НЕТОЛОГИЯ

REDUX SAGA

ВЛАДИМИР ЯЗЫКОВ/ USEFUL WEB

ВЛАДИМИР ЯЗЫКОВ

frontend-разработчик в Useful Web.ru

neizerth@gmail.com

vk.com/neizerth

ПЛАН ЗАНЯТИЯ

1. Side Effects
2. Generators
3. Redux Saga
4. Sagas
5. Effects
6. Watchers and Workers
7. Вызов API
8. Обработка ошибок
9. Retry

SIDE EFFECTS

SIDE EFFECTS

На предыдущих лекциях мы рассмотрели уже три подхода для работы с Side Effects:

1. Side Effects в Action Creator'ax
2. Redux Thunk
3. Redux Observable

Сегодня настала очередь ещё одного популярного инструмента - Redux Saga.

На самом деле, список можно продолжать ещё долго, например, Redux Cycles, Redux Ship и т.д.

Но наша задача - рассмотреть самые популярные.

ЗАДАЧА

Решим всё ту же самую задачу, что мы решали и для Redux Observable: задача поиска на сайте.

Отправляем HTTP-запрос на сервер, дожидаемся ответа и отображаем результаты.

Q: Почему мы решаем ту же задачу?

A: Чтобы сравнить два инструмента и их подходы.

ЗАДАЧА

Проблемы с прошлой лекции не изменились:

- 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!

СЕРВЕР

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

.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);
```

GENERATORS

GENERATORS

Прежде чем мы непосредственно перейдём к рассмотрению Redux Saga, нам необходимо познакомиться с синтаксисом генераторов в JS.

`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.

Посмотрим на примере.

GENERATORS

```
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}
```

Т.е. на самом деле 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.

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());
console.log(random.next());
```

А вот это приведёт (проверяйте на свой страх и риск):

```
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 перейдём 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
```

GENERATORS

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

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

С `yield` можно использовать оператор присваивания. Тогда мы получим значение, используемое при следующем вызове `next`.

Это нужно запомнить, поскольку с первого взгляда, чаще всего, не является очевидным.

`throw`

Генераторы позволяют "прокинуть" не только значение внутрь, но и ошибку (на месте `yield` возникнет исключение), 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'));
```

Причём если мы не перехватим исключение внутри генератора, оно "прорвётся наружу" в точке `error.throw`.

GENERATOR И PROMISE (ASYNC/AWAIT)

Благодаря комбинации рассмотренных выше подходов, мы можем 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:

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

Но что если за нас его будет делать кто-то другой?

Примечание: и на всякий случай напоминаем, что `return` из `async` автоматически заворачивается в `Promise` (как и сгенерированное исключение).

REDUX SAGA

REDUX SAGA

Redux Saga - middleware для Redux, позволяющее управлять побочными эффектами (например, сетевыми запросами).

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'ов.

Для описания подобного потока используется синтаксис генераторов, который мы с вами как раз и повторили.

REDUX SAGA

Установим все необходимые зависимости:

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

Приступим к созданию нашей первой `Saga`!

SAGAS

SAGA

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

Фактически, Saga запускаются при старте приложения и прослушивают все `Action`'ы в ожидании нужного.

```
// файл sagas/index.js
import { take, 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

Вернёмся к нашим Saga.

```
// файл sagas/index.js
export default function* saga() {
  yield spawn(changeSearchSaga);
}
```

Это `rootSaga` - т.е. корневая Saga, задача которой запустить другие Saga и завершиться.

ROOT SAGA

Вообще говоря, она не обязательна, но в её использовании есть ряд ключевых моментов:

1. Saga могут быть организованы в условное "дерево", при этом отмена родительской Saga приводит к отмене запущенных из неё Saga
2. Родительская Saga в общем случае ожидает завершения дочерних
3. Неперехваченные ошибки в порождённых Saga попадают в родительскую Saga, и, если не перехватываются и там, то приводят к уничтожению родителя (а, соответственно, и всех порождённых Saga)

`spawn` - позволяет "отделить" порождённую Saga от родителя, избежав всего этого.

ROOT SAGA

Стоит отметить, что мы продемонстрировали `spawn` в Root Saga для примера. И стоит вам отделять порождённые Saga или нет - зависит от задачи.

Для критичных задач (без работы которых невозможна нормальная работа приложения) стараются либо при ошибках в порождённых Saga обрушать родительскую, либо делать их перезапускаемыми.

SAGA

```
// файл sagas/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`'а (`put`).

В Redux DevTools можно увидеть, что теперь в ответ на `Action` с типом `CHANGE_SEARCH_FIELD` генерируется `Action` с типом `SEARCH_SKILLS_REQUEST`.

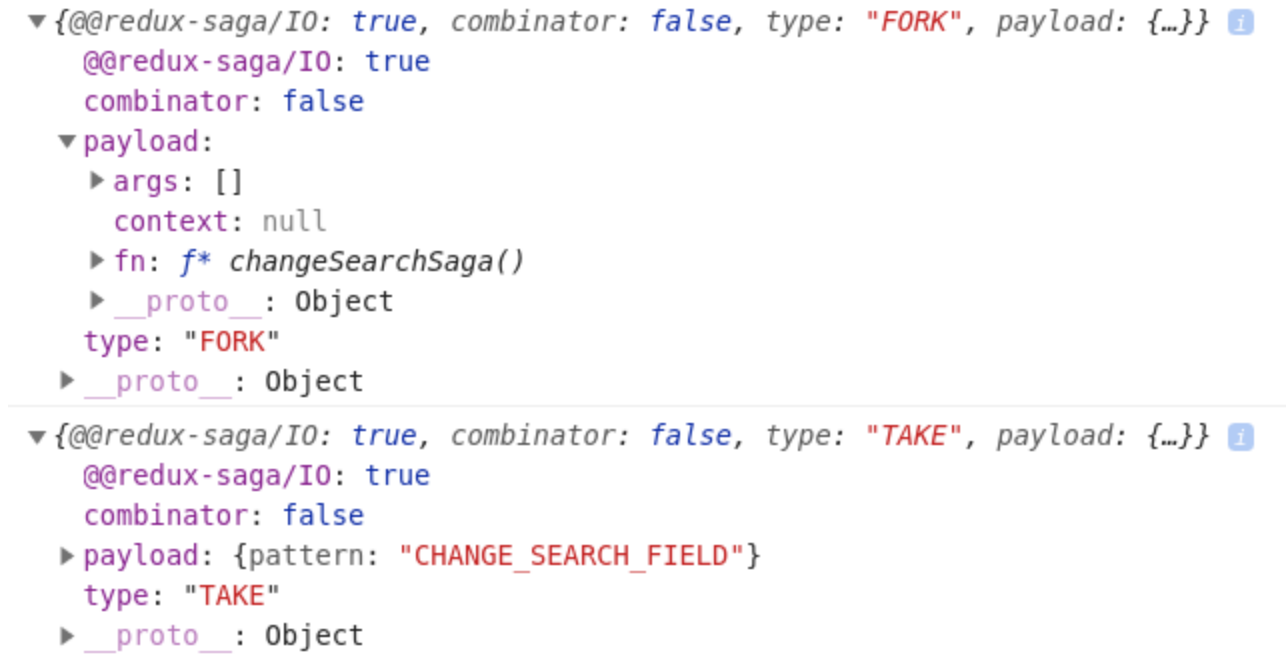
BLOCKING VS NON-BLOCKING

Нужно отметить следующее:

- `take` - "блокирующий вызов", т.е. пока не придёт `Action` с типом `CHANGE_SEARCH_FIELD` Saga заблокируется в этой точке и дальнейшее выполнение приостановится
- `spawn` и `put` - "неблокирующие вызовы", т.е. 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'ами:



EFFECTS

EFFECTS

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

Соответственно, ключевые для нас на данный момент:

- `take` - генерирует эффект для ожидания `Action` определённого типа
- `put` - генерирует эффект для `dispatch` определённого `Action`'а

EFFECTS

Давайте попробуем с помощью эффектов добиться следующего результата:

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

HAPPY PATH

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

```
// 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 так же принимать callback, по которому определять, выбирать этот `Action` или нет:

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

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

ACTION FILTER

Вынесем шаблон фильтрации `Action`'а 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));
  }
}
```

EFFECTS

Эффектов в Redux Saga достаточно много. В общем, их можно разделить на несколько категорий:

- redux specific
- generic
- concurrency

Полный список эффектов вместе с их описанием вы можете найти на странице: API Redux Saga.

debounce

Среди эффектов есть `debounce`, который позволяет запустить Saga только после того, как перестанут поступать `Action` в течение определённого количества ms.

Это высокоуровневый эффект, который уже содержит в себе и циклы, и `take` и всё остальное:

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

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

debounce

Можно посмотреть, как этот эффект устроен внутри (если у вас появится желание написать аналогичный):

```
const debounce = (ms, pattern, task, ...args) => fork(function*() {
  while (true) {
    let action = yield take(pattern);
    while (true) {
      // race - "аналог" Promise.race
      // в ответ получаем объект с полем первого завершившегося эффекта
      const { debounced, latestAction } = yield race({
        debounced: delay(ms),
        latestAction: take(pattern),
      });
      if (debounced) {
        // fork - non-blocking вызов для выполнения действий
        yield fork(task, ...args, action);
        break;
      }
      action = latestAction;
    }
  }
});
```

ОРГАНИЗАЦИЯ КОДА

В Redux Saga принято организовывать код с паттерном Watcher/Worker.

Наши Saga уже следует этому паттерну, но необходимо разобраться с сутью.

WATCHERS AND WORKERS

WATCHERS AND WORKERS

Приято разделять поток управления на две Saga:

- Watcher - ждёт нужный `Action` и запускает `fork` Worker'а
- Worker - обрабатывает `Action` и завершает свою работу

WATCHERS

```
1  // worker
2  function *handleChangeSearchSaga(action) {
3    yield put(searchSkillsRequest(action.payload.search));
4  }
5
6  // watcher
7  function* watchChangeSearchSaga() {
8    yield debounce(100, filterChangeSearchAction, handleChangeSearchSaga);
9  }
```

Опять же, по поводу имён единого соглашения нет, но Watcher'ы принято начинать с префикса `watch`

ВЫЗОВ API

ВЫЗОВ API

Для работы с API напишем отдельный модуль:

```
// файл api/index.js
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`:

```
1  // watcher
2  function* watchSearchSkillsSaga() {
3    while (true) {
4      const action = yield take(SEARCH_SKILLS_REQUEST);
5      yield fork(handleSearchSkillsSaga, action);
6    }
7  }
8
9  export default function* saga() {
10   yield spawn(watchChangeSearchSaga);
11   yield spawn(watchSearchSkillsSaga)
12 }
```

WORKER

И `Worker`:

```
1  // worker
2  function* handleSearchSkillsSaga(action) {
3    const data = yield call(searchSkills, action.payload.search);
4    yield put(searchSkillsSuccess(data));
5  }
```

`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:

```
re
▼(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)
```

Попробуем посмотреть на более высокоуровневые инструменты.

`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

Явно, нам нужно использовать `takeLatest`:

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

ОБРАБОТКА ОШИБОК

ОБРАБОТКА ОШИБОК

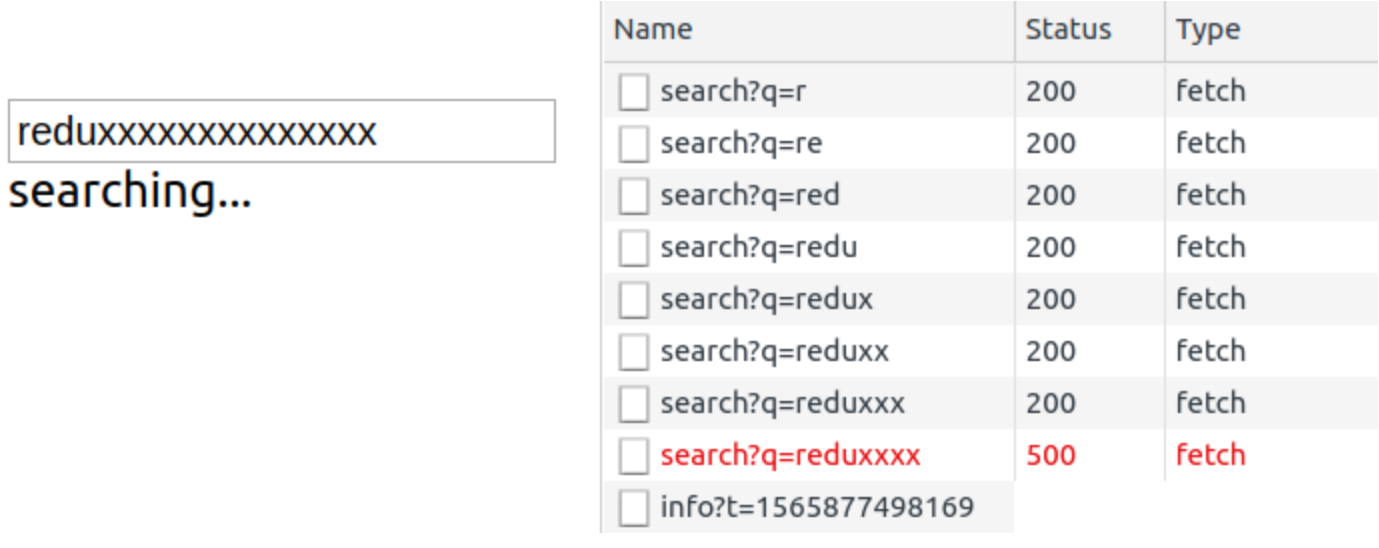
Осталось только разобраться с обработкой ошибок, поскольку для Worker'ов делается `fork` - любое необработанное исключение приведёт к обрушению `Watcher`'а, после чего он уже не сможет следить за `Action`'ами.

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:



ОБРАБОТКА ОШИБОК

В начале лекции мы с вами уже посмотрели, как перехватывать ошибки, здесь всё достаточно стандартно (`try-catch`):

```
// worker
function* handleSearchSkillsSaga(action) {
  try {
    const data = yield call(searchSkills, action.payload.search);
    yield put(searchSkillsSuccess(data));
  } catch (e) {
    yield put(searchSkillsFailure(e.message));
  }
}
```

RETRY

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 = 1000; // in ms
    const data = yield retry(
      retryCount, retryDelay, searchSkills, action.payload.search
    );
    yield put(searchSkillsSuccess(data));
  } catch (e) {
    yield put(searchSkillsFailure(e.message));
  }
}
```

ИТОГИ

Сегодня мы рассмотрели достаточно сложные темы: повторили генераторы и познакомились с Redux Saga.

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

Redux Saga благодаря использованию генераторов предоставляет возможности, позволяющие организовывать целые Flow (логические цепочки действий).

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

нетология
университет интернет-профессий

Задавайте вопросы и напишите отзыв о лекции!

ВЛАДИМИР ЯЗЫКОВ


neizerth@gmail.com


vk.com/neizerth