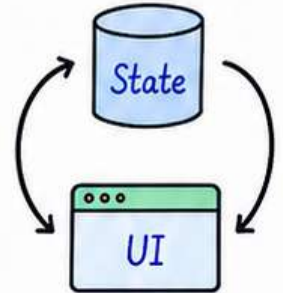


Chapter 11: State Management

11.1 State Management Concepts

11.1.1 What is State?

- * State is an object that holds **information** which can **change over time**.
- * When the state changes, the **UI** is updated to reflect those changes.
- * State allows a component to **remember data** between re-renders.
- * State management helps in **sharing** and **syncing data** across multiple components.



Key Points:

- ✓ State represents the current condition or data of an application.
- ✓ State can be updated and UI will re-render automatically.
- ✓ Effective state management improves performance and maintainability.
- ✓ Good state management avoids unnecessary re-renders and bugs.

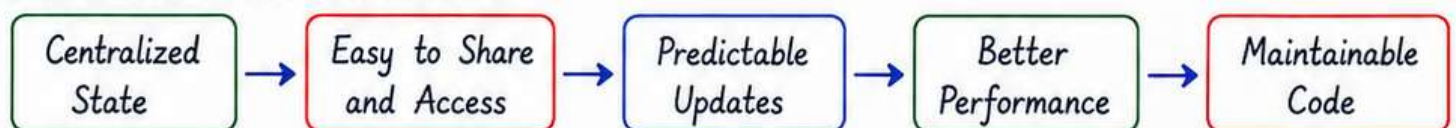


11.1.2 Why is State Management Important?

- * In small applications, managing state in a single component is easy.
- * In large applications, state is shared across many components.
- * Without proper state management, it becomes **hard to track, update and maintain data**.
- * State management provides a **predictable** way to manage application data.



How State Management Helps?



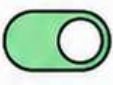
Examples of State:



User
Information



Cart Items in
E-commerce



Theme
(Light/Dark)



Messages in
Chat App



Filters in
Dashboard



Pagination
Details

Key Takeaway:

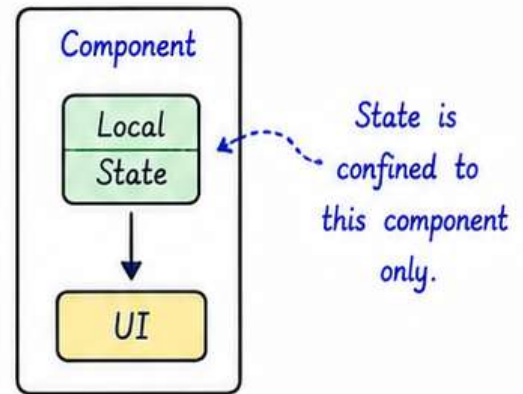


State management is crucial for building **dynamic, scalable** and **maintainable** React applications.

11.2 Local State

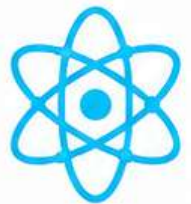
11.2.1 What is Local State?

- * Local state is the state that is managed **within a single component**.
- * It is **private** to that component and cannot be accessed by other components.
- * Local state is created using the **useState()** Hook in React.
- * When the local state changes, only **that component** re-renders.



11.2.2 useState() Hook

- * **useState()** is a built-in Hook that allows us to add state to functional components.
- * It returns an array with two values: the **current state** and a **function** to update it.
- * Syntax:



```
const [state, setState] = useState(initialValue);
```

↑ ↑ ↑
Current state value Function to update Initial value of
 the state the state

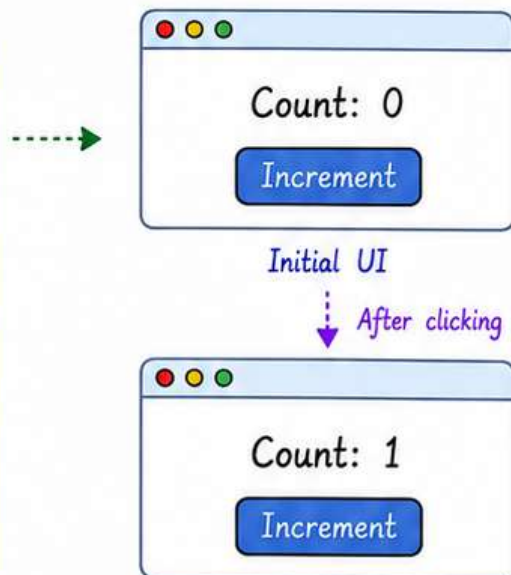
Key Points:

- ✓ Local state is simple and easy to use.
- ✓ It is suitable for managing component-specific data.
- ✓ It does not require any external library.
- ✓ Updates to local state are asynchronous.



11.2.3 Example: Counter with Local State

```
import { useState } from 'react';  
function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <div>  
      <h2>Count: {count}</h2>  
      <button onClick={() => setCount(count + 1)}>  
        Increment  
      </button>  
    </div>  
  );  
}
```



Key Takeaway:

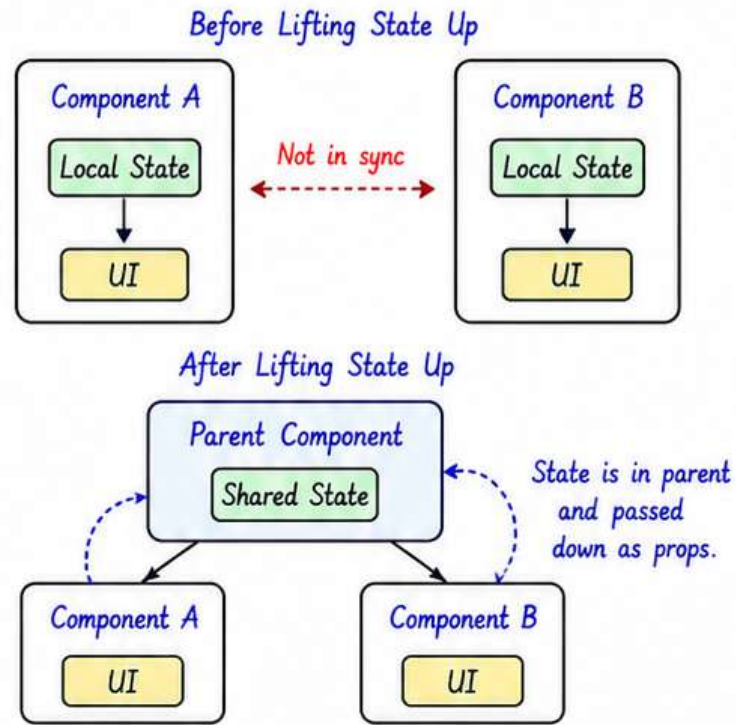


Local state is perfect for managing data that belongs to a **single component**. It keeps components **simple, modular** and **easy to maintain**.

11.3 Lifting State Up

11.3.1 What is Lifting State Up?

- * Sometimes, multiple components need to **share the same state**.
- * Lifting state up means moving the state to their **closest common parent component**.
- * The parent manages the state and passes it down as **props** to child components.
- * This keeps the data **consistent** across components.



11.3.2 How Lifting State Up Works?

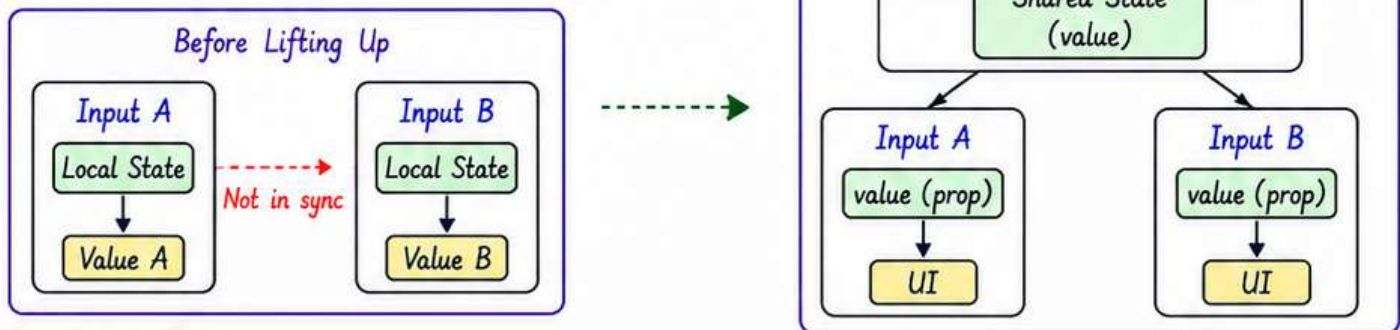
Steps to lift state up:

1. Identify the state that is shared.
2. Move the state to their closest common parent.
3. Pass the state down to child components as props.
4. Update the state in the parent using setter function.
5. Parent re-renders and passes updated state to children.

11.3.3 Example: Two Inputs Sharing State

Problem:

Two input fields need to show and update the same value.



Key Points:

- ✓ Lift state up when multiple components need the **same data**.
- ✓ The common parent becomes the **single source of truth**.
- ✓ It helps in keeping the UI **in sync**.
- ✓ Avoid lifting state up **unnecessarily**; keep state as local as possible.
- ✓ It improves data flow and makes the application **easier to maintain**.



Key Takeaway:



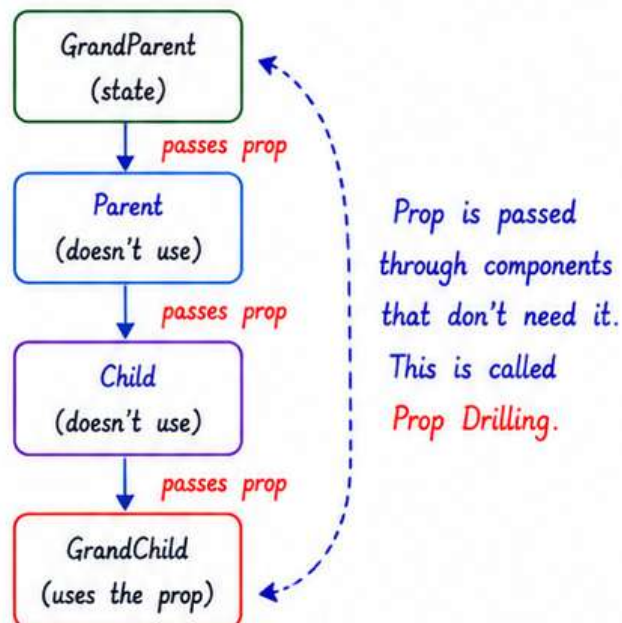
Lifting state up helps components **share** and **stay in sync** by moving the state to their closest common parent and passing it down as **props**.

11.4 Prop Drilling

11.4.1 What is Prop Drilling?

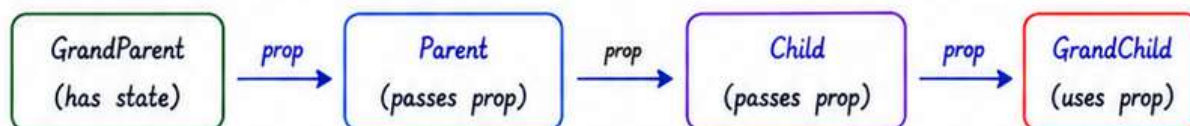
- * Prop drilling is the process of passing data from a **parent** component down through multiple levels of components to a **deeply nested child component**.
- * Intermediate components that **do not need** the data still have to pass it down.
- * This can make the code **harder to read, maintain** and manage.
- * It is a **common problem** in large React applications.

Example of Prop Drilling



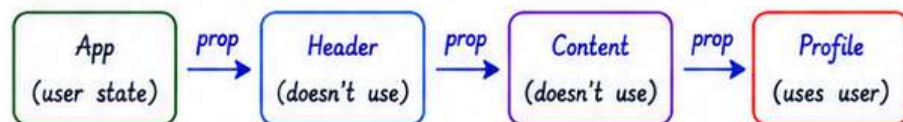
11.4.2 How Prop Drilling Works?

The data is passed down as props from one component to another.

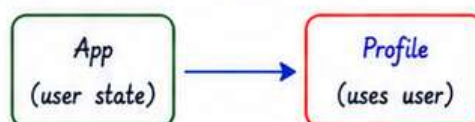


11.4.3 Example: User Data Prop Drilling

Before (Prop Drilling)



After (Without Prop Drilling)



11.4.4 Problems with Prop Drilling

- * Makes components **tightly coupled**.
- * **Hard to manage** in large applications.
- * Adding new props requires passing them through **many components**.
- * Makes the code **messy** and **difficult to debug**.



Key Points:

- ✓ Prop drilling is passing props through many components that don't need them.
- ✓ It makes the code **hard to read, maintain** and debug.
- ✓ Use it in small apps, but **avoid** in large apps.
- ✓ Use **Context API, Redux** or other state management solutions to avoid prop drilling.



Key Takeaway:

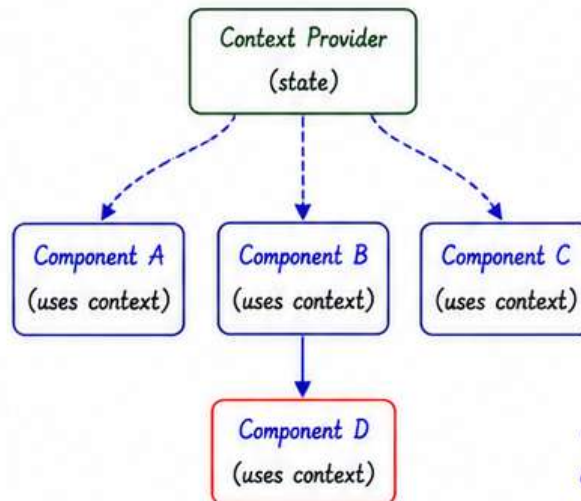


Prop drilling works but is not ideal for big applications. It should be **avoided** and replaced with **better state management** techniques.

11.5 Context API

11.5.1 What is Context API?

- * Context API is a built-in feature in React for **sharing data** across the component tree **without prop drilling**.
- * It provides a way to **pass data** through the tree without passing props down manually at every level.
- * It is useful when many components need the **same data**.
- * Context API makes the code **cleaner, easier** to maintain and **scalable**.



Context API allows components to access data directly without passing props.

11.5.2 How Context API Works?

Steps:

1. Create a Context using `createContext()`.

2. Wrap the components with a Provider.

3. Pass the value to the Provider.

4. Consume the context using `useContext()`.

5. Use the data in any component inside the Provider.

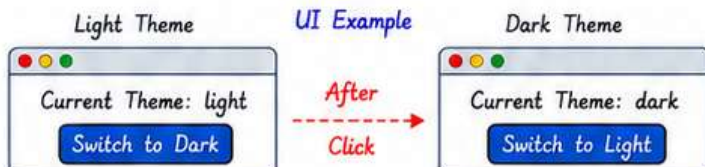
11.5.3 Example: Theme using Context API

```
// 1. Create Context
import { createContext, useContext } from 'react';
const ThemeContext = createContext();

// 2. Provider Component
export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

```
// 3. Consumer Hook
export function useTheme() {
  return useContext(ThemeContext);
}
```

```
// 4. Using in any Component
function Button() {
  const { theme, setTheme } = useTheme();
  return (
    <button onClick={() =>
      setTheme(theme === 'light' ? 'dark' : 'light')}
    >
      Current Theme: {theme}
    </button>
  );
}
```



Key Points:

- ✓ Context API is used to share data across the component tree.
- ✓ It helps to **avoid prop drilling**.
- ✓ It makes the code **clean, maintainable** and **easy to manage**.
- ✓ Use Context API when many components need the **same data**.



Key Takeaway:

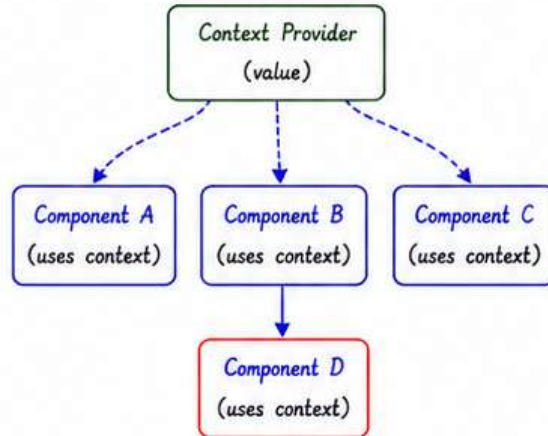


Context API is a powerful way to manage global or shared state in React applications. It **simplifies** data sharing and helps build **scalable** and **maintainable** applications.

11.6 useContext()

11.6.1 What is useContext()?

- * `useContext()` is a React Hook used to **consume values** from the Context.
- * It allows any **functional component** to access context data directly.
- * It **eliminates** the need to pass props through **multiple levels**.
- * It works **only** with components wrapped inside the corresponding **Provider**.



`useContext()` allows components to access the context value directly.

11.6.2 Syntax

```
const value = useContext(MyContext);
```

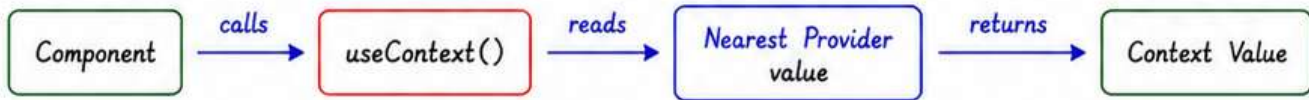
↑
Variable to store the context value

↑
React Hook

↑
The context to consume

11.6.3 How useContext() Works?

`useContext()` reads the current value of the **nearest matching context Provider** above it in the component tree.



11.6.4 Example: Using useContext()

```
// 1. Create Context
import { createContext } from 'react';
export const ThemeContext = createContext();
```

```
// 2. Provider Component
export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

```
// 3. Using useContext() in a Component
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function DisplayTheme() {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
    <div>
      <p>Current Theme: {theme}</p>
      <button onClick={() =>
        setTheme(theme === 'light' ? 'dark' : 'light')}>
        Toggle Theme
      </button>
    </div>
  );
}
```



Key Points:

- ✓ `useContext()` is a Hook used to access context data.
- ✓ It returns the current value from the **nearest Provider**.
- ✓ It avoids **prop drilling** and makes the code **cleaner**.
- ✓ It can be used **only** inside **functional components**.



Key Takeaway:



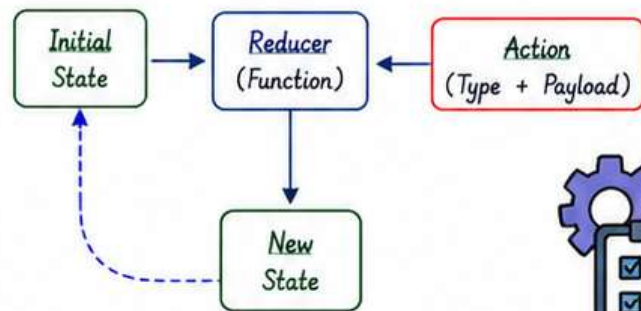
`useContext()` makes it easy to access shared data from the context. It helps in building **scalable**, **maintainable** and **clean** React applications.

11.7 useReducer()

11.7.1 What is useReducer()?

- * `useReducer()` is a React Hook used for managing **complex** state logic.
- * It is an alternative to `useState()` when the state has **multiple sub-values** or depends on the **previous state**.
- * It follows the **reducer pattern** similar to Redux.
- * It makes state logic **predictable** and **easier** to manage.
- * It returns the current state and a **dispatch** function to update the state.

How useReducer() Works



New state becomes the current state.



11.7.2 Syntax

```
const [state, dispatch] = useReducer(reducer, initialState, init?);
```

↑ ↑ ↑ ↑ ↑
Current state Function to send action Reducer function Initial state Optional initializer function

11.7.3 How useReducer() Works?

1. Component calls `dispatch()` with an action.
2. The **action** is sent to the **reducer** function.
3. Reducer calculates the **new state** based on **current state** and **action**.
4. The **new state** is returned to `useReducer()`.
5. Component re-renders with the **updated state**.

11.7.4 Example: Counter using useReducer()

```
// 1. Reducer Function
function counterReducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```



```
// 2. Using useReducer in Component
import { useReducer } from 'react';
function Counter() {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });
  return (
    <div>
      <h2>Count: {state.count}</h2>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>
        Decrement
      </button>
    </div>
  );
}
```



Key Points:

- ✓ `useReducer()` is useful for managing **complex state logic**.
- ✓ It uses a **reducer function** to calculate the new state based on **actions**.
- ✓ It returns the **current state** and **dispatch** function.
- ✓ It is **predictable**, easy to test and maintain.
- ✓ It is similar to **Redux** but built-in in React.



Key Takeaway:



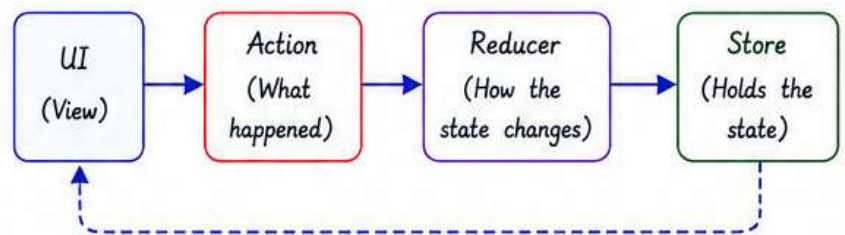
`useReducer()` is the right choice when state logic is **complex** or depends on the **previous state**. It helps write **clean**, **scalable** and **maintainable** React applications.

11.8 Redux Introduction

11.8.1 What is Redux?

- * Redux is a **predictable** state container for JavaScript applications.
- * It helps manage the application state in a **centralized store**.
- * It follows a strict architecture with **unidirectional data flow**.
- * Redux is not limited to React. It can be used with any JavaScript framework.
- * Redux makes large applications **easier to develop, maintain and test**.

Redux Architecture



The Store holds the entire state of the application.
When the state changes, the UI is updated.

Why use Redux?



Centralized State

All state is stored in one place (Store).



Predictable

State changes are predictable and easy to debug.



Maintainable

Easier to maintain and update.



Debugging

Tools like Redux DevTools make debugging simple.



Scalable

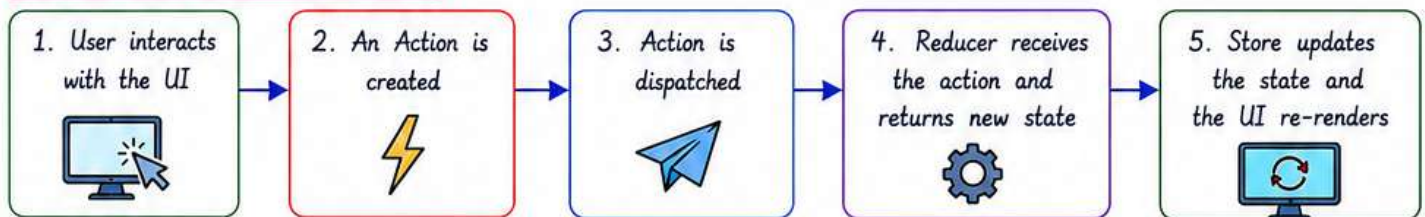
Well-suited for large and complex apps.

11.8.2 Core Concepts of Redux

1. **Store**: A single object that holds the entire state tree of the application.
2. **Action**: An object that describes what happened. It has a **type** and optional **payload**.
3. **Reducer**: A function that takes the current state and an action, and returns the new state.
4. **Dispatch**: A function used to send actions to the store.
5. **Subscribe**: Used to listen for changes in the store.



11.8.3 Basic Flow in Redux



Key Points:

- ✓ Redux provides a centralized store to manage the entire application state.
- ✓ It follows unidirectional data flow which makes the code predictable.
- ✓ It has a strict architecture with Store, Action and Reducer.
- ✓ Redux is useful for medium to large-scale applications.
- ✓ It improves state management, debugging and code maintainability.



Key Takeaway:



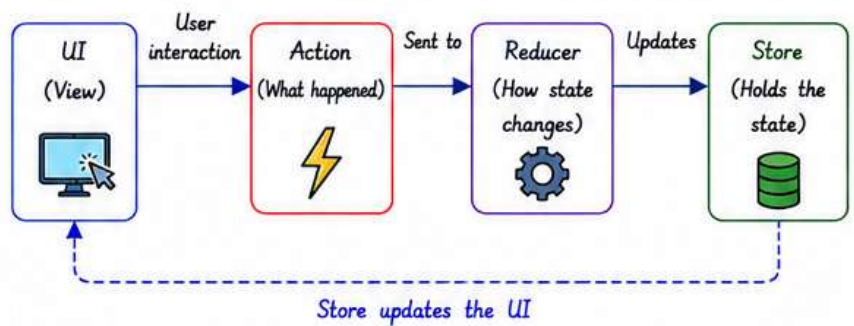
Redux helps manage complex application state in a **predictable** and **maintainable** way using a centralized store, actions and reducers.
It makes large applications **easier** to build, test and debug.

11.9 Redux Architecture

11.9.1 What is Redux Architecture?

- * Redux follows a **unidirectional data flow**.
- * It is built around a central **Store** that holds the entire state tree.
- * The state can only be changed by dispatching an **Action**.
- * A **Reducer** computes the new state based on the current state and the **Action**.

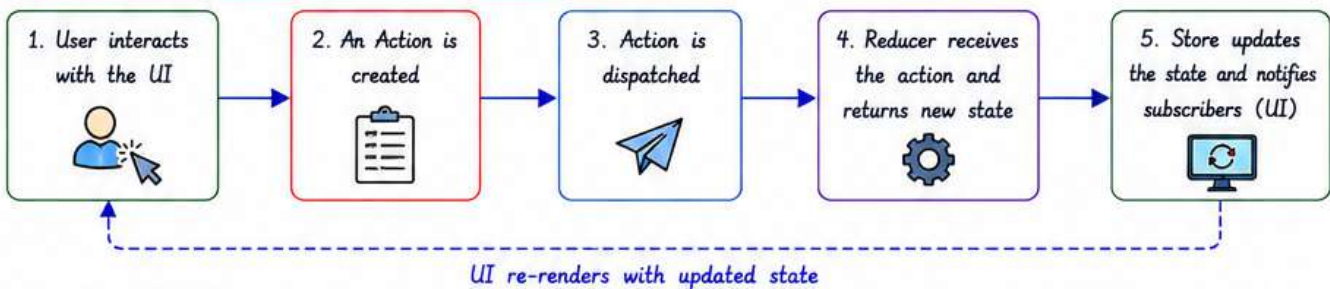
Redux Architecture Overview



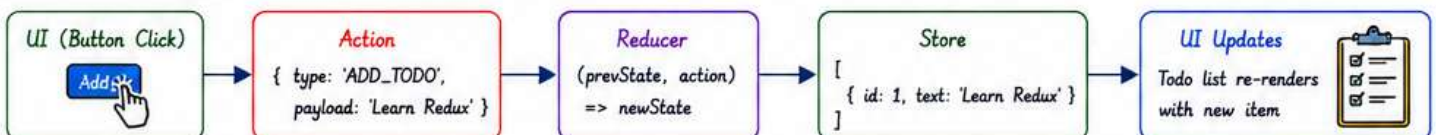
11.9.2 Core Components of Redux

Component	Description	Role in Architecture
Store	A single object that holds the entire state tree of the application.	Central storage for the state. There is only one store in a Redux application.
Action	A plain JavaScript object that describes what happened .	Carries information from the UI to the Reducer. Must have a type property.
Reducer	A pure function that takes the current state and an action, and returns a new state .	Contains the logic to update the state based on the action.
Dispatch	A function provided by Redux to send actions to the store.	Triggers the flow of data by sending actions to the store.
Subscribe	A function used to listen for changes in the store.	Notifies the UI whenever the state changes.

11.9.3 Data Flow in Redux (Step-by-Step)



11.9.4 Example



Key Points:

- ✓ Redux architecture is based on a single Store.
- ✓ Data moves in one direction: **UI → Action → Reducer → Store → UI**.
- ✓ State is read-only and can be changed only by dispatching actions.
- ✓ Reducers are pure functions that return new state.
- ✓ Subscribe helps the UI to re-render when the state changes.



Key Takeaway:



Redux architecture enforces a **predictable** and **maintainable** way to manage state in JavaScript applications using a **single source of truth (Store)**.

11.10 Store

11.10.1 What is Store?

- * The **Store** is the **heart** of a Redux application.
- * It holds the **entire state tree** of the application.
- * There is **only one Store** in a Redux application.
- * The Store allows access to state via **getState()**.
- * It allows state to be updated via **dispatch(action)**.
- * It also allows listeners to **subscribe** to state changes.

11.10.2 Key Points about Store

- ✓ Store holds the complete state of the application.
- ✓ State in the Store is **read-only**.
- ✓ State can only be changed by **dispatching an Action**.
- ✓ Store is created using **createStore()** method.
- ✓ Store is the **single source of truth** in Redux.

11.10.3 Store API

Method	Description	Returns
<code>getState()</code>	Returns the current state from the Store.	Current state
<code>dispatch(action)</code>	Dispatches an action to update the state.	The action
<code>subscribe(listener)</code>	Registers a listener that is called on state change.	A function (unsubscribe)

11.10.4 Creating a Store

```
import { createStore } from 'redux';  
const store = createStore(reducer, preloadedState, enhancer);
```

Store object

Root reducer
(a function)

Initial state
(optional)

Enhancer (e.g.,
applyMiddleware)
(optional)



11.10.5 Example

```
// 1. Reducer  
function counterReducer(state = 0, action) {  
  switch (action.type) {  
    case 'INCREMENT':  
      return state + 1;  
    case 'DECREMENT':  
      return state - 1;  
    default:  
      return state;  
  }  
}
```

```
// 2. Create Store  
import { createStore } from 'redux';  
const store = createStore(counterReducer);  
// 3. Get State  
console.log('Initial State:', store.getState());  
// 4. Dispatch Actions  
store.dispatch({ type: 'INCREMENT' });  
store.dispatch({ type: 'INCREMENT' });  
store.dispatch({ type: 'DECREMENT' });  
// 5. Updated State  
console.log('Updated State:', store.getState());
```

Output
Initial State: 0
Updated State: 1



Key Points:

- ✓ Store holds the entire application state.
- ✓ It provides methods to access, update and listen to state changes.
- ✓ There is only one Store in a Redux application.
- ✓ State can be changed only by dispatching Actions.
- ✓ Store is created using **createStore()** function.



Key Takeaway:



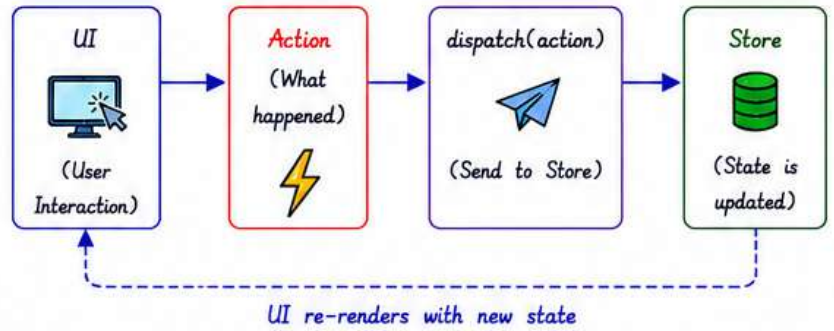
The Store is the central container that holds the state tree.
It is the **single source of truth** and enables **predictable** state management in Redux.

11.11 Actions

11.11.1 What is an Action?

- * An Action is a **plain JavaScript object**.
- * It describes **what happened** in the application.
- * It must have a **type** property.
- * It can have an optional **payload** (data).
- * Actions are sent to the Store using **dispatch()**.
- * They are the **only way** to change the state.

Action Flow in Redux



11.11.2 Structure of an Action

```
{
  type: 'ACTION_TYPE',
  payload: data // optional
}
```



11.11.3 Rules of Actions

- ✓ Actions must be **plain JavaScript objects**.
- ✓ They must have a **type** property.
- ✓ The type should be a **string constant**.
- ✓ **Payload** is optional but useful to send data.
- ✓ Actions are **immutable** (do not modify them).

11.11.4 Examples of Actions

```
// 1. Action without payload
{ type: 'INCREMENT' }

// 2. Action with payload
{ type: 'ADD_TODO', payload: { id: 1, text: 'Learn Redux' } }

// 3. Action with multiple fields in payload
{ type: 'LOGIN', payload: { username: 'admin', token: 'abc123' } }
```



11.11.5 Action Creators

- * Action creators are functions that return **Action** objects.
- * They help avoid repeating the same action object.
- * Makes code **cleaner** and easier to maintain.

```
// Action Creator Example
const addTodo = (text) => ({
  type: 'ADD_TODO',
  payload: { text }
});
```



11.11.6 Using Actions with dispatch()

```
// 1. Import useDispatch
import { useDispatch } from 'react-redux';
// 2. Get dispatch function
const dispatch = useDispatch();
// 3. Dispatch an action
dispatch({ type: 'INCREMENT' });
dispatch({ type: 'ADD_TODO', payload: { text: 'Learn Redux' } });
// 4. Using Action Creator
dispatch(addTodo('Learn Redux'));
```



11.11.7 Built-in Action (Example)

Redux itself does not provide built-in actions. We define our own action types as per requirement. Example:

- * 'INCREMENT'
- * 'DECREMENT'
- * 'ADD_TODO'
- * 'REMOVE_TODO'



Key Points:

- ✓ Actions are plain JavaScript objects that describe what happened.
- ✓ They must have a **type** property and can have an optional payload.
- ✓ Actions are the **only way** to change the state in Redux.
- ✓ Use **dispatch()** to send actions to the Store.
- ✓ Action creators help to write cleaner and reusable code.



Key Takeaway:



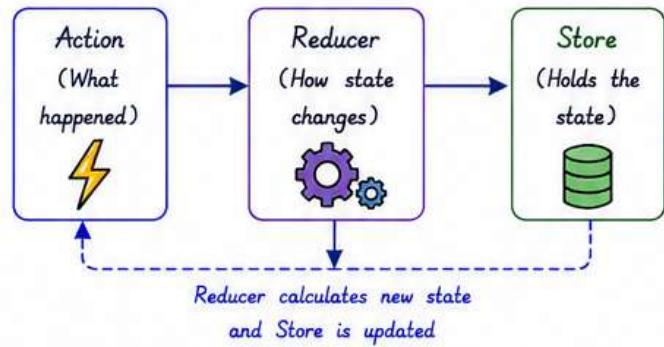
Actions are the events that trigger state changes in Redux. They carry information from the UI to the Store through **dispatch()**. Always define meaningful action types and use action creators for better code structure.

11.12 Reducers

11.12.1 What is a Reducer?

- * A Reducer is a **pure function**.
- * It takes the **current state** and an **action** as arguments.
- * It returns a **new state** based on the action.
- * It **must not modify** the existing state.
- * Reducers specify **how the state changes** in response to actions.

Role of Reducer in Redux



11.12.2 Reducer Syntax

```
function reducer(state = initialState, action) {  
  switch (action.type) {  
    case 'ACTION_TYPE_1':  
      return newState1;  
    case 'ACTION_TYPE_2':  
      return newState2;  
    default:  
      return state; // return current state if no match  
  }  
}
```

Annotations:

- Current state (points to `state = initialState`)
- Action object (points to `action`)
- Action type 1 (points to `case 'ACTION_TYPE_1':`)
- New state for action type 1 (points to `return newState1;`)
- Action type 2 (points to `case 'ACTION_TYPE_2':`)
- New state for action type 2 (points to `return newState2;`)
- Default: return current state (points to `return state;`)

11.12.3 Rules for Writing Reducers

- ✓ Reducers must be **pure functions**.
- ✓ They must **not modify** the existing state.
- ✓ They must return a **new state object**.
- ✓ They must return the **current state** if nothing changes.
- ✓ Do not perform **side-effects** like API calls, `localStorage`, etc. inside reducers.
- ✓ Use **spread operator** (`...`) to update state **immutably**.

11.12.4 Example: Counter Reducer

```
const initialState = { count: 0 };  
function counterReducer(state = initialState, action) {  
  switch (action.type) {  
    case 'INCREMENT':  
      return { ...state, count: state.count + 1 };  
    case 'DECREMENT':  
      return { ...state, count: state.count - 1 };  
    default:  
      return state; // no changes  
  }  
}
```



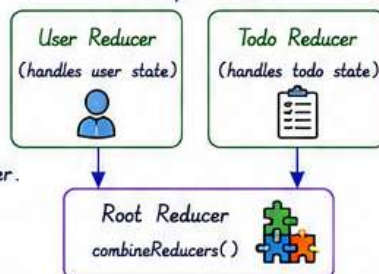
How it Works?

Action Dispatched	Current State	New State (Returned)
{ type: 'INCREMENT' }	{ count: 0 }	{ count: 1 }
{ type: 'INCREMENT' }	{ count: 1 }	{ count: 2 }
{ type: 'DECREMENT' }	{ count: 2 }	{ count: 1 }
Unknown Action	{ count: 1 }	{ count: 1 }

11.12.5 Combining Multiple Reducers

- * In a large application, we split reducers and combine them.
- * Redux provides `combineReducers()` to combine multiple reducers into one root reducer.

Multiple Reducers



```
import { combineReducers } from 'redux';  
import userReducer from './userReducer';  
import todoReducer from './todoReducer';  
  
const rootReducer = combineReducers({  
  user: userReducer,  
  todos: todoReducer  
});  
  
export default rootReducer;
```

Key Points:

- ✓ Reducers are pure functions that calculate the next state.
- ✓ They take the current state and an action as arguments.
- ✓ They must not change the existing state.
- ✓ They return a new state using immutable updates.
- ✓ Reducers define how the application state evolves in response to actions.



Key Takeaway:



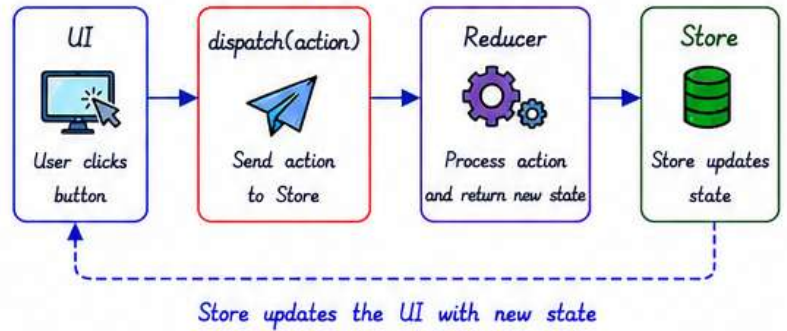
Reducers are the brain of Redux. They decide how the state changes based on actions. Always write reducers as **pure functions** and return new state **immutably**.

11.13 Dispatch

11.13.1 What is Dispatch?

- * **Dispatch** is a function provided by Redux.
- * It is used to send an **Action** to the **Store**.
- * Dispatch is the **only way** to trigger a state change.
- * When we dispatch an action, it reaches the **Reducer** through the **Store**.
- * Then **Reducer** calculates the new state and **Store** updates the state.

How Dispatch Works in Redux



11.13.2 Purpose of Dispatch

- ✓ It sends **Actions** from the application (UI) to the **Store**.
- ✓ It triggers the **Redux data flow**.
- ✓ It connects **UI events** (like clicks, inputs) to state changes.
- ✓ It keeps the state updates **predictable and controlled**.



11.13.3 Dispatch Syntax

`dispatch(action)`

Dispatch function (provided by Redux) Action object (to be sent)



11.13.4 Dispatch in Redux Flow

Step	What Happens?	Description
1.	User Interaction	User interacts with the UI (e.g., clicks a button).
2.	Dispatch Action	<code>dispatch(action)</code> is called with an Action object.
3.	Action Sent to Store	The action is sent to the Store using <code>dispatch()</code> .
4.	Reducer Processes	Reducer receives the action , processes it and returns new state .
5.	Store Updates State	Store updates the state and notifies subscribers.
6.	UI Re-renders	UI re-renders with the updated state.

11.13.5 Example

```
// 1. Import useDispatch
import { useDispatch } from 'react-redux';

// 2. Get dispatch function
const dispatch = useDispatch();

// 3. Dispatch an action
const handleClick = () => {
  dispatch({
    type: 'INCREMENT',
    payload: { amount: 1 }
  });
};

// 4. Use in UI
return (
  <button onClick={handleClick}>Increment</button>
);
```



11.13.6 Dispatch with Action Creators



Action Creator
Functions that return Action objects.

```
// Action Creator
const addTodo = (text) => ({
  type: 'ADD_TODO',
  payload: { text }
});
```

Using Dispatch

```
dispatch(addTodo('Learn Redux'));
// Action object is sent to Store
```

Key Points:

- ✓ Dispatch is a function used to send **Actions** to the **Store**.
- ✓ It is the **only way** to trigger state changes in Redux.
- ✓ It connects **UI events** to the Redux flow.
- ✓ When an action is dispatched, **Reducer** processes it and **Store** updates the state.
- ✓ Always use action creators for cleaner and reusable code.



Key Takeaway:



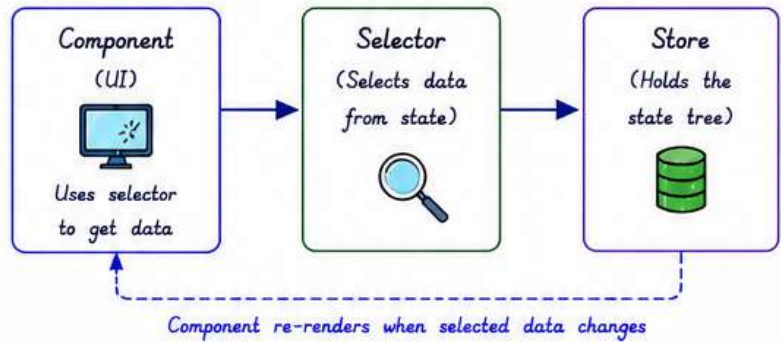
Dispatch is the **bridge** between the **UI** and the **Redux Store**.
It sends **Actions**, triggers state updates and keeps the application **predictable**.

11.14 Selectors

11.14.1 What is a Selector?

- * A Selector is a function that extracts specific data from the **Redux Store** state.
- * It helps components get the data they need without knowing the **structure** of the state.
- * Selectors keep components **clean** and **focused**.
- * They improve **reusability** and **performance**.
- * Commonly used with **useSelector()** from **react-redux**.

Where Selectors Fit in Redux



11.14.2 Purpose of Selectors

- ✓ Extract only the **required data** from the state.
- ✓ Prevent components from depending on **state structure**.
- ✓ Make code more **reusable** and **maintainable**.
- ✓ Improve **performance** by avoiding unnecessary re-renders.
- ✓ Encapsulate **state access logic** in one place.



11.14.3 Example: Without Selector

```
// Component directly reads state structure
function UserName() {
  const name = useSelector(state => state.user.profile.name);
  return <h3>{name}</h3>;
}
```

Tight coupling with state structure!



11.14.4 Example: With Selectors

```
// Selector
const selectUserName = (state) => state.user.profile.name;
// Component uses selector
function UserName() {
  const name = useSelector(selectUserName);
  return <h3>{name}</h3>;
}
```

Clean, reusable and easy to change!



11.14.5 Selector Example (Realistic)

```
// Selector to get all todos
export const selectTodos = (state) => state.todos.items;
// Selector to get completed todos
export const selectCompletedTodos = (state) =>
  state.todos.items.filter(todo => todo.completed);
// Selector to get todo count
export const selectTodoCount = (state) =>
  state.todos.items.length;
```



11.14.6 Using Selectors in Components

```
import { useSelector } from 'react-redux';
import { selectTodos, selectTodoCount } from './selectors';
function TodoSummary() {
  const todos = useSelector(selectTodos);
  const count = useSelector(selectTodoCount);
  return (
    <div>
      <h4>Total Todos: {count}</h4>
    </div>
  );
}
```



11.14.7 Type of Selectors

Type	Description	Example Use
 Simple Selectors	Return a single piece of data.	Get user name, todo count
 Derived Selectors	Return computed or transformed data.	Filter, sort, calculate total
 Memoized Selectors	Use memoization to improve performance.	With reselect library

11.14.8 Memoization with Reselect

```
import { createSelector } from 'reselect';
// Input selector
const selectItems = (state) => state.todos.items;
// Memoized selector
export const selectCompletedCount = createSelector(
  [selectItems],
  (items) => items.filter(item => item.completed).length
);
```



Improves performance!

11.14.9 Best Practices for Selectors

- ✓ Keep selectors **outside** of components.
- ✓ Make selectors **small** and **focused**.
- ✓ **Reuse** selectors in multiple components.
- ✓ Use **memoized** selectors for expensive computations.
- ✓ Do not cause **side-effects** inside selectors.



Key Points:

- ✓ Selectors extract specific data from the **Redux Store**.
- ✓ They keep components decoupled from state structure.
- ✓ They improve **reusability**, **maintainability** and **performance**.
- ✓ Memoized selectors (e.g., using **reselect**) prevent unnecessary recalculations.
- ✓ Use selectors with **useSelector()** in **react-redux**.



Key Takeaway:



Selectors act as a **bridge** between components and the **Redux Store**. They return the **right data**, in the **right shape**, at the **right time**.

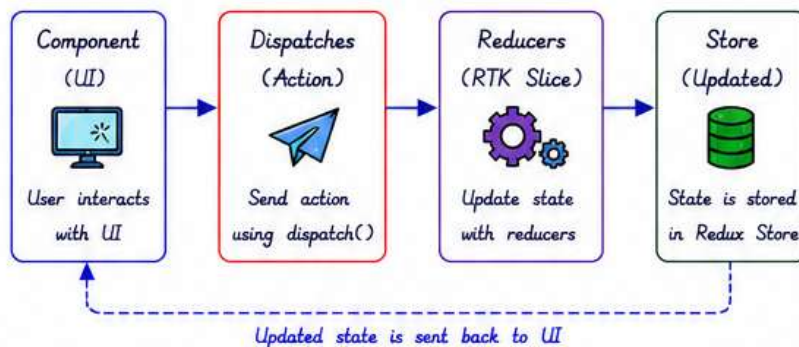


11.15 Redux Toolkit

11.15.1 What is Redux Toolkit?

- * Redux Toolkit (RTK) is the official, recommended way to write Redux logic.
- * It **simplifies Redux** by providing built-in utilities, good defaults and less boilerplate code.
- * It is part of the 'reduxjs/toolkit' package.
- * It helps us write Redux code that is **easier to read, maintain and scale**.
- * It uses **Immer** internally to handle immutable updates.

How Redux Toolkit Simplifies Redux



11.15.2 Why Use Redux Toolkit?

- ✓ Reduces **boilerplate** code.
- ✓ Includes **useful** built-in APIs.
- ✓ Provides **good defaults** and best practices.
- ✓ Works great with **TypeScript**.
- ✓ Includes **Developer Tools** support out of the box.



11.15.3 Core Features of Redux Toolkit

- * **configureStore()** - Simplified store configuration.
- * **createSlice()** - Combines actions and reducers.
- * **createAsyncThunk()** - Handles async logic easily.
- * Built-in **Immer** - Write mutable logic, RTK makes it immutable.
- * **Less boilerplate** - Write more code, less setup.
- * **TypeScript support** - Built with TS in mind.
- * **DevTools integration** - Enabled by default.

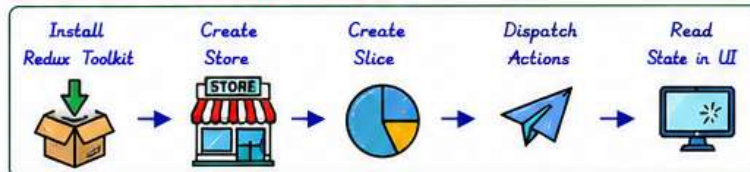


11.15.4 Installation

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



11.15.5 Basic Usage Overview



11.15.6 Example: Counter with Redux Toolkit

```
// 1. Import required functions
import { configureStore, createSlice } from '@reduxjs/toolkit';
// 2. Create a slice
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
    decrement: (state) => { state.value -= 1; },
    reset: (state) => { state.value = 0; }
  }
});
// 3. Export actions and reducer
export const { increment, decrement, reset } = counterSlice.actions;
export default counterSlice.reducer;
// 4. Configure store
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});
```



11.15.7 createSlice() Structure

```
createSlice({
  name: 'sliceName', // unique name
  initialState: { ... }, // initial state
  reducers: {
    action1: (state, action) => { ... },
    action2: (state, action) => { ... }
  },
  extraReducers: { ... } // for async actions
});
```



11.15.8 createAsyncThunk() (For Async Logic)

```
// Example: Fetch todos from API
import { createAsyncThunk } from '@reduxjs/toolkit';
export const fetchTodos = createAsyncThunk(
  'todos/fetchTodos',
  async () => {
    const response = await fetch('https://jsonplaceholder.typicode.com/todos');
    return await response.json();
  }
);
```



11.15.9 Key Points

- ✓ Redux Toolkit is the official way to write Redux logic.
- ✓ It reduces boilerplate and saves development time.
- ✓ Use **configureStore()** to create the store.
- ✓ Use **createSlice()** to write reducers and actions together.
- ✓ Use **createAsyncThunk()** to handle async operations easily.
- ✓ It includes Immer, DevTools and best practices by default.

Benefits

- * Less Code
- * Better Performance
- * Easier Maintenance
- * Built-in Best Practices



11.15.10 Summary



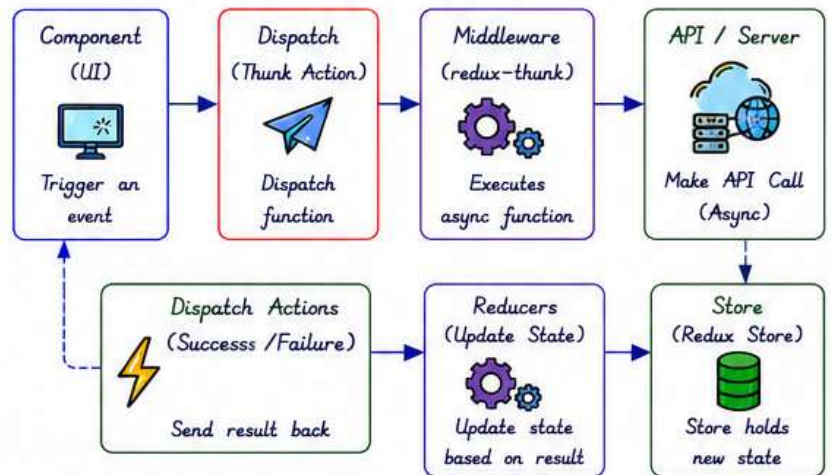
- * Redux Toolkit makes Redux development simple, fast and efficient.
- * It provides powerful APIs like **configureStore()**, **createSlice()** and **createAsyncThunk()**.
- * It helps build scalable and maintainable state management with less code and better structure.

11.16 Async Redux

11.16.1 What is Async Redux?

- Async Redux is used to handle **asynchronous** operations like API calls.
- Since Redux is synchronous by default, we use **middleware** to handle async logic.
- The most popular middleware is **redux-thunk**.
- It allows us to dispatch **functions** instead of plain action objects.
- Async Redux helps in **fetching**, **adding**, **updating** and **deleting** data from servers.

Async Redux Flow



11.16.2 Why Use Async Redux?

- ✓ Handles **API calls** and asynchronous tasks easily.
- ✓ Keeps **components clean** and simple.
- ✓ Centralizes **async logic** in actions.
- ✓ Provides **loading**, **success** and **error** states.
- ✓ Improves user experience in real applications.



11.16.3 Middleware for Async Redux

- **redux-thunk**
 - Allows dispatching functions (thunks).
 - Thunks receive (dispatch, getState) as arguments.
 - Performs async tasks and dispatches actions.
- **Alternative Middleware**
 - **redux-saga** – Uses generator functions.
 - **redux-observable** – Uses Observables (RxJS).



11.16.4 Thunk Syntax

```
function fetchData() {
  return async (dispatch, getState) => {
    dispatch({ type: 'FETCH_REQUEST' }); // start loading
    try {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      dispatch({ type: 'FETCH_SUCCESS', payload: data });
    } catch (error) {
      dispatch({ type: 'FETCH_FAILURE', payload: error });
    }
  }
}
```



11.16.6 Example: Fetch Data with redux-thunk

```
// 1. Action Types
export const FETCH_REQUEST = 'FETCH_REQUEST';
export const FETCH_SUCCESS = 'FETCH_SUCCESS';
export const FETCH_FAILURE = 'FETCH_FAILURE';

// 2. Thunk Action
export const fetchData = () => async (dispatch) => {
  dispatch({ type: FETCH_REQUEST });
  try {
    const res = await fetch('https://jsonplaceholder.typicode.com/posts');
    const data = await res.json();
    dispatch({ type: FETCH_SUCCESS, payload: data });
  } catch (err) {
    dispatch({ type: FETCH_FAILURE, payload: err.message });
  }
};
```



11.16.8 Best Practices

- ✓ Keep async logic in actions (thunks), not in components.
- ✓ Dispatch loading, success and failure actions.
- ✓ Handle errors gracefully.
- ✓ Show loader/spinner during async operations.
- ✓ Do not mutate state inside reducers.
- ✓ Keep API URLs and constants separate.



11.16.5 Async Flow with redux-thunk

Step	Action	What Happens?
1.	Component Event	User performs an action (e.g., button click)
2.	Dispatch Thunk	Component dispatches a thunk function
3.	redux-thunk Middleware	Middleware intercepts and executes the thunk
4.	API Call	Thunk makes async request to server
5.	Dispatch Result Actions	Thunk dispatches success or failure actions
6.	Reducers & Store Update	Reducers update state and Store holds new state

11.16.7 Reducer Handling Async Actions

```
const initialState = {
  loading: false,
  data: [],
  error: null
};

function dataReducer(state = initialState, action) {
  switch (action.type) {
    case FETCH_REQUEST:
      return { ...state, loading: true, error: null };
    case FETCH_SUCCESS:
      return { ...state, loading: false, data: action.payload };
    case FETCH_FAILURE:
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
}
```



11.16.9 Key Points

- Async Redux is used to handle asynchronous operations.
- **redux-thunk** middleware allows dispatching functions (thunks).
- Thunks perform async tasks and dispatch actions.
- Use loading, success and error states for better UX.
- Async Redux makes applications dynamic and real-time.



Summary



- Async Redux helps handle API calls and async operations smoothly.
- It uses middleware like **redux-thunk** to dispatch functions.
- Thunks perform async tasks and dispatch result actions.
- Reducers update state, and Store provides updated data to UI.
- It improves user experience by showing loading, success and error states.



11.17 RTK Query

11.17.1 What is RTK Query?

- * RTK Query is a data fetching and caching tool built into **Redux Toolkit**.
- * It simplifies **API calls** and reduces boilerplate code.
- * It handles **caching, loading, error states, polling, pagination** and more automatically.
- * It is built on top of **createApi** and uses **Redux store** internally.
- * It eliminates the need for extra libraries like **redux-thunk** for most use cases.



11.17.2 Why Use RTK Query?

- ✓ Automatic **caching** and re-fetching.
- ✓ Less boilerplate compared to **redux-thunk**.
- ✓ Built-in **loading, success** and **error states**.
- ✓ **Pagination, polling, refetch** on focus/reconnect.
- ✓ Deduplication of requests.
- ✓ Easy integration with **Redux store**.
- ✓ Improves performance and developer experience.



11.17.4 Basic Setup

```
// 1. Create API slice
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';
export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://jsonplaceholder.typicode.com/',
  }),
  endpoints: (builder) => ({}),
});

// 2. Add reducer to store
import { configureStore } from '@reduxjs/toolkit';
import { api } from './api';
export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(api.middleware),
});
```



11.17.6 Using RTK Query Hooks in Component

```
import React from 'react';
import { useGetPostsQuery, useAddPostMutation } from './api';

function Posts() {
  // Fetch posts
  const { data, isLoading, isError, error } = useGetPostsQuery();
  // Add post
  const [addPost, { isLoading: isAdding }] = useAddPostMutation();

  if (isLoading) return <p>Loading...</p>;
  if (isError) return <p>Error: {error.toString()}</p>;
  const handleAdd = () => {
    addPost({ title: 'New Post', body: 'Hello RTK Query', userId: 1 });
  };

  return (
    <div>
      <h2>Posts</h2>
      <button onClick={handleAdd} disabled={isAdding}>Add Post</button>
      <ul>
        {data?.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

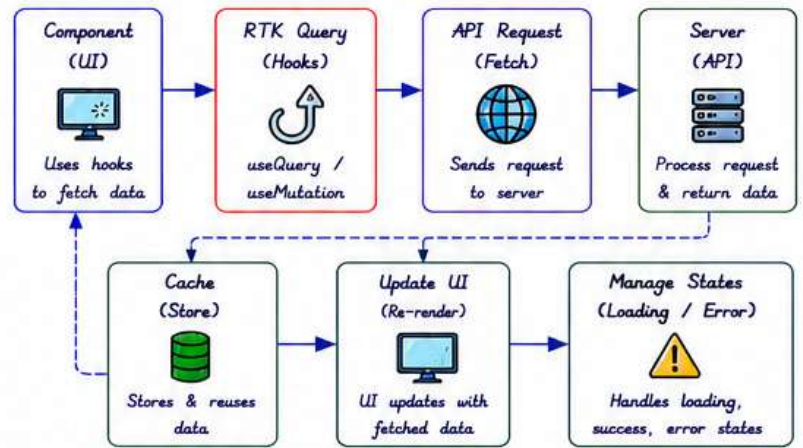


11.17.10 Key Takeaway

- ✓ Use RTK Query for most data fetching needs.
- ✓ Use tags and invalidatesTags for smart cache updates.
- ✓ Keep endpoints modular and in separate api files.
- ✓ Use polling only when necessary.
- ✓ Handle errors and loading states in UI.
- ✓ Take advantage of caching and request deduplication.



RTK Query Flow



11.17.3 Key Features

- * **Automatic Caching** - Stores data and reuses it.
- * **Request Deduplication** - Prevents duplicate requests.
- * **Background Refetching** - Keeps data fresh.
- * **Polling** - Fetch data at regular intervals.
- * **Pagination & Infinite Scroll** - Built-in support.
- * **Error Handling** - Easy error and retry handling.
- * **Optimistic Updates** - Update UI before server response.
- * **Tag-based Cache Invalidation** - Invalidate and refetch smartly.



11.17.5 Define Endpoints

```
export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://jsonplaceholder.typicode.com/' }),
  tagTypes: ['Post'],
  endpoints: (builder) => ({
    // Query endpoint (GET)
    getPosts: builder.query({
      query: () => 'posts',
      providesTags: (result) =>
        result ? [...result.map(({ id }) => ({ type: 'Post', id })), { type: 'Post', id: 'LIST' }]
        : [{ type: 'Post', id: 'LIST' }],
    }),
    // Mutation endpoint (POST)
    addPost: builder.mutation({
      query: (newPost) => ({
        url: 'posts',
        method: 'POST',
        body: newPost,
      }),
      invalidatesTags: [{ type: 'Post', id: 'LIST' }],
    }),
  })
});
```



11.17.7 Hook Reference

Hook	Purpose	Example
 useQuery	Fetch data (GET)	useGetPostsQuery()
 useMutation	Modify data (POST/PUT/DELETE)	useAddPostMutation()
 useLazyQuery	Fetch data on demand	useLazyGetPostsQuery()
 useQueryState	Access query state	useGetPostsQueryState()
 useQuerySubscription	Subscribe to query	useGetPostsQuerySubscription()

11.17.9 Benefits

- * Less boilerplate code.
- * Built-in caching and data management.
- * Better performance and reliability.
- * Easy to use and integrate.
- * Faster development with modern best practices.



RTK Query is the official and recommended way to handle data fetching in Redux applications. It reduces boilerplate, improves performance and provides powerful features like caching, polling, pagination and automatic re-fetching with minimal effort.



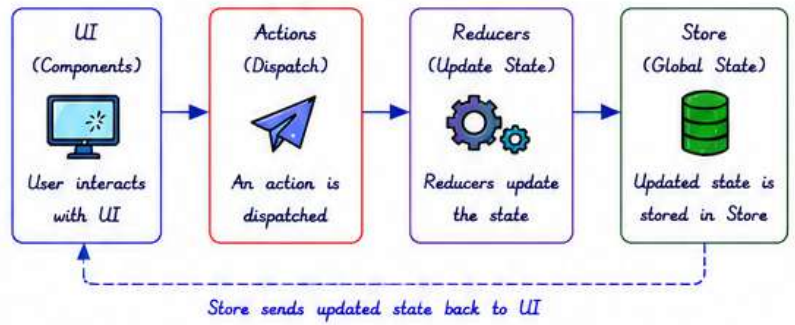
11.18 Global State Architecture

11.18.1 What is Global State Architecture?

- * Global State Architecture is the overall structure of a Redux application.
- * It shows how **data flows** between UI, actions, reducers and the store.
- * It helps in **managing and sharing state** across all components.
- * It makes the application **predictable, scalable and easy to maintain**.



Global State Architecture (Redux Flow)



11.18.2 Principles of Global State Architecture

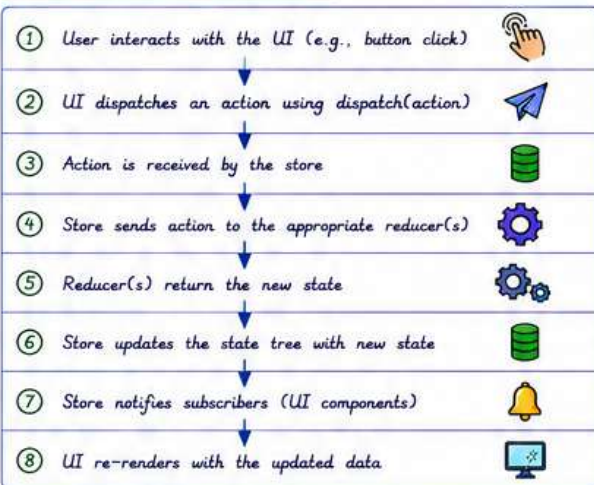
- ✓ **Single Source of Truth** - One store holds the entire state.
- ✓ **State is Read-Only** - State can only be changed by dispatching actions.
- ✓ **Changes are Made with Pure Functions** - Reducers are pure.
- ✓ **Predictable Updates** - Same action + state = same result.
- ✓ **Data Flows in One Direction** - UI → Actions → Store → UI.



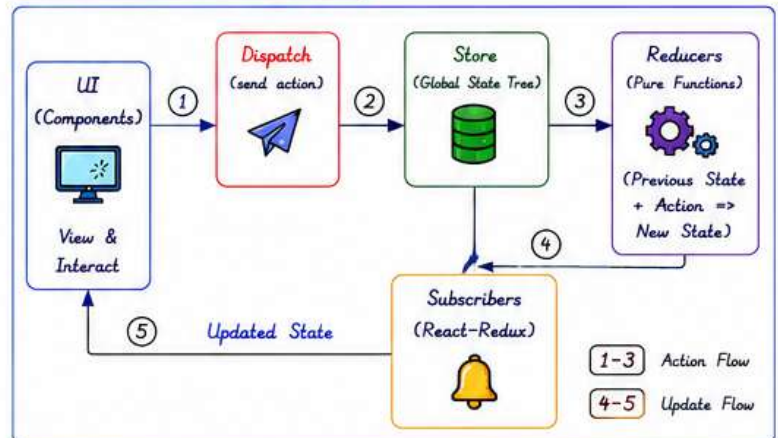
11.18.3 Main Building Blocks

Component	Role	Icon
Store	Holds the entire state tree	
Actions	Describe what happened	
Reducers	Specify how the state changes	
Dispatch	Sends actions to the store	
Selectors	Read specific data from the store	
UI (Components)	Displays data and triggers actions	

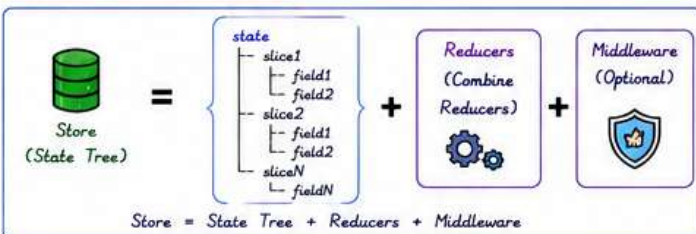
11.18.4 Detailed Data Flow



11.18.5 Architecture Diagram



11.18.6 Store Structure



11.18.7 Best Practices

- * Keep state normalized.
- * Split state into feature-based slices.
- * Use selectors to read only required data.
- * Keep reducers pure and simple.
- * Avoid storing unnecessary or duplicate data.
- * Use RTK (Redux Toolkit) for better structure.



11.18.8 Benefits

- * Centralized and predictable state management.
- * Easy debugging and time-travel.
- * Scalable for large applications.
- * Better performance with optimized updates.
- * Improves code maintainability.



11.18.9 Example: Global State in an App



11.18.10 Key Takeaway

- Global State Architecture in Redux provides a **single source of truth** where actions update the store and the updated state is shared across all components in a predictable and efficient way.

