Redux Saga

Дамир Рысаев

Frontend developer в компании "Здравсити"

Дамир Рысаев

Frontend developer в компании "Здравсити"

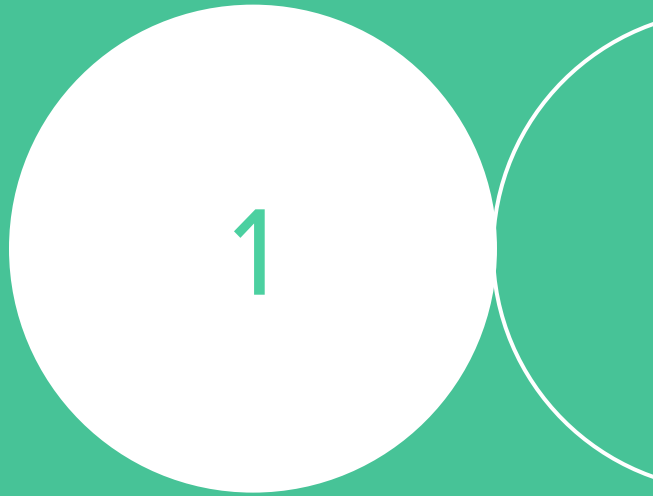
План

1. Generator
2. Redux Saga

Цели занятия

1. Познакомиться с middleware Redux Saga
2. Рассмотреть подход использования Generators

Generator



Задача

Сегодня настала очередь ещё одного популярного инструмента — Redux Saga. Список можно продолжать ещё долго, например, Redux Cycles, Redux Ship и т. д. Но наша задача — рассмотреть самые популярные

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

Возможные проблемы

- 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

Server API

Подготовим сервер для тестирования взаимодействия

npm:

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

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

Iterator

Iterator — объект является итератором, если он умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности.
В JS итератор — объект, который предоставляет метод **next()**, объект с двумя полями: done и value.

Generator

Generator — это функция, которая может приостанавливать своё выполнение.
Приостановка сводится к тому, что мы можем выйти из функции, при этом сохранив контекст-значения переменных и информацию о последней выполненной строке, и войти в неё снова ровно с той точки, с которой мы вышли.
Функция генератор возвращает итератор.

Generator

Вызов функции генератора возвращает объект типа **Generator**, вызовы метода **next** на котором позволяют получить информацию о том, завершён ли генератор (**complete**), и получить значение, которое указано в инструкции **yield**. Повторный вызов функции генератора вернёт новый объект

Generator

```
function* stringGenerator() {
   yield 'first';
   yield 'second';
   yield 'third';
}

const string = stringGenerator();
console.log(string);
console.log(string.next()); // {done: false, value: 'first'}
console.log(string.next()); // {done: false, value: 'second'}
console.log(string.next()); // {done: false, value: 'third'}
console.log(string.next()); // {done: true,  value: undefined}
```

while (true)

Поскольку **yield** приводит к приостановке генератора, то следующий код является вполне легитимным и не приводит к зависанию страницы, в отличие от кода без генератора и **yield**:

```
function* randomNumberGenerator(start, stop) {
   while (true) {
       yield Math.floor(Math.random() * (stop - start + 1)) + start;
   }
}
const random = randomNumberGenerator(1, 10);
console.log(random.next());
console.log(random.next());
```

while (true)

Этот код приведёт к зависанию страницы

```
function infinity() {
   while (true) {
       console.log('infinity');
   }
}

infinity();
```

Generator delegation

Генераторы позволяют делегировать свою работу другим генераторам. При этом пока в том, кому делегировали, не закончатся **yield,** мы не перейдем к **yield** основного:

```
function* delegatedGenerator() {
   yield 'first delegated';
   yield 'second delegated';
}

function* delegatorGenerator() {
   yield 'first delegator';
   yield* delegatedGenerator();
   yield 'second delegator';
}

const delegator = delegatorGenerator();
console.log(delegator.next().value); // first delegator
console.log(delegator.next().value); // first delegated
console.log(delegator.next().value); // second delegated
console.log(delegator.next().value); // second delegator
```

Generator

С **yield** можно использовать оператор присваивания. Тогда мы получим значение, используемое при следующем вызове **next**. Это нужно запомнить, поскольку с первого взгляда это чаще всего не очевидно

```
function* valueGenerator() {
   const value = yield 'value';
   console.warn(`value: ${value}`);
}

const value = valueGenerator();
console.log(value.next());
value.next(42); // в value будет 42, сработает console.warn
```

throw

Генераторы позволяют прокинуть не только значение внутрь, но и ошибку, с помощью метода **throw**

```
function* errorGenerator() {
    try {
        yield 'value';
    } catch (e) {
        console.warn(`Catched: ${e.message}`);
    }
}

const error = errorGenerator();
error.next();
error.throw(new Error('something bad happened'));
```

Generator и promise (async/await)

Благодаря комбинации рассмотренных подходов, можно сделать следующее

```
async function searchRequest() {
    const response = await fetch('http://localhost:7070/api/search?q=Re');
    if (!response.ok) {
        throw new Error(response.statusText);
    }
    return await response.json();
}

function* searchGenerator() {
    while (true) {
        try {
            const data = yield (searchRequest());
            console.info(data);
        } catch (e) {
            console.warn(e.message);
        }
    }
}
```

Generator и promise (async/await)

Вызов этой функции генератора будет выглядеть не очень красиво

```
1 const search = searchGenerator();
2 // первый поиск
3 search.next().value.then(o => search.next(o), o => search.throw(o));
4 // второй поиск
5 search.next().value.then(o => search.next(o), o => search.throw(o));
```

Generator и promise (async/await)

Вызов этой функции генератора будет выглядеть не очень красиво

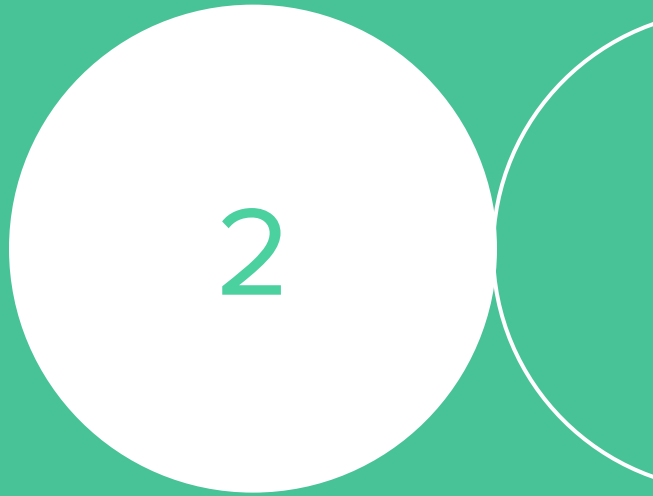
```
1  const search = searchGenerator();
2  // первый поиск
3  search.next().value.then(o => search.next(o), o => search.throw(o));
4  // второй поиск
5  search.next().value.then(o => search.next(o), o => search.throw(o));
```

Примечание: и на всякий случай напоминаем, что **return** из **async** автоматически заворачивается в **Promise,** как и сгенерированное исключение

Практика

Обзор примера на клиенте

Redux Saga



Redux Saga

Redux Saga — middleware для Redux, позволяющее управлять побочными эффектами, например, сетевыми запросами

Redux Saga

Ключевая идея заключается в ментальной модели. Redux Saga предлагает мыслить в терминах дополнительного потока (треда) исполнения, отвечающего за побочные эффекты. Этот поток может быть запущен, поставлен на паузу и отменён из основного потока нашего приложения с помощью обычных Redux Action

Redux Saga

Установим все необходимые зависимости

```
npx create-react-app frontend
cd frontend
npx i redux react-redux redux-saga
npm start
```

Redux Saga

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';
```

Redux Saga

файл actions/actionTypes.js

```
1  import { CHANGE_SEARCH_FIELD, SEARCH_SKILLS_REQUEST,
2           SEARCH_SKILLS_FAILURE, SEARCH_SKILLS_SUCCESS, } from './actionTypes';
3
4  export const searchSkillsRequest = search => ({
5     type: SEARCH_SKILLS_REQUEST, payload: {search}
6  });
7
8  export const searchSkillsFailure = error => ({
9     type: SEARCH_SKILLS_FAILURE, payload: {error}
10 });
11
12 export const searchSkillsSuccess = items => ({
13    type: SEARCH_SKILLS_SUCCESS, payload: {items}
14 });
15
16 export const changeSearchField = search => ({
17    type: CHANGE_SEARCH_FIELD, payload: {search}
18 });
```

Redux Saga

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 React 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 (
       <>
           <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>}
       </>
   )
}
```

Файл .env

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

Saga

Saga — это функция-генератор, которая инкапсулирует логику обработки сложных, то есть многошаговых последовательностей действий. Фактически Saga запускаются при старте приложения и прослушивают все **Action** в ожидании нужного

Saga

файл saga/index.js

```
import {takeLatest, put, spawn} from 'redux-saga/effects';
import {searchSkillsRequest} from '../actions/actionCreators';
import {CHANGE_SEARCH_FIELD} from '../actions/actionTypes';

function* changeSearchSaga() {
   while (true) {
       const action = yield take(CHANGE_SEARCH_FIELD);
       yield put(searchSkillsRequest(action.payload.search));
   }
}

export default function* saga() {
   yield spawn(changeSearchSaga);
}
```

Store

файл store/index.js

```
import {createStore, combineReducers, applyMiddleware, compose,} from 'redux';
import createSagaMiddleware from 'redux-saga';
import skillsReducer from '../reducers/skills';
import saga from '../sagas';

const reducer = combineReducers({ skills: skillsReducer, });
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const sagaMiddleware = createSagaMiddleware();
const store = createStore(reducer, composeEnhancers(
   applyMiddleware(sagaMiddleware)
));

sagaMiddleware.run(saga);
export default store;
```

Root saga

rootSaga, то есть корневая Saga, задача которой запустить другие Saga и завершиться

```
export default function* saga() {
   yield spawn(changeSearchSaga);
}
```

Saga

файл saga/index.js

```
function* changeSearchSaga() {
   while (true) {
       const action = yield take(CHANGE_SEARCH_FIELD);
       yield put(searchSkillsRequest(action.payload.search));
   }
}
```

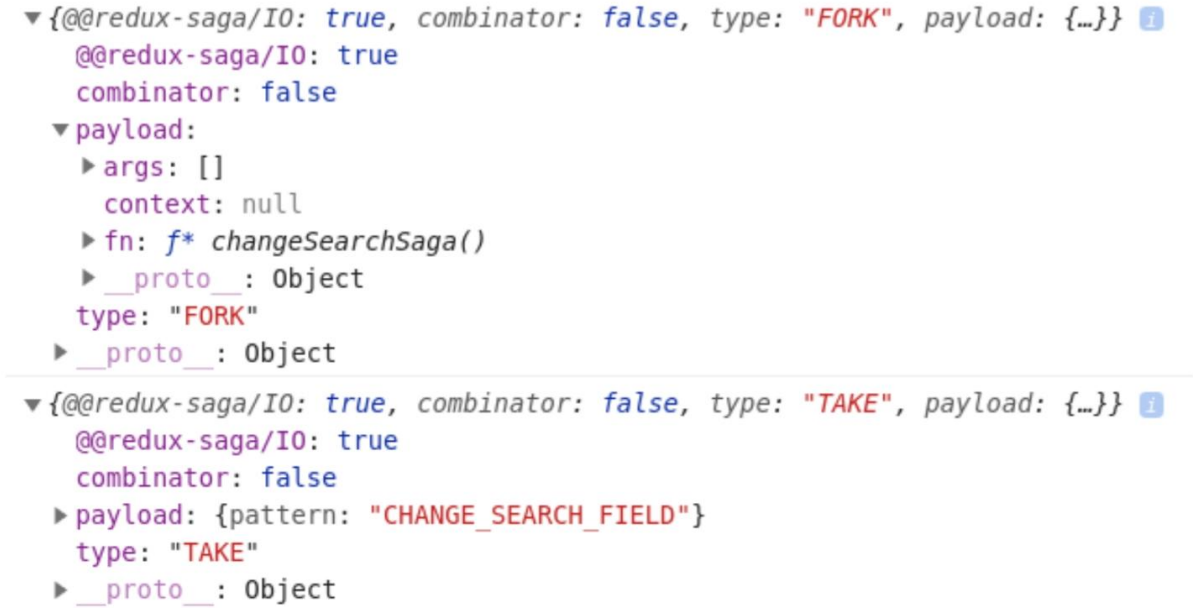
Бесконечный цикл подписки, в котором мы ожидаем появления Action с типом CHANGE_SEARCH_FIELD (take).
В ответ на появление этого Action мы генерируем новый Action с помощью соответствующего Action Creator (put)

Blocking vs non-blocking

- **take** — блокирующий вызов, то есть пока не придёт **Action** с типом **CHANGE_SEARCH_FIELD**, Saga заблокируется в этой точке и дальнейшее выполнение приостановится
- **spawn** и **put** — неблокирующие вызовы, то есть Saga не ждёт, а идёт по коду дальше, исполняет следующие строки

Что такое spawn, take и т. д.

Это всё специальные функции для создания эффектов, которые являются Plain JavaScript Object



Effects

Эффекты представляют из себя специальные объекты, которые обрабатываются **Redux Saga Middleware** для выполнения определённых действий.

Ключевые для нас сейчас:

- **take** — генерирует эффект для ожидания **Action** определённого типа
- **put** — генерирует эффект для **dispatch** определённого **Action**

Effects

Попробуем с помощью эффектов добиться следующего результата:

1. Не реагировать на пустое поле ввода, если данные отсутствуют
2. Организовать задержку в 100 мс после того, как пользователь закончил вводить текст
3. Сделать HTTP-запрос и игнорировать результаты предыдущего, если мы посылаем новый
4. Обрабатывать ошибки

API

Пока мы не перешли к обработке ошибок, закомментируем на сервере код, отвечающий за их генерацию

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

Фильтрация

Мы свободно можем использовать в Saga условия, циклы и другие конструкции

```
function* changeSearchSaga() {
   while (true) {
       const action = yield take(CHANGE_SEARCH_FIELD);
       if (action.payload.trim() === '') {
           continue;
       }
       yield put(searchSkillsRequest(action.payload.search));
   }
}
```

take

Но **take**, помимо типа **Action**, может также принимать **callback**, по которому определять, выбирать этот **Action** или нет

```
function* changeSearchSaga() {
   while (true) {
       const action = yield take(o =>
           o.type === CHANGE_SEARCH_FIELD && action.payload.trim() !== ''
       );
       yield put(searchSkillsRequest(action.payload.search));
   }
}
```

Здесь нет никакого предпочтения, решать вам, какой способ наиболее читабелен для вас

Action filter

Вынесем шаблон фильтрации **Action** в отдельную функцию

```
function filterChangeSearchAction({type, payload}) {
   return type === CHANGE_SEARCH_FIELD && payload.search.trim() !== ''
}

function* changeSearchSaga() {
   while (true) {
       const action = yield take(filterChangeSearchAction);
       yield put(searchSkillsRequest(action.payload.search));
   }
}
```

Категории эффектов в Redux Saga

- redux specific
- generic
- concurrency

debounce

Среди эффектов есть **debounce**, который позволяет запустить Saga только после того, как перестанут поступать **Action** в течение определённого количества мс.
Это высокоуровневый эффект, который уже содержит в себе и циклы, и take, и всё остальное

debounce

```
function* debounceChangeSearchSaga(action) {
   yield put(searchSkillsRequest(action.payload));
}

function* changeSearchSaga() {
   yield debounce(100, filterChangeSearchAction, debounceChangeSearchSaga);
}
```

Организация кода

В Redux Saga принято организовывать код с паттерном Watcher/Worker. Наши Saga уже следуют этому паттерну, но необходимо разобраться с сутью

Watchers and workers

Принято разделять поток управления на две Saga:

- **Watcher** — ждёт нужный **Action** и запускает **fork** Worker
- **Worker** — обрабатывает **Action** и завершает свою работу

Watchers

```
// worker
function *handleChangeSearchSaga(action) {
   yield put(searchSkillsRequest(action.payload.search));
}

// watcher
function* watchChangeSearchSaga() {
   yield debounce(100, filterChangeSearchAction, handleChangeSearchSaga);
}
```

По поводу имён единого соглашения нет, но Watcher принято начинать с префикса **watch**

Вызов API

Для работы с API напишем отдельный модуль

```
export const searchSkills = async (search) => {
    const params = new URLSearchParams({q: search});
    const response = await fetch(`${process.env.REACT_APP_SEARCH_URL}?${params}`);
    if (!response.ok) {
        throw new Error(response.statusText);
    }
    return await response.json();
}
```

Watcher

Напишем отдельный Watcher

```
// watcher
function* watchSearchSkillsSaga() {
   while (true) {
       const action = yield take(SEARCH_SKILLS_REQUEST);
       yield takeLatest(handleSearchSkillsSaga(), action);
   }
}

export default function* saga() {
   yield spawn(watchChangeSearchSaga);
   yield spawn(watchSearchSkillsSaga)
}
```

Watcher

Напишем отдельный Watcher

```
// worker
function* handleSearchSkillsSaga(action) {
   const data = yield retry(searchSkills, action.payload.search);
   yield put(searchSkillsSuccess(data));
}
```

call создаёт эффект, который приводит к вызову функции. Функция может быть как генератором, так и обычной функцией, возвращающей **Promise** или другое значение

Медленные ответы

Но получилось то же самое: ответ на re пришёл раньше, чем на r при вводе последовательности re в строку поиска

```
(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)
r
(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)
```

takeEvery, takeLatest

Redux Saga предлагает в качестве таковых нам **takeEvery** и **takeLatest**.

- **takeEvery** — фактически, в один вызов делает то, что мы делали до этого: **take** + **fork**
- **takeLatest** — **takeEvery** + отмена предыдущей задачи

Нам нужно использовать takeLatest

```
1 // watcher
2 function* watchSearchSkillsSaga() {
3    yield takeLatest(SEARCH_SKILLS_REQUEST, handleSearchSkillsSaga);
4 }
```

Обработка ошибок

Осталось только разобраться с обработкой ошибок, поскольку для Worker делается **fork** — любое необработанное исключение приведёт к обрушению **Watcher**, после чего он уже не сможет следить за **Action**

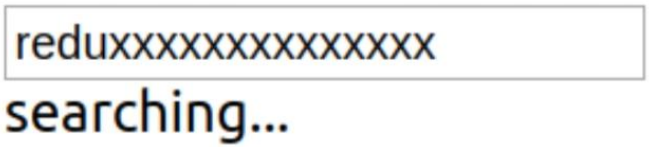
Обработка ошибок

Мы это можем увидеть, раскомментировав соответствующее условие на сервере

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

Обработка ошибок

После получения ошибки, дальнейших попыток запроса не происходит



Name	Status	Type
search?q=r	200	fetch
search?q=re	200	fetch
search?q=red	200	fetch
search?q=redu	200	fetch
search?q=redux	200	fetch
search?q=reduxx	200	fetch
search?q=reduxxx	200	fetch
search?q=reduxxxx	500	fetch
info?t=1565877498169		

Обработка ошибок

Для перехвата ошибки достаточно стандартного **try-catch**

```
1  // worker
2  function* handleSearchSkillsSaga(action) {
3     try {
4         const data = yield retry(searchSkills, action.payload.search);
5         yield put(searchSkillsSuccess(data));
6     } catch (e) {
7         yield put(searchSkillsFailure(e.message));
8     }
9  }
```

Retry

Для реализации нескольких попыток повтора нам достаточно использовать **retry** вместо **call**. Мы это можем увидеть, раскомментировав соответствующее условие на сервере

```
// worker
function* handleSearchSkillsSaga(action) {
   try {
       const retryCount = 3;
       const retryDelay = 1 * 1000; // ms
       const data = yield retry(retryCount, retryDelay, searchSkills,
action.payload.search);
       yield put(searchSkillsSuccess(data));
   } catch (e) {
       yield put(searchSkillsFailure(e.message));
   }
}
```

Практика

Краткий обзор примера

Итоги

! **Generator** — это функция, которая может приостанавливать свое выполнение. Приостановка сводится к тому, что мы можем выйти из функции и войти в неё снова ровно с той точки, с которой мы вышли

Итоги

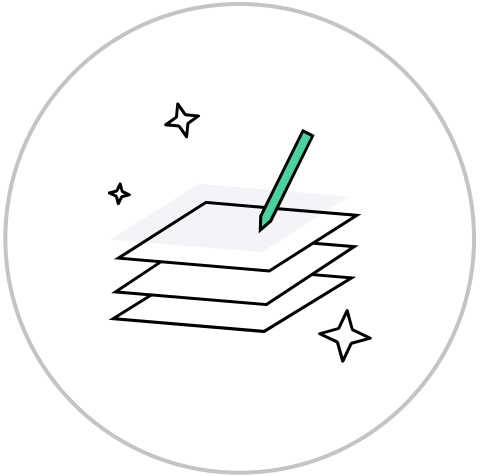
- **Saga** — это функция-генератор, которая инкапсулирует логику обработки сложных (многошаговых) последовательностей действий

- **Redux Saga** — middleware для Redux, позволяющее управлять побочными эффектами, например, сетевыми запросами

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