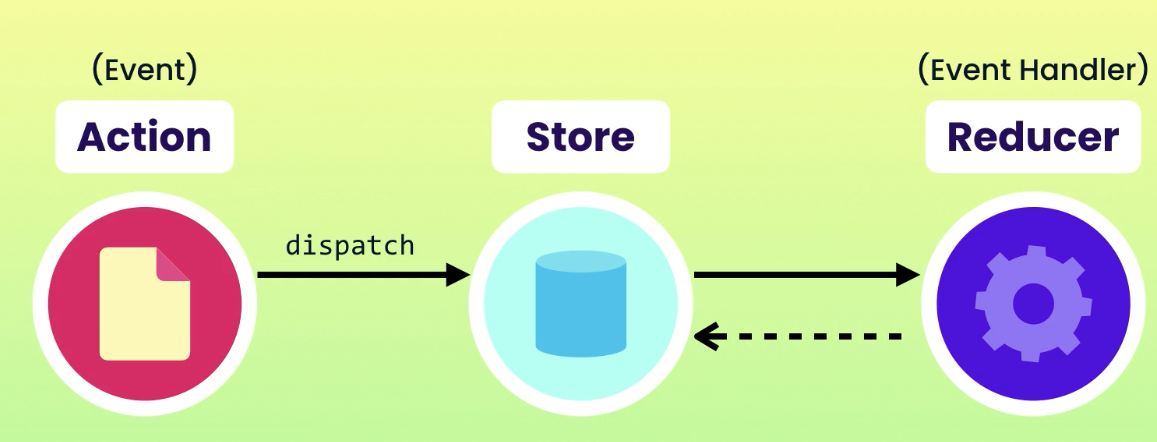
**Store:** In redux we store our application state in a single JavaScript object called store. It is accessible by all parts of our UI. We can use any type of data inside this store object. Store is an **immutable** object.

**Reducer:** To update the store, we need a function that takes the store as argument and returns the updated store. This function is called a reducer. These reducers are pure functions with two parameters -> the current state and the action.

**Action:** These are plain JavaScript objects that describes what changed/event happened, for example- user logged in, added an item to cart etc. **type** is the only property redux expects in the action object, so we must assign it. Type can be any data type that is serializable. Rest of the properties redux doesn’t care about (I’ll use payload as 2nd property).



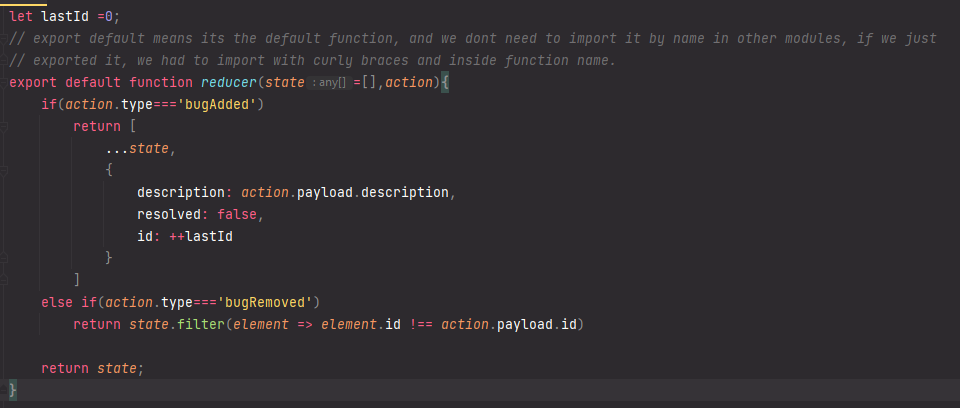
1. When user performs an action, they create an action object and dispatch it.
2. The store object has a dispatch method that takes an action. It will then forward the action to reducers. As we see, we don’t work with the reducers directly, we just work with the store which is in charge of calling the reducer.
3. The reducer computes the new state and returns to the store.
4. Next the store set the state internally and notify the UI component about the update.
5. The UI components then get the updated data and refresh themselves.

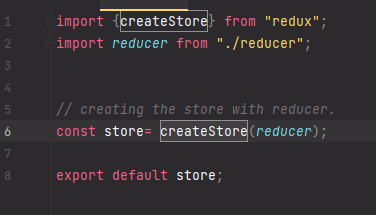


* This architecture has some benefits. Since all actions are dispatched through same pipeline, we can easily log everything. We can implement undo and redo as well.
* Four steps to follow when building a redux application

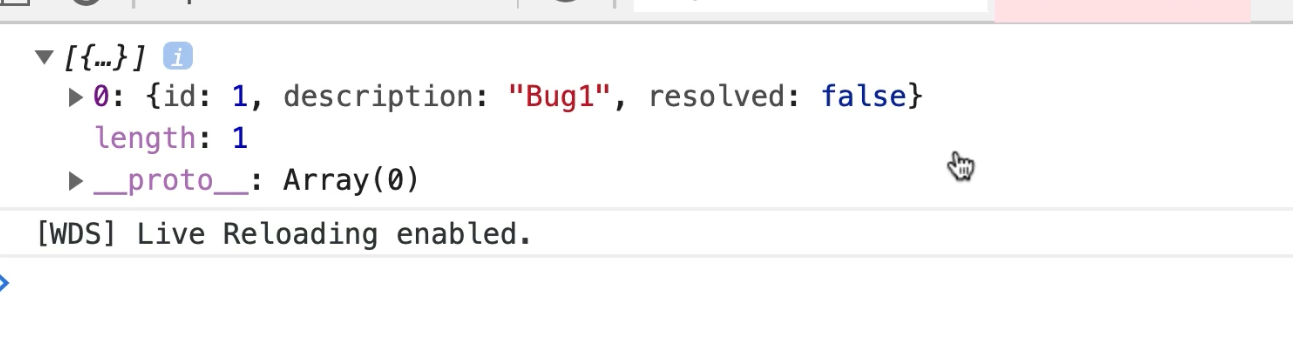


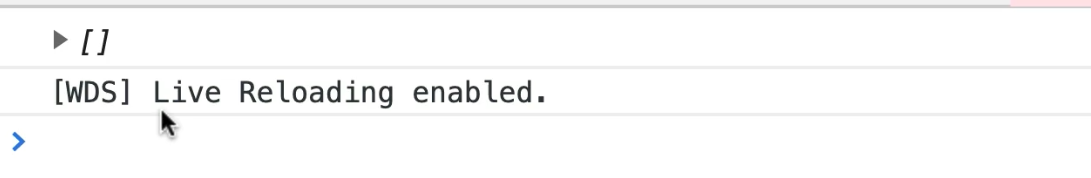
**Creating reducer and store:**



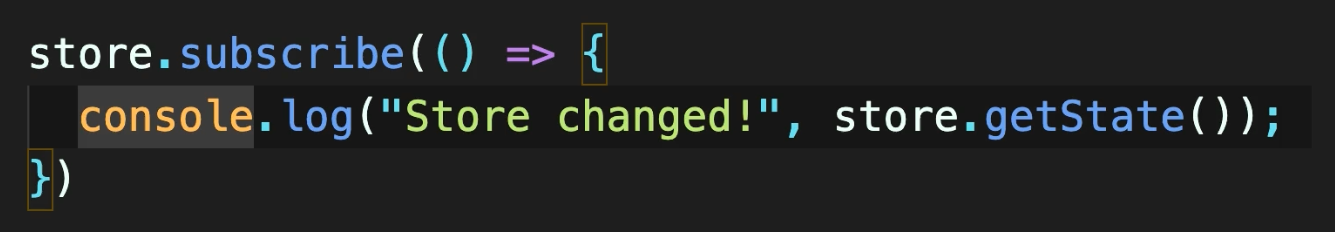
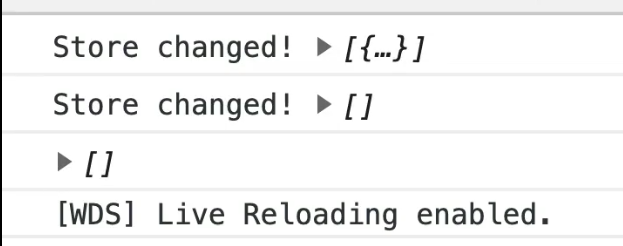


**Dispatching actions:**

 -> 

 -> 

**Subscribing to Store**: Whenever state of the store changes, we want to refresh the UI (in react re render/ in JavaScript update Dom). The UI components that subscribe to a store gets notified when the state of the store changes.

 -> 

This subscribes function returns a unsubscribe function that we can use to **unsubscribe** from a store. If we want to unsubscribe, we just need to call that returned function and we won’t be notified of that store change anymore. This is important cause otherwise it can create memory leak.

* Another good practice is to store the **action Types** into a single file as constant and use them, otherwise it will be hard to change them later.
* Another good practice is to make **action creators** functions in single file, so we can use them everywhere.
* Next good practice is to make **the action, action types and reducers** inside store directory, separated from the UI directories. As our application grow, there will be 100s of actions as well, so for each feature we need a directory inside store directory for action, action type and reducers.
* DUCKS pattern is another solution to above problem. Not going to use it for now

**Redux Toolkit:** this is a starter toolkit which is very handy, gives us bunch of functions predefined. There is bunch of stuff there I will have to check later, but createSlice() is the function we need to work with. It creates the reducers and actiontypes for us. Internally it calls functions **createaction** and **createreducer** for us.

It returns an object with name, reducer (), actions {} and caseReducers{}