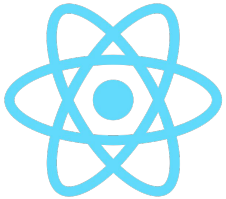


Intro to React JS

Modern Frontend Development

Useful Links

- Office Hour (Sat 3-6PM): https://calendly.com/stanley_nguyen/office_hour
- Q&A Group (with peers and instructors): https://fb.com/groups/276374728052624 7
- Capstone project:
  - Requirements and rubrics https://drive.google.com/file/d/1r1TbBnKgE2LOE3RyMEgkiITV7dy2uyyt/view?usp=sharing
  - Use github.com to host your code

Agenda for today

- Chapter 1: Raw React
- Chapter 2: Modern Frontend Tooling
- Chapter 3: JSX
- =🌭=🍔= Lunch BREAK =🥗=🍜=
- Interlude: Functional Programming
- Chapter 4: Component
- Chapter 5: useState
- Homework / Project Help

Chapter 1: Raw React

Demo - 01

Visit https://codesandbox.io/s/transition-to-react-initial-ch251

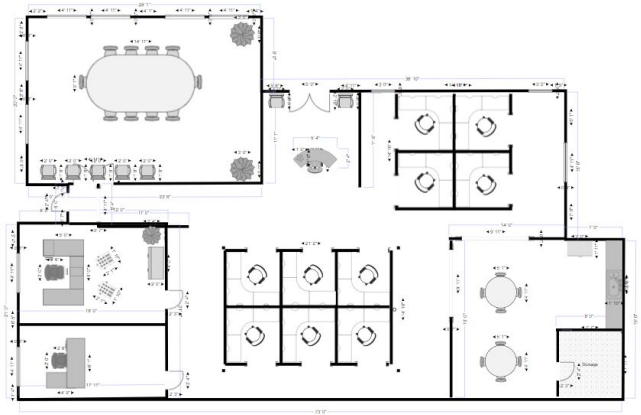
and click the Fork button at top right corner.

file: **`public/index.html`**

A Metaphor: Building/Renovating a House

When we want to build or renovating a house, the steps are:

1 **get architect/engineer to draw out the blueprint**



2 **get architect/engineer to perform the work**

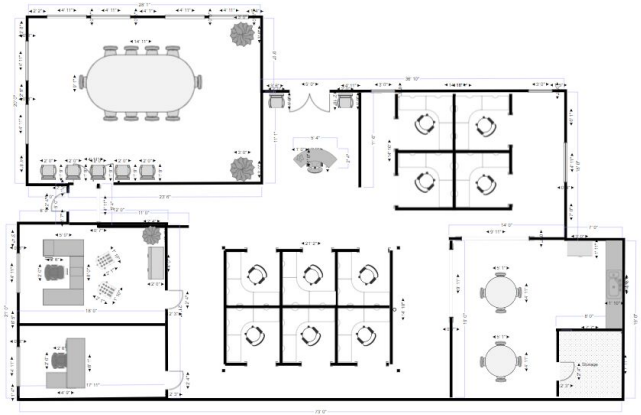
if it is a new building, build from scratch

if it is a current building, find the difference with previous building, then change

React: Creating/Updating a UI

When we want to create or update a UI, the steps are:

1 **use React to create React elements (the blueprint)**



2 **use ReactDOM to perform the work**

if it is a new UI, create from scratch

if it is an existing UI, compare with previous React elements and perform the changes

Hello World

Vanilla JS

```
<script>
  const span = document.createElement('span');
  span.innerHTML = 'Hello world';
  span.className = 'text-xl text-gray-500';
  const target = document.querySelector('#target');
  target.appendChild(span);
</script>
```

Hello World

Vanilla JS

```
<script>
  const span = document.createElement('span');
  span.innerHTML = 'Hello world';
  span.className = 'text-xl text-gray-500';
  const target = document.querySelector('#target');
  target.appendChild(span);
</script>
```

React JS

```
<script
  crossorigin
  src="https://unpkg.com/react@17/umd/react.development.js"
></script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
></script>
<script>
  const span = React.createElement(
    'span',
    {
      className: 'text-xl text-gray-500'
    },
    'Hello world'
  );
  const target = document.querySelector('#target');
  ReactDOM.render(span, target);
</script>
```

Hello World

Vanilla JS

```
<script>
  const span = document.createElement('span');
  span.innerHTML = 'Hello world';
  span.className = 'text-xl text-gray-500';
  const target = document.querySelector('#target');
  target.appendChild(span);
</script>
```

Load **`react`** (exposes **`React`**) and **`react-dom`** (exposes **`ReactDOM`**)

React JS

```
<script
  crossorigin
  src="https://unpkg.com/react@17/umd/react.development.js"
></script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
></script>
<script>
  const span = React.createElement(
    'span',
    {
      className: 'text-xl text-gray-500'
    },
    'Hello world'
  );
  const target = document.querySelector('#target');
  ReactDOM.render(span, target);
</script>
```

Hello World

Vanilla JS

```
<script>
  const span = document.createElement('span');
  span.innerHTML = 'Hello world';
  span.className = 'text-xl text-gray-500';
  const target = document.querySelector('#target');
  target.appendChild(span);
</script>
```

React JS

```
<script
  crossorigin
  src="https://unpkg.com/react@17/umd/react.development.js"
></script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
></script>
<script>
  const span = React.createElement(
    'span',
    {
      className: 'text-xl text-gray-500'
    },
    'Hello world'
  );
  const target = document.querySelector('#target');
  ReactDOM.render(span, target);
</script>
```

Load **`react`** (exposes **`React`**) and **`react-dom`** (exposes **`ReactDOM`**)

Create a React element using **`React.createElement`**.

Hello World

Vanilla JS

```
<script>
  const span = document.createElement('span');
  span.innerHTML = 'Hello world';
  span.className = 'text-xl text-gray-500';
  const target = document.querySelector('#target');
  target.appendChild(span);
</script>
```

React JS

```
<script
  crossorigin
  src="https://unpkg.com/react@17/umd/react.development.js"
></script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
></script>
<script>
  const span = React.createElement(
    'span',
    {
      className: 'text-xl text-gray-500'
    },
    'Hello world'
  );
  const target = document.querySelector('#target');
  ReactDOM.render(span, target);
</script>
```

Load **`react`** (exposes **`React`**) and **`react-dom`** (exposes **`ReactDOM`**)

Create a React element using **`React.createElement`**.

Call **`ReactDOM.render`** to perform DOM manipulations based on the provided React element.

We use two libraries (**`react`**) and (**`react-dom`**).

- **`react`** is used to declare the UI and its behaviors
- **`react-dom`** is used to convert the declarations to actual DOM operations.

`React.createElement` creates the data that describe the UI we want.

`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

```
<span class="text-xl text-gray-500">Hello world</span>
```

`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

```
<span class="text-xl text-gray-500">Hello world</span>
```

```
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello world'
);

console.log(span);
```

`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

```
<span class="text-xl text-gray-500">Hello world</span>
```

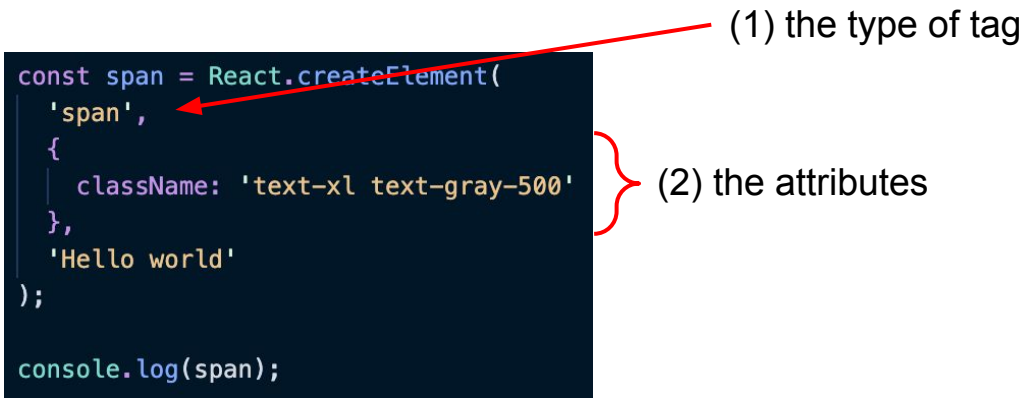
(1) the type of tag

```
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello world'
);

console.log(span);
```

`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

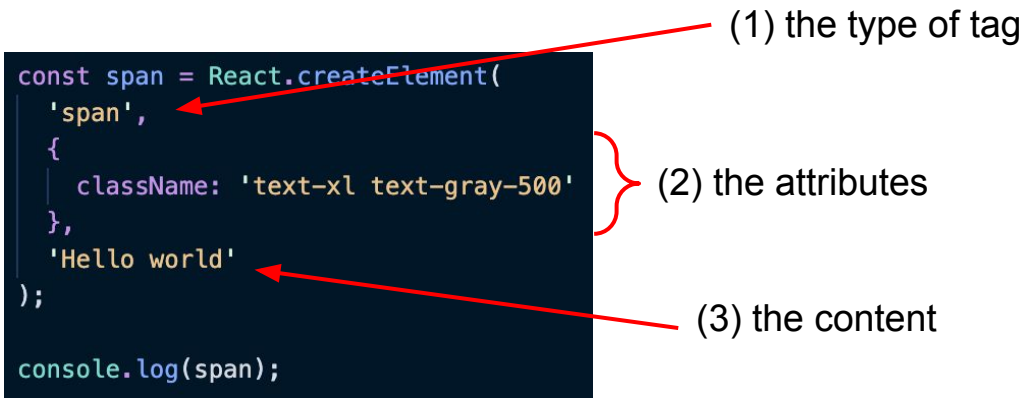
```
<span class="text-xl text-gray-500">Hello world</span>
```



`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

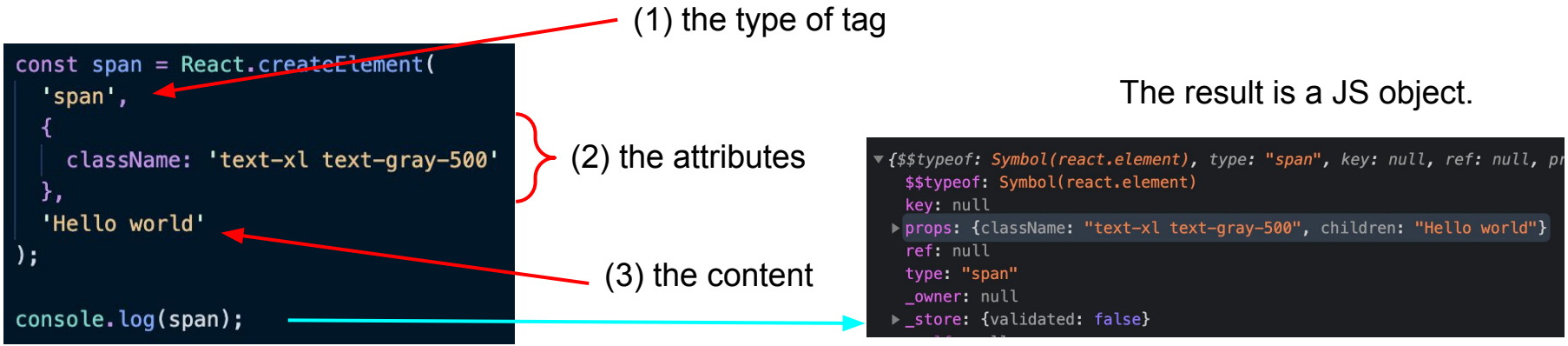
```
<span class="text-xl text-gray-500">Hello world</span>
```



`React.createElement` creates the data that describe the UI we want.

To describe a HTML element, we need to specify (1) the type of tag, (2) the attributes on the element, and (3) the content within the tag.

```
<span class="text-xl text-gray-500">Hello world</span>
```

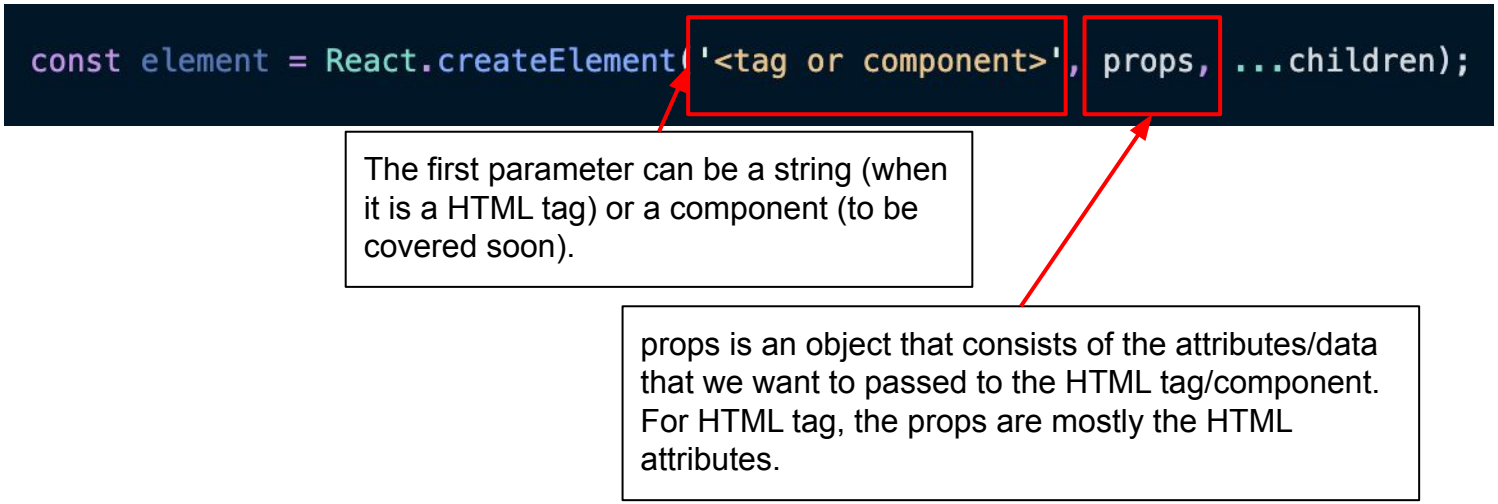


`React.createElement` function definition:

```
const element = React.createElement('<tag or component>', props, ...children);
```

The first parameter can be a string (when it is a HTML tag) or a component (to be covered soon).

`React.createElement` function definition:



`React.createElement` function definition:

The first parameter can be a string (when it is a HTML tag) or a component (to be covered soon).

props is an object that consists of the attributes/data that we want to passed to the HTML tag/component. For HTML tag, the props are mostly the HTML attributes.

children are the content of the tag/component. They can be string or react elements. We can pass multiple children.

`React.createElement` function definition:

The first parameter can be a string (when it is a HTML tag) or a component (to be covered soon).

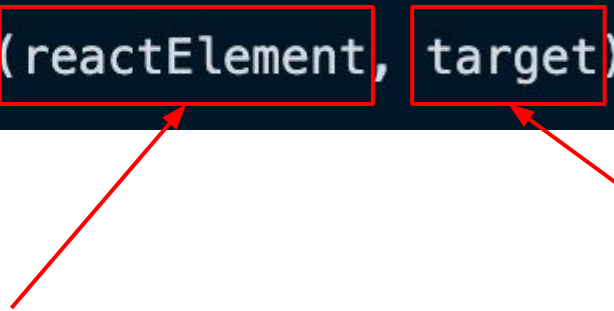
props is an object that consists of the attributes/data that we want to passed to the HTML tag/component. For HTML tag, the props are mostly the HTML attributes.

The return value is known as **React element**, which is a JavaScript object that describes the UI.

children are the content of the tag/component. They can be string or react elements. We can pass multiple children.

`ReactDOM.render` perform DOM manipulation based on provided React elements.

```
ReactDOM.render(reactElement, target);
```

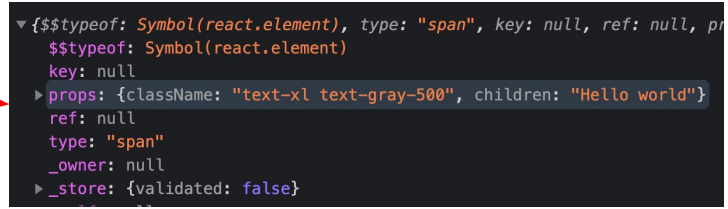


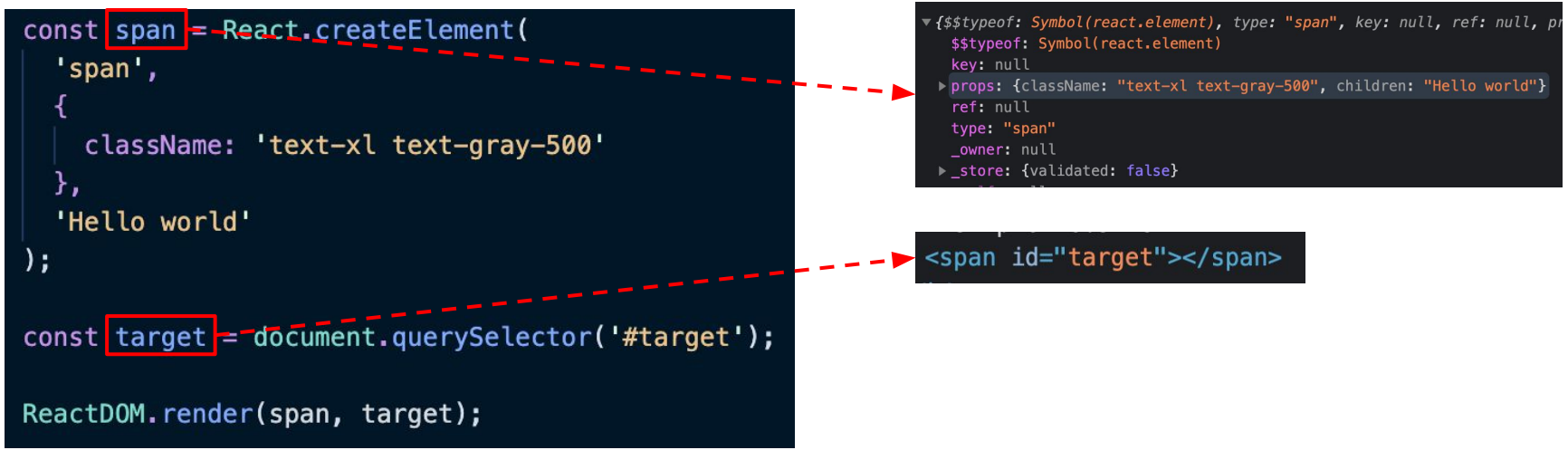
react element created using **`React.createElement`**

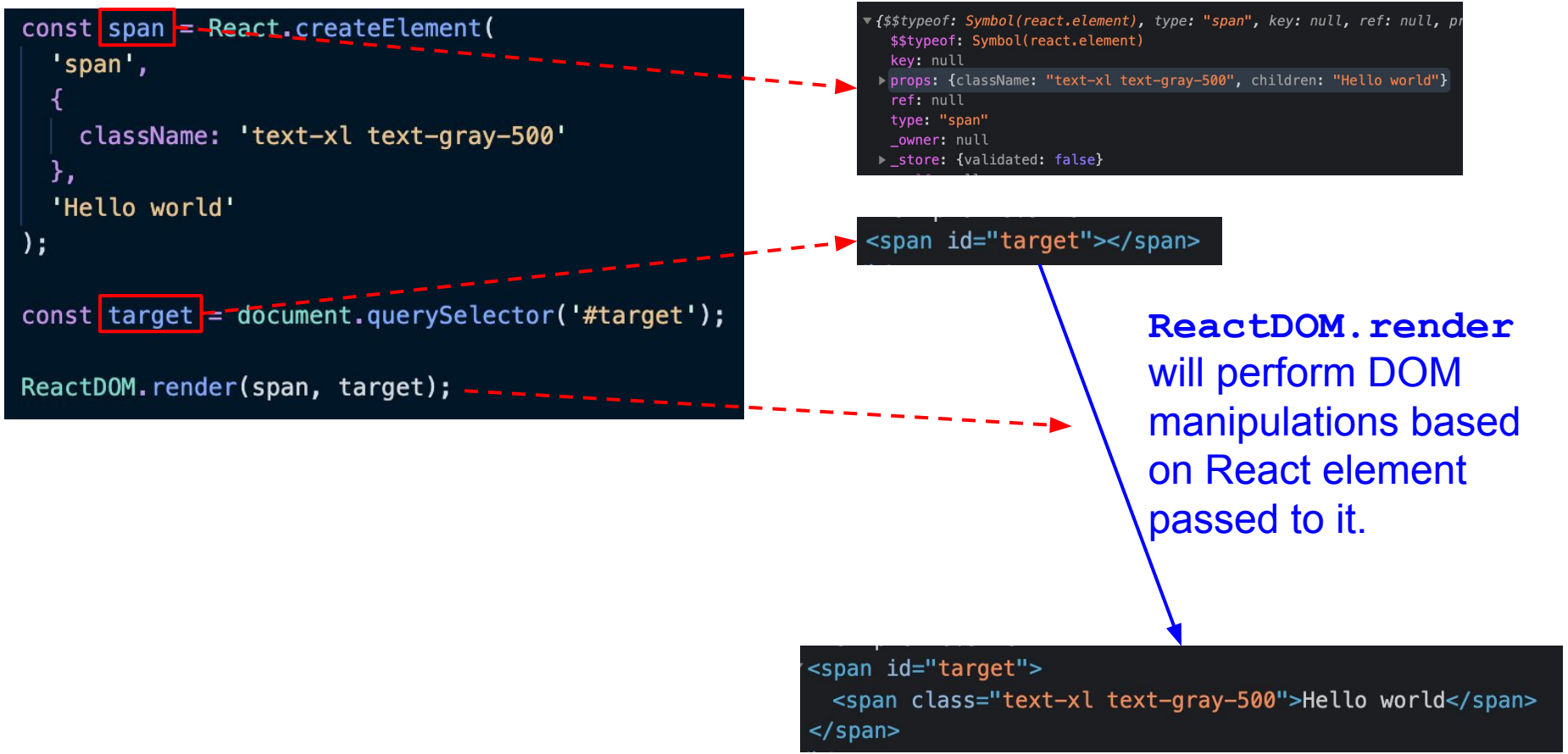
HTML element which you want to insert the UI described by the react element

```
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello world'
);

const target = document.querySelector('#target');

ReactDOM.render(span, target);
```

```
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello world'
);

const target = document.querySelector('#target');

ReactDOM.render(span, target);
```




const span = React.createElement(
'span',
{
className: 'text-xl text-gray-500'
},
'Hello world'
);
const target = document.querySelector('#target');
ReactDOM.render(span, target);
{$$typeof: Symbol(react.element), type: "span", key: null, ref: null, pr
$$typeof: Symbol(react.element)
key: null
props: {className: "text-xl text-gray-500", children: "Hello world"}
ref: null
type: "span"
_owner: null
_store: {validated: false}


const span = React.createElement(
'span',
{
className: 'text-xl text-gray-500'
},
'Hello world'
);
const target = document.querySelector('#target');
ReactDOM.render(span, target);
{$$typeof: Symbol(react.element), type: "span", key: null, ref: null, pr
$$typeof: Symbol(react.element)
key: null
props: {className: "text-xl text-gray-500", children: "Hello world"}
ref: null
type: "span"
_owner: null
_store: {validated: false}

ReactDOM.render will perform DOM manipulations based on React element passed to it.

Hello world

```
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello world'
);

const target = document.querySelector('#target');

ReactDOM.render(span, target);
```

```
{$$typeof: Symbol(react.element), type: "span", key: null, ref: null, pr
  $$typeof: Symbol(react.element)
  key: null
  props: {className: "text-xl text-gray-500", children: "Hello world"}
  ref: null
  type: "span"
  _owner: null
  _store: {validated: false}
```

```
<span id="target"></span>
```

`ReactDOM.render` will perform DOM manipulations based on React element passed to it.

```
<span id="target">
  <span class="text-xl text-gray-500">Hello world</span>
</span>
```

Under the hood, **`ReactDOM.render`** still use DOM API that you can use directly, e.g. **`document.createElement`** and **`.innerHTML`**.

The point of using React is **NOT** to do something that JS cannot do, it is that you can **focus on describe the UI, regardless if it is a new element or an update**, and React figure out what operations to be perform.

As HTML element can be nested, React element can be nested as well.

```
const worldEl = React.createElement(
  'em',
  {
    className: 'text-red-700'
  },
  'world'
);
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello, ',
  worldEl
);      You, 2 hours ago • convert hello world to
const target = document.querySelector('#target');
ReactDOM.render(span, target);
```

As HTML element can be nested, React element can be nested as well.

```
const worldEl = React.createElement(
  'em',
  {
    className: 'text-red-700'
  },
  'world'
);
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello, ',
  worldEl
);      You, 2 hours ago • convert hello world to
const target = document.querySelector('#target');
ReactDOM.render(span, target);
```

span element contains `'Hello, '` string and the `worldEl` element

As HTML element can be nested, React element can be nested as well.

```
const worldEl = React.createElement(
  'em',
  {
    className: 'text-red-700'
  },
  'world'
);
const span = React.createElement(
  'span',
  {
    className: 'text-xl text-gray-500'
  },
  'Hello, ',
  worldEl
);      You, 2 hours ago • convert hello world to
const target = document.querySelector('#target');
ReactDOM.render(span, target);
```

span element contains `'Hello, '` string and the `worldEl` element

```
<span id="target">
  <span class="text-xl text-gray-500">
    "Hello, "
    <em class="text-red-700">world</em>
  </span>
</span>
```

Do It - 01

In **`public/faq.html`**, add code to insert following HTML in footer using React

```
<span class="text-xs">
  "Today is "
  <b class="underline">Sun</b>
</span>
```

Recap Chapter 1: Raw React

1. We use React to declare the UI that we want calling `React.createElement`, which will returns us React element, a data structure that represents our UI *(the blueprint)*.
2. React elements can have nested React elements, just like HTML elements may contains HTML elements.
3. We use ReactDOM to show the UI by calling `ReactDOM.render` with the React element that we have created. ReactDOM will create all the HTML elements (if the elements are never created) or performing the necessary update for the difference (if the elements had been rendered before) *(the work)*.

Chapter 2: Modern Frontend Tooling

featuring Create React App

Demo - 02

How to create a **`react-marketplace`** project using Create React App (CRA).

Create React App is a tool to create a React project.

- Create React App (https://create-react-app.dev/) is a tool to setup a new React project with frontend tooling.
- We use `npx` (a command that comes with `npm`) to auto-download it and then run it.

The project created with CRA has the following features:

1. development server that watch your code change and auto reload the browser
2. compile the code using Babel (example) to use latest JS feature while supporting older browsers
3. optimize the code to be deployed (minification and tree-shaking)
4. auto integrate HTML/CSS/JS (no more manual update the script tag)
5. enable powerful technique like lazy loading
6. enable a module system (splitting code into multiple files for better code organization) - more on this shortly

Do It - 02

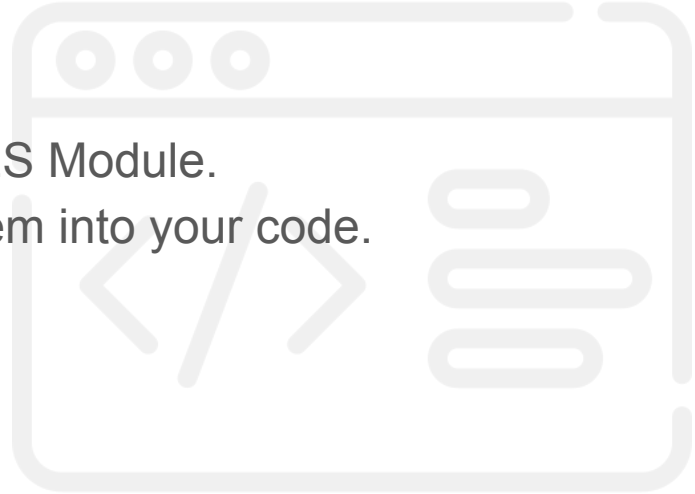
1. Go to https://codesandbox.io/s/ and choose “React” template.
2. Remove all the files in src folder except index.js.
3. Replace the code in **`src/index.js`**.
4. Add links to CSS files in **`public/index.html`**.

Resources: https://bit.ly/react-starter-file

Note: To minimize technical glitch in this session, we will use Codesandbox which provides almost similar features as Create React App. However, you’re required to use Create React App for your Take Home Homework.

Demo - 03

- How to split your code into multiple files using ES Module.
- How to install third-party libraries and import them into your code.



We use module system to specify the relationship between our code.

1. A file is a module

We use module system to specify the relationship between our code.

1. A file is a module
2. Use import to specify the code that we need

src/index.js

```
import { showNow } from "./show-now";

showNow();

setInterval(showNow, 1000);
```

JS

We use module system to specify the relationship between our code.

1. A file is a module
2. Use import to specify the code that we need
3. Use export to specify what can be imported from this module (file).

src/index.js

```
import { showNow } from "./show-now";

showNow();

setInterval(showNow, 1000);
```

src/show-now.js

```
const target = document.querySelector('#root');

export function showNow() {
  const now = new Date().toLocaleTimeString();

  const el = React.createElement(
    'small',
    {},
    'Now is ',
    React.createElement('span', { className: 'font-bold' }, now)
  );

  ReactDOM.render(el, target);
}
```

JS

We use module system to specify the relationship between our code.

1. A file is a module
2. Use import to specify the code that we need
3. Use export to specify what can be imported from this module (file).

src/index.js

```
import { showNow } from "./show-now";

showNow();

setInterval(showNow, 1000);
```

src/show-now.js

```
const target = document.querySelector('#root');

export function showNow() {
  const now = new Date().toLocaleTimeString();

  const el = React.createElement(
    'small',
    {},
    'Now is ',
    React.createElement('span', { className: 'font-bold' }, now)
  );

  ReactDOM.render(el, target);
}
```

JS

We use module system to specify the relationship between our code.

1. A file is a module
2. Use import to specify the code that we need
3. Use export to specify what can be imported from this module (file).

src/index.js

```
import { showNow } from "./show-now";

showNow();

setInterval(showNow, 1000);
```

src/show-now.js

```
const target = document.querySelector('#root');

export function showNow() {
  const now = new Date().toLocaleTimeString();

  const el = React.createElement(
    'small',
    {},
    'Now is ',
    React.createElement('span', { className: 'font-bold' }, now)
  );

  ReactDOM.render(el, target);
}
```

JS

We use module system to specify the relationship between our code.

1. A file is a module
2. Use import to specify the code that we need
3. Use export to specify what can be imported from this module (file).

target variable is not exported, therefore only code in this file has access to it.

This is a good thing, it means we can safely rename it or remove it in future if other code in this file don't need it anymore.

src/index.js

```
import { showNow } from "./show-now";

showNow();

setInterval(showNow, 1000);
```

src/show-now.js

```
const target = document.querySelector('#root');

export function showNow() {
  const now = new Date().toLocaleTimeString();

  const el = React.createElement(
    'small',
    {},
    'Now is ',
    React.createElement('span', { className: 'font-bold' }, now)
  );

  ReactDOM.render(el, target);
}
```

JS

Expose variable (including function) using **`export`**.

There are two types of export:

Expose variable (including function) using **`export`**.

There are two types of export:

named export

```
export const x = 3;

export function hello() {
  console.log('Hello')
}

export const person = {
  name: 'Lady Gaga',
  sing() {
    console.log('rara oh mama')
  }
}
```

Expose variable (including function) using **`export`**.

There are two types of export:

named export

```
export const x = 3;

export function hello() {
  console.log('Hello')
}

export const person = {
  name: 'Lady Gaga',
  sing() {
    console.log('rara oh mama')
  }
}
```

default export

```
const person = {
  name: 'Lady Gaga',
  sing() {
    console.log('rara oh mama')
  }
}

export default person;
```

Only one default export is allowed per module

JS

Expose variable (including function) using **`export`**.

There are two types of export:

named export

```
export const x = 3;

export function hello() {
  console.log('Hello')
}

export const person = {
  name: 'Lady Gaga',
  sing() {
    console.log('rara oh mama')
  }
}
```

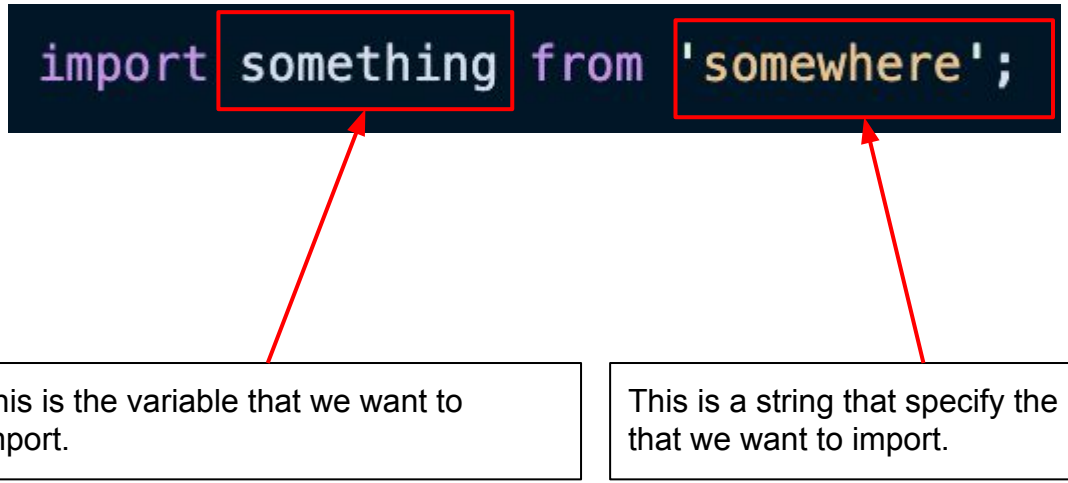
default export

```
const person = {
  name: 'Lady Gaga',
  sing() {
    console.log('rara oh mama')
  }
}

export default person;
```

Only one default export is allowed per module

You can export any variable (object, function).

Import variable (including function) using **`import`**.



JS

We can import our module or third-party module.

```
import something from 'somewhere';
```

- when we import using relative path (starts with `.` character, e.g. `'./filename'`), then the module to be imported will be the file that is located relative to the importing module. The import path follows how file path works in Unix system, i.e.
  - to import file in same folder use `'./filename'`
  - to import file in subfolder use `'./subfolder/filename'`
  - to import file by going to parent folder, use `'../filename'`

- when we import using absolute path (does not start with `.` character, e.g. `'react'`), then the module will be obtained from **`node_modules`** folder.
  - The third-party module in **`node_modules`** are downloaded because they are listed in **`dependencies`** key in **`package.json`**.

JS

The style to import depends on how the variable is exported. (1 of 2)

```
import something from 'somewhere';
```

- To import a variable that is exported with named export, we need to have a curly braces. We can import multiple variables at the same time.

src/x.js

```
export const hello = 'hello';
export const world = 'world';
```

src/y.js

```
import { hello } from './x';
```

src/z.js

```
import { hello, world } from './x';
```

- We can also import all named exports into a object.

src/a.js

```
import * as x from './x';

console.log(x.hello);
```

JS

The style to import depends on how the variable is exported. (2 of 2)

```
import something from 'somewhere';
```

- To import a variable that is exported with default export, we can name it anything that we wish.

src/x.js

```
const hello = 'hello';

export default hello;
```

src/y.js

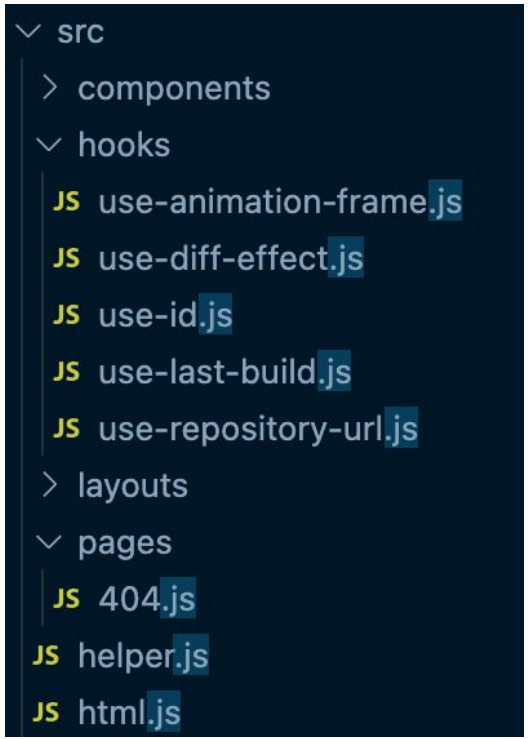
```
import hello from './x';
```

src/z.js

```
import HELLO_VALUE from './x';
```

JS

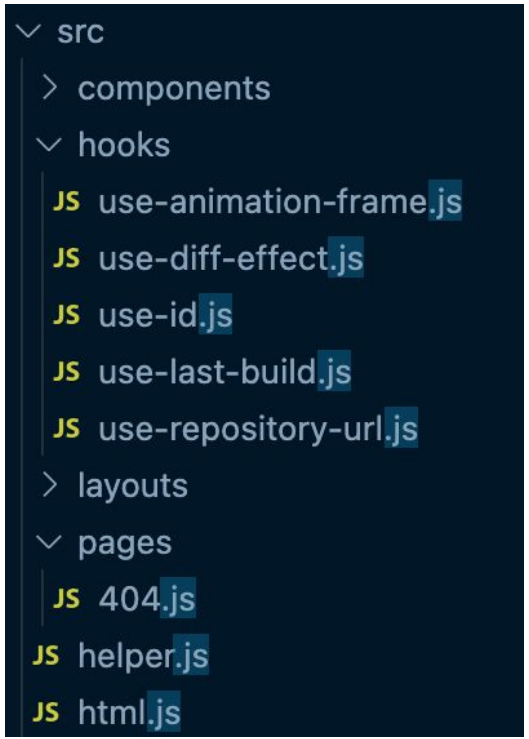
Question



How to import **`multiply`** function exported using named export from **`helper.js`** in **`html.js`**?

1. `import multiply from 'helper';`
2. `import {multiply} from './helper';`
3. `import multiply from './helper';`
4. `from './helper' import multiply;`

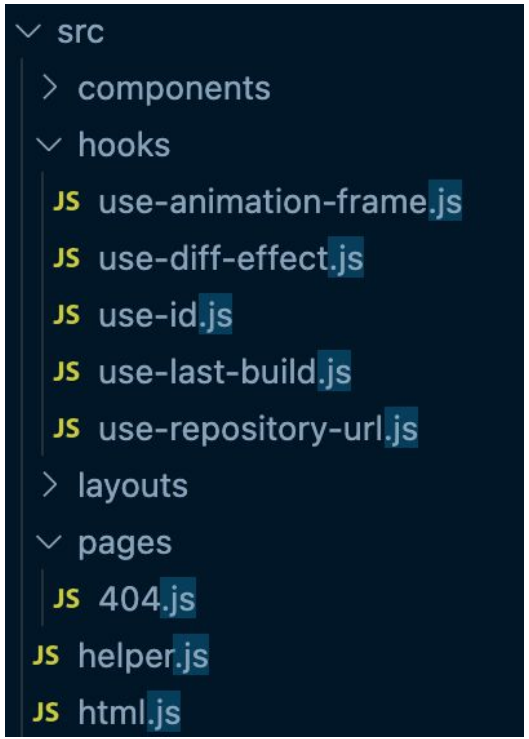
Question



How to import `useId` function exported using default export from `use-id.js` in `html.js`?

1. `import useId from './use-id';`
2. `import {useId} from './hooks/use-id';`
3. `import useId from '../hooks/use-id';`
4. `import useId from './hooks/use-id';`

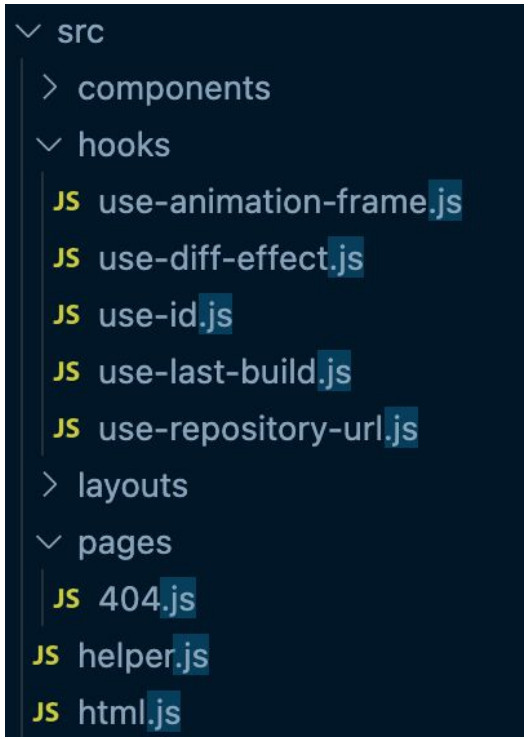
Question



How to import `multiply` function exported using named export from `helper.js` in `use-last-build.js`?

1. `import { multiply } from '../helper';`
2. `import multiply from '../helper';`
3. `import {multiply} from './helper';`
4. `import multiply from './helper';`

Question



How to import `useId` function exported using default export from `use-id.js` in `404.js`?

1. `import useId from './use-id';`
2. `import {useId} from './hooks/use-id';`
3. `import useId from '../hooks/use-id';`
4. `import useId from './hooks/use-id';`

Do It - 03

1. In **`public/index.html`**. add the following content before the **`</body>`** closing tag.

```
<span id="today-weekday"></span>
```

2. Create a file **`src/show-weekday.js`** (next to **`index.js`**). Copy paste the code that you did in last chapter to show the weekday into a **`showWeekday`** function in this file and export it.
3. Import **`showWeekday`** function in **`index.js`** and call it.

Recap Chapter 2: Modern Frontend Tooling

1. We use build tools like Create React App to improve development workflow and optimize the code output.
2. We use **`export`** to expose variable to be used by other module.
3. We use **`import`** to get variable exposed by other modules.

Chapter 3: JSX

Demo - 04

- How to replace `React.createElement` with JSX.

JSX is a language extension to represent **`React.createElement`**

React

```
const el = React.createElement(
  'small',
  {},
  'Now is ',
  React.createElement('span', { className: 'font-bold' }, now)
);
```

```
<small>
  "Now is "
  <span class="font-bold">02:58:31</span>
</small>
```

It takes some effort to translate the **`React.createElement`** call to the UI it represent.

JSX is a language extension to represent **`React.createElement`**

It takes some effort to translate the **`React.createElement`** call to the UI it represent.

React

```
const el = React.createElement(
  'small',
  {},
  'Now is ',
  React.createElement('span', { className: 'font-bold' }, now)
);
```

```
<small>
  "Now is "
  <span class="font-bold">02:58:31</span>
</small>
```

React (with JSX)

```
const el = (
  <small>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

Now it is effortless!

JSX is a language extension to represent `React.createElement`

It takes some effort to translate the `React.createElement` call to the UI it represent.

React

```
const el = React.createElement(
  'small',
  {},
  'Now is ',
  React.createElement('span', { className: 'font-bold' }, now)
);
```

```
<small>
  "Now is "
  <span class="font-bold">02:58:31</span>
</small>
```

React (with JSX)

```
const el = (
  <small>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

Now it is effortless!

This transformation is made possible by Babel, which not only able to compile new JS syntax to old syntax, it is also possible for it to compile non-standard JS syntax (JSX) to valid JS.

JavaScript in JSX

Text inside JSX will be treated as string.

We are not looking for variables **`Now`** and **`is`**. Instead, this is just a string **`'Now is '`**

JavaScript in JSX

```
const el = (
  <small>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

Text inside JSX will be treated as string.

We are not looking for variables **`Now`** and **`is`**. Instead, this is just a string **`'Now is '`**

Use curly braces (**`{ }`**) to switch back to JavaScript

`now` here will be treated as JS, so it will look for a variable **`now`**.

Comments in JSX

```jsx
const el = (
  <small>
    {/* comment in JSX */}
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

It is using `{}` to switch back to JS, then use `/* */` comment style.

style prop in React is an object

- in contrast, style in HTML is a string.

```
const el = (
  <small style={{ backgroundColor: 'red' }}>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

```
<small style="background-color: red;">
  "Now is "
  <span class="font-bold">03:14:13</span>
</small>
```

style prop in React is an object

- in contrast, style in HTML is a string.

```
const el = (
  <small style={{ backgroundColor: 'red' }}>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

```
<small style="background-color: red;">
  "Now is "
  <span class="font-bold">03:14:13</span>
</small>
```

This is not a new `{{ }}` syntax. Instead, the outer curly braces switch the content into JS, and the inner curly braces is declaring an object.

To illustrate, the code can be rewritten to:

```
const style = { backgroundColor: 'red' };

const el = (
  <small style={style}>
    Now is <span className="font-bold">{now}</span>
  </small>
);
```

Most of the time, we use CSS class for styling.

- we use **`className`** in JSX instead of **`class`**.
- The CSS class we use in this course are provided by TailwindCSS, a utility-first CSS framework: https://tailwindcss.com/ . TailwindCSS provide the most commonly-used CSS properties as independent CSS class, e.g. **`p-`** for padding and **`m-`** for margin.
- TailwindCSS is very popular in React community as it allows us to edit all parts of the UI (markup, logic, and styles) in JS files.
- We will not discuss much about tailwind in this class.

Other than **`class`**, there are few attributes difference between HTML and JSX.

- **`for`** (HTML) - **`htmlFor`** (JSX)
- A few other SVG attributes…

But don’t worry! React usually will detect the common mistake of HTML to JSX mapping and give you helpful error message like below in console:

Just read the message and fix accordingly.

JSX must be closed

- In HTML, you can have unclosed tag, like **`<input >`** or **`<img >`**.
- JSX is stricter that you must always close the tag, for those element without inner element, you can do like this **`<input />`**, **`<img />`**.

Do It - 04: Using JSX

Use JSX in **`src/show-weekday.js`**.

Recap Chapter 3: JSX

1. JSX is a extension of JavaScript syntax (enabled by Babel) to help use visualize the end result represented by **`React.createElement`**.
2. We can embed JS expression inside JSX with curly braces (`{}`).
3. JSX is mostly equal to the final HTML that it representing, with few differences:
    a. JSX use **`className`** and **`htmlFor`** instead of **`class`** and **`for`**.
    b. JSX style is an object instead of string.
    c. JSX must be closed (no unclosed tag allowed).

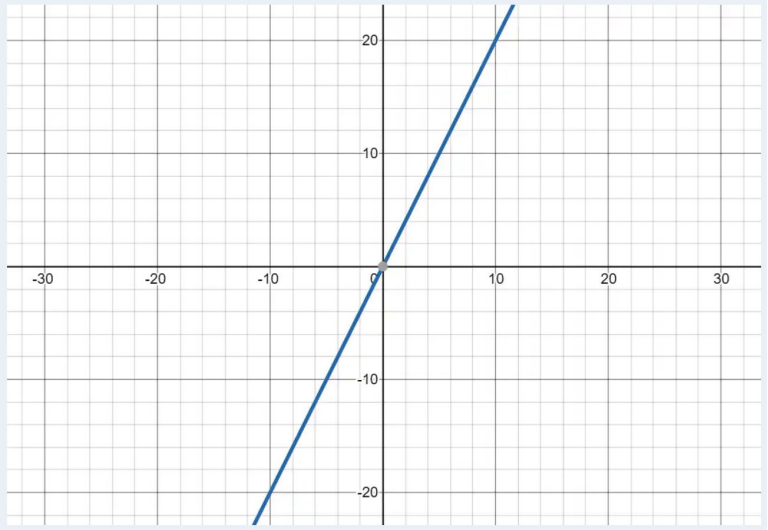
Interlude: Functional Programming & Pure Function

Functional Programming: Defining Function (1 of 2)

- Functional programming is a style of coding that intends to make the code easier to follow and (hopefully) less bugs
- In functional programming, a function is function in mathematical sense, like

```
f(x) = 2x
```

for each value of input (x), there is a result (y).
e.g. x = 10, y = 20.



JS

Functional Programming: Defining Function (2 of 2)

- Note that for a mathematical function,
  - a particular input always returns same result, i.e. input 5 always returns result 10, regardless of who/where/how you use it.
  - even though there may be multiple inputs, there is only one result.
- Function (in functional programming) should have the same characteristics.

  A function accepts parameters and only returns result that purely based on its parameters.

- But because not everyone follow this definition, we usually use the terms ***pure function*** if a function does follow this definition.
- Let's look at some examples of pure function.

What is a pure function (and what is not)

```
function add(x, y, z) {
  return x + y + z;
}

let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}

function addWithRandomNum(x, y, z) {
  return x + y + z + Math.random();
}
```

JS

What is a pure function (and what is not)

```
function add(x, y, z) {
  return x + y + z;
}

let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}

function addWithRandomNum(x, y, z) {
  return x + y + z + Math.random();
}
```

This is a pure function, as same inputs of x, y, z always returns the same result.

JS

What is a pure function (and what is not)

```
function add(x, y, z) {
  return x + y + z;
}

let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}

function addWithRandomNum(x, y, z) {
  return x + y + z + Math.random();
}
```

This is a pure function, as same inputs of x, y, z always returns the same result.

This is **not** a pure function, as the result is different if you call it multiple times.

JS

What is a pure function (and what is not)

```
function add(x, y, z) {
  return x + y + z;
}

let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}

function addWithRandomNum(x, y, z) {
  return x + y + z + Math.random();
}
```

This is a pure function, as same inputs of x, y, z always returns the same result.

This is **not** a pure function, as the result is different if you call it multiple times.

This is **not** a pure function, as the result is different every time you call it due to **`Math.random`**.

JS

A pure function do not have side effect.

Side effect in a function means any operations that is not related to computing the result.

```
let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}
```

```
function updatePerson(person) {
  if (person.age > 18) {
    person.isAdult = true;
  }
}
```

```
function getPokemons() {
  return fetch('/api/pokemons');
}
```

JS

A pure function do not have side effect.

Side effect in a function means any operations that is not related to computing the result.

```
let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}
```

The side effect of this function is **`sum`** variable is changed.

```
function updatePerson(person) {
  if (person.age > 18) {
    person.isAdult = true;
  }
}
```

```
function getPokemons() {
  return fetch('/api/pokemons');
}
```

JS

A pure function do not have side effect.

Side effect in a function means any operations that is not related to computing the result.

```
let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}
```

The side effect of this function is **`sum`** variable is changed.

The side effect of this function is **`person`** variable is changed.

```
function updatePerson(person) {
  if (person.age > 18) {
    person.isAdult = true;
  }
}
```

```
function getPokemons() {
  return fetch('/api/pokemons');
}
```

JS

A pure function do not have side effect.

Side effect in a function means any operations that is not related to computing the result.

```
let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}
```

The side effect of this function is **`sum`** variable is changed.

```
function updatePerson(person) {
  if (person.age > 18) {
    person.isAdult = true;
  }
}
```

The side effect of this function is **`person`** variable is changed.

The side effect of this function is a network request is made.

```
function getPokemons() {
  return fetch('/api/pokemons');
}
```

JS

A pure function do not have side effect.

Side effect in a function means any operations that is not related to computing the result.

```
let sum = 0;
function addUp(x, y, z) {
  sum = sum + x + y + z;
  return sum;
}
```

The side effect of this function is **`sum`** variable is changed.

```
function updatePerson(person) {
  if (person.age > 18) {
    person.isAdult = true;
  }
}
```

The side effect of this function is **`person`** variable is changed.

```
function getPokemons() {
  return fetch('/api/pokemons');
}
```

The side effect of this function is a network request is made.

All of these 3 functions are not pure function as they has side effects.

JS

Question

```
function isNumber(value) {
  return typeof value === 'number';
}

let id = 0;
function getId() {
  return `id-${id++}`
}

function doItLater(callback) {
  return setTimeout(callback, 100);
}
```

1. Is **`isNumber`** a pure function?
2. Is **`getId`** a pure function?
3. Is **`doItLater`** a pure function?

JS

Question

```
function isNumber(value) {
  return typeof value === 'number';
}

let id = 0;
function getId() {
  return `id-${id++}`
}

function doItLater(callback) {
  return setTimeout(callback, 100);
}
```

1. Is `isNumber` a pure function?

   YES

2. Is `getId` a pure function?

3. Is `doItLater` a pure function?

JS

Question

```
function isNumber(value) {
  return typeof value === 'number';
}

let id = 0;
function getId() {
  return `id-${id++}`
}

function doItLater(callback) {
  return setTimeout(callback, 100);
}
```

1. Is `isNumber` a pure function?

   YES

2. Is `getId` a pure function?

   NO

3. Is `doItLater` a pure function?

JS

Question

```
function isNumber(value) {
  return typeof value === 'number';
}

let id = 0;
function getId() {
  return `id-${id++}`
}

function doItLater(callback) {
  return setTimeout(callback, 100);
}
```

1. Is `isNumber` a pure function?

   YES

2. Is `getId` a pure function?

   NO

3. Is `doItLater` a pure function?

   NO

JS

Question

```
function getUser(name, currentAge) {
  const user = {
    name,
    age: currentAge
  }

  if (user.age < 18) {
    user.yearToAdult = 18 - user.age;
  } else {
    user.yearToAdult = 0;
  }

  return user;
}

const numOfSecondInDay = 24 * 60 * 60;
function getSeconds(numOfDay) {
  return numOfDay * numOfSecondInDay;
}

function getLastItem(array) {
  return array.pop();
}
```

1. Is `getUser` a pure function?
2. Is `getSeconds` a pure function?
3. Is `getLastItem` a pure function?

JS

Question

```
function getUser(name, currentAge) {
  const user = {
    name,
    age: currentAge
  }

  if (user.age < 18) {
    user.yearToAdult = 18 - user.age;
  } else {
    user.yearToAdult = 0;
  }

  return user;
}

const numOfSecondInDay = 24 * 60 * 60;
function getSeconds(numOfDay) {
  return numOfDay * numOfSecondInDay;
}

function getLastItem(array) {
  return array.pop();
}
```

1. Is `getUser` a pure function?

   YES

2. Is `getSeconds` a pure function?

3. Is `getLastItem` a pure function?

JS

Question

```
function getUser(name, currentAge) {
  const user = {
    name,
    age: currentAge
  }

  if (user.age < 18) {
    user.yearToAdult = 18 - user.age;
  } else {
    user.yearToAdult = 0;
  }

  return user;
}

const numOfSecondInDay = 24 * 60 * 60;
function getSeconds(numOfDay) {
  return numOfDay * numOfSecondInDay;
}

function getLastItem(array) {
  return array.pop();
}
```

1. Is `getUser` a pure function?

   YES

2. Is `getSeconds` a pure function?

   YES

3. Is `getLastItem` a pure function?

JS

Question

```
function getUser(name, currentAge) {
  const user = {
    name,
    age: currentAge
  }

  if (user.age < 18) {
    user.yearToAdult = 18 - user.age;
  } else {
    user.yearToAdult = 0;
  }

  return user;
}

const numOfSecondInDay = 24 * 60 * 60;
function getSeconds(numOfDay) {
  return numOfDay * numOfSecondInDay;
}

function getLastItem(array) {
  return array.pop();
}
```

1. Is `getUser` a pure function?

   YES

2. Is `getSeconds` a pure function?

   YES

3. Is `getLastItem` a pure function?

   NO

JS

Why Pure Function is Preferred (If Possible)

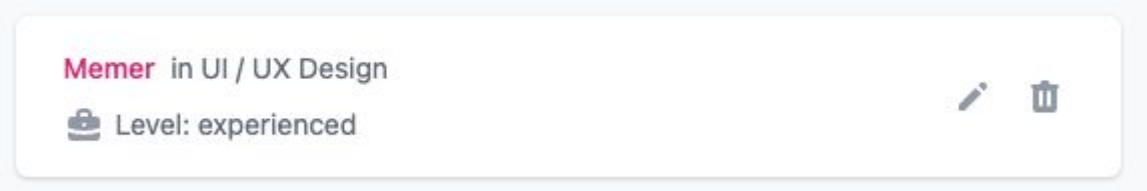
Due to pure function's result is solely based on its inputs:

1. A pure function is easy to test (just pass the input).
2. A pure function is easy to reason (do not need to worry about how a variable is updated by other function).

It is not possible to have a program that does not have side effects (if a program cannot have side effects, it can't even update HTML), but if **we can write pure function for most of our code and only perform side effects minimally** (like getting a framework to do them for us 😉), then most of our code is easy to understand and reason with.

Chapter 4: Component

Demo - 05: Creating Components from HTML



Live version at:
https://transition-to-react-playground.vercel.app/careers

Event Listener in JSX

- to add event listener on element, pass the props **`on<Event>`**

```
<button
  type="button"
  className="js-edit-btn p-1 rounded-full hover:bg-gray-50 focus:
  title="Edit"
  onClick={() => alert("Edit btn clicked, populate the form!")}
>
```


React will translate this to **`addEventListener('click')`**

Component is a function that returns React elements

- Component name must be Pascal case (*Component Rule #1*).
  - ✘ `careeritem`
  - ✘ `careerItem`
  - ✅ `CareerItem`
  - ✅ `Careeritem`

Component must returns single element or `null`

(Component Rule #2)

use `React.Fragment` if you have to render multiple HTML elements.

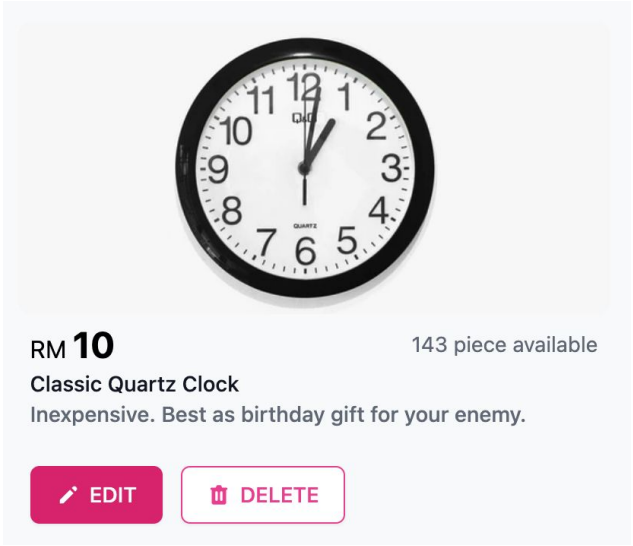
```jsx
export function SomeComponent() {
  // OK
  return (
    <div>
      <div></div>
      <div></div>
    </div>
  );
}

export function InvalidComponent() {
    // NOT OK
    return (
        <div></div>
        <div></div>
    )
}
```

```jsx
export function ValidComponent() {
  // OK
  return (
    <React.Fragment>
      <div></div>
      <div></div>
    </React.Fragment>
  );
}

export function JustLikeValidComponent() {
  // OK
  return (
    <>
      <div></div>
      <div></div>
    </>
  );
}
```

Shorter form of `React.Fragment`

Do It - 05: Convert HTML to Component

Create a **`ListingItem`** component in a **`listing-item.jsx`** file (inside **`src/components`** folder) that renders a listing item like below.

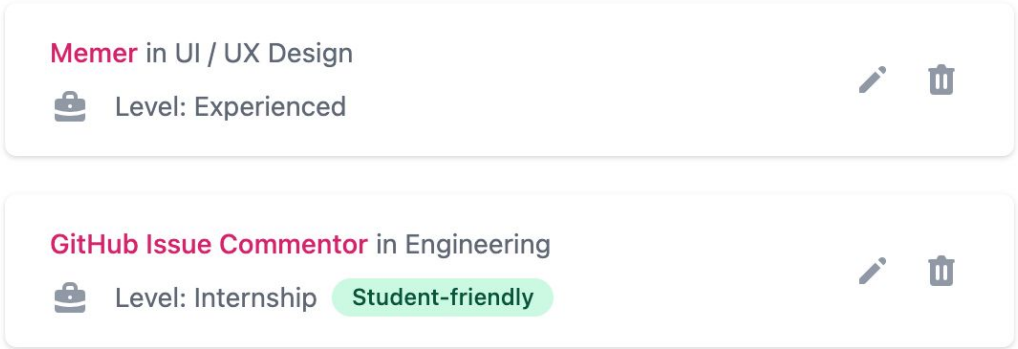


Import the component into **`index.js`** and render it with **`ReactDOM.render`**.

The page should alert message when the buttons on the **`ListingItem`** are clicked.

Live version at:
https://transition-to-react-playground.vercel.app/marketplace

Demo - 06: Props and Component Composition



Live version at:
https://transition-to-react-playground.vercel.app/careers

Composing Components

We can split a components into smaller components.

`CareerItem`

Composing Components

We can split a components into smaller components.

Composing Components

We can split a components into smaller components.

There is no hard-rule on how a components should be split. Usually we pull out a section of a component into a separate component when:

- the component is too big to understand.
- the section can be reused in other components.

Props: Input of Component

Component (a function), can accept one parameter (usually named as **`props`**).

```
<CareerItem
  title="Memer"
  department="UI / UX Design"
  level="Experienced"
  onEdit={() => alert("Edit btn clicked, populate the form!")}
  onDelete={() => alert("Delete btn clicked, delete the item!")}
/>
```

How to pass props to a component

Props: Input of Component

Component (a function), can accept one parameter (usually named as **`props`**).

```jsx
<CareerItem
  title="Memer"
  department="UI / UX Design"
  level="Experienced"
  onEdit={() => alert("Edit btn clicked, populate the form!")}
  onDelete={() => alert("Delete btn clicked, delete the item!")}
/>
```

How to pass props to a component

```jsx
export function CareerItem(props) {
  return (
    <div className="bg-white shadow overflow-hidden sm:rounded-md">
      <div className="px-4 py-4 flex items-center sm:px-6">
        <div className="min-w-0 flex-1 sm:flex sm:items-center sm:justify-between">
          <div>
            <CareerItemTitle
              title={props.title}
              department={props.department}
            />      You, 2 hours ago • add props to components to make them reusable
            <div className="mt-2 flex">
```

How to get the props in the component

Props: Input of Component

Component (a function), can accept one parameter (usually named as **`props`**).

```jsx
<CareerItem
  title="Memer"
  department="UI / UX Design"
  level="Experienced"
  onEdit={() => alert("Edit btn clicked, populate the form!")}
  onDelete={() => alert("Delete btn clicked, delete the item!")}
/>
```

How to pass props to a component

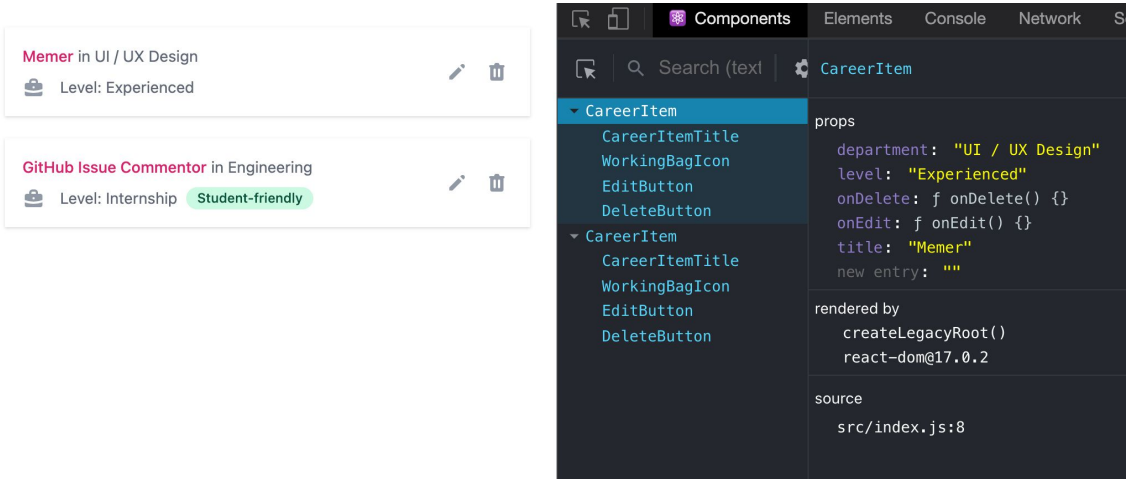
```jsx
export function CareerItem(props) {
  return (
    <div className="bg-white shadow overflow-hidden sm:rounded-md">
      <div className="px-4 py-4 flex items-center sm:px-6">
        <div className="min-w-0 flex-1 sm:flex sm:items-center sm:justify-between">
          <div>
            <CareerItemTitle
              title={props.title}
              department={props.department}
            />      You, 2 hours ago • add props to components to make them reusable
            <div className="mt-2 flex">
```

How to get the props in the component

Use React DevTools to inspect component and props

React DevTools is a browser extension that is useful for React development.

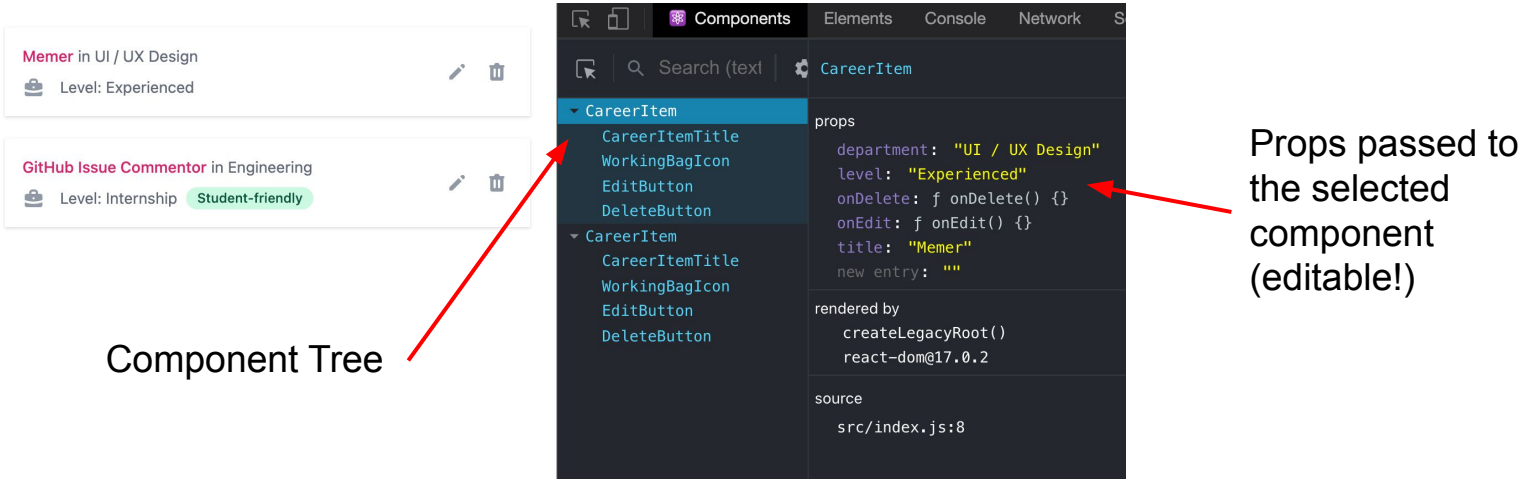
- Chrome (https://bit.ly/install-react-devtools)
- Firefox (https://bit.ly/firefox-react-devtools)



Use React DevTools to inspect component and props

React DevTools is a browser extension that is useful for React development.

- Chrome (https://bit.ly/install-react-devtools)
- Firefox (https://bit.ly/firefox-react-devtools)



Component should be a pure function. *(Component Rule #3)*

- Given same props, component always returns the same result.
- Component cannot have side effect.
  - no API calls, addEventListener, setTimeout etc.
  - do not update props object (read only).

“If” in JSX (1 of 2)

We can assign a variable conditionally using ternary

React will render nothing if it is **`null`**.

```
export function CareerItem(props) {
  const badge = props.studentFriendly ? <span>Student-friendly</span> : null;

  return (
    <div className="flex items-center gap-2 text-sm leading-5 text-gray-500">
      <WorkingBagIcon />
      <span>Level: {props.level} </span>
      {badge}
    </div>
  );
}
```

Put the expression in JSX works too

```
export function CareerItem(props) {
  return (
    <div className="flex items-center gap-2 text-sm leading-5 text-gray-500">
      <WorkingBagIcon />
      <span>Level: {props.level} </span>
      {props.studentFriendly ? <span>Student-friendly</span> : null}
    </div>
  );
}
```

“If” in JSX (2 of 2)

```jsx
export function CareerItem(props) {
  const badge = props.studentFriendly && <span>Student-friendly</span>;

  return (
    <div className="flex items-center gap-2 text-sm leading-5 text-gray-500">
      <WorkingBagIcon />
      <span>Level: {props.level} </span>
      {badge}
    </div>
  );
}
```

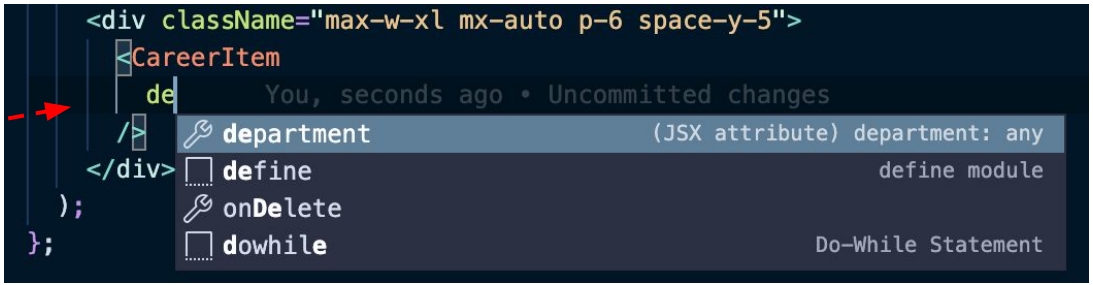
React will render nothing if it is **`null`**, **`undefined`**, or **`boolean`**.

```jsx
export function CareerItem(props) {
  return (
    <div className="flex items-center gap-2 text-sm leading-5 text-gray-500">
      <WorkingBagIcon />
      <span>Level: {props.level} </span>
      {props.studentFriendly && <span>Student-friendly</span>}
    </div>
  );
}
```

Destructuring the props in the component allows intellisense.

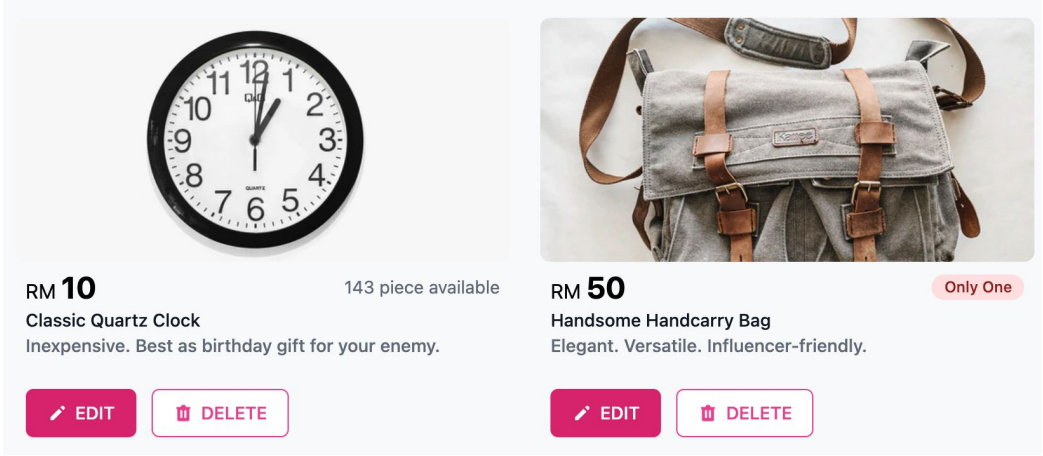
```
export function CareerItem({
  title,
  department,
  level,
  studentFriendly,
  onEdit,
  onDelete,
}) {
  return (
    <div className="bg-white shadow overflow-hidden sm:rounded-md">
      <div className="px-4 py-4 flex items-center sm:px-6">
```

Because we destructure the props in the component, the editor can suggest the props while you type.



Do It - 06: Props and Component Composition

1. Breaking down the components and make the components accept props.
2. Create a **`Marketplace`** component in a new file **`src/pages/marketplace.jsx`** that returns two **`ListingItem`** components like below.
3. Import the **`Marketplace`** component in **`index.js`** and render it.

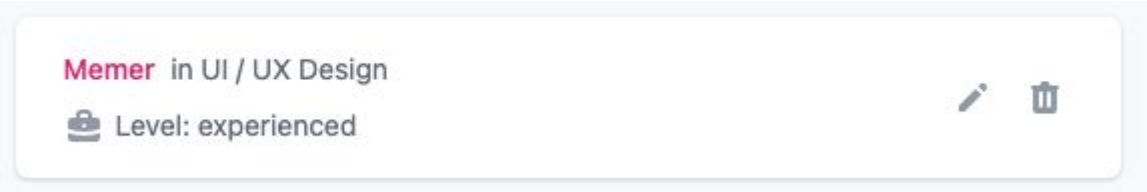


Install React DevTools at:

- Chrome (https://bit.ly/install-react-devtools)
- Firefox (https://bit.ly/firefox-react-devtools)

Live version at:
https://transition-to-react-playground.vercel.app/marketplace

Demo - 07: React element as props value and children props



Live version at:
https://transition-to-react-playground.vercel.app/careers

props value can be any type of JS expression, including React element

```jsx
const IconButton = (props) => (
  <button
    type="button"
    className="p-1 rounded-full hover:bg-gray-50
    title={props.title}
    onClick={props.onClick}
  >
    {props.icon}
  </button>
);
```

```jsx
const EditButton = (props) => {
  return (
    <IconButton
      title="Edit"
      onClick={props.onClick}
      icon={
        <svg
          className="h-5 w-5 text-gray-400"
          fill="currentColor"
          viewBox="0 0 20 20"
          xmlns="http://www.w3.org/2000/svg"
        >
          <path d="M13.586 3.586a2 2 0 112.828 2.828
        </svg>
      }
    />
  );
};
```

We can pass React element (JSX) as props value, because there are just JS object.

Content between open and close tag of a Component will be passed as **`children`** props.

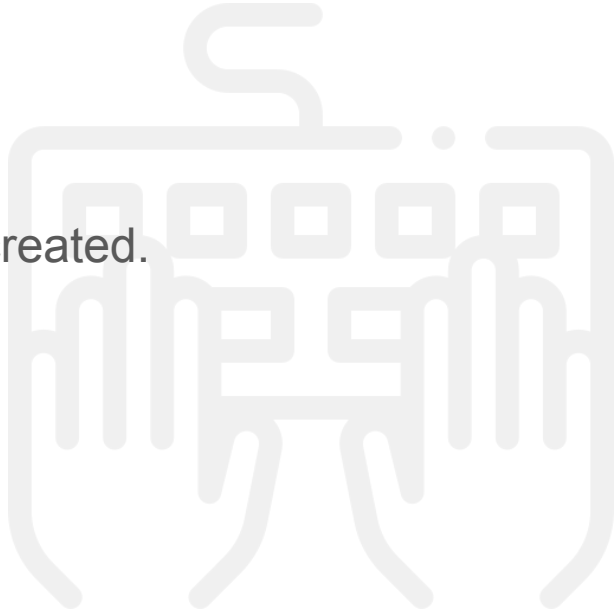
```jsx
const IconButton = (props) => (
  <button
    type="button"
    className="p-1 rounded-full hov
    title={props.title}
    onClick={props.onClick}
  >
    {props.children}
  </button>
);
```

```jsx
const EditButton = (props) => {
  return (
    <IconButton title="Edit" onClick={props.onClick}>
      <svg
        className="h-5 w-5 text-gray-400"
        fill="currentColor"
        viewBox="0 0 20 20"
        xmlns="http://www.w3.org/2000/svg"
      >
        <path d="M13.586 3.586a2 2 0 112.828 2.828l-.793.79
      </svg>
    </IconButton>
  );
};
```

This allows us to create “wrapper” component, which renders a wrapper elements around the elements within it.

Do It - 07: use children props.

Use `children` props in the components that you've created.



Demo - 08: Rendering Array

Data: https://bit.ly/react-jobs-data

When rendering react elements based on Array data, we can use `[].map` to generate list of React elements.

```jsx
export const Career = () => {
  return (
    <div className="max-w-xl mx-auto p-6 space-y-5">
      {jobs.map((job) => (
        <CareerItem
          title={job.title}
          department={job.department}
          level={job.level}
          onEdit={() => alert("Edit btn clicked, populate the form!")}
          onDelete={() => alert("Delete btn clicked, delete the item!")}
        />
      ))}
    </div>
  );
};
```

`[].map` method will create a new Array instead of update it, therefore it is fine to be used in component (which need to be a pure function).

When rendering react elements based on Array data, we can use `[].map` to generate list of React elements.

```jsx
export const Career = () => {
  return (
    <div className="max-w-xl mx-auto p-6 space-y-5">
      {jobs.map((job) => (
        <CareerItem
          title={job.title}
          department={job.department}
          level={job.level}
          onEdit={() => alert("Edit btn clicked, populate the form!")}
          onDelete={() => alert("Delete btn clicked, delete the item!")}
        />
      ))}
    </div>
  );
};
```

`[].map` method will create a new Array instead of update it, therefore it is fine to be used in component (which need to be a pure function).

However, you will see an error in your console.

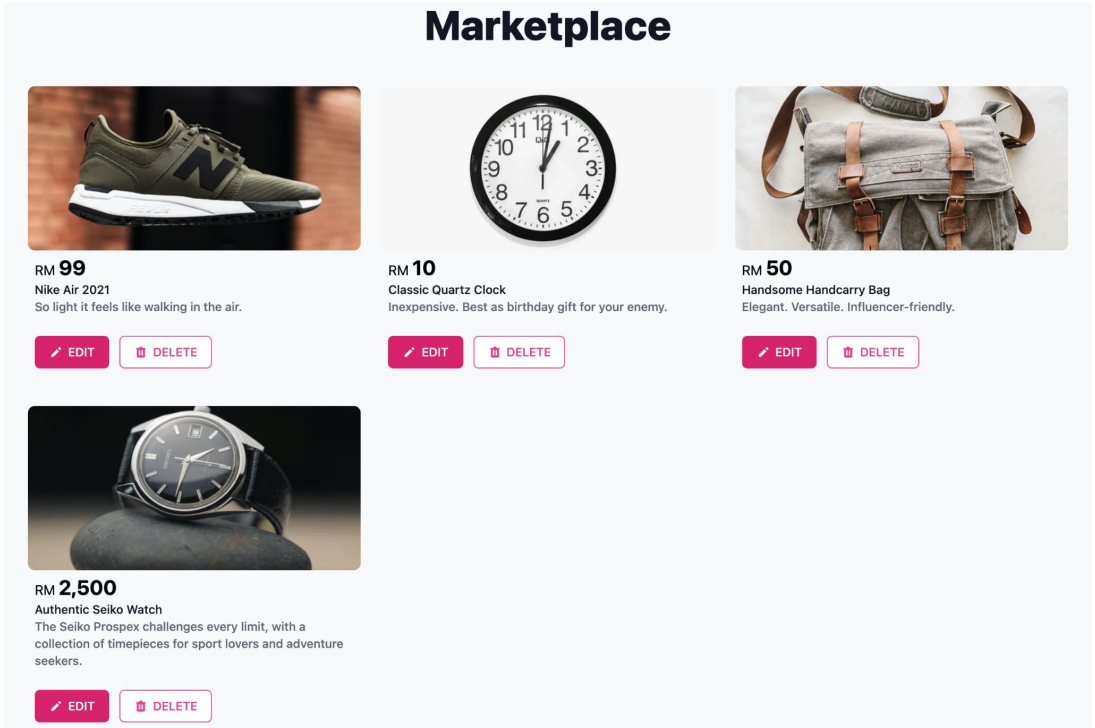
```
▶Warning: Each child in a list should have a unique "key" prop.

Check the render method of `Career`. See https://reactjs.org/link
    at CareerItem (http://localhost:3000/static/js/main.chunk.js:
    at Career
```

To fix the error, pass an unique value to **`key`**.

```
export const Career = () => {
  return (
    <div className="max-w-xl mx-auto p-6 space-y-5">
      {jobs.map((job) => (
        <CareerItem
          title={job.title}
          department={job.department}
          level={job.level}
          onEdit={() => alert("Edit btn clicked, populate the form!")}
          onDelete={() => alert("Delete btn clicked, delete the item!")}
          key={job._id}
        />
      ))}
    </div>
  );
};
```

`key` must be unique among the siblings.

Do It - 08: Rendering Listing Item



Get sample data from:
https://bit.ly/react-listing-data

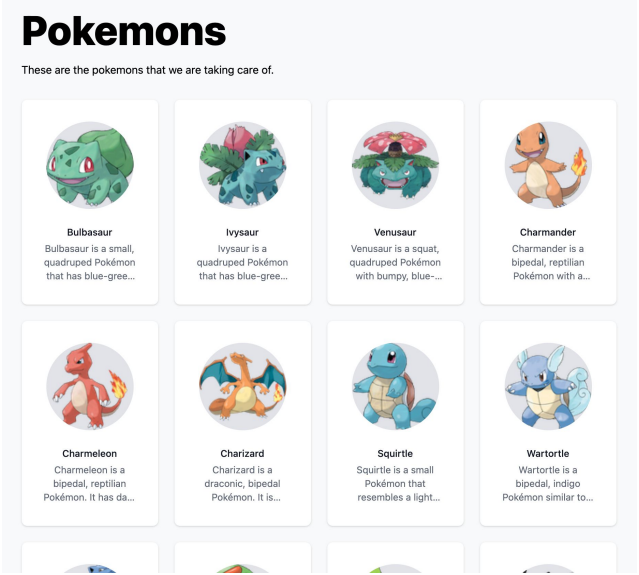
Live version at:
https://transition-to-react-playground.vercel.app/marketplace

Recap Chapter 4: Component

1. Component is a function that accept a props parameter and with the following rules:
   a. Component name must be in Pascal case
   b. Component must return single React element or `null`.
   c. Component should be a pure function
2. We can use React Devtools extension to inspect element props and structure
3. A component can be further broken down into smaller components.
4. Content between opening tag and closing tag in JSX will be passed as `children` props to the component.
5. We can use `Array.map` to render React elements from data in Array. An unique value must be passed as `key` when we do so.

Take Home Homework

Create a new react project using CRA and recreate the Pokemons page (**https://transition-to-react-playground.vercel.app/pokemons**).

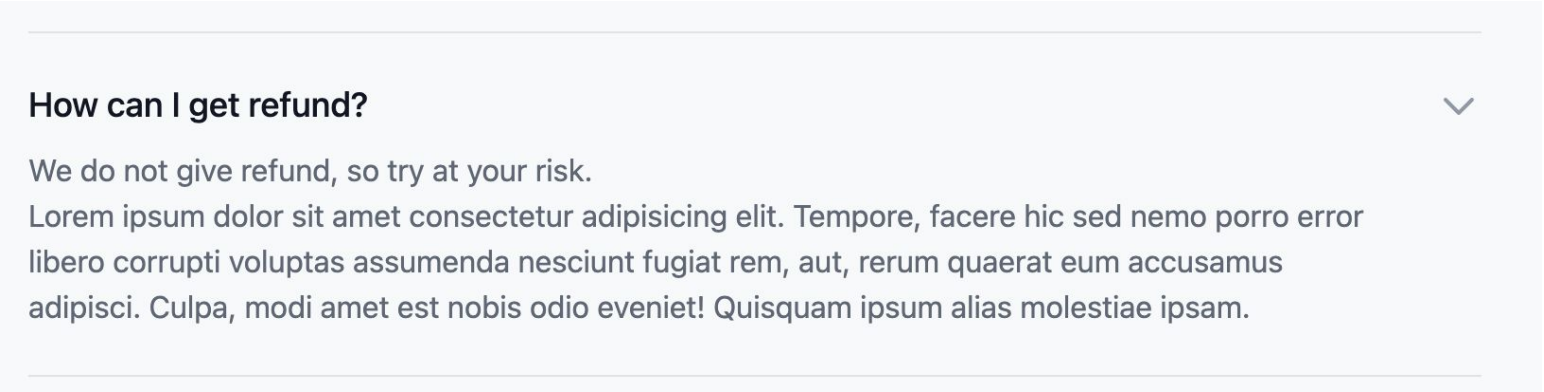


Instructions:

- You do not need to fetch the data in the code. Instead, get the pokemon data from the following link and paste it as a variable in your code: https://bit.ly/pokemon-data
- Try to replicate the UI as much as possible.
- You do not need to include header and footer.

Chapter 5: `useState`

Demo - 09: Creating Accordion in React



Live version at:
https://transition-to-react-playground.vercel.app/faq

Base files:
https://gist.github.com/malcolm-kee/422ca3ffb0a1ff36a89bf13e95787df3

`useState` allows us to declare a variable that can be changed over time.

```
const state = React.useState(defaultValue);
const stateValue = state[0]; // current value of the state
const stateSetter = state[1]; // function to update state value
```

`useState` allows us to declare a variable that can be changed over time.

```
const state = React.useState(defaultValue);
const stateValue = state[0]; // current value of the state
const stateSetter = state[1]; // function to update state value
```

Use array destructuring to make it shorter.

```
const [stateValue, stateSetter] = React.useState(defaultValue);
```

`useState` allows us to declare a variable that can be changed over time.

```
const state = React.useState(defaultValue);
const stateValue = state[0]; // current value of the state
const stateSetter = state[1]; // function to update state value
```

Use array destructuring to make it shorter.

```
const [stateValue, stateSetter] = React.useState(defaultValue);
```

Call the stateSetter function with the latest value you want it to be

```
stateSetter(newValue);
```

`useState` allows us to declare a variable that can be changed over time.

```
const state = React.useState(defaultValue);
const stateValue = state[0]; // current value of the state
const stateSetter = state[1]; // function to update state value
```

Use array destructuring to make it shorter.

```
const [stateValue, stateSetter] = React.useState(defaultValue);
```

Call the stateSetter function with the latest value you want it to be

```
stateSetter(newValue);
```

Note that you can name the state according to your use case.

```
export const Accordion = ({ title, children }) => {
  const [isExpanded, setIsExpanded] = React.useState(false);
```

Rules of hooks

- `useState` is a hook provided by React, and there are a few hooks provided by React (which we will cover later on). All the hooks are functions with name starts with `use`.

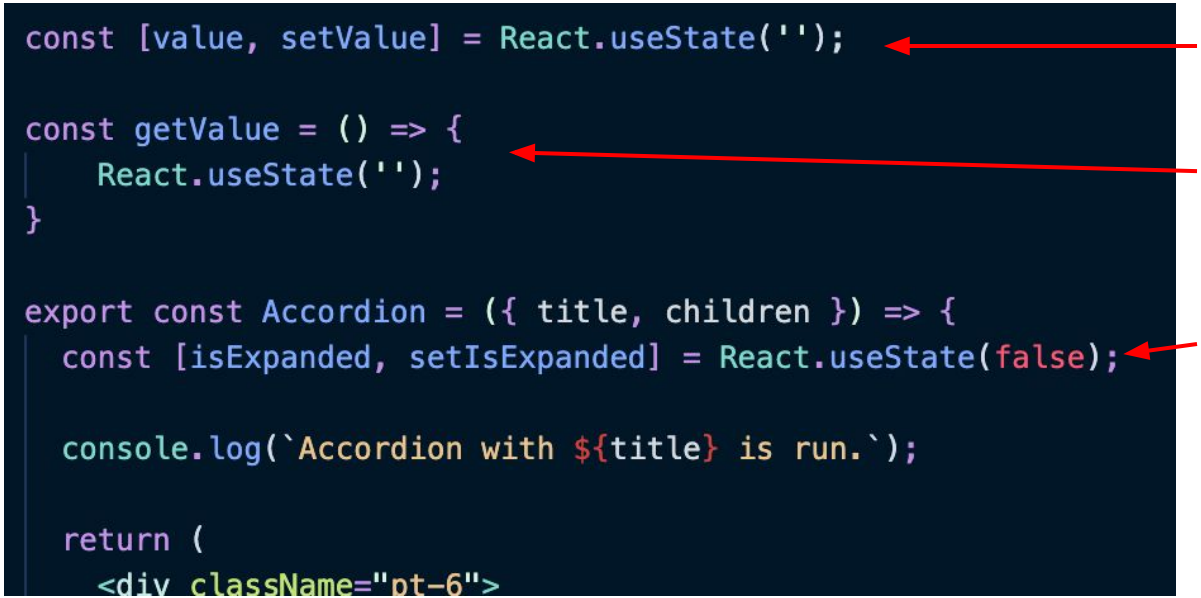
- When using hooks, we must obey two rules:
  1. Only call hooks in React functions (Component or *custom hooks*)
  2. Only call hooks in top level - don't call hooks in loop, condition, or nested function.

We will cover custom hooks later on.

Examples of valid/invalid usage (1 of 3)

```
const [value, setValue] = React.useState('');

const getValue = () => {
    React.useState('');
}

export const Accordion = ({ title, children }) => {
  const [isExpanded, setIsExpanded] = React.useState(false);

  console.log(`Accordion with ${title} is run.`);

  return (
    <div className="pt-6">
```

Examples of valid/invalid usage (1 of 3)



```
const [value, setValue] = React.useState('');

const getValue = () => {
    React.useState('');
}

export const Accordion = ({ title, children }) => {
  const [isExpanded, setIsExpanded] = React.useState(false);

  console.log(`Accordion with ${title} is run.`);

  return (
    <div className="pt-6">
```

invalid, as it is not in component

invalid, as getValue is not a component

valid, used in top level of component

Examples of valid/invalid usage (2 of 3)

```
const Toggle = ({isOpen}) => {
    if (isOpen) {
        const [value, setValue] = React.useState('');
        return <input value={value} onChange={ev => setValue(ev.target.value)} />
    }

    return <div />
}

const List = ({items}) => {
    return <div>
        {items.map(item => {
            const [isOpen] = React.useState(false)

            return <Toggle isOpen={isOpen} item={item} />
        })}
    </div>
}
```

Examples of valid/invalid usage (2 of 3)

```
const Toggle = ({isOpen}) => {
    if (isOpen) {
        const [value, setValue] = React.useState('');
        return <input value={value} onChange={ev => setValue(ev.target.value)} />
    }

    return <div />
}

const List = ({items}) => {
    return <div>
        {items.map(item => {
            const [isOpen] = React.useState(false)

            return <Toggle isOpen={isOpen} item={item} />
        })}
    </div>
}
```

invalid, as it is run conditionally

invalid, as it is run in a loop

Examples of valid/invalid usage (3 of 3)

```
const Details = ({details, show}) => {
    if (!show) {
        return null;
    }

    const [postTax] = React.useState('');

    return <div>{details} {postTax}</div>
}
```

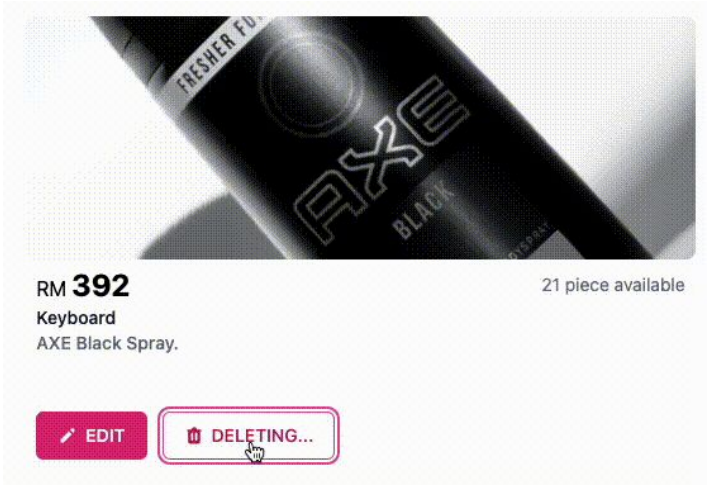
Examples of valid/invalid usage (3 of 3)

```
const Details = ({details, show}) => {
    if (!show) {
        return null;
    }

    const [postTax] = React.useState('');

    return <div>{details} {postTax}</div>
}
```

invalid, as it is run conditionally

Do It - 09: Adding State to Component

Add state to **`ListingItem`** so that when u click the DELETE button, the text will be changed to “DELETING…” and will be changed back to DELETE after u click again.



Demo - 10: Implement a number control with validation

Link to base component:
https://bit.ly/number-control

Live version at:
https://transition-to-react-playground.vercel.app/careers

We can have more than one states in a component by calling **`useState`** multiple times.

```
export const NumberControl = () => {
  const [value, setValue] = React.useState(1);
  const [error, setError] = React.useState("");

  return (
    <div className="mt-1 sm:mt-0 sm:col-span-2">
      <div className="relative w-32">
        <button
          type="button"
          className="absolute left-0 inset-y-0 px-1.5 text-gray-400"
          onClick={() => {
```

The states are independent of each other.

Demo - 11: Number control with editable input

Live version at:
https://transition-to-react-playground.vercel.app/careers

Add **`onChange`** on input so you can update state when input value change.

```
<input
  type="text"
  name="headcount"
  required
  className="block w-full px-9 text-
  value={value}
  onChange={(ev) => {
    setValue(ev.target.value)
  }}
/>
```

`ev.target` gives us the input. Thus **`ev.target.value`** get us the value of the input.

Passing state value to **`value`** props and setState in **`onChange`** callback is how we usually “bind” a state value to an **`input`** element.

React is mostly just javascript, so JS behavior like coercion would happens as well.

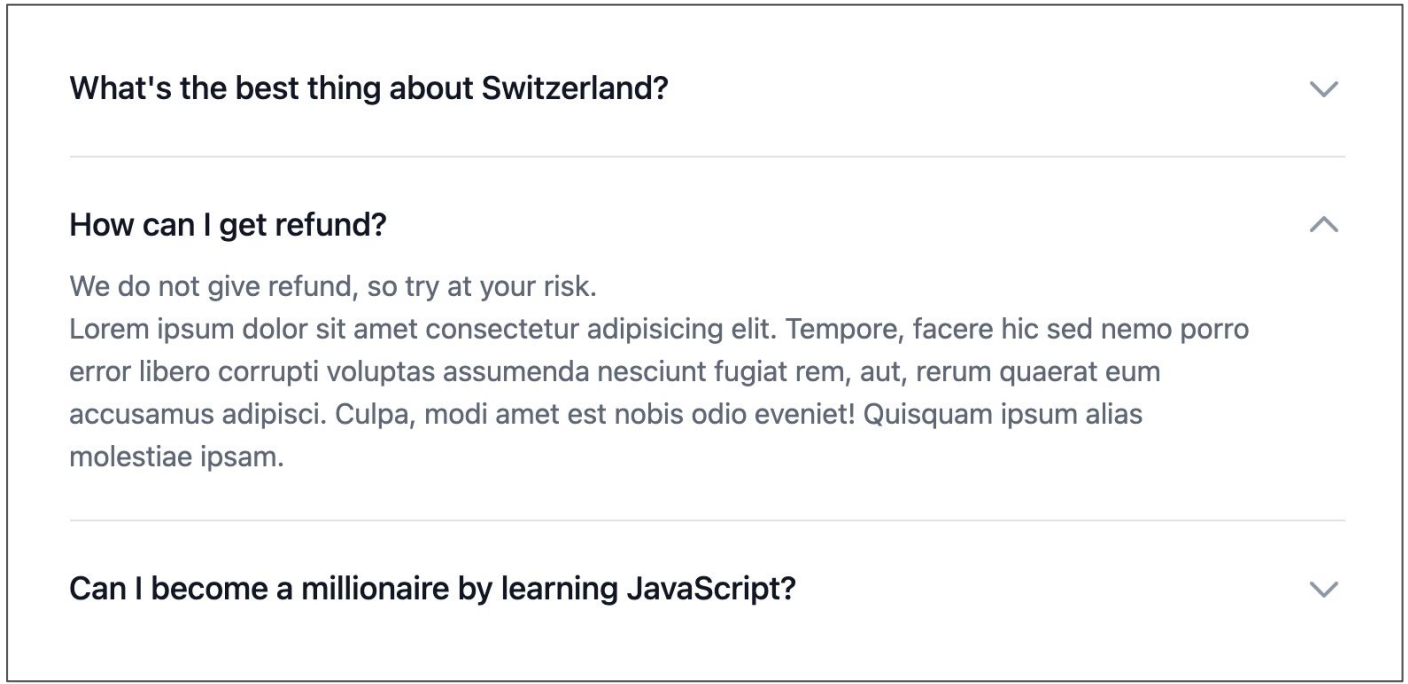
```
<input
  type="text"
  name="headcount"
  required
  className="block w-full px-9 text-center s
  value={value}
  onChange={(ev) => {
    const value = Number(ev.target.value);

    if (!isNaN(value)) {
      setValue(value);
    }
  }}
/>
```

`ev.target.value` get us the value of the input as string. So we need to convert it.
The conversion may return **`NaN`** (as user may type non-numerical value), so need to check first before **`setValue`**.

If a value can be computed from another value, then that should be computed instead of a state.

- A derived value (like error in the example) should be computed in the component.
- Having derived value as state introduces more complexity as we need to make sure the states are in sync.

Demo - 12: Accordion with single expandable content



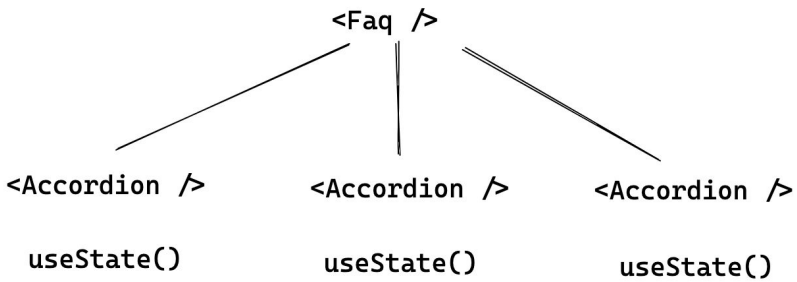
How to make maximum only one Accordion is expanded?

Lifting State

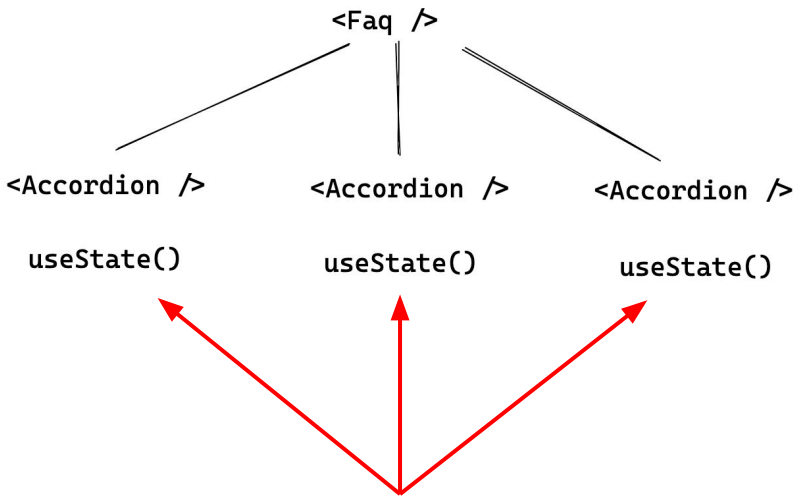
To share state between sibling components, we have to lift the state:

1. find the common parent component of the components and declare state in that parent component
2. pass the state value and the callback to the child components so they can update the state.

Before

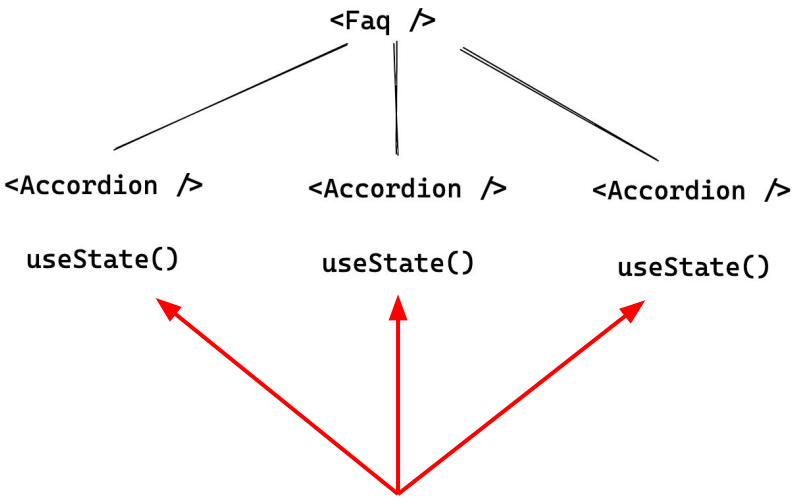

<Faq />
<Accordion />
useState()
<Accordion />
useState()
<Accordion />
useState()

Before



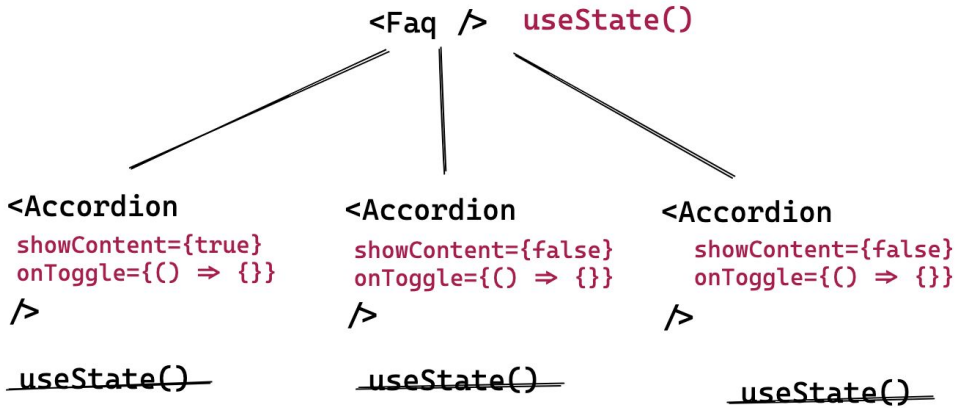
There is no way for those state to update each other.

Before

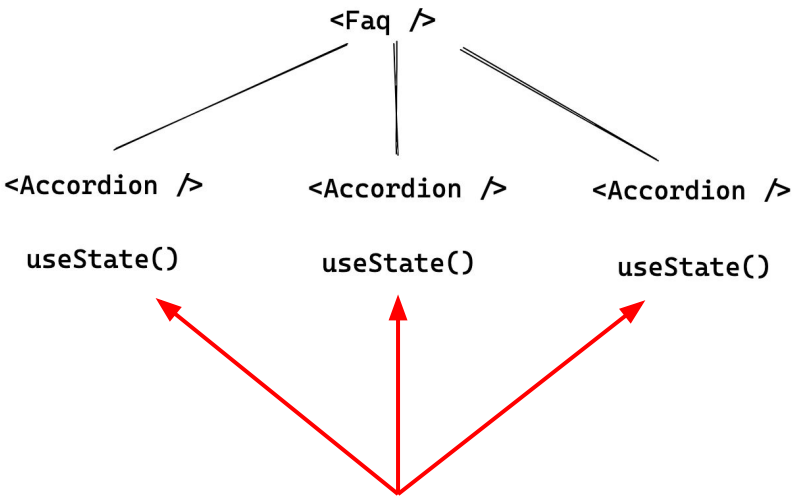


There is no way for those state to update each other.

After



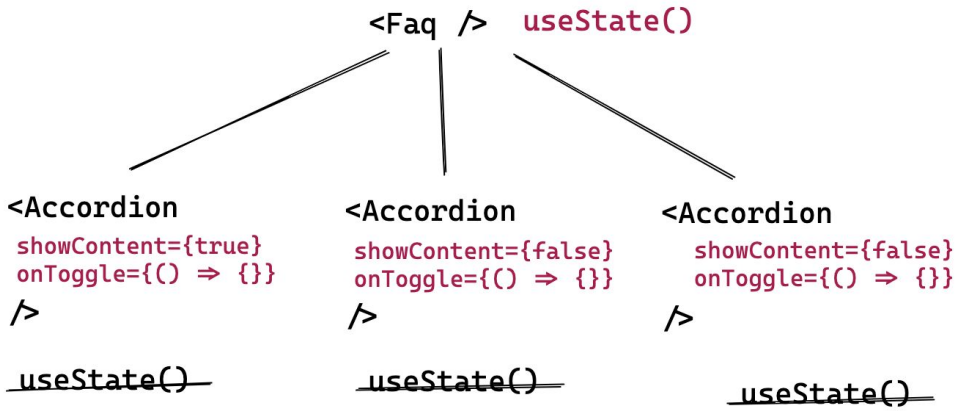
Before



There is no way for those state to update each other.

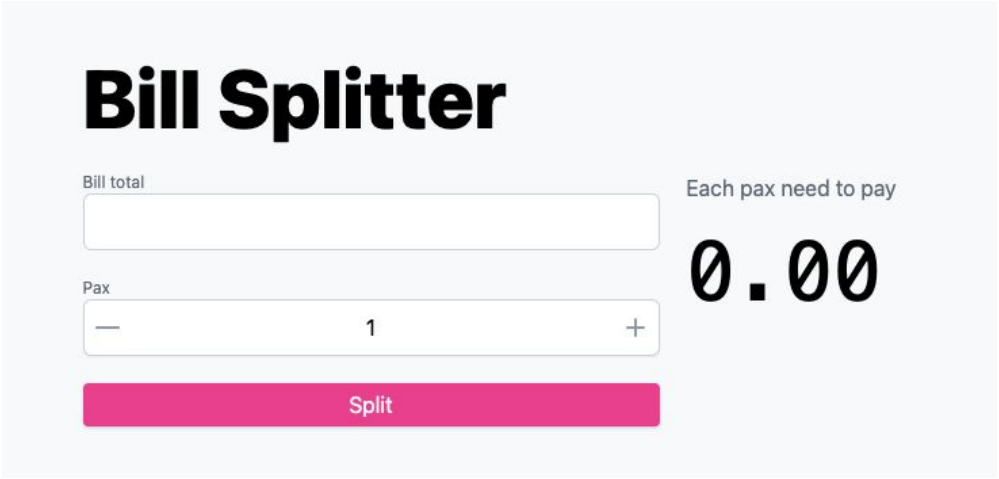
After

Parent owns the state, and pass the callback to children to update it.



Do It - 10: Bill Splitter

Implement Bill Splitter like bit.ly/js-bill-splitter using React.



Resources:

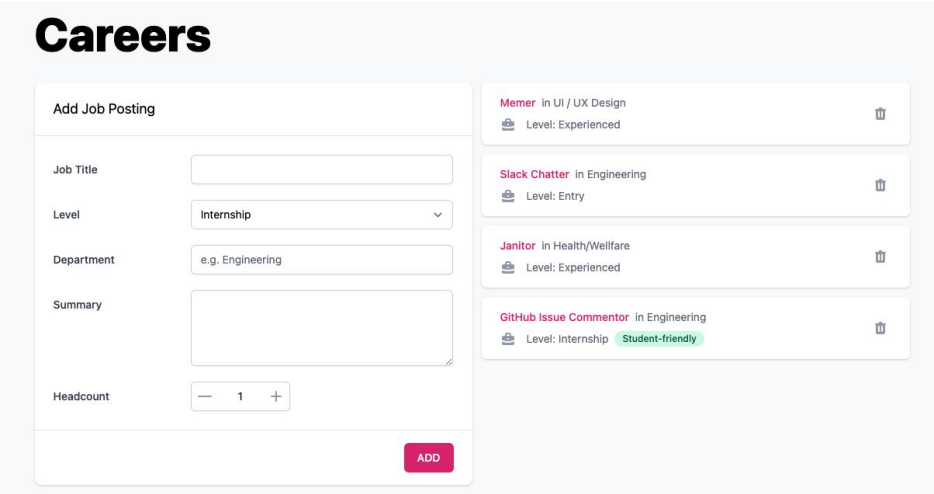
- Starting file to copy as **`pages/bill-splitter.jsx`**: bit.ly/start-bill-splitter
- Alternative, use another codesandbox: bit.ly/bill-splitter-sandbox

Recap Chapter 5: useState

1. React provides a few hooks that allow us to add functionality to component.
2. When using hooks, we must:
   a. Only call hooks in React functions (Component or custom hooks)
   b. Only call hooks in top level - don't call hooks in loop, condition, or nested function.
3. `useState` is a hook that add state (a variable that can be change over time) to our component.
4. The way to manage states across components is to lift the state to their common parents.
5. We can get the latest value when user type in a text input with `event.target.value` from `onChange` callback.

Take Home Homework (Day 1)

Create a new react project using CRA and recreate the Careers page (**bit.ly/react-day1-exercise**).



Resources:

- Read and understand the code for this todo list application, which would help you to understand how to have an array as state (https://codesandbox.io/s/react-to-do-3j4dl)
- If you can't make sense of the code above, then you can go through this step-by-step tutorial.