**Redux Core components:**

Store:

Offerto da Redux.

Reducers: Da implementare e passare.

Pure functions che prendono in input (state, action) e restituiscono un nuovo state.

Actions:

Eventi. (type, payload)  
 Vengono generati dai componenti di React in conseguenza di eventi applicazione.

Lo store e’ al centro di queste interazioni.  
Deve essere disponibile in tutti i componenti che vogliono condividere lo state.  
Le Api dello Store utilizzate sono:

.dispatch (action)  
 .subscribe(listener) -> callback function to react to state changes detected.

**Come e’ implementato uno Store.**

Vedi implementazione basic

**Connect Api:**

Uses an High Order Component as a wrapper around the component who needs to connect with the redux store.

This component implements the boiler plate code necessary for the 2 interactions between React Component and Redux Store:

1. Component dispatch actions -> Store
2. Store sends updates in state to all subscribers.listener-> Component.

<https://react-redux.js.org/api/connect>

1. connect()(MyComponent)
2. connect(mapStateToProps)(MyComponent)
3. connect(  
   mapStateToProps,  
   mapDispatchToProps,

mergeProps,

options)(MyComponent)

**Store sends updates in state to all subscribers.listener-> Component.**

mapStateToProps?: (state, ownProps?) => Object

Implementazione di questo metodo mappa la porzione dello state a cui il componente e’ interessato in un oggetto che verra’ unito (merge) con this.props.

mapDispatchToProps(dispatch)?: Function | Object

Il Componente collegato allo store necessita di inoltrare un evento locale allo store tramite . .dispatch(action)

Questo tipicamente si traduce in N metodi event handlers locali, che devono tradurre il loro evento in un action e chiamare il dispatcher.

Il metodo di connect permette di automatizzare questo collegamento (ripetitivo) creando una mappatura tra Componente.eventHandler -> dispatch(eventAction).

Tipo:

const mapDispatchToProps = dispatch => {

return {

// dispatching plain actions

increment: () => dispatch({ type: 'INCREMENT' }),

decrement: () => dispatch({ type: 'DECREMENT' }),

reset: () => dispatch({ type: 'RESET' })

}

}

E’ opzionale.

Se non implementato l’suo di connect permettera comunque nel componente di poter accedere alla funzione dello store tramite this.props.dispatch();

**ComponentContainer:**Produce props per costruire componente collegato col connect.

Si occupa di automatizzare le due iterazioni tra ComponentUI e Store:

1. Store.dispatch(UI event):  
     
   ComponentUI  
   >callBackEvent>   
     
   Action: (event2Action)  
   actionObj>

Store:  
store.dispatch (Action)>

Reducer:

store.updateState

1. Store.subscribe(listenerFunc)  
   listenerFunc>store.getState() updates prop> ComponentUI.render(newProps)

**Connect api (collega Component a Store)**

Sono gestite rispettivamente dalle function:

1. mapDispatchToProp(dispatchFunc,prop) bind UI.componentCallBack -> dispatch(prop2action)
2. mapStateToProps(state, ownProps) return prop obj
3. collegare con

connect(mapDispatchToProp, mapStateToProps) (componentUI)

Spiegazioni

1. state e’ quello corrente in Store.

**TUTORIAL:**

1. **Starter**  
   ComponentContainer (es PageComponent)  
     
   GroupPage crud   
   .create  
   .saveChanges  
   .undoChanges  
   .delete  
   .loadAll (from store)
2. **Obiettivo: Collegare eventi a store:**

Container.eventCallback -> = store.dispatch(action)  
  
Impl:  
eventCallBack(e)->{

1) var action = <ActionCreator>.<createAction>(e)   
2) store.dispatch(action)

}

Ogni callBack ha una funzione specifica come Action in un ActionCreator(per Container)

Nel nostro caso:

**Action:**

GroupActions.create

GroupActions.saveChanges

GroupActions.delete

1. Inserire tutti i callback eventi in un unica funzione: