React JS

Introduction to React

What is React?

React is a **JavaScript** library

Used for **front end web development**

Created and used by **Facebook**

Famous for implementing a **virtual DOM**

Common tasks in front-end development

App state	Data definition, organization, and storage
User actions	Event handlers respond to user actions
Templates	Design and render HTML templates
Routing	Resolve URLs
Data fetching	Interact with server(s) through APIs and AJAX

Fundamentals of React

1. JavaScript and HTML in the same file (JSX)
2. Embrace functional programming
3. Components everywhere [pure JS or JSX]

JSX: the React programming language

```
const first = "Aaron";
const last = "Smith";
const name = <span>{first} {last}</span>;

const list = (
<ul> <li>Dr. David Stotts</li>
   <li>{name}</li>
</ul>
);
```

```
const listWithTitle = (
<>
   <h1>COMP 523</h1>
   <ul>
     <li>Dr. David Stotts</li>
     <li>{name}</li>
   </ul>
   </>
);
```

Writing markup with JSX

```
function AboutPage() {
   return (
         <>
           <h1>About</h1>
            <p>Hello there.<br />How do you do?</p>
         </>
    );

}
```

JSX is a syntax extension for JavaScript, recommended for use with React to describe what the UI should look like.

Functional programming

1. Functions are “first class citizens”
2. Variables are immutable
3. Functions have no side effects

Functions are “first class citizens”

```
let add = function() { console.log('Now adding numbers’);
const five = 3 + 2;
};
```

```
function performTask(task) {
   task();
    console.log('Task performed!’);
}
performTask(add);
```

Variables are immutable

```
let a = 4;
a = 2; // Mutates `a`
```

```
let b = [1, 2, 3];
b.push(4); // Mutates `b`
let c = [...b, 4]; // Does not mutate `b`
```

Functions have no side effects

```
import React from 'react';
function multiplyByTwo(num) {
  return num * 2;
}
function NoSideEffectsComponent(props) {
  const result = multiplyByTwo(props.number);
  return (
    <div>
      <p>The number is: {props.number}</p>
      <p>Multiplied by two is: {result}</p>
    </div>
  );
}

export default NoSideEffectsComponent;
```

- **multiplyByTwo** is a pure function.
- takes a number as an input, multiplies it by two, and returns the result.
- The function does not modify any external variables, does not depend on any external state,
- and does not perform any I/O operations like network requests or console logs. The output of the function is entirely determined by its input.
- function are **predictable** and **easy to test** because **they always produce the same output** for the same input and have no side effects.

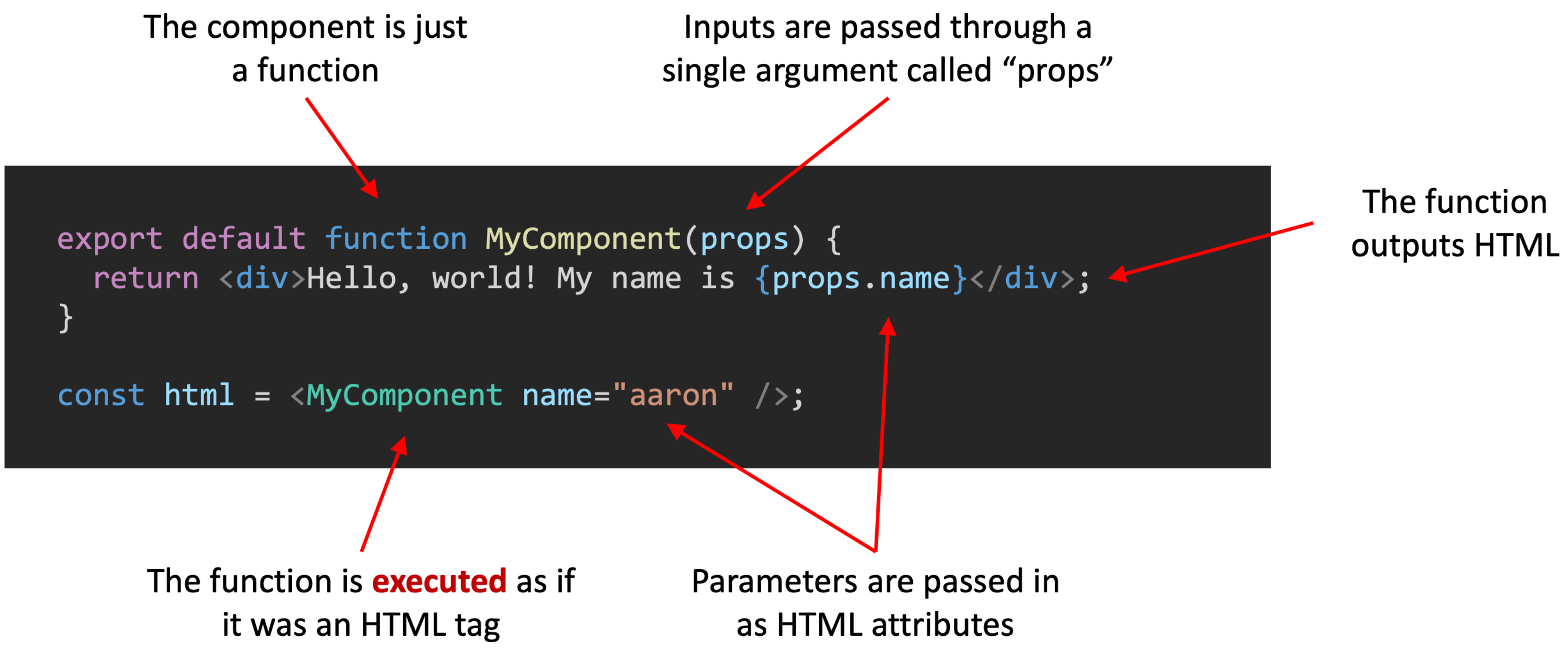
Components

Components are functions for user interfaces

```
let y = f(x);  //output number

let y = <FancyDiv value={x} />; //output HTML
```

Anatomy of a React **component**



The component is just a function

Inputs are passed through a single argument called “props”

```
export default function MyComponent(props) {
  return <div>Hello, world! My name is {props.name}</div>;
}

const html = <MyComponent name="aaron" />;
```

The function outputs HTML

The function is **executed** as if it was an HTML tag

Parameters are passed in as HTML attributes

Component rendering

- When a component function **executes**, we say it "**renders**"
- Assume components may re-render at any time

Our job is to ensure that
every time the component re-renders,
the correct output is produced

Creating a new React app

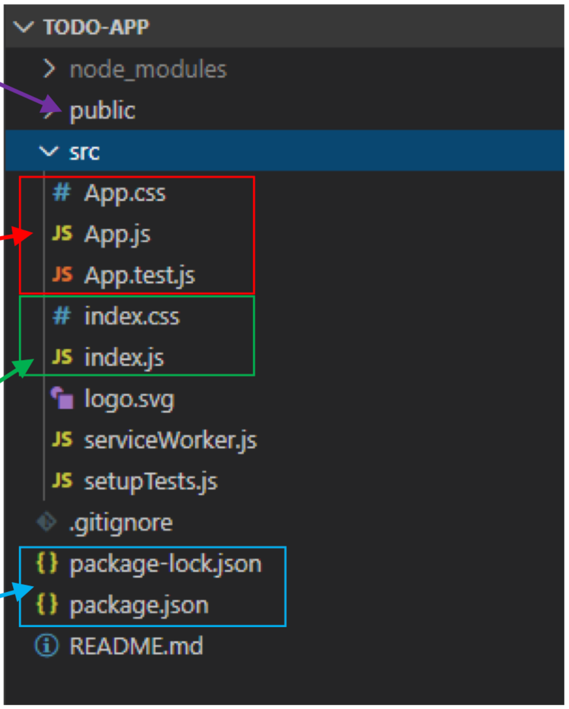
- Install **Node.js**
- Run: **`npx create-react-app`** `app-name`
- New app created in folder: `./app-name`

`public` holds the initial html document and other static assets

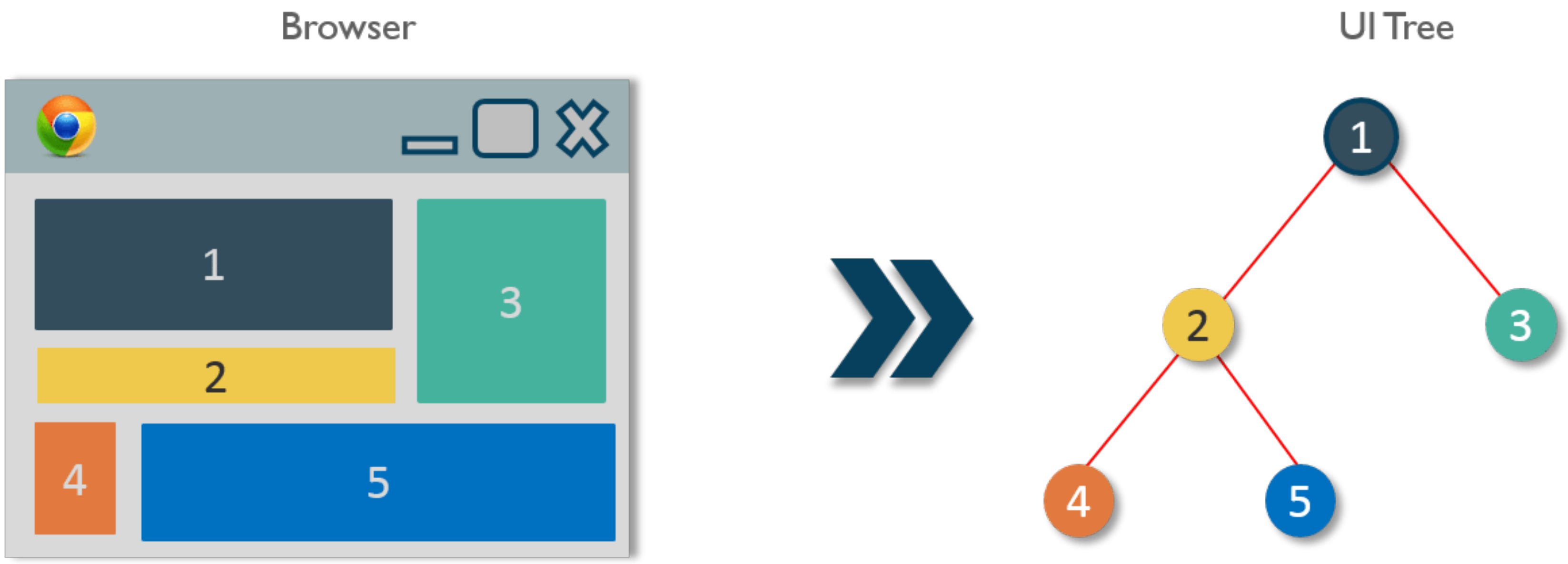
App is a boilerplate starter component

`index.js` binds React to the DOM

`package.json` configures npm dependencies


```
TODO-APP
  node_modules
  public
  src
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    serviceWorker.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

Everything works in components!



Your first functional component

A browser cannot understand JSX Syntax.
A **transpiler** takes a piece of code and transforms it into some other code.

```
function Heading() {
            return ( <h1>This is an h1 heading.</h1> )
 }
function App() {
            return (  <div className="App">  This is the starting code for "Your
                           first component" ungraded lab  <Heading />  </div>  );
}
export default App;
```

The React project structure

- **Node_modules** (for packages)
- **Public** (assists, manifest.json, index.html)
- **Src** (essential components)
- index.js (important part)
- **Root** folder (package.json)

Adding styles

```
<img className="avatar" />


/* In your CSS */
.avatar {
border-radius: 50%;
}
```

- Inline-style
- External CSS stylesheet

Displaying data

```
return (
       <h1> {user.name} </h1>
);
```

```
return ( <img className="avatar"
              src={user.imageUrl}

          />
);
```

Conditional rendering

```
let content;
if (isLoggedIn) {
	content = <AdminPanel />;
} else {
	content = <LoginForm />;
}

return (
	<div>   {content}      </div>

);
```

Responding to events

```
function MyButton() {
     function handleClick()    {
       alert('You clicked me!’);
     }

return (
       <button onClick={handleClick}>
            Click me
       </button>
);

}
```

Principles of components: Props

Recall this in JavaScript

- Parent component send to child component and not the other way around
- Pure functions -> you cannot modify props in react!
- Access them using .dot notation

Using props in components

Parent Component

```
import React from 'react';
import Greeting from './Greeting';

function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}
export default App;
```

Greeting Component

```
import React from 'react';

function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default Greeting;
```

Hooks

Hooks are functions that let you use state and other React features in functional components.

Common Hooks: useState, useEffect, useContext, useReducer, and more.

Advantages:

- Reuse stateful logic without changing component hierarchy.
- Split one component into smaller functions based on related parts.

Using the useState Hook

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
        <div>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>

                Click me
        </button>
        </div>
);
}
```

- **useState** returns the current state and a function to update it.
- The state persists across re-renders.
- Can use multiple **useState** hooks in a single component.

Using the useEffect Hook

```
function User({ userId }) {

const [user, setUser] = useState(null);

useEffect(() => {
fetch(`https://api.example.com/users/${userId}`)
        .then(response => response.json())
        .then(setUser);
    }, [userId]); //userId is dependency

return <div>{user ? user.name :
        "Loading..."}</div>; //return cleanup
}
```

- **Side effects**: data fetching, subscriptions, or manual DOM manipulations.
- **Cleanup function**: *return* a function from useEffect for cleanup.
- **Dependency array**: If values in the list change, the effect runs again.