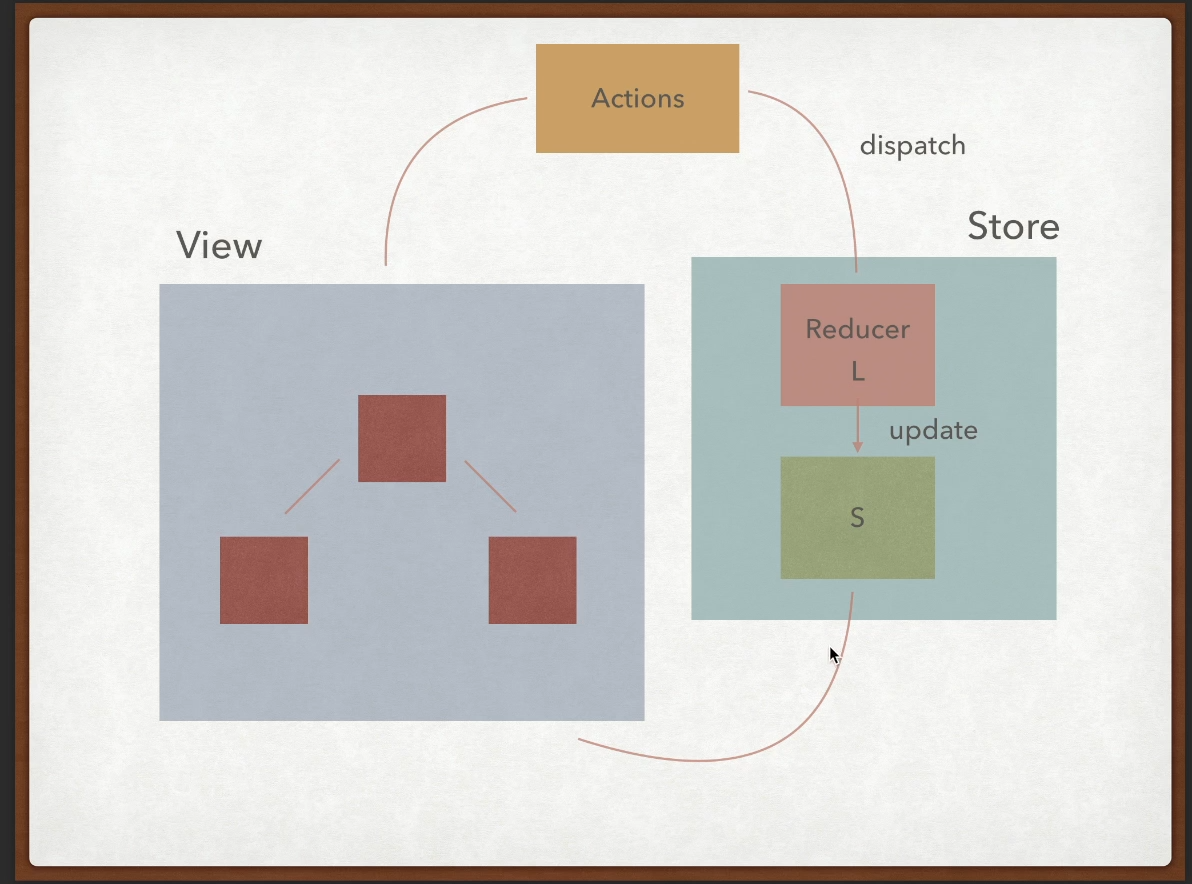
***Redux.***

У нас есть state и View (наши компоненты). Также мы вынесем логику по изменению нашего стейта и назовем его reducer L. Он будет знать как обновлять стейт при абсолютно любом действии. Теперь нам остается передавать только запросы к нему из каждого компонента. Делаем это при помощи Actions. Компонент Store объединяет Reducer и State.

Те сначала создает action, который знает ЧТО нужно обновить в стейте. Далее он попадает в Reducer, который знает КАК его обновить и он уже обновляет стейт.

Передача из Actions в Reducer называет dispatch.

<https://redux.js.org/assets/images/ReduxDataFlowDiagram-49fa8c3968371d9ef6f2a1486bd40a26.gif>

***Чистые функции***

Она должна соблюдать два условия:

1. Она должна возвращать одинаковый результат, когда в нее приходит одинаковые данные
2. Она не вызывает внутри себя побочных эффектов.

Пример:

const rndSum = (a) => Math.random + a;

console.log(rndSum(5));

console.log(rndSum(5));

console.log(rndSum(5));

Это не чистая функция

2 нельзя например использовать запросы на сервер, лог, работа с дом деревом, изменение файлов, изменение входных данных !

Оптимизация через actionCreators и bindActionCreator

Просто создадим функцию bindActionCreator, чтобы понимать как она работает. Она принимает в себя creator и dispatch. А дальше эта функция должна возвращать функцию. И в ней dispatch(creator), который вызывает, поскольку creator это action креэйторы, которые вызываются в actions.js должны вернуть объект с каким-то типом. И если у нас возвращает функция, то мы должны добавить агрументы через рест оператор …args.

Встроенная в redux такая же функция позволяет забиндить сразу несколько функций в виде объекта. Те на выходе у нас будет объект ключами которого будут все те же функции.

 Соединяем React и Redux при помощи connect

Connect используется чтобы связать стор с каким то компонентом.

Connect это компонент высшего порядка, те это такие функции, которые оборачивают какой-то компонент и возвращают этот компонент с измененными параметрами (добавление пропсов, функционала итд).

function connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)

Принимает в себя несколько функций:

1. mapStateToProps – она вытаскивает определённые пропсы из стейта и передавать их в пропсы нашего компонента. Принимает в себя state (по факту наш initialState). Возвращает то, что нам нужно (в нашем случае counter:state.value)
2. Для того, чтобы передать сами функции (inc, dec) используется mapDispatchProps. Возвращает что нам нужно:

const mapDispatchToProps = (dispatch) => {

  const { inc, dec } = bindActionCreators(actions, dispatch)

  return {

    inc,

    dec: () => {

      const value = 2

      dec(value)

    },

  }

}

Функция reducer должна быть чистой, однако мы вручную создали функции, которые называются action creator’ами, на которые это ограничение не распространяется. Те все запросы на сервер, случайные числа мы можем поместить в эту функцию.

Важная особенность функции connector. Если вторым аргументом передать не функцию, а объект, то она обертнет все наши action creator’ы, которые приходят в качестве объекта уже в функцию dispatch и сделает это автоматически и можно сделать вот так:

export default connect(mapStateToProps, actions)(Counter)

Соединяем React и Redux при помощи хуков

  const { counter } = useSelector((state) => state)

  const dispatch = useDispatch()

***Отличие***

1. Хук useSelector может вернуть все что угодно, а не только объект, который потом идет на пропс. В колбек функции мы можем делать все что угодно (но она должна быть чистой и синхронной).
2. В useSelector нет возможности передать собственные пропы для отслеживания.
3. Когда вызывается dispatch в useSelector, в таком случае хук ссылочно проверяет что если знач не изменилось, то и компонент не будет ререндериться.
4. Разница в механизме сравнения, когда мы хотим получить комбинированный объект из стора

**Типа state => {**

**return {**

**counter: state.counter**

**}**

**}**

И в таком случае хук будет сравнивать не по отдельным полям, а по всему объекту (строгое сравнение). Любое изменение стейта будет вызывать перерендеринг (тк всегда новый объект), это так сказать проблема оптимизации при использовании хуков.

***Решение проблемы оптимизации***

1. Использовать несколько раз useSelector, для того чтобы получать разные значения:

**Const a = useSelector(state => state.a)** и так ля каждой пер.

1. Использовать стороннюю библиотеку.
2. Добавить функцию shallowEqual, которая внутри реакт-редакс, и поместить вторым аргументом в useSelector.

Мы будем использовать 2.

***Углубление в useDispatch***

Если мы собираемся передавать действие ниже по иерархии, то всегда нужно использовать useCallback.

***Зомби дети <протухшие пропы>***

Может редко возникать при использовании хуков.

<https://vadim-budarin.medium.com/react-%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%BD%D0%BE-%D0%BE-zombie-children-and-stale-props-d31247ea08>

***Redux devtools***

Добавить window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

в createStore.

Правило названия action

И так на данном моменте у нас есть crud приложение на реакт.

Все типы действий в верхнем регистре. Каждое послед слово через \_. Кроме этого если функционал подразумевает запрос на сервер с изменением каких-то состояний, то обычно прописывается три разных состояния:

1. \_FETCHING
2. \_FETCHED
3. \_FETCHING\_ERROR

Задание:

1. Удалить по кнопке элемент из списка. Для этого я сначала добавил action метод и передал туда АЙДИ того элемента, который хотим удалить !!! Далее в reducer я добавил case в котором по классике, точно также я бы сделал в обычном реакте, просто сейчас мы все эти методы выносим в reducer. Опять же просто формируем массив, в который добавляем все элементы кроме удаленного, те применяем фильтр по айди. Теперь в самом компоненте мы сознаем функцию обратного вызова, чтобы передать в компонент ниже по иерархии (list -> item).
2. Добавление персонажа. Для этого сначала добавим action и потом reducer. Далее нам нужно собрать данные с формы, для этого не нужно использовать redux, а достаточно локального стейта. Создаем поля для формы или подключаем формик. Создаем управляемые инпуты. Далее вешаем обработчик на форму и создаем функц отправки, где создаем новый объект и отправляем данные на сервер и также вызываем dispatch heroCreated, те сохраняем в redux в сторе.
3. Загрузка фильтров с сервера. Ну то есть в файле HeroesFilter мы также посылаем запрос на сервер и потом изменяем через dispatch состояние в сторе фильтров.
4. Фильтрация. В редьюсере мы записываем не только поле выбранного фильтра, но и СРАЗУ же фильтруем наш массив. Фильтрация должна происходить когда мы добавляем/удаляем/выбираем фильтр героев и еще когда только получаем их.
5. Класс active

***Комбинирование reducers и красивые селекторы. CreateSelector()***

Проблема: у нас может разрастаться reducer.

Решение: combineReducers.

Можно разделить героев и фильтры в разные файлы, но это плохая идея, тк все переплетено.

Нам нужно разделить логику, более лучше применить селекторы, сделать reducer более чистым.

Например фильтрацией мы можем заниматься еще на этапе useSelector. Те на этапе получения каких-то данных со стора, а не держать эти действия внутри reducer. Те мы ее вынесли из reducer:

  const filteredHeroes = useSelector((state) => {

    if (state.activeFilter === 'all') {

      return state.heroes

    } else {

      return state.heroes.filter((item) => item.element === state.activeFilter)

    }

  })

Теперь когда у нас в этом файле нет никаких связей, мы можем разделить его на два разных.

Для этого в store:

const store = createStore(

  combineReducers({ heroes: heroesReducer, filters: filtersReducer }),

  window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

)

ОДНАКО вариант с вынесением логики фильтрации в компонент также явл плохой практикой, тк каждый раз при изменении глобального стейта (например при нажатии одного и того же фильтра) у нас вызывается useSelector.

Здесь нужно использовать мемоизацию, а именно библиотеку Reselect, функции createSelector. Она сама будет мемоизировать.

И так:

  // лучший вариант

  const filteredHeroesSelector = createSelector(

    // результат первой функции - получение activeFilter из стейта

    (state) => state.filters.activeFilter,

    (state) => state.heroes.heroes,

    (filter, heroes) => {

      if (filter === 'all') {

        return heroes

      } else {

        return heroes.filter((item) => item.element === filter)

      }

    }

  )

  const filteredHeroes = useSelector(filteredHeroesSelector)

***Store enhancersc – усилители стора***

Так, мы знаем что в dispatch мы можем передавать только объекты у которых должно быть св-во type и доп поля. Но иногда нам нужно передать не объект, а строку или функцию.

Чтобы решить эту проблему, в сторе работаем:

const enhancer = (createStore) => (...args) => {

  const store = createStore(...args)

  // идет шаблонный код из решений

  const oldDispatch = store.dispatch

  store.dispatch = (action) => {

    if(typeof action === 'string'){

      return oldDispatch({

        type: action

      })

    }

    return oldDispatch(action)

  }

  return store

}

Здесь идет работа с большим объектом store.

Механизм который работает также, но меняет только работу функции dispatch называется middleware (используются чаще, код готов в npm)

***Middleware***

const stringMiddleware =

  ({ dispatch, getState }) =>

  (next) =>

  (action) => {

    if (typeof action === 'string') {

      return next({

        type: action,

      })

    }

    return next(action)

  }

***Redux-thunk***

Позволяет в качестве действий отправлять не объекты, а функции. В свою очередь если будут диспетчится функции, то внутри мы можем делать все что угодно, в том числе и асинхронные операции.

И так, представим что нам нужно фильтровать с задержкой. Кончено мы бы могли обернуть вызов в сеттаймаут, но это не совсем правильно:

onClick={() => dispatch(activeFilterChanged(name))}

Реализуем это с помощью фанков. Мы знаем, что мы можем передать теперь и функцию. Когда мы используем фанк миддлваре, dispatch приходит автоматически. Те:

export const activeFilterChanged = (filter) => (dispatch) => {

  setTimeout(() => {

    dispatch({

      type: 'ACTIVE\_FILTER\_CHANGED',

      payload: filter,

    })

  }, 3000)

}

Теперь разберем более реальный случай.

У нас есть действие по получение данных с сервера, и проблема в том, что мы часто повторяем dispatch и запрос. Почему бы нам не создать одно действие, которое будет включать все задачи. Как раз мы можем создать action creator в виде функции. Те мы создадим комплексный action creator, который сделает все за нас:

// запрос

export const fetchHeroes = (request) => (dispatch) => {

  dispatch(heroesFetching())

  request('http://localhost:3001/heroes')

    .then((data) => dispatch(heroesFetching(data)))

    .then(() => dispatch(heroesFetchingError()))

}

Теперь мы можем убрать лишний код:

  useEffect(() => {

    dispatch(fetchHeroes(request))

  }, [])