React
(Advance Topics)

Module 6

ADC502 : Web Development

Functional components- Refs, Use effects, Hooks

Hook

- In React, a hook is a function that allows you to "hook into" or use state and other React features in functional components.
- Before the introduction of hooks, state management and lifecycle features were primarily used in class components. However, with the release of React 16.8, hooks were introduced, enabling functional components to have access to the same capabilities.
- Hooks provide a way to reuse stateful logic without changing the component hierarchy. They allow functional components to manage component state, perform side effects, and access React features such as context, refs, and more.

Rules on Hook

- Don’t call Hooks inside loops, conditions, or nested functions.
- Always use Hooks at the top level of your React function, before any early returns.
- Don’t call Hooks from regular JavaScript functions. Instead, call Hooks from React function components or from custom Hooks.

Note: Hooks will not work in React class components.

List of Hooks in React

- useState
- useEffect
- useContext
- useRef
- useReducer
- useCallback
- useMemo
- useImperativeHandle
- useLayoutEffect
- useDebugValue

Hooks mentioned at top of the list are more commonly used.

We can create custom hooks.

useState

- useState is a React hook that allows functional components to manage state.
- useState hook
  - takes an **initial state value as an argument**
  - returns **an array** containing the **current state value** and a **function to update that state**.

useState : Example

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

function Counter() {
  // Using useState to manage the count state
  const [count, setCount] = useState(0);

  // Event handler to increment the count
  const incrementCount = () => {
    setCount(count + 1);
  };

  // Event handler to reset the count
  const resetCount = () => {
    setCount(0);
  };

  return (
    <div>
      <h1>Counter</h1>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
      <button onClick={resetCount}>Reset</button>
    </div>
  );
}

export default Counter;
```

useState – Example Exaplained

In this example, we've created a Counter functional component.

Here's a breakdown of how useState is used:

- We import useState from the React library.
- Inside the Counter component, we call useState(0) to initialize a state variable named count with an initial value of 0. The useState hook returns an array with two elements:
  - The first element (count) is the current state value.
  - The second element (setCount) is a function to update the state.
- We define two event handlers, incrementCount and resetCount, which use setCount to update the count state.
- In the JSX part of the component, we display the current value of count using curly braces {count}. When the "Increment" or "Reset" buttons are clicked, it triggers the corresponding event handler.
- When you click the "Increment" button, the count state is updated, and React automatically re-renders the component to reflect the new state. This is one of the fundamental principles of React: components re-render when their state or props change.

useEffect

- In React, an **effect** refers to any side effect that a component can perform, such as data fetching, DOM manipulation, or subscription handling.
- The **useEffect** hook is used to manage these side effects in functional components.
- It allows you to perform tasks that need to occur after the component has rendered, as well as when the component updates or unmounts.

- Here's a brief example of useEffect in a React functional component:

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

function ExampleComponent() {
  const [count, setCount] = useState(0);

  // This effect runs after every render and updates the document title
  useEffect(() => {
    document.title = Count: ${count};
  });

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}

export default ExampleComponent;
```

useEffect – Example Explained

- Inside the ExampleComponent functional component, we define a state variable count using useState to manage a count value.
- We use the useEffect hook to perform a side effect. In this case, we're updating the document title. The effect is defined as a function that will run after every render of the component. The function sets the document title to include the current count value. Since this effect is run after every render, it keeps the document title in sync with the count state.
- We define an event handler incrementCount that increases the count state when the "Increment" button is clicked.
- In the JSX part of the component, we display the count value and a button to increment it. When the "Increment" button is clicked, it triggers the incrementCount function, which updates the count state.
- The useEffect hook allows you to manage side effects in a declarative way within functional components. In this example, it updates the document title, but you can use it for a wide range of side effects, including data fetching, setting up subscriptions, and more. The effect runs after rendering and can also be configured to run conditionally based on certain dependencies, providing fine-grained control over when it executes.

Refs

- React refs are a mechanism that allows developers to access and interact with DOM elements or React components directly.
- Unlike the typical data flow in React, which propagates changes from parent components to their children, refs provide a way to bypass this flow and reach into a component or a DOM element.
- This direct access is often necessary for a variety of reasons, including imperative DOM manipulation, integration with third-party libraries, managing focus, and accessing component instances.
- Refs are a powerful feature that should be used sparingly and with caution because they bypass the typical React data flow and can make the code less predictable.

Refs - Scenarios where refs are useful

1. Accessing the DOM: To interact with a specific DOM element, like focusing an input, triggering animations, or reading the value of an input field. Using refs, one can gain direct access to the underlying DOM element.

```
class MyComponent extends React.Component {
  myInputRef = React.createRef();
  componentDidMount() {
    // Focus the input field
    this.myInputRef.current.focus();
  }
  render() {
    return <input ref={this.myInputRef} />;
  }
}
```

Refs - Scenarios where refs are useful

2. Integrating with Third-Party Libraries: Some third-party libraries might require direct access to DOM elements. Refs help to integrate these libraries seamlessly into React application.

```
class MyComponent extends React.Component {
 myChartRef = React.createRef();
 componentDidMount() {
   // Initialize a chart using a third-party library
   initializeChart(this.myChartRef.current);
 }
 render() {
   return <div ref={this.myChartRef} />;
 }
}
```

Refs - Scenarios where refs are useful

3. Managing Focus and Text Selection: Refs can be used to control focus within components or to select text. Example to select the text within an input field when it receives focus:

```
class MyComponent extends React.Component {
 myInputRef = React.createRef();
 handleFocus = () => {
   this.myInputRef.current.select();
 }
 render() {
   return (
        <input
          ref={this.myInputRef}
          onFocus={this.handleFocus}
          defaultValue="Click to select text"
        />
   );
 }
}
```

Refs - Scenarios where refs are useful

4. Interacting with Child Components: Refs can be used to communicate with child components, allowing to trigger methods or access their internal state.

```
class ParentComponent extends React.Component {
 childRef = React.createRef();

 handleClick = () => {
  // Access and call a method in the child component
  this.childRef.current.doSomething();
 }

 render() {
  return (
        <div>
         <ChildComponent ref={this.childRef} />
         <button onClick={this.handleClick}>Call Child Method</button>
        </div>
  );
 }
}
```

```
class ChildComponent extends React.Component {
 doSomething() {
  // Perform some action
 }

 render() {
  return <div>Child Component</div>;
 }
}
```

Point to ponder!

- Note that in the context of React or component-based frameworks, the concept of "parent components containing child components" is different from inheritance in object-oriented programming (OOP), such as in languages like Java.
- These are two distinct approaches to structuring and organizing code, and they serve different purposes.

Refs and Hook !

- Did we use React hook in any of the previous 4 examples for refs.
- No! All the 4 examples were class components.
  - Hooks can only be used in functional components.
- To use refs in functional components we must use useRef hook.

Refs in functional component

```
import React, { useRef } from 'react';

function RefExample() {
  // Create a ref using the useRef hook
  const inputRef = useRef(null);

  const focusInput = () => {
    // Access and focus the input element using the ref
    inputRef.current.focus();
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}

export default RefExample;
```

Refs in functional component

In this example:

- We **import React** and the **useRef hook** from the React library.
- We create a **functional component** named **RefExample**.
- Inside the component, we use the **useRef hook** to create a **ref named inputRef** and initialize it with **null**. The **inputRef ref will be used** to reference **the input element** in the JSX.
- We define a **function focusInput** that, when called, uses the inputRef to focus on the input element. This function is triggered when the "Focus Input" button is clicked.
- In the JSX part of the component, we render an input element with the ref attribute set to inputRef. This associates the ref with the input element. We also render a button with an onClick handler that calls the focusInput function.

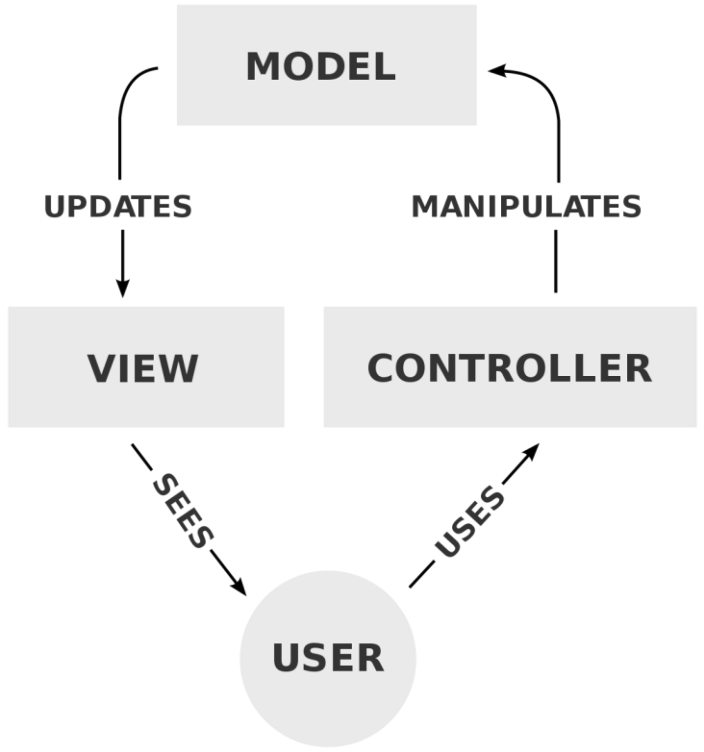
Flow architecture

Is covered under topics MVC and Flux (Unidirectional Flow of data in React)

Model-View-Controller framework

Model-View-Controller (MVC)

- Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements.
- These elements are the internal representations of information (the Model), the interface (the View) that presents information to and accepts it from the user, and the Controller software linking the two.



Model-View-Controller (MVC)

- Model:
  - The Model represents the application's data and business logic. It is responsible for data storage, retrieval, and manipulation.
  - The Model component encapsulates the application's data, database interactions, and all the business rules.
- View:
  - The View is responsible for presenting the user interface to the user. It displays data from the Model to the user and presents the user's interactions to the Controller.
  - In web applications, the View often comprises HTML templates, CSS for styling, and user interfaces.
- Controller:
  - The Controller receives and manages user input and interactions. It processes user requests, communicates with the Model to retrieve data, and instructs the View to update the user interface.
  - The Controller essentially acts as an intermediary between the Model and the View.

Advantages of Using MVC

1. Separation of Concerns
2. Modularity
3. Code Reusability
4. Flexibility
5. Scalability
6. Maintainability
7. Parallel Development
8. User Interface and Logic Separation
9. Testing is easier
10. Compatibility with various programming languages and platforms

React and MVC

- In the context of React, the traditional Model-View-Controller (MVC) architectural pattern is often replaced or extended by the Model-View-Controller (MVC) architectural pattern with components and state management.
- React's component-based architecture aligns more closely with the Component-View-Controller (CVC) pattern, where components take on the role of both the view and the controller, and state management libraries (such as Redux or React's built-in state management) handle the model.

How the MVC maps to a React application

- Model (Data and Logic):
  - In a traditional MVC pattern, the model represents the application's data and business logic. In React, this role is often split between component state and external state management libraries like Redux.
  - React components can manage their internal state using ***this.state***. It's suitable for local component-specific data.
  - Redux/Context/State Management Libraries in react handle global or shared state and act as the application's data model. They store and manage data that needs to be accessed and modified across multiple components.
- View (UI):
  - In both traditional MVC and React, the view represents the user interface. In React, the view is primarily defined through the component's ***render*** method, which describes how the UI should look based on the component's state and props.

How the MVC maps to a React application

- Controller (User Input and Business Logic):
  - In traditional MVC, the controller receives user input and updates the model and view accordingly. In React, this role is largely taken over by React components.
  - Components in React encapsulate both the view and controller responsibilities. They render UI elements and handle user interactions. When user interactions trigger changes, components update the model (state or state management library) and re-render the view.
  - React components define event handlers (e.g., onClick) to respond to user input and initiate state updates.
  - React components have lifecycle methods like componentDidMount and componentDidUpdate, which can be used for executing business logic and interacting with external data sources.

React and MVC - Summary

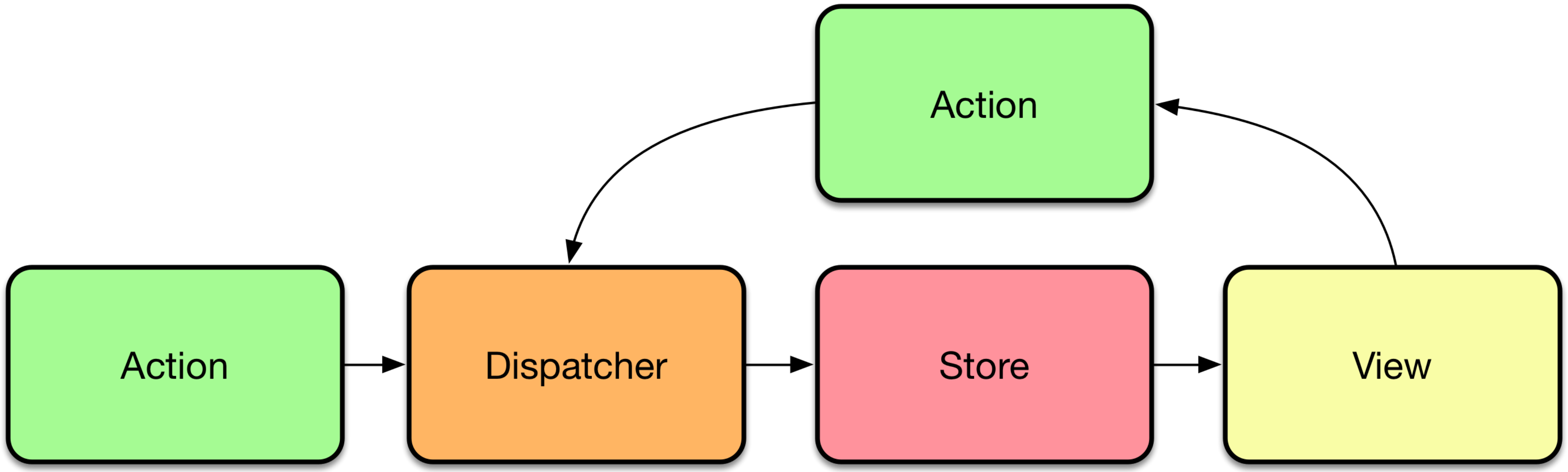
- React's component-based architecture simplifies the separation of concerns. Each component can have its own local state and handle its user interactions. External state management libraries (such as Redux) help manage shared state and enable global state updates across components.
- React doesn't strictly adhere to the traditional MVC pattern. Instead, it embraces a Component-View-Controller (CVC) approach where components serve as both the view and controller, and state management libraries handle the model or global state. This architecture makes React highly flexible and suitable for building user interfaces with complex interactions and data flows.

Flux

Flux

- Flux is an architectural pattern used in React and other JavaScript libraries to manage the flow of data in a unidirectional manner.
- To support React's concept of unidirectional data flow, the Flux architecture was developed as an alternative to the popular model–view–controller architecture.
- Flux features actions which are sent through a central dispatcher to a store, and changes to the store are propagated back to the view.
- Flux provides a clear and organized structure for managing data and interactions within a React application.

Flux Dataflow



Flux

- Actions:
  - Actions are simple JavaScript objects that describe an event or user interaction. They contain a type and any necessary data.
  - Actions are created and dispatched to indicate that something has occurred within the application, such as a button click or an API request completion.
- Dispatcher:
  - The Dispatcher is responsible for receiving and dispatching actions to registered stores. It acts as a central hub for managing the flow of data within the application.
  - When an action is dispatched, it is sent to all registered stores, ensuring that each store can decide whether to respond to the action.

Flux

- Stores:
  - Stores are responsible for managing the application's data and state. Each store contains the application's data and business logic.
  - Stores listen for actions from the Dispatcher and update their data based on the action's type and payload.
  - Stores emit change events to notify the views when the data they manage has been updated.
- Views:
  - Views represent the user interface components in a React application. They are responsible for rendering the UI and displaying data from the stores.
  - Views subscribe to stores and update themselves when the stores emit change events.
  - Views can also dispatch actions when user interactions occur, completing the unidirectional data flow cycle.

Flux

- The Flux architecture in React is particularly useful for managing the state of large and complex applications. It helps maintain a predictable and organized structure for handling data and interactions, reducing the potential for data inconsistencies and bugs.
- While Flux provides a clear pattern, it doesn't prescribe a specific implementation, and there are variations and libraries built on the Flux concept, such as Redux and Mobx, that offer additional features and enhancements.

Bundling the application - webpack.

Bundling the application

- The React app bundled their files using tools like Webpack.
- Bundling is a process which takes multiple files and merges them into a single file, which is called a bundle.
- The bundle is responsible for loading an entire app at once on the webpage.
- We can understand it from the below example.

```
// App.js
import { add } from './math.js';
console.log(add(16, 26)); // 42

// math.js
export function add(a, b) {
  return a + b;
}
```

Bundling the application

- Bundled file will be like below:

```
function add(a, b) {
  return a + b;
}
console.log(add(16, 26)); // 42
```

Bundling the application

- As the app grows, the bundle will grow too, especially when using large third-party libraries. If the bundle size gets large, it takes a long time to load on a webpage. For avoiding the large bundling, it's good to start splitting the bundle.
- React 16.6.0, released in October 2018, and introduced a way of performing code splitting. Code-Splitting is a feature supported by Webpack, which can create multiple bundles that can be dynamically loaded at runtime.
- Code splitting uses React.lazy and Suspense tool/library, which helps to load a dependency lazily and only load it when needed by the user.

What is Webpack?

- Webpack is a tool that takes all the different pieces of your React application (JavaScript files, CSS, images, etc.) and bundles them together into a single, optimized package.
- This makes the application load faster and helps the developer organize their code.

Why Use Webpack in React?

- Bundling:
  - When you create a React application, you typically write many separate JavaScript files.
  - Webpack bundles these files together, so they can be easily included in your HTML.
- Transpilation:
  - React code is often written using modern JavaScript features, like JSX.
  - Webpack can transpile your code to an older version of JavaScript that's compatible with most browsers.
- Optimization:
  - Webpack can optimize your code for production by minimizing its size and making it load quickly.
- Asset Management:
  - Webpack can handle other assets like CSS, images, and fonts, allowing you to import them directly into your JavaScript code.

Setting Up Webpack

- To use Webpack in your React project, you need to install it and set up a configuration file.

- Install Webpack and related packages:

```
npm install webpack webpack-cli --save-dev
```

- Create a webpack.config.js file in your project root to configure Webpack.

Configuration

- In the webpack.config.js file, you can specify how Webpack should handle different types of files and define your entry and output points. For example:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

- In this configuration, you're telling Webpack to use src/index.js as the entry point and output the bundled code as dist/bundle.js.

Using Webpack in React Application

- Now, Webpack can be used to manage assets and import them into code:

```
import React from 'react';
import logo from './logo.png'; // Importing an image
const App = () => {
  return (
    <div>
      <img src={logo} alt="Logo" />
      <h1>Hello, React with Webpack!</h1>
    </div>
  );
};
export default App;
```

- In this example, we're importing an image using Webpack, and it will be included in the bundle.

Running Webpack

- To bundle the React application, run Webpack using the command:

```
npx webpack --mode development
```

- This command bundles your code in development mode. You can also run it in production mode to create optimized bundles:

```
npx webpack --mode production
```

End