Redux Observable

Егор Наумов
frontend-разработчик веб-приложений

Егор Наумов

- frontend-разработчик веб-приложений
- эксперт в области разработки промышленного ПО
- преподаватель по программированию
- соавтор курса PCR-JS (полный цикл разработки JS) в УрФУ

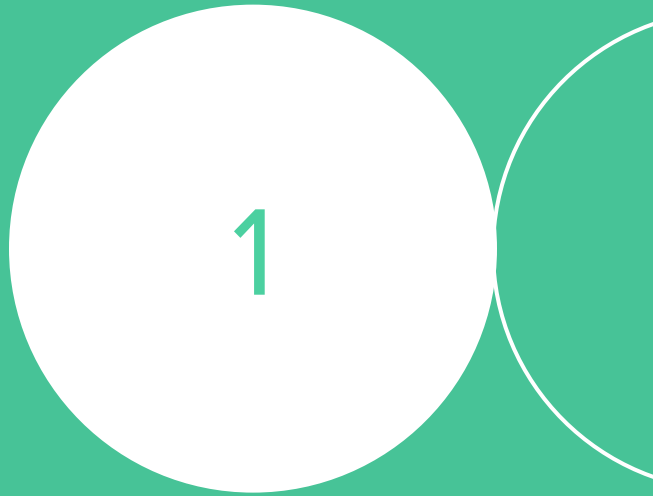
План

1. RxJS
2. Redux Observable

Цели занятия

1. Познакомиться с библиотекой RxJS
2. Рассмотреть подход использования Redux Observable

RxJS



Задача

Реализовать поиск на сайте. Именно поиск, а не фильтрацию. Это значит, что мы должны отправлять HTTP-запрос на сервер, дожидаться ответа и отображать результаты

Задача

Для реализации задачи можно сделать кнопку «Поиск» и отправлять запрос на сервер только тогда, когда пользователь нажмет на неё. Но современные пользователи привыкли к другому виду взаимодействия: что поиск осуществляется параллельно с вводом текста

Возможные проблемы

Проблем будет несколько:

- из-за сетевых задержек мы можем получать неактуальные результаты
- флуд сервера, так как если мы будем реагировать на каждое изменение, то на слово react нам придётся послать 5 запросов

Способы решения

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

Этот процесс кажется сложным, но существует инструмент,
который может упростить решение задачи

Server API

Подготовим сервер для тестирования взаимодействия.
Мы специально сделаем его медленным со случайными задержками

npm:

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

Server API

Не стоит думать что проблема медленного сервера надумана и решается покупкой быстрого сервера или рекомендацией клиенту использовать более быстрый интернет

Server API

scripts в package.json

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

Server API

server.js

```
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"},
];
```

Server API

```
const router = new Router();
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);
```

Практика

Краткий обзор сервера

RxJS

RxJS — это готовая библиотека для создания приложений, работающий с асинхронными вызовами и событиями

npm:

```
npm i rxjs
```

CDN:

```
<script
     src='https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.1.0/rxjs.umd.js'
> </script>
```

Особенности RxJS

RxJS комбинирует достаточно много идей, но ключевыми являются следующие:

- все события представляются в виде распределённого во времени набора значений
- предоставляются расширенные операторы для преобразования потоков: Array.map , Array.filter, Array.reduce и т. д.

Observable

Observable — это поток из значений, распределённых во времени. Значений может не быть совсем, может быть одно, а может быть и множество

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 и т. д. При этом поток распределен во времени, так как 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'),
);
```

Observable

Observables ленивы — ничего не произойдёт, пока никто не подпишется на Observable

Observable

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'),
});
```

Observable

Отписываться от потока стоит тогда, когда нам больше не нужны данные в потоке. Для того, чтобы отписаться, нужно использовать объект подписки, полученный при выполнении **subscribe**

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

После **unsubscribe** все ресурсы освобождаются, в противном случае вы получите утечку ресурсов

Пример

Рассмотрим на примере пока без React и Redux

```
<script
   src='https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.1.0/rxjs.umd.js'>
</script>
<script>
   const inputEl = document.createElement('input');
   document.body.appendChild(inputEl);
</script>
```

Пример

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** — создаёт поток из своих аргументов
- и другие

Пример

```
<script
   src='https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.1.0/rxjs.umd.js'>
</script>
<script>
   const { fromEvent } = rxjs;
   const { ajax } = rxjs.ajax;

   const inputEl = document.createElement('input');
   document.body.appendChild(inputEl);
   //...
</script>
```

Пример

```
<script>
    //..
    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'),
    });
</script>
```

$ — это общепринятое соглашение для обозначения переменных, указывающих на потоки

Пример

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*

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

Пример

Нам нужно, чтобы почти на каждое изменение поля ввода мы могли посылать запрос на поиск. **RxJS** предоставляет нам возможность выстраивать **pipe** для **Observable**, фактически выстраивая конвейер 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

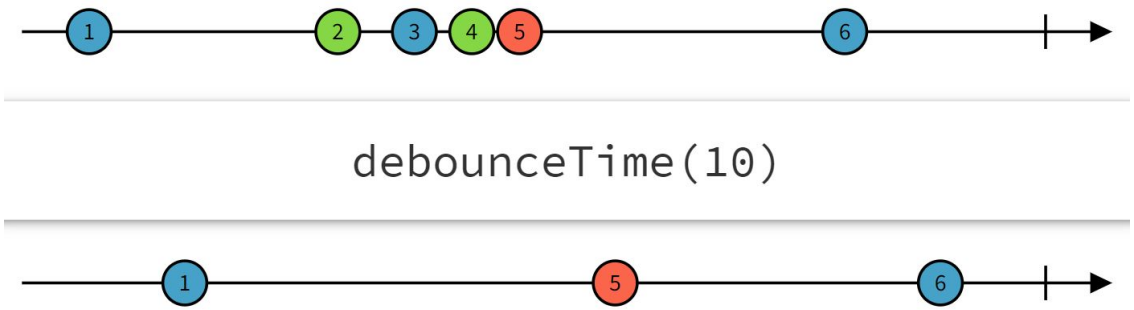
map & filter

Самые простые операторы: 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



debounceTime

```
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 одного раза в 100 мс.
Попробуйте нажать клавишу в поле поиска и не отпускать

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

Операторы, позволяющие переключить поток внутри pipe:

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

Демо

Для демонстрации раскомментируем на сервере генерацию ошибки

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

mergeMap

Создаёт **Observable** для каждого входящего значения и результаты всех потоков объединяет в один без учёта 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 выводятся 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

concatMap

Создает **Observable** для каждого входящего значения создаёт и ставит его в очередь. Когда один **Observable** завершится, подписывается на следующий. Таким образом гарантирует порядок значений

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

concatMap

1. Когда придёт значение, **r** начнёт новый запрос
2. Когда придёт значение, **re** дождётся ответа от запроса **r**, только после чего начнёт новый запрос
3. Когда придет значение, **rea** дождётся, пока придёт ответ от запроса **re**, только после чего начнёт новый запрос

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

exhaustMap

Создаёт **Observable** для входящего значения. Пока этот **Observable** не завершится, игнорирует все новые значения

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

exhaustMap

1. Когда придёт значение, **r** начнёт новый запрос
2. Значение **re** может быть проигнорировано, если ещё не завершился запрос для **r**
3. Значение **rea** может быть 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

Создаёт **Observable** для входящего значения и отписывается от потока, созданного для предыдущего значения

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

switchMap

1. Начнёт запрос для **r**, но не дождётся 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

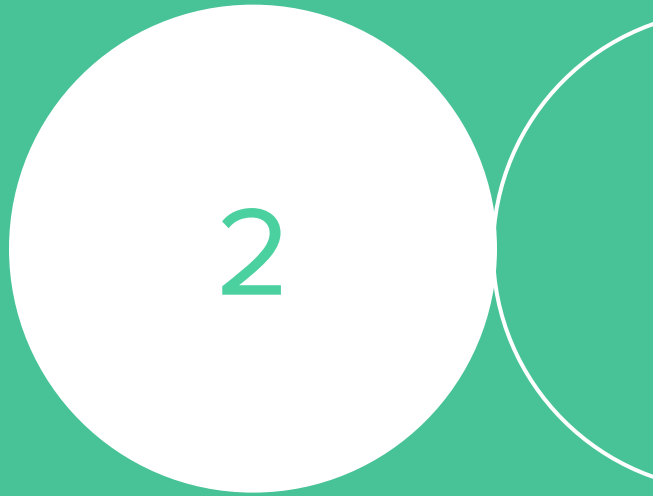
```
const { fromEvent } = rxjs;
const { ajax } = rxjs.ajax;
const { map, filter, debounceTime, switchMap } = rxjs.operators;

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'),
});
```

Практика

Обзор примера на клиенте

Redux Observable



Redux Observable

Redux Observable — это middleware для Redux, позволяющее работать с **Action** с помощью инструментов **RxJS**, а именно предлагая модель потока для **Action**.

Установка

```
1 npm i redux-observable
```

Redux Observable

Пример в обычном HTML

```
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

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, loading: true, error: null};
       case SEARCH_SKILLS_FAILURE:
           const {error} = action.payload;
           return {...state, 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;
   }
}
```

Component Skills

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

.evn

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

Epic

Epic — ключевой примитив Redux Observable.
Представляет из себя функцию, которая на вход принимает поток **Action** и на выходе возвращает поток **Action**

Общий вид выглядит следующим образом

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

Для **Action**, которые поступают во входном потоке, **dispatch** уже был вызван, а для **Action**, которые будут в выходном потоке, **dispatch** будет вызван

Epic

Epic работают после того, как **Action** уже получены **Reducer**

То есть если вы делаете **map** на новый **Action**, то старый не отменяется, так как он уже попал в **Reducer**

Код **const epic = action$ => action$** приведёт к бесконечному циклу

Epic

Наши **Epic** могут выглядеть следующим образом
Файл: epics/index.js

```
import {ofType} from 'redux-observable';
import {of} from 'rxjs';
import {ajax} from 'rxjs/ajax';
import {map, tap, retry, 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

Файл: epics/index.js

```
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** в **Store**

Epic

Файл: epics/index.js

```
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}`)),
   map(o => searchSkillsSuccess(o)),
);
```

На выходе мы получаем поток из **SEARCH_SKILLS_SUCCESS** вместе с ответом, которые **dispatch** в **Store**

Epic

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

Настройка Store

Файл: store/index.js

```
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;
```

Пример

В примере мы специально сделали так, что при очистке формы ввода остаются результаты предыдущего поиска. Иногда это бывает разумно, иногда — нет

Ошибки в RxJS

Сейчас наше решение достаточно оптимистично — мы всегда получаем ответ и отправляем **SEARCH_SKILLS_SUCCESS**

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, то есть больше значений в этом потоке быть не может.

Ошибка может произойти в потоке всего один раз

Ошибки в RxJS

Итого:

1. Возникновение ошибки ведёт к остановке потока
2. Мы нигде не генерируем Action **SEARCH_SKILLS_FAILURE**

Ошибки в RxJS

RxJS предоставляет нам оператор **catchError,** и там мы можем выполнять те действия, которые нам нужны при возникновении ошибки. А мы хотим сгенерировать новый Action

Ошибки в RxJS

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))),
   )),
);
```

Ошибки в RxJS

Возникают вопросы:

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

retry и retryWhen

- 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

Ошибки в RxJS

Поиск можно безопасно повторить несколько раз. Удаление объекта — тоже, но больше одного раза он не удалится.

Отправка письма может привести к появлению дубликата, не говоря уже о переводе денежных средств, что критично. Стоит подумать, прежде чем выполнять повторную попытку, возможно, сервер уже выполнил всю работу, а вы просто не получили ответ

Ошибки в RxJS

Оператор **retry**

```
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(
       retry(3),
       map(o => searchSkillsSuccess(o)),
       catchError(e => of(searchSkillsFailure(e))),
   )),
);
```

Практика

Краткий обзор примера

Redux Thunk

Redux Thunk — middleware, которая позволяет нам диспатчить функции, которые могут производить side-effect, вызывать другие функции и диспатчить экшены. Создаются через thunkCreator

```
export const getData = text => dispatch => {
     dispatch(setLoading(true));
     fetch('...')
          .then(r => r.json())
          .then(j => {
               dispatch(setLoading(false));
               dispatch(setData(j));
          })
};
```

Observable vs 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 строк кода придётся написать и отладить в разы больше.

Больше кода — больше ошибок

Observable vs Thunk

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

Epic

1. **action$** — поток из входных **Action,** тип **Observable<Action>**
2. **state$** — объект для доступа к **state,** тип **StateObservable<State>**
3. **newAction$** — поток из **Action** для которых будет вызван **dispatch,** тип **Observable<Action>**

Итоги

- **RxJS** — это готовая библиотека для создания приложений, работающая с асинхронными вызовами и событиями
- **Observable** — это поток из значений, распределённых во времени. Значений может не быть совсем, может быть одно, а может быть и множество

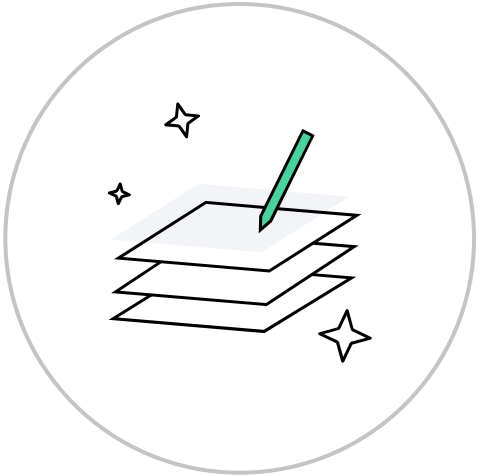
Итоги

Redux Observable — это middleware для Redux, позволяющее работать с Action с помощью инструментов RxJS, а именно предлагая модель потока для Action

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Ваши вопросы