redux-saga

Бібліотек, яка призначена для організації асинхронних запитів (без використання побічних ефектів та нечистих функцій).

<https://redux-saga.js.org/docs/introduction/BeginnerTutorial.html>

<https://flaviocopes.com/redux-saga/>

<https://decembersoft.com/posts/changing-react-route-programmatically-with-redux-saga/>

Саги описують за допомогою генераторів

Генератори

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Iterators_and_Generators>

<https://frontender.info/es6-in-depth-generators/>

<https://metanit.com/web/javascript/14.2.php>

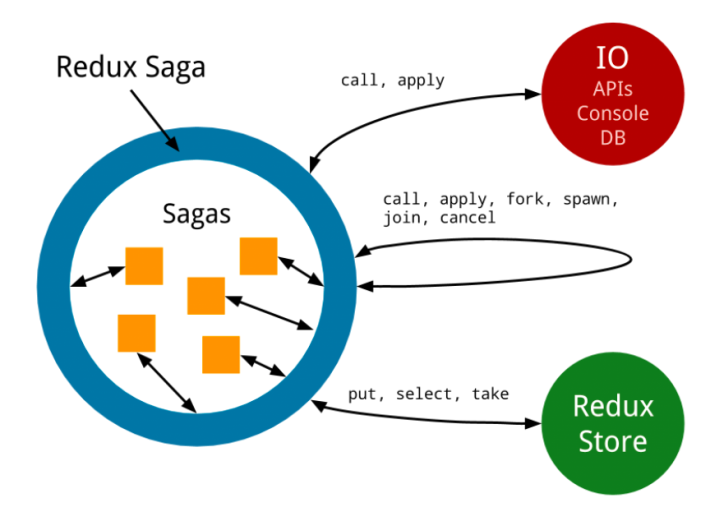
|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | |
| Опис генераторів  function\* назва\_генератора () {  yield оператор-результат1;  yield оператор-результат 2;  . . . . . . . . . .  return оператор-результат …;  } | function\* getNumber(){      yield 5;      yield 25;      yield 125;  } | | |
| Створення генератора (для цього треба викликати функцію, виклик як звичайної функції)  let генератор = виклик\_функції\_нератора () | let **numberGenerator** = getNumber(); | | |
| **Використання. Поступове одержання значень**. При цьому кожного разу функція повертає значення і її виконання призупиняться до наступного виклику (який буде продовжено з місця зупинки)  генератор . next()  Функція-геератор одержує значення-об’єкт  {  value : повернуте значення ,  done : true- якщо генератор закінчив роботу, false – ще не кінець  } | *Фукція-генератор*  function\* getNumber(){ | *Виклик генератора* | *Результат виклику* |
|  | yield 5; | console.log(  numberGenerator.next()); | {value: 5, done: false} |
|  | yield 25; | console.log(  numberGenerator.next()); | {value: 25, done: false} |
|  | yield 125;  } | console.log(  numberGenerator.next()); | {value: 125, done: false} |
|  |  | console.log(  numberGenerator.next()); | {value: undefined, done:true} |
| Використання. У циклі  for(let змінна of генератор ){    використання значення змінної  } | function\* getNumber(){      yield 5;      yield 25;      yield 125;  }  let numberGenerator = getNumber();    **for(let num of numberGenerator){**      console.log(num);  }  5  25  125 | | |
| **Передача данных в генератор**  function\* назва\_генератора () {  let змінна1 = yield оператор-результат1;  let змінна2 =yield оператор-результат 2;  . . . . . . . . . .  return оператор-результат …;  }  генератор . next() // Перший викли  //Наступні виклики  генератор . next( знач. що заміняє **оператор1** і збер. у **зм.1** )  генератор . next(знач. що заміняє **оператор2** і збер. у **зм.2** )  . . . . . . . . . .. | function\* getNumber(){      let n = yield 5;      console.log("n:", n);      let m = yield 25 \* n;      console.log("m:", m);      yield 125 \* m;  }  let numberGenerator = getNumber();    console.log(numberGenerator.next().value); //5  console.log(numberGenerator.next(2).value); //50  console.log(numberGenerator.next(3).value); | | |
| **Генерування виключинх ситуацій** [generator.throw](https://learn.javascript.ru/generator" \l "generator-throw) | function\* gen() {  try {    let result = yield "Сколько будет 2 + 2?"; // в этой строке возникнет ошибка  alert("выше будет исключение ^^^");  } catch(e) {  alert(e); // выведет ошибку  }  }  let generator = gen();  let question = generator.next().value;  generator.throw(new Error("ответ не найден в моей базе данных")); | | |
| [Плоский асинхронный код](https://learn.javascript.ru/generator" \l "ploskiy-asinhronnyy-kod) | // генератор для получения и показа аватара  // он yield'ит промисы  function\* showUserAvatar() {  let userFetch = yield fetch('/article/generator/user.json');  let userInfo = yield userFetch.json();  let githubFetch = yield fetch(`https://api.github.com/users/${userInfo.name}`);  let githubUserInfo = yield githubFetch.json();  let img = new Image();  img.src = githubUserInfo.avatar\_url;  img.className = "promise-avatar-example";  document.body.appendChild(img);  yield new Promise(resolve => setTimeout(resolve, 3000));  img.remove();  return img.src;  }  // вспомогательная функция-чернорабочий  // для выполнения промисов из generator  function execute(generator, yieldValue) {  let next = generator.next(yieldValue);  if (!next.done) {  next.value.then(  result => execute(generator, result),  err => generator.throw(err)  );  } else {  // обработаем результат return из генератора  // обычно здесь вызов callback или что-то в этом духе  alert(next.value);  }  }  execute( showUserAvatar() ); | | |
|  |  | | |
|  |  | | |

Загальна схема використання саг

<https://redux-saga.js.org/docs/introduction/BeginnerTutorial.html>

<https://flaviocopes.com/redux-saga/>

|  |  |  |
| --- | --- | --- |
| опис саг |  | export function\* helloSaga() {  console.log('Hello Sagas!')  } |
| підключення до redux |  | import { createStore, applyMiddleware } from 'redux'  import createSagaMiddleware from 'redux-saga'  // ...  import { helloSaga } from './sagas'  //------------- створення sagaMiddleware -------------  const sagaMiddleware = createSagaMiddleware()  const store = createStore(  reducer,  applyMiddleware(sagaMiddleware) //**🡨Підключення до store**  ) |
| Запуск саг |  | sagaMiddleware.run(helloSaga) //🡨Запуск  const action = type => store.dispatch({type}) |



* [Basic Helpers](https://flaviocopes.com/redux-saga/#basic-helpers)
  + [takeEvery()](https://flaviocopes.com/redux-saga/#takeevery)
  + [takeLatest()](https://flaviocopes.com/redux-saga/#takelatest)
  + [take()](https://flaviocopes.com/redux-saga/#take)
  + [put()](https://flaviocopes.com/redux-saga/#put)
  + [call()](https://flaviocopes.com/redux-saga/#call)
* [Running effects in parallel](https://flaviocopes.com/redux-saga/#running-effects-in-parallel)
  + [all()](https://flaviocopes.com/redux-saga/#all)
  + [race()](https://flaviocopes.com/redux-saga/#race)

Асинхронні виклики

|  |  |  |  |
| --- | --- | --- | --- |
| Для actions | | Для функцій | |
|  | Блокування |  | Блокування |
| take(pattern) | Yes | call(fn, ...args) | Yes |
| takeEvery(pattern, saga, ...args) | No | fork(fn, ...args) | No |
| takeLatest(pattern, saga, ...args) | No |  |  |
| **put(action)** | No |  |  |
|  |  |  |  |

Методи

|  |  |  |  |
| --- | --- | --- | --- |
|  | blocking |  |  |
| take(pattern) | Yes | Creates an Effect description that instructs the middleware to wait for a specified action on the Store. The Generator is suspended until an action that matches pattern is dispatched.  The result of yield take(pattern) is an action object being dispatched. |  |
| takeEvery(pattern, saga, ...args) | No | Spawns a saga on each action dispatched to the Store that matches pattern.   * pattern: String | Array | Function - for more information see docs for [take(pattern)](https://redux-saga.js.org/docs/api/#takepattern) * saga: Function - a Generator function * args: Array<any> - arguments to be passed to the started task. takeEvery will add the incoming action to the argument list (i.e. the action will be the last argument provided to saga) |  |
| takeLatest(pattern, saga, ...args) | No | Spawns a saga on each action dispatched to the Store that matches pattern. And automatically cancels any previous saga task started previously if it's still running.  Each time an action is dispatched to the store. And if this action matches pattern, takeLatest starts a new saga task in the background. If a saga task was started previously (on the last action dispatched before the actual action), and if this task is still running, the task will be cancelled.   * pattern: String | Array | Function - for more information see docs for [take(pattern)](https://redux-saga.js.org/docs/api/#takepattern) * saga: Function - a Generator function * args: Array<any> - arguments to be passed to the started task. takeLatest will add the incoming action to the argument list (i.e. the action will be the last argument provided to saga) |  |
| **put(action)** | No | Creates an Effect description that instructs the middleware to dispatch an action to the Store. This effect is non-blocking and any errors that are thrown downstream (e.g. in a reducer) will not bubble back into the saga. |  |
| call(fn, ...args) | Yes | Creates an Effect description that instructs the middleware to call the function fn with args as arguments.   * fn: Function - A Generator function, or normal function which either returns a Promise as result, or any other value. * args: Array<any> - An array of values to be passed as arguments to fn |  |
| fork(fn, ...args) | No | Creates an Effect description that instructs the middleware to perform a **non-blocking** call on fn |  |
| join(task) | Yes | Creates an Effect description that instructs the middleware to wait for the result of a previously forked task.   * task: Task - A [Task](https://redux-saga.js.org/docs/api/" \l "task) object returned by a previous fork |  |
| cancel(task) | No | Creates an Effect description that instructs the middleware to cancel a previously forked task.   * task: Task - A [Task](https://redux-saga.js.org/docs/api/" \l "task) object returned by a previous fork | function\* mySaga() {  const task = yield fork(myApi)  // ... later  // will call promise[CANCEL] on the result of myApi  yield **cancel(task)**  }  Можна перевірити, чи була сага відмінена  function\* deleteRecord({ payload }) {  try {  const { confirm, deny } = yield call(prompt);  if (confirm) {  yield put(actions.deleteRecord.confirmed())  }  if (deny) {  **yield cancel()**  }  } catch(e) {  // handle failure  } finally {  **if (yield cancelled()) {**  // shared cancellation logic  yield put(actions.deleteRecord.cancel(payload))  }  }  } |
| select(selector, ...args) | No | Creates an effect that instructs the middleware to invoke the provided selector on the current Store's state (i.e. returns the result of selector(getState(), ...args)).   * selector: Function - a function (state, ...args) => args. It takes the current state and optionally some arguments and returns a slice of the current Store's state * args: Array<any> - optional arguments to be passed to the selector in addition of getState.   If select is called without argument (i.e. yield select()) then the effect is resolved with the entire state (the same result of a getState() call). |  |
| delay(ms, [val]) | Yes | Returns an effect descriptor to block execution for ms milliseconds and return val value. |  |
| all(effects) | Blocks if there is a blocking effect in the array or object | To execute them in parallel, wrap them into all(): | import { all, call } from 'redux-saga/effects'  // correct, effects will get executed in parallel  const [users, repos] = yield all([  call(fetch, '/users'),  call(fetch, '/repos')  ]) |
| race(effects) | Yes | RACE()  race() differs from all() by not waiting for all of the helpers calls to return. It just waits for one to return, and it’s done. | import { take, call, race } from `redux-saga/effects`  import fetchUsers from './path/to/fetchUsers'  function\* fetchUsersSaga() {  const { response, cancel } = yield race({  response: call(fetchUsers),  cancel: take(CANCEL\_FETCH)  })  }  ======  It’s typically used to cancel a background task that runs forever until something occurs:  import { race, call, take } from 'redux-saga/effects'  function\* someBackgroundTask() {  while(1) {  //...  }  }  yield race([  bgTask: call(someBackgroundTask),  cancel: take('CANCEL\_TASK')  ]) |

**Task**

The Task interface specifies the result of running a Saga using fork, middleware.run or runSaga.

|  |  |
| --- | --- |
| **method** | **return value** |
| task.isRunning() | true if the task hasn't yet returned or thrown an error |
| task.isCancelled() | true if the task has been cancelled |
| task.result() | task return value. `undefined` if task is still running |
| task.error() | task thrown error. `undefined` if task is still running |
| task.toPromise() | a Promise which is either:   * resolved with task's return value * rejected with task's thrown error |
| task.cancel() | Cancels the task (If it is still running) |

### Blocking / Non-blocking

| **Name** | **Blocking** |
| --- | --- |
| takeEvery | No |
| takeLatest | No |
| takeLeading | No |
| throttle | No |
| debounce | No |
| retry | Yes |
| take | Yes |
| take(channel) | Sometimes (see API reference) |
| takeMaybe | Yes |
| put | No |
| putResolve | Yes |
| put(channel, action) | No |
| call | Yes |
| apply | Yes |
| cps | Yes |
| fork | No |
| spawn | No |
| join | Yes |
| cancel | No |
| select | No |
| actionChannel | No |
| flush | Yes |
| cancelled | Yes |
| race | Yes |
| delay | Yes |
| all | Blocks if there is a blocking effect in the array or object |

AXIOS

<https://github.com/axios/axios>

<https://flaviocopes.com/axios/>

В первую очередь axios — это замена fetch. Axios выполняет все то же самое, но умеет работать как на клиенте, так и на сервере, позволяя легко изменять общие настройки всех запросов (например, добавить хедер авторизации).   
  
  
Для создания определенных типов запросов служат команды: axios.get, axios.delete, axios.head, axios.options, axios.post, axios.put, axios.patch. Каждая команда создает соответственного типа http-запрос.

npm install axios

Приклад 1.

// GET request for remote image

axios({

method:'get',

url:'http://bit.ly/2mTM3nY',

responseType:'stream'

})

.then(function (response) {

response.data.pipe(fs.createWriteStream('ada\_lovelace.jpg'))

});

Приклад 2.

const axios = require('axios');

// Make a request for a user with a given ID

axios.get('/user?ID=12345')

.then(function (response) {

// handle success

console.log(response);

})

.catch(function (error) {

// handle error

console.log(error);

})

.then(function () {

// always executed

});

// Optionally the request above could also be done as

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

})

.then(function () {

// always executed

});

// Want to use async/await? Add the `async` keyword to your outer function/method.

async function getUser() {

try {

const response = await axios.get('/user?ID=12345');

console.log(response);

} catch (error) {

console.error(error);

}

}

Приклад.

Performing multiple concurrent requests

function getUserAccount() {

return axios.get('/user/12345');

}

function getUserPermissions() {

return axios.get('/user/12345/permissions');

}

axios.all([getUserAccount(), getUserPermissions()])

.then(axios.spread(function (acct, perms) {

// Both requests are now complete

}));

Прилад .

const axios = require('axios')

const getBreeds = () => {

try {

return

} catch (error) {

console.error(error)

}

}

const countBreeds = async () => {

axios.get('https://dog.ceo/api/breeds/list/all')

.then(response => {

if (response.data.message) {

console.log(

`Got ${Object.entries(response.data.message).length} breeds`

)

}

})

.catch(error => {

console.log(error)

})

}

countBreeds()