# Генераторы и какое отношение они имеют к redux-saga

Вы уже знаете, что такое мидлвары и знакомы [с библиотекой redux-thunk](https://practicum.yandex.ru/trainer/web-plus/lesson/f1b5c9bc-d445-4102-b8b4-ae226669be0d/task/57c9d7bc-f391-4f2a-a07a-bcdaeb56c5fc/), которая позволяет работать со сторонними эффектами. В этой теме вы познакомитесь с её конкурентом — redux-saga.

И redux-thunk, и redux-saga широко используются в проектах. Thunk стал популярен благодаря своей простоте, в то время как саги (redux-saga) пугают многих из-за сложности концепта (с которым мы сегодня и познакомимся). Но если разобраться в redux saga, вы увидите, что приложения с этой библиотекой проще масштабируются, а всю асинхронную логику удобнее тестировать, когда она сразу собрана в одном месте.

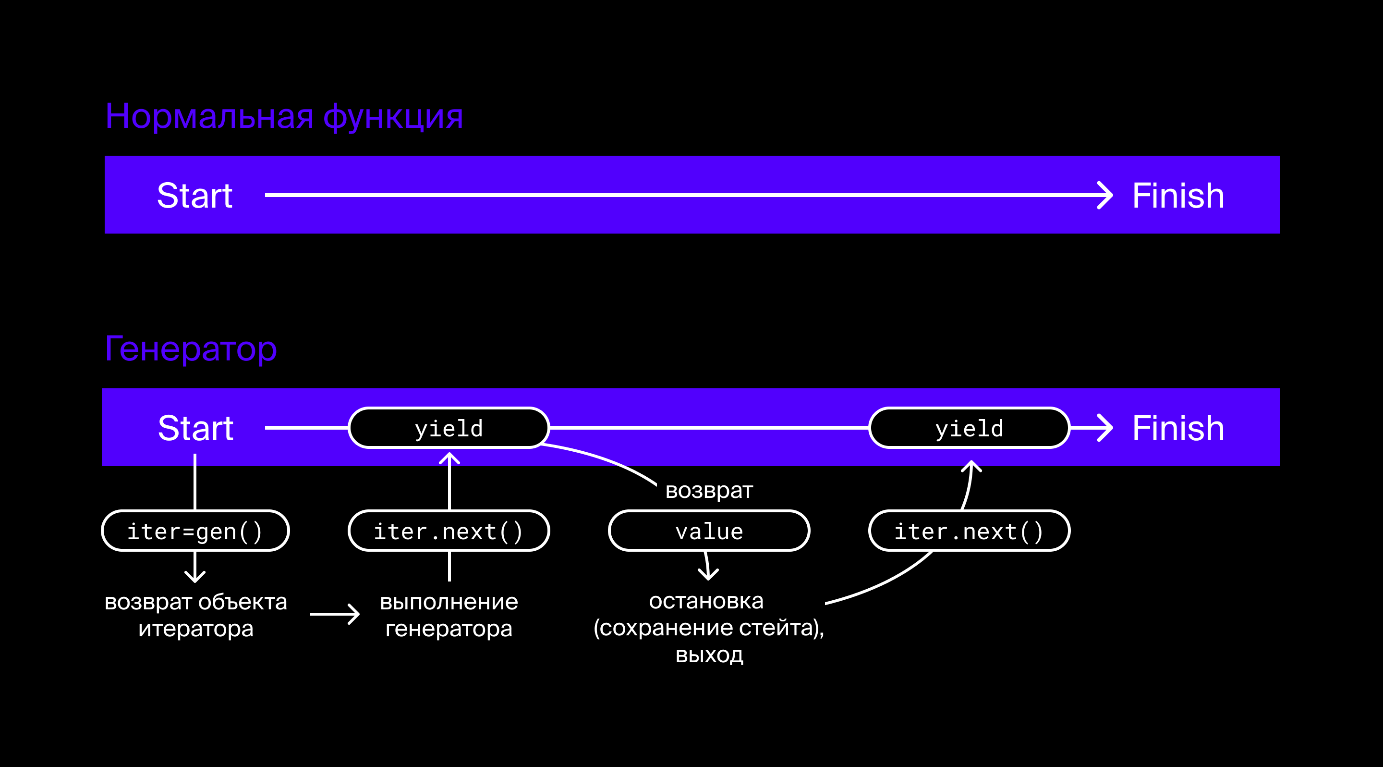
Но прежде чем углубиться в изучение новой библиотеки, вспомним, что такое генераторы. Спойлер — на них в сагах строится примерно всё!

## Генераторы

Генератор — функция, из которой можно выйти с сохранением контекста, а позже вернуться и продолжить с того же момента.

Если бы генераторы можно было применять не только в программировании, это как минимум сделало бы вечеринки веселее: вы могли бы уйти в разгар веселья, выспаться перед работой, а на следующий день вернуться с новыми силами и ничего не пропустить!

Из обычной функции мы можем выйти только в конце её выполнения, а при наличии return она вернёт нам какое-то значение. Генератор же возвращает объект-итератор, с помощью которого мы управляем продолжением выполнения генератора:



Ключевое слово yield указывает на остановки выполнения генератора и возвращает из функции значение. Благодаря ему мы и можем выйти из функции, затем вернуться и продолжить с того же места.



Напишем простой пример генератора:

Скопировать кодJSX

function\* generator(i) {

console.log("I’m first");

yield i + 2;

console.log(`i = ${i}`);

yield i + 10;

}

Обратите внимание, что мы объявляем генератор с помощью function\*, а не function.

Чтобы выполнить содержимое генератора, нам потребуется итератор:

Скопировать кодJSX

const generatorObj = generator(10);

Пока мы не вызовем функцию итератора next(), его содержимое не будет выполнено.

Скопировать кодJSX

generatorObj.next()

Теперь вы можете увидеть строку I’m first в консоли, а generatorObj.next() вернёт объект:

Скопировать кодJSX

{ value: 12, done: false }

Этот объект состоит из:

* value — значения, которое возвращает функция;
* done — булево значения, которое указывает, закончился ли генератор. Если done=true, то из генератора больше нечего вернуть. При дальнейших вызовах value примет значение undefined.

В нашем генераторе есть еще один yield, давайте получим значение value, которое он возвращает:

Скопировать кодJSX

generatorObj.next().value

Теперь генератор полностью выполнен, и значение done стало true.

## Генератор + промис = 🔥💪

Генераторы — мощный механизм асинхронного программирования в JavaScript, а если ещё и использовать их в связке с промисами... В общем, если бы Саурон был программистом, он мог бы не создавать Кольцо Всевластия, чтобы управлять другими кольцами, а просто использовать генератор:

Скопировать кодJSX

const getRings = function \* (url) {

try {

let requestRings = yield fetch(url); *// один раз отправил запрос и получил данные*

let allRings = yield requestRings.json(); *// теперь у Саурона есть все кольца*

yield allRings.people; *// для получения колец людей ему не придётся делать ещё один запрос*

yield allRings.dwarf; *// как и для колец гномов: он может получить их, когда потребуется*

} catch (error) {

console.log(`ERROR: no rings`);

}

});

Так Саурон мог бы один раз получить все кольца и избежать повторных запросов. Теперь он может в любой момент управлять именно теми кольцами, которые его интересуют.

## Бесконечный генератор

Всё в этом мире заканчивается, даже у генератора есть done=true. Вот только Саурон хотел абсолютной и пожизненной власти, и в этом генератор ему бы помог (хорошо, что Саурон всё-таки не знал о программировании и генераторах).

Генератор может быть бесконечным. Да-да, можно создать сколько угодно новых колец:

Скопировать кодJSX

function \* createNewRing() {

let num = 1;

while (true) {

yield getRing(num);

num = num + 1

}

}

const ringMaker = createNewRing();

console.log(ringMaker.next().value) *// one ring*

console.log(ringMaker.next().value) *// two ring*

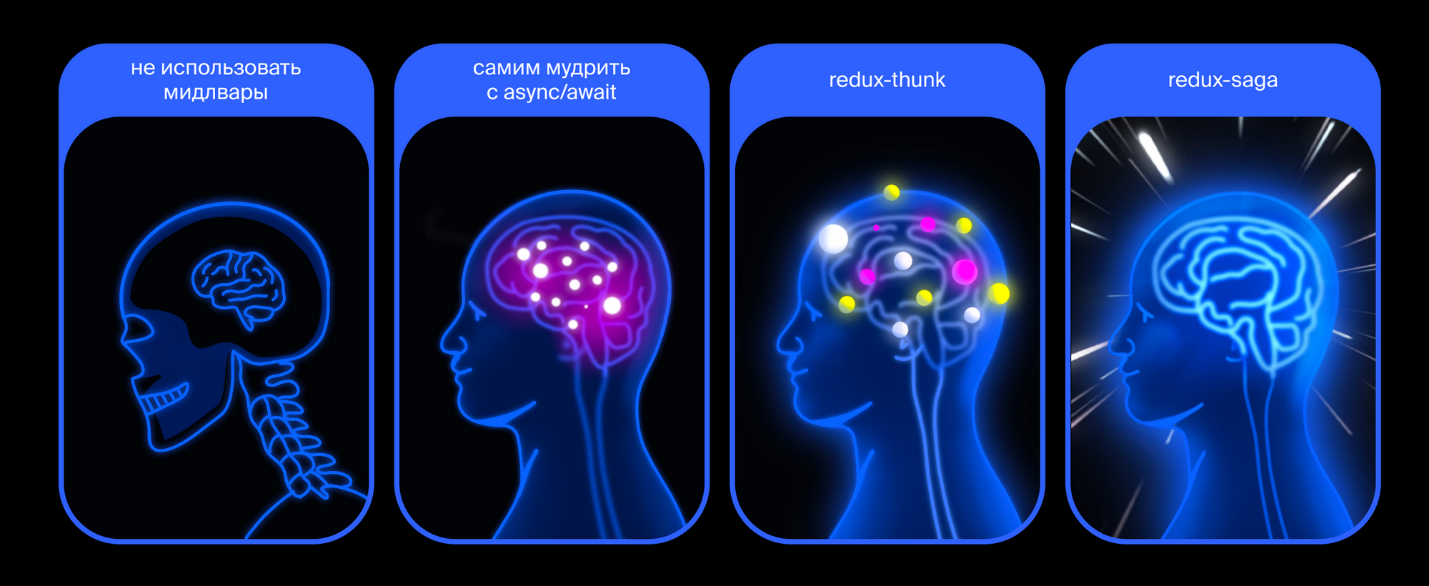
...

Генератор createNewRing содержит бесконечный цикл, внутри которого мы получаем новое кольцо с помощью yield.

А вот наш урок — не бесконечный, и с этим багажом знаний о генераторах можно смело знакомиться с сагами!

# redux-saga во всей красе

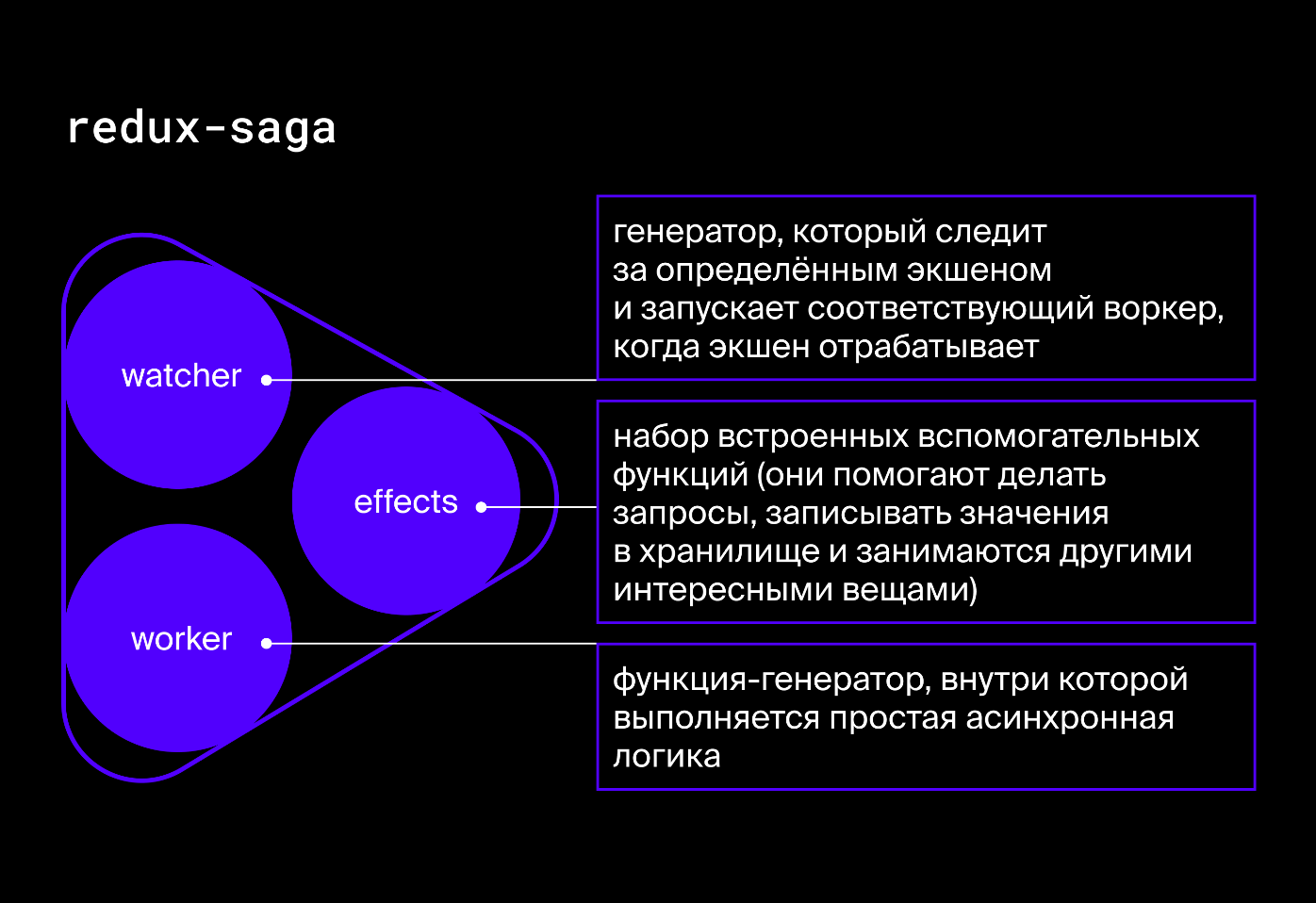
В предыдущем уроке мы вспомнили, что такое генераторы. Теперь настало время познакомиться с библиотекой redux-saga. Для этого перепишем уже хорошо знакомые вам примеры с redux-thunk на redux-saga, а заодно сравним подходы.



## Основные абстракции redux-saga

redux-saga — библиотека, которая позволяет работать с побочными эффектами.

Чтобы у вас появилось общее представление о redux-saga, нужно знать 3 основные составляющие, которые мы подробно рассмотрим позже:



Теперь, когда мы имеем общее представление о библиотеке, можно создать вашу первую сагу.

## Worker

Помните код, который мы использовали в приложении с thunk? В функции getItems мы получаем некоторые значения и, в случае успешного запроса на сервер, записываем их в хранилище:

Скопировать кодJSX

export function getItems() {

return function(dispatch) {

dispatch({

type: GET\_ITEMS\_REQUEST

});

getItemsRequest().then(res => {

if (res && res.success) {

dispatch({

type: GET\_ITEMS\_SUCCESS,

items: res.data

});

} else {

dispatch({

type: GET\_ITEMS\_FAILED

});

}

});

};

}

Теперь перепишем этот пример в концепции саг. Создадим воркер, который будет содержать ту же асинхронную логику с запросом на сервер и записью значений в хранилище:

Скопировать кодJSX

function\* getItemsWorker(dispatch) {

try {

const { data } = yield getItemsRequest();

yield dispatch({ type: GET\_ITEMS\_SUCCESS, items: data });

} catch {

yield dispatch({ type: GET\_ITEMS\_FAILED });

}

}

Пока мы опустили событие GET\_ITEMS\_REQUEST, вернёмся к нему чуть позже.

В этом примере мы вызываем API-запрос напрямую из генератора, а ещё кладём значения в хранилище. Можно оставить и так — это рабочий код. А можно сделать логику воркера более декларативной и применить для этого вспомогательные функции, которые предоставляет redux-saga — эффектов.

## Effects

Благодаря эффектам описанную логику воркера мы можем сделать более декларативной. Мы ещё не говорили про написание unit-тестов, но поверьте на слово, — декларативность будет большим плюсом при тестировании. И весомым преимуществом redux-saga перед redux-thunk.

Чтобы использовать эффекты, их следует импортировать:

Скопировать кодJSX

import { call } from 'redux-saga/effects';

Функция call возвращает объект, который описывает переданную в него функцию, а redux-saga уже самостоятельно отвечает за её вызов.

С помощью call вместо прямого вызова запроса из генератора мы получим объект, который описывает этот запрос. Его можно представить так:

Скопировать кодJSON

{

CALL: {

fn: () => {…}

args: []

}

}

Затем импортируем ещё один эффект — put. Он отвечает за запись значений в хранилище. Перепишем наш воркер:

Скопировать кодJSX

function\* getItemsWorker() {

try {

const { data } = yield call(getItemsRequest);

yield put({ type: GET\_ITEMS\_SUCCESS, items: data });

} catch {

yield put({ type: GET\_ITEMS\_FAILED });

}

}

Мы получили более декларативное описание функции. Теперь воркер даёт инструкции redux-saga, показывает объявления функций и говорит: «тут сходи на сервер», «а тут запиши полученные значения в хранилище».

## Watcher

Для полноты картины не хватает только саги-наблюдателя (watcher), которая запустит воркер.

Импортируем ещё один эффект — takeEvery. Он порождает сагу на каждый выполненный экшен, то есть будет запускать воркер getItemsWorker каждый раз, когда в приложении будет вызван соответствующий экшен GET\_ITEMS\_REQUEST:

Скопировать кодJSX

function\* getItemsWatcher() {

yield takeEvery(GET\_ITEMS\_REQUEST, getItemsWorker);

}

Теперь в приложении вместо вызова getItems() можно написать dispatch({ type: GET\_ITEMS\_REQUEST }) ииии… ничего не произойдёт! Ведь мы ещё не подключили redux-saga к хранилищу. Сделаем это!

## Подключение redux-saga

Импортируем в файл с инициализацией хранилища сагу-наблюдатель и функцию createSagaMiddleware, которая отвечает за использование стороннего программного кода (middleware):

Скопировать кодJSX

import createSagaMiddleware from 'redux-saga'

import { getItemsWatcher } from './sagas';

Подключаем усилитель с сагами к хранилищу:

Скопировать кодJSX

const sagaMiddleware = createSagaMiddleware();

const enhancer = composeEnhancers(applyMiddleware(sagaMiddleware)); *// enhancer, если нам нужны усилители*

export const store = createStore(rootReducer, enhancer);

И вишенка на торте! Чтобы созданная сага-наблюдатель могла следить за тем, что происходит в приложении, необходимо вызвать метод run:

Скопировать кодJSX

sagaMiddleware.run(getItemsWatcher);

Теперь всё заработает как часы.

Но как масштабировать проект, ведь пока у нас обрабатывается всего один watcher? Тут следует воспользоваться эффектом all, который позволяет работать с несколькими сагами-наблюдателями:

Скопировать кодJSX

import { all } from 'redux-saga/effects';

import { getItemsWatcher, bigBrother } from './all.sagas';

export const rootSaga = function\* root() {

yield all([

getItemsWatcher,

bigBrother,

]);

};

Заменим в сторе getItemsWatcher на rootSaga и сможем создавать сколько угодно саг-наблюдателей, пусть хоть обнаблюдаются.

Скопировать кодJSX

sagaMiddleware.run(rootSaga);

Теперь работают оба вотчера. getItemsWatcher следит за getItemsWorker, а bigBrother — за  своим воркером.

В этой теме вы познакомились с ещё одной библиотекой для работы с побочными эффектами — redux-saga. Научились создавать саги и узнали про [эффекты](https://redux-saga.js.org/docs/api/#effect-creators) в них. Сравнили код, написанный на thunk, с реализацией на сагах.

Библиотека redux-thunk проще для начала работы — она понятнее. Но если разобраться с генераторами в redux-saga, то писать на них код становится не так страшно. Саги позволяют описать более сложную логику, тогда как thunk ограничен в действиях. Да и тестировать асинхронную логику, которая описана в одном месте, в сагах гораздо проще.

Что выбирать для проектов, решать вам. Стоит отталкиваться от поставленной задачи. Если у вас нагруженные побочные эффекты, то лучше подойдут саги — с ними проще масштабировать большой проект. Для небольшого хранилища может быть достаточно redux-thunk. В общем, на вкус и цвет все библиотеки разные, подключайте и используйте ту, которая лучше вам подходит.