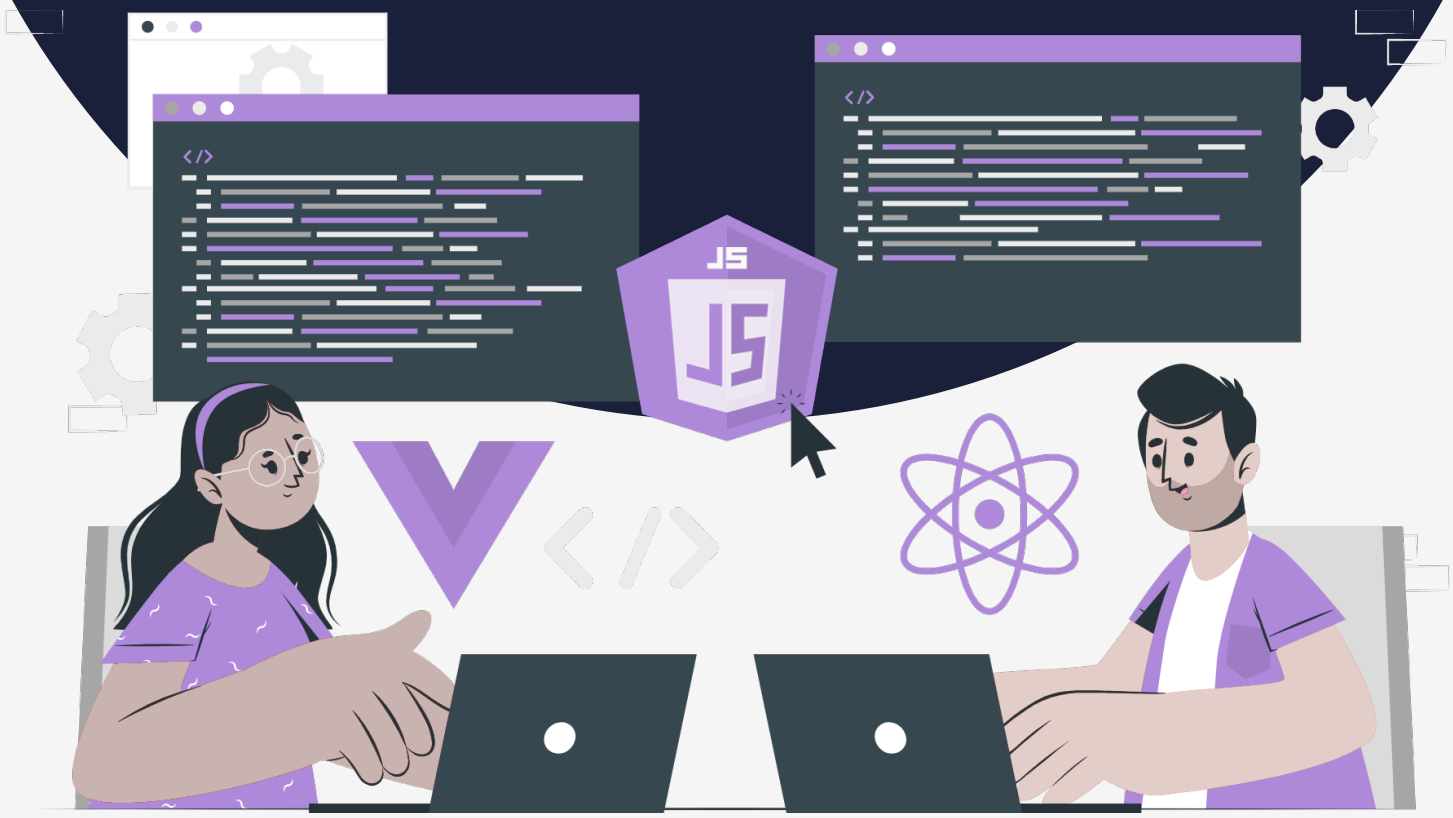
Lesson:

The Problem with Redux



PW SKILLS

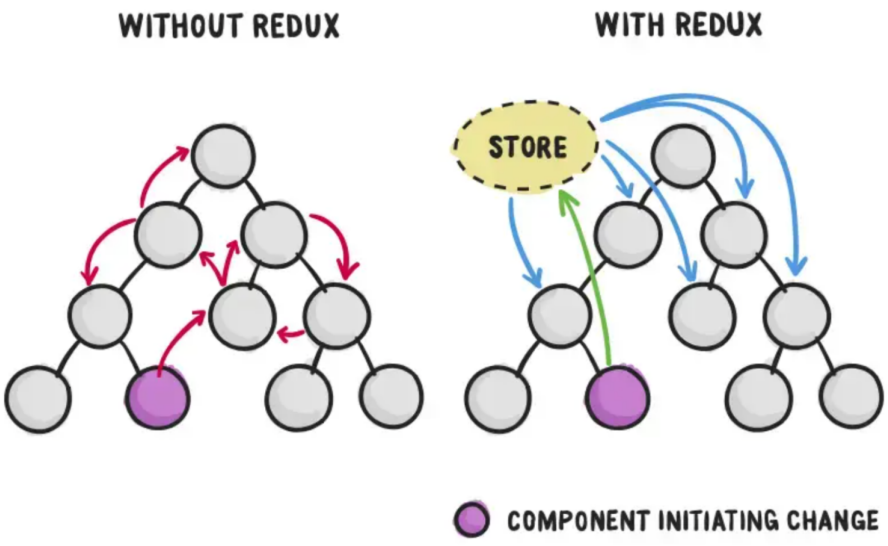
Topics Covered:

1. What is Redux?
2. How does Redux work?
3. The Problem with Redux

What is Redux?
Redux is a popular state management library. It first came to the frontend world in 2015 as the revolutionary state management solution built by Dan Abramov and Andrew Clark. Redux is a predictable state container for JavaScript apps.

In frontend frameworks like React, each component internally manages its own states. As the app gets more and more complex, managing states across many components becomes tedious, complicated, and difficult. Redux became the solution for this issue.

Redux works by providing a centralized and predictable ‘store’, single source of truth, which holds all the states within the app.The term "predictable" refers to the fact that the state of an application is stored and updated in a consistent and reliable way. This means that any changes to the application state can be tracked and predicted with a high degree of accuracy, allowing developers to easily reason about how the application behaves in response to different actions or events. Each component in the app can access this store without having to pass props around in the component tree.



Let's build a basic Todo app without using Redux and using Redux also.

Without Redux

JavaScript

```
// src/App.js

import { useState } from 'react';
function App() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = (todo) => {
    setTodos([...todos, todo]);
  };

  return (
    <div>
      <h1>Todo List</h1>
      <AddTodo onAddTodo={handleAddTodo} />
      <TodoList todos={todos} />
    </div>
  );
}

function AddTodo({ onAddTodo }) {
  const [text, setText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    onAddTodo({
      id: Date.now(),
      text,
      completed: false,
    });
    setText('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={text} onChange={(e) =>
setText(e.target.value)} />
      <button>Add Todo</button>
    </form>
  );
}


function TodoList({ todos }) {
  return (
    <ul>
      {todos.map((todo) => (
      <li key={todo.id}>
            <input type="checkbox" checked={todo.completed} />
            {todo.text}
      </li>
      ))}
    </ul>
  );
}

export default App;
```

In this code, we use the **useState** hook to manage the state of the todo list. We pass down a **handleAddTodo** function as a prop to the **AddTodo** component, which adds a new todo to the list when

the form is submitted. We pass down the todos array as a prop to the **TodoList** component, which renders each todo item in the list.

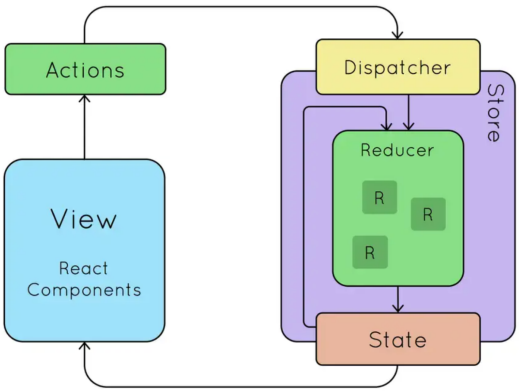
While this approach works fine for a small app like this, it can become unwieldy and difficult to manage as the app grows in complexity and more components need access to the state.

We will understand how we can make this app using Redux in next section.

How does Redux work?

The typical Redux flow is:

- A user interacts with the View (component) and triggers a state update .
- When a state update is required, the View dispatches an action.
- The reducers receive the action from the dispatch and update the state in the Store according to what is described by the action .
- The View is subscribed to the Store to listen for state changes. The changes are notified via the subscription methods and the View updates its UI accordingly.



The key concepts of Redux are:

Store: A single source of truth that holds the state of an application.
Actions: Plain JavaScript objects that represent changes to the state of the application.
Reducers: Functions that handle actions and return a new state of the application.
Selectors: Functions that extract specific pieces of data from the state.

The core concept of Redux is the store, which is a single object that holds the state of the entire application. The store is responsible for managing the state of the application and dispatching actions to update that state

An **action** is a plain JavaScript object that describes a change to the state of the application. **Actions** are typically triggered by user events, such as button clicks or form submissions.

Reducers are functions that take the current state of the application and an action as input, and return a new state as output. Reducers are responsible for handling actions and updating the state of the application accordingly. They are pure functions, meaning that they do not have any side effects and always return the same output given the same input.

Selectors are functions that extract specific pieces of data from the state of the application. They are used to make it easier to access and work with data from the store. Selectors can be used to compute derived data, transform data, or filter data.

In Redux, the flow of data is unidirectional. Components dispatch actions to the store, which triggers reducers to update the state of the application. Once the state is updated, the store notifies all subscribed components of the change, and they can re-render with the new state. This predictable data flow makes it easier to reason about how changes to the state of the application will affect the rest of the application.

Redux can be used in a variety of different applications, but it is especially useful for large-scale applications with complex data flows. By providing a predictable, centralized way to manage the state of an application, Redux makes it easier to build and maintain complex applications over time. However, because it requires a certain amount of boilerplate and can be more complex than other state management solutions, it may not be the best choice for smaller, simpler applications.

Let's understand same todo example with Redux now,

First, we need to create a **Redux store** to hold the state of the application. We can do this using the createStore function from the **redux library**:

```javascript
// todoReducer.js

import { createStore } from 'redux';

const initialState = {
  todos: [],
};

function reducer(state = initialState, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        todos: [...state.todos, { text: action.payload,
completed: false }],
      };
    case 'TOGGLE_TODO':
      return {
        todos: state.todos.map((todo) =>
          todo.text === action.payload
            ? { ...todo, completed: !todo.completed }
            : todo
        ),
      };
    default:
      return state;
  }
}

const store = createStore(reducer);

default export store;
```

In this code, we define an initial state for the store with an empty todos array. We also define a reducer function that takes the current state and an action as input, and returns a new state based on the action. The reducer handles three types of actions: **ADD_TODO and TOGGLE_TODO.**

Next, we can create some **action creators** to dispatch actions to the store. Action creators are functions that return an action object with a type and payload(a payload refers to the data that is passed along with an action):

```javascript
// todoActions.js

function addTodo(text) {
  return { type: 'ADD_TODO', payload: text };
}

function toggleTodo(text) {
  return { type: 'TOGGLE_TODO', payload: text };
}
```

Next, we can create some action creators to dispatch actions to the store. Action creators are functions that return an action object with a type and payload(a payload refers to the data that is passed along with an action):
These action creators can be used to dispatch actions to the store. For example, to add a new todo item to the list, we can **dispatch** (returned from **useDispatch** hook) the **addTodo** action with the text of the new item:

```javascript
dispatch(addTodo('Buy Web Dev course'));
```

Lets implement **TodoList** and **AddTodo** Components, which will update the store todos using useDispatch().

```javascript
// TodoList.js

import React from 'react';
import { toggleTodo } from './todoActions.js';
import { useDispatch } from 'react-redux'
const TodoList = ({ todos }) => {
```

```
      const dispatch = useDispatch();

      return (
        <ul>
            {todos.map(todo => (
            <li
                  key={todo.id}
                  onClick={() => dispatch(toggleTodo(todo.id))}
                  style={{ textDecoration: todo.completed ?
            'line-through' : 'none' }}
            >
            {todo.text}
            </li>
            ))}
        </ul>
      );
}
default export TodoList;
```

JavaScript

```
// AddTodo.js

import React, { useState } from 'react';
import { addTodo } from './todoActions.js';
import { useDispatch } from 'react-redux'

const AddTodo = ({ addTodo }) => {
  const [inputValue, setInputValue] = useState('');
  const dispatch = useDispatch();

  const handleSubmit = e => {
      e.preventDefault();
      if (inputValue.trim()) {
            dispatch(addTodo(inputValue));
            setInputValue('');
      }
  };
```

Finally, we need to provide our store to all Components by wrapping them in a Provider component and passing the store as a prop:

```javascript
// index.js
import store from 'todoReducer.js';

function App() {
  return (
      <Provider store={store}>
            <App />
      </Provider>
  );
}
```

The problem with Redux?

Redux is a popular choice for state management. Its pattern makes states predictable, as reducers are pure functions, which means the same state and actions passed will always result in the same output.

It is also easily maintainable and scalable due to the strict organization on how each part in the Redux flow should behave and work.

React and Redux are believed to be the best combo for managing states in large-scale React applications. However, with time, the popularity of Redux has fallen due to the following reasons:

- Configuring a Redux store is not simple.
- Several packages are needed to get Redux to work with React.
- Redux requires too much boilerplate code.
- Writing actions and reducers becomes more complex and cumbersome in huge applications.

To overcome the challenges Redux had, the Redux team came up with Redux Toolkit, the official recommended approach for writing Redux logic.