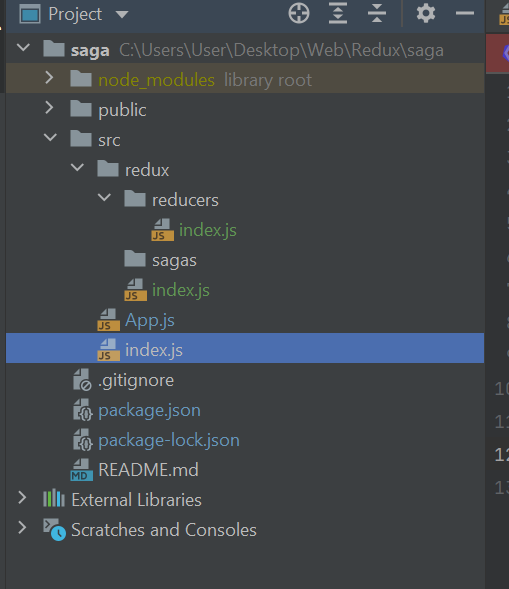
# Создаем приложение

npx create-react-app mysaga

npm –i redux react-redux redux-saga

react-redux redux-saga redux @reduxjs/toolkit react-dnd react-dnd-html5-backend

# Структура проекта



# Создаем reducer

export default function reducer*(*state = *{}*, action*) {* return state;  
*}*;

# Создаем store в index.js в папке redux

import *{* ***createStore***, applyMiddleware*}* from "redux";  
import createSagaMiddleware from 'redux-saga';   
import reducer from "./reducers";  
  
const sagaMiddleware = createSagaMiddleware*()*; // создаем мидлвэр  
const ***store*** = ***createStore****(*reducer, applyMiddleware*(*sagaMiddleware*))*; // прокидываем созданный мидлвэр   
  
  
export default ***store***;

Далее в App.js используем useSelector для доступа к store

import *{*useSelector*}* from "react-redux";  
  
function App*() {* const store = useSelector*(*store => store*)* return *(  
 <*div*>* redux-saga tutorial  
 *</*div*>  
 )*;  
*}*export default App;

# Используем Provider в основном index.js для передачи store

import ***React*** from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';  
import *{* Provider*}* from "react-redux";  
import ***store*** from "./redux";  
  
ReactDOM.***render****(  
 <*React.StrictMode*>  
 <*Provider store=*{****store****}>  
 <*App */>  
 </*Provider*>  
 </*React.StrictMode*>*,  
 ***document***.getElementById*(*'root'*)  
)*;

Чтобы подключить Redux-DevTools, делаю так:

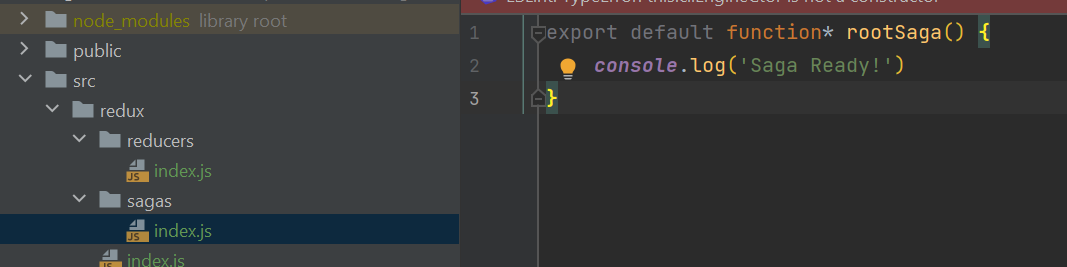
const ***store*** = ***createStore****(* reducer,  
 ***window***.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_*(* applyMiddleware*(*sagaMiddleware*)  
 )*

А лучше

npm install --save redux-devtools-extension

import *{* composeWithDevTools *}* from 'redux-devtools-extension';  
  
const sagaMiddleware = createSagaMiddleware*()*; // создаем мидлвэр  
const ***store*** = ***createStore****(* reducer,  
 composeWithDevTools*(* applyMiddleware*(*sagaMiddleware*)  
 )*

# Создаем саги



## Запускаем run

Но обязательно после того, как был вызван метод applyMiddleware, т.к. он уже имеет доступ к store, dispatch и прочим вещам в redux. А функция, которую мы в него передаем (в нашем случае rootSaga) обязательно должна быть генератором. Корневая сага (в нашем случае rootSaga) создает наш самый верхний процесс (остальные саги будут отрабатывать много других разных процессов по взаимодействию с разными API). Далее наша rootSaga всего лишь будет запускать watchers.

const sagaMiddleware = createSagaMiddleware(); *// создаем мидлвэр*const *store* = *createStore*(  
 reducer,  
 composeWithDevTools(  
 applyMiddleware(sagaMiddleware)  
 )  
  
); *// прокидываем созданный мидлвэр*sagaMiddleware.run(rootSaga); *//run динамически запускает сагу*export default *store*;

## saga Watchers

Следят за actions в приложении. И когда происходят экшены, то выполняет какие-то действия.

export function\* watchClickSaga**() {  
  
}**export default function\* rootSaga**() {** yield watchClickSaga**()**;  
**}**

Здесь описываются всего лишь экшены, которые будут происходить в приложении

## saga Worker

Это saga, которая будет запускаться в зависимости от какого-то экшена.

export function\* workerSaga() {  
  
}  
export function\* watchClickSaga() {  
  
}

В этих воркерах и описывается бизнес-логика приложения: запросы, обращения в кэш, работа с браузерными API, и любые другие асинхронные действия. Эти воркеры потом и тестируют.

Далее наша rootSaga всего лишь будет запускать watchers.

export default function\* rootSaga**() {** yield watchClickSaga**()**;  
**}**

## saga Эффекты

Это вспомогательные функции, которые создают простые объекты с инструкциями, которые выполняет reduxSagaMiddleware. Т.е. эффект прокидывает в сагу инструкции, которые ей необходимо выполнять.

### take эффект

import **{** take **}** from **'redux-saga/effects'**

export function\* watchClickSaga**() {** yield take**('CLICK')  
 *console***.log**('click from saga')  
  
}**

Теперь take ждет пока не выполнится действие по шаблону 'CLICK'. В генераторе watchClickSaga дальнейший код и дальнейшие действия приостанавливаются пока не выполнится действие по 'CLICK'.

Теперь этот 'CLICK' нужно задиспатчить.

#### useDispatch()

import **{**useDispatch, useSelector**}** from **"react-redux"**;  
  
function App**() {** const store = useSelector**(***store* => *store***)** const dispatch = useDispatch**()**;  
 ***console***.log**(**store**)** return **(  
 <div>  
 redux-saga tutorial  
 <button** *onClick*=**{()**=> dispatch**({type**: **'CLICK'})}  
 >  
 </button>**

#### while()

Теперь код в watchClickSaga ждет клика по кнопке. Как только произошел клик, то выполняется код console.log('click from saga'). Но при повторном нажатии ничего не произойдет.

Чтобы этот код выполнялся всегда по клику, нужно его зациклить с помощью while()

export function\* watchClickSaga**() {** while **(**true**) {** yield take**('CLICK')  
 *console***.log**('click from saga')  
 }  
}**

Но правильно вынести логику из вотчера в воркер. А внутри вотчера вызывать воркер сагу.

export function\* workerSaga**() {  
 *console***.log**('click from saga')  
}**export function\* watchClickSaga**() {** while **(**true**) {** yield take**('CLICK')** yield workerSaga**()  
 }  
}**

### takeEvery эффект

Позволяет не писать while. В него мы сразу передаем наш тип 'CLICK' и вторым аргументом функцию, которую он по этому типу должен запускать.

export function\* workerSaga**() {  
 *console***.log**('click from saga')  
}**export function\* watchClickSaga**() {** yield takeEvery**('CLICK'**, workerSaga**)  
}**

### takeLatest эффект

При работе с асинхронными операциями он ждет результата выполнения асинхронного кода, и сколько бы мы не кликали, результат по клику произойдет только один последний, после получения результата выполнения асинхронного кода (в примере это таймаут в 1 сек., т.е. воркер по последнему клику выполнится только один через 1 секунду. А потом опять сколько бы ни кликали, следующий раз воркер выполнится еще через 1 сек.).

const wait = **(***t***)** => new Promise**((***resolve* => **{** setTimeout**(***resolve*, *t***)  
}))**export function\* workerSaga**() {** yield wait**(**1000**)  
 *console***.log**('click from saga')  
}**export function\* watchClickSaga**() {** yield takeLatest**('CLICK'**, workerSaga**)  
}**

### takeLeading эффект

Делает то же, что и takeLatest, но выполняет первое действие после выполнения асинхронного кода.

При работе с асинхронными операциями он ждет результата выполнения асинхронного кода, и сколько бы мы не кликали, результат по клику произойдет только один первый, после получения результата выполнения асинхронного кода (в примере это таймаут в 1 сек., т.е. воркер по первому клику выполнится только один через 1 секунду. А потом опять сколько бы ни кликали, следующий раз воркер выполнится еще через 1 сек.).

import **{** take, takeEvery, takeLatest, takeLeading **}** from **'redux-saga/effects'**const wait = **(***t***)** => new Promise**((***resolve* => **{** setTimeout**(***resolve*, *t***)  
}))**export function\* workerSaga**() {** yield wait**(**1000**)  
 *console***.log**('click from saga')  
}**export function\* watchClickSaga**() {** yield takeLeading**('CLICK'**, workerSaga**)  
}**

# Реальный запрос на API

// sagas\index.js

import **{** takeEvery, put **}** from **'redux-saga/effects'**;  
  
async function getPeople**() {** const request = await fetch**('https://kitsu.io/api/edge/anime?page[limit]=20&page[offset]=0')**;  
 const data = await request.json**()**;  
 return data;  
**}**export function\* workerSaga**() {** const data = yield getPeople**()**;  
 // эффект put указывает мидлвэр отправить действие в store, т.е задиспатчить  
 yield put**( {type**: **'SET\_PEOPLE'**, **payload**: data.**data})  
}**export function\* watchClickSaga**() {** yield takeEvery**('CLICK'**, workerSaga**)**;  
**}**export default function\* rootSaga**() {** yield watchClickSaga**()**;  
**}**

// reducers\index.js

const initial = **{  
 people**: **[]**,  
**}**;  
  
export default function reducer**(***state* = initial, *action***) {** switch **(***action*.**type) {** case **'SET\_PEOPLE'**: **{** return **{** ...*state*,  
 **people**: **[** ...*state*.**people**,  
 ...*action*.**payload  
 ]  
 }  
 }** default:  
 return *state*;  
 **}  
}**;

## put эффект

указывает мидлвэр отправить действие в store, т.е задиспатчить

yield put**( {type**: **'SET\_PEOPLE'**, **payload**: data.**data})**

## call эффект

нужен для вызова асинхронных действий.

import **{** takeEvery, put, call **}** from **'redux-saga/effects'**;

export function\* workerSaga**() {** const data = yield call**(**getPeople, 1**)**;  
 // эффект put указывает мидлвэр отправить действие в store, т.е задиспатчить  
 yield put**( {type**: **'SET\_PEOPLE'**, **payload**: data.**data})  
}**

В эффект call передаем асинхронную функцию. Если функции нужны доп. Параметры, то передаем их следующим аргументом. Например, если бы функция getPeople(id) ждала на вход id, то мы можем этот id передать вторым аргументом.

Эффект call, как и эффект take, останавливает выполнение кода, пока не зарезолвится промис. Поэтому в примере ниже сначала выполняется запрос к people , потом к planets , и только после этого эффект put добавляет это в store

mport **{** takeEvery, put, call **}** from **'redux-saga/effects'**;  
  
async function swapiGet**(***pattern***) {** const request = await fetch**(`https://swapi.dev/api/**$**{***pattern***}`)**;  
 const data = await request.json**()**;  
 return data;  
**}**export function\* workerSaga**() {** const people = yield call**(**swapiGet, **'people')**;  
 const planets = yield call**(**swapiGet, **'planets')**;  
 // эффект put указывает мидлвэр отправить действие в store, т.е задиспатчить  
 yield put**( {type**: **'SET\_PEOPLE'**, **payload**: people.**results})** yield put**( {type**: **'SET\_PLANETS'**, **payload**: planets.**results})  
}**export function\* watchClickSaga**() {** yield takeEvery**('LOAD\_DATE'**, workerSaga**)**;  
**}**export default function\* rootSaga**() {** yield watchClickSaga**()**;  
**}**

## fork эффект

если мне нужно чтобы запросы не были блокирующими (как call), то использую эффект fork.

Т.е. с помощью эффекта fork мы можем запускать неблокирующие (асинхронные) вызовы саг.

Для этого нужно создать 2 новые саги, и перенести туда запрос к АПИ (например запрос к people) и put для записи в store

А уже в workerSaga с помощью эффекта fork передаю эти новые саги

export function\* loadPeople**() {** const people = yield call**(**swapiGet, **'people')**;  
 yield put**( {type**: **'SET\_PEOPLE'**, **payload**: people.**results})  
}**export function\* loadPlanets**() {** const planets = yield call**(**swapiGet, **'planets')**;  
 yield put**( {type**: **'SET\_PLANETS'**, **payload**: planets.**results})  
}**export function\* workerSaga**() {** yield fork**(**loadPeople**)**;  
 yield fork**(**loadPlanets**)**;  
**}**export function\* watchClickSaga**() {** yield takeEvery**('LOAD\_DATE'**, workerSaga**)**;  
**}**export default function\* rootSaga**() {** yield watchClickSaga**()**;  
**}**

Теперь эти запросы выполняются асинхронно, и записывают в store тоже асинхронно.

Теперь вы можем и в rootSaga использовать fork (поменяли название вотчера – необязательно).

export function\* watchLoadDataSaga**() {** yield takeEvery**('LOAD\_DATE'**, workerSaga**)**;  
**}**export default function\* rootSaga**() {** yield fork**(**watchLoadDataSaga**)**;  
**}**

При этом, все ошибки в дочерних задачах, вызванных fork, всплывают к родительской задаче. И если случай ошибки не обработан, то выполнение дальнейших задач блокируется. Т.е. если в people произошла ошибка, то и загрузка planets тоже не выполнится.

Если необходимо в корне родительской задачи создать совсем независимую задачу, то используют не эффект fork, а эффект spawn

## spawn эффект

Если в примере используем эффект spawn вместо fork, то при появлении ошибки в people, загрузка planets все равно выполнится. Т.е. эффект spawn создает совершенно независимые задачи, которые не зависят ни от родителя, ни друг от друга.

export function\* workerSaga**() {** yield spawn**(**loadPeople**)**;  
 yield spawn**(**loadPlanets**)**;  
**}**export function\* watchLoadDataSaga**() {** yield takeEvery**('LOAD\_DATE'**, workerSaga**)**;  
**}**export default function\* rootSaga**() {** yield fork**(**watchLoadDataSaga**)**;  
**}**

При этом, несмотря на то что и spawn и fork создают неблокирующие задачи, к ним в дальнейшем можно присоединиться используя оператор join

## оператор join

Например, получаем объект task (допустим loadPeople возвращает people)

export function\* loadPeople**() {** const people = yield call**(**swapiGet, **'people')**;  
 yield put**( {type**: **'SET\_PEOPLE'**, **payload**: people.**results})** return people  
**}**export function\* loadPlanets**() {** const planets = yield call**(**swapiGet, **'planets')**;  
 yield put**( {type**: **'SET\_PLANETS'**, **payload**: planets.**results})  
}**export function\* workerSaga**() {** const task = yield fork**(**loadPeople**)**;  
 yield spawn**(**loadPlanets**)**;  
  
 const people = yield join**(**task**)  
 *console***.log**(**people**)  
}**export function\* watchLoadDataSaga**() {** yield takeEvery**('LOAD\_DATE'**, workerSaga**)**;  
**}**

В этом коде ***const people = yield join(task)*** становится блокирующим. Он ждет выполнения ***yield fork(loadPeople);,*** и только потом выводит ***console***.log**(**people**)**

## select эффект

Позволяет получить доступ к store. Не является блокирующим. И мы получаем store на момент выполнения кода.

export function\* workerSaga**() {** const task = yield spawn**(**loadPeople**)**;  
 yield spawn**(**loadPlanets**)**;  
  
 const people = yield join**(**task**)  
 *console***.log**(**people**)** const store = yield select**(***s* => *s***)  
}**

# Правильное написание root-saga

Используем spawn, т.к. он не блокирует выполнение следующих саг, и не блокирует в случае возникновения ошибок в какой-либо саге

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

export function\* saga1() {

    console.log('saga1');

}

export function\* saga2() {

    console.log('saga2');

}

export function\* saga3() {

    console.log('saga3');

}

export default function\* rootSaga() {

    yield spawn(saga1); //auth

    yield spawn(saga2);  //users

    yield spawn(saga2);  //payments

}

Но чтобы иметь возможность обработать ошибки при выполнении какой-либо саги, лучше использовать структуру ниже:

export default function\* rootSaga() {

    const sagas = [saga1, saga2, saga3];

    const retrySagas = yield sagas.map(saga => {

        return spawn(function\* () {

            while (true) {

                try {

                    yield call(saga);

                    break;

                }

                catch (e) {

                    console.log(e);

                }

            }

        });

    });

    yield all(retrySagas);

}

Так мы с помощью эффекта all запускаем наши саги в цикле, но параллельно. Эффект all похож на Promise.all

# Доменная логика

Если данные нужно загружать сразу при входе на страницу, то нужно делать так:

import **{** spawn, call, all, delay, fork **}** from **'redux-saga/effects'**;  
  
export function\* auth**() {** yield delay**(**2000**)**; // эффект delay позволяет подождать сколько-то миллисекунд  
  
 ***console***.log**('auth Ok')**;  
 return true;  
**}**;  
  
export function\* loadUsers**() {** const request = yield call**(**fetch, **'https://swapi.dev/api/people')**;  
 //const data = yield call(request.json.bind(request));// можно сделать так, но call может принимать массив:  
 const data = yield call**([**request, request.**json])**;  
 ***console***.log**('data'**, data**)  
}**;  
  
export function\* loadBasicSaga**() {** yield all**( [** fork**(**auth**)**,  
 fork**(**loadUsers**)  
  
 ])  
}**;

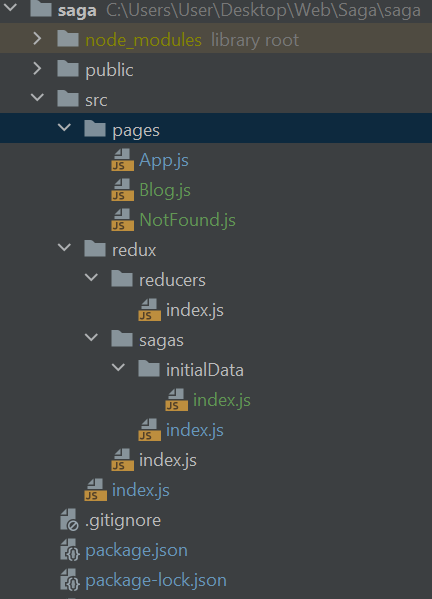
В эти моменты(в функциях auth, loadUsers) мы уже можем диспатчить какие-то данные в store

### Используем роутинг

npm i react-router react-router-dom –save

npm i history –save

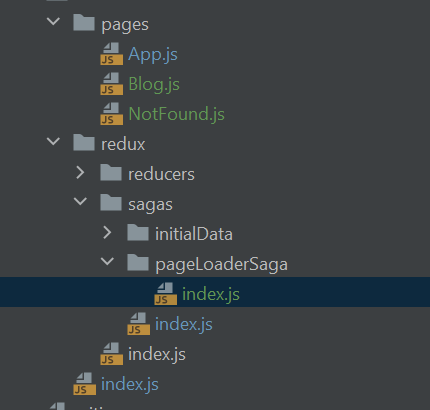
import **{** createBrowserHistory**}** from **"history"**;  
import **{** Switch, Route **}** from **'react-router-dom'**;  
import Blog from **"./pages/Blog"**;  
import NotFound from **"./pages/NotFound"**;  
  
const history = createBrowserHistory**()**;  
  
ReactDOM.***render*(  
 <React.StrictMode>  
 <BrowserRouter>  
 <Provider** *store*=**{*store*}>  
 <Router** *history*=**{*history*}>  
 <Routes>  
 <Route** *path*='/' *exact element*=**{<App/>} />  
 <Route** *path*='/blog' *element*=**{<Blog />} />  
 <Route** *path*='\*' *element*=**{<NotFound />}/>  
 </Routes>  
 </Router>  
 </Provider>  
 </BrowserRouter>  
 </React.StrictMode>**,  
 ***document***.getElementById**('root')  
)**;



В компоненте, например, Blog диспатчим экшн для роута на другую старницу

import **{**useDispatch**}** from **"react-redux"**;  
import **{**useEffect**}** from **"react"**;  
  
const Blog = **()** => **{** const dispatch = useDispatch**()**;  
  
 useEffect**(()** => **{** dispatch**({type**: **'LOAD\_BLOG\_DATA'})  
 }**, **[])** return **(  
 <div>  
 Blog  
 </div>  
 )  
}**export default Blog;

Создаем сагу для загрузки страницы



import **{**call, apply, takeEvery**}** from **"redux-saga/effects"**;  
  
function\* loadBlogData**() {** const request = yield call**(**fetch, **'http://swapi.dev/api/vehicles')**;  
 const data = yield apply**(**request, request.**json)** // как и call принимает контекст и метод  
  
 ***console***.log**('blog data'**, data**)  
}**export default function\* pageLoaderSaga**() {** yield takeEvery**('LOAD\_BLOG\_DATA'**, loadBlogData**)  
}**

И добавляем ее в root-saga

export default function\* rootSaga**() {** const sagas = **[**loadBasicSaga, pageLoaderSaga**]**;

Так, при роуте на страницу Blog, в useEffect[] диспатчится

useEffect**(()** => **{** dispatch**({type**: **'LOAD\_BLOG\_DATA'})  
}**, **[])**

Срабатывает наша сага pageLoaderSaga, и в консоли отображаются загруженные данные (а могли передать их в store).

# connected-react-router

Чтобы не приходилось постоянно использовать useEffect[] в компонентах-страницах, можно поставить дополнительную библиотеку

npm i connected-react-router –save

Далее нежно переделать наш reducer

Перенесем сюда создание history

const history = createBrowserHistory**()**;

В redux/index.js импортируем history

И добавим в applyMiddleware

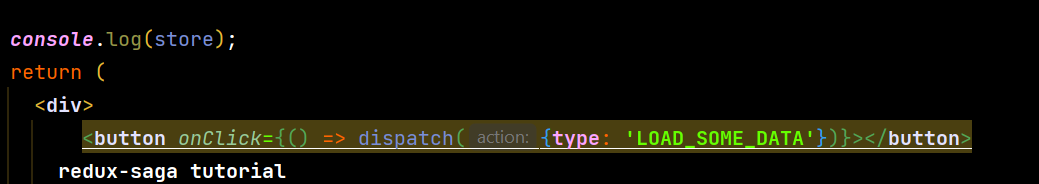
const composeEnhancers =  
 typeof ***window*** === **'object'** && ***window***.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_  
 ? ***window***.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_**({})** : compose;  
  
const sagaMiddleware = createSagaMiddleware**()**; // создаем мидлвэр  
  
const enhancer = composeEnhancers**(**applyMiddleware**(**routerMiddleware**(*history*)**, sagaMiddleware**))**;  
  
const ***store*** = ***createStore*(*reducer***, enhancer**)**; // прокидываем созданный мидлвэр  
  
sagaMiddleware.run**(**rootSaga**)**; //run динамически запускает сагу  
export default ***store***;

А в корневом index.js заменим Router на ConnectRouter

import ***React*** from **'react'**;  
import ReactDOM from **'react-dom'**;  
import App from **'./pages/App'**;  
import **{** Provider**}** from **"react-redux"**;  
import ***store*** from **"./redux"**;  
import **{*history*}** from **'./redux/reducers'**;  
import **{** Route, Routes, BrowserRouter **}** from **'react-router-dom'**;  
import **{** ConnectedRouter**}** from **"connected-react-router"**;  
  
import Blog from **"./pages/Blog"**;  
import NotFound from **"./pages/NotFound"**;  
  
  
ReactDOM.***render*(  
 <React.StrictMode>  
 <BrowserRouter>  
 <Provider** *store*=**{*store*}>  
 <ConnectedRouter** *history*=**{*history*}>  
 <Routes>  
 <Route** *path*='/' *element*=**{<App />} />  
 <Route** *path*='/blog' *element*=**{<Blog />} />  
 <Route** *path*='\*' *element*=**{<NotFound />}/>  
 </Routes>  
 </ConnectedRouter>  
 </Provider>  
 </BrowserRouter>  
 </React.StrictMode>**,  
 ***document***.getElementById**('root')  
)**;

Теперь можно убрать useEffect из компонентов, и отредактировать pageLoaderSaga

# Действия по кнопке



И создаю саги вотчер и воркер

export function \* fetchPlanets **() {** const responce = yield call**(**fetch, **'https://swapi.dev./api/planets')** const data = yield call**([**responce, responce.**json])  
 *console***.log**(**data**)  
}**export function\* watchLoadOnAction**() {** takeLatest**('LOAD\_SOME\_DATA'**, fetchPlanets**)  
  
}**export default function\* rootSaga**() {** const sagas = **[**loadBasicSaga, pageLoaderSaga, watchLoadOnAction**]**;