toggle

9. REDUX

Perchè Usarlo

Pattern Gestione dello Stato

- Maggiore controllo
- Minori Side Effects

Rende lo stato di un'applicazione "predictable"

- Per cambiare lo stato, deve essere lanciata un'azione
- Lo store contiene solo getter, non setter

Redux DevTools

- Lo stato applicativo non può essere modificato direttamente
- Ispezione dello stato applicativo
- Storico delle azioni eseguite e conseguenti effetti sullo stato
- Time-Traveling
- Export / Import

Applicazioni React potrebbero essere controllate via this.setState()

- Difficoltà di manutenzione

Componenti

Store

E' un oggetto che contiene l'intero stato dell'applicazione

Reducers

Lo stato non è modificabile direttamente

- Lo store interroga i reducers per sapere come un'azione modifica lo stato
- Il root reducer prende in carico la richiesta e suddivide lo stato in singoli oggetti identificati da una chiave
- Ogni porzione viene passata ad uno specifico Reducer che ne gestirà le operazioni
- Viene creata una copia di ogni singola porzione, e alla fine vengono unite tutte le porzioni in un nuovo State Object — `combineReducers()`

```
import {
  GET_DATA,
  SUCCESS_GET_DATA,
  ERROR_GET_DATA
} from '../actions/whatever';

const initialState = {
  items: [],
  status: null,
}

export default (state = initialState, action) => {
  switch (action.type) {
    case START_GET_DATA:
      return {
        ...state,
        status: 'LOADING'
      }

    case SUCCESS_GET_DATA:
      return {
        ...state,
        items: action.data,
        status: 'SUCCESS'
      }

    case ERROR_GET_DATA:
      return {
        ...state,
        status: 'ERROR'
      }

    default:
      return state
  }
}
```

Possono ricevere actionCreators da action diverse

Views

Smart Components

triggerano le actions

- `bindActionCreators()`
- Propagano le informazioni ai Componenti

Dumb Components

Non triggerano direttamente una action

- Ricevono le informazioni dagli Smart Components
- Chiamano una proprietà dello Smart Component padre, che effettuerà il trigger dell'azione
- Le azioni sono passate via props

View layer binding

Connette lo store alle view

- `connect()`
- `<Provider />`

Root Component

- Governa l'intero flusso di Redux
- Genera lo Store
- Smista le informazioni al reducer corretto
- Connette il View Layer con le Views

Action Creators

Ritornano una funzione

- L'unico modo in cui lo stato può essere modificato

```
export function throwAction(data) {
  return {
    type: ACTION_IDENTIFIDR,
    data
  }
}
```

- Vengono triggerate tramite **dispatch(action())**

Redux Thunk

- E' un middleware che permette di scrivere actions che ritornano una funzione, anzichè una action direttamente.
- Una **thunk** è un funzione che contiene un'espressione il cui valore viene eseguito con un delay
- Viene usato per ritardare il dispatch delle azioni, o per eseguire solo in determinate condizioni
- `npm install --save redux-thunk`

Real World Example

```
/*
 * action types
 */

export const START_GET_DATA = 'START_GET_DATA'
export const SUCCESS_GET_DATA = 'SUCCESS_GET_DATA'
export const ERROR_GET_DATA = 'ERROR_GET_DATA'

/*
 * action creators
 */

export function startGetData() {
  return { type: START_GET_DATA }
}

export function successGetData(text) {
  return { type: SUCCESS_GET_DATA, text }
}

export function errorGetData(text) {
  return { type: ERROR_GET_DATA, text }
}

export function loadData() {
    return (dispatch) => { //REDUX THUNK
        dispatch(startGetData());
        axios.post(url,data).then((data) =>
            dispatch(successGetData(data)) //Action dispatch
        ).catch(error =>
            dispatch(errorGetData(error)) //Action dispatch
        )
    }
}
```

Il Flusso dei Dati

Action creator formatta la richiesta e ne esegue il return

View triggera una actions

Viene eseguito il dispatch() della action

Lo store riceve la action

Invia lo stato attuale e la action al Root Reducer

Passa ogni porzione ai singoli reducer che gestiranno l'informazione

Il Root Reducer suddivide lo stato in porzioni

Invia la copia della propria porzione di stato al Root Reducer

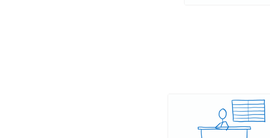
Ogni reducer copia la porzione di stato, ed effettua le modifiche sulla copia

Lo store sostituisce lo stato con la nuova versione

Quando tutti i Reducers hanno inviato la propria porzione di stato, il Root Reducer unisce tutte le copie che formeranno il nuovo stato, e lo invia allo Store

Il View Layer Binding chiede allo Store di inviargli lo Stato

Lo Store avvisa il View Layer Binding della presenza di una nuova versione dello Stato



Il View Layer Binding triggera un rendering di tutti i componenti coinvolti nel cambio di stato.

Esempio Completo di Setup

```
npm i --save redux react-redux redux-thunk connected-react-router
npm i --save-dev redux-logger redux-devtools-extension
```

Perchè si usa

- Sincronizza lo stato del Router con lo Store di Redux
- Permette di eseguire dispatch di metodi dell'history
- push('/path')
- Ispezionabile con Redux-devtools

```
npm install --save connected-react-router
```

Redux With React-Router

store.js

```
//STEP 1
//store.js
import { createStore, applyMiddleware } from 'redux';
import thunkMiddleware from 'redux-thunk';
import { createLogger } from 'redux-logger';
import {
    connectRouter,
    routerMiddleware
  } from 'connected-react-router';
import {composeWithDevTools} from 'redux-devtools-
extension/logOnlyInProduction';
import rootReducer from './reducers';
import history from './utils/history';

const composeEnhancers = composeWithDevTools({
  name: 'MyApp'
});

const loggerMiddleware = createLogger({predicate: (getState, action) => true})

export default function store(preloadedState) {
  const store = createStore(
    connectRouter(history)(rootReducer), //History e Stato vengono connessi
    preloadedState,
    composeEnhancers(
      applyMiddleware(
        routerMiddleware(history),
        thunkMiddleware,
        loggerMiddleware
      )
    )
  )

  if (module.hot) {
    module.hot.accept('./reducers', () => {
      store.replaceReducer(connectRouter(history)(rootReducer));
    });
  }

  return store;
}
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { AppContainer } from 'react-hot-loader'
import './index.css';
import App from './containers/App';

//history e store vengono importati e inizializzati dalla index
import history from './utils/history';
import configureStore from './store';

const store = configureStore();

const render = Component => {
    ReactDOM.render(
      <AppContainer>
        //history e store vengono passati come proprietà al componente
principale
        <Component store={store} history={history} />
      </AppContainer>,
      document.getElementById('root'),
    )
  }

  render(App)

  // Webpack Hot Module Replacement API
  if (module.hot) {
    module.hot.accept('./containers/App', () => {
      render(App)
    })
  }
...
```

App.js

```
//STEP3
//App.js
import { Provider } from 'react-redux'
import { Route, Switch } from 'react-router'
import { ConnectedRouter } from 'connected-
react-router'

render() {
    //history e props vengono ricevuti come
proprietà
    const {history, store} = this.props;

    return (
        <Provider store={store}>
            <ConnectedRouter history=
{history}>
                <Switch>
                ...
                </Switch>
            </ConnectedRouter>
        </Provider>
    )
}
...
```

Middlewares

applyMiddleware(...middleware)

Istruzioni date "in pasto" allo Store

Permettono di estendere le funzionalità di base dello Store

- redux-thunk
- dispatch(actionCreator())
- redux-devtools-extension
- redux-logger
- ImmutableJS

```
import { createStore, applyMiddleware } from 'redux'
import thunkMiddleware from 'redux-thunk'
import { createLogger } from 'redux-logger'
import { composeWithDevTools } from 'redux-devtools-extension/logOnlyInProduction';
import rootReducer from './reducers';

const composeEnhancers = composeWithDevTools({
  name: 'MyApp'
});

const loggerMiddleware = createLogger({predicate: (getState, action) => true})

export default function store(preloadedState) {
  const store = createStore(
    rootReducer,
    preloadedState,
    composeEnhancers(
      applyMiddleware(
        thunkMiddleware,
        loggerMiddleware
      )
    )
  )

  if (module.hot) {
    module.hot.accept('./reducers', () => {
      store.replaceReducer(rootReducer);
    });
  }

  return store;
}
```

Esercizio

- Installare redux , react-redux, redux-thunk
- Configurare lo store in un file **store.js** nella root applicativa
- Configurare il Provider in App.js
- Predisporre le actions per il fetch dei dati
- Creare un file /actions/actions.js
- Configurare i reducers per la porzione di state 'expenses'
- Creare un file /reducers/expenses.js
- Configurare combineReducers()
- Creare un file /reducers/index.js
- Modificare il container Main in modo da eseguire la action per il fetch dei dati, e aggiornare le props provenienti dai reducer

```
npm install --save redux react-redux
```

```
import {createStore} from 'redux';
import reducers from './reducers';

const store = createStore(reducers);

export default store;
```

Crea lo Store

Root Component

1. Preparare lo Store

```
import { combineReducers } from 'redux';
import expenses from './expenses';

const rootReducer = combineReducers({
    expenses
});

export default rootReducer;
```

combineReducers() Unisce i Reducers

Connette lo store ai reducers

```
import {Provider} from 'react-redux';
import store from '../../store';

export default class App extends Component {
    render() {
        return <Provider store={store}>
                <Router>
                ...
                </Router>
        </Provider>
    }
}
```

Crea un Network per mantenere aggiornati i component

View Layer

2. Far comunicare lo Store con i Components

```
st mapStateToProps = state => ({
tatus: state.reducer.status, //Now we can access to this.props.status
tems: state.reducer.items //Now we can access to this.props.items
```

```
//reducer
export function getStatus(state) {
  return state.reducer.status;
}

export function getItems(state) {
  return state.reducer.items;
}

//Component
import {getStatus, getItems} from
'../reducer/reducer.js';

...

const mapStateToProps = state => {
    const status = getStatus(state);
    const items = getItems(state);

    //Now we can access to this.props.status and
this.props.items
    return {
        status,
        items
    }
}
```

riceveno i dati dai reducers

Components

connect() ()

si connettono al Network creato dal provider

```
import {bindActionCreators} from 'redux';
import {loadData} from '../actions/action.js';

function mapDispatchToProps(dispatch) {
  return bindActionCreators(
    {
      loadData //Now we can trigger this.props.loadData()
    },
    dispatch
  );
}
```

```
import {connect} from 'react-redux'; //Import lib connect from redux
import {bindActionCreators} from 'redux';
import {loadData} from '../actions/action.js';
import {getStatus, getItems} from '../reducer/reducer.js';

...

const mapStateToProps = state => {
    const status = getStatus(state);
    const items = getItems(state);

    return {
        status,
        items
    }
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators(
    {
      loadData
    },
    dispatch
  );
}

//Putting all togheter
export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

Smart Components

3. Predisporre le callback per le actions