07. REDUX Usage with REACT.

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux works especially well with libraries like **React** and Deku because they let you describe UI as a function of state, and Redux emits state updates in response to actions.

We will use React to build our simple todo app, and cover the basics of how to use React with Redux.

Note: see the official React-Redux docs at https://react-redux.js.org for a complete guide on how to use Redux and React together.

7.1. Installing REACT REDUX.

React bindings are not included in Redux by default. You need to install them explicitly:

Example:

```
npm install --save react-redux
```

If you don't use npm, you may grab the latest UMD build from unpkg (either a development or a production build). The UMD build exports a global called window.ReactRedux if you add it to your page via a <script> tag.

https://unpkg.com/react-redux@latest/dist/react-redux.js

7.2. Presentational and Container Components

React bindings for Redux separate presentational components from container components. This approach can make your app easier to understand and allow you to more easily reuse components. Here's a summary of the differences between presentational and container components (but if you're unfamiliar, we recommend that you also read **Dan Abramov**'s original article describing the concept of presentational and container components):

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux store. This and the design brief below do not imply container components must be near the top of the component tree. If a container component becomes too complex (i.e. it has heavily nested presentational components with countless callbacks being passed down), introduce another container within the component tree as noted in the FAQ.

Technically you could write the container components by hand using **store.subscribe().** We don't advise you to do this because React Redux makes many performance optimizations that are hard to do by hand. For this reason, rather than write container components, we will generate them using the **connect()** function provided by React Redux, as you will see below.

7.3. Designing Component Hierarchy

Remember how we designed the shape of the root state object? It's time we design the UI hierarchy to match it. This is not a Redux-specific task. Thinking in React is a great tutorial that explains the process. https://facebook.github.io/react/docs/thinking-in-react.html

Our design brief is simple. We want to show a list of todo items. On click, a todo item is crossed out as completed. We want to show a field where the user may add a new todo. In the footer, we want to show a toggle to show all, only completed, or only active todos.

7.3.1. Designing Presentational Components

I see the following presentational components and their props emerge from this brief:

- **TodoList** is a list showing visible todos.
  - **todos**: Array is an array of todo items with { id, text, completed } shape.
  - **onTodoClick**(id: number) is a callback to invoke when a todo is clicked.

- **Todo** is a single todo item.
  - **text**: string is the text to show.
  - **completed**: boolean is whether the todo should appear crossed out.
  - **onClick**() is a callback to invoke when the todo is clicked.

- **Link** is a link with a callback.
  - **onClick**() is a callback to invoke when the link is clicked.

- **Footer** is where we let the user change currently visible todos.

- **App** is the root component that renders everything else.

They describe the **look** but don't know **where** the data comes from, or **how** to change it. They only **render** what's given to them. If you migrate from Redux to something else, you'll be able to keep all these components exactly the same. **They have no dependency on Redux.**

7.3.2. Designing Container Components

We will also need some container components to connect the presentational components to Redux. For example, the presentational **TodoList** component needs a container like **VisibleTodoList** that subscribes to the Redux store and knows how to apply the current visibility filter. To change the visibility filter, we will provide a **FilterLink** container component that renders a Link that dispatches an appropriate action on click:

- **VisibleTodoList** filters the todos according to the current visibility filter and renders a TodoList.

- **FilterLink** gets the current visibility filter and renders a Link.
  - **filter**: string is the visibility filter it represents.

7.3.3. Designing Other Components

Sometimes it's hard to tell if some component should be a presentational component or a container. For example, sometimes form and function are really coupled together, such as in the case of this tiny component:

- **AddTodo** is an input field with an "Add" button

Technically we could split it into two components but it might be too early at this stage. It's fine to mix presentation and logic in a component that is very small. As it grows, it will be more obvious how to split it, so we'll leave it mixed.

7.4. Implementing Components

Let's write the components! We begin with the presentational components so we don't need to think about binding to Redux yet.

7.4.1. Implementing Presentational Components

These are all normal React components, so we won't examine them in detail. We write functional stateless components unless we need to use local state or the lifecycle methods. This doesn't mean that **presentational components have to be functions**—it's just easier to define them this way. If and when you need to add local state, lifecycle methods, or performance optimizations, you can convert them to classes.

Example: components/Todo.js

```
import React from 'react'
import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

Example: components/TodoList.js

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={index} {...todo} onClick={() =>
onTodoClick(index)} />
    ))}
  </ul>
```

```
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
      text: PropTypes.string.isRequired
    }).isRequired
  ).isRequired,
  onTodoClick: PropTypes.func.isRequired
}

export default TodoList
```

Example: components/Link.js

```
import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a
      href=""
      onClick={e => {
        e.preventDefault()
        onClick()
      }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

Example: components/Footer.js

```
import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink
filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {', '}
    <FilterLink
filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {', '}
    <FilterLink
filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)

export default Footer
```

7.4.2. Implementing Container Components

Now it's time to hook up those presentational components to Redux by creating some containers. Technically, a container component is just a React component that uses **store.subscribe()** to read a part of the Redux state tree and supply props to a presentational component it renders. You could write a container component by hand, but we suggest instead generating container components with the React Redux library's **connect()** function, which provides many useful optimizations to prevent unnecessary re-renders. (One result of this is that you shouldn't have to worry about the React performance suggestion of implementing **shouldComponentUpdate** yourself.)

https://facebook.github.io/react/docs/advanced-performance.html

To use **connect**(), you need to define a special function called **mapStateToProps** that describes how to transform the current Redux store state into the props you want to pass to a presentational component you are wrapping. For example, VisibleTodoList needs to calculate todos to pass to the TodoList, so we define a function that filters the **state.todos** according to the **state.visibilityFilter**, and use it in its mapStateToProps:

Example:

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
```

```
        return todos.filter(t => !t.completed)
      case 'SHOW_ALL':
      default:
        return todos
    }
  }

  const mapStateToProps = state => {
    return {
      todos: getVisibleTodos(state.todos, state.visibilityFilter)
    }
  }

  function visibleTodoFilter(state = 'SHOW_ALL', action) {
    // Somehow calculate it...
    return nextState
  }

  let todoApp = combineReducers({
    todos,
    visibleTodoFilter
  })
```

In addition to reading the state, container components can dispatch actions. In a similar fashion, you can define a function called mapDispatchToProps() that receives the dispatch() method and returns callback props that you want to inject into the presentational component. For example, we want the VisibleTodoList to inject a prop called onTodoClick into the TodoList component, and we want onTodoClick to dispatch a TOGGLE_TODO action:

Example:

```
  const mapDispatchToProps = dispatch => {
    return {
      onTodoClick: id => {
        dispatch(toggleTodo(id))
      }
    }
  }
```

Finally, we create the VisibleTodoList by calling connect() and passing these two functions:

Example:

```
  import { connect } from 'react-redux'

  const VisibleTodoList = connect(
```

```
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

These are the basics of the React Redux API, but there are a few shortcuts and power options so we encourage you to check out its documentation in detail. In case you are worried about **mapStateToProps** creating new objects too often, you might want to learn about computing derived data with reselect. https://redux.js.org/recipes/computing-derived-data

Find the rest of the container components defined below:

Example: containers/FilterLink.js

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

Example: containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
```

```
    switch (filter) {
      case 'SHOW_ALL':
        return todos
      case 'SHOW_COMPLETED':
        return todos.filter(t => t.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(t => !t.completed)
    }
  }

  const mapStateToProps = state => {
    return {
      todos: getVisibleTodos(state.todos, state.visibilityFilter)
    }
  }

  const mapDispatchToProps = dispatch => {
    return {
      onTodoClick: id => {
        dispatch(toggleTodo(id))
      }
    }
  }

  const VisibleTodoList = connect(
    mapStateToProps,
    mapDispatchToProps
  )(TodoList)

  export default VisibleTodoList
```

7.4.3. Implementing Other Components.

Recall as mentioned previously, both the presentation and logic for the **AddTodo** component are mixed into a single definition.

Example:

```
  import React from 'react'
  import { connect } from 'react-redux'
  import { addTodo } from '../actions'

  let AddTodo = ({ dispatch }) => {
    let input

    return (
      <div>
        <form
```

```
          onSubmit={e => {
            e.preventDefault()
            if (!input.value.trim()) {
              return
            }
            dispatch(addTodo(input.value))
            input.value = ''
          }}
        >
          <input
            ref={node => {
              input = node
            }}
          />
          <button type="submit">Add Todo</button>
        </form>
      </div>
    )
  }
  AddTodo = connect()(AddTodo)

  export default AddTodo
```

If you are unfamiliar with the **ref attribute**, please read this documentation to familiarize yourself with the recommended use of this attribute.

https://facebook.github.io/react/docs/refs-and-the-dom.html

7.4.5. Tying the containers together within a component

Example: components/App.js

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

7.4.6. Passing the Store

All container components need access to the Redux store so they can subscribe to it. One option would be to pass it as a prop to every container component. However it gets tedious, as you have to wire store even through presentational components just because they happen to render a container deep in the component tree.

The option we recommend is to use a special React Redux component called **<Provider>** to magically make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:

https://react-redux.js.org/api/provider

Example: index.js

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```