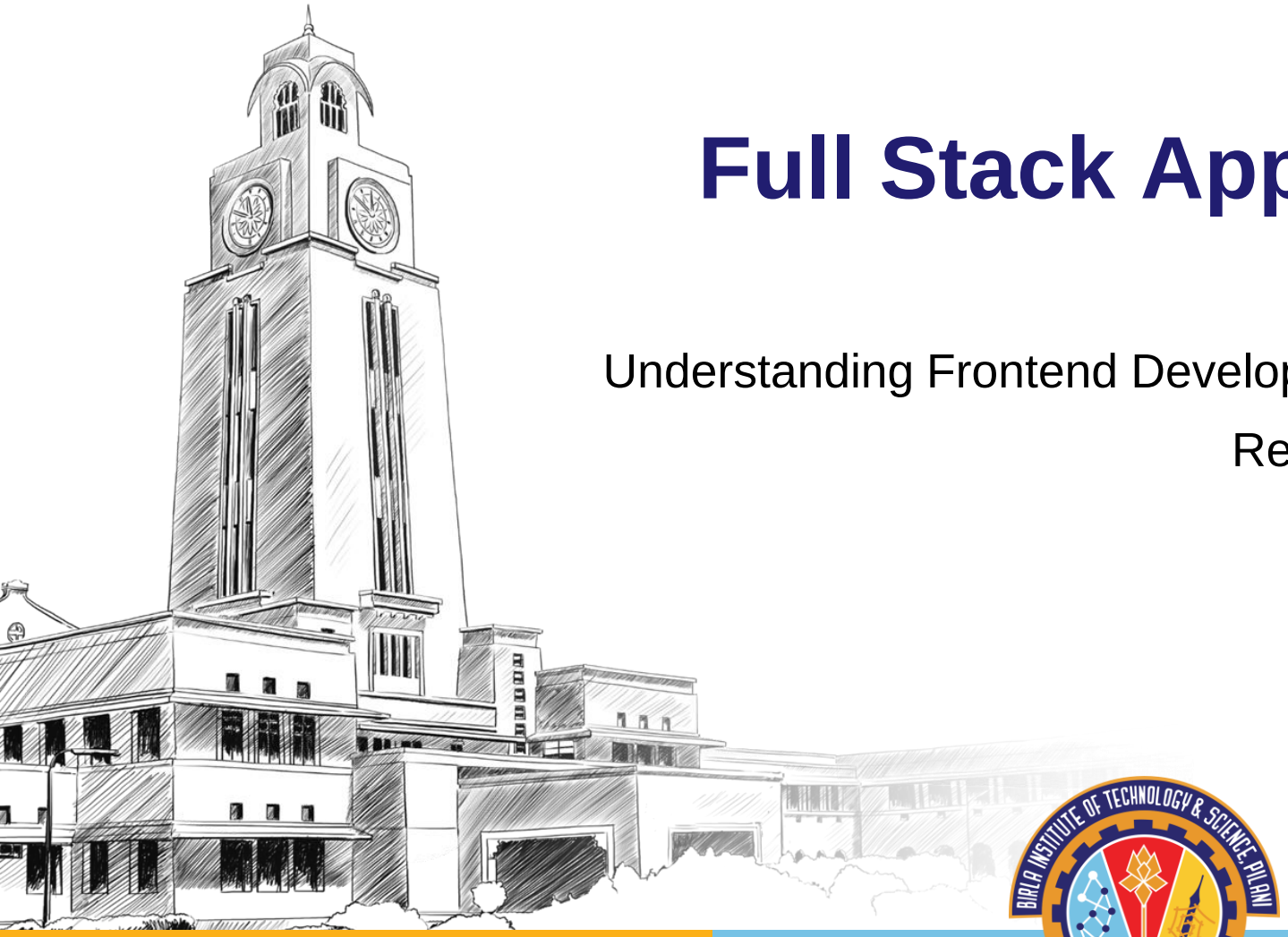


Full Stack Application Development

Understanding Frontend Development

ReactJS

Akshaya Ganesan

State

State

- Components need to “remember” things
- React Class components have a built-in state object.
- You can add state to a component with a useState Hook.
- The state object is where you store property values that belongs to the component.
- When the state object changes, the component re-renders.

State in Functional components

- Earlier, Functional Components were stateless.
- React Hooks made it possible to have state in Function Components.
- Hooks are a new addition in React 16.8.

Example

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

function Example() {
  // Declare a new state variable, "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Points to be Noted

- Do Not Modify State Directly - Use setState()
  - For example, this will not re-render a component:
  - this.state.comment = 'Hello';
  - If two state variables always update together, consider merging them into one.

- React may batch multiple setState() calls into a single update for performance.
- A component may choose to pass its state down as props to its child components

Hooks

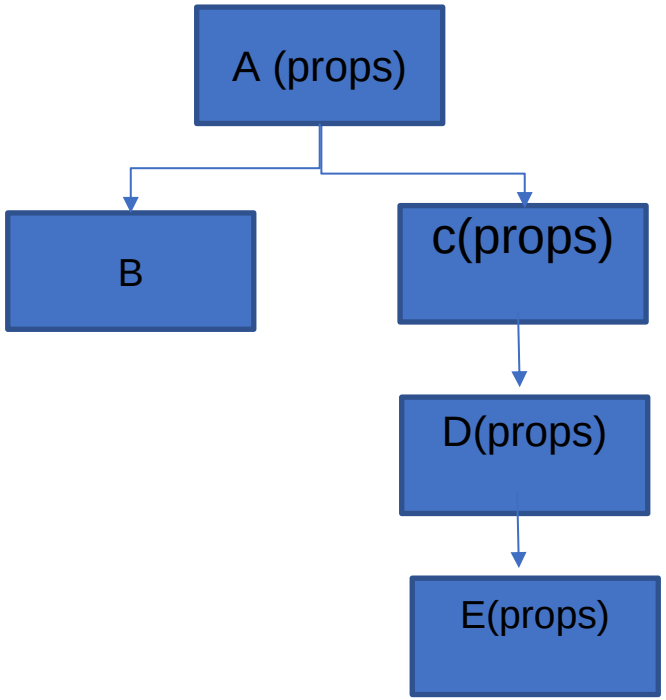
- React provides a few built-in Hooks like useState.

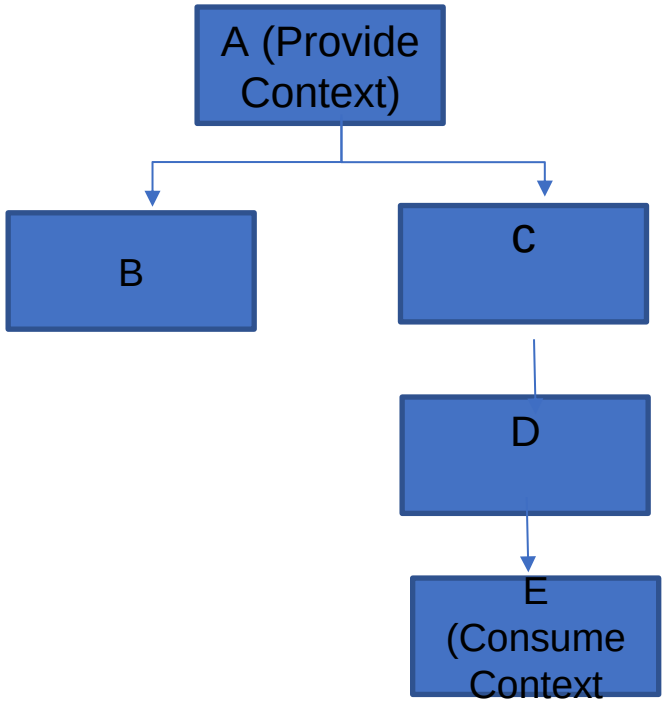
- You can also create your own Hooks to reuse stateful behavior between different components.

- Rules of Hooks
- Hooks are JavaScript functions, but they impose two additional rules
- Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components. Don’t call Hooks from regular JavaScript functions.

Context

- In a typical React application, data is passed top-down (parent to child) via props. When you had to pass props several components down your component tree. It results in prop drilling.





React Context

- There are two use cases when to use it:
- When your React component hierarchy grows vertically in size and you want to be able to pass props to child components without bothering components in between.
- When you want to have advanced state management in React. Doing it via React Context allows you to create a shared and global state.

Conditional Rendering

- **Conditional rendering** as a term describes the ability to render different UI markup based on certain conditions.
- Conditional rendering in React works the same way conditions work in JavaScript.

- Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

- If/else
- element variables
- Ternary operator
- Short Circuit Evaluation with &&

Responding to events

- React lets you add *event handlers* to your JSX.
- Built-in components like <button> only support built-in browser events like onClick.

React Router

Routing

- SPAs dynamically update sections of a page without full-page reloads
- SPAs introduce challenges, such as managing browser history and routing.

React Router

- React Router is the most popular routing library for React
- Enables navigation among views
- Handle routing *declaratively*.
- <Route path="/home" component={Home} />

React Router

- React Router includes three main packages:
  - react-router: This is the core package for the router
  - react-router-dom: It contains the DOM bindings for React Router. In other words, the router components for websites

- To get started:
  - npm install — save react-router-dom

Using React Router

- The React Router API is based on three components:
- Router: At the core of React Router v6 is the Router component, which provides routing capabilities to the application.
- Routes: The Routes component is used to define route configurations within the application. It replaces the previous Switch component and allows for more flexible route matching and rendering. Routes can be nested and include route elements defined with the Route component.
- The Route component defines a route within the application and specifies the component to render when that route matches the current URL. Routes in React Router v6 use an element prop instead of component, and they match paths inclusively by default.

<Router>

- React Router v6 offers a single <Router> component that abstracts away the underlying routing strategy
- The main job of a <Router> component is to create a history object to keep track of the location (URL).
- When the location changes because of a navigation action, the child component is re-rendered.
- The react-router-dom package offers three higher-level, ready-to-use router components, as
- BrowserRouter: The BrowserRouter component handles routing by storing the routing context in the browser URL and implements backward/forward navigation with the inbuilt history stack
- HashRouter: Unlike BrowserRouter, the HashRouter component doesn’t send the current URL to the server by storing the routing context in the location hash (i.e.,i ndex.html#/profile)
- MemoryRouter: This is an invisible router implementation that doesn’t connect to an external location, such as the URL path or URL hash. The MemoryRouter stores the routing stack in memory but handles routing features like any other router

<Route>

- It renders some UI if the current location matches the route’s path.
- <Route path="/about" component={About}/>
- By default, routes are inclusive, more than one <Route> component can match the URL path and render at the same time.

<Link>

- Create a navigational link to navigate to particular URL

```
<ul>
<li> <Link to="/">Home</Link> </li>
<li> <Link to="/about">About</Link> </li>
</ul>
```

<Outlet>

- An <Outlet> should be used in parent route elements to render their child route elements.
- This allows nested UI to show up when child routes are rendered.
- If the parent route matched exactly, it will render a child index route or nothing if there is no index route.

Redux

Redux

- Redux is a predictable state container for JavaScript apps.
- Used for application State Management
- It is inspired by Facebook’s Flux Architecture

Need for Redux

- In React, data flows from one parent component to the child component.
- The Data flow is unidirectional
- The child components cannot pass data to parents
- The non-parents components cant communicate with each other

- Redux offers a Store- to store all your application state in one place-single source of truth
- Components can dispatch the state changes to the Store- instead of sending it to each components
- Components need the data can subscribe to the State

Principles of Redux

- A single object tree stores all of your application state
- State is read only and changes are triggered by actions.
- The state can only be manipulated by pure functions that are triggered by your actions

Actions

- An action is simply an object that describes a change you want to make to your state.
- Actions are payload on information that send data from your application to the store.
- A standard pattern for actions is the following structure:

```
const action = {
  type: 'ACTION_TYPE',
  payload: 'Some data'
};
```

- The payload is the data that will be used to transform our state.

```
const increment = {
  type: 'INCREMENT'
};
```

- Some actions like incrementing a number do not require any additional data

Action Creator

- Function which create actions

```
export const addNumber = (number) => ({
  type: ADD_NUMBER,
  payload: number
});
```

```
const action = addNumber(7);
```

- Action creators are simply a function that allow us to abstract away the creation of actions, allowing us to easily dispatch an action without having to define all of its properties.

Reducers

- A reducer is the pure function that we will use to transform our store state.
- Actions describe the data that needs to be change, But How the state change happens is the responsibility of the Reducer
- Reducers are triggered whenever an action is dispatched and receive both the current state of that reducer (which will be undefined to begin with) and the action that was dispatched.

```
export const count = (state = 0, action) => {
  switch (action.type) {
    case ADD_NUMBER:
      return state + action.payload;
    default:
      return state;
  }
};
```

Reducer

- Takes in the prevstate and action, and determines what sort of update needs to be done based on action type
- It returns the newstate value
- It returns the prevstate, If no change occurred

- Reducers are pure functions, they do not modify the passed original state, but make their own copy and updates them

- Single Store , Multiple reducer

- Root reducer slices up the state based on keys

Store

- The Store is the object that hold application state.

- Create a Store
- export const store = createStore(count);

- store.dispatch(action) is the method that is used to dispatch an action and subsequently trigger our reducers.

- store.getState() simply returns the current state of the store.

- Register listeners via the subscribe() method

- Store Calculates the state changes and communicate to the Root reducer

Combining Reducers

- For most applications we are going to want to store more than a single number

- export const store = createStore(combineReducers({
- count,
- someOtherReducer
- }));

- Behind the scenes is creating another function that calls all our our reducers with the state that is relevant to them.

Thank You!