Agenda:

- Store
- Reducer
- Action
- Dispatching Actions

Context API makes it easier to share data between components and eliminates prop-drilling. But as the size of application grows, and the componen not only demand the data, but also need to modify and manage state, Context API falls quite short of keeping the codebase maintainable and organized

Redux

Redux is a state-management library, that does not just centralize the state but can also manage it. It is especially helpful over Context API when:

- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people

To understand how Redux manages state, it’s important to understand the building blocks of Redux:

1. Store
2. Reducer
3. Action

Store

Store is a single, immutable JavaScript object which Redux uses the store the entire state of an application. It is the single source of state, and accessible by all parts of the UI. Store can properties like list of items in a user’s cart and so on. The content of the Store is entirely up to us as Redu has no opinion about it. We can only use plain JS such as Arrays, Objects, Numbers, Strings, Booleans, etc to describe the state of our application. Th means you may not put other things into the Redux state - no class instances, built-in JS types like Map / Set Promise / Date, functions, or anything els that is not plain JS data. We start by defining an initial state ****value to describe the application:

```
// Define an initial state value for the app
const initialState = {
  value: 0
}
```

Now, since the Store is immutable, this means that we cannot directly modify or mutate this store. So to update it we need a function that takes the sto as an argument, and returns the updated value of the Store (We can use the spread operator to create a copy of the store inside this function).

Reducer

The reducer receives two arguments, the current `state` and an `action` object describing what happened. Based on the action object, Reducer ca run complex logic or calculations, and return the updated store.

Reducers must *always* follow some specific rules:

- They should only calculate the new state value based on the `state` and `action` arguments
- They are not allowed to modify the existing `state` . Instead, they must make *immutable updates*, by copying the existing `state` and making changes to the copied values.
- They must not do any asynchronous logic, calculate random values, or cause other "side effects"

The logic inside reducer functions typically follows the same series of steps:

- Check to see if the reducer cares about this action
- If so, make a copy of the state, update the copy with new values, and return it
- Otherwise, return the existing state unchanged

Action

Now, when a part of our application wants to update state, it'll need to call the Reducer. This is done through plain JS objects called Actions. Actions a plain JavaScript objects that have a `type` field. You can think of an action as an event that describes something that happened in the application. W normally put any extra data needed to describe what's happening into the action.payload field. This could be a number, a string, or an object with multip fields inside.

The Redux store doesn't care what the actual text of the action.type field is. However, your own code will look at action.type to see if an update needed. So, try to choose action types that are readable and clearly describe what's happening - it'll be much easier to understand things when you loo at them later!

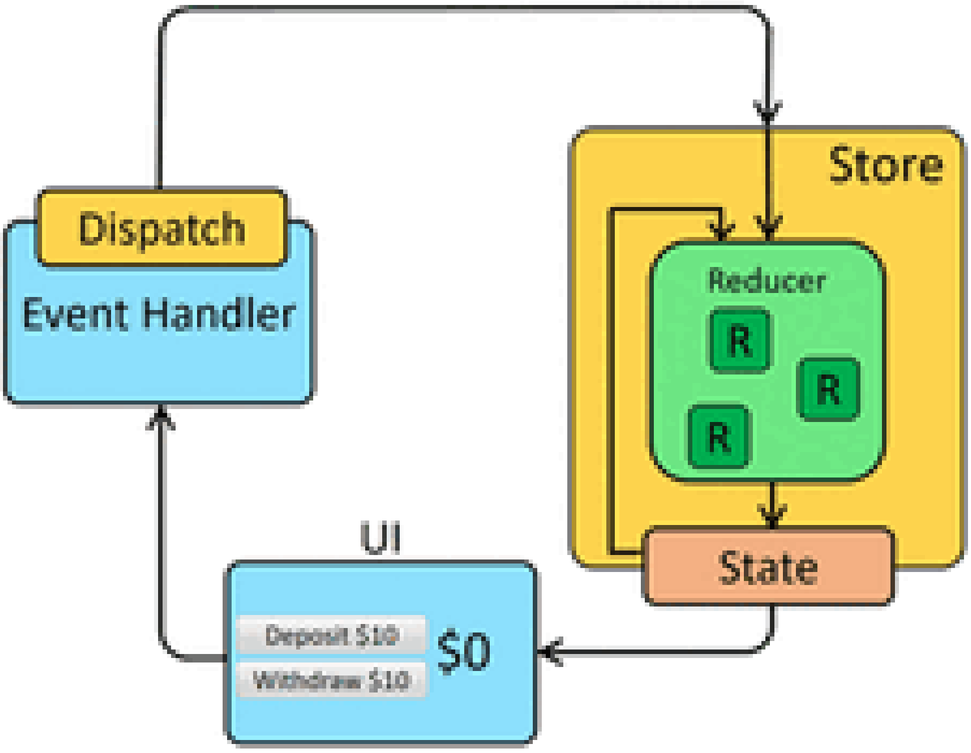
An example of an action could be :

```
const action = {
    type: SET_PRODUCTS,
    payload: products,
}
```

Dispatching Actions

The Redux store has a method called `dispatch`. **The only way to update the state is to call `store.dispatch()` and pass in an actic object**. **You can think of dispatching actions as "triggering an event"** in the application. Something happened, and we want the store to know abo it. Reducers act like event listeners, and when they hear an action they are interested in, they update the state in response.

So, the basic flow of a Redux application can be summarized as :



Application

Now that we have a basic understanding of Redux, let's utilize it in an application to deepen our understanding. We'll be building a simple E-commerc web page where users can add and remove products from Cart using :

- ReactJS
- Redux
- TailwindCSS

To begin, run the following command to create a new react app :

```
npx create-react-app redux-shop
```

`cd` into root directory of app and run following commands to initialize tailwindCSS in the app :

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

This should generate `tailwind.config.js` and `postcss.config.js` . Open `tailwind.config.js` and add the paths of all template files. should look like this :

```
module.exports = {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

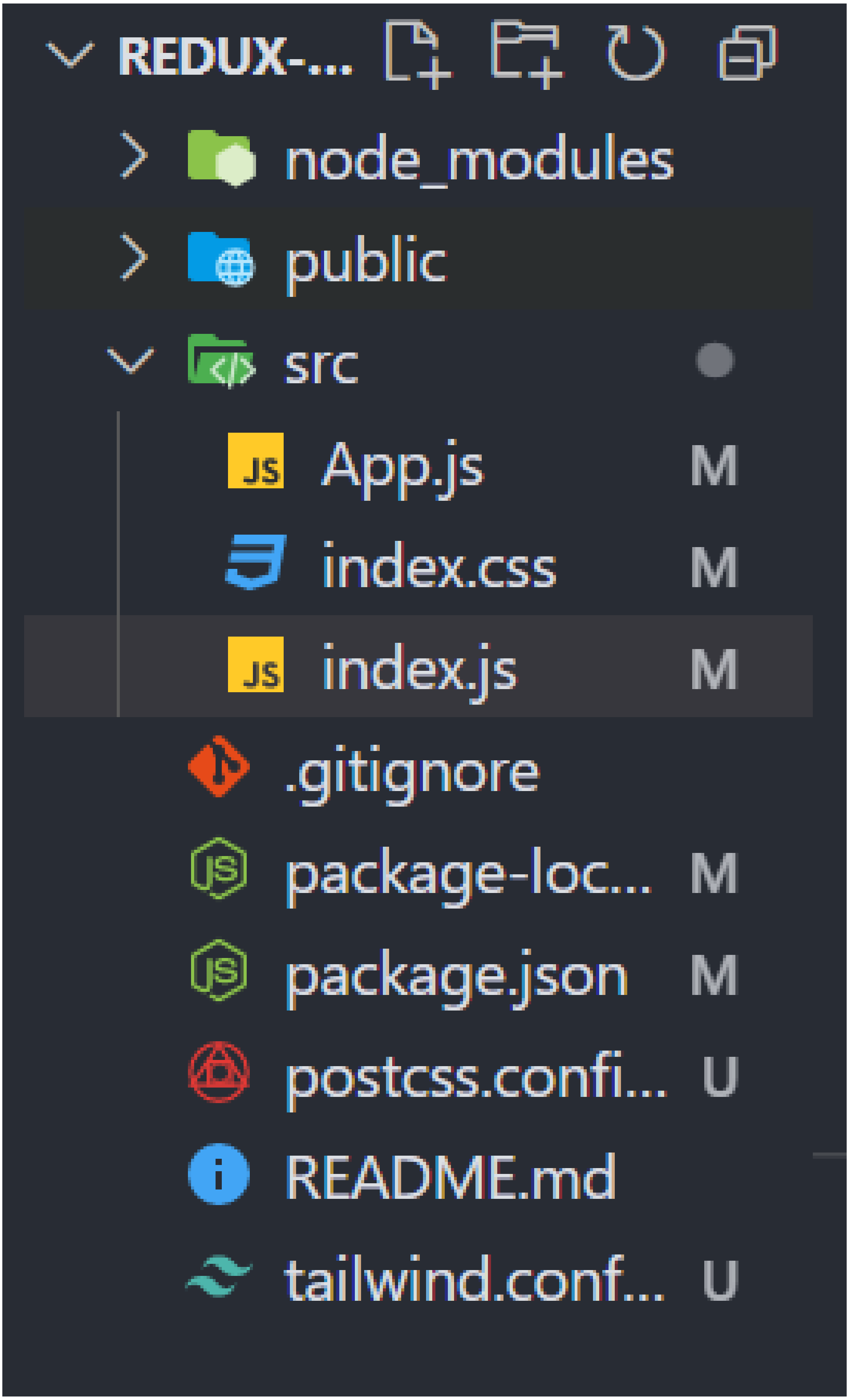
Add tailwind directives to `./src/index.css` . It should look like :

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Run the build process with :

```
npm run start
```

After deleting unnecessary files, folder structure should look like this:



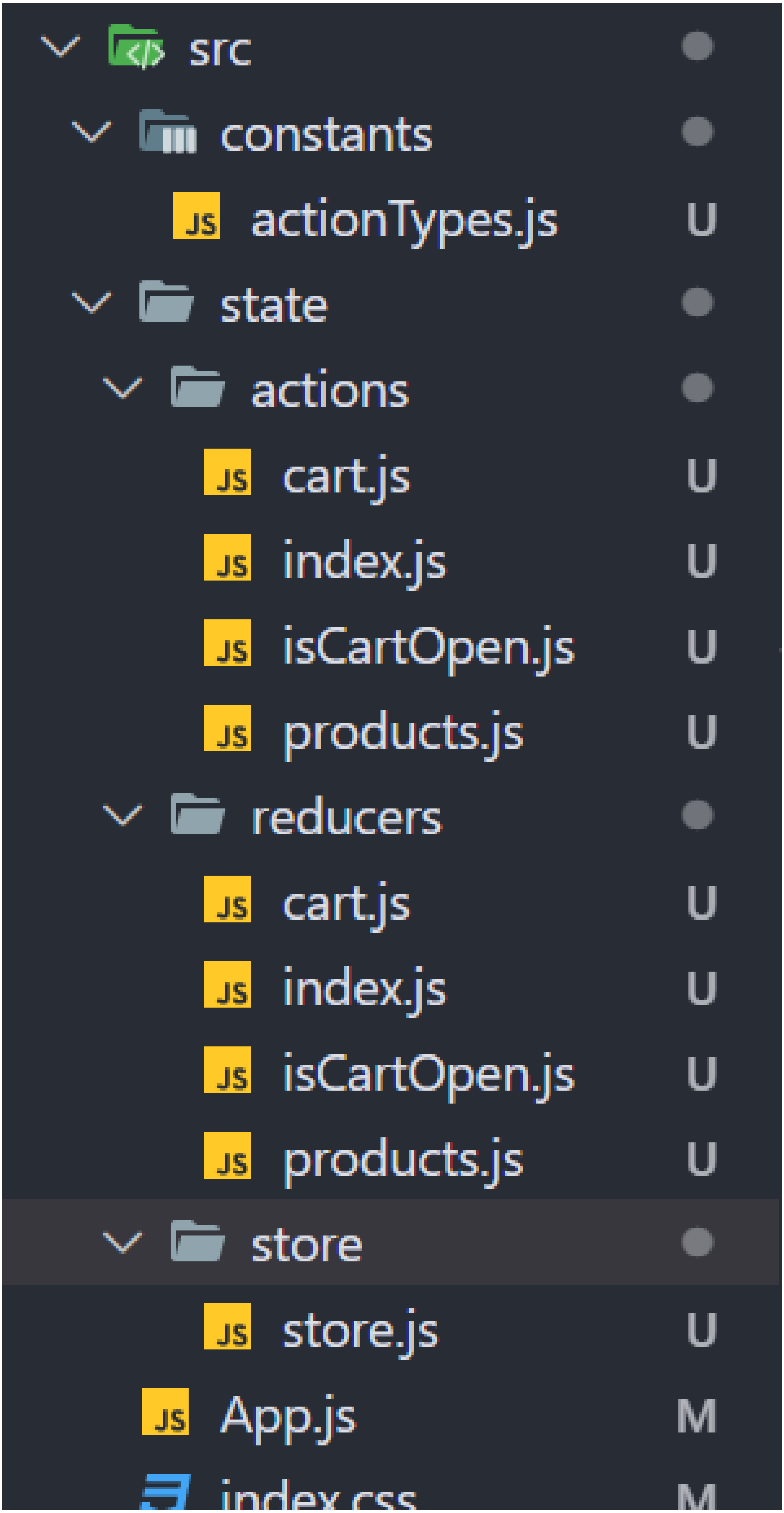
```
REDUX-...
  node_modules
  public
  src
    App.js          M
    index.css       M
    index.js        M
  .gitignore
  package-loc...    M
  package.json      M
  postcss.confi...  U
  README.md
  tailwind.conf...  U
```

Now run the following commands in terminal to install Redux and Redux-thunk:

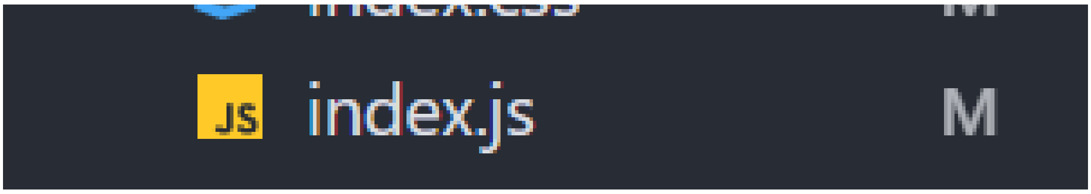
```
npm install redux react-redux
```

Create a folder `state` inside `./src` . This is where we'll manage our entire application's stat
Inside `state` create `actions` , `reducers` and `state` subfolders and the required files inside each folder. Also, create th

file `./constants/actionTypes.js` . This is where we'll define types of actions for state management. The structure of `./src` folder should look lik this now:



```
∨ src                    ●
  ∨ constants            ●
      actionTypes.js     U
  ∨ state                ●
    ∨ actions            ●
        cart.js          U
        index.js         U
        isCartOpen.js    U
        products.js      U
    ∨ reducers           ●
        cart.js          U
        index.js         U
        isCartOpen.js    U
        products.js      U
    ∨ store              ●
        store.js         U
    App.js               M
    index.css            M
```



Define and export the types of actions inside `./src/constants/actionTypes.js` :

```
export const OPEN_CART = 'OPEN_CART'
export const CLOSE_CART = 'CLOSE_CART'
export const SET_PRODUCTS = 'SET_PRODUCTS'
export const ADD_TO_CART = 'ADD_TO_CART'
export const REMOVE_FROM_CART = 'REMOVE_FROM_CART'
```

Creating Actions

Create `action creators` inside `./src/state/actions` . The content of files should look like following:

`./src/state/actions/cart.js` :

```
import { ADD_TO_CART, REMOVE_FROM_CART } from '../../constants/actionTypes'

const addToCart = (product) => {
  return {
    type: ADD_TO_CART,
    payload: product,
  }
}

const removeFromCart = (product) => {
  return {
    type: REMOVE_FROM_CART,
    payload: product,
  }
}

export { addToCart, removeFromCart }
```

`./src/state/actions/isCartOpen.js` :

```
import { OPEN_CART, CLOSE_CART } from '../../constants/actionTypes'

const openCart = () => {
  return {
    type: OPEN_CART,
  }
}

const closeCart = () => {
  return {
    type: CLOSE_CART,
  }
}

export { openCart, closeCart }
```

`./src/state/actions/products.js` :

```
import { SET_PRODUCTS } from '../../constants/actionTypes'

const setProducts = (products) => {
  return {
    type: SET_PRODUCTS,
    payload: products,
  }
}

export { setProducts }
```

Create `index.js` inside `./src/state/actions` to export all action creators inside the folder. Its content should be :

```
import { openCart, closeCart } from './isCartOpen'
import { addToCart, removeFromCart } from './cart'
import { setProducts } from './products'

export { openCart, closeCart, addToCart, removeFromCart, setProducts }
```

Creating Reducers

Next, lets create the `reducers` inside `./src/state/reducers`. The content for these is as follows :

`./src/state/reducers/cart.js` :

```
import {
  ADD_TO_CART,
  REMOVE_FROM_CART,
} from '../../constants/actionTypes';

const INIT_STATE = [];

const cartReducer = (state = INIT_STATE, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      const isInCart = state.find(
        (cartItem) => cartItem.id === action.payload.id
      );
      return isInCart
        ? state.map((cartItem) =>
            cartItem.id === action.payload.id
              ? { ...cartItem, quantity: cartItem.quantity + 1 }
              : cartItem
          )
        : [...state, { ...action.payload, quantity: 1 }];

    case REMOVE_FROM_CART:
      const cartItemToRemove = state.find(
        (cartItem) => cartItem.id === action.payload.id
      );
      return cartItemToRemove.quantity === 1
        ? state.filter(
            (cartItem) => cartItem.id !== action.payload.id
          )
        : state.map((cartItem) =>
            cartItem.id === action.payload.id
              ? { ...cartItem, quantity: cartItem.quantity - 1 }
              : cartItem
          );

    default:
      return state;
  }
};

export default cartReducer;
```

`./src/state/reducers/isCartOpen.js` :

```
import { OPEN_CART, CLOSE_CART } from '../../constants/actionTypes';

const INIT_STATE = false;

const isCartOpenReducer = (state = INIT_STATE, action) => {
  switch (action.type) {
    case OPEN_CART:
      return true;

    case CLOSE_CART:
      return false;

    default:
      return state;
  }
};

export default isCartOpenReducer;
```

`./src/state/reducers/products.js` :

```
import { SET_PRODUCTS } from '../../constants/actionTypes';

const INIT_STATE = [];

const productsReducer = (state = INIT_STATE, action) => {
  switch (action.type) {
    case SET_PRODUCTS:
      return [...action.payload];

    default:
      return state;
  }
};

export default productsReducer;
```

Create `index.js` inside `./src/state/reducers` to combine all the reducers and export them as the `rootReducer` . It looks like this:

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

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

export default rootReducer
```

Create Store

Now, we'll create the store for Redux. For this, create `store.js` inside `./src/store` . Here we will pass in `thunk` from the `redux-thunk` libra as a middleware for our app. This allows us to perform the required asynchronous tasks, like API calls, before an action is dispatched. These are th contents of `store.js` :

```
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 :

- `**[createStore(...functions)](https://redux.js.org/api/createstore)**`

Connecting React app to store

The `<Provider>` component makes the Redux `store` available to any nested components that need to access the Redux store.

Since any React component in a React Redux app can be connected to the store, most applications will render a `<Provider>` at the top level, with tl entire app's component tree inside of it.

Since the top level component for our application is `<App />`, we'll wrap this with the `<Provider>` component inside `./src/index.js`. As of nov this is what `index.js` looks like:

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

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

The `[Hooks](https://react-redux.js.org/api/hooks)` and `[Connect](https://react-redux.js.org/api/connect)` APIs can now acces the store instance.

Since we're using `React Hooks` in our current app, let's go throug the Hooks provided by Redux. And to learn about `Connect` API, follow this link.

useSelector()

Allows you to extract data from the Redux store state, using a selector function. The selector will be run whenever the function compone renders. `useSelector()` will also subscribe to the Redux store, and run your selector whenever an action is dispatched.

Basic Usage :

```
import React from 'react'
import { useSelector } from 'react-redux'

export const Component = () => {
  const isCartOpen = useSelector((state) => state.isCartOpen)
  return <div>{isCartOpen}</div>
}
```

useDispatch()

This hook returns a reference to the `dispatch` function from the Redux store. You may use it to dispatch actions as needed.

Basic Usage :

```
import React from 'react'
import { useDispatch } from 'react-redux'
import { closeCart } from '../../state/actions'

export const Component = () => {
  const dispatch = useDispatch()
  return (
    <div>
      <button onClick={() => dispatch(closeCart())}>
        Close Cart
      </button>
    </div>
  )
}
```

Now that we have set up the Redux store and connected it with React, let's work on creating the E-Commerce UI. For this we'll be usir the `Hooks` API, as discussed previously, and `TailwindCSS` for styling.

TailwindCSS

Tailwind CSS is a utility-first CSS (Cascading Style Sheets) framework with predefined classes that you can use to build and design web pages directly your markup. It lets you write CSS in your HTML in the form of predefined classes.

What is a Utility-First framework?

When we say utility-first CSS, we refer to classes in our markup (HTML) with predefined functionalities. This implies that you only have to write a clas with predefined styles attached to it, and those styles will be applied to the element.

In a case where you are working with vanilla CSS (CSS without any framework or library), you would first give your element a class name and the attach different properties and values to that class, which will, in turn, apply styling to your element.

We'll show you an example. Here, we'll create a button with rounded corners and a text that says "Click me."

We'll first do this using vanilla CSS, and then using utility classes available in Tailwind CSS.

With Vanilla CSS

```
<button class="btn">Click me</button>
```

We've given button tags the class `btn` , which will be styled using an external stylesheet. That is:

```
.btn {
  background-color: #000;
  color: #fff;
  padding: 8px;
  border: none;
  border-radius: 4px;
}
```

With Tailwind CSS

```
<button class="bg-black p-2 rounded">Click me</button>
```

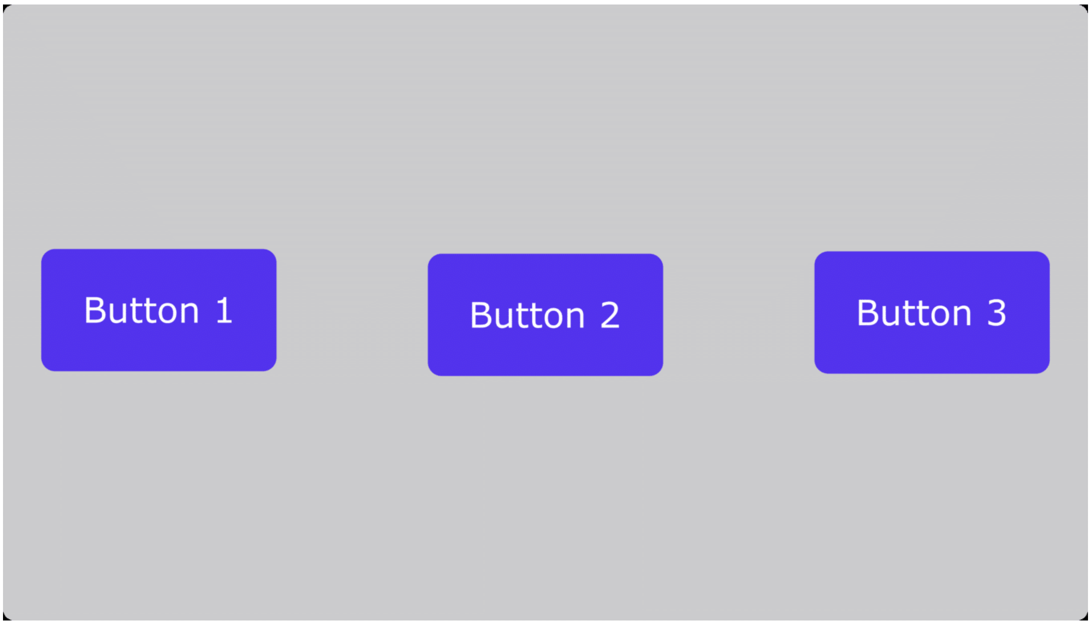
This is all required to achieve the same effect as the example with vanilla CSS. No external stylesheet where you have to write the styles was create because each class name we used already has a predefined style.

Using TailwindCSS

Since we've already installed and configured tailwindCSS, let's take a look at some examples to fully understand how it works.

Flexbox Example To use flex in Tailwind CSS, you need to add a class of flex and then the direction of the flex items:

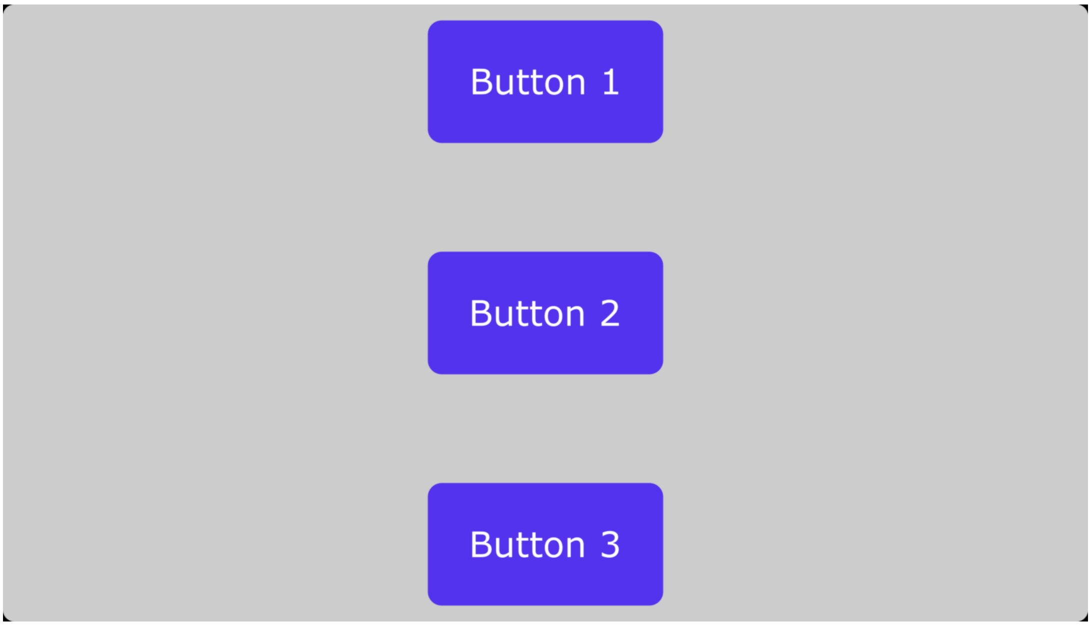
```
<div class="flex flex-row">
      <button> Button 1 </button>
      <button> Button 2 </button>
      <button> Button 3 </button>
</div>
```



Using `flex-row-reverse` will reverse the order in which the buttons appear.

`flex-col` stacks them above each other. Here is an example:

```
<div class="flex flex-col">
      <button> Button 1 </button>
      <button> Button 2 </button>
      <button> Button 3 </button>
</div>
```

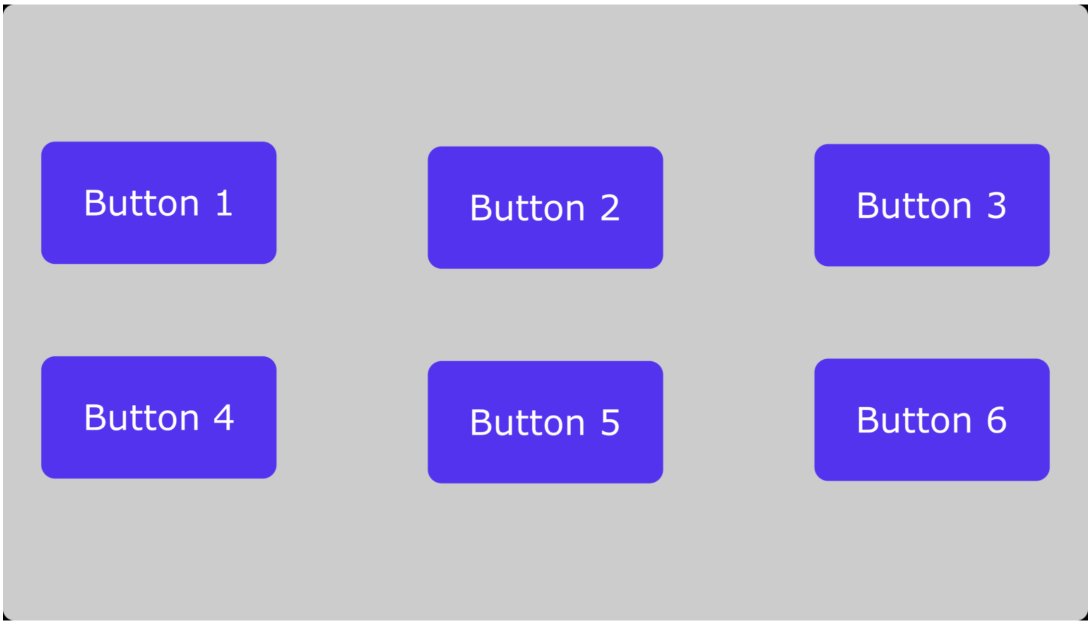


Just like the previous example, `flex-col-reverse` reverses the order.

Grid Example

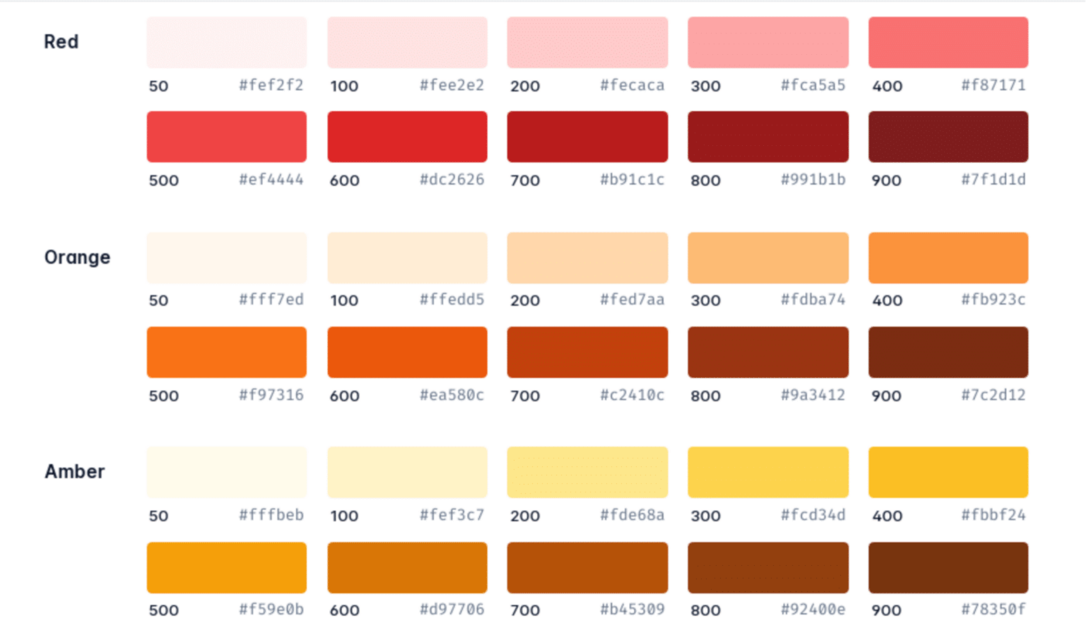
When specifying columns and rows in the grid system, we take a different approach by passing in a number that will determine how the elements w occupy available space:

```
<div class="grid grid-cols-3">
      <button> Button 1 </button>
      <button> Button 2 </button>
      <button> Button 3 </button>
      <button> Button 4 </button>
      <button> Button 5 </button>
      <button> Button 6 </button>
</div>
```



Colors

Tailwind comes with a lot of predefined colors. Each color has a set of different variations, with the lightest variation being 50 and the darkest being 900. Here is a picture of multiple colors and their HTML hex codes to illustrate this



Red

50 #fef2f2	100 #fee2e2	200 #fecaca	300 #fca5a5	400 #f87171
500 #ef4444	600 #dc2626	700 #b91c1c	800 #991b1b	900 #7f1d1d

Orange

50 #fff7ed	100 #ffedd5	200 #fed7aa	300 #fdba74	400 #fb923c
500 #f97316	600 #ea580c	700 #c2410c	800 #9a3412	900 #7c2d12

Amber

50 #fffbeb	100 #fef3c7	200 #fde68a	300 #fcd34d	400 #fbbf24
500 #f59e0b	600 #d97706	700 #b45309	800 #92400e	900 #78350f

We'll give an example of how you can do this using the red color above to give a button element a background color:

```
<button class="bg-red-50">Click me</button>
<button class="bg-red-100">Click me</button>
<button class="bg-red-200">Click me</button>
<button class="bg-red-300">Click me</button>
<button class="bg-red-400">Click me</button>
<button class="bg-red-500">Click me</button>
<button class="bg-red-600">Click me</button>
<button class="bg-red-700">Click me</button>
```

```
<button class="bg-red-800">Click me</button>
<button class="bg-red-900">Click me</button>
```

This syntax is the same for all colors in Tailwind except for black and white, which are written the same way but without the use of numbers: `bg` `black` and `bg-white` .

To add text color, you use the class `text-{color}`

Padding

Tailwind already has a design system that would help you keep a consistent scale across your designs. All you have to know is the syntax for applyir each utility.

The following are utilities for adding padding to your elements:

- `p` denotes padding across the whole element.
- `py` denotes padding padding-top and padding-bottom.
- `px` denotes padding-left and padding-right.
- `pt` denotes padding-top.
- `pr` denotes padding-right.
- `pb` denotes padding-bottom.
- `pl` denotes padding-left

To apply them to your elements, you'd have to use the appropriate numbers provided by Tailwind — a bit similar to the numbers that represented col variants in the last section. Here's what we mean:

```
<button class="p-0">Click me</button>
<button class="pt-1">Click me</button>
<button class="pr-2">Click me</button>
<button class="pb-3">Click me</button>
<button class="pl-4">Click me</button>
```

Margin

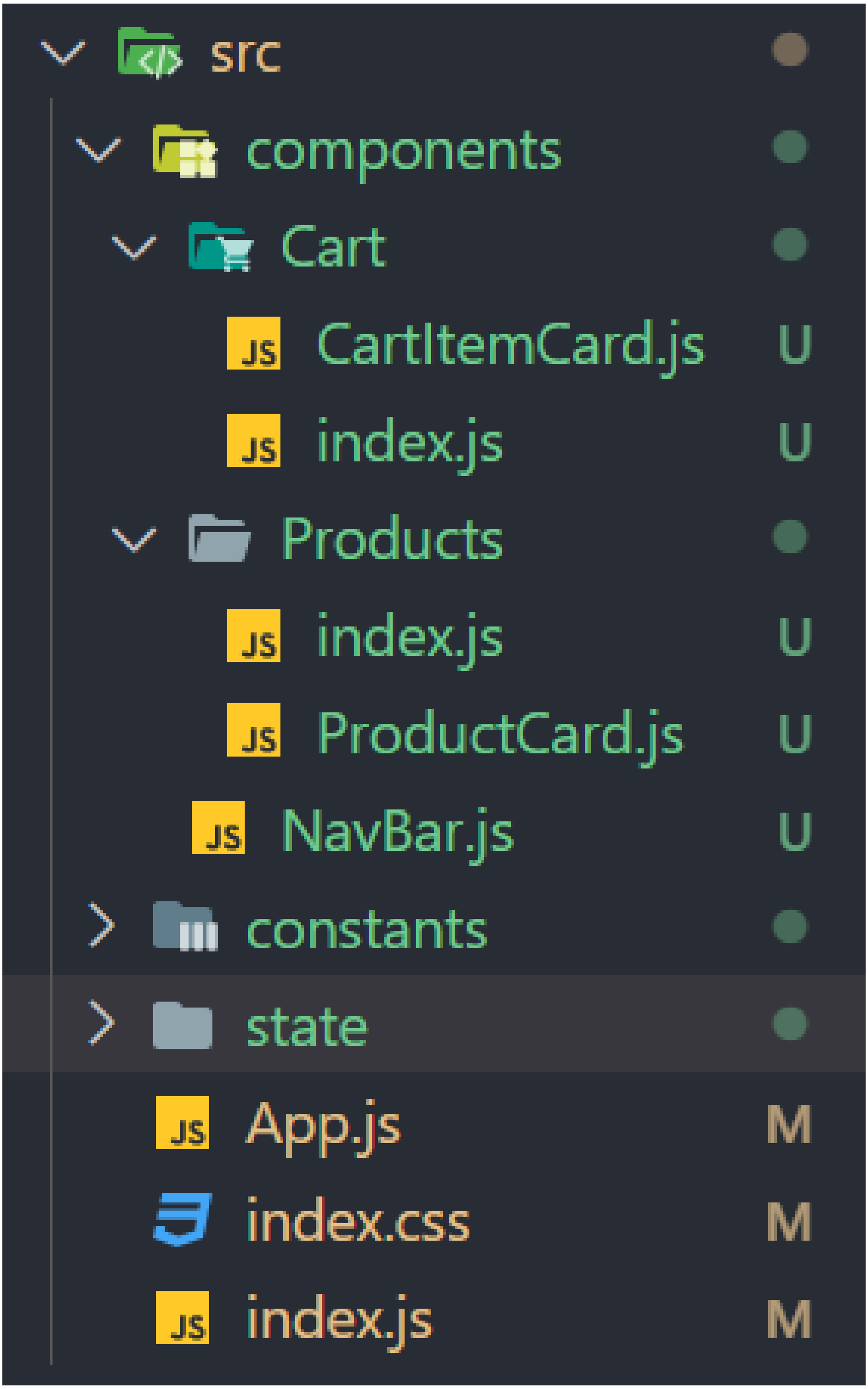
The predefined utilities for padding and margin are very similar. You have to replace the `p` with an `m` :

- `m`
- `my`
- `mx`
- `mt`
- `mr`
- `mb`
- `ml`

For a more in-depth guide to tailwindCSS classes, have a look at the Official TailwindCSS Docs.

Creating the UI

Now, let's use what we've learned so far to create the UI for our application. First, create the a folder named `components` inside `./src` . Also crea the required files. The directory structure should look like this:



```
src
  components
    Cart
      CartItemCard.js   U
      index.js          U
    Products
      index.js          U
      ProductCard.js    U
    NavBar.js           U
  constants
  state
  App.js                M
  index.css             M
  index.js              M
```

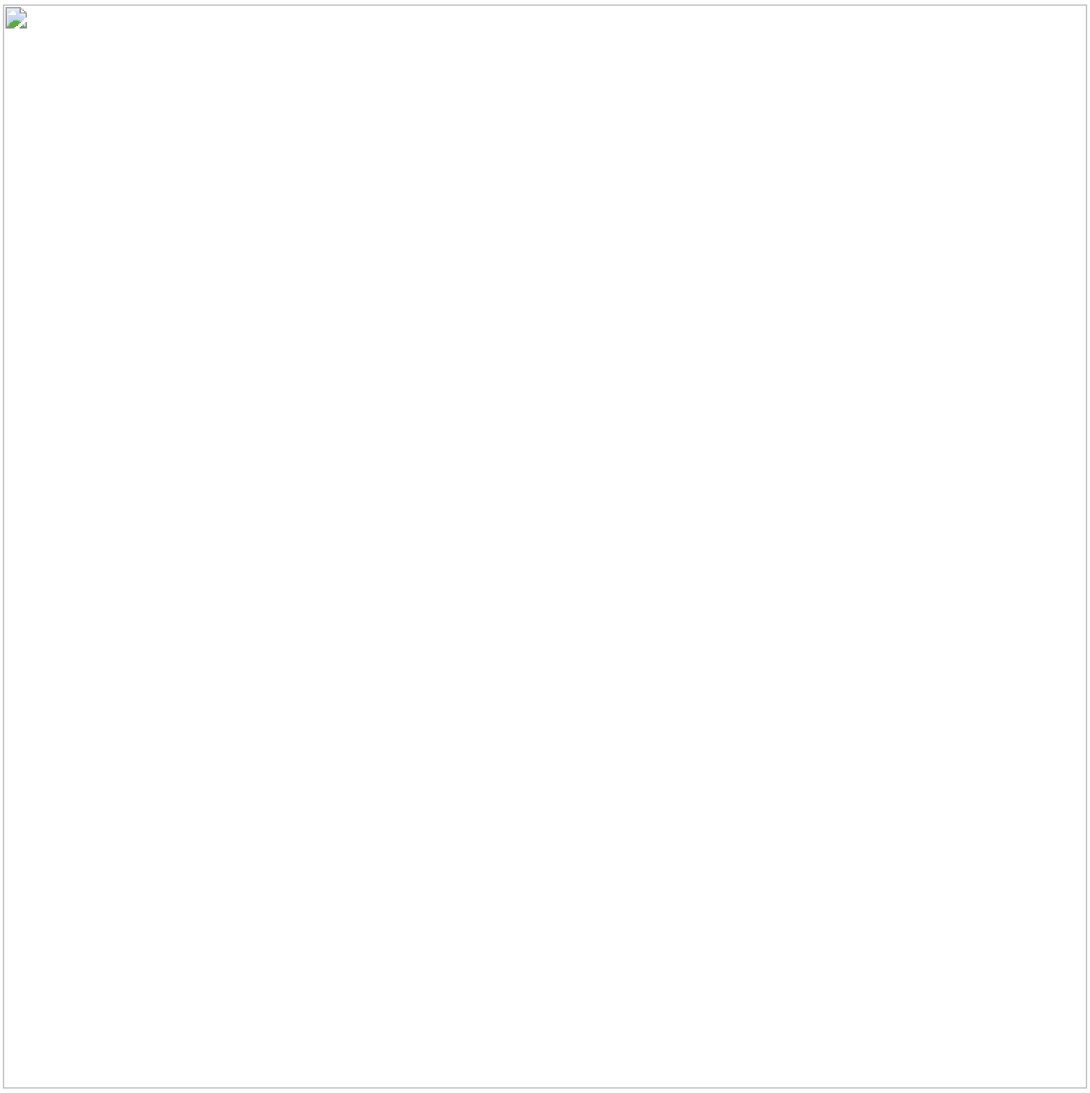
Next, Create the basic skeleton of each file and import them in `App.js` . This is what `App.js` should be like:

```
import Products from './components/Products';
import NavBar from './components/NavBar';
import Cart from './components/Cart';

function App() {
  return (
    <div className="bg-gray-100">
```

```
      <NavBar />
      <div className="my-0 mx-auto p-4 max-w-[140rem] md:p-12">
        <Products />
      </div>
      <Cart />
    </div>
  );
}

export default App;
```

Navbar

Let’s start by creating the `<NavBar />` component first.



Here we’ll just display the title and use the `useDispatch()` and `useSelector()` hooks to allow the user to open and view their cart, and also displa the number of items currently inside the user’s cart. This is what the content of this component is:

```
import React from 'react';
import { FaShoppingCart } from 'react-icons/fa';
import { useDispatch, useSelector } from 'react-redux';
import { openCart } from '../state/actions';

const NavBar = () => {
  const cart = useSelector((state) => state.cart);
  const dispatch = useDispatch();

  const sumQuantity = () => {
    return cart.reduce(
      (quantity, cartItem) => quantity + cartItem.quantity,
      0
    );
  };

  return (
    <header className="bg-black px-4">
      <div className="flex flex-col items-center justify-between w-full max-w-[140rem] my-0 mx-auto p-4 md:flex-row">
        <h1 className="font-bold text-2xl text-[#46ffd3] min-w-min">
          Redux-Shop
        </h1>
        <nav className="flex items-center w-full justify-end text-lg">
          <div
            className="relative cursor-pointer hover:scale-110 active:scale-105"
            onClick={() => dispatch(openCart())}
          >
            <button className=" rounded-full flex items-center justify-center p-2 font-bold bg-white">
              <FaShoppingCart />
            </button>
            {sumQuantity() > 0 ? (
              <div className="absolute top-5 right-5 flex items-center justify-center w-6 h-6 rounded-full font-bold tex
                {sumQuantity()}
              </div>
            ) : (
              ''
            )}
          </div>
        </nav>
      </div>
    </header>
  );
};

export default NavBar;
```

Products

Next we'll develop the UI to display fetch products from Fake Store API. This will be done using two components: `<Products />` and `<ProductCar` `/>` . Create the `<Products />` component inside `./src/components/Products/index.js` . Here's the code for this 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`
    );
  };

  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;
```

Next, create `ProductCard.js` inside the same directory and add the following code:

```
import React from 'react';
import { useDispatch } from 'react-redux';
import { addToCart } from '../../state/actions/cart';

const ProductCard = ({ id, title, price, image, category }) => {
  const product = { id, title, price, image, category };
  const dispatch = useDispatch();

  return (
    <div className="flex flex-col rounded-lg text-xl bg-white border border-slate-800">
      <div className="h-72 p-12 my-0 mx-auto">
        <img className="h-full" src={image} alt={title} />
      </div>
      <div className="flex flex-col justify-between gap-4 p-4 h-full border-t border-t-slate-800">
        <div className="flex flex-col justify-between h-full gap-4">
          <div className="font-bold">{title}</div>
          <div>${price.toFixed(2)}</div>
        </div>
        <button
          className="flex items-center justify-center p-2 font-bold w-full bg-black text-white hover:bg-slate-800"
          onClick={() => dispatch(addToCart(product))}
        >
          Add to cart
        </button>
      </div>
    </div>
  );
};

export default ProductCard;
```

Cart

Now we just need to develop UI to display items that a user has in their cart. This too will be done using two components: `<Car` `/>` and `<CartItemCard />`. These components are conditionally-rendered and the user can hide/show them by interacting with the UI. This another state that is being managed by Redux.

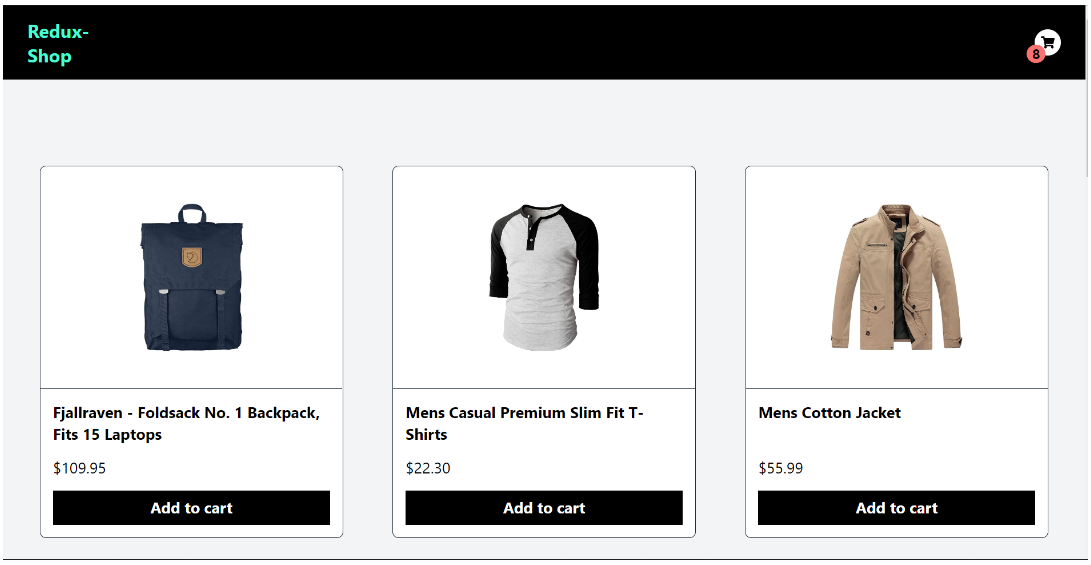
Create the `<Cart />` component inside `./src/components/Cart/index.js`.

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import CardItemCard from './CartItemCard';
import { closeCart } from '../../state/actions';

const Cart = () => {
  const cart = useSelector((state) => state.cart);
  const isCartOpen = useSelector((state) => state.isCartOpen);
  const dispatch = useDispatch();

  const sumTotal = () => {
    return cart
      .reduce(
        (total, cartItem) =>
          total + cartItem.price * cartItem.quantity,
        0
      )
      .toFixed(2);
  };

  const cartItems = cart.map((cartItem) => (
    <CardItemCard
      key={Math.random()}
      id={cartItem.id}
      title={cartItem.title}
      price={cartItem.price}
      image={cartItem.image}
      quantity={cartItem.quantity}
    ></CardItemCard>
  ));

  if (isCartOpen) {
    return (
      <>
        <div
          className="fixed z-50 top-0 right-0 flex flex-col items-center gap-8 w-full h-full p-6 bg-white text-xl md:w-[
          isOpen={isCartOpen}
        >
          <div className="font-bold mb-6 text-xl">
            Your shopping cart
          </div>
          <div className="flex flex-col gap-8 w-full overflow-auto">
            {cartItems}
          </div>
          <div className="font-bold">Total: ${sumTotal()}</div>
          <button className="flex items-center justify-center p-4 font-bold w-3/4 bg-[#46FFD3] hover:bg-[#35eec2]">
            Checkout
          </button>
          <button
            className="flex items-center justify-center p-4 font-bold w-3/4 bg-red-500 hover:bg-red-300"
            onClick={() => dispatch(closeCart())}
          >
            Close
          </button>
        </div>

        <div
          className="fixed top-0 w-full h-full opacity-60 bg-black"
          onClick={() => dispatch(closeCart())}
        />
      </>
    );
  }
  return <></>;
};

export default Cart;
```

Next, Create the `<CartItemCard />` inside same directory.

```
import React from 'react';
import { FaMinus, FaPlus } from 'react-icons/fa';
import { useDispatch } from 'react-redux';
import { addToCart, removeFromCart } from '../../state/actions';

const CardItemCard = ({ id, title, price, image, quantity }) => {
  const cartItem = { id, title, price, image, quantity };
  const product = { id, title, price, image };
  const dispatch = useDispatch();

  const formatTitle = (title) => {
    return title.length <= 14 ? title : title.substr(0, 14) + '...';
  };

  const sumPrice = () => {
    return (cartItem.price * cartItem.quantity).toFixed(2);
  };

  return (
    <div className="flex">
      <div className="w-1/5 h-[6rem] m-auto">
        <img className="h-full w-auto" src={image} alt=""></img>
      </div>
      <div className="flex flex-col items-center justify-between w-full text-lg gap-2">
        <div className="overflow-hidden font-bold h-8">
          {formatTitle(title)}
        </div>
        <div>${sumPrice()}</div>
        <div className="flex items-center gap-4">
          <button
            className="flex items-center justify-center p-2 font-bold w-full bg-gray-200 text-black hover:bg-black hover
            onClick={() => dispatch(removeFromCart(product))}
          >
            <FaMinus />
          </button>
          <div>{cartItem.quantity}</div>
          <button
            className="flex items-center justify-center p-2 font-bold w-full bg-gray-200 text-black hover:bg-black hover
            onClick={() => dispatch(addToCart(product))}
          >
            {' '}
            <FaPlus />
          </button>
        </div>
      </div>
    </div>
  );
};

export default CardItemCard;
```

With this our Redux-Shop is finally complete. This is what the project finally looks like:



Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops

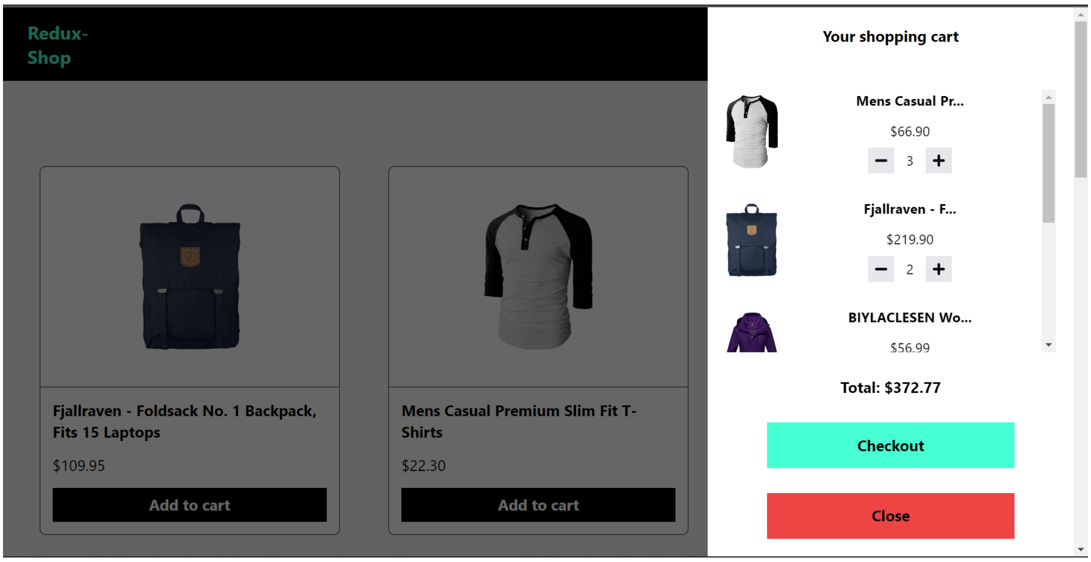
$109.95

Mens Casual Premium Slim Fit T-Shirts

$22.30

Mens Cotton Jacket

$55.99



Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops

$109.95

Mens Casual Premium Slim Fit T-Shirts

$22.30

Your shopping cart

Mens Casual Pr...

$66.90

Fjallraven - F...

$219.90

BIYLACLESEN Wo...

$56.99

Total: $372.77

Thank You !