jobs state variable.

```
const [jobs, setJobs] = useState(null);

if (jobs) {
 return <App jobs={jobs} />;
} else {
 return null;
}
```

Now, we will create a use effect that will fetch data from the server and once the data is successfully fetched it will update the jobs state variable.

11.1 Fetch API

The fetch API returns a promise. Once the promise resolves with the response object, it can be converted to JSON using response.json function, which also returns a promise.

An easier way of dealing with promises is to use async functions.

11.2 Fetch data before rendering App

We only need to fetch our jobs data from the server once. To run an effect only once all we need to do is to pass in the second parameter as an empty array. This will ensure that the use effect callback only runs once. Once the jobs data is set, we can render our App component.

```
function AppWrapper() {
 //src/index.js
 const [jobs, setJobs] = useState(null);

 useEffect(() => {
  async function loadJobs() {
   const response = await fetch(
    "https://delightful-react-assets.imbhargav5.com/public/jobs-final.json"
   );
   const jobsJson = await response.json();
   setJobs(jobsJson);
  }

  loadJobs();
 }, []);

 if (!jobs) {
  return null;
 } else {
  return <App jobs={jobs} />;
 }
}


// src/index.js
function App(props) {
 const { jobs } = props;

 <JobsList
  jobs={jobs}
  vertical={vertical}
  searchText={searchTextInputState.value}
  showOnlyFeaturedJobs={showOnlyFeaturedCheckboxState.checked}
  showOnlyRemoteJobs={showOnlyRemoteCheckboxState.checked}
 />;
}
```

Finally, we need to pass in the `jobs` state variable to the app component once it is loaded. And within the App component, we will need to pass the `jobs` variable to `JobsList` component.

```
// src/components/JobsList.js

function JobsList(props){
  ...
  const {jobs} = props;
  ...
}
```

Once that is done, we can now render the `AppWrapper` component to the DOM using `ReactDOM.render`

```
// before
// ReactDOM.render(<App />, mountNode);

// after
ReactDOM.render(<AppWrapper />, mountNode);
```

Awesome. So we have finished building our app for this course. We have covered almost every topic there is within React basics to understand how we can create React app using the modern syntax today. In the next chapter let us talk about deployment and wrap with some closing thoughts.

12. Closing thoughts

12.1 Deploying our app to production

We have built a fantastic app, and now, let us see how we can deploy our app to production. Since our project was built using create-react-app, it comes with a build script that generates a static build from for our website.

From the root of our project, run

```
npm run build
```

The build directory contains a static build of our project, and that can be with hosting providers like now.sh or Netlify or Github pages to deploy our website to production.

12.2 Next Steps

The Delightful React is an excellent book to get started with the fundamentals of React quickly. To become an advanced React programmer, there are a few topics that might interest you

- styling components using frameworks like styled-components or emotion
- Data management solutions like redux and mobx
- GraphQL and Apollo
- server-side rendering
- and more

The list is endless. However, with sufficient planning and practice, I am sure you can achieve your goal of becoming an excellent React programmer. Let me know how it turns out, and if you have any feedback and would like to help me improve this book, feel free to approach me on twitter (@imbhargav5).

I hope you find great success in your endeavors.

Thank you.