Agenda:

- Redux middleware
- Asynchronous data flow in Redux
- Redux-thunk

Async Logic and Redux Middleware

By itself, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling th root reducer function, and notify the UI that something has changed. Any asynchronicity has to happen outside the store.

Earlier, we said that Redux reducers must never contain "side effects". **A "side effect" is any change to state or behavior that can be seen outsid of returning a value from a function**. Some common kinds of side effects are things like:

- Logging a value to the console
- Saving a file
- Setting an async timer
- Making an AJAX HTTP request
- Modifying some state that exists outside of a function, or mutating arguments to a function
- Generating random numbers or unique random IDs (such as `Math.random()` or `Date.now()`

However, any real app will need to do these kinds of things *somewhere*. So, if we can't put side effects in reducers, where *can* we put them?

Redux middleware were designed to enable writing logic that has side effects.

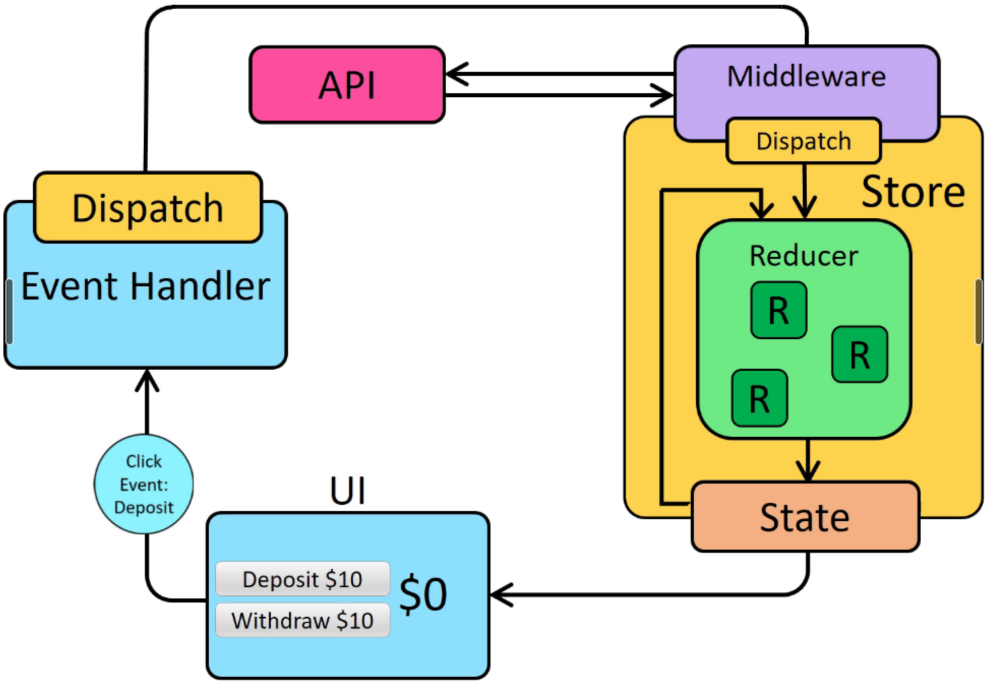
Redux-thunk

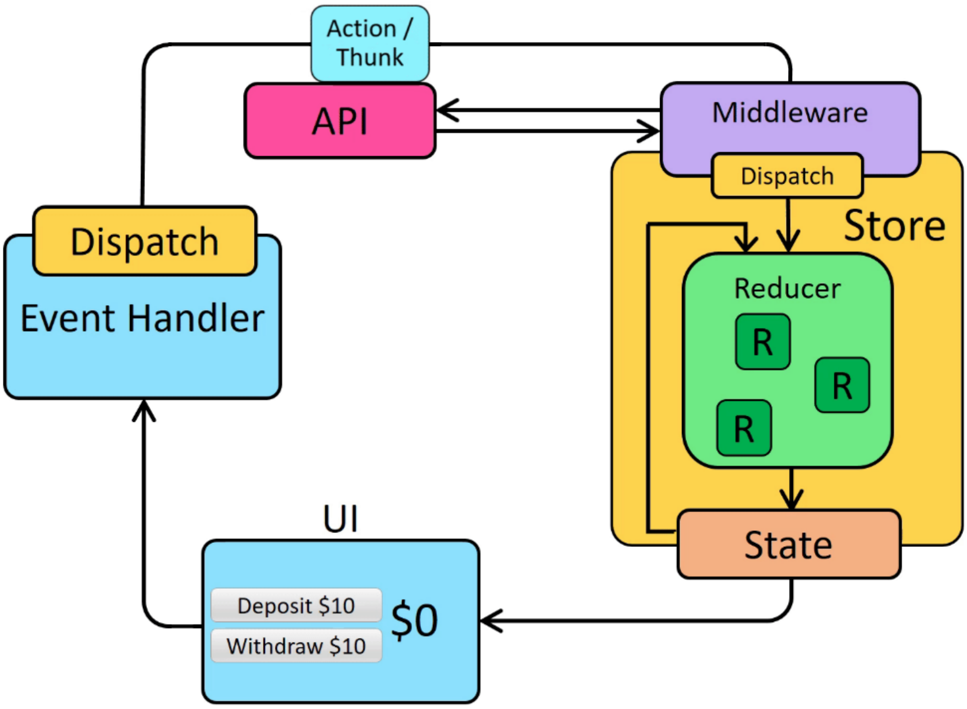
Thunk is a programming concept where a function is used to delay the evaluation/calculation of an operation. Here’s an example:

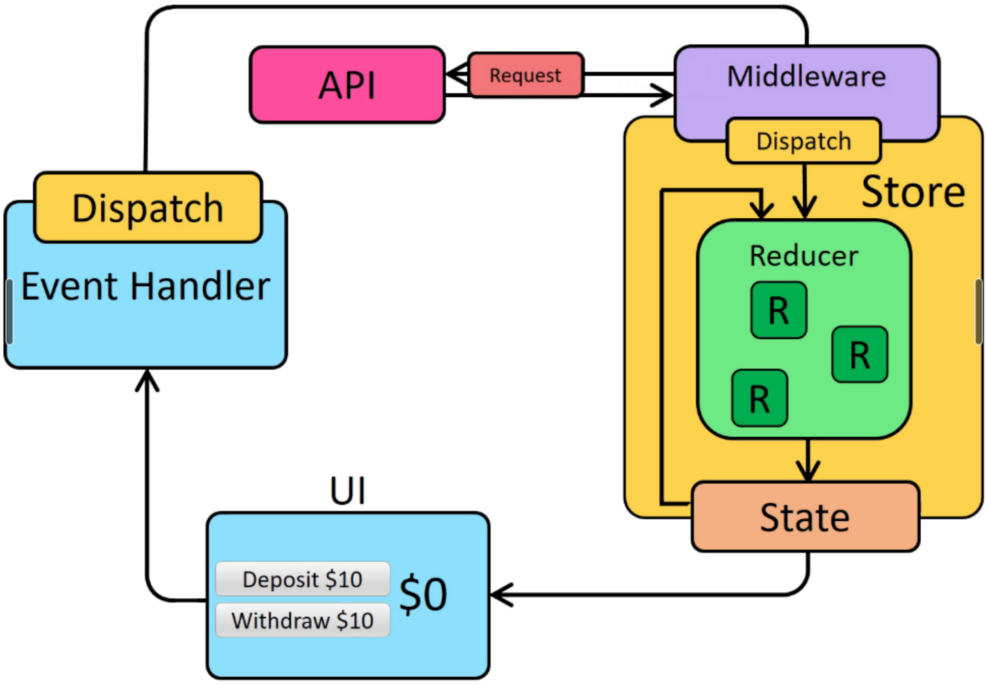
```
// calculation of 1 + 2 is immediate
// x === 3
let x = 1 + 2

// calculation of 1 + 2 is delayed
// foo can be called later to perform the calculation
// foo is a thunk!
let foo = () => 1 + 2
```

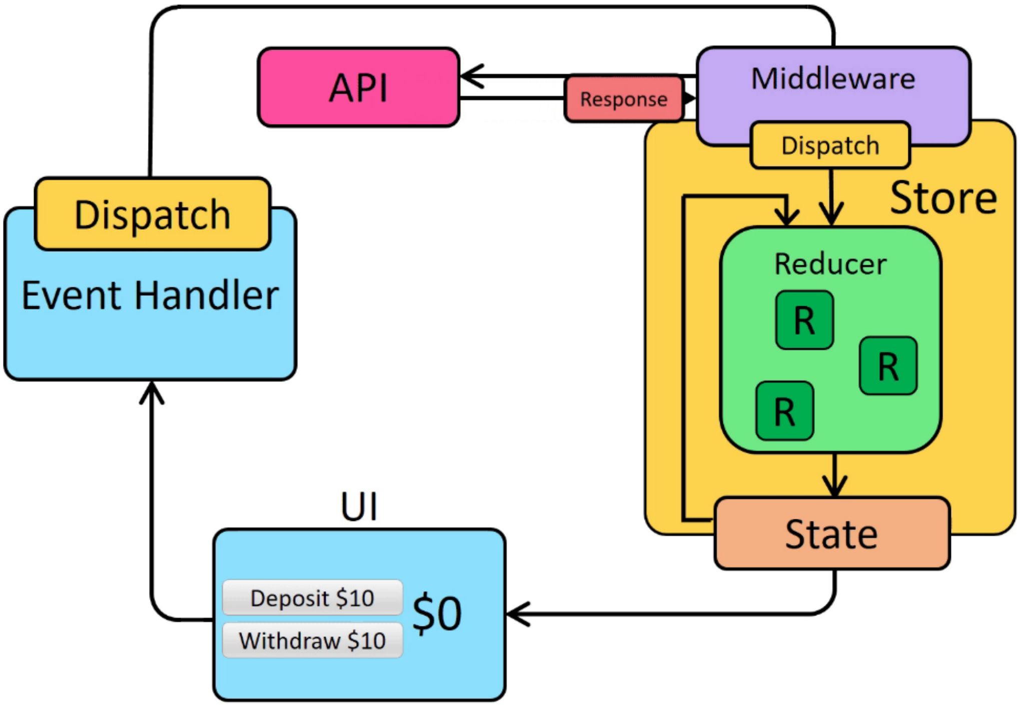
Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object. The returned function receives the store dispatch method. This means that the middleware can perform asynchronous operation and once they have been completed, it can dispatch regul synchronous actions inside the function’s body.

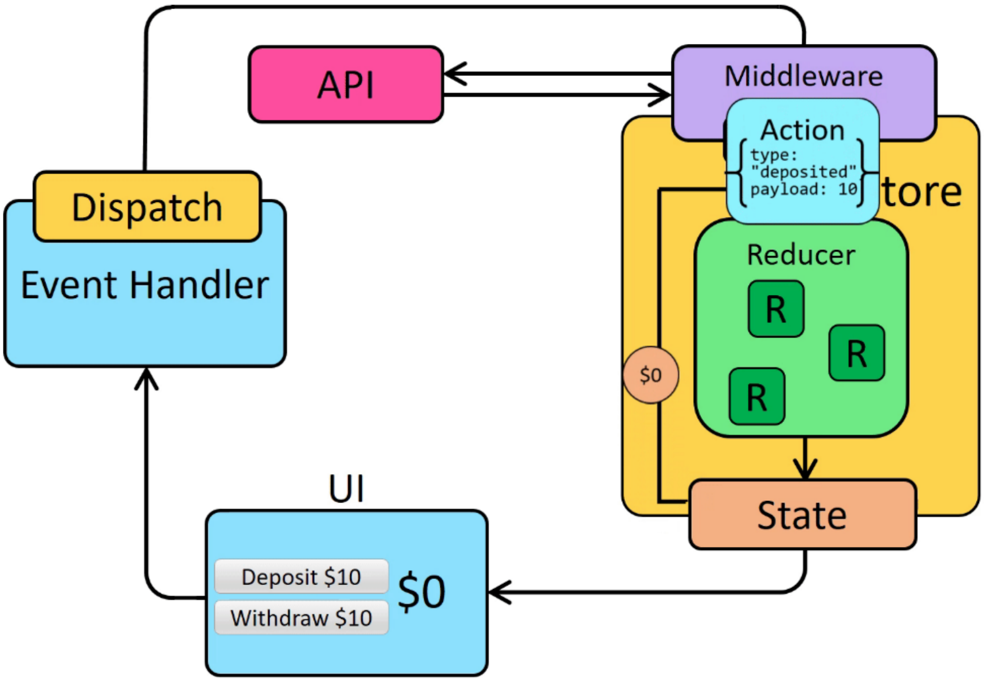
This makes the async data flow look like:


API
Middleware
Dispatch
Store
Reducer
R
R
R
State
Dispatch
Event Handler
Click Event: Deposit
UI
Deposit $10
Withdraw $10
$0


Action / Thunk
API
Middleware
Dispatch
Store
Reducer
R
R
R
State
Dispatch
Event Handler
UI
Deposit $10
Withdraw $10
$0


API
Request
Middleware
Dispatch
Store
Dispatch
Event Handler
Reducer
R
R
R
State
UI
Deposit $10
Withdraw $10
$0


API
Response
Middleware
Dispatch
Store
Dispatch
Event Handler
Reducer
R
R
R
State
UI
Deposit $10
Withdraw $10
$0



Application

Now, let's utilize what we've learned about Asynchronicity in Redux so far and fetch some real-time data for the E-Commerce application we built previous module. We'll be making calls to the Fake Store API for this data.

Start by opening the root directory of our application in terminal and run the following command to install `redux-thunk` :

```
npm install redux-thunk
```

Now open `./src/store/store.js` in text editor. Here we will pass in `thunk` from the `redux-thunk` library as a middleware for our app. Th allows us to perform the required asynchronous tasks, like API calls, before an action is dispatched. This is the code that `store.js` should now have:

```
import { createStore, applyMiddleware, compose } from 'redux'
import thunk from 'redux-thunk'
import rootReducer from '../reducers'

export const store = createStore(
  rootReducer,
  compose(applyMiddleware(thunk))
)
```

Go through the following links for more info about the in-built library functions used here :

- `**[applyMiddleware(...middleware)](https://redux.js.org/api/applymiddleware)**`
- `**[compose(reducer, [preloadedState], [enhancer])](https://redux.js.org/api/compose)**`

Fetching Products

Let's begin by deleting `data.json` from source directory since it isn't needed anymore. Now go to `<Products />` component and instead of passir in the hard-coded data, create an async function `fetchProducts()` that fetches data from Fake Store API. Here's what the function looks like:

```
const fetchProducts = async () => {
    const response = await fetch('https://fakestoreapi.com/products');
    let data = await response.json();
    return data;
};
```

Next, turn the `loadProducts()` function to an asynchronous function and pass in the data received from API to the `setProducts` action.

Here's the complete code for the component:

```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import ProductCard from './ProductCard';
import { setProducts } from '../../state/actions/products';

const Products = () => {
  const products = useSelector((state) => state.products);
  const dispatch = useDispatch();

  useEffect(() => {
    loadProducts();
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  const loadProducts = async () => {
    dispatch(setProducts(filterProducts(await fetchProducts())));
  };

  const fetchProducts = async () => {
    const response = await fetch('https://fakestoreapi.com/products');
    let data = await response.json();
    return data;
  };

  const filterProducts = (products) => {
    return products.filter(
      (product) =>
        product.category === `men's clothing` ||
        product.category === `women's clothing` ||
        product.category === `jewelery`
    );
  };

  const productCards = products.map((product) => (
    <ProductCard
      key={Math.random()}
      id={product.id}
      title={product.title}
      price={product.price}
      image={product.image}
    />
  ));

  return (
    <div className="grid grid-cols-1 gap-16 justify-center mt-16 md:grid-cols-2 lg:grid-cols-3">
      {productCards}
    </div>
  );
};

export default Products;
```

Fetching the data from an API takes some time, and while our application waits for the API to send back the data we have nothing to display on scree
You may notice the application looks like this for some time before it is filled in with products :

The same will happen if the request to API isn’t successful. Hence, to improve the User Experience, let’s add in a simple loading spinner and an err message. Since this directly affects the UI, this is another piece of state that we’ll be managing with Redux.

Creating Loading Spinner

The states for making the call to API can be defined as `loading` , `error` , and `success` . Let’s start by defining the changing of these state as `actionTypes` . Add the following lines to `./src/constants/actionTypes.js` :

```
export const SET_LOADING = 'SET_LOADING';
export const SET_ERROR = 'SET_ERROR';
export const SET_SUCCESS = 'SET_SUCCESS';
```

Create `action creators` for this inside `./src/state/actions/loadingStates.js` .

```
import {
  SET_ERROR,
  SET_LOADING,
  SET_SUCCESS,
} from '../../constants/actionTypes';

const setError = (val) => {
  return {
    type: SET_ERROR,
    payload: val,
  };
};

const setSuccess = (val) => {
  return {
    type: SET_SUCCESS,
    payload: val,
  };
};

const setLoading = (val) => {
  return {
    type: SET_LOADING,
    payload: val,
  };
};

export { setError, setSuccess, setLoading };
```

Next, lets create the `reducers` inside `./src/state/reducers/loadingStates.js` .

```
import {
  SET_ERROR,
  SET_LOADING,
  SET_SUCCESS,
} from '../../constants/actionTypes';

const INIT_STATE = {
  loading: false,
  error: false,
  success: false,
};

const loadingStatesReducer = (state = INIT_STATE, action) => {
  switch (action.type) {
    case SET_ERROR:
      return { ...state, error: action.payload };

    case SET_LOADING:
      return { ...state, loading: action.payload };

    case SET_SUCCESS:
      return { ...state, success: action.payload };

    default:
      return state;
  }
};

export default loadingStatesReducer;
```

Import this `reducer` inside `./src/state/reducers/index.js` , and add the `loadingStatesReducer` to the `rootReducer` .

```
import isCartOpenReducer from './isCartOpen';
import productsReducer from './products';
import cartReducer from './cart';
import loadingStatesReducer from './loadingStates';
import { combineReducers } from 'redux';

const rootReducer = combineReducers({
  isCartOpen: isCartOpenReducer,
  products: productsReducer,
  cart: cartReducer,
  loadingState: loadingStatesReducer,
});

export default rootReducer;
```

This is how easy it can be to provide a new piece of state to our entire application with Redux. It especially comes in handy when developing ar managing large applications.

Now let's create an animated `Loading Spinner` to be displayed while we wait to receive the data.

Create `Loader` inside `./src/components` and add the following code:

```
import React from 'react';

function Loader() {
  return (
    <div className="flex justify-center items-center">
      <button type="button" className="" disabled>
        <svg
          role="status"
          className="w-16 h-16 mr-2 text-slate-700 animate-spin dark:text-gray-600 fill-[#46ffd3]"
          viewBox="0 0 100 101"
          fill="none"
          xmlns="http://www.w3.org/2000/svg"
        >
          <path
            d="M100 50.5908C100 78.2051 77.6142 100.591 50 100.591C22.3858 100.591 0 78.2051 0 50.5908C0 22.9766 22.3858
            fill="currentColor"
          />
          <path
            d="M93.9676 39.0409C96.393 38.4038 97.8624 35.9116 97.0079 33.5539C95.2932 28.8227 92.871 24.3692 89.8167 20
            fill="currentFill"
          />
        </svg>
      </button>
    </div>
  );
}

export default Loader;
```

Now, import this inside `<Products />` component and conditionally render it while the products are being fetched. To track this, we first s the `loading` state as `true` as soon as the component mounts and makes the request to API.

```
useEffect(() => {
    dispatch(setLoading(true));
    loadProducts();
  }, []);
```

Then, if the request resolves in an error, the `loading` state is set as `false` and the `error` state is set as `true`. If the expected response received from the API, the `loading` state is set as `false` and the `success` state is set as `true`. We'll then conditionally render the product loading spinner or suitable error message based on this state.

This is how it's being implemented:

```
import React, { useEffect } from 'react';
import Loader from '../Loader';
import { useDispatch, useSelector } from 'react-redux';
import ProductCard from './ProductCard';
import { setProducts } from '../../state/actions/products';
import {
  setError,
  setSuccess,
  setLoading,
} from '../../state/actions/loadingStates';

const Products = () => {
  const products = useSelector((state) => state.products);
  const loadingState = useSelector((state) => state.loadingState);
  const dispatch = useDispatch();

  useEffect(() => {
    dispatch(setLoading(true));
    loadProducts();
  }, []);

  const loadProducts = async () => {
    dispatch(setProducts(filterProducts(await fetchProducts())));
  };

  const fetchProducts = async () => {
    try {
      const response = await fetch(
        'https://fakestoreapi.com/products'
      );
      let data = await response.json();
      dispatch(setLoading(false));
      dispatch(setSuccess(true));
      return data;
    } catch (err) {
      dispatch(setLoading(false));
      dispatch(setError(true));
    }
  };

  const filterProducts = (products) => {
    return products.filter(
      (product) =>
        product.category === `men's clothing` ||
        product.category === `women's clothing` ||
        product.category === `jewelery`
    );
  };

  const productCards = products.map((product) => (
    <ProductCard
      key={Math.random()}
      id={product.id}
      title={product.title}
      price={product.price}
      image={product.image}
    />
  ));

  if (loadingState.loading) {
    return <Loader />;
  }

  if (loadingState.error) {
    return <div>Sorry, We're facing an error</div>;
  }

  return (
    <div className="grid grid-cols-1 gap-16 justify-center mt-16 md:grid-cols-2 lg:grid-cols-3">
      {productCards}
    </div>
  );
};

export default Products;
```

Conclusion

With this, we’ve completed a sample E-Commerce project using `Redux` and `Redux-thunk` and learned about the synchronous and asynchronou data flow in `Redux` .