What is Redux-toolkit?

React and Redux believed to be the best combo for managing state in large-scale React applications. However, the configuration and the enormously required boilerplate made it a little bit hard to understand and manipulate.

Here came the role of the redux toolkit.

Redux-toolkit is a new way to implement Redux, a more functional way. It's cleaner, you write fewer lines of code and we get the same Redux state management, we have come to love and trust. The best part is it comes with `redux-thunk` already built into it. Plus they use `immerJs` to handle all the immutability, so all we need to think about is what needs to get done.

Main features of Redux Tool Kit

The following function is used by Redux Took Kit, which is an abstract of the existing Redux function. These function does not change the flow of Redux but only streamline them in a more readable and manageable manner.

- **configureStore**: Creates a Redux store instance like the original createStore from Redux, but accepts a named options object and sets up the Redux DevTools Extension automatically.
- **createAction**: Accepts an action type string and returns an action creator function that uses that type.
- **createReducer**: Accepts an initial state value and a lookup table of action types to reducer functions and creates a reducer that handles all action types.
- **createSlice**: Accepts an initial state and a lookup table with reducer names and functions and automatically generates action creator functions, action type strings, and a reducer function.

You can use the above function to simplify the boilerplate code in Redux, especially using the createAction and createReducer methods. However, this can be further simplified using createSlice, which automatically generates action creator and reducer functions.

What is so special about createSlice?

It is a helper function that generates a store slice. It takes the slice's name, the initial state, and the reducer function to return reducer, action types, and action creators.

First, let's see how reducers and actions look like in traditional React-Redux applications.

- Actions

```
import {GET_USERS,CREATE_USER,DELETE_USER} from "../constant/constants";
export const GetUsers = (data) => (dispatch) => {
 dispatch({
  type: GET_USERS,
  payload: data,
 });
};
export const CreateUser = (data) => (dispatch) => {
 dispatch({
  type: CREATE_USER,
  payload: data,
 });
};
export const DeleteUser = (data) => (dispatch) => {
 dispatch({
  type: DELETE_USER,
  payload: data,
 });
};
```

- Reducers

```
import {GET_USERS,CREATE_USER,DELETE_USER} from "../constant/constants";
const initialState = {
 errorMessage: "",
 loading: false,
 users:[]
};
const UserReducer = (state = initialState, { payload }) => {
switch (type) {
 case GET_USERS:
  return { ...state, users: payload, loading: false };
case CREATE_USER:
  return { ...state, users: [payload,...state.users],
 loading: false };
case DELETE_USER:
  return { ...state,
  users: state.users.filter((user) => user.id !== payload.id),
, loading: false };
default:
  return state;
 }
};
export default UserReducer;
```

Now let's see how we can simplify and achieve the same functionality by using `createSlice`.

```
import { createSlice } from '@reduxjs/toolkit';
export const initialState = {
  users: [],
  loading: false,
  error: false,
};
const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    getUser: (state, action) => {
      state.users = action.payload;

      state.loading = true;

      state.error = false;

    },

    createUser: (state, action) => {

      state.users.unshift(action.payload);

      state.loading = false;

    },

    deleteUser: (state, action) => {

      state.users.filter((user) => user.id !== action.payload.id);

      state.loading = false;

    },

  },

});

export const { createUser, deleteUser, getUser } = userSlice.actions;

export default userSlice.reducer;
```

As you can see now all the actions and reducers are in a simple place wherein a traditional redux application you need to manage every action and its corresponding action inside the reducer. when using `createSlice` you don't need to use a switch to identify the action.

When it comes to mutating state, a typical Redux flow will throw errors and you will require special JavaScript tactics like `spread operator` and `Object assign` to overcome them. Since the Redux toolkit uses `Immer`, you do not have to worry about mutating the state. Since a slice creates the actions and reducers you can export them and use them in your component and in Store to configure the Redux without having separate files and directories for actions and reducers as below.

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

import userSlice from "./features/user/userSlice";

export default configureStore({

 reducer: {

  user: userSlice,

 },

});
```