

Intro to React

Lab 1

Fundamentals

1	ONLINE REFERENCES	1
1.1	BASIC REFERENCES.....	1
1.2	INTERMEDIATE REFERENCES.....	2
1.3	OFFICIAL REFERENCES	2
2	LAB SETUP	2
3	GENERATING A REACT PROJECT USING CREATE REACT APP (CRA).....	2
4	COMPONENT BASICS.....	4
4.1	NESTING COMPONENTS	5
4.2	EXPORTING AND IMPORTING COMPONENTS.....	7
4.3	USING BROWSER EXTENSIONS (REACT DEVELOPER TOOLS)	8
4.4	USING JSX IN COMPONENTS.....	8
4.5	EMBEDDING VARIABLES / EXPRESSIONS IN JSX	9
5	PROPS	10
5.1	ACCESSING PROPS VIA PARAMETER DESTRUCTURING	10
5.2	USING CHILDREN PROPERTY TO ACCESS CONTENT BETWEEN JSX TAGS.....	11
5.3	USING SPREAD OPERATOR SYNTAX FOR JSX	11
5.4	CONDITIONAL RENDERING.....	11
5.5	RENDERING LISTS	12
5.6	COMPONENTS AS PURE FUNCTIONS.....	12
6	CREATE REACT PROJECT FOLDER STRUCTURE	13
7	STYLING IN REACT	16
7.1	USING INLINE STYLES.....	16
7.2	IMPORTING A SEPARATE STYLE SHEET.....	16
7.3	USING CSS-IN-JS (STYLED-COMPONENTS)	16
7.4	ACCESSING IMAGES IN A REACT APP	17
7.5	OTHER APPROACHES TO STYLING	17

1 Online References

1.1 Basic references

<https://www.w3schools.com/REACT/DEFAULT.ASP>

1.2 Intermediate references

<https://ibaslogic.com/react-tutorial-for-beginners>

<https://www.geeksforgeeks.org/reactjs-components/>

<https://scrimba.com/learn/learnreact>

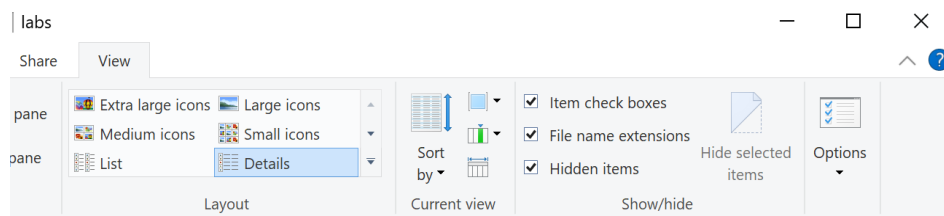
1.3 Official references

<https://react.dev/learn>

2 Lab Setup

You should have received installation instructions on setting up Visual Studio Code IDE as well as installing Create React App (CRA).

We will need to be able to view file name extensions directly in order to be able to directly manipulate JavaScript source code files. If you are using Windows, make sure that you have checked the File name extensions in your File Explorer in order for us to be able to directly manipulate the file extensions when creating or modifying files.



For MacOS:

[Show or hide file name extensions - Article 1](#)

[Show or hide file name extensions - Article 2](#)

Create a suitable top level folder (for e.g. `react-labs`) to hold the various React project folders that you will be generating in this lab.

3 Generating a React project using Create React App (CRA)

We can generate a React app using Create React App (CRA). CRA is a CLI tool that generates boilerplate code for creating a React project structure along with the latest configuration features that help in building the app as well as tools to optimize the app for production deployment. CRA is linked to a specific build tool (Webpack) and it will auto-generate files for this type of build.

Webpack is a static module bundler for modern JavaScript applications. Many modern JavaScript libraries (such as React) or frameworks (such as Angular) have many module dependencies: Webpack helps to combine all these modules into a smaller number of bundles to make it easier to deliver to a browser.

CRA also provides a built-in development server that dynamically rerenders the app everytime a change that is made to the relevant source code files.

In the top-level folder that you created for this workshop, open a command prompt and generate a new project using CRA with:

```
create-react-app first-app
```

Open a command prompt in the newly generated project folder `first-app`. Then type

```
npm start
```

to start the development server, which will then serve up the newly generated app in the browser at `localhost:3000` (this is the default port for the server)

This generated app has a default view which incorporates the React logo as shown below:



Use VS Code to open the project folder and navigate into the `src` subfolder of the generated project folder (`first-app`) which contains the various JavaScript and CSS files that are responsible for the view currently being rendered. We will be changing the content of some of these files in to illustrate the various features / concepts of React in the subsequent lab sessions.

In this (and subsequent lab sessions), we will be demonstrating various features / concepts of React using the source code files from this single project. However, sometimes you may wish to compare features / concepts demonstrated in different lab sessions to solidify your understanding of them and how they relate and support each other.

In that case, you can create another project in the same way.

Open another command prompt in the top level folder that you created for this workshop, and generate another project using CRA with a different project name, for e.g.

```
create-react-app second-app
```

Open a command prompt in this project folder. Since there is already an existing development server running at the default port 3000, we will now specify a different port for the next server. Here, we use 5000, but you can use any port that is free on your local machine:

```
SET PORT=5000
```

```
npm start
```

The newly generated app should provide the same default view as previously.

Create another instance of VS Code (File -> New Window) to open this project folder and navigate into the `src` subfolder as we did previously.

4 Component basics

Project: `first-app`

The `App.js` in `src` subfolder contains the root React component which currently has a predefined boilerplate code generated for it. We will replace it with our own versions to demonstrate a simple use of a component.

You can view the DOM tree in the browser view for the React app in the Elements tab of the Console Developer tools.

Demo basic component

Folder to use: `components`

File to use: `App-v1.js`

Demo component with basic CSS style rules

Folder to use: `components`

File to use:

`App-v2.js`

`App-v2.css`

Key points regarding components:

- Components encapsulate HTML, CSS styling and JavaScript logic into reusable UI elements.
- `App.js` is the root component file, which contains the main function component that will be rendered by default (the root component)
- The name of the root component in `App.js` which is associated with the `export default` statement can be also `App`, or it can be any other name
- The `export default` can be used directly together with the definition of the component function, or can be provided as a separate statement:

```
function App() {  
  return (  
    <h1>Hello there</h1>  
  );  
}  
export default App;
```

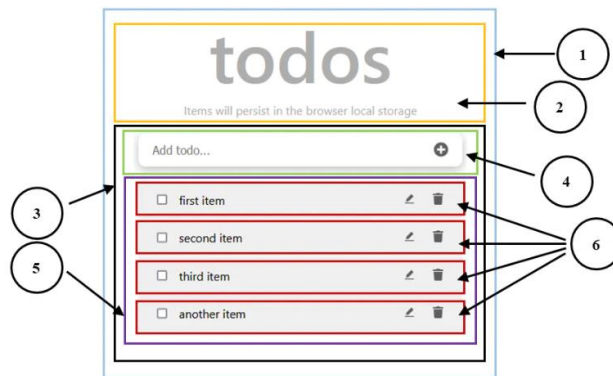
```
export default function App() {  
  return (  
    <h1>Hello there</h1>  
  );  
}
```

The final `return` statement in the component function must be renderable content. This is nearly most of the time JSX (to be covered in more detail in an upcoming lab). Some points:

- a) JSX content must be wrapped in a pair of parenthesis if it spans multiple lines.
- b) If you return multiple HTML elements, you must enclose all of them with a single top level parent tag (e.g. `<div>`)
- c) As an alternative to returning JSX in the final return statement, you can also return strings or numbers as well as arrays that hold JSX elements or strings or numbers.

4.1 Nesting components

Components are reusable UI elements which form the building blocks of a React project, whereby various smaller components are combined to form the larger overall UI of the web app. Consider the sample UI below:

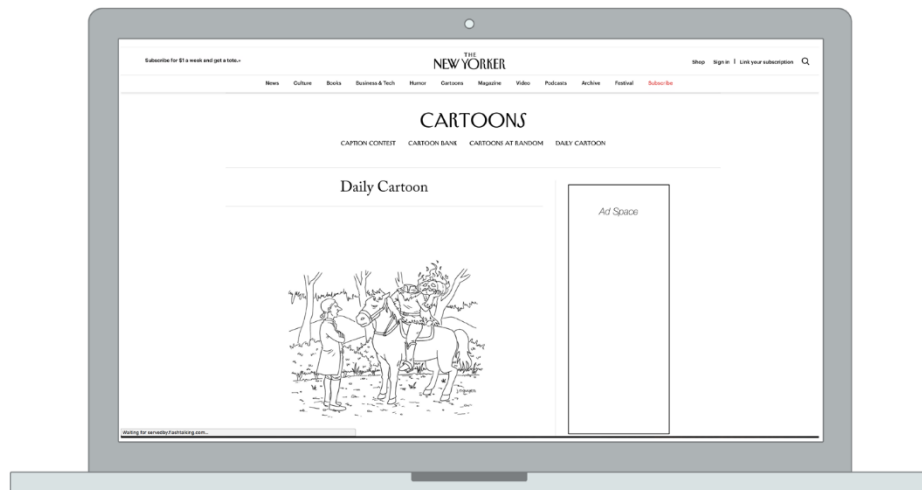
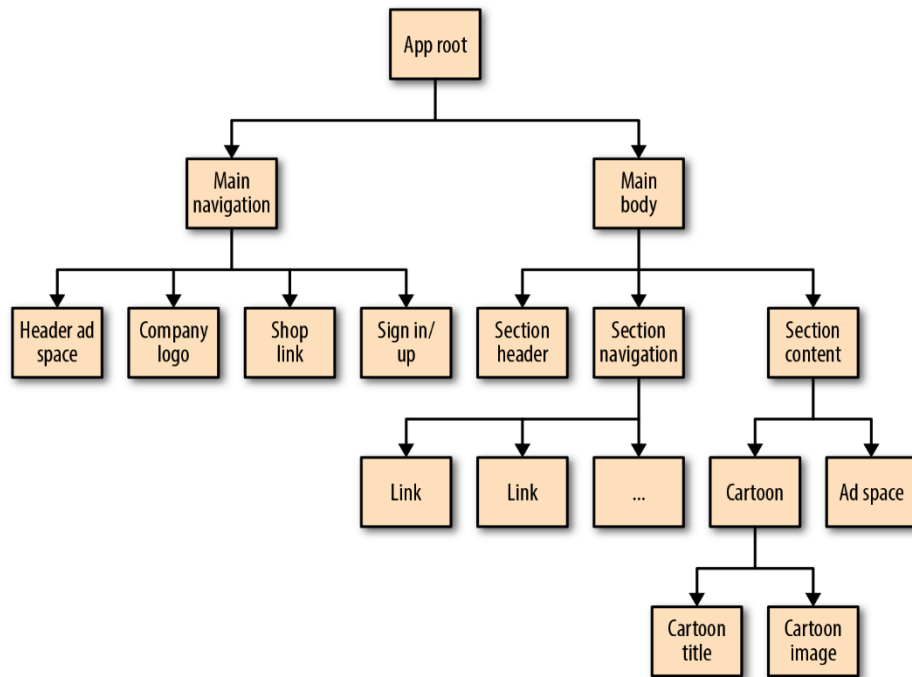
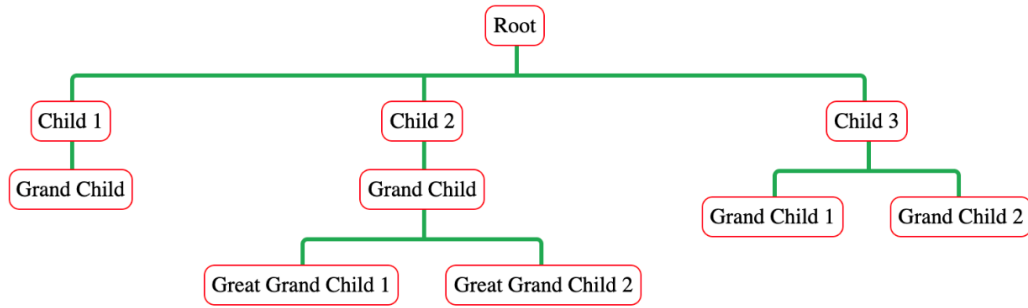


The main UI above can be built by decomposing it into a component hierarchy:

1. `TodoApp`: the parent or root component. It holds two direct child components (`Header` and `TodosLogic`)
2. `Header`: display the todos heading text.
3. `TodosLogic`: contains the application logic. It includes two direct child components (`InputTodo` and `TodosList`)
4. `InputTodo`: will take the user's input.
5. `TodosList`: serves as a parent container for the multiple `TodoItem` components
6. `TodoItem`: renders the individual todos item.

Components can be nested within each other by having the JSX returned by a parent component containing other function components (the child components), whereby these components are referenced using JSX tags (similar to conventional HTML tags)

The component hierarchy formed from the nesting of components to any arbitrary depth would look something like this:



There are two types of components: built-in components and custom components

- a) Built-in components are JSX tags that correspond to standard HTML elements (such as `<div>`, `<h1>`, `<section>`, etc) which are ultimately translated into JavaScript that directly manipulates the DOM. They can be used directly without any definition.
- b) Custom function components are defined by the user and must have names that start with a capital letter when they are embedded in JSX when nesting components. This is in order to distinguish it from built-in components. Names typically follow PascalCase naming convention

There can be multiple components (child and parent components) in `App.js`, in which case the main one that will be used in rendering is identified with `export default`.

Child components can be defined as arrow functions, but not main components that are associated with `export default` (they must have a name in order to be exported)

Each child component definition should be provided at the top level of JavaScript file (`App.js`) containing the parent component or in a separate file and then imported. You should NOT define the child component as an inner function of the parent component

Project: `first-app`

Demo basic nesting of components

Folder to use: `components`

File to use: `App-v3.js`

4.2 Exporting and importing components

Function components can be defined in either `App.js` or separate JavaScript file. Components in separate JavaScript files can be imported into other component files that require them as a dependency. The standard convention is to store component definition files in `src/components` folder.

Project: `first-app`

Demo export / import of components with default export

Folder to use: `components`

File to use: `App-v4.js`

`Gallery-v4.js` (create new `Gallery.js` in `src` subfolder)

Demo export / import of components with a mixture of named and default exports

Folder to use: `components`

File to use: `App-v5.js`

`Gallery-v5.js`

More info on named and default imports / exports

<https://react.dev/learn/importing-and-exporting-components#default-vs-named-exports>

4.3 Using browser extensions (React Developer Tools)

Most major browsers (Chrome, Firefox, Edge, etc) provide extensions that enhance the functionality of the particular browser. There are several extensions that exist which enhance the development / debugging workflow in building a React app via the development server.

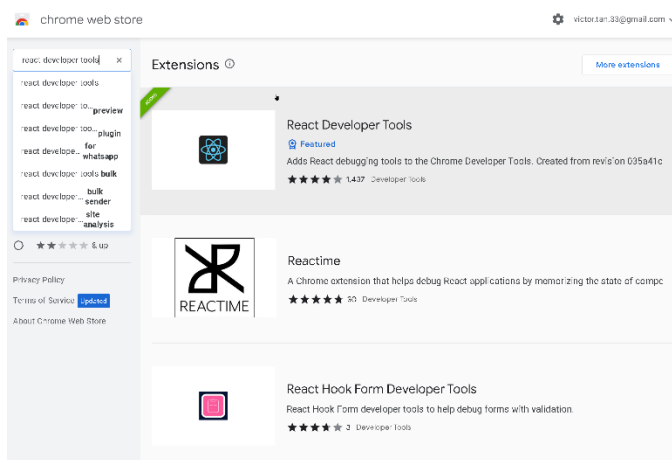
The most well known one is React Developer Tools

<https://react.dev/learn/react-developer-tools>

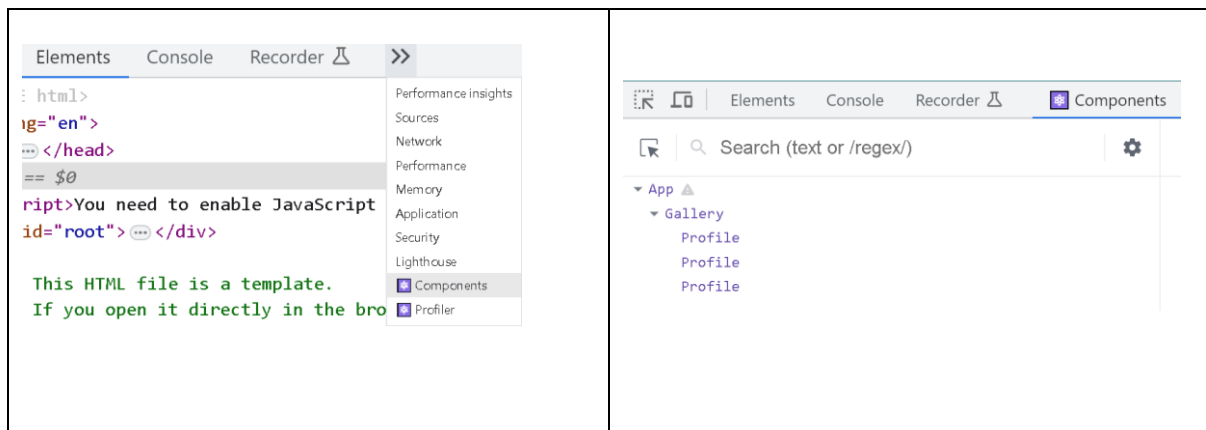
You can search for it in the Chrome Web store and install and enable it

https://support.google.com/chrome_webstore/answer/2664769?hl=en

As you can see, there are a few React related development extensions for Chrome from a search at the Web store:



After installing the React Developer Tools, you should be able to view and access it as one of the menu options from Chrome Developer Tools:



You can play around with the component nesting in the previous lab session to see how the component tree hierarchy renders in the Component view.

4.4 Using JSX in components

JSX is a syntax extension that resembles HTML but with more stricter rules and is capable of rendering dynamic content.

JSX rules:

- a) To return multiple elements from a component, wrap them with a single top level parent tag (e.g. `<div>`). You can use `<>` and `</>` instead as well - this is called a fragment (note that this construct does not exist in standard HTML)
- b) JSX requires tags to be explicitly closed (for e.g. ``, `
` and `oranges`)
- c) Attributes written in JSX become keys of JavaScript objects. JavaScript has limitations on key / variable names: e.g. cannot have dashes, cannot be reserved words. Therefore, attributes in JSX typically are written in camelCase.

Project: `first-app`

Using `<div>` or fragment as parent tag

Folder to use: `jsx`

File to use: `App-v1.js`

Attributes of HTML in JSX must follow camelCase convention

Folder to use: `jsx`

File to use:

`App-v2.js`

`App-v2.css`

If you change between the `<div>` or fragment, you will notice that a fragment does not translate into an actual node in the DOM tree (in the Elements panel of the Chrome Dev Tools), unlike a `<div>`. Therefore a `<div>` should really only be used if there is a some purpose for it (for e.g. to apply a `class` or `id` attribute for styling effects that will then be inherited or applied to all the child elements within the `<div>`).

4.5 Embedding variables / expressions in JSX

You can use the curly braces to embed JavaScript variables or expressions that return values into JSX.

You can only use curly braces in two ways inside JSX:

- As text directly inside a JSX tag
- As attributes immediately following the `=` sign

You can use double curlies `{ { }}` to include JavaScript objects into JSX

Using curly braces to interpolate variable values in JSX

Folder to use: `jsx`

File to use:

`App-v3.js`

If you are migrating normal HTML from an existing web page into a React app, you can use any one of the many HTML to JSX converters to ensure that the HTML is in correct JSX form.

<https://transform.tools/html-to-jsx>

<https://htmltojsx.in/>

5 Props

React components use props to communicate with each other. Every parent component can pass some information to its child components by giving them props: this is usually in the form of strings or numbers, but can also include objects, arrays and functions. Passing props to child components allow us to configure them in the same way that attributes are used to configure regular HTML elements.

Using props involves:

- a) Passing props to the child component (as attribute values enclosed within braces)
- b) Reading props in child component directly via a single `props` object parameter (which contains those attributes from the previous step a) as properties)
- c) Use these values in child component (typically by passing them as attribute values to a built-in component that is returned by the child component)

List of all props for built-in components

<https://react.dev/reference/react-dom/components/common#reference>

Project: `first-app`

Props as attributes of standard HTML elements

Folder to use: `props`

File to use: `App-v1.js`

Passing props to custom (user-defined) child components

Folder to use: `props`

File to use: `App-v2.js`

If you have already installed the React Developer Tools from a previous lab, you should be able to see the props being passed to the respective child components.

5.1 Accessing props via parameter destructuring

Parameter destructuring in the component function definition provides a short cut syntax for accessing the properties in the `props` object

Demonstrating destructuring: `Demo-Destructuring.js`

Accessing props in child component via parameter destructuring

Folder to use: `props`

File to use: `App-v3.js`

Using default parameter values in parameter destructuring

Folder to use: `props`

File to use: `App-v4.js`

5.2 Using children property to access content between JSX tags

When you nest content between JSX tags, the child component will receive that content in a special property called `children` (name is fixed) within the main `props` object (you can thus obtain it via destructuring or directly referencing the `props` object). This content be basic text or can be another child component.

Using the `children` special property to access text content between JSX tags

Folder to use: `props`

File to use:

`App-v5.js`

`App-v5.css`

Using the `children` special property to access child component between JSX tags

Folder to use: `props`

File to use:

`App-v6.js`

`App-v6.css`

You can thus choose to pass content either via attributes of the child component (which are then received as properties of the `props` object) or content nested between the tags of the child component (which is then received as a special property `children` of the `props` object).

5.3 Using spread operator syntax for JSX

It is possible to forward props with JSX spread operator syntax (`...`). This is useful when we provide a custom component that acts as a "wrapper" around some other component such as a built-in component. Using the spread syntax allows you to configure the built-in component using its standard attributes in a short form manner, while at same time allowing you to pass content nested between the tags of the custom component.

Using the JSX spread operator syntax (`...`) to configure the built-in component using its standard attributes in a short form manner

Folder to use: `props`

File to use: `App-v7.js`

5.4 Conditional rendering

Components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, `&&`, and `? :` operators.

Conditionally returning JSX in a variety of ways

Folder to use: `props`

File to use: `App-v8.js`

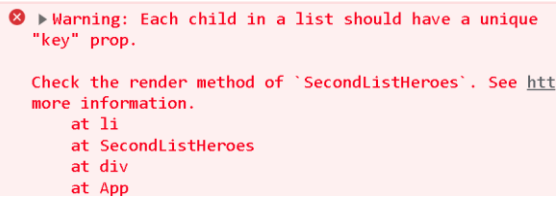
5.5 Rendering lists

You will often want to display multiple similar components from a collection of data, usually stored in an array. You can use the various array methods to manipulate an array of data such as `filter()` or `map()`

Folder to use: `props`

File to use: `App-v9.js`

Notice that the Console Tab in the Developer tools shows a warning regarding the various component functions



```

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

Check the render method of `SecondListHeroes`. See https://reactjs.org/docs/lists-and-keys.html for more information.
    at li
    at SecondListHeroes
    at div
    at App
  
```

This is because in React, you are strongly recommended to give each child item in a top-level list (either `` or ``) a `key` prop which has a unique value for each item. Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted. A well-chosen key helps React infer what exactly has happened and make the correct updates to the DOM tree.

Repeating previous example using `key` props

Folder to use: `props`

File to use: `App-v10.js`

Different sources of data provide different sources of keys:

- a) Data from a database: If your data is coming from a database, you can use the database keys/IDs, which are unique by nature.
- b) Locally generated data: If your data is generated and persisted locally (e.g. notes in a note-taking app), use an incrementing counter or a package like `uuid` when creating items.

Rules of keys

- a) Keys must be unique among sibling items. However, it's okay to use the same keys for JSX nodes in different lists.
- b) Keys must not change after they are initially defined

NOTE: You can use the index of an item in an array as its key AS LONG AS you are sure that the content of the array will never change. If the array content is going to change dynamically during the life time of the React app (for e.g. inserting, deleting or adding items to the array) - this will cause the index position of a specific item in the array to change which violates the 2nd rule above.

5.6 Components as pure functions

Components should behave as pure functions in order to avoid unintended side effects in React.

A pure function is a function with the following characteristics:

- It does not change any objects or variables that existed before it was called.
- Given the same inputs, a pure function should always return the same result

React assumes that every component you write is a pure function because its rendering process relies on this. This means that React components you write must always return the same JSX given the same inputs. The reason is that the components may be rendered in any particular order, so any particular reliance on a specific order of rendering may result in subtle bugs

An example of a violation of this rule is a component that changes a pre-existing variable that is defined outside the function component.

Folder to use: `props`

File to use: `App-v11.js`

The output from the Console Tab when the app above is rendered shows that each component in the App is called twice for each appearance in the JSX of the root component. This is the reason why Baking cake gives the unexpected result of #2, #4 and #6 (instead of #1, #2, and #3).

React offers a Strict Mode in which it calls each component's function twice during development. By calling the component functions twice, Strict Mode helps find components that break these rules. Strict Mode is defined by default in the `root.render` method call inside `index.js` of all React projects created using Create React App. It basically wraps the root component `<App>` inside `<React.StrictMode>`

Open `index.js` and remove the enclosing tags: `<React.StrictMode>`
`</React.StrictMode>`.

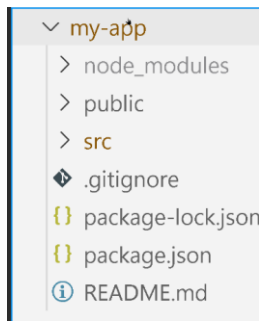
Save and Reload the app.

Notice now that each component is called exactly once and BakingCake gives the expected result of #1, #2, and #3.

You can remove StrictMode if you wish during development by performing the action above. However, keep in mind that StrictMode is in place to alert you to potential non-pure components during the development process. Just because you remove StrictMode to avoid getting weird results in your app rendering, it doesn't prevent subtle bugs from occurring because React does not render your components in the exact order that they appear in the JSX for optimization reasons.

6 Create React project folder structure

The top-level folder for a project generated using [Create React App \(CRA\)](#) should look something like this:



`node_modules` - Contains all the dependencies (JavaScript libraries and modules) related to a React project that is required for the React app to run correctly

`.gitignore` - Used for version tracking with the Git VCS. Specifies the files that Git should intentionally ignore.

`package.json` - Every Node.js project (and most JavaScript libraries such as React) has this file. It is automatically generated in the root folder of the project and contains metadata about the project (such as project name, description, version number and a list of dependencies required by the project). The metadata allows the NPM package manager to work with the project properly and is fully customizable.

`package-lock.json` - Extends on `package.json` by recording the exact version of every installed dependency, including its sub-dependencies and their versions. Its purpose is to ensure that the same dependencies are installed consistently across different environments, such as development and production environments. This helps to prevent issues with installing different package versions, which can lead to conflicts and errors.

`Public` - root folder that gets served as the React App.

`favicon.ico` - icon file that is used in `index.html` as a favicon. A favicon is a small pixel icon that serves as branding for your website. Its main purpose is to help visitors locate your page easier when they have multiple tabs open

`index.html` - serves as a template for generating the actual `index.html` that is served up to the browser. The DOM representing the content of this file is what you see in the Elements view of Console Developer in your browser. This file mainly contains metadata within the `<head>` element, and also provides the React library dependencies and your app code in a single JavaScript bundle generated by Webpack, which is the static module bundler used by [Create React App \(CRA\)](#).

You can view the contents of this bundle by right clicking on the web page of the React app and selecting View page Source, clicking on the `bundle.js` link.

```
<title>React App</title>
<script defer src="/static/js/bundle.js"></script>
```

The file also contains a single `<div id="root"></div>` which the React components from `bundle.js` will hook into to create the DOM tree that renders the actual browser view.

`manifest.json` - another pre-generated file which mostly contains metadata. Can be extended on to transform the basic React app into a hybrid Progressive Web App (PWA). A progressive web app (PWA) is a website that looks and behaves as if it is a mobile-native app.

`logo192.png`, `logo512.png` - These are the logo images of differing pixel size referenced in the template file (`index.html`) by default. You can replace this with your own and change the reference in `index.html`.

`robots.txt` - Defines rules for spiders, crawlers and scrapers for accessing your app.

`src` - folder containing the source code and other files related to the actual React app. The files in this folder is bundled by WebPack together with the React library and served up to the browser in `bundle.js`

`App.js` - This file contains the definition of the root component for the React app. It will typically contain definitions of other child components or import JavaScript files containing definition of other child components (as we have already seen in a previous lab).

`index.js` - The root entry file into the React app and renders the root of all React components (as defined in `App.js`). This imports the relevant React classes, and in particular the `ReactDOM.render` method, which renders the `<App>` component from `App.js` in the DOM element root (referenced in `index.html`).

`App.css` - style sheet containing style rules specifically to style `App.js`. You will typically import this into `App.js` when needed.

`index.css` - Global style sheet that contains style rules for the entire app (applying to both `App.js` and other components defined in separate files). Style rules defined in `App.css` (when imported into `App.js`) will override any style rules defined here.

`reportWebVitals.js` - checks the performance of the React app according to the five most common Web Vitals metrics (CLS, FID, FCP, LCP, TTFB). It is imported and called from `index.js`. See: <https://web.dev/vitals/>

`logo.svg` - Svg file of React logo, used in `App.js` in the default auto generated app.

`serviceWorker.js` - Service worker for pre-caching the scripts files of the React App to improve performance. Typically used when the React app is transformed into a PWA.

`App.test.js` - A very basic default test for `App.js` based on the Jest testing framework

`setupTests.js` - Setups tests and runs them, starting with the default `App.test.js`. This file is directly invoked when we run tests from CLI (`npm run test`) using the Jest testing framework.

7 Styling in React

There are several ways to apply styling to a React app

7.1 Using inline styles

Inline styles have exactly the same effect in a React app as they do in standard HTML/CSS. Styles applied directly to HTML elements have higher precedence and override other style rules that may be applied to an element

Inline styles are only an acceptable choice for very small applications. For larger and more complex applications, inline styles make it very difficult to understand application code logic.

Project: `first-app`

Folder to use: `styles`

File to use:

`App-v1.js`

7.2 Importing a separate style sheet

Importing a style sheet is done in exactly the same way in a React app as in standard HTML/CSS, except that the import statement is placed within a JavaScript file (either main `App.js` or the JavaScript file for other child components). The default style sheet for `App.js` is `App.css`

Folder to use: `styles`

File to use:

`App-v2.js`

`App-v2.css`

7.3 Using CSS-in-JS (Styled-Components)

CSS-in-JS allows us to create CSS style rules directly in our component's JavaScript files without needing a separate style sheet. This is conceptually similar to writing HTML as JavaScript via JSX.

CSS-in-JS ensures that styles are scoped to individual components. Changing one component's styles will not impact the styles of the rest of your application.

CSS-in-JS makes use of a special type of JavaScript function called a tagged template literal.

The two most popular CSS-in-JS libraries for React: Emotion and Styled Components.

For this demo, we will use the `styled-components` library, which we will first need to install by typing this into a CLI in the root of your project folder:

```
npm install styled-components
```

Folder to use: `styles`

File to use:

App-v3.js

The main project page for this library provides more examples if you wish to use this approach:
<https://styled-components.com/>

7.4 Accessing images in a React app

There are multiple ways to access image files from a React app.

- 1) You can directly use an URL to a remote image for the `src` attribute, which is suitable for public image sharing websites such as Unsplash.
- 2) Place the image file in `src` (or a subfolder within `src` such as `assets`) and import it as a string which is then referenced in the `src` attribute. This is the most common approach and ensures the files will be accessible to Webpack's asset bundling.
- 3) Place the image file in `public` subfolder and then access it as an absolute path in the `src` attribute. No imports are required here but the name of the file must be specified directly in the `src` attribute.
- 4) Use `require` keyword to access the image directly in the `src` attribute (similar to 3), but the image file must be located within `src` (or a subfolder within `src` such as `assets`). This keyword can also be used for including audio, video or document files in your project

Folder to use: `styles`

Copy `coffee.jpg` into the `public` folder of the current React project

Create an `images` subfolder in the `src` folder of the current React project. Copy the remaining 3 jpg files (`cabbage`, `cereal`, `muffins`) into this `images` subfolder

File to use:

App-v4.js

7.5 Other approaches to styling

Other alternatives to work with styling for a React App:

- Use a CSS preprocessor - such as SASS or LESS
- Use a CSS module - these are CSS files in which all class names and animation names are scoped locally by default

<https://blog.logrocket.com/styling-react-5-ways-style-react-apps>