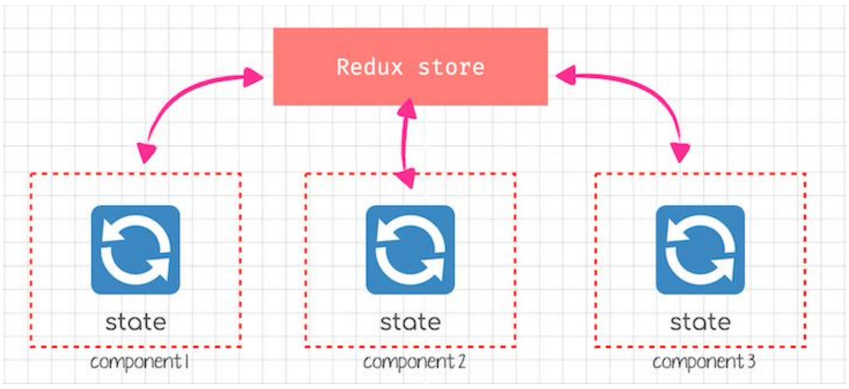
Redux Toolkit

1. Redux is a JS library for predictable and maintainable global state management. With Redux, the state of your application is kept in a centralized store, from where any component can access the state it needs.
2. Redux is used to maintain and update data across your application for multiple components to share while also remaining independent of them. A large application often demands storing the state at a central location and sharing it among the different components. That is where the Redux store comes into the picture.



3. In Redux, the state is always predictable because of immutability. If the same state and action are passed to a reducer, they will always produce the same results because the state is immutable, as reducers are pure functions that won't cause any side effects.

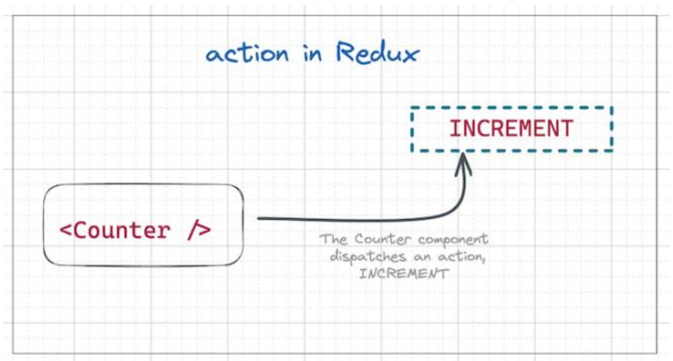
4. How Redux Works:

   There are five core Redux components — store, actions, reducers, dispatch, and selectors.

   - Store holds the state of application The state of whole application is stored in an object within a single store
   - Action describes the changes in the state of the application Cannot directly update the state object that is done by redux only
   - Reducer which carries out the actual state transition based upon the action. Pure reducers which take state and action and returns a new state

4.1. **ACTIONS:**

Redux actions can be seen as events and are the only way to send data from your application to your Redux store. The data can be based on any event such as user interactions like form submissions, or API calls.



Actions are plain JavaScript objects that must have:

- A type property (required in every Redux action) to indicate the type of action to be carried out

- A payload object (optional but important) that contains the information that should be used to change the state

Actions are created via an action creator, which is a function that returns an action. Actions are generally executed using the dispatch() method to get sent to the store:

Here's a simple example of a Redux action:
Without payload

const INCR = 'INCREMENT';

const INCRBYVALUE = 'INCRBYVALUE';

```
{
  type: INCR
}
```

With payload

```
{
  type: INCRBYVALUE,
  payload: 5
}
```

Below is an example of a Redux action creator, which is just a helper function returning the action, and we can export it to further dispatch the associated action to the store as needed:

```
const increment = ()=>{
  return {type: INCR}
}

const incrementbyvalue = (value)=>{
    return {type: INCRBYVALUE, payload:value }
}
```

Alternatively, Redux Toolkit's createAction utility saves us some effort by returning a function that takes an optional payload object as an argument.

This function, when called, returns an object with type and payload properties as specified.

```
const increment= createAction(INCR)

console.log(increment())
```

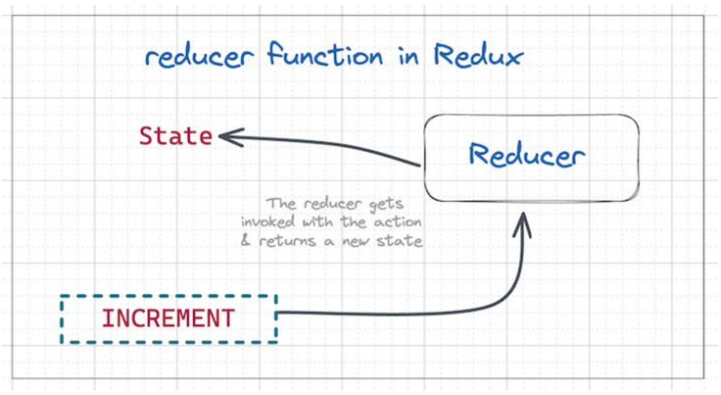
Returns {type:'INCREMENT', payload:undefined}

```
const incrementbyvalue = createAction(INCRBYVALUE)
console.log(incrementbyvalue(5))
```

Returns {type:'INCREMENT', payload:5}

4.2. REDUCERS:

Reducers are pure functions that take the current state of an application, perform an action, and return a new state. The reducer handles how the application data (i.e., the state) will change in response to an action:


reducer function in Redux
State
Reducer
The reducer gets invoked with the action & returns a new state
INCREMENT

The reducer in Redux is a normal, pure function that takes care of the various possible values of state using the switch case syntax. But that means several things need to be taken care of — most importantly, keeping the state immutable.

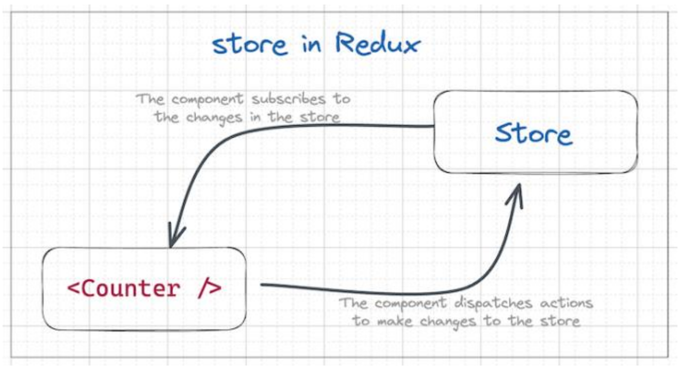
```
const initialState = {
  value: 0,
};

function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  switch(action.type){
    case  INCR:
        // If so, make a copy of `state`
        return {
        ...state,
        // and update the copy with the new value
        value: state.value + 1
        }
    case INCRBYVALUE:
        // If so, make a copy of `state`
        return {
        ...state,
        // and update the copy with the new value
        value: state.value +action.payload
        }
    default:
            // otherwise return the existing state unchanged
            return state
    }
}
```

4.3. STORE

There is a central store that holds the entire state of the application. Each component can access the stored state without sending down props/input properties from one component to another.
The only way to change the state in Redux is through dispatching associated actions to the store.

Redux allows individual components to connect to the store to grab the required state using selectors.


store in Redux
The component subscribes to the changes in the store
Store
<Counter />
The component dispatches actions to make changes to the store

You can access the stored state with **selectors**, update the state by dispatching actions, register or unregister listeners via helper methods, and subscribe to the changes to the store so they know when to re-render:

```
const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

4.4. REDUX DISPATCH

Finally, we can read the state value using the selector in our component and dispatch changes to the store using dispatch function of the store:

// here we provide the action creator function that dispatches the action with type and payload if any

store.dispatch(increment())

4.5. REDUX MIDDLEWARE

Redux allows developers to intercept all actions dispatched from components before they are passed to the reducer function. This interception is done via middleware functions, which can help you modify the intercepted actions, dispatch new actions, or perform side effects as and when required.

A middleware function receives the next method as an argument, which it can call to pass the action to the next middleware or reducer. It can process the action before and or after calling next, modify the action, dispatch additional actions, or even decide not to call next at all — it’s totally up to the task at hand

Steps to implement Redux

1. Install redux toolkit from within the folder that has **package.json file:**

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

2. Create a file counter.js
3. Add below imports :

   ```
   import { configureStore , createAction,} from '@reduxjs/toolkit';
   ```

4. Create initial state:

   ```
   const initialState = { value: 0 }
   ```

5. Create Actions via action creators as follows [Either use createAction or toolkit or your own actions]:

   ```
   const INCR = 'INCREMENT';
   const DECR = 'DECREMENT';
   const INCRBYVALUE = 'INCRBYVALUE';

   const increment = createAction(INCR);
   const incrementbyvalue = createAction(INCRBYVALUE)
   const decrement = createAction(DECR)
   ```

6. Create reducers:

   ```
   function counterReducer(state = initialState, action) {
     switch(action.type){
       case  INCR:
           return {
           ...state,
           value: state.value + 1
           }
       case DECR:
           return {
           ...state,
           value: state.value - 1
           }
       case INCRBYVALUE:
           return {
               ...state,
               value: state.value + action.payload
             }
       default:
               return state
       }
   }
   ```

7. Create store:

   ```
   const store = configureStore({
       reducer: {counterReducer},
     })
   ```

8. Time to test the store in action. Add below lines to dispatch actions to the store and get the latest state of the store:

   ```
   // get the current state
   console.log(store.getState())
   ```

```
// can subscribe to listen for state changes in the store
store.subscribe(()=>{console.log(store.getState())})

// dispatch an action to the store to update the state
store.dispatch(increment())
store.dispatch(decrement())
store.dispatch(incrementbyvalue(5))
```

9. Add logger middleware:

   ```
   npm i redux-logger
   ```

10. Add imports and create logger as follows:

    ```
    import reduxlogger from 'redux-logger'
    const logger = reduxlogger.createLogger()
    ```

11. Update store for this logger:

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

    Run the application logger will generate logs

Steps to implement Redux in Angular application

1. Install the ngrx library:

   ```
   npm install @ngrx/store
   ```

2. Create components as follows:

   ```
   ng g c redux
   ```

3. Execute below command from within the **redux** folder

   ```
   ng g c create
   ng g c read
   ```

4. Create model folder within **redux** and a file tutorial.model.ts file and add below code as follows :

   ```
   export interface Tutorial{
      id:number;
      name:string;
      url:string;
   }
   ```

5. Create actions folder within **redux** folder and add a file tutorial.action.ts file and add below code:

   ```
   import {Action, createAction} from '@ngrx/store'
   import { Tutorial } from '../model/tutorial.model'

   // the name between brackets is a convention for where I am dispatching an action
   // and the rest of the string is the action name which indicates his intention
   export const ADD_TUTORIAL = '[TUTORIAL] Add'
   export const REMOVE_TUTORIAL = '[TUTORIAL] Remove'
   export class AddTutorialAction implements Action{
   ```

```
    readonly type: string = ADD_TUTORIAL;
    constructor(public payload:Tutorial){}
}
export class RemoveTutorialAction implements Action{
    readonly type: string = REMOVE_TUTORIAL;
    constructor(public payload:number){    }
}
export type Actions = AddTutorialAction | RemoveTutorialAction
```

6. Create reducer folder within **redux** folder and add a file tutorial.reducer.ts file and add below code:

```
import { Tutorial } from '../model/tutorial.model'
import * as TutorialActions from '../actions/tutorial.action'

const initalState: Tutorial = {
    id:1,
    name:'First Tutorial',
    url:'http://someimage'
}

export function tutorialReducer(state:Tutorial[] =[initalState],
    action:TutorialActions.Actions )
    {
        switch(action.type){
            case TutorialActions.ADD_TUTORIAL:
                return [...state, action.payload];
            case TutorialActions.REMOVE_TUTORIAL:
                console.log(state)
                state = state.filter(tut => tut.id !== action.payload)
                console.log(state)
                return state
            default:
                return state
        }
     }
```

7. Create store in app.module.ts imports[] and configure the reducer as follows:

```
StoreModule.forRoot({tutorial: tutorialReducer} as ActionReducerMap<any, any>)
```

8. Update redux component html as follows:

```
<app-create></app-create>
<app-read></app-read>
```

9. Update read component ts and html as follows:

```
tutorials : Observable<Tutorial[]>;
  constructor(private store : Store<any>)
  {
    // "tutorial" is coming from the app.module.ts
    this.tutorials = store.select("tutorial")
  }
  delTutorial(id:any){
    console.log(id)
    this.store.dispatch(
      new TutorialActions.RemoveTutorialAction(parseInt(id)))
  }
```

```
<div class="right" *ngIf="tutorials; else empty">
   <h3>Tutorials</h3>
   <ul>
      <li  *ngFor="let tut of tutorials | async; let i = index">
         <a [href]="tut.url" target="_blank">{{tut.name}}</a>
         <span style="color: red;margin-left: 10px;cursor: pointer;"
(click)="delTutorial(tut.id)">X</span>
      </li>
   </ul>
</div>
<ng-template #empty>
<div class="right">
   No Tutorials Yet
</div>
</ng-template>
```

10. Update create component html and ts as follows:

```
constructor(private store:Store<any>){
  }

  addTutorial(id:any, name:string, url:string){
    this.store.dispatch(new TutorailActions.AddTutorialAction({id:parseInt(id),name:name, url:url}))
  }

<div class="left">
   <input type="text" #id placeholder="Id" value="2">
   <input type="text" #name placeholder="Name" value="Second">
   <input type="text" #url placeholder="URL" value="https://www.google.com">
   <button (click) ="addTutorial(id.value, name.value, url.value)">Add Tutorial</button>
</div>
```

11. Update routes to create a path for redux and load ReduxComponent for this route
12. Update the header html for adding link for the path redux
13. Test the application

For REST API calls:

https://medium.com/@nateogbonna/making-api-calls-efficiently-with-ngrx-and-rxjs-in-angular-50ce6f2c8267