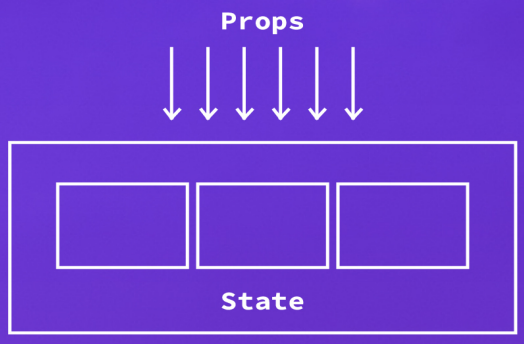
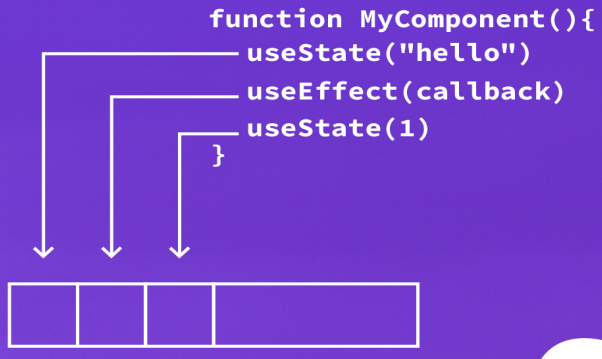
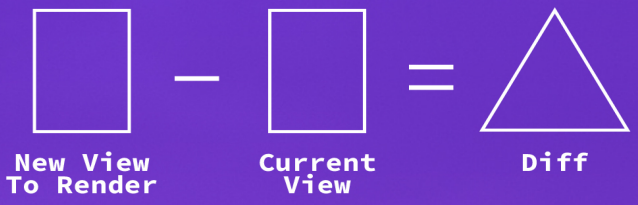
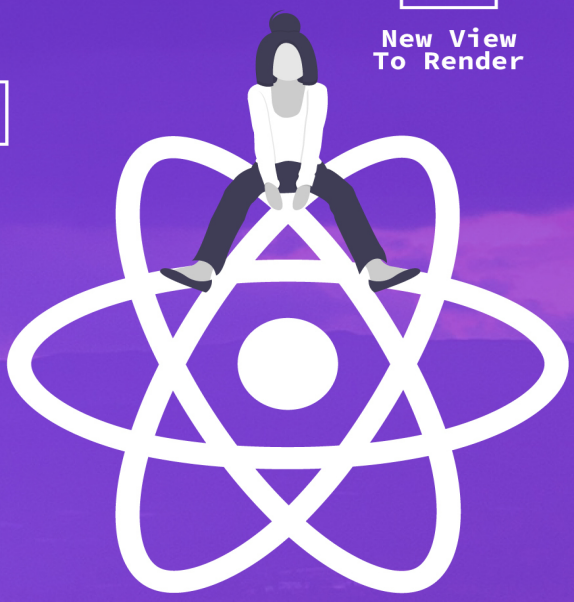
DELIGHTFUL REACT

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 2019-11-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

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.

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>
```

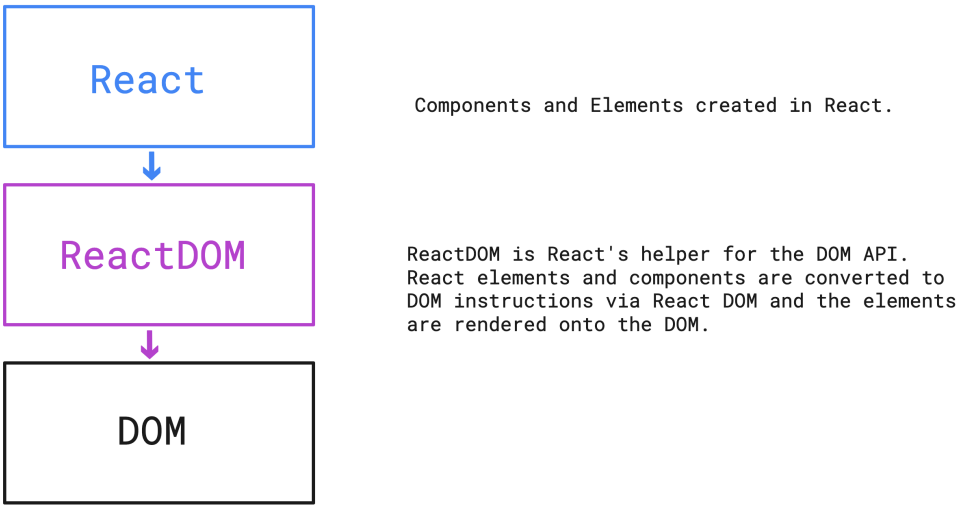
If we open this HTML file in a browser we should see the text `Hello world` on the screen.

React in Plain HTML

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 Element

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.

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
```

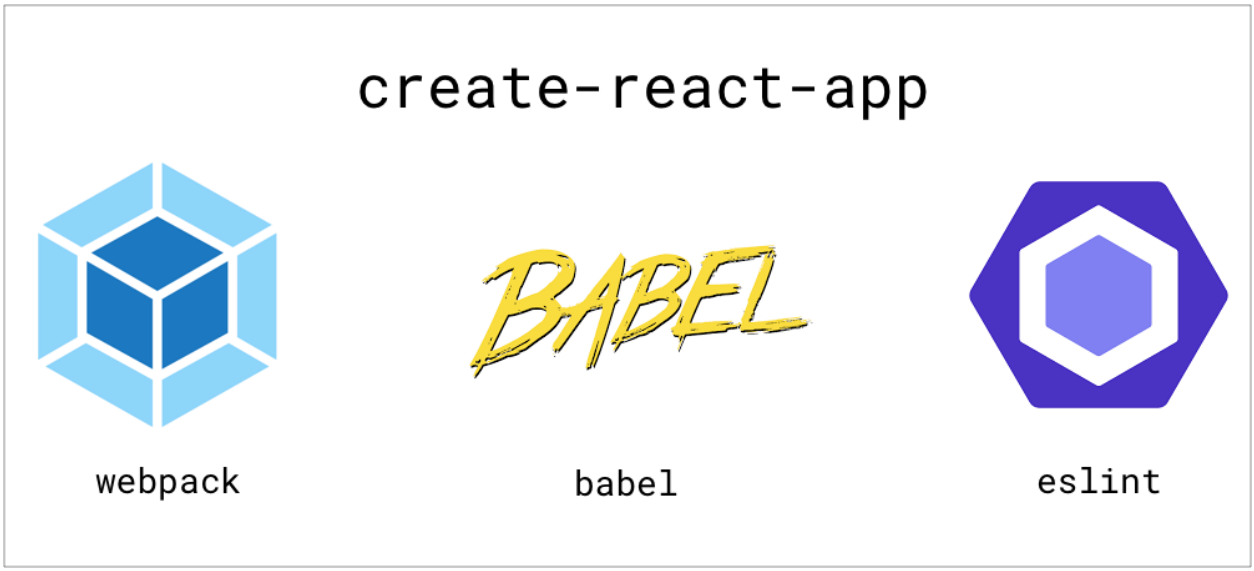
This command starts a project and runs it on port 3000. Now, open your browser and open `http://localhost:3000`, you should see our project running with this response.

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.



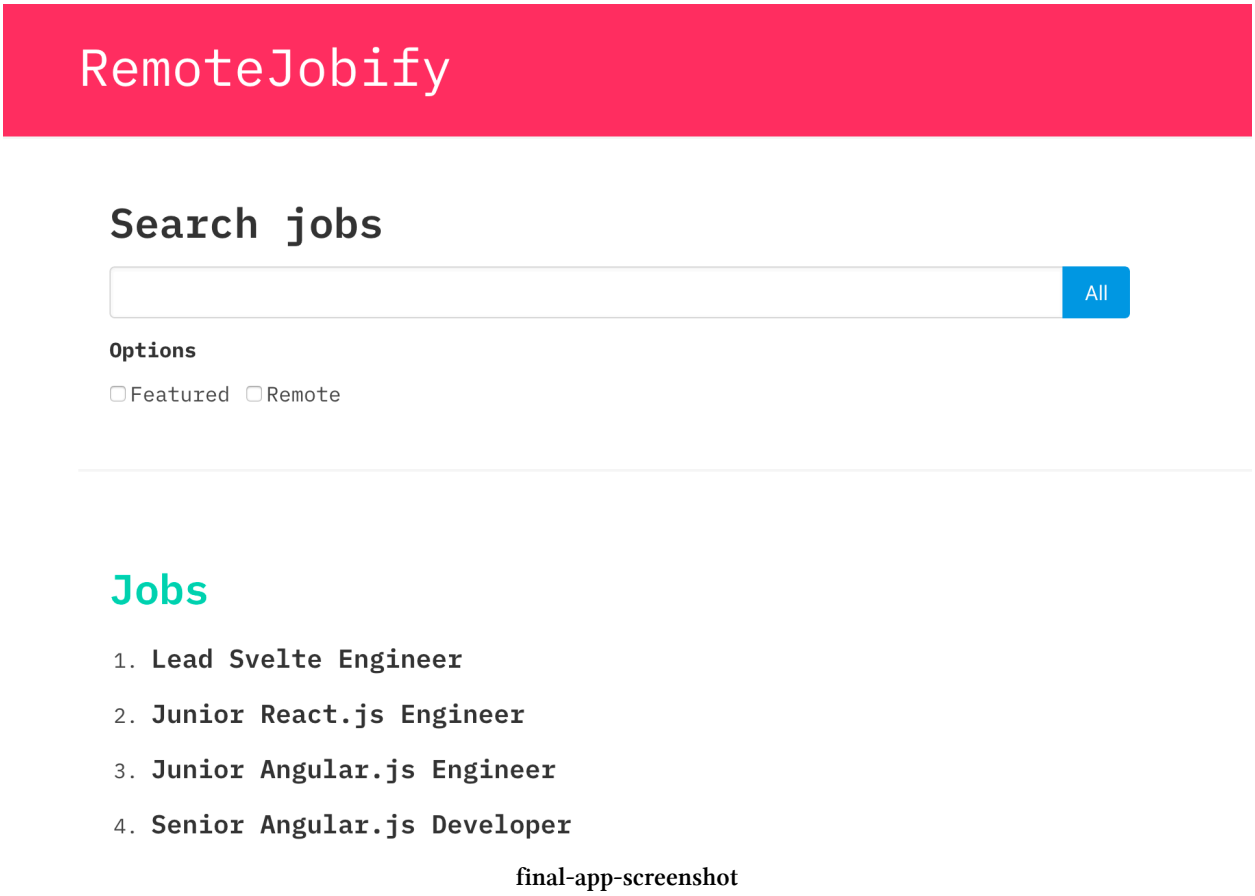
create-react-app

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.



final-app-screenshot

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.

The JSX Syntax

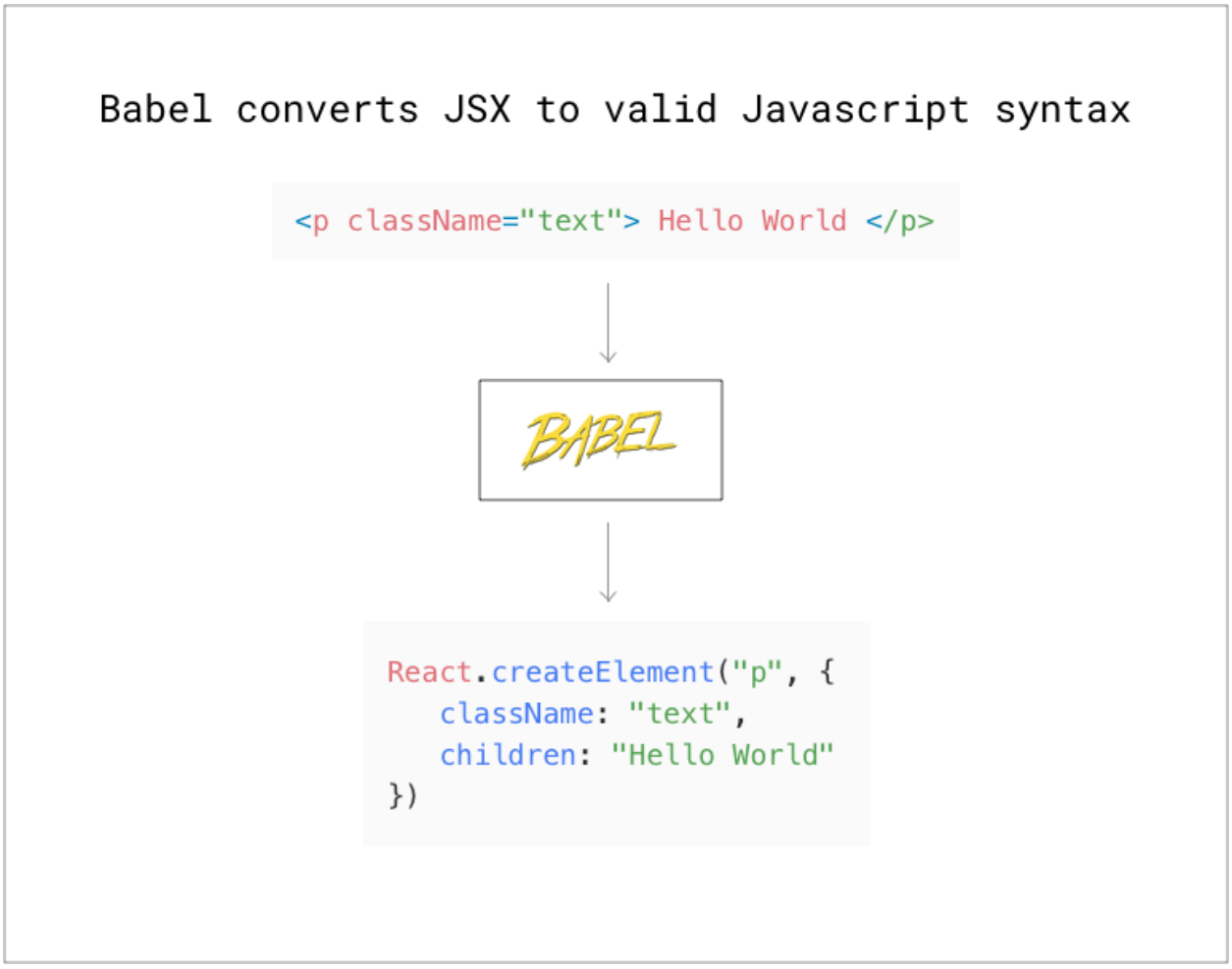
The React code written in plain javascript is verbose. Let us write this snippet.

```
//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 extermely similar to HTML to make creating and composing React Elements as easy as possible.



jsx-original

`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!

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.

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);
```

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>
```

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.

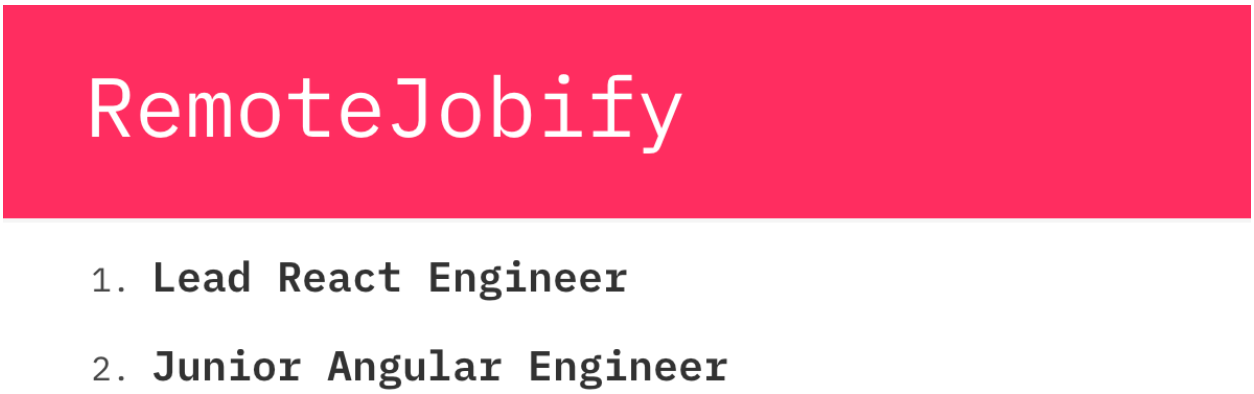
This way we can create an element for each value in the source array and render the resultant array within our React elements.

```
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.



chapter2-end-screenshot

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.

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?

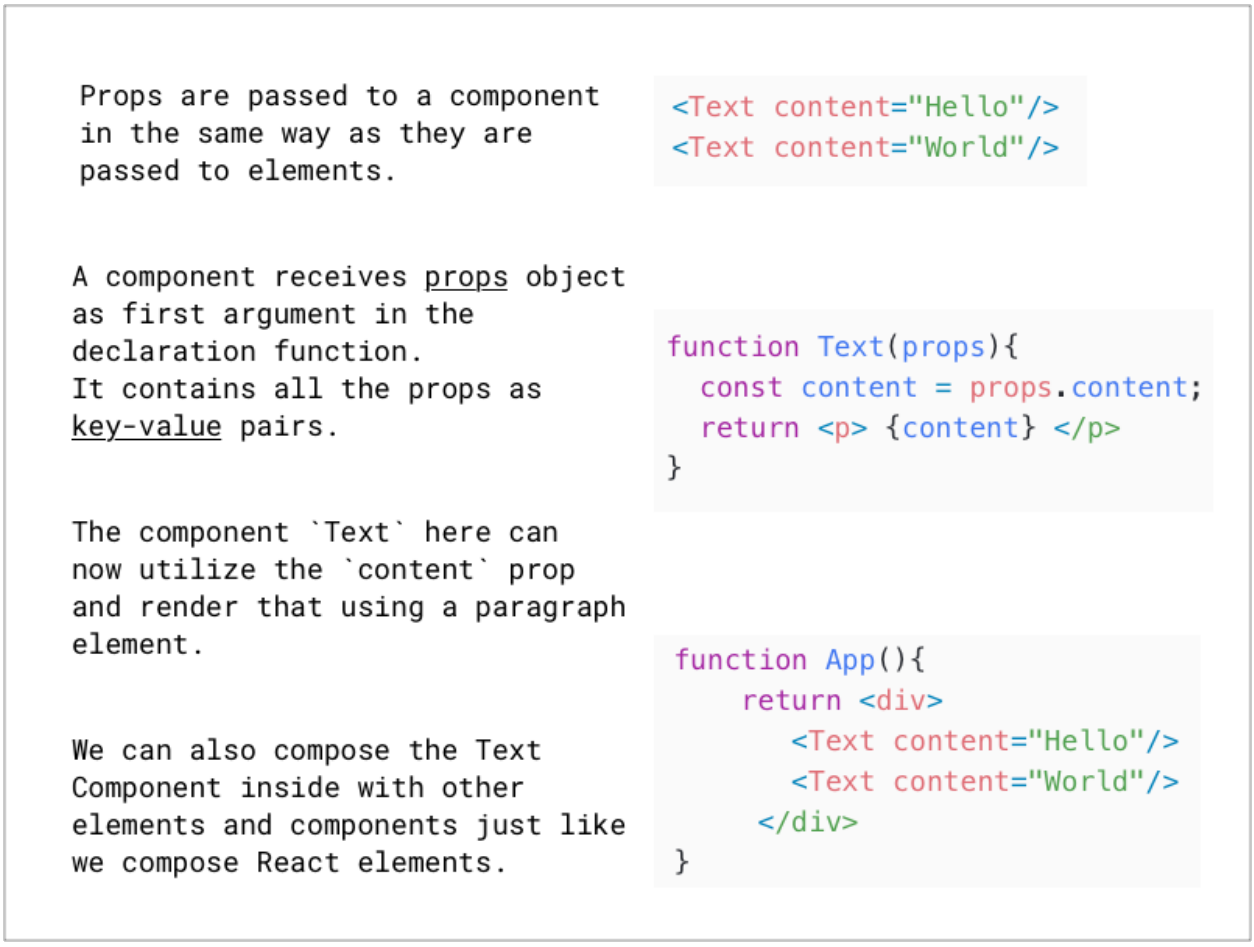
```
function Job() {
  const jobTitle = ???
  return (
    <li>
      <article className="media job">
        <div className="media-content">
          <h4>{jobTitle}</h4>
        </div>
      </article>
    </li>
  );
}
```

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} />
));
```

Props examples

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 simply 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} />);
```

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.

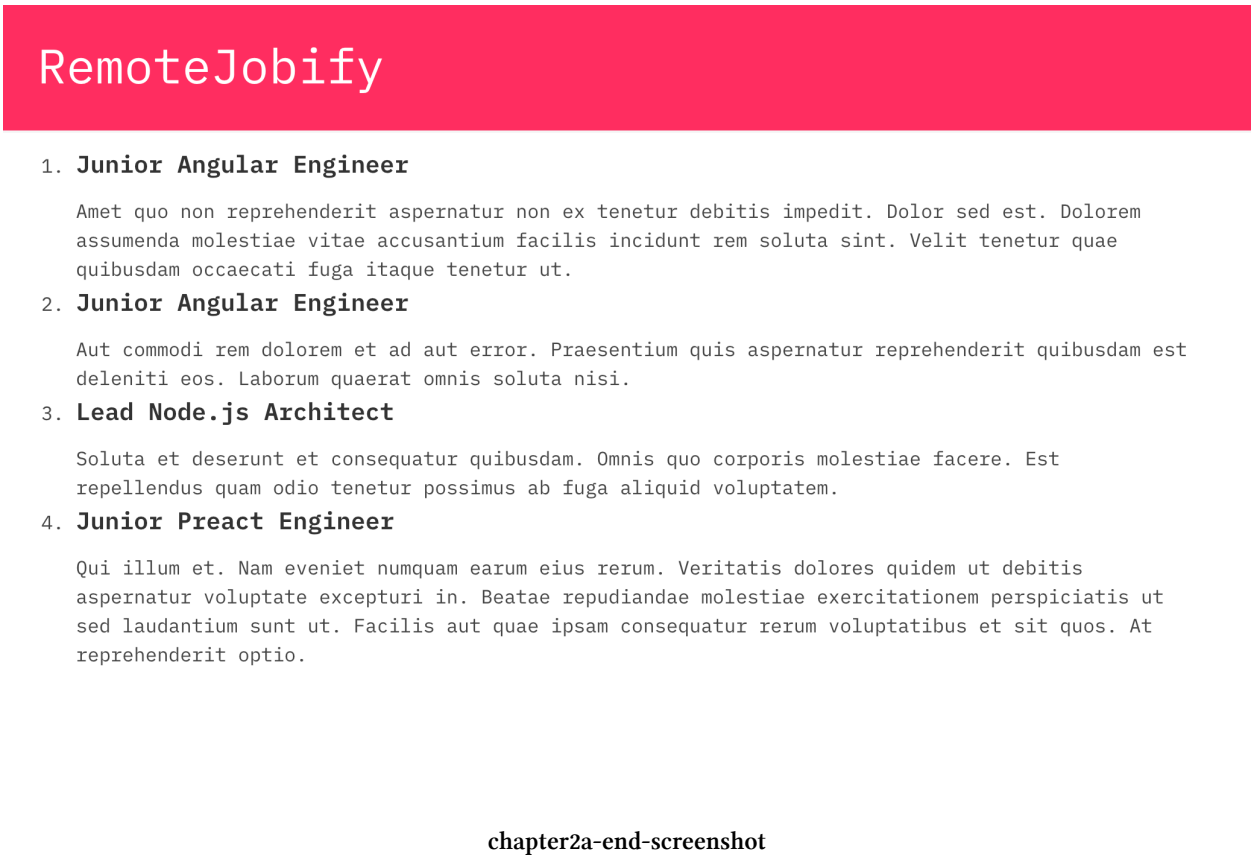
```
// src/data/jobs.js

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: "Junior 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);
```



chapter2a-end-screenshot

`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`.

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.

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.

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.

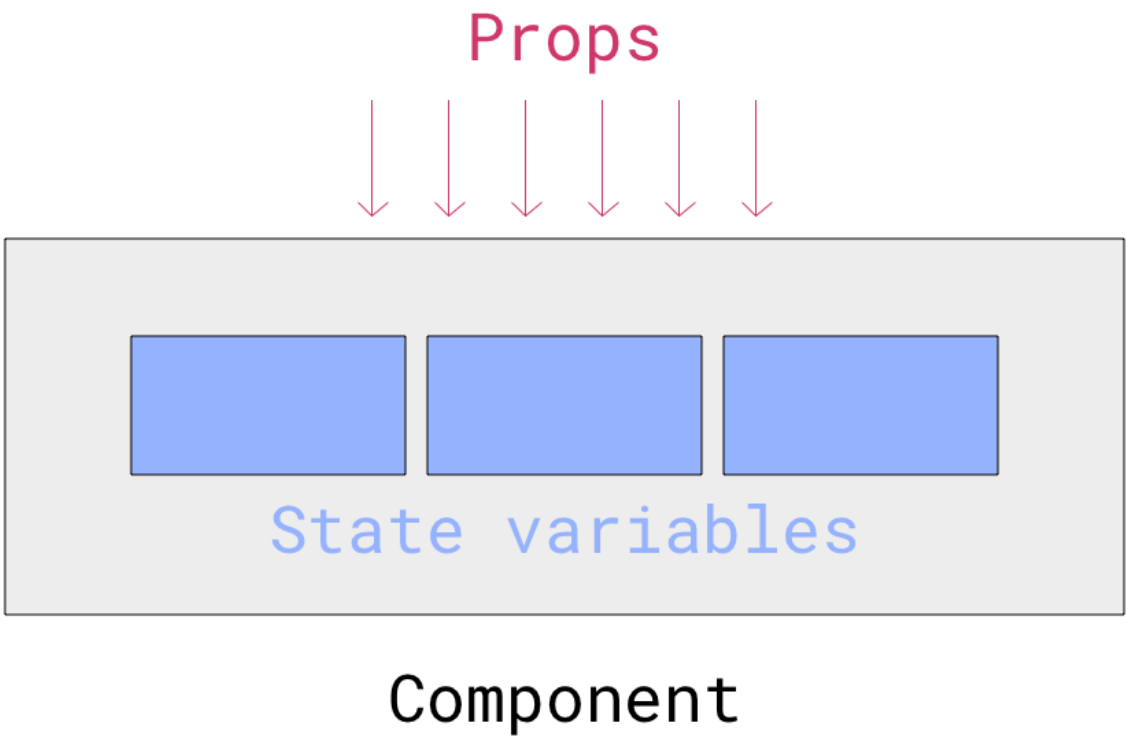
```
function handleClick() {
  setIsDetail(!isDetail);
  // if isDetail is false
  // setIsDetail(true) is called
  // and vice versa
}
```

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.*

Props and State

Option state are very essential for real world React applications. They often in fact work together.Props are sometimes used to initial state variables and state is sometimes sent as props to child components or to child elements. So let us take a look at the similarities and the differences between props and state with these two diagrams.



props-vs-state-2

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

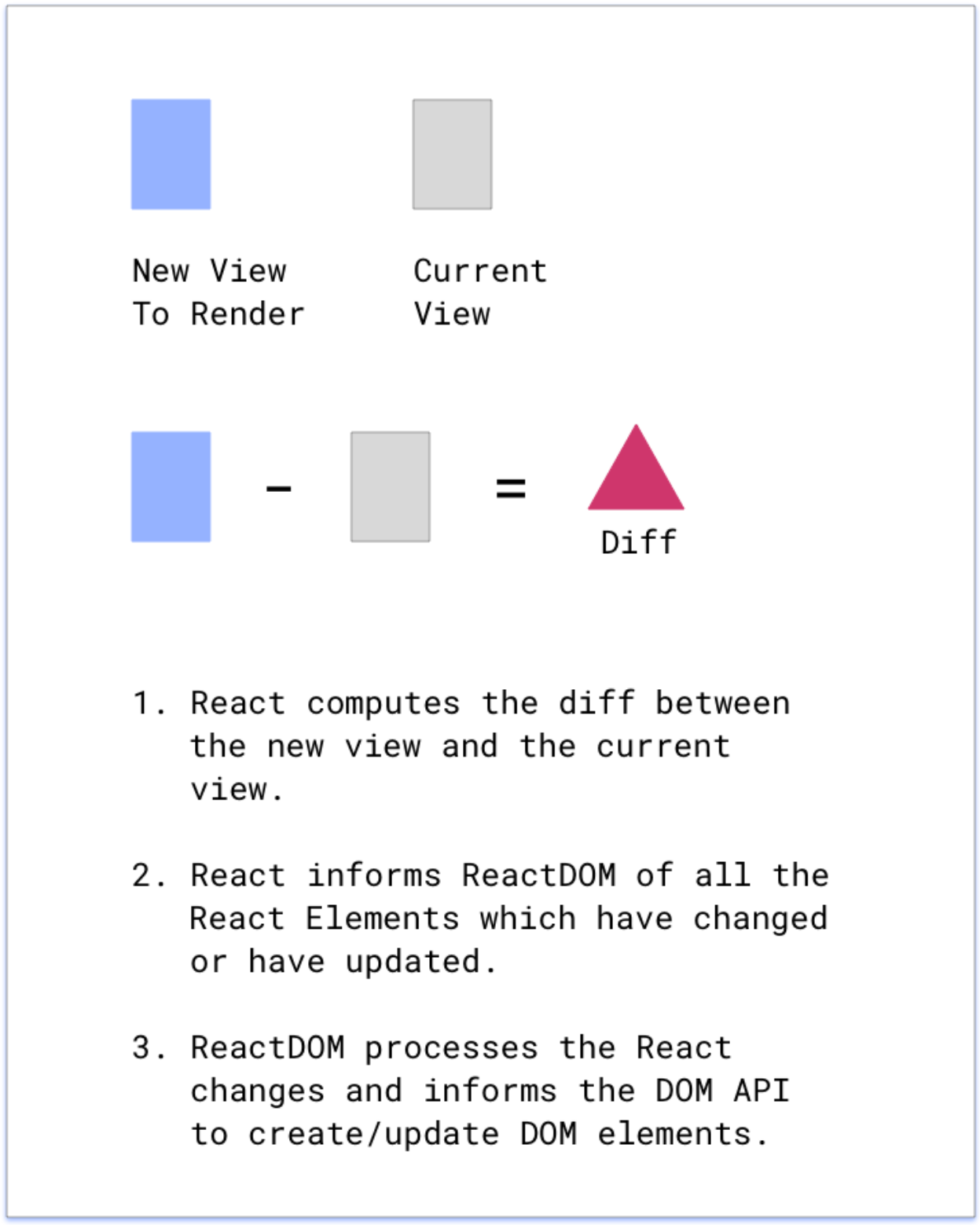
props-vs-state

The main difference is that props and state end of the component and all the children of that component when they change.Props are sent from outside the component and state is created internal to the component.

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.


New View
To Render
Current
View
-
=
Diff
1. React computes the diff between the new view and the current view.
2. React informs ReactDOM of all the React Elements which have changed or have updated.
3. ReactDOM processes the React changes and informs the DOM API to create/update DOM elements.

Diff

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.

Forms

```
<input />
// This is an uncontrolled input
// React cannot access it's value without
// accessing the DOM node


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

// This is a controlled input
// React manages the value of the input and
// only allows user to update it when the `value`
// prop changes. The onChange event can be used
// to update the value
<input value={value} onChange={handleChange} />;


// 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 rerenders onChange, hence it
// behaves like a normal input element
// except React state is always in sync
// with the input's value
<input value={value} onChange={handleChange} />;
```

```
function handleClick() {
  console.log("clicked");
}

function handleChange(event) {
  console.log("new input value", event.target.value);
}

<input onClick={handleClick} onChange={handleChange}>
  ...
</input>;
```

```
//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>
  );
}
```

```
//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>
  );
}
```

```
// 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>
  );
}
```

```
// src/data/jobs.js
// seniority values are "Lead", "Senior" and "Junior"
// verticals is an array of strings
// valid verticals are Frontend and Backend

 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,
    seniority: "Lead",
    verticals: ["Frontend"]
  },
  {
    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,
    seniority: "Junior",
    verticals: ["Frontend"]
  },
  ...
}]
```

https://gist.github.com/imbhargav5/0d5630f486c86cc95b5b2dc3990c5ceb

```
//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>
  );
}

//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>
  );
}


// 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 ...
}
```

Form Elements

Controlled Input

Filter Jobs

Props Data flow

Check box elements

Event handlers

Just like Dom HTML elements have. Class style and other attributes they also have even handlers similarly. Even handless like on-click on focus etc similarly. JSX elements also have corresponding react events that can track the HTML events inside the dom elements. So we have earlier seen a diagram where we where we understood how react-dom is a glue between react and dorm.

But it also acts as a recipient for the events from the DOM APA, and it sends them to the component that has rendered the HTML. Or the element that has rendered HTML, so this way, we can track which element it was that that created the HTML element, and we can react to that by updating the state or updating crops and so on.

So these events are called synthetic events, and they are react specific reacts internal implementation of events specific to react elements.

The. Important thing here to understand. Is that.

To keep it simple the the on click event looks like like this and html and it looks like this in camel case in in JSX so they they have a one is to one correspondence so on click event also receives the event attribute as the first argument inside the function so we can use that we would to determine which element or which sub element must click on and what is the target of the event and so on.

And we can also do the event even prop event stop propagation even prevent default and so on that we normally do with events. These work like normal JavaScript. Dom API events.

Form Elements

Where the most important elements in our webpage are form elements are inputs radio boxes checkboxes and select boxes that help the user add information into the website. Let us see how we can build a form element an input element using react. We are discussed how an input element when an element is rendered by react.

The elements props are managed by react, and the data to render it into the browser is sent by react to the dom via the react-dom api, and when there is an event in the dom, the dome sends the even details to react via react on. The same approach will be used for creating input elements as well.

An input element can simply be created like so just like any paragraph element or a heading element; an input element can simply be created using the input react element tag. The problem here is that when the value of the input element changes, they are not tracking the change in the value, so react has a unique way of handling these scenarios, so react stores the value of the current input inside.

A state variable and when the input value is is changed, the upgraded value is sent to the state variable via the modifier function. So let us see how that works.

Controlled Input

Controlled inputs are inputs whose values entirely managed to react. So, even so, when the value property of an input element is set, then even if the value even if the user starts typing into the field react does not update the value.

Reacts full control of the input element, and the only way to modify the input elements value is through the on change, even handler.

This ensures that the data flow into the input is one way, and the value is only stored in react and nowhere else, so there is only a single source of truth.

Filter Jobs

It would be nice if you are able to filter our job by text, so maybe filter all the jobs which match a certain substring. And list only the jobs that match the substrate. So let us create a job filter component, a filter jobs component that can handle this for us, and this filters of component will contain an input field that will track the value that user is trying to search.

So the filters of component let us create the component in the components file, and let's import them import that into our source index.js file. And it the code for the filter jobs component will look something like this. What we want to do now is we want this input value to be accessible inside the jobs component as well.

So that within the jobs array, we can filter out all the values that contain the substring. From this input and then only render the jobs that match substring, but how do we do that? Because the input value is present inside the filter jobs component, how can we send it to the jobs component?

So now, let us talk a little bit about a props and understand what how to how to make data flow from assembling to another sibling. So jobs and filter list components are both siblings. So in react props only for flow from the parent to the child and not vice versa so a parent can send information to a child.

But a child cannot send any information to the parent however so what we can do is we can create this state variable inside our parent component so inside the source inside our app component let us create a state variable for the input and then pass those values the value function and the on change function as props to filter jobs component.

And inside our jobs component, we only need the value, so let us send the text string as value to our jobs component. Now the filters of component will continue to work, but the value field is now also accessible to the jobs component, and all you have to do is filter out all the titles which match the substring typed in the input field.

To do that, we can simply use. Title dot includes substring, and that will automatically filter out all the elements that do not contain â€" the substring. Now let us see that in action now, if we are typing if we type the text into the input field, we should now be able to sort we should not be able to filter out all the titles that match our subject.

Props Data flow

The important takeaway here is that while props do flow from the parent to the child and not vice versa the child can still send data to the parent the way it works is the parent function can send a function as props to the child and the child can call that function when a certain event happens and it can call the function with the certain argument, and this this argument will be available to the parent prop inside the function declaration.

So this way parents and child competition communicate. And data flow can happen seamlessly in react. Of course, there is also a problem when there is data across when there is data spread across. Siblings. So, either the one option is to move the data into a common parent. Or to make one component the parent of another.

Check box elements

We have seen how we can render an input element using state. Let us add a couple of more input elements. So that we can filter our jobs a bit more thoroughly. So our jobs list also contain. A couple of Boolean variables within values called feature and remote. So let us filter our jobs by these various as well.

So we'll create two checkboxes one checkbox for featured in one chuck box for remote to do that we can use the use state hook again. And similar to how we have created used it for the input, we can create a used state for a checkbox as well, and it looks something like this.

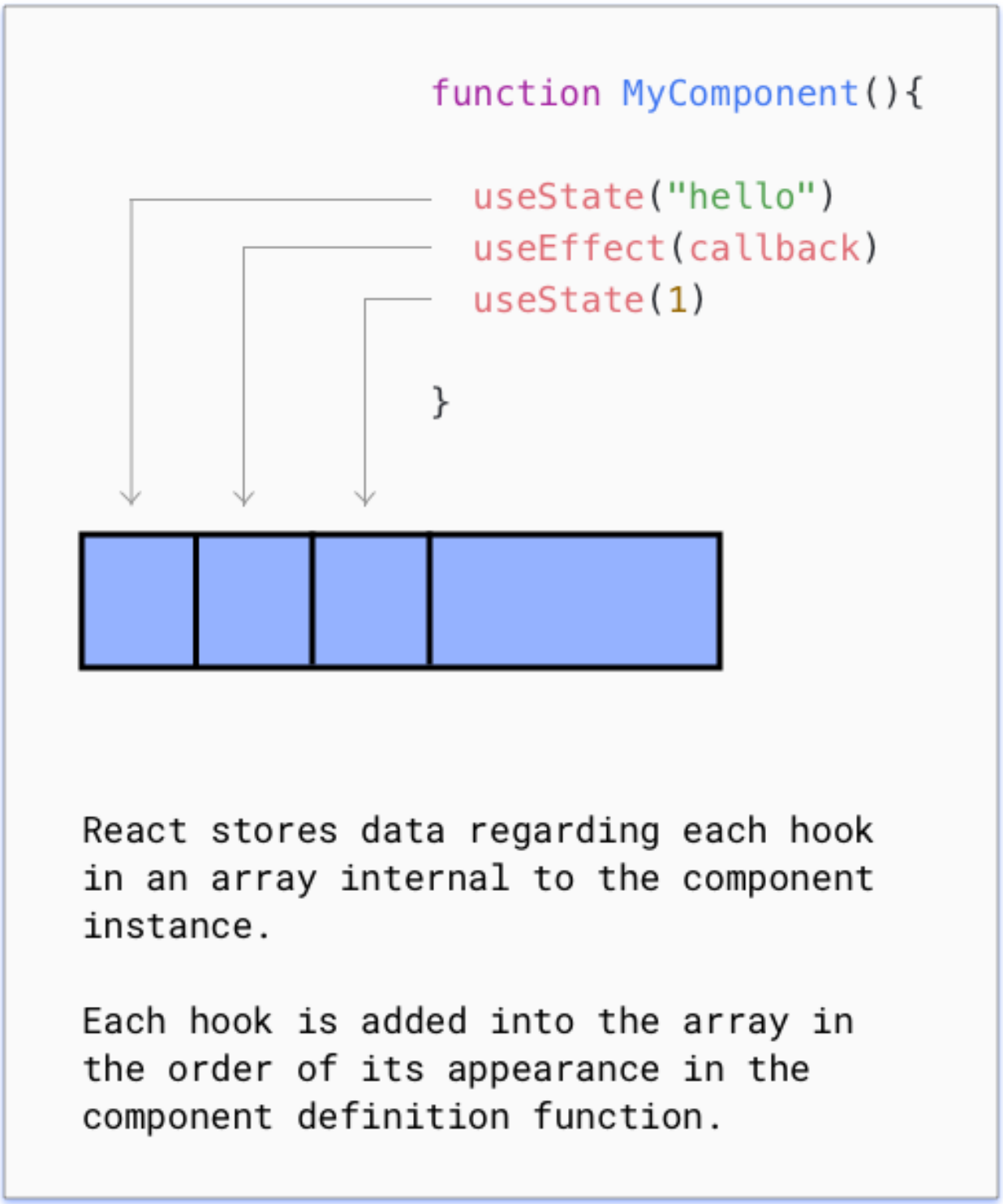
The only difference here is that the input type checkbox. Has a property called checked instead of value.

It it still has the on change property. And everything after that point is still the same. So now, if we should, we can go back to our jobs component and add a couple of more statements to filter the jobs that are featured and remote. So now, what this means is that if the filter text exists, then the jobs are filtered by the text.

If the show featured jobs, Boolean is set, then the list is filled at even further to only show jobs which are featured and the same thing for remote jobs as well. Okay, so now in this chapter, we have seen how we can filter our jobs from a component in the jobs list component.

In the next chapter, we will talk more about cooks.

Understanding Hooks



Do

Maintain hooks order

Don't

Dont use hooks inside

```
const [value, setValue] = useState(0);


// A custom hook that computes the square of a value
function useSquareValue(initialValue) {
  const [value, setValue] = useState(initialValue);
  return [value * value, setValue];
}
const [squareValue, setValue] = useIncrementedState(5);
// squareValue = 25
//  Calling setValue(6) will make squareValue 36


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

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

  return {
    value,
    onChange
  };
}


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

function App() {
  const searchTextInputState = useInputState("");

  return ...
    <FilterJobs
        searchTextInputState={searchTextInputState}
        ...
    />
    <JobsList value={searchTextInputState.value} .../>
  ...;
}
```

```
//src/components/FilterJobs.js

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

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

```
// 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}
    />
    ...
}
```

How do hooks really work?

Dos and Don’t of hooks

Custom hooks

So what exactly are hooks are special. JavaScript functions that are relevant only in component. Functions hooks have an ability to give the component some more behavior. By maintaining the internal state of the component inside a data structure.

How do hooks really work?

So the way it works is whenever a hook is created when the component renders for the first time the value of the initial value of the hook is added on to an array and that state is maintained inside the component across vendors, and the next time the component rerenders it does not initialize the state value but it just simply grabs the value that has been stored in the array and since it back as the value.

So that way across multiple renders the value of. The hook is simply a whatever value was stored inside the array inside the component internal when multiple components are declared they are added. Each hook is added as a new element into the array and. And when the components, the values on the array is simply picked.

So this means that if the for the full five hooks to. To be to function seamlessly, it is essential that the list size stays the same throughout the life cycle of the component. What happens is that if the hooks are conditionally rendered so if the hook is present inside an if else condition, that means that if the condition is true, the hook is present otherwise, the focus is not that means that the length of the hook changes.

This fx the updation process and the value across renders is not consistent that is why it is absolutely important that the number of hooks remains constant and hooks cannot be put inside if else conditions are for loops or inside other functions, which they can only appear inside the component in the top level.

So let us so here is diagram that illustrates all the do’s and don’ts of react hooks.

Dos and Don’t of hooks

Custom hooks

Another beautiful feature about hooks is that hooks can be composed, that means that since hooks are simply jousted functions, you can create a new custom hook by composing a couple of hooks together.

So in our case, we have created a couple of input components which have similar behavior, so our input input state. Is simply a string and a function that can track an event. So that is essentially the behavior of an input look so we can create a custom hook by by doing something like this.

You. If you look at this checkbox state input. These two fields the the on-gen and value props across multiple components, they seem redundant. So we can grab these values and make a custom hook out of them. We can do that like so. So now, we have created a function called use checkbox state, and we can use that inside our app dot app component and create a state value that is specific for checkboxes.

We can also create something similar for input components. For so, we can create a hook called use input state. Let us maintain all our hooks inside. SRC hooks folder. So that it’s easier for us to find our modules.

So to recap hooks are simply functions, but they need only to be used inside JavaScript inside components, and they require that the array in that the number of hook stage the same throughout the components life cycle. So we need to follow a set of practices to ensure that we do not misuse them.

We have already seen one hook in the next three chapters, we’ll talk about some of the other built-in react hooks and how we can build better components using them.

Refactor

Since we are refactoring our code a little bit let us do one small refactor where we will create a new jobs list component and move that into our components folder so now we should have three components jobs list job and filter jobs, let’s also move our date our jobs data into a SRC data folder.

And import that into our SRC index file.

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.

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

Updating document title

We want our `FilterJobs`component 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>
  );
}
```

RemoteJobify

Searching for 'angular'

angular All

Options

Featured Remote

Jobs

1. Senior Angular.js Developer
2. Senior Angular.js Engineer
3. Senior Angular.js Engineer
4. Senior Angular.js Engineer
5. Junior Angular.js Engineer
6. Lead Angular.js Architect
7. Senior Angular.js Engineer
8. Lead Angular.js Engineer
9. Lead Angular.js Architect

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-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]);
```

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 book inside our filter FilterJobs component.

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>
);
```

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>
```

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>
  );
}
```

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.

chapter7-end-screenshot

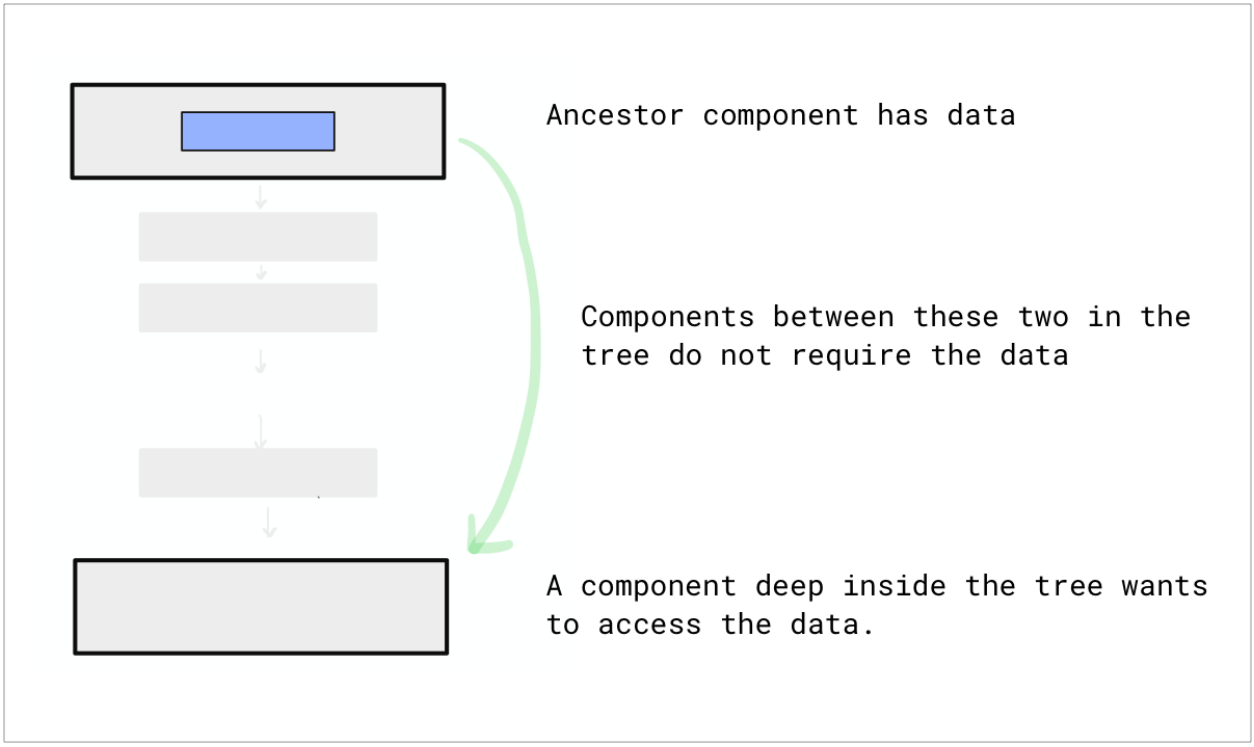
Context

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.



context-usefulness

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.

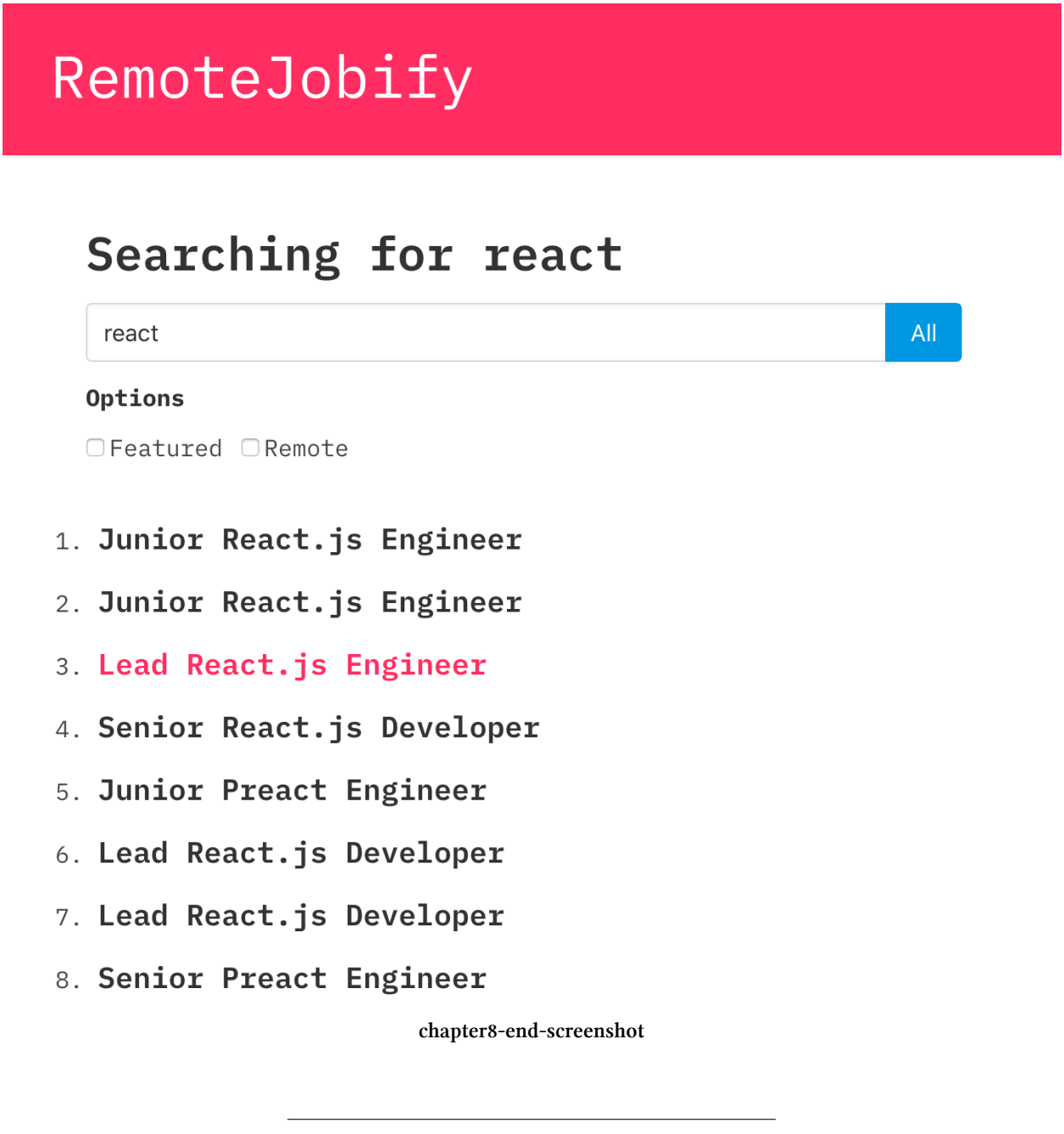
Finally within the job component we can request the context by using the `useContext` hook. And we can now access the `selectedJobContextValue` inside the job component. Finally when let's add a click handler and set the `selectedJob` value.

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;
```



chapter8-end-screenshot

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.se` changes.

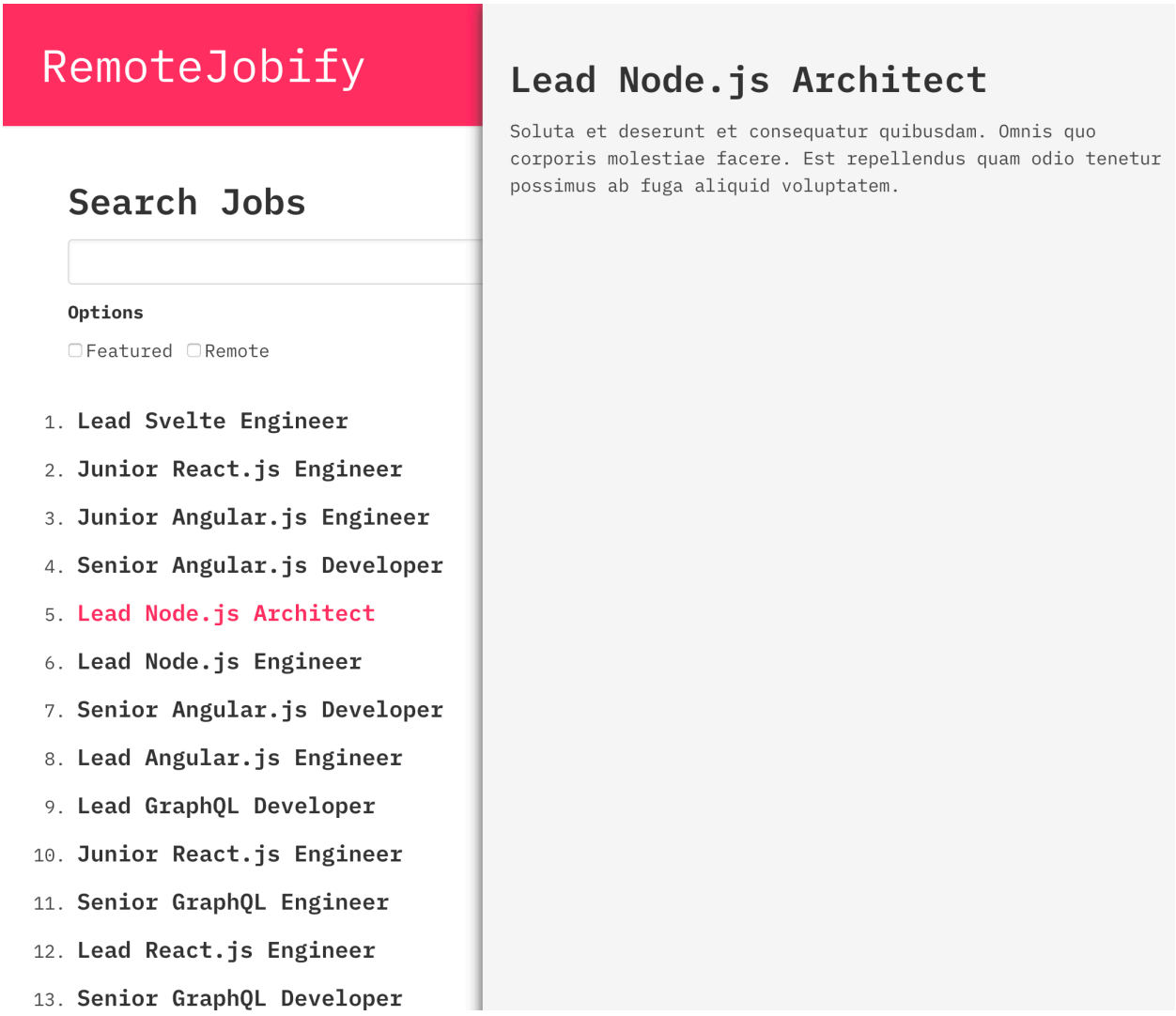
```
//src/components/JobDrawer

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>
}
```



chapter8-end-screenshot2

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.

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.

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.

Closing thoughts

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.

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.