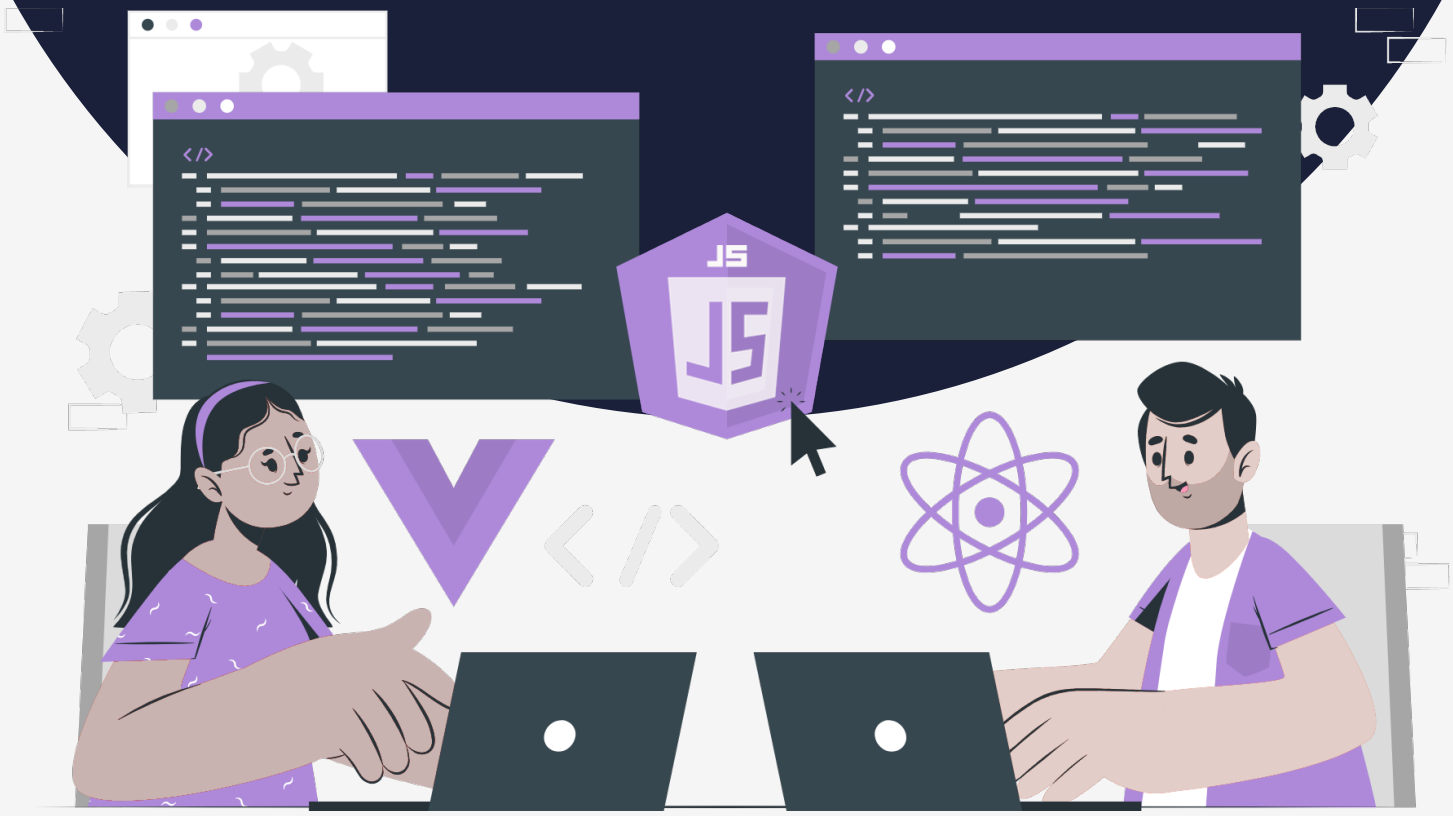
Lesson:

Redux Toolkit – The New Way of Writing Redux



PW SKILLS

Topics Covered:

1. What is Redux Toolkit
2. How to use Redux Toolkit
3. Example

What is Redux Toolkit?
To overcome the challenges Redux had, the Redux team came up with Redux Toolkit, the official recommended approach for writing Redux logic. It aims to speed up Redux development by including Redux Core with the packages that they think are essential for building a Redux app. It is an opinionated derivative of Redux, with many best-practice configurations for Redux beginners or developers who want simple, fast, and clean Redux code.

Redux Toolkit (previously known as Redux Starter Kit) provides some options to configure the **global store** and create both **actions** and **reducers** more streamlined by abstracting the **Redux API** as much as possible.

- In previous versions of Redux a lot of boilerplate code was required to do simple things.
- The other biggest gripe with Redux development was the need to install a lot of packages depending on what you want to do.Inside the Toolkit, you will have everything you need.
- As in the first point, configuring the store was complicated and with a lot of boilerplate code. Now, the process is an abstraction and they already have a lot of configuration done for us.
- Redux Toolkit is an abstraction and an updated version of the common Redux that tries to standardise the way of managing the state with Redux.

Install Redux Toolkit and React-Redux

Add the Redux Toolkit and React-Redux packages to your project:

```
npm install @reduxjs/toolkit react-redux
```

Create a Redux Store

We can create a Redux store using the configureStore function from Redux Toolkit. Import the configureStore API from Redux Toolkit

```javascript
import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({
  reducer: {},
})
```

Provide the Redux Store to React

Once the store is created, we can make it available to our React components by putting a React-Redux <Provider> around our application in src/index.js. Import the Redux store we just created, put a <Provider> around your <App>, and pass the store as a prop:

```
JavaScript
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'


import { store } from './app/store'
import { Provider } from 'react-redux'

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

Create a Redux State Slice

Add a new file named counterSlice.js. In that file, import the **createSlice** API from Redux Toolkit.

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.

```
JavaScript
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in
reducers. It
      // doesn't actually mutate the state because it uses the
Immer library,
```

```
      // which detects changes to a "draft state" and produces
a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer
function
export const { increment, decrement, incrementByAmount } =
counterSlice.actions

export default counterSlice.reducer
```

Add Slice Reducers to the Store

Next, we need to import the reducer function from the counter slice and add it to our store. By defining a field inside the reducer parameter, we tell the store to use this slice reducer function to handle all updates to that state.

```
JavaScript
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

Use Redux State and Actions in React Components

Now we can use the React-Redux hooks to let React components interact with the Redux store. We can read data from the store with useSelector, and dispatch actions using useDispatch. Create a Counter.js file with a <Counter> component inside, then import that component into App.js and render it inside of <App>.

```javascript
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>


      </div>
    </div>
  )
}
```

Now, any time We click the "Increment" and "Decrement" buttons:

- The corresponding Redux action will be dispatched to the store
- The counter slice reducer will see the actions and update its state
- The <Counter> component will see the new state value from the store and re-render itself with the new data

Let's take the same example of a todo list application and use Redux Toolkit to manage the state .

First, we need to create a slice using the **createSlice** function from Redux Toolkit. A **slice** is a collection of **reducer** logic and **actions** for a specific feature of the application. In this case, we'll create a slice for the todo list:

```javascript
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  todos: [],
};

const todoSlice = createSlice({
  name: 'todo',
  initialState,
  reducers: {
    addTodo: (state, action) => {
```

```
      state.todos.push({ text: action.payload, completed:
false });
    },
    removeTodo: (state, action) => {
      state.todos = state.todos.filter((todo) => todo.text !==
action.payload);
    },
    toggleTodo: (state, action) => {
      const todo = state.todos.find((todo) => todo.text ===
action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
  },
});

export const { addTodo, removeTodo, toggleTodo } =
todoSlice.actions;
export default todoSlice.reducer;
```

In this code, we define an initial state for the slice with an empty todos array. We also define three reducers for adding, removing, and toggling todo items.

Next, we can create a Redux store using the **configureStore** function from Redux Toolkit.

JavaScript

```
import { configureStore } from '@reduxjs/toolkit';
import todoReducer from './todoSlice';

const store = configureStore({
  reducer: {
    todo: todoReducer,


  },
});
```

In this code, we pass our todoReducer to the configureStore function, and give it a key of todo in the root reducer.

Finally, we can connect our React components to the Redux store using the **useSelector** and **useDispatch** hooks from the react-redux library. This allows our components to access the state of the store and dispatch actions to update the state:

```javascript
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, removeTodo, toggleTodo } from './todoSlice';

function TodoList() {
  const [newTodo, setNewTodo] = useState('');
  const todos = useSelector((state) => state.todo.todos);
  const dispatch = useDispatch();

  const handleSubmit = (event) => {
    event.preventDefault();
    if (newTodo.trim()) {
      dispatch(addTodo(newTodo));
      setNewTodo('');
    }
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={newTodo}
          onChange={(event) => setNewTodo(event.target.value)}
        />

        <button type="submit">Add Todo</button>
      </form>
      <ul>
        {todos.map((todo) => (
          <li key={todo.text}>
            {todo.text}{' '}
            <button onClick={() =>
dispatch(removeTodo(todo.text))}>Remove</button>{' '}
            <button onClick={() =>
dispatch(toggleTodo(todo.text))}>
              {todo.completed ? 'Mark Incomplete' : 'Mark
Complete'}
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

In this code, we use the useSelector hook to access the todos array from the todo slice of the store. We also use the useDispatch hook to get a reference to the dispatch function.