Immer JS

- Immer JS is a JavaScript library that simplifies handling immutable data structures in applications.
- It helps developers work with immutable states by providing a way to update them more easily.
- Immer creates "draft" copies of your state, allows you to modify them as if they were mutable, and then produces a new immutable state based on the changes.
- Immer JS is a powerful tool that abstracts the complexity of managing immutable data.
- It's especially useful in applications where immutability is required, such as in React or Redux-based applications.
- By using Immer, you can write cleaner, more maintainable code while ensuring that your state remains immutable.

Why is Learning Immer JS Important?

- **Ease of Use:** It allows developers to write simpler and more readable code while maintaining immutability.
- **Immutable State Management:** In modern frontend frameworks like React, maintaining immutability is important for performance optimizations (e.g., to avoid unnecessary re-renders).
- **Cleaner Code:** Without Immer, you often have to write complex logic to deeply clone and update nested objects or arrays, which can make code harder to understand and maintain.

Problems Solved by Immer JS

- **Immutability Without Deep Cloning:** In vanilla JavaScript, updating deeply nested objects requires complex deep copying. Immer handles this automatically, which makes working with immutable data structures easier.
- **Simplifies Redux Reducers:** When working with Redux for state management, reducers need to return a new state. Immer simplifies writing these reducers by making the immutable state updates feel like mutable ones.

Vanilla JavaScript

- Vanilla JavaScript refers to plain, standard JavaScript without any additional libraries or frameworks (like React, Vue, or jQuery).
- It uses the core features of JavaScript as defined by ECMAScript (ES) standards to manipulate the DOM, handle events, and build web applications.
- Using vanilla JavaScript, you interact directly with the browser's built-in APIs (such as the DOM API) to create dynamic web pages.

Example Without Immer JS (Vanilla JavaScript)

Suppose we have a nested object representing a user, and we want to update the user's address.

index.html

```
<script src="index.js"> </script>
```

index.js

```
const user = {
                    name: "Neha Rathore",
                    address: {
                      city: "Ghaziabad",
                      state: "Uttar Pradesh"
                    }
                };

// To update the city, you would have to deep copy the object
const updatedUser = {
  ...user,
  address: {
    ...user.address,
    city: "Noida"
  }
};

console.log(updatedUser);
```

Example Using Immer JS

Install parcel and Immer

```
npm init -y
npm install --save-dev parcel
npm install immer
```

index.html

```
<script src="index.js" type="module"> </script>
```

index.js

```
import {produce} from 'immer';

const user = {
                    name: "Neha Rathore",
                    address: {
                     city: "Ghaziabad",
                     state: "Uttar Pradesh"
                    }
                };

// Use Immer to update the city
const updatedUser = produce(user, draft => {
                draft.address.city = "NOIDA";
});

console.log(updatedUser);
```

In the above code, Immer makes the process of updating a deeply nested object simpler. You can directly "mutate" the draft object as if it were mutable, but Immer ensures that the original object stays unchanged and a new immutable object is returned.

How Immer Works

- **Draft State:** When using produce(), Immer creates a draft of the current state. This draft is a proxy of the original state, allowing you to "mutate" it.
- **Detect Changes:** Immer tracks changes to the draft. When the produce function finishes, Immer produces a new immutable state with only the changes applied.
- **Optimized Cloning:** Immer only clones the parts of the object that have changed. This makes it efficient in terms of memory and performance, especially with large and deeply nested data structures.

When to Use Immer

- **State Management in React:** If you're using React and want to manage state immutably without writing complex update logic, Immer is extremely useful.
- **Redux Reducers:** Writing reducers in Redux often involves updating state immutably, and Immer can drastically reduce boilerplate code.

Example : Redux Reducer Without Immer

```
const initialState = {
  user: {
    name: "Neha Rathore",
    address: { city: "Ghaziabad", state:"Uttar Pradesh" }
  }
};
function userReducer(state = initialState, action) {
  switch (action.type) {
    case "UPDATE_CITY":
      return {
                ...state,
                    user: {
                                ...state.user,
                               address: {
                                                ...state.user.address,
                                                city: action.payload          }
                                }
      };
    default:       return state;
  }}
```

Example: Redux Reducer With Immer

```
import produce from 'immer';

const initialState = {
  user: {
    name: "Neha Rathore",
    address: { city: "Ghaziabad", state:"Uttar Pradesh" }
  }
};

function userReducer(state = initialState, action) {
        switch (action.type) {
                case "UPDATE_CITY":
                                return produce(state, draft => {
                                draft.user.address.city = action.payload;
                                });
                default:
                                return state;
        }
}
```

With Immer, the reducer logic becomes simpler and more intuitive without worrying about deeply copying and spreading the nested state.

Redux Toolkit

- Redux Toolkit (RTK) is the official, recommended way to write Redux logic.
- It is introduced to simplify Redux's setup and code structure, solving common issues developers face with vanilla Redux, such as excessive boilerplate and complex configuration.

Problems Solved by Redux Toolkit

- **Boilerplate Reduction:** Redux has a lot of setup, including writing action types, action creators, reducers, and handling immutability. RTK simplifies this with utility functions.
- **Immutability Handling:** RTK uses Immer.js internally, allowing developers to write "mutative" logic inside reducers while keeping the state immutable.
- **Simplified Store Configuration:** RTK provides configureStore() to set up the Redux store with sensible defaults (e.g., adding middleware like redux-thunk).
- **Developer Experience:** RTK integrates tools like the Redux DevTools and middleware by default, improving the development process.
- **Enhanced Readability:** Code written with RTK is generally cleaner and easier to maintain, reducing the cognitive load on developers.

Why Redux Toolkit is Popular Compared to Redux

- **Simpler Code:** RTK drastically reduces boilerplate and simplifies the Redux code structure.
- **Opinionated Structure:** It enforces a consistent project structure, making it easier for developers to collaborate and follow best practices.
- **Built-in Middleware:** It includes useful middleware (like redux-thunk) for handling async logic by default, reducing configuration overhead.
- **Integrated DevTools:** Redux DevTools are integrated with minimal setup, making debugging easier.

Core Concepts of Redux Toolkit

- **Slices:** A "slice" is a collection of reducer logic and actions for a specific feature of the application. It simplifies the combination of action types and reducers.
- **createSlice():** This is a key function that creates a slice, which includes actions and reducers in a single file.
- **configureStore():** It simplifies the setup of the Redux store, automatically adding redux-thunk middleware and enabling Redux DevTools.
- **createAsyncThunk():** It simplifies handling async actions (e.g., fetching data from an API), eliminating the need to manually write complex async logic.
- **createReducer():** It allows you to write reducers that can directly "mutate" the state using Immer, removing the need for writing immutability logic manually.

Key Use Cases

- **State Management:** Managing application state across large, complex applications with predictable behavior.
- **Asynchronous Logic:** Using createAsyncThunk to handle async operations like API calls (fetching, posting data).
- **Boilerplate Reduction:** Simplifies the amount of code and configuration needed, making Redux easier to adopt in small to medium-sized apps.
- **Data Fetching:** Using built-in thunks for making API calls and handling loading states efficiently.

Example : perform increment and decrement operations using Redux Toolkit with HTML (without React)

Step 1: Install parcel and Redux Toolkit

```
npm init -y
npm install @reduxjs/toolkit parcel
```

Step 2: Create the HTML File (index.html)

Create HTML file for incrementing and decrementing the counter and a display area for the counter value.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Redux Toolkit Test</title>
</head>
<body>
        <h1>Counter: <span id="counter-value">0</span></h1>
        <button id="increment-btn">Increment</button>
        <button id="decrement-btn">Decrement</button>
        <script type="module" src="main.js"></script>
</body>
</html>
```

Step 3: Create the Slice (counterSlice.js)

This file defines the Redux slice for managing the counter state and its actions (increment and decrement).

counterSlice.js

```
import { createSlice } from '@reduxjs/toolkit';
const initialState = { value: 0 };
const counterSlice = createSlice({
                        name: 'counter',
                        initialState,
                        reducers: {
                                increment: state => { state.value += 1; },
```

```
                                        decrement: state => { state.value -= 1; }
                                    }
});
export const { increment, decrement } = counterSlice.actions;
export const counterReducer = counterSlice.reducer;
```

Step 4: Configure the Store (store.js)

This file sets up the Redux store and adds the counterReducer to it.

store.js

```
import { configureStore } from '@reduxjs/toolkit';
import { counterReducer } from './counterSlice.js';
export const store = configureStore({
                                    reducer: { counter: counterReducer }       });
```

Step 5: Write the Logic to Connect UI and Redux (main.js)

This file listens to the button clicks and dispatches Redux actions to update the counter state. It also updates the counter display in the HTML.

main.js

```
import { store } from './store.js';
import { increment, decrement } from './counterSlice.js';
const counterValueElement = document.getElementById('counter-value');
const incrementBtn = document.getElementById('increment-btn');
const decrementBtn = document.getElementById('decrement-btn');
function render() {
                        const state = store.getState();
                        counterValueElement.textContent = state.counter.value;
}
store.subscribe(render);
render();
incrementBtn.addEventListener('click', () => store.dispatch(increment()));
decrementBtn.addEventListener('click', () => store.dispatch(decrement()));
```

Step 6: In package.json, add a start script

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "parcel build index.html",
    "start": "parcel index.html"  }
```

Example: To-do list application using Redux Toolkit

Step 1: Install parcel and Redux Toolkit

```
npm init -y
npm install @reduxjs/toolkit parcel
```

Step 2: Create the HTML File (index.html)

Create an HTML file to set up the structure of the to-do list.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Todo List with Redux Toolkit</title>
  <style>
        body {
                font-family: Arial, sans-serif;
                margin: 0;
                padding: 20px;
                max-width: 600px;
                margin: auto;
        }
        ul {
                list-style-type: none;
                padding: 0;
        }
        li {
                display: flex;
                align-items: center;
                margin-bottom: 10px;
        }
        .completed {
                text-decoration: line-through;
                color: gray;
        }
```

```
        button {
                                margin-left: 10px;
          }
  </style>
</head>
<body>
  <h1>Todo List</h1>
  <input type="text" id="new-todo" placeholder="Add a new todo" />
  <button id="add-todo-btn">Add Todo</button>
  <ul id="todo-list"></ul>
  <script type="module" src="main.js"></script>
</body>
</html>
```

Step 3: Define the Redux slice for managing to-dos

todoSlice.js

```
import { createSlice } from '@reduxjs/toolkit';
const todoSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
        addTodo: (state, action) => {
          state.push({ id: Date.now(), text: action.payload, completed: false });
        },
        toggleTodo: (state, action) => {
          const todo = state.find(todo => todo.id === action.payload);
          if (todo) {
            todo.completed = !todo.completed;
          }
        },
        removeTodo: (state, action) => {
          return state.filter(todo => todo.id !== action.payload);
        }
  }
});
export const { addTodo, toggleTodo, removeTodo } = todoSlice.actions;
export const todoReducer = todoSlice.reducer;
```

Step 4: Configure the Redux store.

store.js

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

export const store = configureStore({
  reducer: {
    todos: todoReducer
  }
});
```

Step 5: Handle UI interactions and connect to the Redux store.

main.js

```
import { store } from './store.js';
import { addTodo, toggleTodo, removeTodo } from './todoSlice.js';

const newTodoInput = document.getElementById('new-todo');
const addTodoBtn = document.getElementById('add-todo-btn');
const todoList = document.getElementById('todo-list');

// Render the todo list
function render() {
  const todos = store.getState().todos;
  todoList.innerHTML = '';
  todos.forEach(todo => {
    const li = document.createElement('li');
    li.className = todo.completed ? 'completed' : '';
    li.innerHTML = `
      <span>${todo.text}</span>
      <button onclick="toggle(${todo.id})">${todo.completed ? 'Undo' : 'Complete'}</button>
      <button onclick="remove(${todo.id})">Remove</button>
    `;
    todoList.appendChild(li);
  });
}
```

```
// Add a new todo
addTodoBtn.addEventListener('click', () => {
  const text = newTodoInput.value.trim();
  if (text) {
    store.dispatch(addTodo(text));
    newTodoInput.value = '';
  }
});

// Toggle todo status
window.toggle = (id) => {
  store.dispatch(toggleTodo(id));
};

// Remove a todo
window.remove = (id) => {
  store.dispatch(removeTodo(id));
};

// Subscribe to store changes and render the list
store.subscribe(render);
render();
```

Step 6: In package.json, add a start script

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "parcel build index.html",
    "start": "parcel index.html"
}
```