Redux and Redux Thunk

A Centralized State Management Library

Redux is a predictable state container for JavaScript applications.

It helps you write applications that behave consistently, run in different environments (client, server, and native),[1] and are easy to test.

While often used with React, Redux is a standalone library and can be used with other UI frameworks or even without any UI.

Note: react-redux: connects your redux store with React Components
https://react-redux.js.org/

The Core Problem Redux Solves:

As applications grow in complexity, especially single-page applications (SPAs) built with frameworks like React, managing the application's state can become challenging.

Components might need to share data, and keeping track of where the state lives and how it's being updated can lead to:

- **Prop Drilling:** Passing data down through many nested components that don't actually need it.
- **Difficult Debugging:** It becomes hard to trace how a particular piece of state changed and which component triggered the change.
- **Unpredictable State Updates:** Multiple components might directly modify shared state, leading to unexpected behaviour and making it difficult to reason about the application's logic.

Redux is the Solution: A Centralized Store and Strict Data Flow

Redux addresses these issues by introducing a single, global store that holds the entire application's state. It also enforces a strict unidirectional data flow, making state updates predictable and easier to manage.

The Core Concepts of Redux:

1. **Store:**
    - The single source of truth for your application's state.
    - It's a JavaScript object that holds all the data.
    - You can access the current state of the store.
    - You can dispatch actions to update the state.
    - You can register listeners to be notified when the state changes.
    - Analogy: Think of it as a single, well-organized database for your front-end application.

2. **Actions:**
    - Plain JavaScript objects that describe what happened.
    - They are the *only* way to interact with the store and trigger a state change.
    - Actions must have a type property (a string constant) that indicates the type of action being performed.
    - They can also carry additional data (payload) needed to update the state.
    - Analogy: Think of them as instructions or events that you want to tell the store about (e.g., "user logged in," "item added to cart," "fetch todos started").

```
// Example Actions
{ type: 'USER_LOGGED_IN', payload: { userId: 123, username: 'Alice'
} }
{ type: 'ADD_TO_CART', payload: { productId: 'abc', quantity: 1 } }
```

3. **Action Creators:**
    - Functions that return action objects.
    - They abstract the creation of action objects, making your code more organized and less prone to errors (e.g., typos in action types).
    - Analogy: Think of them as factories that produce consistent action objects.

```
// Example Action Creators
const userLoggedIn = (userId, username) => {
  return { type: 'USER_LOGGED_IN', payload: { userId, username } };
};

const addToCart = (productId, quantity) => {
  return { type: 'ADD_TO_CART', payload: { productId, quantity } };
};
```

4. **Reducers:**
    - Pure functions that specify how the application's state changes in response to an action.
    - They take the **current state** and an **action** as arguments and return a **new state object**.

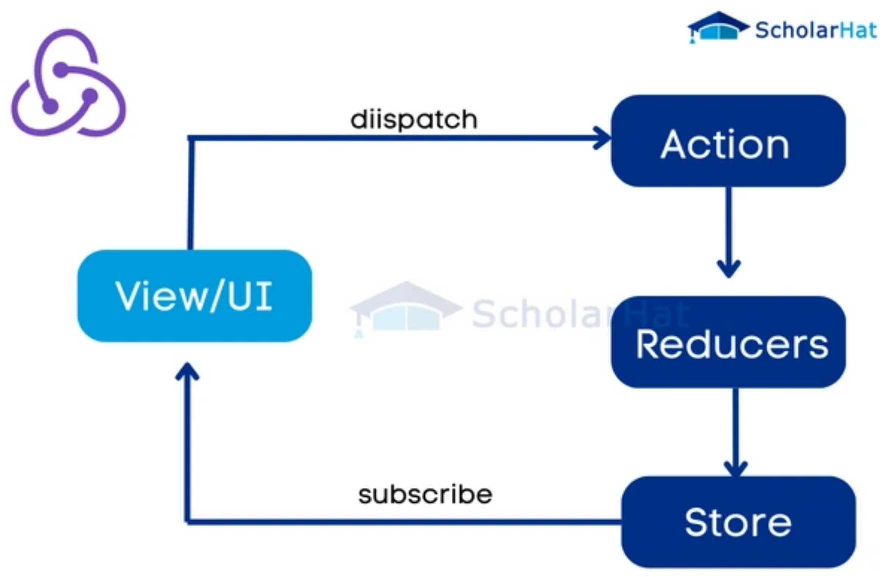
    - Important Principles of Reducers:
        - Purity: They should not have any side effects (e.g., modifying the original state, making API calls, logging). They should only compute the next state based on the input.
        - Immutability: They should not modify the existing state directly. Instead, they should create a new state object with the necessary updates. This is crucial for predictability and efficient change detection.

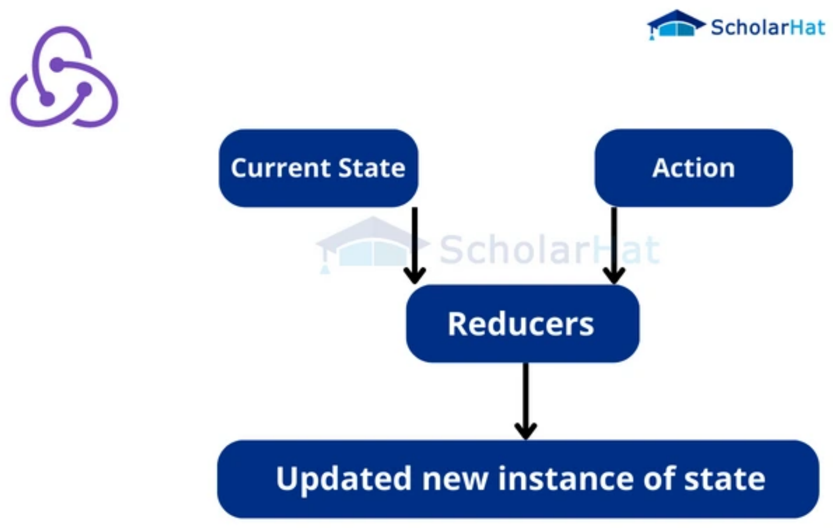
- Analogy: Think of them as the logic that determines how the "database" (the store) should be updated based on the "instructions" (actions).

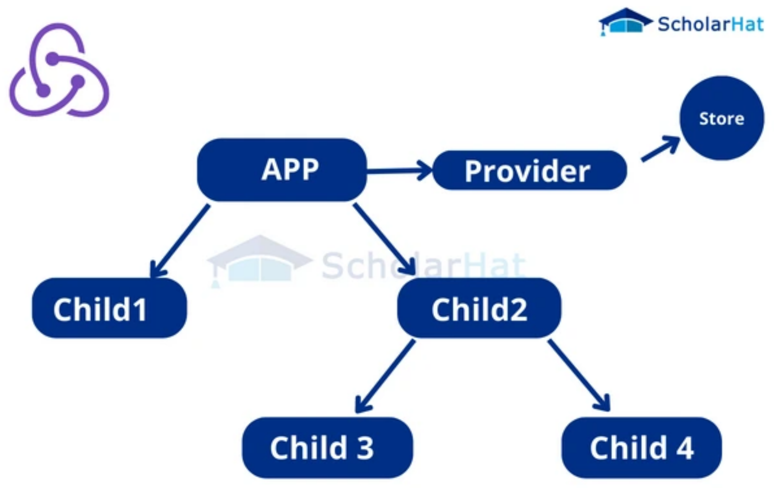
```
// Example Reducer
const initialState = {
 user: null,
 cart: []
};

const rootReducer = (state = initialState, action) => {
 switch (action.type) {
  case 'USER_LOGGED_IN':
   return { ...state, user: action.payload };
  case 'ADD_TO_CART':
   return { ...state, cart: [...state.cart, action.payload] };
  default:
   return state; // Return the current state if the action type is not
recognized
 }
};
```

5. **Dispatcher:**
   - A function (typically store.dispatch()) that is used to send actions to the store.
   - It's the only way to trigger a state change.
   - When you call dispatch(action), Redux takes that action and passes it to all the reducers in your application.
   - Analogy: Think of it as the post office where you mail your "instructions" (actions) to the central "database manager" (Redux).

The Flow of Actions Between Actions, Dispatcher, and Reducers:







Provider: The **Provider** component in React Redux ensures the application's store is available to all components. By wrapping the root component with the Provider and passing the store as a prop, all child components can access the store seamlessly.

The data flow in a Redux application is strictly unidirectional:

1. **Action is Created**: An event occurs in the application (e.g., a user clicks a button, a form is submitted, data is fetched from an API). This event triggers the creation of an action object, often using an action creator.
2. **Action is Dispatched:** The action object is then passed to the *store.dispatch()* function. This is how you tell Redux that something has happened and the state might need to be updated.
3. **Action Reaches Reducers**: The dispatch() function sends the action to all the reducers that have been registered with the store.
4. **Reducers Update State (Immutably)**: Each reducer receives the current state and the dispatched action. Based on the *action.type*, the reducer decides how to update the state.
   - Important: Reducers must create a new state object with the changes. They should not modify the original state directly.
5. **Store is Updated:** Once all reducers have processed the action, the store holds the new state returned by the root reducer (which combines the results of individual reducers).
6. **Components are Notified: If** any parts of the application's UI are connected to the Redux store (e.g., using connect in React Redux), they will be notified that the state has changed.
7. **Components Re-render:** The connected components can then access the updated state from the store and re-render themselves to reflect the changes.

Key Takeaways:

- Redux provides a centralized way to manage application state.
- State updates follow a strict unidirectional flow:
    - **Action -> Dispatcher -> Reducer -> Store.**
- Actions describe *what happened*.
- Reducers describe *how the state changes* in response to actions and must be pure functions that update state immutably.
- The dispatcher is the mechanism for sending actions to the store.
- This strict flow makes state changes predictable, easier to debug, and helps in building more maintainable applications.

While Redux introduces some boilerplate code, its benefits in managing complex state in large applications often outweigh this cost.

Libraries like Redux Toolkit aim to simplify Redux development and reduce the amount of boilerplate.

One of the common concerns when working with global state management like Redux is how to isolate state of the individual components.

While the Redux store is indeed globally accessible to any connected component, the way you structure your state and how components interact with it allows you to effectively achieve a level of isolation where components primarily access and interact with "their" relevant state.

The Key Principles for Component-Specific State Management within Redux:

1. Thoughtful State Design (Granular Slices):
    - Organize by Feature/Domain: Instead of having one massive state object, break down your global state into smaller, manageable "slices" based on different features or domains of your application.

    - For example, you might have slices for user, products, cart, notifications, etc.

    - Component-Related Data in Specific Slices: If a particular component or a group of related components manages a specific piece of data, that data should ideally reside within a slice that is conceptually related to that feature.

2. Selective Subscription with useSelector:
    - Targeted Data Extraction: In your components, use the useSelector hook (from react-redux) to subscribe *only* to the specific parts of the global state that the component actually needs.

    - Selector Functions: Create well-defined selector functions that know how to extract the necessary data from the relevant slices. This not only makes your component code cleaner but also provides a level of abstraction. If the underlying state structure changes, you only need to update the selector, not every component that uses it.
    -

3. Action Namespaces:
    - Slice Names as Prefixes: When using Redux Toolkit's `createSlice`, the `name` property automatically prefixes your action types. This helps in understanding which part of the application an action originates from and reduces the risk of accidental action type collisions.

4. Component Composition and Local State:
    - Prioritize Local State: For data that is truly local to a component and doesn't need to be shared across the application, use React's built-in `useState` hook. This keeps the state encapsulated within the component.
    - Pass Data Down via Props: If a parent component has data that a child component needs, pass it down as props. This is the standard way of data flow in React and should be preferred for direct parent-child relationships.

Redux is designed for managing *global* application state. The idea is to have a single source of truth that different parts of your application can rely on.

Trying to create strict, enforced isolation at the component level within Redux would go against this core principle and likely lead to more complexity and less predictability.

Analogy:

Think of a real-world organization. There's a central database (the Redux store) that holds information relevant to different departments (components/features).

- Granular Slices: The database is organized into tables for different departments (e.g., "HR," "Sales," "Inventory").
- Selective Access: Employees in the "Sales" department primarily access and modify the "Sales" tables, even though they technically *could* look at the "HR" tables (global accessibility).
- Action Namespaces: When someone makes a change request (an action), it's usually clear which department's data it's related to (action type prefix).
- Local Information: Each employee also has their own desk and notes (local useState) for temporary or private information.
- Communication via Reports: Departments share necessary information through reports and memos (passing props).

In Summary:

While Redux provides global access to the store, you achieve component-specific state management through:

- Well-organized and granular state slices.
- Components selectively subscribing only to the data they need using useSelector and well-defined selectors.
- Using action namespaces to understand the origin of actions.
- Prioritizing local useState for truly component-specific data.
- Using props for direct parent-child data sharing.

By following these patterns, you can effectively manage your application's state in a structured way with Redux, ensuring that components primarily interact with their relevant data without unnecessary coupling or interference.

Example:

Component-specific Store Management

The Core Idea:

The primary goal of this example is to illustrate how to manage different parts of your application's state in a Redux store in a way that keeps concerns separated and prevents components from being unnecessarily coupled to the entire global state. We achieve this by:

- **Dividing the Global State into Granular Slices:** Instead of one big state object, we have separate "slices" for different features (e.g., `userProfile` and `productList`).
- **Components Selectively Subscribing:** Each component uses the `useSelector` hook to subscribe only to the specific slice of the state it needs, using dedicated selector functions.
- **Actions Targeting Specific Slices:** Actions dispatched by a component

1. `src/stores/store.js` (The Redux Store):

```
import { configureStore } from '@reduxjs/toolkit';
import userProfileReducer from '../features/userProfile/userProfileSlice';
import productListReducer from '../features/productList/productListSlice';

export const store = configureStore({
  reducer: {
    userProfile: userProfileReducer,
    productList: productListReducer,
  },
});
```

- **`configureStore`:** This function from `@reduxjs/toolkit` sets up our Redux store in a simplified way.

- **`reducer`:** This is a crucial option. It's an object where:

  - The **keys** (`userProfile`, `productList`) represent the **names of the slices** of our global state. These will be the top-level properties of our Redux state object.
  - The **values** (`userProfileReducer`, `productListReducer`) are the **reducer functions** responsible for managing the state within their respective slices. These reducers are imported from our slice files.

After this setup, our Redux store's state will look something like this

```
{
 "userProfile": {
  "username": "Guest",
  "email": "",
  "isLoading": false
 },
 "productList": {
  "products": [
   { "id": 1, "name": "Laptop", "price": 1200 },
   { "id": 2, "name": "Mouse", "price": 25 },
   { "id": 3, "name": "Keyboard", "price": 75 }
  ],
  "sortBy": "price"
 }
}
```

2. `src/features/userProfile/userProfileSlice.jsx` (The `userProfile` Slice)

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

const initialState = {
 username: 'Guest',
 email: '',
 isLoading: false,
};

export const userProfileSlice = createSlice({
 name: 'userProfile',
 initialState,
 reducers: {
  updateUsername: (state, action) => {
   state.username = action.payload;
  },
  updateEmail: (state, action) => {
   state.email = action.payload;
  },

  setLoading: (state, action) => {
   state.isLoading = action.payload;
  },
 },
});
```

```
export const { updateUsername, updateEmail, setLoading } = userProfileSlice.actions;

// Selectors specific to the userProfile slice
export const selectUsername = (state) => state.userProfile.username;
export const selectEmail = (state) => state.userProfile.email;
export const selectUserProfileLoading = (state) => state.userProfile.isLoading;

export default userProfileSlice.reducer;
```

- **`createSlice`:** This function from `@reduxjs/toolkit` helps us define a slice of our Redux state.
- **`name: 'userProfile'`:** This is the name of the slice. It's used as a prefix for the generated action types (e.g., `userProfile/updateUsername`).
- **`initialState`:** This defines the initial data for the `userProfile` slice.
- **`reducers`:** This object contains functions that define how the `userProfile` state should change in response to certain actions.

    - `updateUsername`, `updateEmail`, `setLoading`: These are reducer functions. When an action with a matching type is dispatched, these functions will update the `state` argument (which is the `userProfile` part of the global state). Thanks to Immer (used by `createSlice`), we can write these as if we're directly mutating the state, but Immer handles the immutability for us.

- **`userProfileSlice.actions`:** `createSlice` automatically generates action creator functions for each reducer we defined. We export these (`updateUsername`, `updateEmail`, `setLoading`) so our `UserProfile` component can dispatch them.
- **`Selectors`:** These are functions that know how to extract specific pieces of data from the `userProfile` slice of the state.

    - `selectUsername`: Takes the entire Redux `state` as an argument and returns `state.userProfile.username`.
    - `selectEmail`: Takes the entire Redux `state` and returns `state.userProfile.email`.
    - `selectUserProfileLoading`: Takes the entire Redux `state` and returns `state.userProfile.isLoading`.

- **`export default userProfileSlice.reducer;`:** We export the reducer function generated by `createSlice`. This is what we imported into `store.js` and assigned to the `userProfile` key in the `reducer` object.

3. `src/features/userProfile/UserProfile.jsx` (The User Profile Component)

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { selectUsername, selectEmail, updateUsername, updateEmail } from
'./userProfileSlice';



function UserProfile() {
  const username = useSelector(selectUsername);
  const email = useSelector(selectEmail);
  const dispatch = useDispatch();
  const [localUsername, setLocalUsername] = useState(username);
  const [localEmail, setLocalEmail] = useState(email);

  const handleUsernameChange = (e) => {
    setLocalUsername(e.target.value);
  };

  const handleEmailChange = (e) => {
    setLocalEmail(e.target.value);
  };

  const handleUpdateProfile = () => {
    dispatch(updateUsername(localUsername));
    dispatch(updateEmail(localEmail));
  };

  return (
    // ... JSX to display and update username and email
  );
}

export default UserProfile;
```

- **`useSelector(selectUsername)`:** This hook from `react-redux` allows the component to subscribe to a specific part of the Redux state. We pass it the `selectUsername` selector. Whenever `state.userProfile.username` in the Redux store changes, this component will re-render, and the `username` variable will hold the updated value.
- **`useSelector(selectEmail)`:** Similarly, this subscribes to the `email` from the `userProfile` slice.
- **`useDispatch()`:** This hook returns a reference to the Redux `dispatch` function, which we use to dispatch actions.

- **`dispatch(updateUsername(localUsername))`:** When the "Update Profile" button is clicked, we dispatch the `updateUsername` action creator (imported from `userProfileSlice`). The value from the local input (`localUsername`) is passed as the `payload` of the action. This action will be intercepted by Redux, and the `userProfileReducer` will handle it, updating the `username` in the `userProfile` slice of the state. The same happens for `updateEmail`.
- **Local State (`useState`):** The component uses local state (`localUsername`, `localEmail`) to manage the input field values before they are dispatched to the Redux store. This is a common pattern for form inputs.

3. `src/features/productList/productListSlice.jsx` and `src/features/productList/ProductList.jsx`:

These files follow the same principles as the `userProfile` feature:

- **`productListSlice.jsx`:** Defines the `productList` slice with its `initialState` (an array of products and a `sortBy` option), `reducers` (`setSortBy`, `addProduct`), and **selectors** (`selectProducts`, `selectSortByOption`) specific to the product list data.
- **`ProductList.jsx`:** Uses `useSelector` with the `selectProducts` and `selectSortByOption` selectors to subscribe only to the product list-related data. It uses `useDispatch` to dispatch actions (`setSortBy`, `addProduct`) that will update the `productList` slice. It also uses local state for adding new products.

4. **`src/App.jsx` and `src/main.jsx`:**

- **`App.jsx`:** This is the main application component that simply renders the `UserProfile` and `ProductList` components. It's responsible for composing the UI.
- **`main.jsx`:** This is the entry point of the Vite application. It imports the Redux `store` and uses the `<Provider store={store}>` component to make the store available to all connected components in the application.

How This Demonstrates "Focused" Redux:

- **State Isolation:** The state for user profiles and product lists is managed in separate slices. Changes to the `userProfile` slice do not directly affect the `productList` slice, and vice versa.
- **Component Decoupling:** The `UserProfile` component only knows about the `userProfile` part of the Redux state through the specific selectors it uses. It doesn't need to know or care about the `productList` state. Similarly, the `ProductList` component is only concerned with its own slice of the state.
- **Selective Rendering:** When the Redux state is updated, only the components that are subscribed to the changed part of the state will re-render. For example, if the username is updated, only the `UserProfile` component (and any other component that selects the username) will re-render, not the `ProductList`.
- **Clearer Organization:** By organizing the state and logic into feature-based slices, the codebase becomes more modular and easier to understand and maintain.

Redux Thunk

https://daveceddia.com/what-is-a-thunk/

https://redux.js.org/tutorials/essentials/part-5-async-logic

A thunk is another word for a *function*. But it's not just any normal function. It is a name for a function that's returned by another function.

Like this:

```
function wrapper_function() {
 // this one is a "thunk" because it defers work for later:
 return function thunk() {   // it can be named, or anonymous
   console.log('do stuff now');
 };
}
```

We already know this pattern. We just don't call it "thunk." If we want to execute, we have to call it like `wrapper_function()()` – calling it twice, basically.

redux-thunk

So how does this apply to Redux?

We are aware of concepts in Redux such as "actions", "action creators", "reducers", and "middleware."

Actions are just objects. As far as Redux is concerned, out of the box actions must be *plain objects*, and they must have a `type` property. Aside from that, they can contain whatever you want – anything you need to describe the action you want to perform.

Actions look like this:

```
// 1. plain object
// 2. has a type
// 3. whatever else you want
{
  type: "USER_LOGGED_IN",
  payload: {username: "dave"}
}
```

And, since it's kind of annoying to write those objects by hand all the time (not to mention error-prone), Redux has the concept of "action creators" to stamp these things out:

```
function userLoggedIn() {
  return {
    type: 'USER_LOGGED_IN',
    payload: {username: "dave"}
  };
}
```

It's the same exact action, but now you can "create" it by calling the `userLoggedIn` function. This just adds one layer of abstraction.

Instead of writing the action object yourself, you call the function, which returns the object. If you need to dispatch the same action in multiple places around your app, writing action creators will make your job easier.

The Redux's so-called "actions" don't actually *do* anything? They're just objects. Plain and simple.

What if we make them do something? Like, say, make an API call, or trigger other actions?

Since reducers are supposed to be "pure" (as in, they don't change anything outside their scope) we cannot do any API calls or dispatch actions from inside a reducer.

If you want an action to *do* something, that code needs to live inside a function. That function (the "thunk") is a bundle of work to be done.

It would be nice if an action creator could return that function **– the bundle of work – instead of an action object**.

Something like this:

```
function getUser() {
  return function() {
    return axios.get('/current_user');
  };
}
```

If only there were some way to teach Redux how to deal with functions as actions…

Well, this is exactly what redux-thunk does**: it is a *middleware* that looks at every action that passes through the system, and if it's a function, it calls that function. That's all it does.**

The only thing we left out of that little code snippet is that Redux will pass two arguments to thunk functions: `dispatch`, so that they can dispatch new actions if they need to; and `getState`, so they can access the current state. So you can do things like this:

```
function logOutUser() {
  return function(dispatch, getState) {
    return axios.post('/logout').then(function() {
      // pretend we declared an action creator
      // called 'userLoggedOut', and now we can dispatch it
      dispatch(userLoggedOut());
    });
  };
}
```

The `getState` function can be useful for deciding whether to fetch new data, or return a cached result, depending on the current state.

That's about it. That's what `redux-thunk` is for.

After you install redux-thunk in your project, and apply the middleware, every action you dispatch will pass through this bit of code.

Setting up the middleware pipeline for your Redux application

1. **Redux Store Configuration with Thunk (store.js):**

```
import { configureStore } from '@reduxjs/toolkit';
import usersReducer from '../features/users/usersSlice';
import thunk from 'redux-thunk';

export const store = configureStore({
  reducer: {
    users: usersReducer,
  },
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(thunk),

});
```

----

middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(thunk)

The above line, within the configureStore function of Redux Toolkit, is responsible for setting up the middleware pipeline for your Redux application.

Middleware in Redux provides a way to intercept and process actions as they are dispatched to the store, but before they reach the reducers.

This allows you to add custom logic for various purposes, such as handling asynchronous operations, logging actions, or preventing certain actions from reaching the reducers under specific conditions.

1. middleware::

- This is a configuration option within configureStore. It tells Redux Toolkit how to set up the middleware for the store.

2. (getDefaultMiddleware) => ...:

- This is an arrow function that receives a single argument: getDefaultMiddleware.
- **getDefaultMiddleware():** This is a function provided by Redux Toolkit. When you call it, it returns an array containing the **default middleware** that Redux Toolkit includes for you. This typically includes middleware for common Redux patterns and optimizations.
  - For example, in recent versions of Redux Toolkit, getDefaultMiddleware often includes middleware to prevent accidental mutations of state outside of reducers and potentially other helpful utilities.

3. .concat(thunk):

- **.concat():** This is a standard JavaScript array method. It's used here to create a new array by taking the array returned by getDefaultMiddleware() and appending the thunk middleware to it.
- **thunk:** This is the Redux Thunk middleware that you imported: import thunk from 'redux-thunk';. As we discussed earlier, Redux Thunk allows you to dispatch functions (thunks) instead of plain action objects, enabling you to handle asynchronous operations.

In essence, this line of code does the following:

1. **Gets the default set of middleware provided by Redux Toolkit.** This ensures you have the recommended baseline middleware for your application.
2. **Adds the redux-thunk middleware to this default set.** This enables the asynchronous action handling capabilities that Redux Thunk provides.
3. **The resulting array of middleware is then applied to your Redux store.**

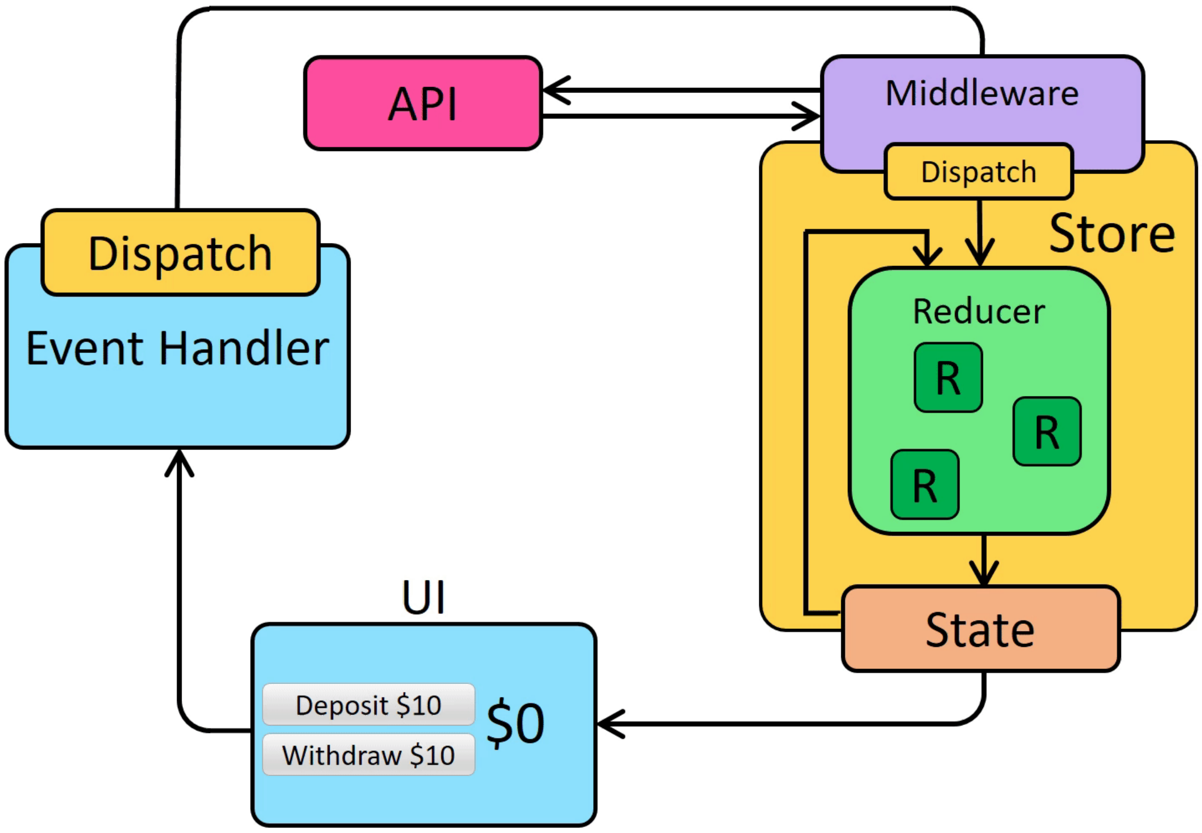
Why use getDefaultMiddleware().concat(thunk)?

- **Best Practices:** Redux Toolkit is designed to encourage best practices. By using getDefaultMiddleware, you benefit from the standard middleware that the library recommends for a typical Redux application.
- **Extensibility:** This pattern makes it easy to add other middleware to your store in the future. For example, if you wanted to add a logging middleware, you could simply do:

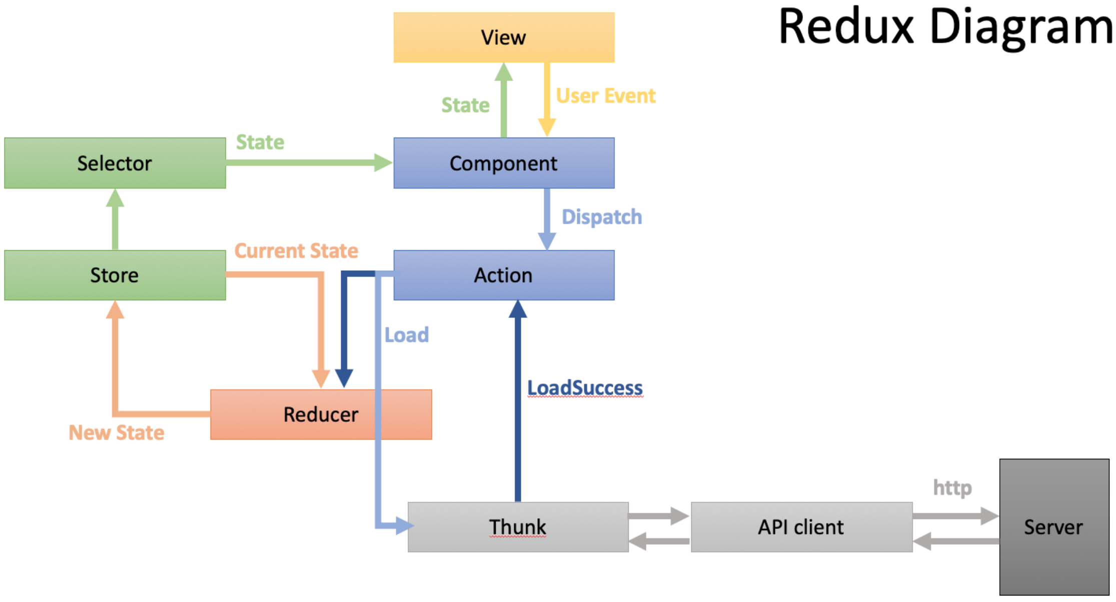
2. Providing the Redux Store - Vite Entry Point

`src/main.jsx`:

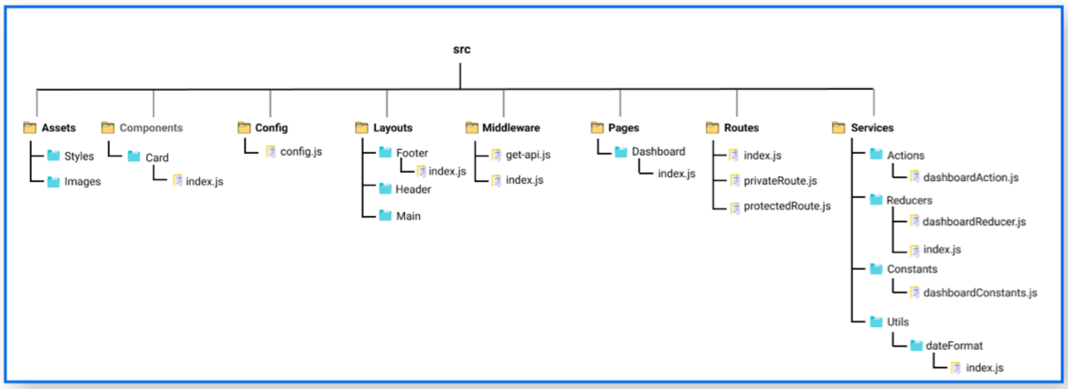
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { store } from './app/store';
import { Provider } from 'react-redux';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```



Redux Diagram



XENONSTACK



```
src
├─ Assets
│  ├─ Styles
│  └─ Images
├─ Components
│  └─ Card
│     └─ index.js
├─ Config
│  └─ config.js
├─ Layouts
│  ├─ Footer
│  │  └─ index.js
│  ├─ Header
│  └─ Main
├─ Middleware
│  ├─ get-api.js
│  └─ index.js
├─ Pages
│  └─ Dashboard
│     └─ index.js
├─ Routes
│  ├─ index.js
│  ├─ privateRoute.js
│  └─ protectedRoute.js
└─ Services
   ├─ Actions
   │  └─ dashboardAction.js
   ├─ Reducers
   │  ├─ dashboardReducer.js
   │  └─ index.js
   ├─ Constants
   │  └─ dashboardConstants.js
   └─ Utils
      └─ dateFormat
         └─ index.js
```