PB138: React and State Management

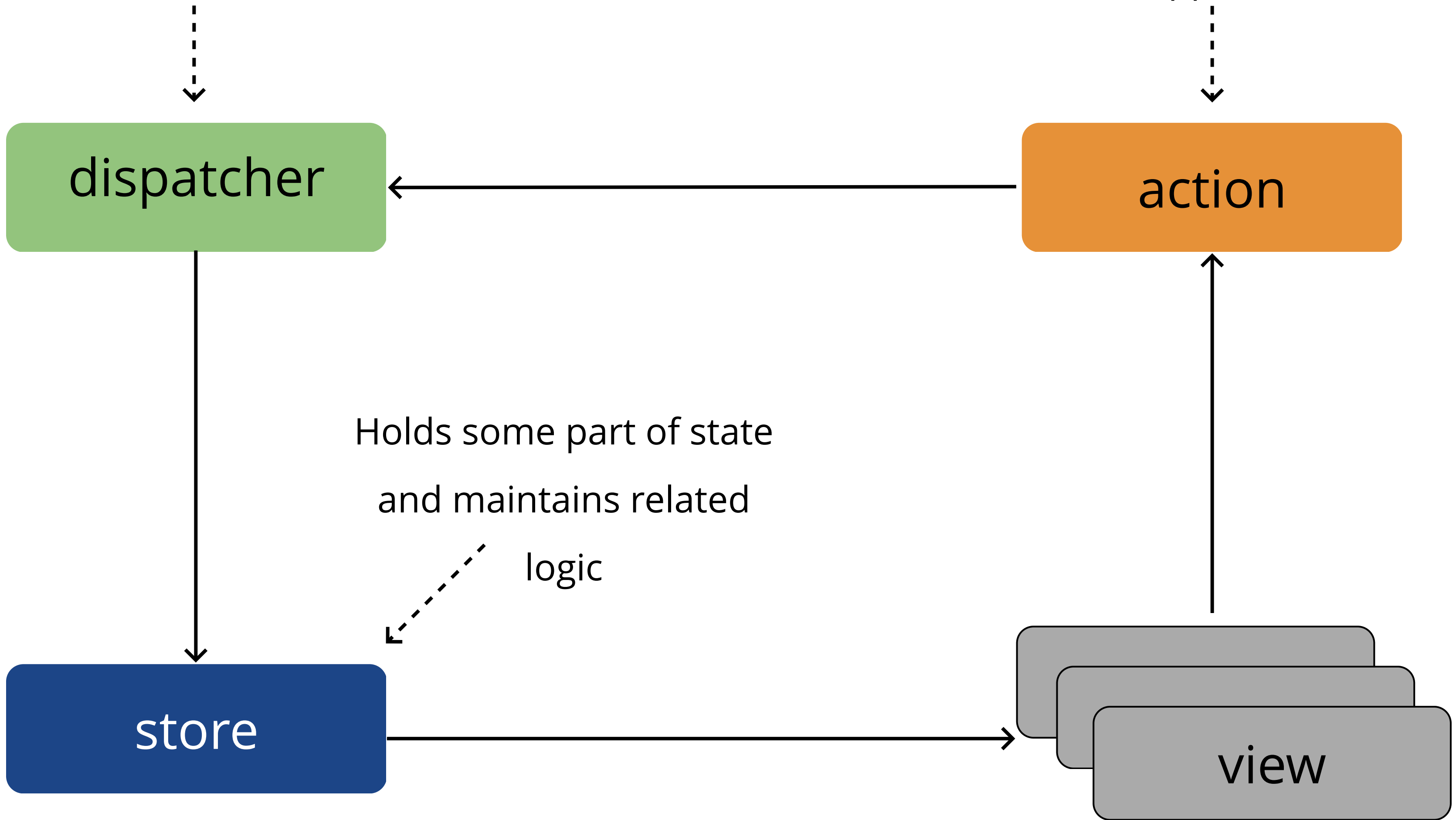
presented by Lukas Grolig

What is state management?

- Something that stores data for your components so you don‘t have to pass them from the parent.
- It also handles actions that happen in your components.
- Shares state across multiple components

What is Flux?

- It is architure for building client-side websites. It is utiling unidirectional data flow between component. It‘s pattern implemented by state management frameworks.


Component routing data to stores
(usually by registered callbacks)
Object describing what
happened
dispatcher
action
Holds some part of state
and maintains related
logic
store
view

Different implementations of Flux

React Context

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

Creating context

```
1 const ThemeContext = React.createContext('light');
```

First create context and provide default value. We suggest to export context and import it in components later.

Providing values in context

```
class App extends React.Component {
    render() {
        return (
            <ThemeContext.Provider value="dark">
                <Toolbar />
            </ThemeContext.Provider>
        );
    }
}
```

The provider will pass a value to all components in the tree below. No matter how deep.

Or pass state from any component to the top.

Consuming values from context

```
class MyComponent extends React.Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {(context) => {
                    <Toolbar theme={context} />
                }}
            </ThemeContext.Consumer>
        );
    }
}
```

Component consuming context is wrapped by it.

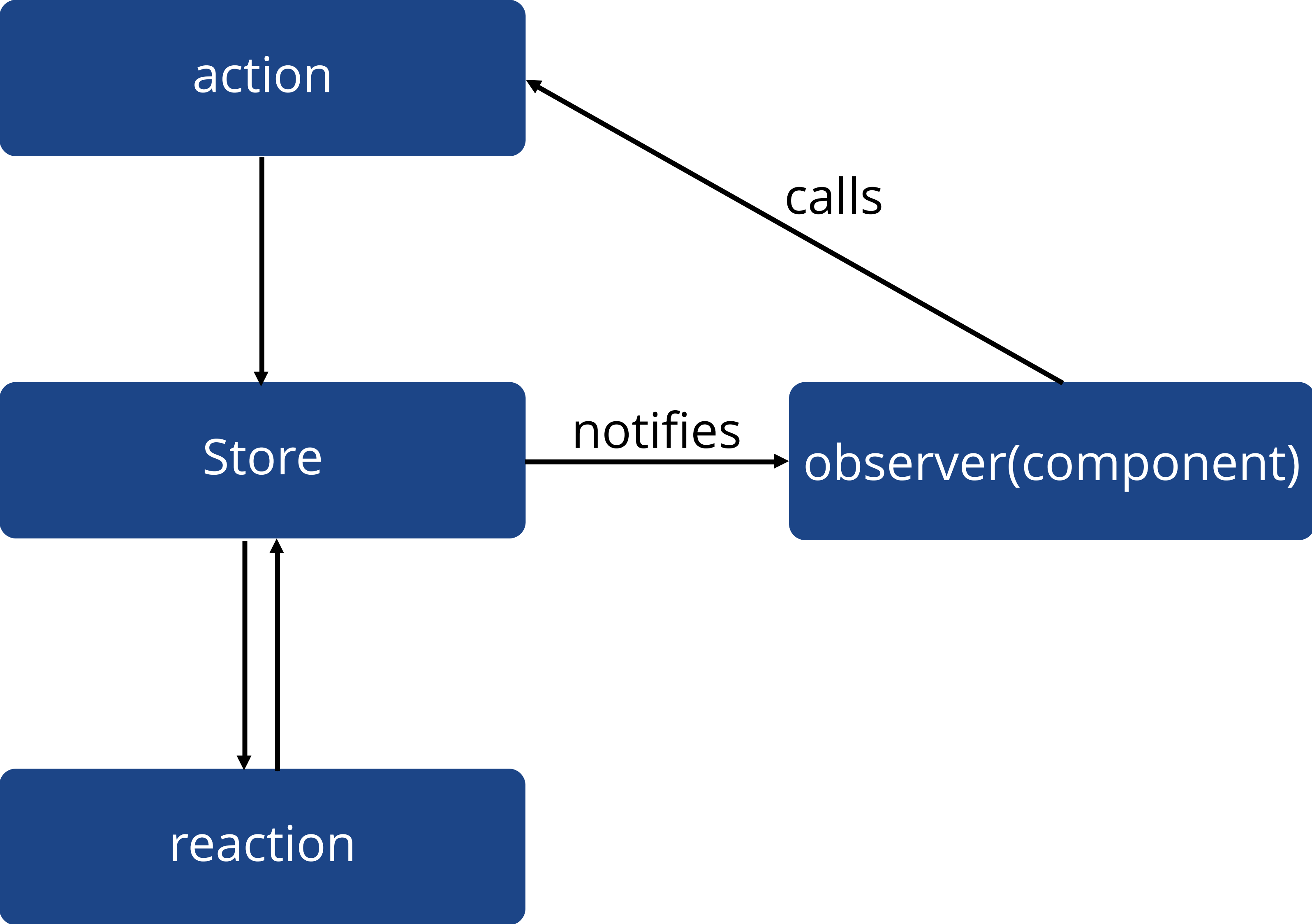
Or using hooks

```
1  function ThemedButton() {
2    const theme = useContext(ThemeContext);
3    return (
4      <button style={{
5          background: theme.background,
6          color: theme.foreground }}>
7        I am styled by theme context!
8      </button>
9    );
10 }
```

MobX

- It is a simple, scalable, and battle-tested state management solution.
- MobX uses an object-oriented approach to state management.
- Realy easy to use.

Data flow in MobX is similar as in Flow


action
calls
Store
notifies
observer(component)
reaction

How to implement a store

Observable

```
import { makeObservable, observable, computed, action,

class Doubler {
    value

    constructor(value) {
        makeObservable(this, {
            value: observable,
            double: computed,
            increment: action,
            fetch: flow
        })
        this.value = value
    }
}
```

makeObservable will provide notification mechanisms for your data to be consumed by the observer.

If you don't have subclasses or inherit from super try to use makeAutoObservable.

Computed property

```
1  class Doubler {
2      value
3
4      constructor(value) {
5          // ...
6      }
7
8      get double() {
9          return this.value * 2
10     }
11 }
```

The computed
value uses other observables to
calculate some value on top of them.

Observer

```
1 const doubler = new Doubler(2);
2
3 const DoublerView = observer(({ doubler })
4         => <span>Double: {doubler.double}</span>)
```

Observer enables a component to listen to store changes. The store is passed in props.

Action

```
import { makeObservable, observable, action } from "mobx"

class Doubler {
    value = 0

    constructor(value) {
        makeObservable(this, {
            value: observable,
            increment: action
        })
    }

    increment() {
        // Intermediate states will not become visible to observers.
        this.value++
        this.value++
    }
}
```

If some function in the store modifies state, it is an action.

Autorun

```
1 const counter = observable({ count: 0 })
2
3 // Sets up the autorun and prints 0.
4 const disposer = autorun(() => {
5     console.log(counter.count)
6 })
7
8 // Prints: 1
9 counter.count++
```

When something in the store changes than your reaction is called. Autorun is like reaction, but you don't specify on what props you listen.

Integration of a store

```
1  import {observer} from 'mobx-react-lite'
2  import {createContext, useContext} from "react"
3
4  const TimerContext = createContext<Timer>()
5
6  const TimerView = observer(() => {
7      // Grab the timer from the context.
8      const timer = useContext(TimerContext) // See the Timer definition abov
9      return (
10         <span>Seconds passed: {timer.secondsPassed}</span>
11     )
12 })
13
14 ReactDOM.render(
15     <TimerContext.Provider value={new Timer()}>
16         <TimerView />
17     </TimerContext.Provider>,
18     document.body
19 )
```

Redux

Redux is a predictable state container for JavaScript apps. As the requirements for JavaScript single-page applications have become increasingly complicated, our code must manage more state than ever before.

This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server.

UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.

Reducer hooks

```
1  const initialState = {count: 0};
2
3  function reducer(state, action) {
4    switch (action.type) {
5      case 'increment':
6        return {count: state.count + 1};
7      case 'decrement':
8        return {count: state.count - 1};
9      default:
10       throw new Error();
11   }
12 }
13
14 function Counter() {
15   const [state, dispatch] = useReducer(reducer, initialState);
16   return (
17     <>
18       Count: {state.count}
19       <button onClick={() => dispatch({type: 'decrement'})}>-</button>
20       <button onClick={() => dispatch({type: 'increment'})}>+</button>
21     </>
22   );
23 }
```

State is read-only

The only way to change the state is to emit an action, an object describing what happened.

```
1 store.dispatch({
2         type: 'COMPLETE_TODO‘,
3         index: 1
4 })
```

Changes are made with pure functions

To specify how the state tree is transformed by actions, you write pure reducers.

```
1 function todos(state = [], action) {
2         switch (action.type) {
3                 case 'COMPLETE_TODO':
4                         return state.map((todo, index) => { ... })
5                 default:
6                         return state
7 }
```

Actions

Actions are **payloads of information** that send data from your application to your store. They are the only source of information for the store. You send them to the store using *store.dispatch()*.

```
const addTodoAction = {
  type: 'todos/todoAdded',
  payload: 'Buy milk'
}
```

Reducers

Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions only describe what happened, but don't describe how the application's state changes.

```
1  const initialState = { value: 0 }
2
3  function counterReducer(state = initialState, action) {
4    // Check to see if the reducer cares about this action
5    if (action.type === 'counter/incremented') {
6      // If so, make a copy of `state`
7      return {
8        ...state,
9        // and update the copy with the new value
10       value: state.value + 1
11     }
12   }
13   // otherwise return the existing state unchanged
14   return state
15 }
```

Store

The Store is the object that brings everything together.
The store has the following responsibilities:
Holds application state;
Allows access to state via *selectors;*
Allows the state to be updated via *dispatch(action);*
Registers listeners via *subscribe(listener);*
Handles unregistering of listeners via the function returned by *subscribe(listener)*

```
1 import { configureStore } from '@reduxjs/toolkit'
2
3 const store = configureStore({ reducer: counterReducer })
4
5 console.log(store.getState())
6 // {value: 0}
```

Selector

```
1  import React from 'react'
2  import { useSelector } from 'react-redux'
3  import TodoListItem from './TodoListItem'
4
5  const selectTodos = state => state.todos
6
7  const TodoList = () => {
8    const todos = useSelector(selectTodos)
9
10   // since `todos` is an array, we can loop over it
11   const renderedListItems = todos.map(todo => {
12     return <TodoListItem key={todo.id} todo={todo} />
13   })
14
15   return <ul className="todo-list">{renderedListItems}</ul>
16 }
17
18 export default TodoList
```

Dispatch

```
import React, { useState } from 'react'
import { useDispatch } from 'react-redux'

const Header = () => {
  const [text, setText] = useState('')
  const dispatch = useDispatch()

  const handleChange = e => setText(e.target.value)

  const handleKeyDown = e => {
    const trimmedText = e.target.value.trim()
    // If the user pressed the Enter key:
    if (e.key === 'Enter' && trimmedText) {
      // Dispatch the "todo added" action with this text
      dispatch({ type: 'todos/todoAdded', payload: trimmedText })
      // And clear out the text input
      setText('')
    }
  }

  return (
    <input
      type="text"
      placeholder="What needs to be done?"
      autoFocus={true}
      value={text}
      onChange={handleChange}
      onKeyDown={handleKeyDown}
    />
  )
}
```

Pass store using Context

```jsx
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'

import App from './App'
import store from './store'

ReactDOM.render(
  // Render a `<Provider>` around the entire `<App>`,
  // and pass the Redux store to as a prop
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)
```

Recoil

New kid on the block

Root

```
import React from 'react';
import {
  RecoilRoot,
  atom,
  selector,
  useRecoilState,
  useRecoilValue,
} from 'recoil';

function App() {
  return (
    <RecoilRoot>
      <CharacterCounter />
    </RecoilRoot>
  );
}
```

Root

```
1  import React from 'react';
2  import {
3    RecoilRoot,
4    atom,
5    selector,
6    useRecoilState,
7    useRecoilValue,
8  } from 'recoil';
9
10 function App() {
11   return (
12     <RecoilRoot>
13       <CharacterCounter />
14     </RecoilRoot>
15   );
16 }
```

Atom

```
1 const textState = atom({
2   key: 'textState', // unique ID (with respect to
3   default: '', // default value (aka initial valu
4 });
```

useRecoilState

```
function CharacterCounter() {
  return (
    <div>
      <TextInput />
      <CharacterCount />
    </div>
  );
}

function TextInput() {
  const [text, setText] = useRecoilState(textState);

  const onChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <input type="text" value={text} onChange={onChange} />
      <br />
      Echo: {text}
    </div>
  );
```

Selector

```
const charCountState = selector({
  key: 'charCountState', // unique ID (with respect to other ato
  get: ({get}) => {
    const text = get(textState);

    return text.length;
  },
});

function CharacterCount() {
  const count = useRecoilValue(charCountState);

  return <>Character Count: {count}</>;
}
```

Working with REST API

Using SWR

```
1  import useSWR from 'swr'
2
3  const fetcher = url => fetch(url).then(r => r.json());
4
5  function Profile() {
6    const { data, error } = useSWR('/api/user', fetcher)
7
8    if (error) return <div>failed to load</div>
9    if (!data) return <div>loading...</div>
10   return <div>hello {data.name}!</div>
11 }
```

Using SWR with boundaries

```
import { ErrorBoundary, Suspense } from 'react'
import useSWR from 'swr'

function Profile () {
  const { data } = useSWR('/api/user', fetcher, { suspense: true })
  return <div>hello, {data.name}</div>
}

function App () {
  return (
    <ErrorBoundary fallback={<h2>Could not fetch posts.</h2>}>
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <Profile />
      </Suspense>
    </ErrorBoundary>
  )
}
```

Questions?

Ok, that's it for today.