How to declare, read and modify the shared state

Redux (with the redux toolkit)

Create a slice

```
const userSlice = createSlice({
    name: 'user',
    initialUser,
    reducers: {
        updateUser(state, action) {
            return { ...state, ...action.payload }
        }
    }
})

const { updateUser } = userSlice.actions
```

note Redux strongly-recommends to put all the logic in the reducers.

Create a store and subscribe the reducers

```
const store = configureStore({
    reducers: {
        user: userSlice.reducer
    }
})
```

With selectors and actions

```
const user = useSelector(state => state.user)
const dispatch = useDispatch()
...
const onSave = () => {
    dispatch(updateUser({ newUserName, newUserEmail }))
}
...
<div>
    <h1>{user.name}</h1>
    <small>{user.email}</small>
</div>
```

MobX

Create an observable

```
class User {
    constructor(name, email) {
        this.name = name
        this.email = email

        makeAutoObservable(this)
    }
}
```

And should be initialized instanciated. Each instance in individual.

Modify the state directly

```
const user = new User()  // This could be done in the context
...
const onSave = () => {
    user.name = newUserName
    user.email = newUserEmail
}
...
<div>
    <h1>{user.name}</h1>
    <small>{user.email}</small>
</div>
```

note: the hole component needs to be wrapped in the function `observer` provided by `MobX`.

Zustand

Create a store (zustand is based on individual stores)

```
const userStore = create((set) => ({
    user: initialUser,
    updateUser: (name, email) => set((state) => ({ ...state.user, name,
email }))
}))
```

Modify the state through the `store`

```
const user = userStore((state) => state.user)
const updateUser = userStore((state) => state.updateUser)
...
const onSave = () => {
    updateUser(newUserName, newUserEmail)
}
...
```

```
<div>
    <h1>{user.name}</h1>
    <small>{user.email}</small>
</div>
```

Recoil

Create an atom

```
const userAtom = atom({
    key: 'user',
    default: initialUser
})
```

Modify the state through the `useRecoilState` hook

```
const [user, setUser] = useRecoilState(userAtom)  // This is the same atom
created above
...
const onSave = () => {
    setUser({ ...user, name: newUserName, email: newUserEmail })
}
...
<div>
    <h1>{user.name}</h1>
    <small>{user.email}</small>
</div>
```

Context

Create the context and a provider

```
const context = createContext();

const Provider = ({children}) => {
    const [user, setUser] = useState(initialUser)

    return (
        <context.Provider value={{user, setUser}}>
            {children}
        </context.Provider>
    )
}
```

Modify the state with the context hook

```
const { user, setUset } = useContext(context)  // The same context created
above
...
const onSave = () => {
    setUser({ ...user, name: newUserName, email: newUserEmail })
}
...
<div>
    <h1>{user.name}</h1>
    <small>{user.email}</small>
</div>
```

note: For each of this libreries once you read inside a component a shared state it is subscribed and it will re-render if an update occurs.

Async process

Redux

Redux support async process by itself but it gives you some helpfull middlewares to deal with them. The first one is `createAsyncThunk`

```
const fetchUser = createAsyncThunk('user/fetchUser', async () => {
  const response = await client.get('/fakeApi/user')
  return response.data
})
```

And then you need to subscribe it to the slice reducers

```
const userSlice = createSlice({
    name: 'user',
    initialUser,
    reducers: {
        ...
        extraReducers(builder) {
            builder
            .addCase(fetchUser.pending, (state, action) => {
                state.status = 'loading'
            })
            .addCase(fetchUser.fulfilled, (state, action) => {
                state.status = 'succeeded'
                state.user = action.payload
            })
            .addCase(fetchUser.rejected, (state, action) => {
                state.status = 'failed'
                state.error = action.error.message
            })
            }
        }
    })
```

Then based on the `.status` you are going to use it in the component.

Zustand

The functions to modify the state in Zustand has no problems with `async/await`

```
const userStore = create((set, get) => ({
    user: null,
    status: 'idle',
    fetchUser: async () => {
        set((state) => ({ status: 'loading' }))

        const user = await fetchUser()

        set((state) => ({ status: 'done', user: { ...state.user } }))
    }
}))
```

Then based on the `.status` you are going to use it in the component.

Recoil

You need to use the `selectors` that is the seconds the main concepts of recoil. A `selector` is a derived satate (a state that is derived from another).

```
const queryUser = selector({
  key: 'user',
  get: async ({get}) => {
    const response = await fetchUser(get(userID))  // Let userID be an atom
    return response.data;
  },
});
```

And to use to use it after that you can use the hook `useRecoilValue` passing the `selector`.

```
const user = useRecoilValue(queryUser)
```

One important thing is that recoil is compatible with `React.Suspense` so we can make this

```
const CurrentUser = () => {
    const user = useRecoilValue(queryUser)
    return (
        <div>
            <h1>{user.name}</h1>
            <small>{user.email}</small>
        </div>
    )
}


const ComponentA = () => {
    <React.Suspense fallback={<div>Loading...</div>}>
        <CurrentUser />
    </React.Suspense>
}
```

My opinion about the options

I think that the best options are `Redux`, `Zustand` and `Recoil`. `MobX` is very mature but it also seems to be not very loved by the community, also since it doesn't define a strict interface to modifies the states it could be hard to trace back a change. `Context` is great for small apps and simple states but there are a lot of things that should be done completely manually.

Recoil

Pros

1. Easy to learn (Compared to Redux).
2. It is similar to basic React.
3. It is less problematic to change our mind.

Cons

1. It is very new current version.
2. It has a small community.

Zustand

Pros

1. Easy to learn (Compared to Redux).
2. Based on stores similar to Redux.
3. Support multy stores and async await.
4. Support state management cahnges that doesnt trigger re renders in the subscribers.
5. The listeners to the state changes could be simple functions (not just components)

Cons

1. Currently the documentation is very small.
2. It has a small community.
3. The several stores should be managed individualy (for example to clean state)