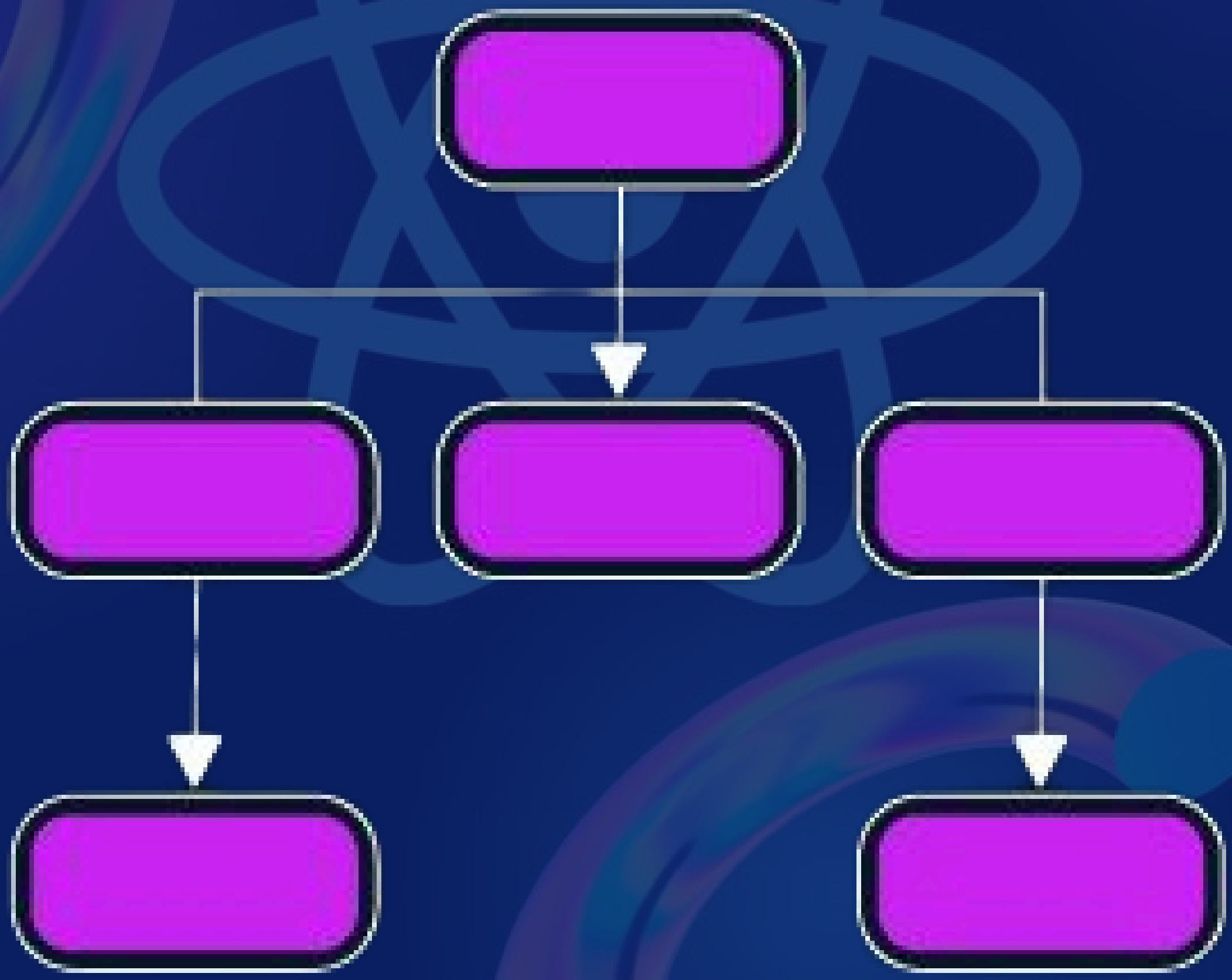
Day 22

Managing Side Effects In Redux



@oluwakemi Oluwadahunsi

What Are Side Effects In Redux?

In Redux, **side effects** refer to any operations that interact with the **outside world** or produce results that cannot be predicted by just looking at the function's input.

They **break the pure function rule** in Redux because they don't rely solely on the data provided to them.

Common side effects include:

- API calls (fetching or submitting data to a server)
- Timers (setTimeout, setInterval)
- Reading from local storage
- Logging

@oluwakemi Oluwadahunsi

For example, when fetching data from an API, you need to trigger an action once the data has been retrieved. This operation is asynchronous and relies on external factors (like network status), which makes it a **side effect**.

Redux, being a synchronous state management tool, doesn't natively handle asynchronous operations, so middleware is required to handle these side effects.

In Redux, side effects are often handled by **middleware** to keep the reducers pure and free of such operations.

The two most popular middleware libraries for handling side effects in Redux are **Redux Thunk** and **Redux Saga**.

@oluwakemi Oluwadahunsi

Redux Thunk

if you went through the last part of the Global State Management techniques (Part 3, Day 21), **redux thunk** shouldn't sound new to you.

Well, here is a broader explanation:

Redux Thunk is a **middleware** that allows you to write **action creators** that return a **function** instead of an **action**.

Inside this function, you can perform side effects like API calls, data fetching, or executing dispatch calls conditionally.

The function receives **dispatch** and **getState** as arguments, allowing you to interact with the Redux store.

@oluwakemi Oluwadahunsi

Key Features of Redux Thunk:

- Allows for delayed dispatching of actions
- Provides access to the current state (getState) to base logic or decisions.
- Perfect for handling simple asynchronous operations, such as fetching data or submitting forms.

When to Use Redux Thunk:

- For small-to-medium sized applications.
- When async operations are simple and straightforward.
- When you want to conditionally dispatch actions based on the current state or the result of an async operation.

To see an example of how Redux Thunk works, kindly check the examples in the Global State Management Techniques (Part 3, Day 21) of this series.

@oluwakemi Oluwadahunsi

Redux Saga

Redux Saga is another middleware that allows you to handle side effects in a more controlled manner.

Unlike **Redux Thunk**, which uses functions inside action creators, **Redux Saga** uses **generator functions (function*)** to yield effects.

It provides greater control over **complex async operations**, like orchestrating multiple API calls, retries, debouncing, or handling side effects in parallel.

Key Features of Redux Saga:

- Based on generator functions, making async logic easier to reason about.
- Can pause and resume tasks, which is great for handling complex async scenarios.
- Offers advanced functionality like canceling tasks, debouncing, throttling, and managing complex workflows.

@oluwakemi Oluwadahunsi

When to Use Redux Saga:

- For larger, more complex applications where advanced side effect management is necessary.
- When you need to handle complex async flows such as parallel tasks, retries, or complex error handling.
- When your app requires task cancellation, or you want to handle many concurrent API requests.

Example: Using Redux Saga for API Calls

Here's a step-by-step example of how to build a project that uses **Redux Saga** to manage state and fetch data from an API.

Step 1: Install Redux Saga

To begin, install the redux-saga package using npm or yarn.

```
1 npm install redux-saga
```

Step 2: Configure Redux Store with Saga Middleware

In this step, we set up the Redux store and integrate Saga middleware.

As we did in the last lesson, create a **store.js** file. Here's the code:

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import createSagaMiddleware from 'redux-saga';
import rootReducer from './reducers';
import rootSaga from './sagas';  // Root saga will be defined later

const sagaMiddleware = createSagaMiddleware();

const store = configureStore({
  reducer: rootReducer,
  middleware: [sagaMiddleware], // Integrating saga middleware into Redux store
});

sagaMiddleware.run(rootSaga); // Running the root saga

export default store;
```

In this file:

- **createSagaMiddleware():** This creates a sagaMiddleware instance, which will be added to Redux's middleware pipeline to handle asynchronous actions using generator functions.

- **configureStore():** This method from Redux Toolkit sets up the Redux store and accepts the root reducer (which combines all your reducers) and middleware (we are adding sagaMiddleware here).

- **sagaMiddleware.run(rootSaga):** This starts the root saga, which will handle the side effects like fetching data.

Step 3: Define Redux Saga for API Calls

Here, we define the actual **saga** that handles fetching data from an API asynchronously.

The **saga** listens for specific action types and performs the necessary side effect (API call).

@oluwakemi Oluwadahunsi

Create another file named **sagas.js**:

```
// sagas.js
import { call, put, takeEvery } from 'redux-saga/effects';

function* fetchDataSaga() {
  try {
    const response = yield call(fetch, 'https://jsonplaceholder.typicode.com/posts');
    const data = yield response.json();
    yield put({ type: 'FETCH_DATA_SUCCESS', payload: data });
  } catch (error) {
    yield put({ type: 'FETCH_DATA_FAILURE', error });
  }
}

export function* watchFetchData() {
  yield takeEvery('FETCH_DATA_REQUEST', fetchDataSaga);
}

// rootSaga.js
import { all } from 'redux-saga/effects';
import { watchFetchData } from './sagas';

export default function* rootSaga() {
  yield all([watchFetchData()]);
}
```

In this file:

- **call:** This effect is used to make asynchronous calls in Redux Saga. In this example, it calls the fetch API to retrieve posts from the JSONPlaceholder API. The **call** effect ensures that the **saga waits** for the API response before moving to the next step.

@oluwakemi Oluwadahunsi

- **put:** This is used to dispatch actions from within the saga. For example, after receiving the API response, we dispatch the **FETCH_DATA_SUCCESS** action along with the fetched data. If an error occurs, the **FETCH_DATA_FAILURE** action is dispatched.

- **takeEvery:** This is a Redux Saga effect that listens for every dispatched action of a specified type. In this case, it listens for the '**FETCH_DATA_REQUEST**' action, and every time this action is dispatched, it runs the fetchDataSaga to handle the API call.

- **watchFetchData:** This generator function watches for the **FETCH_DATA_REQUEST** action type and triggers the fetchDataSaga whenever the action is dispatched.

- **rootSaga:** This combines all the saga watchers. The all effect runs multiple sagas concurrently. In this example, there's only one watcher (**watchFetchData**), but this structure allows you to **add more sagas** easily in the future.

Step 4: Create Reducer for Data Handling

Reducers handle the state changes based on the actions dispatched. In this step, we define how the state should be updated when **FETCH_DATA_REQUEST**, **FETCH_DATA_SUCCESS**, or **FETCH_DATA_FAILURE** actions are dispatched.

Create a **reducer.js** file:

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

export const dataReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_DATA_REQUEST':
      return { ...state, loading: true };
    case 'FETCH_DATA_SUCCESS':
      return { ...state, loading: false, data: action.payload };
    case 'FETCH_DATA_FAILURE':
      return { ...state, loading: false, error: action.error };
    default:
      return state;
  }
};
```

stores the fetched data

track whether the data is currently being fetched.

Will store any error message if the data fetching fails.

In the file above:

1. dataReducer: This reducer listens for different action types and updates the state accordingly:

- **FETCH_DATA_REQUEST:** When this action is dispatched (indicating that data fetching is starting), the loading state is set to true.
- **FETCH_DATA_SUCCESS:** Once the data is successfully fetched, the loading state is set to false, and the fetched data is stored in the state.
- **FETCH_DATA_FAILURE:** If the API request fails, the error is updated with the error message, and loading is set to false.

Step 5: Dispatch Actions in Component

Finally, in the React component, we dispatch the action that triggers the Redux Saga to fetch the data.

We also retrieve the **current state** (loading, error, data) from the **Redux store** using the **useSelector hook**.

@oluwakemi Oluwadahunsi

Create a component to fetch the data:

```jsx
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';

const UserList = () => {
  const dispatch = useDispatch();
  const { data, loading, error } = useSelector(state => state.dataReducer);

  useEffect(() => {
    dispatch({ type: 'FETCH_DATA_REQUEST' }); // Dispatch the action to trigger the saga
  }, [dispatch]);

  if (loading) return <p>Loading ...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
   <div>
      <h1>Users List</h1>
      <div className="container">
        {data.map((user) => (
          <div key={user.id} className="wrapper">
            <img
              src={user.img}
              alt="user-image"
              title="user"
              className="avatar"
            />
            <h2>{user.name}</h2>
            <p>{user.role}</p>
            <p>Phone: {user.phone}</p>
            <address>
              Address: {user.address.suite}, {user.address.street},{" "}
              {user.address.city}.{" "}
            </address>
            <a href={`mailto:${user.email}`}>{user.email}</a>
            <p>Website: {user.website}</p>
          </div>
        ))}
      </div>
    </div>
  );
};
export default UserList;
```

@oluwakemi Oluwadahunsi

In the component above:

- **useDispatch:** This hook provides the dispatch function from Redux, allowing us to dispatch actions from within the component.
- **useSelector:** This hook allows us to access the current state from the Redux store. We are using it to access data, loading, and error from our dataReducer.
- **useEffect:** This hook is used to dispatch the FETCH_DATA_REQUEST action when the component mounts. This action triggers the fetchDataSaga, which handles the API call.
- **Conditional Rendering:**
  - If **loading is true**, we display a loading message.
  - If **error is present**, we display the error message.
  - Once the **data is successfully fetched**, we map through the data array and render it in a list.

@oluwakemi Oluwadahunsi

Tabular Differences

Features	Redux Thunk	Redux Saga
Complexity	Simpler, suitable for small-medium apps	More complex, ideal for larger apps
Async Handling	Basic async handling using promises	Advanced async handling using generator functions
Side Effect Control	Limited control over side effects	Fine-grained control over side effects
Performance	Suitable for simple tasks, can become messy with more complex tasks	Handles large-scale async flows better with non-blocking performance.
Type of Middleware	Handles async actions using functions.	Uses generator functions to manage side effects.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app