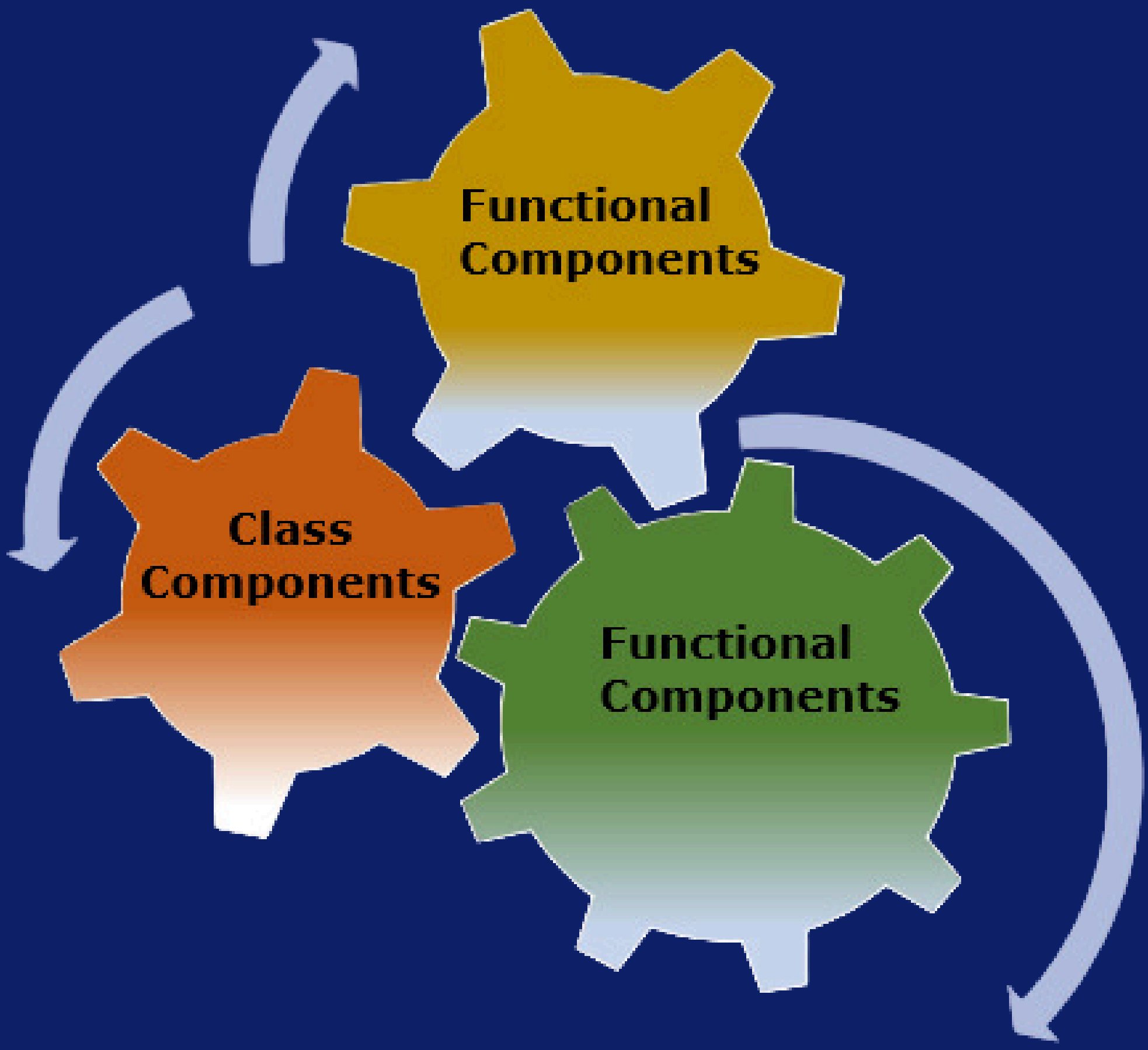
Day 3

React Components



About React.js Components

React components are the building blocks of any React application. Think of components like small, reusable pieces of a puzzle that fit together to create the full picture of your web app.

A **React component** is essentially a JavaScript function or class that returns a piece of UI (often HTML wrapped in JavaScript syntax, known as JSX).

The key idea is to **split** the UI into smaller, manageable parts, making it easier to develop, test, and maintain.

Components can be as simple as a button or as complex as an entire form or page layout.

Why Use Components?

1. Reusability

When you create a React component, you are building a self-contained, reusable piece of code. Think of it like a building block or a LEGO piece.

Once you've created a component (like a button or a form), you can use that same component multiple times across different parts of your application without having to write the code again.

Imagine you have a button that says "Submit" on it. If you build it as a reusable component, you don't need to write the button code every time you need it; you just reuse the component. This saves time and effort and makes your app more consistent.

```jsx
import React from 'react';

// Create a reusable button component
function Button({ label }) {
  return <button>{label}</button>;
}

// Reuse the Button component with different labels
function App() {
  return (
    <div>
      <Button label="Submit" />
      <Button label="Cancel" />
      <Button label="Save" />
    </div>
  );
}

export default App;
```

Here, the **Button component** is defined once and used three times with different labels. This reduces duplication and makes your code cleaner.

2. Maintainability

Maintainability means making your code easier to understand, fix, and update over time. By breaking the UI into smaller components, you isolate each part of your application into distinct sections.

If there's an issue or a change needed, you know exactly where to look and what part to update, without affecting the rest of your application.

Suppose you have a bug in a large, single file of code. It can be overwhelming to find and fix the issue. But if your UI is divided into smaller components, each handling a specific task, it’s easier to locate the problem and fix it without worrying about breaking other parts of your app.

```jsx
import React from 'react';

// Separate components for each part of the UI
function Header() {
  return <header><h1>My App</h1></header>;
}

function Content() {
  return <main><p>This is the main content area.</p></main>;
}

function Footer() {
  return <footer><p>Footer text here</p></footer>;
}

// Use the components to build the UI
function App() {
  return (
    <div>
      <Header />
      <Content />
      <Footer />
    </div>
  );
}

export default App;
```

In this example, we have separated the **header**, **content**, and **footer** into their own components. If we need to change the footer text, we only need to modify the Footer component without affecting the other parts.

3. Separation of Concerns

Separation of concerns means **dividing a program into different sections**, where each section handles a specific responsibility or "concern." In React, this is achieved by creating individual components for each part of your user interface.

By separating concerns, each component focuses on a single task or part of the UI. This makes your code easier to read and understand because each component has a clear purpose.

It also makes it easier to test and debug, as any issue is likely isolated within a specific component.

For example:

```jsx
import React, { useState } from 'react';

// Input component handles input field and user input logic
function InputField({ onChange }) {
  return <input type="text" onChange={onChange}
placeholder="Type something..." />;
}

// Display component handles displaying the entered text
function Display({ text }) {
  return <p>You typed: {text}</p>;
}

// App component brings everything together
function App() {
  const [inputText, setInputText] = useState('');

  const handleInputChange = (event) => {
    setInputText(event.target.value);
  };

  return (
    <div>
      <InputField onChange={handleInputChange} />
      <Display text={inputText} />
    </div>
  );
}

export default App;
```

- **InputField** is responsible only for handling the input field.
- **Display** is responsible for showing the text entered by the user.
- **App** is the main component that coordinates these smaller components.

By keeping each component focused on a single concern, you make your code more organized and easier to work with.

4. Performance Optimization

React components help optimize the performance of your application by allowing you to update only the parts of the UI that need changes.

React uses a **virtual DOM** (a lightweight copy of the actual DOM) to efficiently determine what changes are needed and only updates the real DOM when necessary.

Directly manipulating the **real DOM** can be slow because it takes a lot of resources to re-render the entire page.

React's virtual DOM optimizes this process, ensuring your application is fast and responsive.

For Example;

```jsx
import React, { useState } from 'react';

function Counter() {
  // State to track the counter value
  const [count, setCount] = useState(0);

  // Function to increment the counter
  const increment = () => {
    setCount(count + 1); // Updates only this part of the UI
  };

  return (
    <div>
      <h2>Counter: {count}</h2>
      <button onClick={increment}>Increase</button>
    </div>
  );
}

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

export default App;
```

When the button is clicked, only the **Counter** component re-renders to reflect the updated value. This avoids unnecessary updates to other parts of the app, enhancing performance.

5. Consistent UI Design

Using React components helps maintain a consistent user interface across your application. Components encapsulate both logic and styling, so you can ensure the same design and behavior wherever they are used.

Consistency in design makes your application more user-friendly and professional. Reusing components also ensures that any updates or changes to a component (like a button style) are automatically reflected throughout the application, maintaining a uniform look.

React’s virtual DOM optimizes this process, ensuring your application is fast and responsive.

For Example:

```jsx
import React from 'react';
import './Button.css'; // Shared CSS file for styling

// Button component with consistent styling
function Button({ label, onClick }) {
  return <button className="custom-button" onClick={onClick}>{label}</button>;
}

function App() {
  return (
    <div>
      <Button label="Submit" onClick={() => alert('Submitted!')} />
      <Button label="Cancel" onClick={() => alert('Cancelled!')} />
      <Button label="Save" onClick={() => alert('Saved!')} />
    </div>
  );
}

export default App;
```

The **Button** component has a consistent style and behavior defined in a shared CSS file. Using it throughout the app ensures that all buttons look and function the same way.

Types Of React Components

There are two main types of components in React:

1. Functional Components

2. Class Components

Functional Components

A **functional component** in React is simply a JavaScript function that returns React elements (usually written in JSX).

Functional components are easier to write, understand, and test. They are also more efficient because they don't need the overhead of managing a class or dealing with the complexities of **this**.

A basic example:

```jsx
// Example of a Functional Component
import React from 'react';

function Greeting() {
  return <h1>Hello, World!</h1>;
}

export default Greeting;
```

- **Greeting** is a functional component that returns a simple h1 element.

- The **return statement** outputs the JSX that gets rendered to the browser.

Why Use Functional Components?

- **Simpler Syntax:** Functional components are simpler and more concise. You don't need to extend a base class or manage state using this.
- Performance Benefits: Functional components are more lightweight, leading to faster rendering. React can optimize functional components better, especially when using hooks.
- Encourages Best Practices: Functional components encourage writing cleaner, more modular code.
- Hooks Support: React introduced hooks in version 16.8, allowing functional components to use state and other React features.

Class Components

- Class components are ES6 classes that extend from React.Component. They were the traditional way to create components before the advent of React Hooks.
- They have a more complex syntax compared to functional components and manage state with this.state. Example of a class component:

```
1 import React, { Component } from 'react';
2
3 class Welcome extends Component {
4   render() {
5     return <h1>Welcome to React, Class Components!</h1>;
6   }
7 }
8
9 export default Welcome;
```

- **Component** class is imported from react which is extended to create our own components.
- Just like in **Object Oriented Programming** (OOP), **Welcome** is a class component that extends the React.Component.

- The **class keyword** defines a new JavaScript class named Welcome.
- It has a render method that returns the JSX to display.
- The use of **extends** allows the **Welcome class** to access all the properties and methods of the Component class.
- The **render()** method returns an **<h1>** element that displays the text "Welcome to React, Class Components!".

Comparing Class and Funtional Components

Feature	Class	Functional
• Syntax	Verbose and more complex.	Very Fast
• State Management	Uses this.state and setState.	Uses hooks, more simpler
• Performance	Larger weight, Slower	More lightweight, Fast
• Context API	Consume context using the Consumer component or contextType.	Consume context using the useContext hook.
• Code Reusability	Limited in reusing stateful logic.	Highly reusable with hooks
• Learning Curve	Steeper learning curve	Easier to learn and understand
• Lifecycle Methods	Has built-in lifecycle methods, componentDidMo-unt() for example.	Uses useEffect hook for managing lifecycle events.

How to Create a Component in React

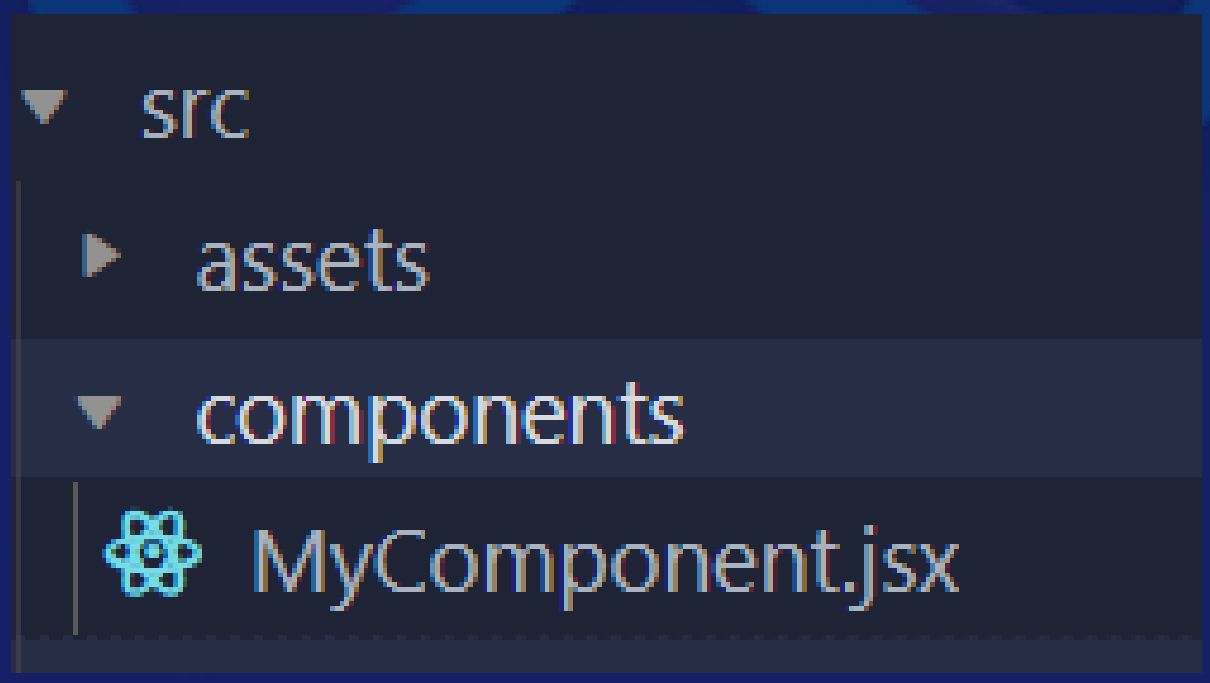
We learned about React project structures yesterday, for CRA and Vite. In one of the common files (src) is where we will create our component.

Here’s a step-by-step guide to creating a simple React component:

- **Create a New File:** In your src folder, create a folder (usually named “components”, and in it, create a new file, for example, MyComponent.jsx.

  A component name in React **must** begin with an **upper case letter**, and it is best practice to write using Pascal casing (each word in the name starts with upper case. For example, PatientForm.

For CRA, the extension will be “.js” but for Vite, the extension will be “.jsx”



- **Import React:** Import React at the top of the file. This is not necessary if you created your project with Vite method, it’s already imported as a plugin in your vit.config.js file.

MyComponent.js U

src > components > MyComponent.js

```
1   import React from "react";
```

- **Define the Component:** Use a function or a class to define your component.

```
1   const MyComponent = () => {
2     return (
3       <div>MyComponent</div>
4     )
5   }
6
7   export default MyComponent
```

A functional component structure with arrow function

src > components > MyComponent.js > default

```
1   import React, { Component } from "react";


2   class Welcome extends Component {
      Codiumate: Options | Test this method
3     render() {
4       return <h1>Welcome to React, Class Components!</h1>;
5     }
6   }
7
8   export default Welcome;
```

A class component structure

- **Return JSX:** The component should return some JSX to render, i.e some HTML elements. As above in the class component, the component returns an “**h1**” element which will be displayed on the UI when this component is called.

- **Export the Component:** Use export default to make your component available to other parts of your app.

```
1  const MyComponent = () => {
2      return (
3          <div>MyComponent</div>
4      )
5  }
6
7  export default MyComponent
```

This will make sure that other parts of the project have access to the “**MyComponent**” component.

React Component Lifecycle

The React Component Lifecycle refers to the series of events that happen from the moment a component is **created (mounted)** to when it is **removed (unmounted)** from the DOM.

React components go through several phases during their life:

1. Mounting: When a component is being created and inserted into the DOM. Common methods for mounting include:

- constructor(): Initializes state and binds event handlers.
- componentDidMount(): Runs after the component is added to the DOM, perfect for API calls or setting up subscriptions.

2. Updating: When a component is being re-rendered due to changes in props or state. The method used for updating a component is:

- componentDidUpdate(prevProps, prevState): Runs after an update, ideal for comparing props and triggering actions on changes.

3. Unmounting: When a component is removed from the DOM, and the method used is:

- componentWillUnmount(): Used to clean up resources like event listeners or timers to prevent memory leaks.

4. Error Handling: When there is an error during rendering, in a lifecycle method, or in a child component. The method used for this is:

- componentDidCatch(error, info): Catches errors and allows you to display a fallback UI.

In summary, Mounting is for setup, Updating is for responding to changes, Unmounting is for cleanup.

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

kodemaven-portfolio.vercel.app

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

KodeMaven

kodemaven-portfolio.vercel.app