Redux Intro

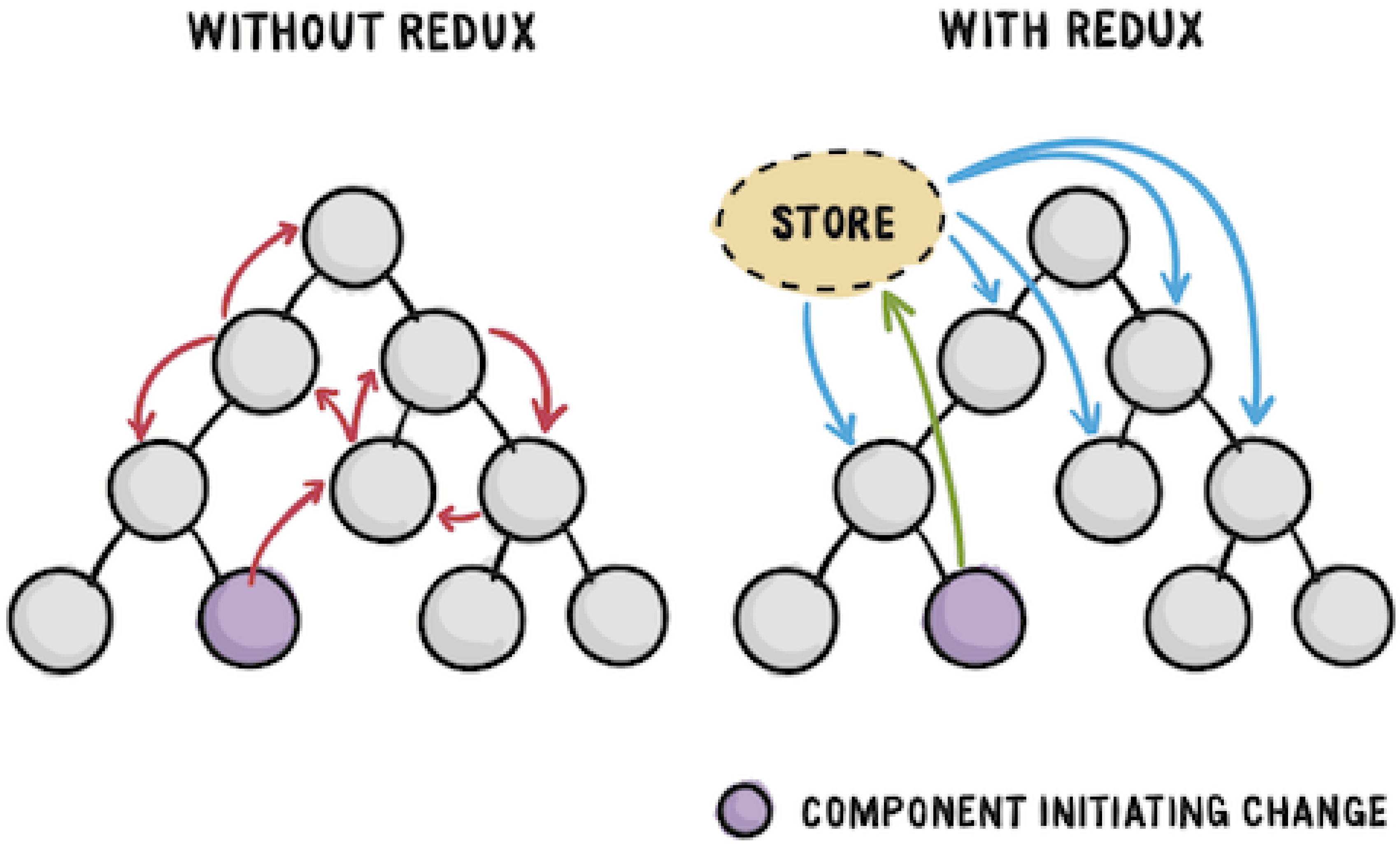
MISS XING

Why Redux?

While building a complex web application that has lots of different components, like user authentication, shopping carts, coupons, product cards, and user preferences, each of these components has its own data or more specifically “state” that can change over time, managing all such data could become difficult and messy work. Keeping track of this data at each and every individual component can become difficult.

Redux helps us solve this problem by providing a centralized place formally called as “store”, where all the application data exists. This store holds the current state of the entire application. Instead of scattering the data across various different parts of the app, Redux helps us keep it all in one place.

https://redux.js.org/understanding/thinking-in-redux/motivation


WITHOUT REDUX
WITH REDUX
STORE
COMPONENT INITIATING CHANGE

What is Redux?

Redux is a state management library for JavaScript applications, particularly popular in React, that helps manage application state in a predictable way.

It provides a centralized store for all application state, allowing components to access and update this state through a well-defined structure of actions and reducers.

This approach promotes a unidirectional data flow, making it easier to understand how data changes over time and simplifying debugging and testing. Redux is especially useful for large or complex applications where managing state can become challenging.

Redux is for any JS application

- Redux can be used with various frameworks and libraries, primarily those in the JavaScript ecosystem.
- It can be used in Node.js, React.js, Angular.js and so on.
- Redux is most commonly associated with React, but its principles can be adapted to work with many other frameworks and libraries in the JavaScript ecosystem.

Redux: Manage State

In Redux, state refers to the data that represents the current condition of your application.

The state can include anything relevant to the application, such as user information, UI settings, or data fetched from an API.

Example of State:

```
const state = {
   firstname: 'John',
   lastname: 'Smith',
   age: 20
}
```

```
const state = [
   {
      "street": “Southview Dr",
      "suite": "Apt. 556",
      "city": “Williamsburg",
      "zipcode": "92598-3874"
   },
   {
      "street": “Jiefang jie",
      "suite": "Suite 879",
      "city": “Nanjing",
      "zipcode": "90336-7771"
   }
]
```

```
const state = {
   todos: [
      { id: 1, text: 'Learn JavaScript', completed: false },
      { id: 2, text: 'Learn Redux', completed: false },
      { id: 3, text: 'Build a To-Do App', completed: true },
   ],
   loading: false,
};z
```

Why do we say Redux is predictable?

1. Single Source of Truth
   - a single, centralized state object (the store) for the entire application.
2. Unidirectional Data Flow: Data flows in a single direction
   - from the store to the view (components)
   - Components dispatch actions to change state
3. Immutability
   - The state in Redux is immutable, meaning that you never directly modify the existing state.
4. Actions and Reducers
   - State changes are made only through actions (plain objects that describe what happened) and reducers (pure functions that take the current state and an action and return a new state). This separation of concerns means that every state change is traceable and can be easily understood.
5. Predictable State Management
   - Because reducers are pure functions, given the same input (current state and action), they will always produce the same output (new state).

Demo

```
{ type: 'INCREMENT' }
```

```
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    default:
      return state;
  }
};
```

What is Redux Toolkit?

The official, opinionated, batteries-included toolset for efficient Redux development.

Redux Toolkit is an official, recommended library for managing state in Redux applications.

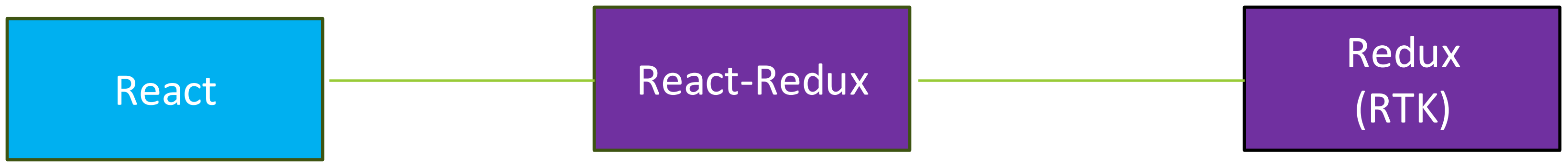
Redux Toolkit over vanilla Redux

1. Reduced Boilerplate Code
   - Redux Toolkit consolidates multiple steps into single functions (like `createSlice`).
2. Built-in Best Practices
   - It enforces best practices by default, such as using `immer` for immutable state updates, which makes writing reducers easier and less error-prone.
3. Easy Store Configuration
4. Enhanced Async Logic Handling
   - `createAsyncThunk`: This utility simplifies handling asynchronous actions (like API calls) by automatically dispatching the appropriate actions for pending, fulfilled, and rejected states, making async flows more manageable.
5. TypeScript Support
   - provides better TypeScript support out of the box, including type definitions for actions and reducers

What is React Redux?

React Redux is a library that provides bindings for integrating Redux with React applications.

React Redux connects React to Redux.



When should we use Redux?

1. At any given time, the state of a specific component must be available and shared with other components.
2. One component needs to change the state of another component (communication)
3. General guideline: If you don’t need to use, don’t use it. If encounter lots of challenges without using it, use it.

Summary

- **Redux**: A core state management library that organizes and centralizes application state, but requires more manual setup and code.
- **Redux Toolkit**: A simplified, modern version of Redux that reduces boilerplate and includes helpful tools like `createSlice` and `configureStore`.
- **`React-Redux`**: A library that connects Redux with React, allowing React components to easily access and update the Redux store.

What will we learn?

1. Redux
2. Redux Toolkit
3. React Redux

Redux Core

MISS XING

React Core Concepts

Pizza Store Case	Purpose	Redux	Purpose
Store	holds all the current pizzas in one place	Store	Hold current state of an application
Pizza Order	Tells what kind of pizza and how many pizza being ordered	Action	tells the store **what happened** or **what needs to happen**
Pizza Chef	Make pizzas based on order	Reducer	Generate new state based on actions

Three Principles:
Single source of truth

1. Single source of truth: The global state of your application is stored in an object within a single store.
   - In Redux, a *store* is a container that manages the global state of your application.
   - As the single source of truth, the store is the center of every Redux application.
   - It has the ability to update the global state and subscribes elements of an application's UI to changes in the state.
   - Rather than accessing the state directly, Redux provides functions through the store to interact with the state.

```
const store =  createStore(reducer);
```

```
{ numOfPizza: 12 }
```

Three Principles:
State is read-only

2. **The only way to change the state is to emit an action, an object describing what happened.**
   - In Redux, an action is a plain JavaScript object that represents an intention to change the store's state.
   - Action objects must have a type property with a user-defined string value that describes the action being taken.
   - Optional properties can be added to the action object. One common property added is conventionally called `payload`, which is used to supply data necessary to perform the desired action.

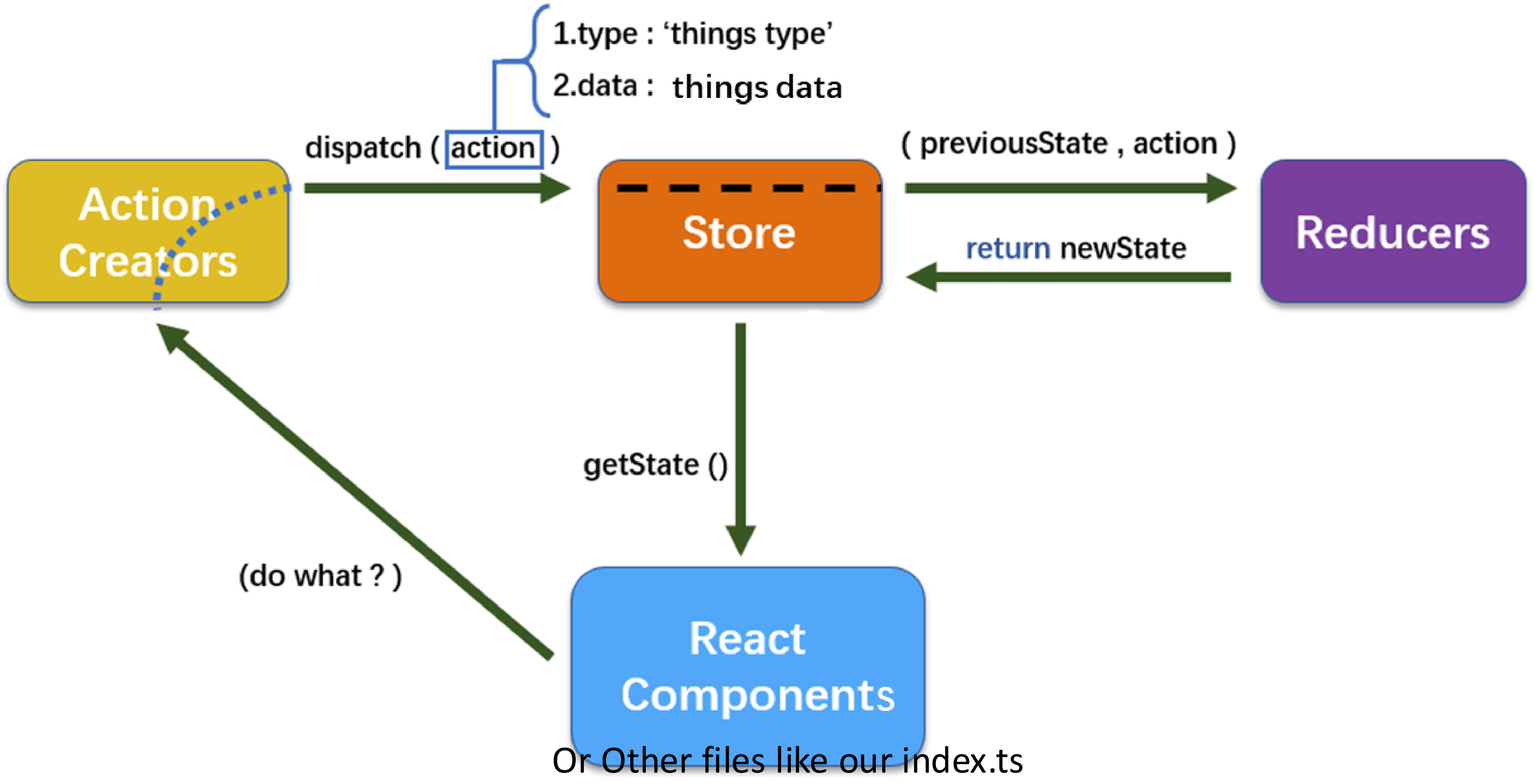
```
{
   type: PIZZA_ORDER
}
```

Three Principles:
Changes are made with pure functions

3. **To specify how the state tree is transformed by actions, you write pure reducers.**
   - In Redux, a reducer, also known as a reducing function, is a JavaScript function that takes the current `state` of the store and an `action` as parameters.
   - Reducers calculate the new state based on the action it receives.
   - Reducers are the only way the store's current state can be changed within a Redux application.
   - They are an important part of Redux's one-way data flow model.

```
const reducer = (state = initialState, action: ActionType) => {
    switch (action.type) {
        case PIZZA_ORDER:
            return {...state, numOfPizza: state.numOfPizza - 1};
        default:
            state;
    }
}
```

Redux workflow



Set Up Redux TypeScript Project

1. npm init -y
2. npm install redux
3. tsc –init
   - add extra configuration into tsconfig.json

```json
{
  "include": [
    "src"
  ],
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "outDir": "./dist",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Demo: Pizza Store - Actions

A Redux `action creator` is a function that creates and returns an `action` object. Instead of manually writing the action object each time, action creators provide a convenient way to create actions.

Why Use Action Creators?

1. Cleaner code: It avoids repeating the structure of the action object in multiple places.
2. Consistency: Ensures that actions are always created in a consistent way.

```
const PIZZA_ORDER = 'PIZZA_ORDER';

const createOrderPizzaAction = () => ({
   type: PIZZA_ORDER
});
```

Demo: Pizza Store - Reducer

```
type ActionType = {
    type: string;
}

const initialState = {
    numOfPizza: 10
}

const reducer = (state = initialState, action: ActionType) => {
    switch (action.type) {
        case PIZZA_ORDER:
            return {...state, numOfPizza: state.numOfPizza - 1};
        default:
            state;
    }
}
```

Demo: Pizza Store - Store

1. Access store state via `store.getState()`
2. Update state via `dispatch(action)`
3. Register listeners via `subscribe(listener)`
4. Unregister via `unsubscribe()`

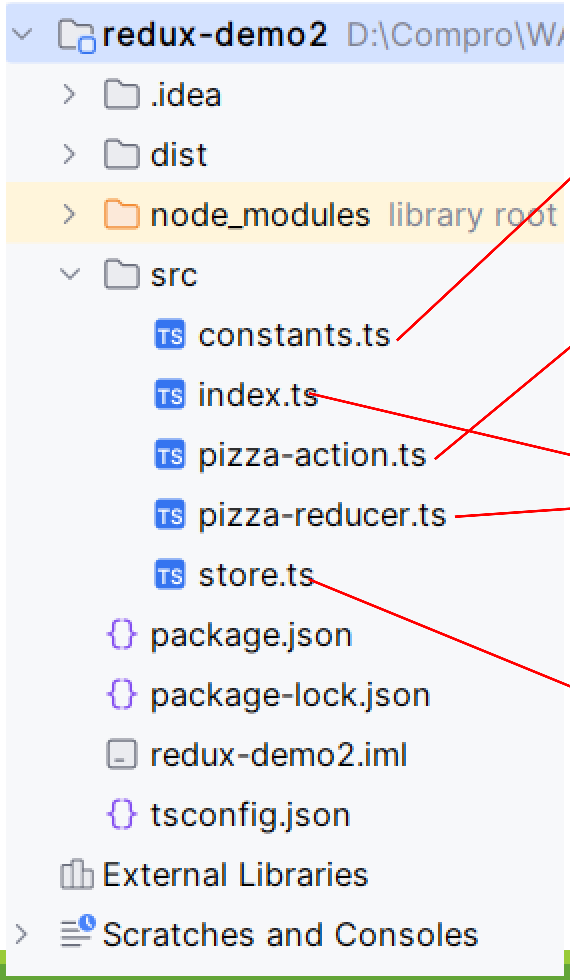
```
const store = createStore(reducer);
console.log('initial state: ', store.getState());
const unsubscribe = store.subscribe(() => {
   console.log('updated state: ', store.getState());
});

store.dispatch(createOrderPizzaAction());
store.dispatch(createOrderPizzaAction());

unsubscribe();
```

package.json

```
"scripts": {
  "build": "tsc",
  "start": "node dist/index.js"
},
```

```
PS D:\...\redux-demo2>npm run build

> redux-demo2@1.0.0 build
> tsc

PS D:\...\redux-demo2>npm start

> redux-demo2@1.0.0 start
> node dist/index.js

Initial State:  { numOfPizza: 10 }
Updated State:  { numOfPizza: 9 }
Updated State:  { numOfPizza: 8 }
```

Refactor Code into different files.



```
export const PIZZA_ORDER = 'PIZZA_ORDER';
```

```
import {PIZZA_ORDER} from "./constants";

export const createOrderPizzaAction = () => ({
    type: PIZZA_ORDER
});
```

```
const reducer = (state = initialState, action: ActionType) => {
//…
}

export default reducer;
```

```
import {createStore} from "redux";
import reducer from "./pizza-reducer";

export default createStore(reducer);
```

```
import store from './store';
import {createOrderPizzaAction} from "./pizza-
action";

console.log('Initial State: ', store.getState());
const unsubscribe = store.subscribe(() => {
    console.log('Updated State: ', store.getState());
});
store.dispatch(createOrderPizzaAction());
store.dispatch(createOrderPizzaAction());

unsubscribe();
```

Exerise:

Add a new action to restock pizza with quantity.

Redux Combined Reducers Sell Chicken too

MISS XING

Sell Fried Chicken too…

Pizza Store	Purpose	Pizza Store	Purpose
Store		holds all the pizzas and chicken in one place	
Pizza Order	Tells what kind of pizza and how many pizza being ordered	Chicken Order	Tells what kind of chicken and how many chicken being ordered
Pizza Chef	Make pizzas based on order	Chicken Chef	Make Chicken based on order

Exercise: Sell Fried Chicken!

In Redux, `combineReducers` is a utility function that helps manage different pieces of the application's state by splitting the state into multiple reducers. Each reducer is responsible for managing a specific part of the state.

Using `combineReducers`, you can combine these smaller reducers into a single root reducer, making it easier to manage complex state in larger applications.

```
import {combineReducers, createStore} from "redux";
import pizzaReducer from "./pizza-reducer";
import chickenReducer from "./chicken-reducer";

const combinedReducers = combineReducers({
    pizza: pizzaReducer,
    chicken: chickenReducer
});
export default createStore(combinedReducers);
```

```
state = {
    pizza: {
        numOfPizza: 12
    },
    chicken: {
        numOfChicken: 21
    }
}
```

Redux - immer

MISS XING

Complicated Object Update

```
type ActionType = {
    type: string;
    payload: string;
}

const initialState = {
    name: 'John Smith',
    address: {
        street: '1000 N 4th ST',
        city: 'Fairfield',
        state: 'IA'
    }
}
```

```
const reducer = (state = initialState, {type, payload}: ActionType) =>
{

    switch (type) {
        case STREET_UPDATE:
            return {...state, address: {...state.address, street: payload}};
        default:
            return state;
    }
}

export default reducer;
```

Immer

Immer (German for: always) is a library that makes it easier to work with immutable state in JavaScript, particularly in Redux.

It allows you to write code that looks like you're mutating the state directly, while under the hood it keeps the state immutable. This is useful for making your Redux reducers cleaner and easier to maintain.

Key Concepts

1. Draft State: Immer provides a "draft" state that you can modify directly. When you finish, Immer creates a new immutable state based on your changes.
2. Immutable Updates: It simplifies the process of updating nested objects without having to use complex spread operators or Object.assign.

```
npm install immer
```

Immer produce

The Immer package exposes a produce function that does all the work.

```
produce(baseState, recipe: (draftState) => void): nextState
```

```
import {produce} from "immer";

const userReducer = (state = initialState, {type, payload}: ActionType) => {
    switch (type) {
        case UPDATE_STREET:
            return produce(state, draft => {
                draft.address.street = payload;
            });
        default:
            return state;
    }
}

export default userReducer;
```

Redux Middleware redux-logger

MISS XING

Middleware

Redux middleware is a function that sits between an action being dispatched and the action reaching the reducers. It allows you to add custom logic, such as logging actions, handling asynchronous code, or modifying the dispatched actions.

`redux-logger` is a popular middleware used to log actions and the state changes they cause. It helps developers understand how the state is evolving in response to actions, which is useful for debugging.

It logs the **previous state**, the **action** being dispatched, and the **next state** after the action has been processed.

```
npm install redux-logger
```

```
npm i --save-dev @types/redux-logger
```

Demo

```
import {combineReducers, createStore, applyMiddleware} from "redux";
import logger from 'redux-logger';
import pizzaReducer from "./pizza-reducer";
import chickenReducer from "./chicken-reducer";
import personReducer from "./person-reducer";

const combinedReducers = combineReducers({
    pizza: pizzaReducer,
    chicken: chickenReducer,
    person: personReducer
});

// @ts-ignore
export default createStore(
    combinedReducers,              // Combine all reducers
    applyMiddleware(logger)  // Apply middleware (redux-logger)
);
```

Redux Async Action
redux-thunk

MISS XING

Is Async Action necessary?

Now let’s say customer will order Pizza in 5 seconds, not immediately.

Do we need Async Action here?

The simplest solution would be:

```
import store from "./store";
import {createOrderPizzaAction} from "./pizza-action";

const unsubscribe = store.subscribe(() => {
});

setTimeout(() => {
    store.dispatch(createOrderPizzaAction());
}, 5000);

unsubscribe();
```

But, if we want the setTimeout() is also part of the action?

Async Action

In Redux, actions can be classified into synchronous and asynchronous based on how they are dispatched and how they handle side effects like API calls, timers, or other async operations.

1. **Synchronous actions**
   - are dispatched and handled immediately, without any delays. Once an action is dispatched, the reducer processes the action, and the state is updated immediately.
   - Plain object: {type: ‘’, data: ‘’}
2. Asynchronous Actions
   - involve side effects like API calls, which take time to complete.
   - These actions cannot be dispatched immediately because they depend on the completion of an asynchronous operation (like fetching data).
   - Function – only function can perform async tasks(setTimeout, fetch, etc), other types like array, string, they cannot perform.
   - Example: () => {setTimeout()}

Redux thunk

redux-thunk is a middleware for Redux for asynchronous action.

Handling Asynchronous Actions: Redux is primarily designed for handling synchronous actions, where an action triggers a state change immediately. However, in real-world applications, you often need to deal with asynchronous actions like making network requests (e.g., fetching data from an API). This is where redux-thunk comes in.

```
npm install redux-thunk
```

`applyMiddleware()` – apply other middlewares in redux

Redux Thunk demo

```
import {thunk} from "redux-thunk";

const store = createStore(reducer, applyMiddleware(logger, thunk));
export default store;
```

```
import {Dispatch} from "redux";

export const createOrderPizzaAsyncAction = (deplay: number) => (dispatch:
Dispatch) => {
    setTimeout(() => {
        dispatch(createOrderPizzaAction());
    }, deplay);
}
```

```
const unsubscribe = store.subscribe(() => {});

store.dispatch(createOrderPizzaAsyncAction(5000));
```

Exercise: Fetch Users from Server

- `loading`: indicate if it’s still loading, not finished.
- `users`: List of Users
- `error`: error message

```
const state = {
   loading: false,
   users: [],
   error: ''
}
```

- Action types: FETCH_USERS_REQUEST, FETCH_USERS_SUCCESS, FETCH_USERS_FAILURE.
- Fetch users from an API and dispatch actions based on the result.

Redux Cons

1. Boilerplate Code:
   - Redux can involve a lot of boilerplate code, especially for defining actions, action types, and reducers, which can make it verbose and cumbersome.
2. Apply Middlewares by developers
   - redux-logger
   - Immer
   - Redux-thunk
3. Learning Curve:
   - For newcomers, understanding concepts like reducers, actions, and middleware can be challenging, leading to a steeper learning curve compared to simpler state management solutions.

Redux Toolkit Intro

MISS XING

Redux Toolkit

```
npm install @reduxjs/toolkit
```

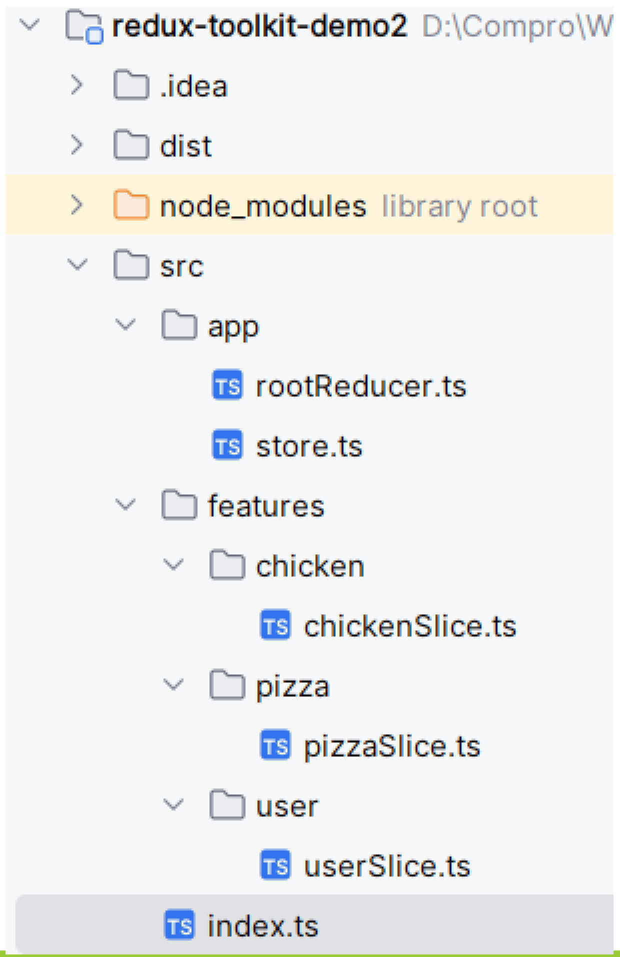
Redux Toolkit is an official, opinionated library for managing state in applications built with Redux. It simplifies the process of setting up and using Redux by providing a set of tools and best practices out of the box. Here are the key features of Redux Toolkit:

1. Simplified Store Configuration:
   - The `configureStore` function helps you create a Redux store with good default settings, including middleware like Redux Thunk and support for Redux DevTools.
2. Slice API:
   - With `createSlice`, you can define reducers and actions in one place. It reduces boilerplate by automatically generating action creators and action types based on your reducer functions.
3. Immutable Updates:
   - Redux Toolkit uses the `Immer` library, allowing you to write "mutable" code in reducers while ensuring that the state remains immutable.
4. Async Logic Handling:
   - The `createAsyncThunk` utility simplifies the creation of asynchronous actions, managing their lifecycle (pending, fulfilled, rejected) automatically.
5. TypeScript Support:
   - Redux Toolkit is designed to work well with TypeScript, providing better type inference and integration for action creators and reducers.

Folder Structure

When organizing a Redux Toolkit project, a clear and logical folder structure can enhance maintainability and scalability.

1. store.ts: Store configuration
2. rootReducer.ts // Combine reducers if needed
3. pizza: feature name
4. pizzaSlice.ts: Slice definition



```
redux-toolkit-demo2 D:\Compro\W
  .idea
  dist
  node_modules library root
  src
    app
      rootReducer.ts
      store.ts
    features
      chicken
        chickenSlice.ts
      pizza
        pizzaSlice.ts
      user
        userSlice.ts
    index.ts
```

createSlice

Simplifies the process of creating a Redux slice, which includes the reducer logic and the corresponding actions.

Helps you manage a specific part of the Redux state in a more organized and less verbose manner.

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

const mySlice = createSlice({
  name: 'sliceName',         // A string that represents the slice name
  initialState: {            // The initial state of the slice
    // Define your initial state here
  },
  reducers: {                // An object containing reducer functions
    actionName(state, action: PayloadAction<Type>) {
      // Reducer logic here
    },
    // More reducers can be defined here
  },
  extraReducers: {}
});

// Export actions
export const { actionName } = mySlice.actions;

// Export reducer
export default mySlice.reducer;
```

Demo – pizzaSlice.ts

```
import {createSlice, PayloadAction} from "@reduxjs/toolkit";

const initialState = {
    numOfPizza: 10
}

const pizzaSlice = createSlice({
    name: 'pizza',
    initialState,
    reducers: {
        ordered: (state) => {
            state.numOfPizza--
        },
        restocked: (state, action: PayloadAction<number>) => {
            state.numOfPizza += action.payload
        }
    }
});

export default pizzaSlice.reducer;
export const pizzaActions = pizzaSlice.actions;
```

configureStore

The `configureStore` function from Redux Toolkit simplifies the process of setting up a Redux store with sensible defaults.

```
import { configureStore } from '@reduxjs/toolkit';
import rootReducer from './rootReducer'; // Import your root reducer

const store = configureStore({
  reducer: rootReducer,              // The root reducer
  middleware: (getDefaultMiddleware) => getDefaultMiddleware(),          // Default middleware
});

// Optional: Type for RootState and AppDispatch
export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;

export default store;
```

Demo – store.ts

```
import {configureStore} from "@reduxjs/toolkit";
import pizzaReducer from './features/pizza/pizzaSlice';

const store = configureStore({
    reducer: {
        pizza: pizzaReducer
    }
});

export default store;
```

Demo – index.ts

```
import store from './app/store';
import {pizzaActions} from './app/features/pizza/pizzaSlice';

console.log('initial state: ', store.getState());
const unsubscribe = store.subscribe(() => {
   console.log('updated sate: ', store.getState());
});

store.dispatch(pizzaActions.ordered());
store.dispatch(pizzaActions.ordered());
store.dispatch(pizzaActions.restocked(5));

unsubscribe();
```

Redux Toolkit: Combine Reducers

MISS XING

Add Chicken feature (Cont.)

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

const initialState = {
    numOfChicken: 10
}

const chickenSlice = createSlice({
    name: 'chicken',
    initialState,
    reducers: {
        restock(state, action: PayloadAction<number>) {
            state.numOfChicken += action.payload;
        },
        ordered(state) {
            state.numOfChicken--;
        }
    }
});

export const {restock, ordered} = chickenSlice.actions;
export default chickenSlice.reducer;
```

Add Chicken feature

```
const store = configureStore({
    reducer: {
        pizza: pizzaReducer,
        chicken: chickenReducer
    }
});

export default store;
```

```
import {restock, ordered} from "./app/features/chicken/chickenSlice";

console.log('initial state', store.getState());
const unsubscribe = store.subscribe(() => {
    console.log('updated sate: ', store.getState());
});

store.dispatch(ordered());
store.dispatch(restock(6));
store.dispatch(ordered());

unsubscribe();
```

Combine Reducers as RootReducer

In Redux, the `combineReducers` function is used to combine multiple reducer functions into a single reducer function, allowing you to manage different slices of state independently. When using Redux Toolkit, you can also utilize this function to set up your reducers.

```
import {combineReducers} from "@reduxjs/toolkit";

const rootReducer = combineReducers({
    pizza: pizzaReducer,
    chicken: chickenReducer
});

export default rootReducer;
```

```
import rootReducer from "./rootReducer";

const store = configureStore({
    reducer: rootReducer
});
```

Redux Toolkit Middlewares

MISS XING

Apply middlewares

```
npm install redux-logger
```

```
npm i --save-dev @types/redux-logger
```

```
import {configureStore} from "@reduxjs/toolkit";
import logger from "redux-logger";

import rootReducer from "./rootReducer";

const store = configureStore({
    reducer: rootReducer,
    middleware: getDefaultMiddleware => getDefaultMiddleware().concat(logger)
});

export default store;
```

Redux Toolkit extraReducers

MISS XING

createSlice - extraReducers

In Redux Toolkit, `extraReducers` are a powerful feature that allows you to handle actions that are not defined in the slice itself. They are particularly useful when you want to respond to actions created by asynchronous operations (like API calls) using `createAsyncThunk` or actions from other slices.

```
const chickenSlice = createSlice({
    name: 'chicken',
    initialState,
    reducers: {
      …
    },
    extraReducers: builder => {
        builder.addCase(pizzaActions.ordered, (state) =>{
            state.numOfChicken--;
        });
    }
});
```

Redux Toolkit Async Actions `createAsyncThunk`

MISS XING

Async Actions

The `createAsyncThunk` function in Redux Toolkit is used to create asynchronous actions that can be dispatched. It accepts a Redux action type string and a callback function that should return a promise.

It simplifies the process of handling async logic (like API calls) in Redux by automatically dispatching pending, fulfilled, and rejected actions based on the promise's state.

Demo

```
interface User {
   id: number;
   name: string;
   username: string;
}

type StateType = {
   loading: boolean;
   users: User[];
   error: string;
}

const initialState: StateType = {
   loading: false,
   users: [],
   error: ''
}
```

```
export const fetchUsers = createAsyncThunk('user/fetchUsers', async () => {
   const response = await axios.get('https://jsonplaceholder.typicode.com/users')
   return response.data;
});
```

Demo (Cont.)

```
const userSlice = createSlice({
    name: 'user',
    initialState,
    reducers: {},
    extraReducers: builder => {
        builder.addCase(fetchUsers.pending, state => {
            state.loading = true;
        })
            .addCase(fetchUsers.fulfilled, (state, action) => {
                state.loading = false;
                state.users = action.payload;
                state.error = '';
            })
            .addCase(fetchUsers.rejected, (state, action) => {
                state.loading = false;
                state.users = [];
                state.error = action.error.message || 'Whoops!';
            });
    }
});

export default userSlice.reducer;
```

Demo (Cont.)

```
const rootReducer = combineReducers({
    pizza: pizzaReducer,
    chicken: chickenReducer,
    user: userReducer
});
```

```
console.log('initial state', store.getState());
const unsubscribe = store.subscribe(() => {
    console.log('updated sate: ', store.getState());
});

store.dispatch(fetchUsers());
```

React Redux

MISS XING

Integrating Redux with a UI

Redux itself is a standalone library that can be used with any UI layer or framework, including React, Angular, Vue, Ember, and vanilla JS. Although Redux and React are commonly used together, they are independent of each other.

Using Redux with *any* UI layer requires the same consistent set of steps:

1. Create a Redux store
2. Subscribe to updates
3. Inside the subscription callback:
    i. Get the current store state
    ii. Extract the data needed by this piece of UI
    iii. Update the UI with the data
4. If necessary, render the UI with initial state
5. Respond to UI inputs by dispatching Redux actions

While it is possible to write this logic by hand, doing so would become very repetitive. In addition, optimizing UI performance would require complicated logic.

React Redux

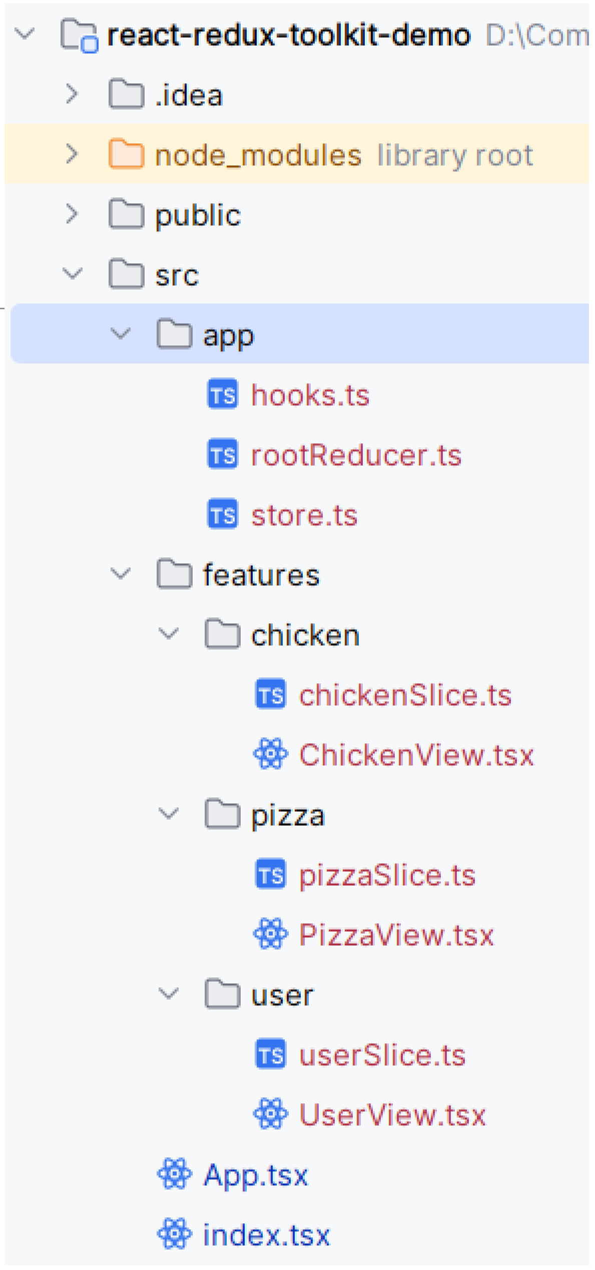
The process of subscribing to the store, checking for updated data, and triggering a re-render can be made more generic and reusable. **A UI binding library like React Redux handles the store interaction logic, so you don't have to write that code yourself.**

If you are using Redux with any kind of UI framework, you will normally use a "UI binding" library to tie Redux together with your UI framework, rather than directly interacting with the store from your UI code.

React Redux is the official Redux UI binding library for React. If you are using Redux and React together, you should also use React Redux to bind these two libraries.

React Redux Project setup

1. Create a React TypeScript project
2. Copy redux toolkit code (`app` and `features` folders) into `src` folder
3. Under the `features` folder, create React components to connect to store later.



```
react-redux-toolkit-demo D:\Com
  .idea
  node_modules library root
  public
  src
    app
      hooks.ts
      rootReducer.ts
      store.ts
    features
      chicken
        chickenSlice.ts
        ChickenView.tsx
      pizza
        pizzaSlice.ts
        PizzaView.tsx
      user
        userSlice.ts
        UserView.tsx
    App.tsx
    index.tsx
```

Provider

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 the entire app's component tree inside of it.

```
root.render(
    <React.StrictMode>
        <Provider store={store}>
            <App/>
        </Provider>
    </React.StrictMode>
);
```

`useSelector`

The `useSelector` hook is a part of the React Redux library that allows you to extract data from the Redux store state in a functional component. It enables you to subscribe to the store and re-render the component whenever the selected state changes.

Key Features of `useSelector`

- Selects State: You can use it to select specific pieces of state from the Redux store.
- Automatic Re-Rendering: Components that use useSelector will automatically re-render when the selected state changes.
- Memoization: By default, it uses strict equality (===) to compare the selected state, which can help avoid unnecessary re-renders.

- **A `useSelector` call returning the entire root state is almost always a mistake**, as it means the component will rerender whenever *anything* in state changes. Selectors should be as granular as possible, like `state => state.some.nested.field`.

useSelector Demo

```
export default function PizzaView(){

    const {numOfPizza} = useSelector((state: RootState) => state.pizza);

    return (
        <>
            <h2>Pizza State: {numOfPizza}</h2>
        </>
    );
}
```

```
const store = configureStore({
    reducer: rootReducer,
    middleware: getDefaultMiddleware => getDefaultMiddleware().concat(logger)
});

export default store;
export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>
```

useDispatch

The `useDispatch` hook is a part of the React Redux library that allows you to dispatch actions to the Redux store from functional components. This is essential for updating the state in your application based on user interactions or asynchronous operations.

Key Features of `useDispatch`:

1. Access to Dispatch Function: It provides the `dispatch` function, which you can use to send actions to the Redux store.
2. No Subscriptions: Unlike `useSelector`, `useDispatch` does not subscribe to the store, so it won't cause your component to re-render when the state changes.
3. Flexible Action Dispatching: You can dispatch both synchronous and asynchronous actions (like those created with `createAsyncThunk`).

useDispatch Demo

```
export default function PizzaView(){

   const {numOfPizza} = useSelector((state: RootState) => state.pizza);
   const dispatch = useDispatch<AppDispatch>();

   return (
      <>
         <h2>Pizza State: {numOfPizza}</h2>
         <button onClick={() => dispatch(pizzaActions.ordered())}>Order</button>
         <button onClick={() => dispatch(pizzaActions.restocked(5))}>Restock</button>
      </>
   );
}
```

Custom hooks

```
import {TypedUseSelectorHook, useDispatch, useSelector} from "react-redux";
import {AppDispatch, RootState} from "./store";

export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;
export const useAppDispatch = () => useDispatch<AppDispatch>();
```

```
const {numOfPizza} = useSelector((state: RootState) => state.pizza);
const dispatch = useDispatch<AppDispatch>();
```

$\downarrow$

```
const {numOfPizza} = useAppSelector(state => state.pizza);
const dispatch = useAppDispatch();
```

Fetch Users

```
export default function UserView() {
    const dispatch = useAppDispatch();
    const {loading, users, error} = useAppSelector(state => state.user);

    useEffect(() => {
        dispatch(fetchUsers());
    }, []);

    return (
        <>
            <h2>List of Users</h2>
            {
                loading? <div>Loading</div>:
                    error? <div>{error}</div>:
                        <ul>
                            {users.map(u => <li key={u.id}>{u.name}</li>)}
                        </ul>
            }
        </>
    )
}
```

Thank you!