нетология

REDUX OBSERVABLE

АЛЕКСАНДР СИВЦОВ

АЛЕКСАНДР СИВЦОВ

Frontend-разработчик в Aliexpress Russia

Александр Сивцов

ПЛАН ЗАНЯТИЯ

1. RxJS
2. Observable
3. RxJS операторы
4. Higher Order Observables
5. Redux Observable
6. Epics
7. Обработка ошибок

ЗАДАЧА

Мы с вами уже разобрались, как делать HTTP-запросы, CRUD и т.д.

Даже посмотрели на то, как делать примитивный Polling.

Теперь же задача перед нами следующая: реализовать поиск на сайте. Именно поиск, а не фильтрацию.

Это значит, что мы должны отправлять HTTP-запрос на сервер, дожидаться ответа и отображать результаты.

Если у вас предложения, как это сделать?

ЗАДАЧА

Конечно, можно сделать кнопку "Поиск" и отправлять запрос на сервер только тогда, когда пользователь её нажмёт.

Но современные пользователи привыкли к другому виду взаимодействия: что поиск осуществляется паралельно с вводом текста.

Но давайте подумаем, к чему это может привести?

ЗАДАЧА

Проблем будет несколько:

- из-за сетевых задержек мы можем получать неактуальные результаты (см. лекцию про хуки, там мы это решали "отменой" предыдущих запросов)
- флудом сервера, т.к. если мы будем реагировать на каждое изменение, то на слово "react" нам придётся послать 5 запросов!

Для борьбы со второй проблемой придётся ставить таймауты, т.е. если пользователь ввёл несколько букв и остановился - то можем посылать запрос, а если продолжил - то нужно очищать предыдущий таймаут и выставлять новый.

Сложновато, не правда ли?

Возможно, уже есть инструмент, который позаботился обо всём этом?

СЕРВЕР

Как всегда, нам нужен будет сервер для тестирования взаимодействия.

Мы специально его сделаем медленным со случайными задержками.

```
npm init
npm install forever koa koa-router koa2-cors koa-body
```

МЕДЛЕННЫЕ СЕРВЕРА

Не стоит думать, что проблема надумана и решается покупкой "быстрого сервера" или рекомендацией клиенту купить "более быстрый интернет".

СЕРВЕР

.foreverignore:

```
node_modules
```

scripts в `package.json`:

```
"scripts": {
  "prestart": "npm install",
  "start": "forever server.js",
  "watch": "forever -w server.js"
},
```

API

```
const http = require('http');
const Koa = require('koa');
const Router = require('koa-router');
const cors = require('koa2-cors');

const app = new Koa();
app.use(cors());

let nextId = 1;
const skills = [
  { id: nextId++, name: "React" },
  { id: nextId++, name: "Redux" },
  { id: nextId++, name: "Redux Thunk" },
  { id: nextId++, name: "RxJS" },
  { id: nextId++, name: "Redux Observable" },
  { id: nextId++, name: "Redux Saga" },
];
```

```
let isEven = true;
router.get('/api/search', async (ctx, next) => {
  if (Math.random() > 0.75) {
    ctx.response.status = 500;
    return;
  }

  const { q } = ctx.request.query;
  return new Promise((resolve, reject) => {
    setTimeout(() => {

      const response = skills.filter(o =>
        o.name.toLowerCase().startsWith(q.toLowerCase())
      );
      ctx.response.body = response;
      resolve();
    }, isEven ? 1 * 1000 : 5 * 1000);
    isEven = !isEven;
  });
});

app.use(router.routes())
app.use(router.allowedMethods());

const port = process.env.PORT || 7070;
const server = http.createServer(app.callback());
server.listen(port);
```

RXJS

RXJS

`RxJS` - это готовая библиотека для создания приложений, работающий с асинхронными вызовами и событиями.

Ее можно установить через npm/yarn

```
npm install rxjs
yarn add rxjs
```

Либо можно подключить через CDN - тогда глобально будет доступна переменная `rxjs`:

```
<script src='https://unpkg.com/rxjs/bundles/rxjs.umd.min.js'></script>
```

Мы не будем использовать Webpack для сборки, поэтому будем работать через глобальную переменную `rxjs`.

RXJS

`RxJS` комбинирует достаточно много идей, но ключевыми являются две:

- все события представляются в виде распределённого во времени набора значений
- предоставляются расширенные операторы для преобразования потоков (представьте, что можно сделать распределённый во времени `Array.map`, `Array.filter`, `Array.reduce` и т.д.)

OBSERVABLE

OBSERVABLE

Observable - это поток из значений, распределённых во времени.

Значений может не быть совсем, может быть одно, а может быть более одного.

Это такая специальная абстракция, в которую (в нашем случае) можно уложить практически любое взаимодействие.

OBSERVABLE

Например, ввод пользователем строки "react" - это поток значений, распределённый во времени.

Каких значений - это уже зависит от того, как мы построим этот самый поток, например, это может быть: "r", "re", "rea", "reac", "react".

При этом поток распределён во времени, т.к. пользователь не мгновенно вводит текст.

Кроме того, поток может быть пустым (если пользователь не воспользовался строкой поиска).

OBSERVABLE

Ключевое в Observable - это то, что мы можем на него подписаться и получать уведомления обо всём, что с ним происходит:

```
observable.subscribe(
  // next
  value => console.log('next', value),
  // error
  error => console.error('error', error),
  // complete
  () => console.info('complete'),
);
```

Observables - ленивы (ничего не произойдёт, пока никто не подпишется на `Observable`).

OBSERVABLE

Кроме того, все три функции можно передать в виде одного объекта:

```
observable.subscribe({
  next: value => console.log('next', value),
  error: error => console.error('error', error),
  complete: () => console.info('complete'),
});
```

UNSUBSCRIBE

Отписываться от потока нужно тогда, когда нам больше не нужны данные в потоке. Например, больше не нужно получать данные из формы.

Для того, чтобы отписаться нужно использовать объект подписки, который мы получаем когда выполняем `subscribe`:

```
const stream$ = observable.subscribe(...);
// где-нибудь в willUnmount:
stream$.unsubscribe();
```

После `unsubscribe` все ресурсы освобождаются (в противном случае вы получите утечку ресурсов).

ПРИМЕР

Давайте рассмотрим всё на нашем примере, пока без React и Redux.

```
<script src='https://unpkg.com/rxjs/bundles/rxjs.umd.min.js'></script>
<script>
  const inputEl = document.createElement('input');
  document.body.appendChild(inputEl);
</script>
```

СОЗДАНИЕ OBSERVABLE

RxJS предлагает готовые функции для создания `Observable`:

- `fromEvent` - из события
- `ajax` - из AJAX-запроса
- `timer` - из срабатывания таймаутов при `setTimeout`
- `interval` - из срабатывания интервалов при `setInterval`
- `from` - помимо промиса может принимать массив значений и по одному отправляет их в поток
- `of` - создает поток из своих агрументов
- и другие

ПРИМЕР

```
<script src='https://unpkg.com/rxjs/bundles/rxjs.umd.min.js'></script>
<script>
const { fromEvent } = rxjs;
const { ajax } = rxjs.ajax;

const inputEl = document.createElement('input');
document.body.appendChild(inputEl);
```

```
const inputElChange$ = fromEvent(inputEl, 'input')
inputElChange$.subscribe({
  next: value => console.log('next', value),
  error: error => console.error('error', error),
  complete: () => console.info('complete'),
});

const params = new URLSearchParams({ q: 'Re' });
const search$ = ajax.getJSON(
  `http://localhost:7070/api/search?${params}`
);
search$.subscribe({
  next: value => console.log('next', value),
  error: error => console.error('error', error),
  complete: () => console.info('complete'),
})
</script>
```

`$` - это общепринятое соглашение для обозначения переменных, указывающих на потоки.

OBSERVABLE

Мы получили два потока:

1. `inputChange$` - эмитирует столько значений, сколько раз пользователь изменит поле, никогда не заканчивается и не генерирует ошибок.
2. `search$` - эмитирует ровно одно значение, после чего завершается, либо ошибку*.

Примечание:* `ajax.getJSON` сам обрабатывает коды не 2xx, генерируя ошибку.

OBSERVABLE

Но нам нужно, чтобы почти на каждое изменение поля ввода мы могли посылать запрос на поиск!

RxJS предоставляет нам возможность выстраивать так называемый `pipe` для `Observable`, фактически, выстраивая конвеер обработки:

- трансформировать значения
- вставлять задержку по времени, выбирая только последние значения в заданном временном окне
- запускать новые потоки в ответ на пришедшие значения и обрабатывать их
- и т.д.

Всё это делается с помощью операторов.

RXJS OPERATORS

MAP & FILTER

Самые простые операторы уже знакомы вам по массивам - `map` и `filter`:

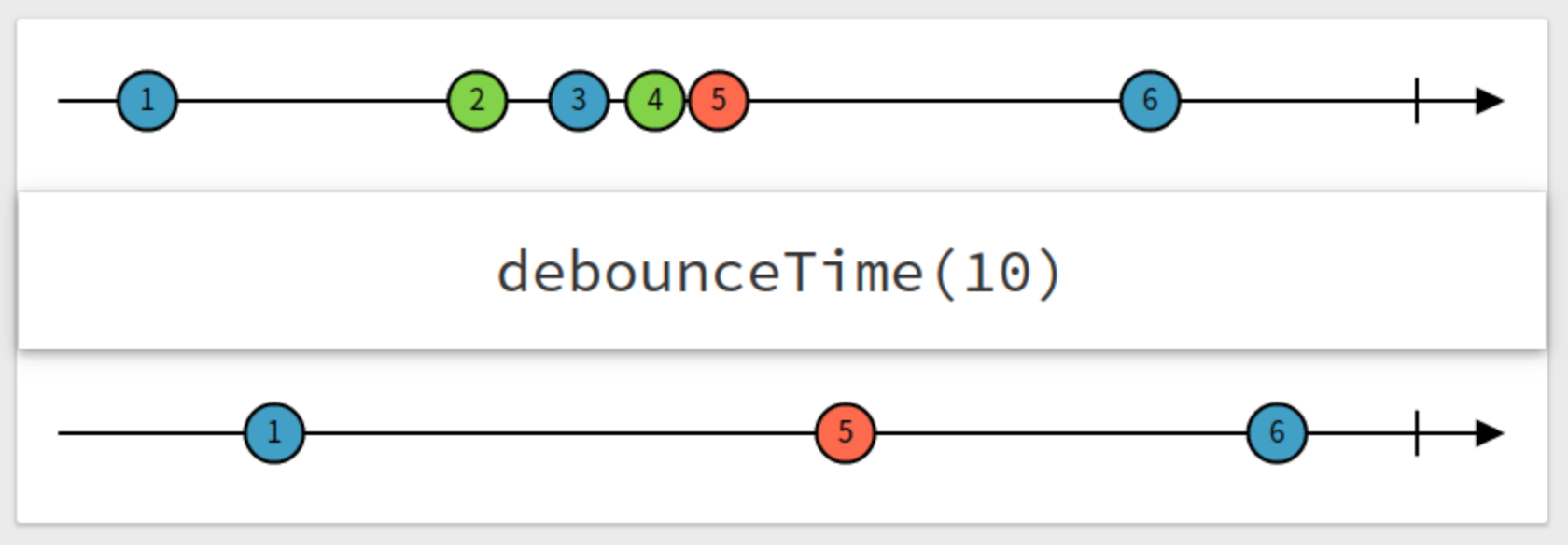
```
const { fromEvent } = rxjs;
const { ajax } = rxjs.ajax;
const { map, filter, debounceTime } = rxjs.operators;
const { mergeMap, concatMap, exhaustMap, switchMap } = rxjs.operators;

...

const inputElChange$ = fromEvent(inputEl, 'input')
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
).subscribe({
  next: value => console.log('next', value),
  error: error => console.error('error', error),
  complete: () => console.info('complete'),
});
```

DEBOUNCETIME

`debounceTime` позволяет "задерживать" значения, поступающие в поток, при этом отбрасывает предыдущие значения, если появились новые:



Очень хорошо в изучении операторов и функций помогает сайт RxMarbles, с которого мы и взяли это изображение.

```
const { fromEvent } = rxjs;
const { ajax } = rxjs.ajax;
const { map, filter, debounceTime } = rxjs.operators;
const { mergeMap, concatMap, exhaustMap, switchMap } = rxjs.operators;

...

const inputElChange$ = fromEvent(inputEl, 'input')
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
  debounceTime(100),
).subscribe({
  next: value => console.log('next', value),
  error: error => console.error('error', error),
  complete: () => console.info('complete'),
});
```

Теперь значение генерируется не чаще 1 раза в 100 мс (попробуйте нажать клавишу в поле поиска и не отпускать).

HOO - HIGHER ORDER OBSERVABLES

HOO - OBSERVABLE ВЫСШЕГО ПОРЯДКА

Существует возможность переключения потоков и их комбинирования. Например, при появлении значений в одном потоке, запустить другой поток - как в нашем случае.

Для этого в RxJS есть специальные функции, которые принимают в качестве аргумента `Observable`.

*Map

Мы рассмотрим несколько операторов, позволяющих переключить поток внутри pipe:

- `mergeMap` : выполняет все параллельно
- `concatMap` : выполняет все последовательно
- `exhaustMap` : игнорирует все новые, пока предыдущий не завершится
- `switchMap` : выполняет новый, а предыдущий отменяет

ДЕМО

Для демонстрации закомментируем на сервере генрацию ошибки:

```
// if (Math.random() > 0.75) {
//     ctx.response.status = 500;
//     return;
// }
```

mergeMap

Создает `Observable` для каждого входящего значения и результы всех потоков объединяет в один без учета очереди:

```
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
  // debounceTime(100) - закомментировали для демо
  map(o => new URLSearchParams({q: o})),
  mergeMap(o => ajax.getJSON(`http://localhost:7070/api/search?${o}`)),
).subscribe(v => console.log(v));
```

Параллельно выполнит три запроса. В консоль выведутся результаты всех запросов, но порядок не гарантирован.

mergeMap

```
next ▼(5) [{…}, {…}, {…}, {…}, {…}]
       ▶0: {id: 1, name: "React"}
       ▶1: {id: 2, name: "Redux"}
       ▶2: {id: 3, name: "Redux Thunk"}
       ▶3: {id: 5, name: "Redux Observable"}
       ▶4: {id: 6, name: "Redux Saga"}
        length: 5
       ▶__proto__: Array(0)
next ▼(6) [{…}, {…}, {…}, {…}, {…}, {…}]
       ▶0: {id: 1, name: "React"}
       ▶1: {id: 2, name: "Redux"}
       ▶2: {id: 3, name: "Redux Thunk"}
       ▶3: {id: 4, name: "RxJS"}
       ▶4: {id: 5, name: "Redux Observable"}
       ▶5: {id: 6, name: "Redux Saga"}
        length: 6
       ▶__proto__: Array(0)
```

Получили классическую проблему при поиске "re": результат от "r" пришёл позже, чем от "re".

Не подходит.

concatMap

Создает `Observable` для каждого входящего значения создает и ставит его в очередь. Когда один `Observable` завершится подписывается на следующий. Таким образом гарантирует порядок значений:

```
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
  // debounceTime(100) - закомментировали для демо
  map(o => new URLSearchParams({q: o})),
  concatMap(o => ajax.getJSON(`http://localhost:7070/api/search?${o}`)),
).subscribe(v => console.log(v));
```

`concatMap`

1. Когда придет значение `r` начнет новый запрос.
2. Когда придет значение `re` дождется ответа от запроса `r`, только после чего начнет новый запрос.
3. Когда придет значение `rea` дождется пока придет ответ от запроса `re`, только после чего начнет новый запрос.

Таким образом в консоль будут выведены результаты всех трёх запросов по порядку.

Подходит ли нам?

exhaustMap

Создает `Observable` для входящего значения. Пока этот `Observable` не завершится игнорирует все новые значения:

```
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
  // debounceTime(100) - закомментировали для демо
  map(o => new URLSearchParams({q: o})),
  exhaustMap(o => ajax.getJSON(`http://localhost:7070/api/search?${o}`)),
).subscribe(v => console.log(v));
```

`exhaustMap`

1. Когда придет значение `r` начнет новый запрос.
2. Значение `re` может быть проигнорировано, если ещё не завершился запрос для `r`.
3. Значение `rea` может проигнорировано, если ещё не завершился запрос для `r`.

Таким образом в консоль может быть выведено как все три результата, так и два, так и только первый (если он был очень долгий).

switchMap

Создает `Observable` для входящего значения и отписывается от потока, созданного для предыдущего значения:

```
inputElChange$.pipe(
  map(o => o.target.value),
  filter(o => o.trim() !== ''),
  // debounceTime(100) - закомментировали для демо
  map(o => new URLSearchParams({q: o})),
  switchMap(o => ajax.getJSON(`http://localhost:7070/api/search?${o}`)),
).subscribe(v => console.log(v));
```

`switchMap`

1. Начнет запрос для `r`, но не дождется запроса из-за нового значения `re` и отменит запрос
2. Аналогично для `re` и `rea`
3. Для `rea` следующего значения нет, поэтому запрос выполнится и в консоль выведется только результат последнего запроса.

То, что нам нужно!

ИТОГОВЫЙ КОД

```
const { fromEvent } = rxjs;
const { ajax } = rxjs.ajax;
const { map, filter, debounceTime, switchMap } = rxjs.operators;

const inputEl = document.createElement('input');
document.body.appendChild(inputEl);
const inputElChange$ = fromEvent(inputEl, 'input')
inputElChange$.pipe(
    map(o => o.target.value),
    filter(o => o.trim() !== ''),
    debounceTime(100),
    map(o => new URLSearchParams({ q: o })),
    switchMap(o => ajax.getJSON(`http://localhost:7070/api/search?${o}`)),
).subscribe({
    next: value => console.log('next', value),
    error: error => console.error('error', error),
    complete: () => console.info('complete'),
});
```

REDUX OBSERVABLE

REDUX OBSERVABLE

Redux Observable - это middleware для Redux, позволяющее работать с `Action`'ами с помощью инструментов RxJS, а именно - предлагая модель потока для `Action`'ов.

Установим все необходимые зависимости:

```
npx create-react-app frontend
cd frontend
npm install prop-types redux react-redux redux-observable
npm start
```

```
// файл actions/actionTypes.js
export const SEARCH_SKILLS_REQUEST = 'SEARCH_SKILLS_REQUEST';
export const SEARCH_SKILLS_FAILURE = 'SEARCH_SKILLS_FAILURE';
export const SEARCH_SKILLS_SUCCESS = 'SEARCH_SKILLS_SUCCESS';
export const CHANGE_SEARCH_FIELD   = 'CHANGE_SEARCH_FIELD';

// файл actions/index.js
import { CHANGE_SEARCH_FIELD, SEARCH_SKILLS_REQUEST,
         SEARCH_SKILLS_FAILURE, SEARCH_SKILLS_SUCCESS, } from './actionTypes';

export const searchSkillsRequest = search => ({
  type: SEARCH_SKILLS_REQUEST, payload: {search}
});

export const searchSkillsFailure = error => ({
  type: SEARCH_SKILLS_FAILURE, payload: {error},
});

export const searchSkillsSuccess = items => ({
  type: SEARCH_SKILLS_SUCCESS, payload: {items},
});

export const changeSearchField = search => ({
  type: CHANGE_SEARCH_FIELD, payload: {search},
});
```

```
// файл reducers/skills.js
const initialState = { items: [], loading: false, error: null, search: '', };

export default function skillsReducer(state = initialState, action) {
  switch (action.type) {
    case SEARCH_SKILLS_REQUEST:
    return { ...state, items: [], loading: true, error: null, };
    case SEARCH_SKILLS_FAILURE:
    const {error} = action.payload;
    return { ...state, items: [], loading: false, error, };
    case SEARCH_SKILLS_SUCCESS:
    const {items} = action.payload;
    return { ...state, items, loading: false, error: null, };
    case CHANGE_SEARCH_FIELD:
    const {search} = action.payload;
    return { ...state, search };
    default:
    return state;
  }
}
```

КОМПОНЕНТ

```
import React, { Fragment } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { changeSearchField } from './actions/actionCreators';

export default function Skills() {
  const { items, loading, error, search } = useSelector(state => state.skills);
  const dispatch = useDispatch();

  const handleSearch = evt => {
    const { value } = evt.target;
    dispatch(changeSearchField(value));
  };

  const hasQuery = search.trim() !== '';
  return (<Fragment>
    <div><input type="search" value={search} onChange={handleSearch} /></div>
    {!hasQuery && <div>Type something to search</div>}
    {hasQuery && loading && <div>searching...</div>}
    {error ? <div>Error ...</div> : <ul>{items.map(
      o => <li key={o.id}>{o.name}</li>
    )}</ul>}
  </Fragment>)
}
```

.ENV

```
REACT_APP_SEARCH_URL=http://localhost:7070/api/search
```

EPICS

EPIC

Epic - ключевой примитив Redux Observable.

Представляет из себя функцию, которая на вход принимает поток `Action`'ов и на выходе возвращает поток `Action`'ов.

Общий вид выглядит следующим образом:

```
const epic = (action$, state$) => newAction$
```

Важно: для `Action`, которые поступают во входном потоке `dispatch` уже был вызван, а для `Action`'ов, которые будут в выходном потоке, `dispatch` **БУДЕТ** вызван.

EPIC

`action$` - поток из входных `Action`'ов (тип `Observable<Action>`)

`state$` - объект для доступа к `state` (тип `StateObservable<State>`)

`newAction$` - поток из `Action`'ов, для которых будет вызван `dispatch` (тип `Observable<Action>`)

EPIC

Ключевое: `Epic`'и работают после того, как `Action`хы уже получены `Reducer`'ом.

Т.е. если вы делаете `map` на новый `Action`, то старый не отменяется (т.к. он уже попал в `Reducer`).

Поэтому код: `const epic = action$ => action$` приведёт к бесконечному циклу.

EPIC

Наши Epic'и будут выглядеть следующим образом:

```
// файл epics/index.js
import { ofType } from 'redux-observable';
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map, filter, debounceTime, switchMap, catchError } from 'rxjs/operators';
import { CHANGE_SEARCH_FIELD, SEARCH_SKILLS_REQUEST } from '../actions/actionTypes'
import {
  searchSkillsRequest,
  searchSkillsSuccess,
  searchSkillsFailure,
} from '../actions/actionCreators';
```

EPIC

```
export const changeSearchEpic = action$ => action$.pipe(
  ofType(CHANGE_SEARCH_FIELD),
  map(o => o.payload.search.trim()),
  filter(o => o !== ''),
  debounceTime(100),
  map(o => searchSkillsRequest(o))
)
```

`ofType(CHANGE_SEARCH_FIELD)` это эквивалент `filter(o => o.type === CHANGE_SEARCH_FIELD)`.

На выходе мы получаем поток из `SEARCH_SKILLS_REQUEST`, которые `dispatch`'аться в `Store`.

EPIC

```
export const searchSkillsEpic = action$ => action$.pipe(
  ofType(SEARCH_SKILLS_REQUEST),
  map(o => o.payload.search),
  map(o => new URLSearchParams({ q: o })),
  switchMap(o => ajax.getJSON(`${process.env.REACT_APP_SEARCH_URL}?${o}`)),
  map(o => searchSkillsSuccess(o)),
);
```

На выходе мы получаем поток из `SEARCH_SKILLS_SUCCESS` вместе с ответом, которые `dispatch`'аться в `Store`.

EPIC

Как вы видите, ни `subscribe`, ни `unsubscribe` нам делать не нужно, за нас это сделает middleware.

НАСТРОЙКА `Store`

```
import { createStore, combineReducers, applyMiddleware, compose, } from 'redux';
import { combineEpics, createEpicMiddleware } from 'redux-observable';
import skillsReducer from '../reducers/skills';
import { changeSearchEpic, searchSkillsEpic } from '../epics';

const reducer = combineReducers({ skills: skillsReducer, });
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const epic = combineEpics(
  changeSearchEpic,
  searchSkillsEpic,
);
const epicMiddleware = createEpicMiddleware();
const store = createStore(reducer, composeEnhancers(
  applyMiddleware(epicMiddleware)
));

epicMiddleware.run(epic);
export default store;
```

ТЕКУЩИЙ ВАРИАНТ

На данный момент мы специально сделали так, что при очистке формы ввода остаются результаты предыдущего поиска.

Иногда это бывает разумно, иногда - нет.

Одним из ваших заданий будет переработать текущее решение так, чтобы при очистке поля поиска, результаты тоже очищались.

ОБРАБОТКА ОШИБОК

ОПТИМИЗМ

На данный момент наше решение достаточно оптимистично - мы всегда получаем ответ и отправляем `SEARCH_SKILLS_SUCCESS`.

Давайте раскомментируем генерацию ошибки на сервере и посмотрим, что получится во фронтенде: мы зависнем в состоянии "Searching...", но следующих попыток поиска не будет, как и не будет срабатывать `Action SEARCH_SKILLS_REQUEST`.

Почему?

ОШИБКИ В RXJS

Возникновение ошибки в потоке RxJS приводит к тому, что поток "останавливается" - не завершается (тогда срабатывает callback `complete`), а именно останавливается, т.е. больше значений в этом потоке быть не может.

Ошибка может произойти в потоке всего один раз.

Итого:

1. Возникновение ошибки ведёт к "остановке" нашего потока.
2. Мы по-прежнему нигде не генерируем `Action` `SEARCH_SKILLS_FAILURE`.

`catchError`

RxJS предоставляет нам оператор `catchError` и там мы можем выполнять те действия, которые нам нужны при возникновении ошибки (а мы хотим сгенерировать новый `Action`).

`catchError` предоставляет новый поток, вместо того, который был "остановлен" в результате возникновения ошибки:

```
export const searchSkillsEpic = action$ => action$.pipe(
  ofType(SEARCH_SKILLS_REQUEST),
  map(o => o.payload.search),
  map(o => new URLSearchParams({ q: o })),
  tap(o => console.log(o)),
  switchMap(o => ajax.getJSON(`${process.env.REACT_APP_SEARCH_URL}?${o}`).pipe(
    map(o => searchSkillsSuccess(o)),
    catchError(e => of(searchSkillsFailure(e))),
  )),
);
```

А ЕСЛИ ПОПЫТАТЬСЯ?

Возникают вопросы:

1. Если RxJS настолько мощный, есть ли в нём возможность повторно выполнить что-то при возникновении ошибки?
2. Когда это безопасно?

Какие у вас идеи по поводу второго вопроса?

ОТВЕТЫ:

- есть операторы `retry` и `retryWhen`, которые позволяют попробовать выполнить действия в потоке снова
- безопасно, в случае, если ваши запросы не изменяют состояние на сервере, либо изменение состояния безопасно

Т.е. поиск можно безопасно повторить несколько раз. Удаление объекта - тоже (больше одного раза он не удалится).

А вот отправка письма может привести к появлению дубликата (что не желательно), не говоря уже о переводе денежных средств (что критично).

Поэтому несколько раз подумайте, прежде чем выполнять повторную попытку, возможно, сервер уже выполнил всю работу, а вы просто не получили ответ.

retry

```
export const searchSkillsEpic = action$ => action$.pipe(
  ofType(SEARCH_SKILLS_REQUEST),
  map(o => o.payload.search),
  map(o => new URLSearchParams({ q: o })),
  switchMap(o => ajax.getJSON(`${process.env.REACT_APP_SEARCH_URL}?${o}`).pipe(
    retry(3),
    map(o => searchSkillsSuccess(o)),
    catchError(e => of(searchSkillsFailure(e))),
  )),
);
```

А КАК ЖЕ REDUX THUNK?

А КАК ЖЕ REDUX THUNK?

Redux Thunk отлично подходит для простых действий с side-effects.

В принципе, всё, что мы сделали, можно сделать и с помощью Redux Thunk, но кода придётся написать (и отладить) в разы больше.

Больше кода - больше ошибок.

REDUX OBSERVABLE

Redux Observable предоставляет вам более мощную альтернативу с уже готовыми RxJS-операторами, которые значительно упрощают обработку комплексных сценариев (как с поиском).

Но при этом Redux Observable требует знания RxJS, на получение которого нужно потратить дополнительное время.

ИТОГИ

Сегодня мы рассмотрели достаточно сложные темы: RxJS и Redux Observable.

Мы рассмотрели только малую часть тех возможностей, которые предоставляет Redux Observable.

RxJS (и Redux Observable вместе с ним) достаточно мощный инструмент, с которым мы вам рекомендуем познакомиться детальнее.

RxJS используется не только в связке с Redux, но и сам по себе, позволяя решать комплексные проблемы даже тогда, когда вы используете Vanilla JS без библиотек вроде React и фреймворков.

Итоговые исходники к материалам сегодняшней лекции будут размещены в репозитории с кодом к лекциям.

нетология

Спасибо за внимание!

Время задавать вопросы ☒

АЛЕКСАНДР СИВЦОВ

Александр Сивцов