React Native

Desenvolvimento de Software e Sistemas Móveis (DSSMV)
Licenciatura em Engenharia de Telecomunicações e Informática
LETI/ISEP

2024/25

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Instituto Superior de Engenharia do Porto

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

1. State Management
2. React Context API
3. Flux
4. Class vs Functional Components
5. Hooks
6. Bibliography

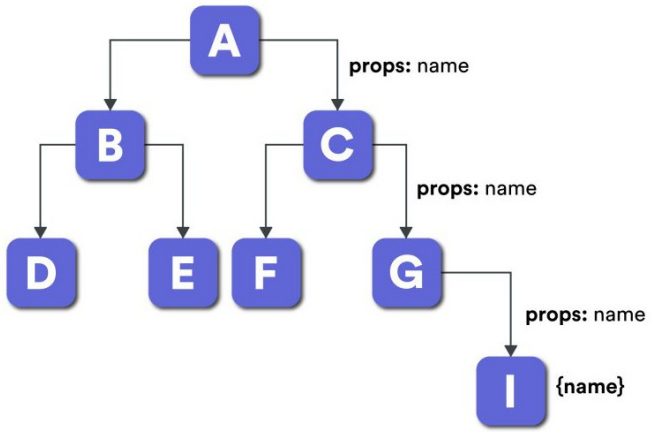
State Management

State management (I)

- A React application will often require some **state to be available in multiple components**.
  - One way to achieve this is to **keep the state in a top-level component** and **pass it down in `props` into every child that needs it**.
    - Passing the state through props is a perfectly good approach to this and will incur no performance penalty
  - Other way is to use a **state management library**, such as Redux or MobX, that, among many other things, provides an API **to inject states into components**.
    - This approach creates dependency of such library

State management (II)

- Passing the state from top-level to low-level component (**prop drilling**)
  - In **a deep component tree**, passing the state through many different levels (many where it may not be used) can clutter (create confusion) the component props and make it a little more **difficult to manage the code**.

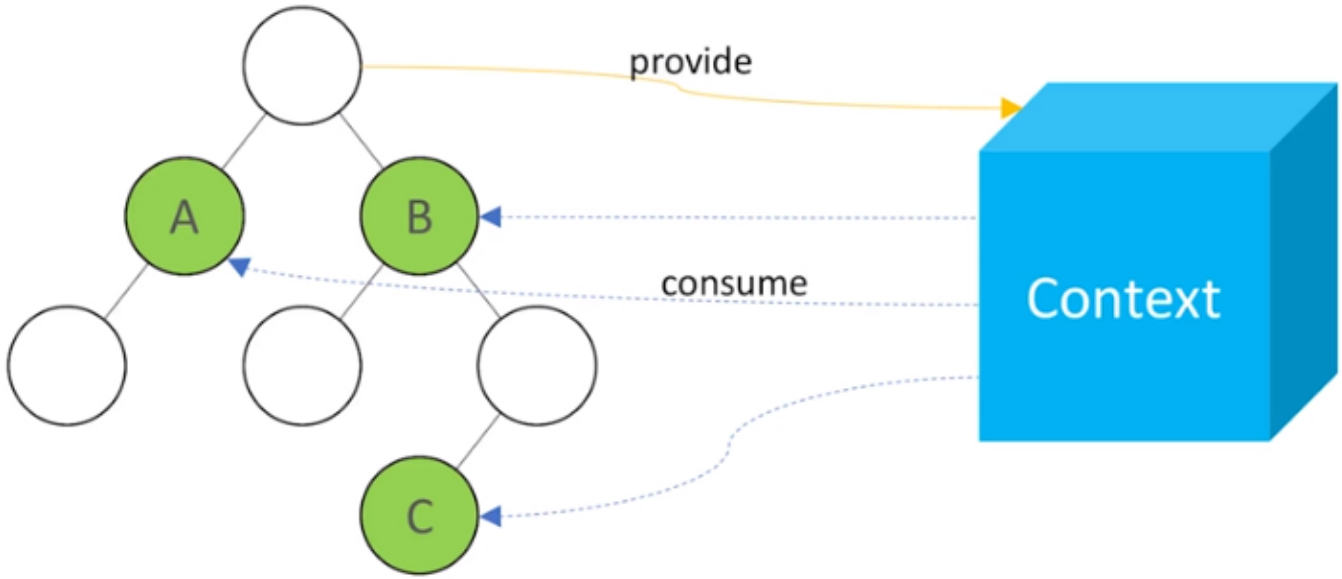


  - Further, it becomes tedious passing props several components down, because (in many cases) **most of the components in between are not interested in these props** and just pass the props to the next child component until it reaches the desired child component.

React Context API

The React Context API (I)

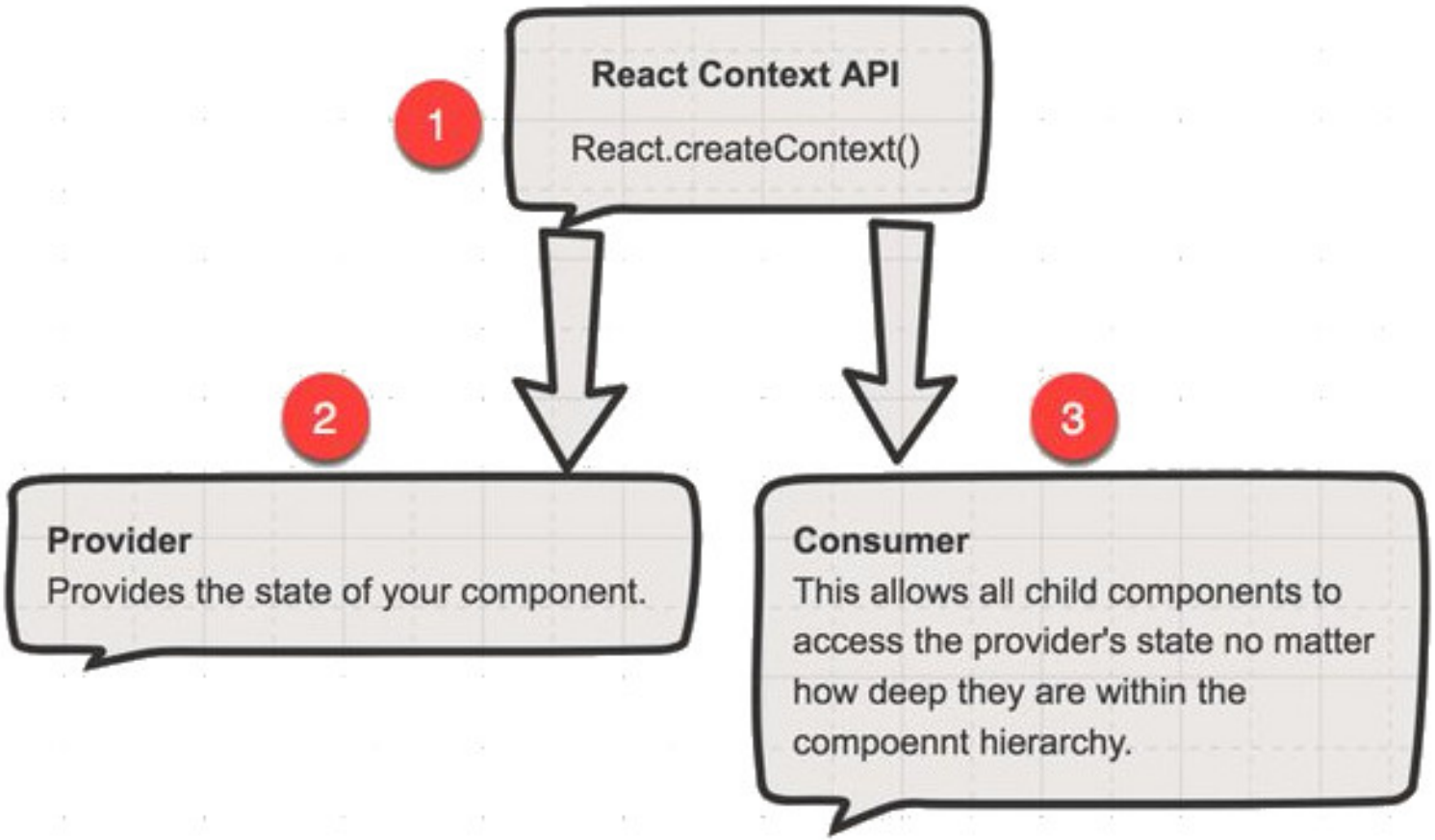
- React uses **provider pattern** in Context API to share data across the tree descendant nodes.
- React Context API is a way to essentially **create global variables that can be passed around** in a React app.



- **Context provides a way to pass data through the component tree** without having to pass props down manually at every level.

The React Context API (II)

1. Create **React context**
2. **Provider** sets the component **state**.
3. **Consumer** (children) pulls the **state** from Provider (via Context).



Creating a Context

- A React context is created by using the `createContext` function.
  - This function returns an object containing `Provider` and `Consumer` members.
    - These are exported to be used into the other components

```
import React from 'react';
const AppContext = React.createContext();
export const {Provider, Consumer} = AppContext;
export default AppContext;
```

Providing Context (I)

- `AppProvider` component is a, normal, component that will provide its state and functions to all its child components.

```
const darkTheme = {
  name: "dark",...
};
const lightTheme = {
  name: "light",...
};
class AppProvider extends Component {
  constructor(props) {
    super(props)
    this.state = {
      selectedTheme: darkTheme,
      availableThemes: [darkTheme, lightTheme],
    };
  }
  selectTheme = (name) => {
    const theme = this.state.availableThemes.find(theme => theme.name === name);
    this.setState({...this.state, selectedTheme: theme });
  }
...
```

Providing Context (II)

- `AppProvider` component renders a `Provider` component.

```
class AppProvider extends Component {
  ...

  render() {
    const availableThemeNames = this.state.availableThemes.map((theme) =>
          theme.name);
    return (
    <Provider value={{
       availableThemeNames: availableThemeNames,
       selectedTheme: this.state.selectedTheme,
       selectTheme: this.selectTheme,
    }}>
      {this.props.children}
    </Provider>
    );
   }
}
```

- `Provider` component takes a prop, `value`, that will be provided as the context.
  - `props.children` is a special property of React components which contains any child elements defined within the component

Providing Context (III)

- `AppProvider` provides it context to the `Screen` component as well as the all components in the tree, where `Screen` is the root.

```
import React, { Component } from 'react';
import AppProvider from './AppProvider';
import Screen from '../screens/Screen';
class App extends Component {
  render() {
    return (
      <AppProvider>
        <Screen />
      </AppProvider>
    );
  }
}
export default App;
```

- All consumer components that are descendants of a `AppProvider` will re-render whenever the `Provider`'s `value` prop changes.

Consuming Context

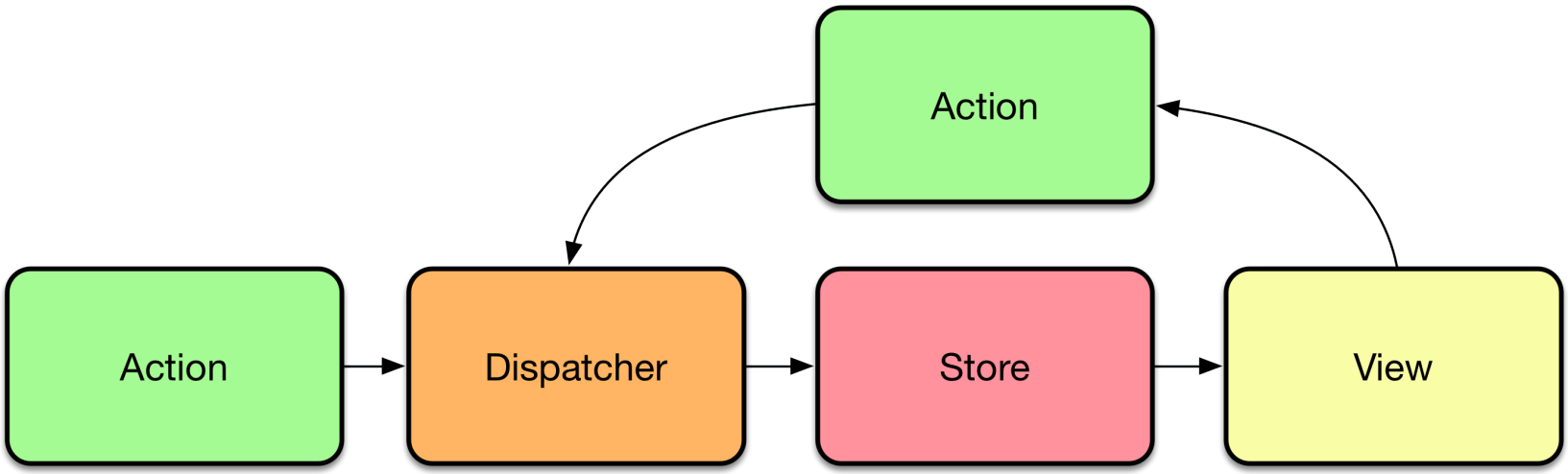
- `Consumer` component subscribes the context provided by `Provider`.
- Requires a function as a child.
  - The function receives the current context `value` and **returns a React node**.
    - The `context` argument passed to the function will be equal to the `value` prop of the `Provider` for this context.

```
import { Consumer } from './AppContext';
class Footer extends Component {
  render() {
  return (
    <Consumer>{
      (context) => {
        const { name, textColor } = context.selectedTheme;
        return (
          <View>
            <Text style={{ color: textColor }}>{name}</Text>
          </View>
        )
      }
    }
    </Consumer>
  );
```

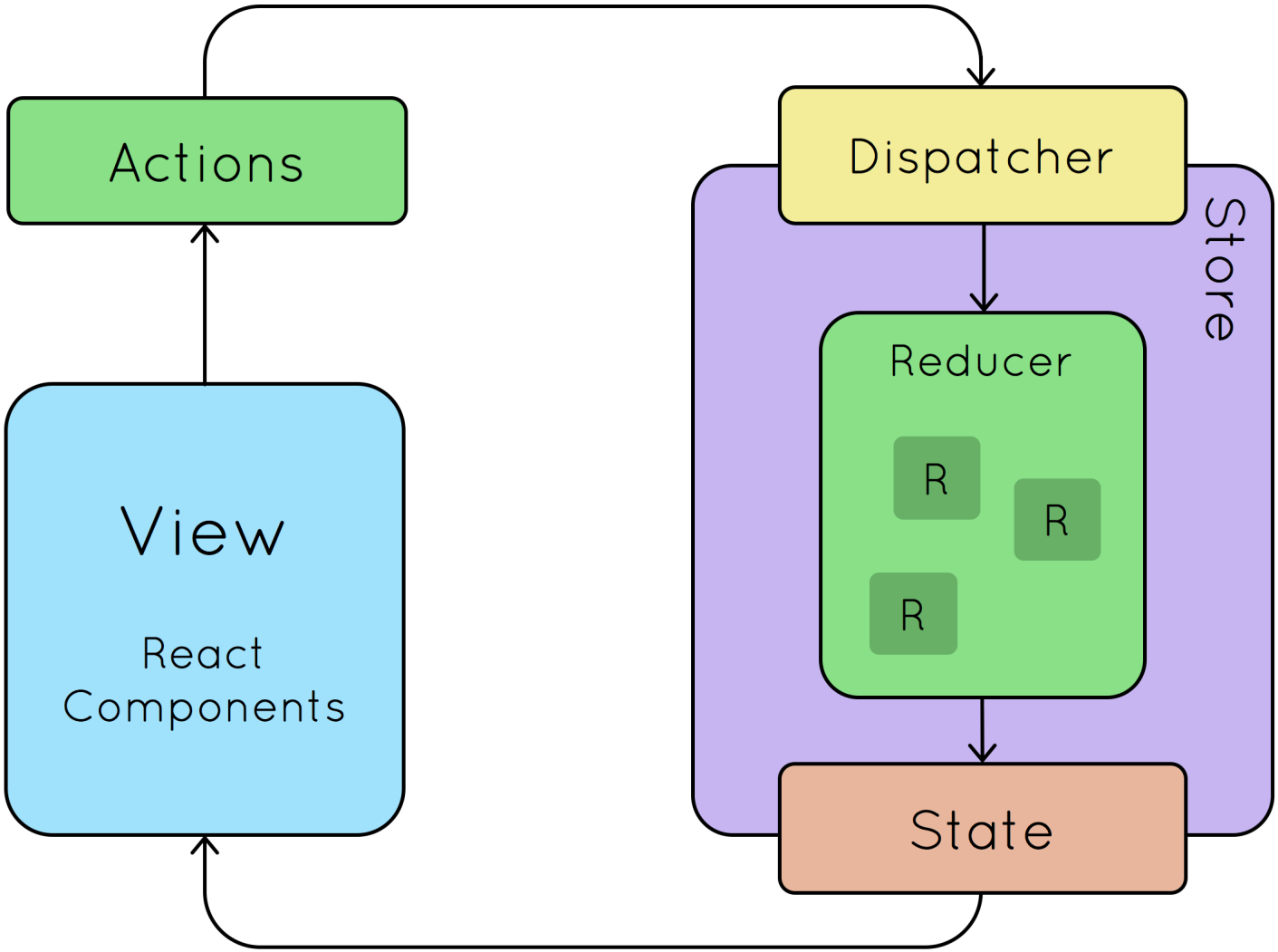
Flux

What is Flux?

- Flux is an **architecture**.
  - It is not a framework or a library.
  - It is simply architecture that employs **Unidirectional Data Flow** concept.



Flux with Context API (I)



Flux with Context API (II)

- **Store**: The store is where the application's `state` is housed.
- **Actions**: These are objects that are used to send data to the store.
  - They typically have two properties:
    - A `type` property for describing what the action does
    - A `payload` property that contains the information that should be changed in the `state`.
- **Reducers**: These are pure functions that implement the action behavior.
  - A `reducer` is a function that determines changes to an application's `state` and it uses the `action` it receives to determine this change.
    - It takes the current application `state`, perform an `action`, and then return a new `state`

.

Flux with Context API: Store (I)

- The store is implemented into `AppProvider` component.

```
const darkTheme = {...};
const lightTheme = {...};
class AppProvider extends Component {
  constructor(props) {
    super(props)
    this.state = {
      selectedTheme: darkTheme,
      availableThemes: [darkTheme,lightTheme],
    };
  }
  dispatch = (action) => this.setState((state) => reducer(state, action));
  render() {
    return (
      <Provider value={{
        state: this.state,
        dispatch: this.dispatch}}>
      {this.props.children}
      </Provider>
    );
  }
}
```

Flux with Context API: Store (II)

- `setState()`: arguments
  - Receiving an object

```
constructor(props) {
  super(props)
  this.state = {value: 0,};
}
...
this.setState({value: 1});
```

    - Such object will be merged into component current `state`
  - Receiving a function

```
constructor(props) {
  super(props)
  this.state = {value: 0,};
}
...
this.setState((state) => ({ value: state.value + 1}));
```

    - If `setState` receive a function as argument, React will call it with the at-call-time-current `state` and expect that it returns an object to merge into `state`.

Flux with Context API: Reducers & Actions (I)

- Action

```
export const SELECT_THEME = 'SELECT_THEME';
export const selectTheme = (name) =>{
  return {
    type: SELECT_THEME,
    payload: {
      theme: name
    }
  }
};
```

- An action is an object that contains two keys (`type` and `payload`) and their values.
  - The `state` update that happens in the reducer is always dependent on the value of `type`.
  - The `payload` contains what is used to update the application' `state`.

Flux with Context API: Reducers & Actions (II)

- Reducer

```
import {SELECT_THEME} from './Actions'
const reducer = (state, action) => {
  switch (action.type) {
    case SELECT_THEME:
      const name = action.payload.theme;
      const theme = state.availableThemes.find((theme) => (theme.name === name))
            ;
      return {
        ...state,
        selectedTheme: theme,
      };
    default:
      return state;
  }
};
export default reducer;
```

- The `reducer` function takes two parameters(`state` and `action`) and returns a new `state`.
  - The `state` is meant to be immutable, meaning it should not be changed directly.
    - To create an updated state, we can make use of the spread(. . .)

Flux with Context API: View

```
import { selectTheme } from './Actions';
class ThemeItem extends Component {
  ...
  render() {
  return (
    <Consumer>{
      (context) => {
        const { state, dispatch } = context;
        const { selectedTheme } = state;
        const { themeName } = this.props;
        const action = selectTheme(themeName);
        return (
          <View>
            <TouchableOpacity style={[styles.button, { backgroundColor: selectedTheme.
                 btBackgroundColor }]}
            onPress={() => dispatch(action)}>
              <Text style={{ color: selectedTheme.btTextColor }}> {themeName} </Text>
            </TouchableOpacity>
          </View >
        );
        }
      }
    </Consumer>
  );
...
```

Class vs Functional Components

Class component

```
class Person extends Component {
  constructor(props){
    super(props);
    this.state = { .... }
  }
  render() {
    return (
    <View>
      <Text>Hello, {this.props.name}</Text>
    </View>);
  }
}
  export default Person;
```

- Class components extend the `Component` class in React.
- They have `state` and `props`
- Sometimes called **smart** or **stateful** components as they tend to implement logic and state.

Functional Components (I)

- Functional components are basic JavaScript functions.
  - **Arrow functions**

```
const Person = (props) => {
  return(
    <View>
      <Text>Hello, {props.name}</Text>
    </View>);
};
export default Person;
```

  - **Regular functions**, `function` keyword.

```
    function Person (props){
    return (
      <View>
        <Text>Hello, {props.name}</Text>
      </View>);
    }
    export default Person;
```

Functional Components (II)

- Sometimes referred to as **dumb** or **stateless** components as they simply accept data and display them.
  - They are mainly responsible for rendering UI.
- There is no `render` method.
- They can accept and use `props`.
- They are easier to read, debug, and test.
- They offer performance benefits, decreased coupling, and greater reusability.
- They should be favored if you do not need to make use of React state.

Hooks

What are React Hooks?

- React Hooks are in-built functions that allow React developers to use **state** and **lifecycle** methods inside **functional components**,
- Useful hooks
  - `useState`, it adds React state to function components.
  - `useContext`, it allows access to the context.
  - `useReducer`, it is more suitable than `useState` for complex state.
  - `useEffect`, it allows side effects within the functional component.
    - For instance, the lifecycle methods are all all handled by the `useEffect` hook in functional components.

useState (I)

- `const [state, setState] = useState(initialState)`
  - Returns a stateful value (`state`), and a function to update it, (`setState`).
  - During the initial render, the returned state (`state`) is the same as the value passed as the first argument (`initialState`).
  - The `setState` function is used to update the `state`.
    - It accepts a new `state` value and enqueues a **re-render of the component**.

```
function Counter() {
  const [count, setCount] = React.useState(0);
  return (
    <View>
      <Text> Count: {count}</Text>
      <Button onPress={() => setCount(0)} title="Reset"/>
      <Button onPress={() => setCount(count => count - 1)} title="-"/>
      <Button onPress={() => setCount(count => count + 1)} title="+"/>
    </View>
    );
  }
}
```

`useState` (II)

- The `initialState` argument is the state used during the **initial render**.

```
const initialState = {...};
const [state, setState] = useState(initialState);
}
```

- **In subsequent renders, it is disregarded**.
- If the initial state is the result of an expensive computation, you may provide a **function** instead, which will **be executed only on the initial render**:

```
const [state, setState] = useState(() => {
   const initialState = someExpensiveComputation();
   return initialState;
  });
}
```

useContext (I)

- `const value = useContext(MyContext);`
  - Accepts a context object (the value returned from `React.createContext`) and returns the current context value for that context.
  - A component calling `useContext` will **always re-render** when the context value changes.

```
import React, { useContext } from 'react';
import { View, Text } from 'react-native';
import AppContext from './AppContext';

const Footer = () => {
  const { state } = useContext(AppContext);
  const { name, textColor } = state.selectedTheme;
  return (
    <View>
      <Text style={{ color: textColor }}>{name}</Text>
    </View>
  );
}
export default Footer;
```

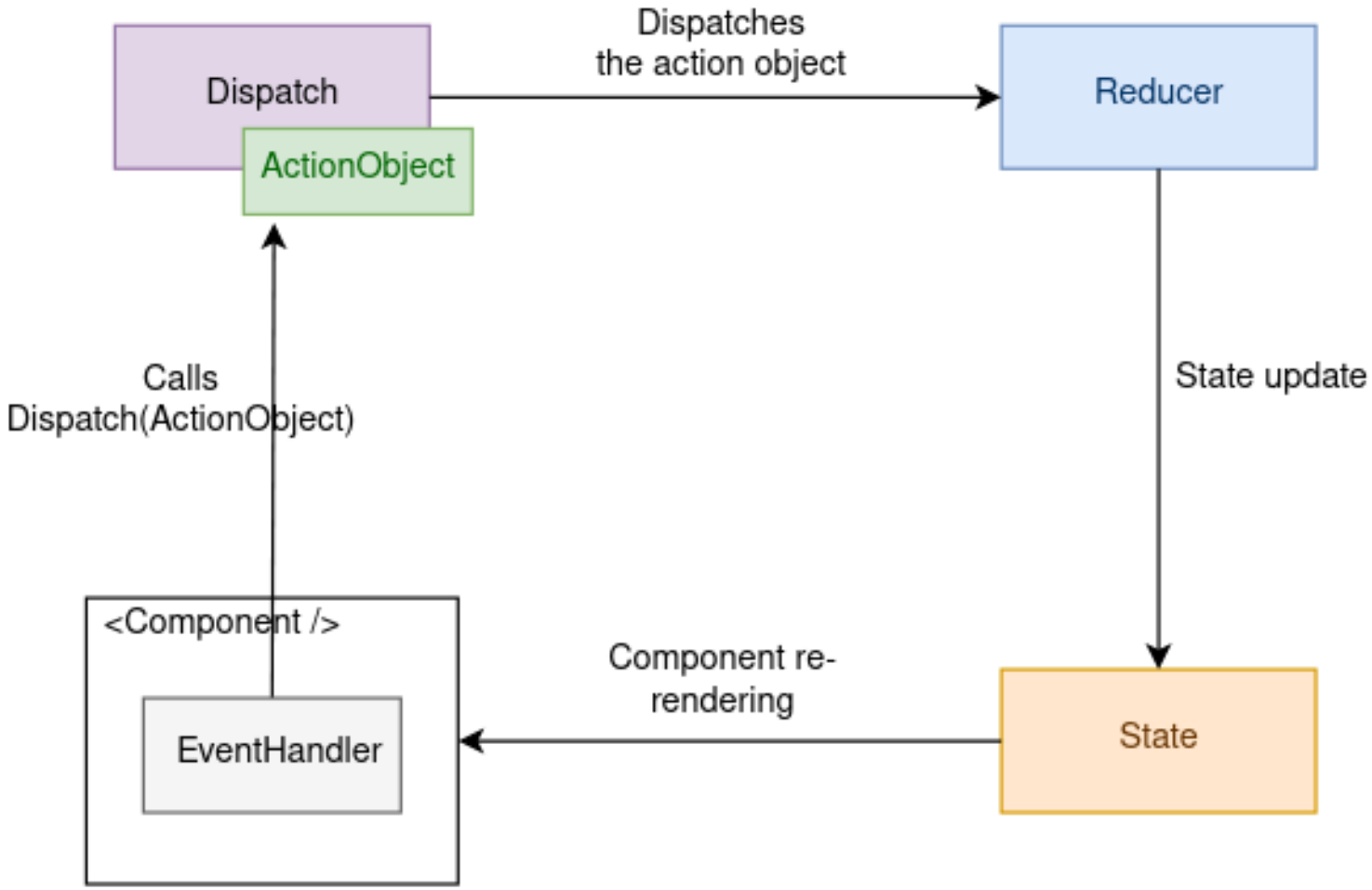
`useReducer` (I)

- `const [state, dispatch] = useReducer(reducer, initialState)`
  - An alternative to `useState`.
  - Accepts a `reducer(state, action) => newState`, and returns the current `state` paired with a `dispatch` method.

useReducer (II)

```
function reducer(state, action) {
  switch (action.type) {
    case 'reset':
      return {...state, count: action.payload.count};
    case 'increment':
      return {...state,count: state.count + 1};
    case 'decrement':
      return {...state,count: state.count - 1};
    default:
      return state;
  }
}
function Counter(props) {
  const initialState = {count: props.initialCount};
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <View>
      <Text>Count: {state.count}</text>
      <Button onPress={() => dispatch({type: 'reset', payload:initialState})} title="
          Reset"/>
      <Button onPress={() => dispatch({type: 'decrement'})} title="-"/>
      <Button onPress={() => dispatch({type: 'increment'})} title="+"/>
    </View>
  );
}
```

useReducer (III)



`useEffect` (I)

- `useEffect(fn,[])`
  - Accepts two arguments: a function, `fn` and optionally an array of values `[]`.
  - The function runs when the component is first rendered, and on every subsequent re-render/update.
    - By default, function run after every completed render, but you can choose to fire them only when certain values have changed.
  - It handles lifecycle events directly inside function components
    - `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`, all methods with one function.

`useEffect` (II)

- `useEffect(fn)`, without second argument

```
useEffect(
  () => {
  console.log('Hi from the effect hook!');
  }
);
```

  - Function will be executed every time the component renders or re-renders.
  - Equivalent to :`componentDidMount` and `componentDidUpdate`

`useEffect` (III)

- `useEffect(fn), []`, with second argument (an empty array)

```
useEffect(
  () => {
  console.log('Hi from the effect hook!');
  }, []
);
```

  - Passing an empty array makes the hook execute the function only once when the component is mounted.
  - Equivalent to :`componentDidMount`

`useEffect` (IV)

- `useEffect(fn), [a,b,c,...]`, with second argument (an array of variables)

```
useEffect(
() => {
console.log('Hi from the effect hook!');
}, [someVariable, ...]
);
```

  - If any of these variables into the array change after an update, the function `fn` will be executed.
  - Equivalent to : `componentDidUpdate`

useEffect (V)

- `useEffect(fn), []`, return a function inside the `fn` of the `useEffect`

```
useEffect(
() => {
  console.log('Hi from the effect hook!');
   return () => console.log('Hi, I am passing away!');
},[]
);
```

  - Equivalent to : `componentWillUnmount`

Bibliography

Resources

- David Flanagan, "JavaScript: The Definitive Guide", O'Reilly Media, Inc., 2020
- Adam Boduch and Roy Derks, "React and React Native, Third Edition,A complete hands-on guide to modern web and mobile development with React.js"
- Dan Ward, "React Native Cookbook Second Edition Step-by-step recipes for solving common React Native development problems"
- `https://reactnative.dev/`
- `https://reactjs.org/`
- `https://reactnavigation.org/`