**Введение во фронтенд-аутентификацию**

В первых уроках темы мы расскажем про процессы аутентификации, авторизации и идентификации. Они применяются почти в каждом сервисе, с которым вы взаимодействуете в интернете. Также вы узнаете больше про стандарт JWT.

Ещё вы научитесь создавать интерфейс для аутентификации в приложениях на «Реакте». Соедините авторизацию на бэкенде с интерфейсом приложения на фронтенде. Узнаете, как обезопасить роуты на фронтенде. Реализуете регистрацию, авторизацию и выход из системы, а также персонализируете контент в зависимости от того, кто использует приложение.

Научитесь применять стандарт JWT (JSON Web Token) в «Реакт-приложении» и узнаете больше об использовании HOC-компонентов и библиотеки React Router. А ещё изучите локальное хранилище браузера и узнаете, как можно использовать его в приложениях.

Впереди много нового. Приступим!

# Аутентификация и авторизация: в чём отличия?

При голосовании на выборах мы предъявляем паспорт. Эта процедура не даёт голосовать одним людям за других, а также голосовать одному избирателю несколько раз.

То же самое происходит в интернете. Мы выкладываем фотографии, комментируем их, ставим лайки. Сервер может отличить одного пользователя от другого, потому что за каждым закреплён свой аккаунт.

Разные пользователи обладают разными правами: среди них могут быть модераторы, администраторы, работники службы поддержки. Сотруднику интернет-магазина нужно дать возможность отслеживать заказ по номеру: так он сможет по телефону проконсультировать покупателя. Пользователям, которые не имеют к этому заказу отношения, такая возможность ни к чему.

В этом уроке разберёмся с тремя длинными словами, которые многие путают: идентификация, аутентификация и авторизация.

## Идентификация

Идентифицировать — значит присваивать кому-то или чему-то имя, номер или другой символ, который позволит отделить объект от других.

Стас выпивает в баре с друзьями и решает познакомиться с девушкой за соседним столиком. Он подходит и спрашивает, как её зовут. Девушка отвечает: «Стася». В этот момент произошла идентификация. Сначала это была абстрактная девушка, а теперь у неё есть идентификатор — «Стася».

## Аутентификация

Аутентифицировать — значит доказать, что идентифицированный является тем, кем назвался. Идентификация — это часть процесса аутентификации. Стася могла назвать не своё имя и быть на самом деле Мариной. Но если бы кроме идентификации Стася прошла аутентификацию, это было бы невозможно.

Стас проводил Стасю и пошёл домой. Поскольку час поздний, его остановили полицейские и потребовали предъявить паспорт. Когда Стас показывает паспорт полицейскому, тот его аутентифицирует. В этом случае доказательством того, что Стас — это Стас, выступает его паспорт.

В интернете процесс аутентификации часто сводится к вводу имени пользователя и пароля. Но бывают и другие способы: ввод кода из смс, ответ на секретный вопрос, снятие отпечатка пальца или снимка лица.

## Авторизация

Авторизация происходит после аутентификации. Это процесс, при котором система принимает решение, какие права вам можно выдать.

Стас зашёл в магазин. На кассе перед ним стоит молодой человек, покупающий сигареты. Кассир спрашивает у него паспорт. Это авторизация: кассир принимает решение, можно продавать этому человеку сигареты или нет.

После авторизации в соцсети пользователи получают права на редактирование собственных публикаций, отправку сообщений от своего имени.

## Итого

Ещё пример. Стас просыпается с утра от звонка в дверь. Он подходит и спрашивает: «Кто там?». «Это Колян, твой сосед,» — отвечают Стасу. Это была идентификация. Затем Стас смотрит в глазок и определяет, действительно ли это Колян. Это аутентификация. И, наконец Стас решает, пустить Коляна или сказать, чтобы тот приходил в другой раз — это авторизация.

Эти три понятия важно отличать одно от другого: так вы избежите путаницы при изучении этой темы. Дальше мы перейдём к практике: реализации аутентификации и авторизации на сервере.

Сотрудник ДПС останавливает машину и просит водителя показать права.



Аутентификация

Увы, неверно. Проверка документов не оставляет шанса выдать себя за другого. Кроме того, она даёт понять сотруднику полиции, что у вас есть право водить машину.



Идентификация



Авторизация

Верно! Проверка документов не только не оставляет шанса выдать себя за кого-то другого, но и позволяет определить, есть ли у вас разрешение управлять автомобилем.

Пользователь оформляет заказ в интернет-магазине и оставляет имя, телефон и электронную почту.



Аутентификация



Идентификация

Правильно. Пользователь идентифицирует себя, но никак не подтверждает, что является именно тем человеком, что указан в заказе. Так можно оставить данные коллеги или родственника, чтобы тот забрал заказ.



Авторизация

Пользователь получает заранее оплаченный заказ в пункте самовывоза:



Аутентификация



Идентификация



Авторизация

Верно! При получении заказа необходимо предъявить паспорт — так интернет-магазин поймёт, что заказ оплачивали именно вы.

Человек показывает паспорт, чтобы получить посылку на почте.



Аутентификация



Идентификация



Авторизация

Правильно! Только один человек может забрать посылку с почты: её получатель.

# Как не проходить аутентификацию при каждом заходе на сайт. JWT

Вы узнали, что такое авторизация и аутентификация, и чем они отличаются. Наверняка вы замечали, что при посещении одной и той же социальной сети вам не приходится каждый раз проходить аутентификацию: повторно вводить почту и пароль при каждом посещении сайта.

Удобно, когда браузер запоминает, что вы уже вводили данные и заходили на сайт. Это реализуется путём создания токена. Что это такое — расскажем прямо сейчас. А уже в следующем спринте вы на практике создадите токен .

# Алгоритм

Мы реализуем аутентификацию так:

1. Пользователь входит в аккаунт по почте и паролю.
2. Сервер генерирует токен — уникальный набор символов — и отправляет его пользователю.
3. Токен сохраняется в браузере пользователя.
4. При повторном открытии сайта пользователь отправляет токен серверу.
5. Сервер проверяет, есть ли токен в запросе и тот ли это токен, что был выдан пользователю раньше.
6. Если проверка прошла успешно, пользователь авторизуется, иначе — получает сообщение об ошибке.
7. Когда пользователь выходит из системы, браузер удаляет токен из памяти. После этого нужно заново входить в систему по почте и паролю.

Весь алгоритм построен на формировании токена. Рассмотрим, что он собой представляет.

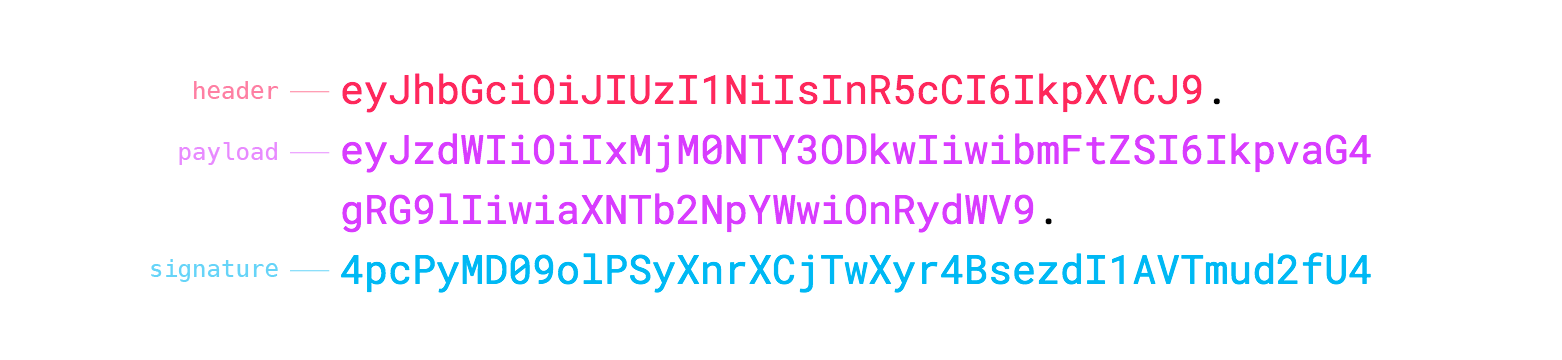
# Структура токена

Токен — набор символов, сгенерированный согласно структуре и правилам. Эти структура и правила определяют стандарт создания токена. Мы будем пользоваться стандартом JWT (JSON Web Token).

JWT основан на стандарте JSON. Токен, созданный по этому стандарту, состоит из трёх частей, каждая несёт в себе определённую информацию:

* header (англ. «хедер») — служебную информацию;
* payload (англ. «полезная нагрузка») — данные, которые токен несёт в себе;
* signature (англ. «подпись») — подпись, которая предотвращает подмену информации в токене.

Эти три части разделены точками:



Разберём каждую из них подробнее.

# Header

Хедер, как правило, содержит два поля:

* тип токена (строка "JWT");
* алгоритм создания подписи (обычно применяется алгоритм HMAC SHA256 или RSA):

Скопировать кодJAVASCRIPT

{

"alg": "HS256",

"typ": "JWT"

}

Полученный JSON-объект кодируется в строку распространённым алгоритмом Base64Url. Это первая часть токена.

# Payload

Пейлоуд — содержит саму информацию, которая была закодирована. В нашем случае это информация о пользователе:

Скопировать кодJAVASCRIPT

{

"name": "Стас Басов",

"\_id": "39dow8ak8402jf23u4do057s"

}

Аналогично хедеру, пейлоуд кодируется в строку. Получаем вторую часть токена.

# Signature

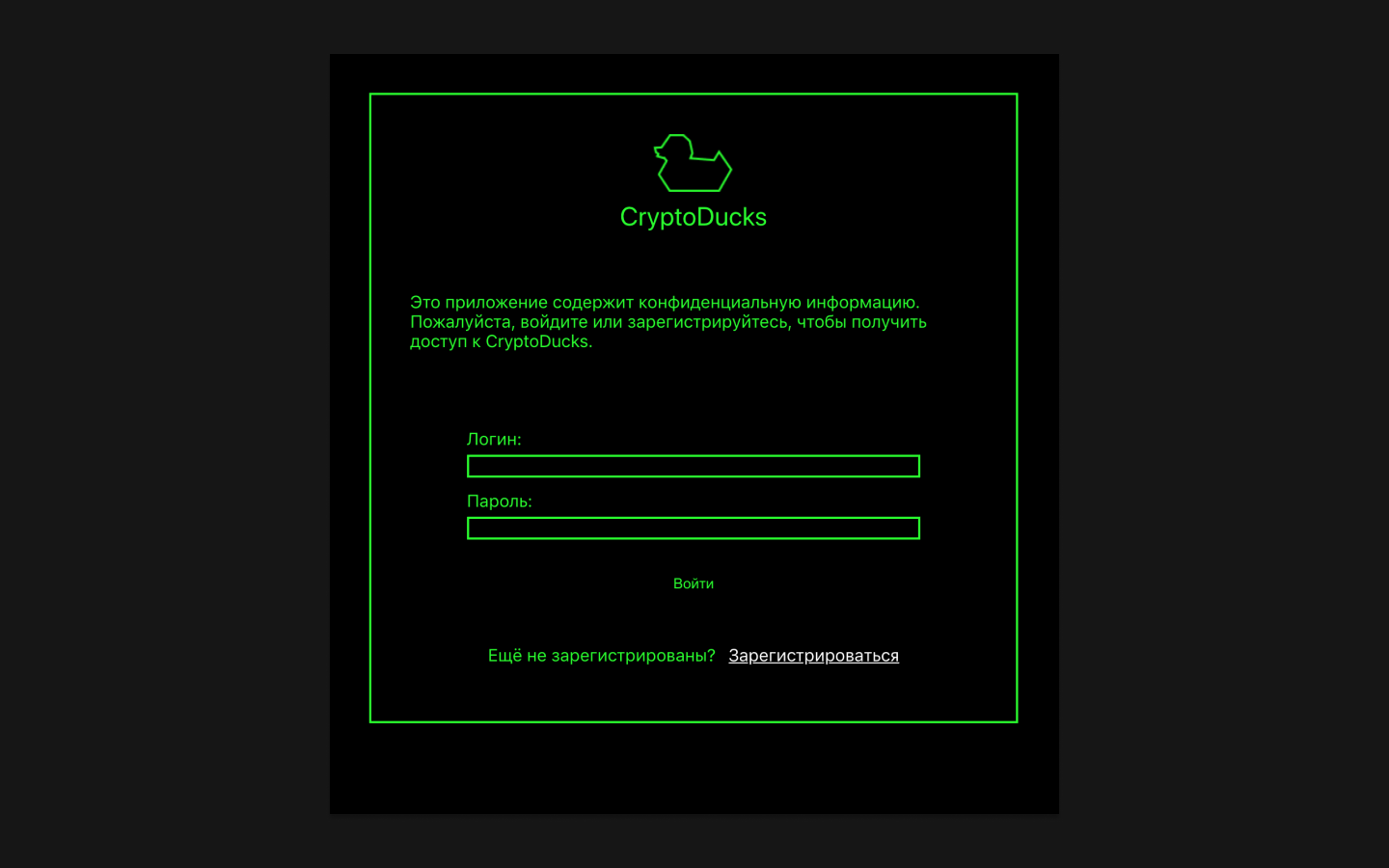
Подпись гарантирует, что содержимое хедера и пейлоуда не изменились после создания токена. Специальный алгоритм высчитывает подпись, исходя из содержимого хедера и пейлоуда. Также алгоритм использует секретный ключ, который известен только серверу.

Теперь вы знаете о самой структуре токена и алгоритме работы с ним. Пока этого достаточно. В следующих уроках вы научитесь работать с токеном на стороне пользователя — сохранять полученный от сервиса токен, и отправлять его при наличии.

Переходите дальше — там вы познакомитесь с новым проектом.

# Встречайте: новый проект!

Чтобы практиковаться было интереснее, мы подготовили новый проект — «Криптоутки». Компоненты, экраны приложения и стили для проекта уже описаны. Сосредоточимся на создании аутентификации и авторизации «Реакт-проекта». Но для начала разберёмся с его содержимым и структурой.



# Компоненты

Проект сгенерирован с помощью [Create React App](https://praktikum.yandex.ru/trainer/web/lesson/0a1aa246-3671-4697-a82c-53a6b603866e" \t "_blank). Мы добавили компоненты для четырёх основных экранов приложения: Login, Register, Ducks и MyProfile. Как видно из названия, Login относится к авторизации, а Register — к регистрации. Компонент Ducks будет отображать фид с картинками и информацией о криптоутках. А при обращении к компоненту MyProfile пользователь получит информацию о своём профиле.

# Маршруты

Для каждого из четырёх основных компонентов мы создали соответствующие флаги и настроили маршрутизацию с помощью React Router: /login, /register, /my-profile и /ducks. Наша цель — сделать первый и второй маршрут публичными (доступными для любого пользователя), а третий и четвёртый — приватными, чтобы доступ к этим экранам был только у авторизованных пользователей.

# Протестируем

Ниже вы можете протестировать готовое приложение в интерактивном примере. На главном экране вы увидите ссылку на страницу регистрации. Можно перейти по ней, а можно ввести уже готовые логин и пароль. Вот логин: Mallard Вот пароль: Quack42

Важно, что экран «Утки» должен быть доступен только авторизованным пользователям. То есть вы попадёте на него автоматически только после входа в свой аккаунт. Вверху экрана появится панель навигации со ссылками: «Утки», «Мой профиль» и выход из системы. Как и «Утки», экран «Мой профиль» должен быть приватным и показывать пользователю его данные. При выходе из системы пользователь должен быть переадресован на экран авторизации.

К концу темы вы реализуете всю эту функциональность. Переходите к следующему уроку и приступим!

# Переадресация пользователя

Рассмотрим уже знакомый компонент App.js. Именно внутри него мы будем настраивать роуты. Так код выглядит сейчас:

Скопировать кодJSX

*// App.js*

import React from 'react';

import { Route, Switch } from 'react-router-dom';

*// импортируем компоненты приложения*

import Login from './Login.js';

import Register from './Register.js';

import Ducks from './Ducks.js';

import MyProfile from './MyProfile.js';

*// импортируем CSS*

import './styles/App.css';

class App extends React.Component {

constructor(props){

super(props);

}

render(){

return (

<Switch>

<Route path="/ducks">

<Ducks />

</Route>

<Route path="/my-profile">

<MyProfile />

</Route>

<Route path="/login">

<div className="loginContainer">

<Login />

</div>

</Route>

<Route path="/register">

<div className="registerContainer">

<Register />

</div>

</Route>

</Switch>

)

}

}

export default App;

В коде компонента App можно насчитать четыре роута и столько же компонентов: Ducks, MyProfile, Login и Register.

Не забудьте, что все Switch и Route обернуты в BrowserRouter в index.js:

Скопировать кодJSX

*// index.js*

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import './index.css';

import App from './components/App';

import \* as serviceWorker from './serviceWorker';

ReactDOM.render(

<React.StrictMode>

<BrowserRouter>

<App/>

</BrowserRouter>

</React.StrictMode>,

document.getElementById('root')

);

*// If you want your app to work offline and load faster, you can change*

*// unregister() to register() below. Note this comes with some pitfalls.*

*// Learn more about service workers: [https://bit.ly/CRA-PWA](https://bit.ly/CRA-PWA)*

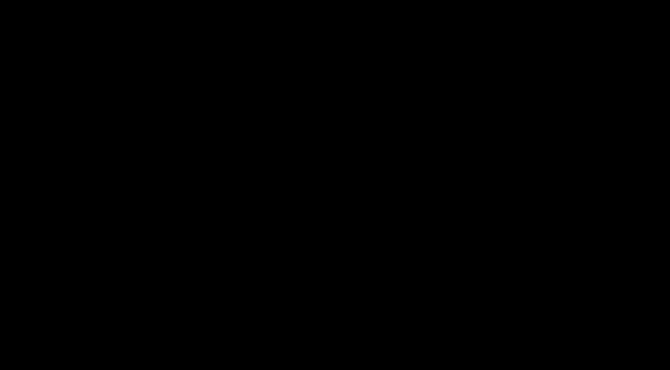
serviceWorker.unregister();

Но есть две проблемы. Когда пользователь загружает приложение, путь по умолчанию / не имеет соответствующего роута. При переходе по этому адресу ничего не отрисуется — пользователь увидит чёрный экран. Кроме этого, неавторизованные и незарегистрированные пользователи могут посетить любой из этих роутов, если введут URL вручную в адресной строке браузера. То есть роуты не защищены.

Посмотрим, что отобразят роуты, если ничего не менять в коде.

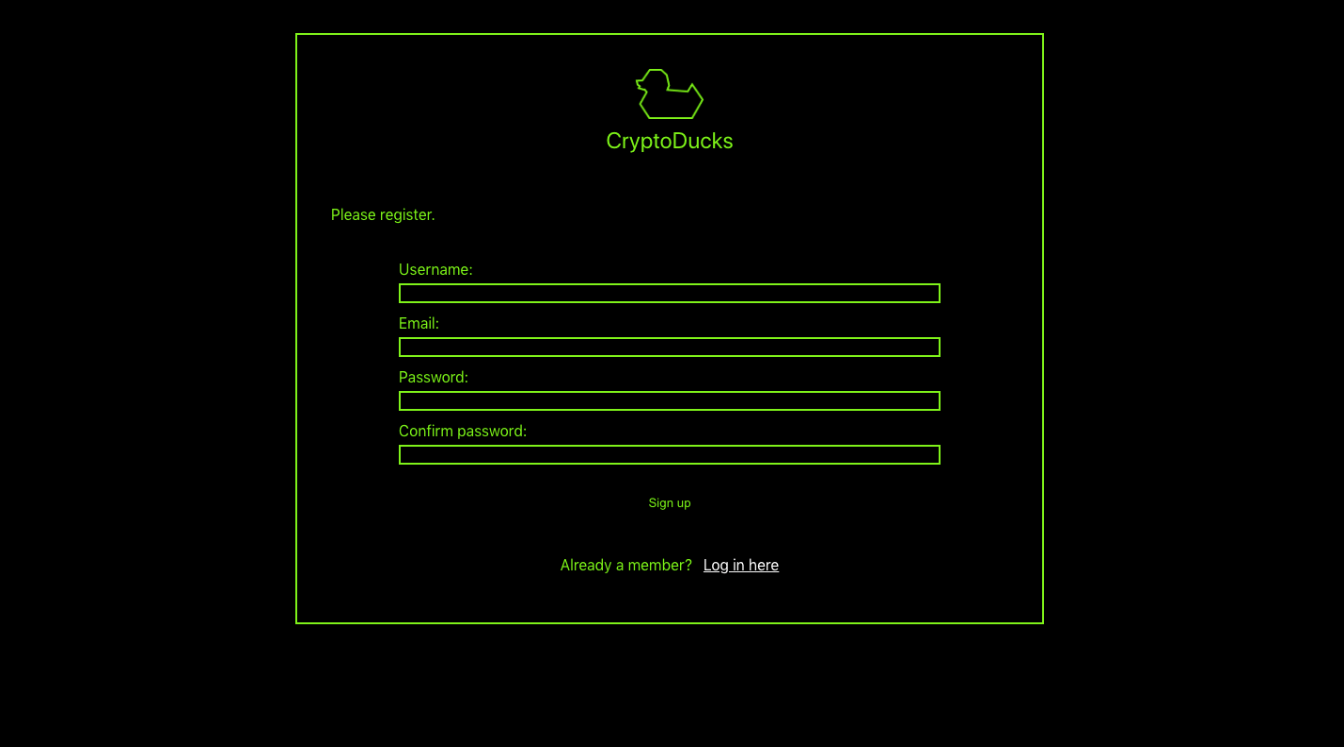
## Лучше один раз увидеть, чем сто раз услышать

Если пользователь перейдёт по /, то увидит «Чёрный квадрат» Малевича (только прямоугольный):



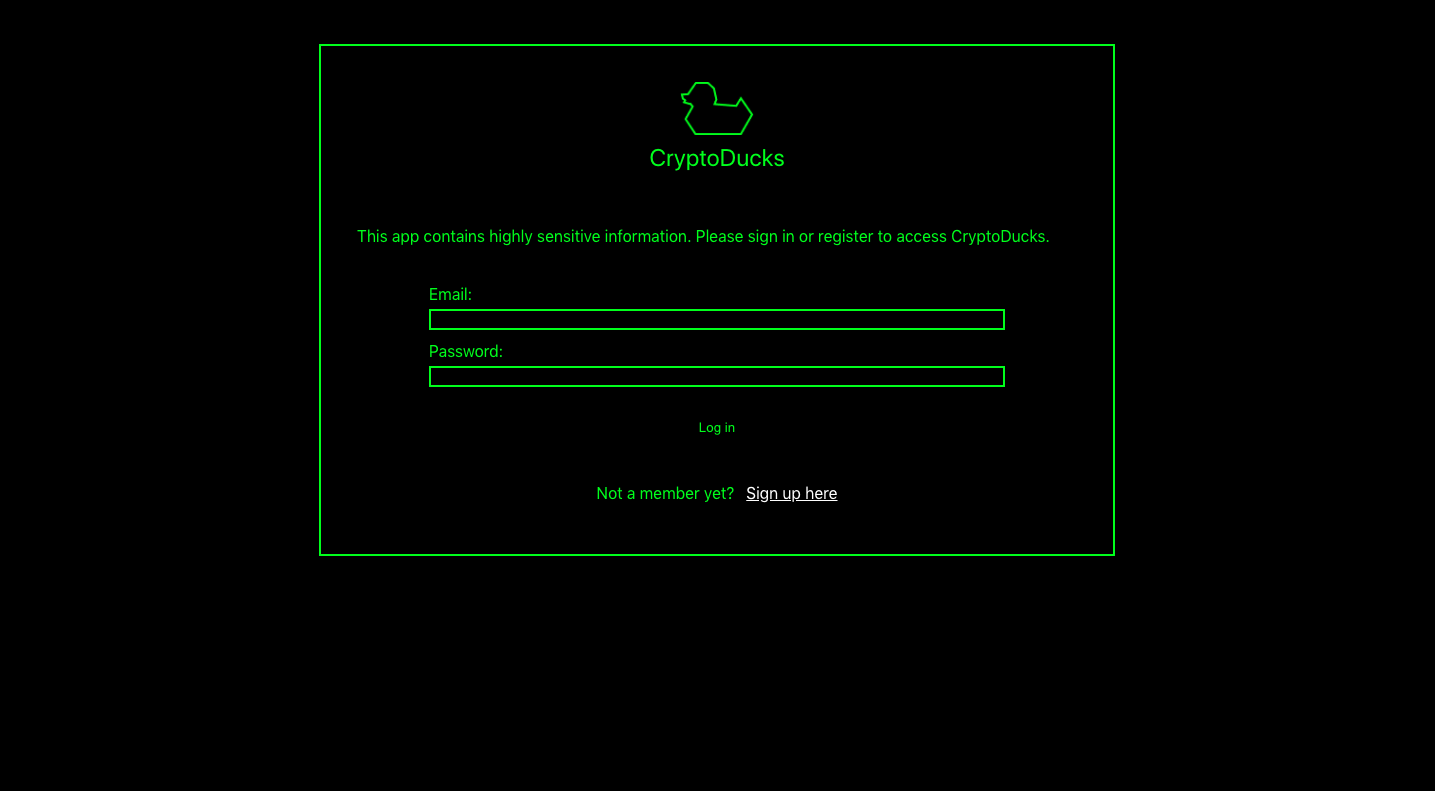
Вглядитесь в загадочное чёрное пространство

Роут /register отрисует экран регистрации:



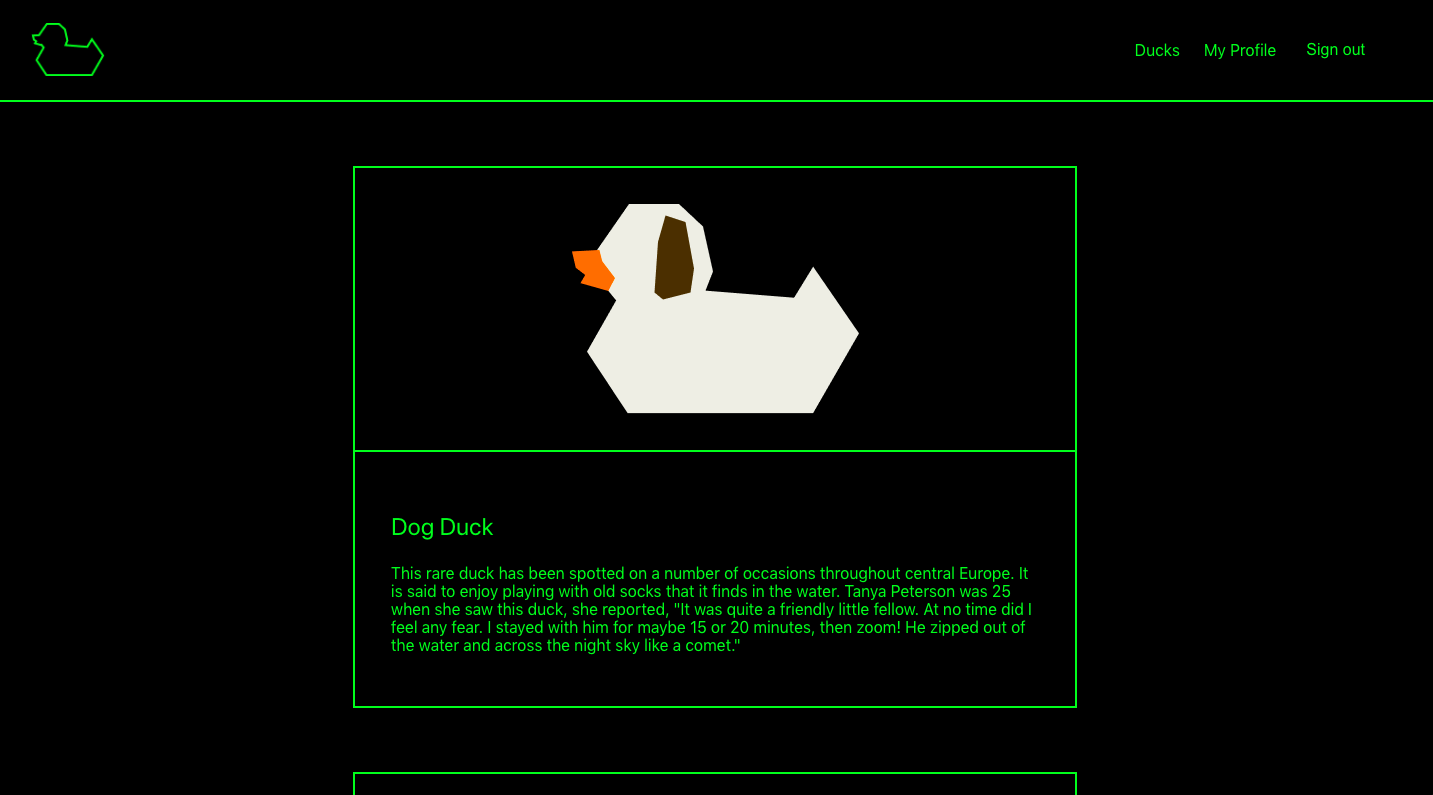
Экран регистрации

По адресу /login пользователь увидит экран авторизации:



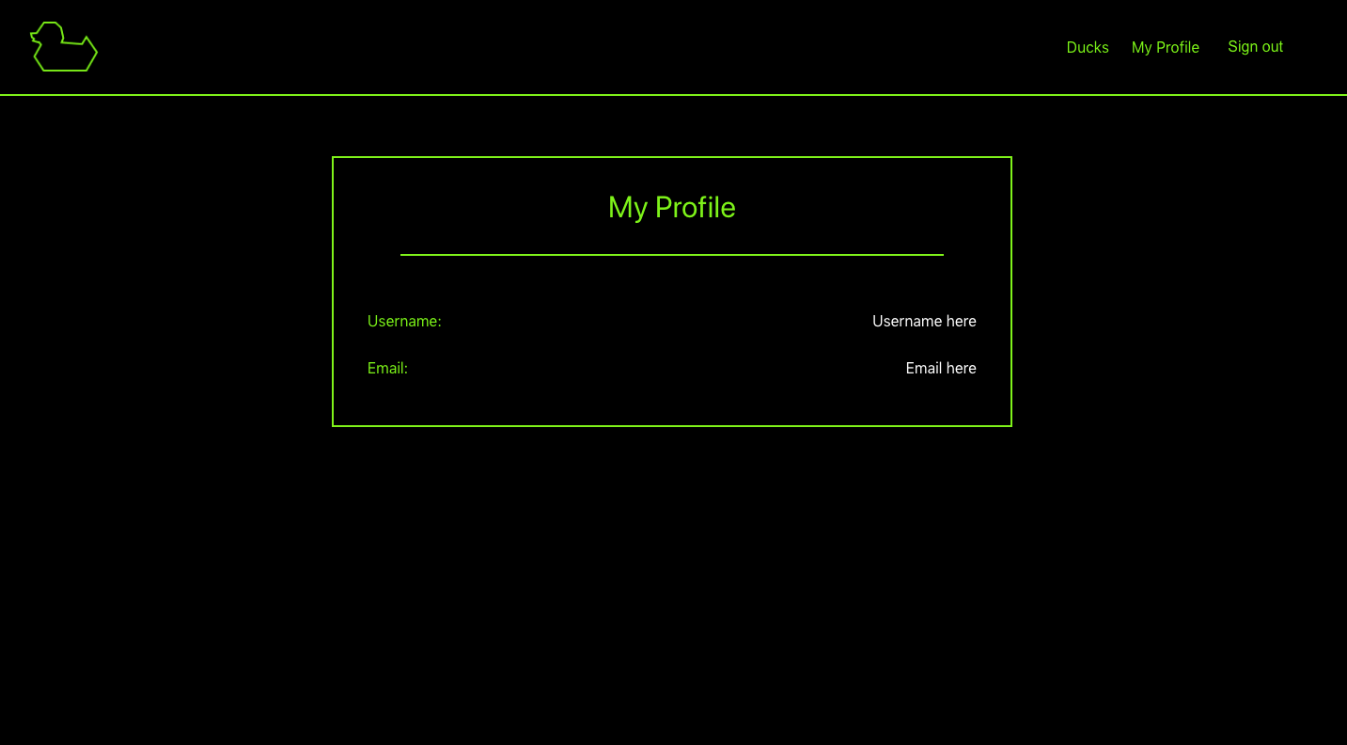
Наше дело — предупредить

Роут /ducks покажет важную информацию о криптоутках:



Море полезной информации

Когда пользователь обратится к /my-profile, он увидит экран с плейсхолдерами вместо реальных данных:



В ожидании данных

Что бы вы исправили в первую очередь?

# Переадресация

Мы бы начали с роута, которого нет внутри приложения. Когда пользователь переходит по нему, он видит чёрный экран. Чтобы такого не происходило, нужно перенаправить пользователя обратно в приложение в зависимости от того, авторизован он или нет. Сделаем так, чтобы авторизованные пользователи были перенаправлены в /ducks, а неавторизованные — на страницу входа в приложение.

Внутри App.js подготовим стейт-переменную loggedIn. Она будет содержать статус пользователя — вошёл он в систему или нет. Временно установим значение этой переменной false, но позже сделаем так, чтобы значение подставлялось динамически в зависимости от статуса пользователя:

Скопировать кодJSX

*// App.js*

constructor(props){

super(props);

this.state = {

loggedIn: false

}

}

[В уроке про программную навигацию](https://praktikum.yandex.ru/trainer/web/lesson/d656f2cd-73bb-4ae5-ba46-22d0dc42aa32) вы узнали о компоненте Redirect. Его можно использовать как последний компонент внутри Switch, чтобы перенаправить пользователя на определённый путь в зависимости от статуса его авторизации. Сначала мы импортируем компонент Redirect из react-router-dom, затем размещаем Redirect внутри компонента Route.

Код выглядит так:

Скопировать кодJSX

*// App.js*

<Route exact path="/">

{this.state.loggedIn ? <Redirect to="/ducks" /> : <Redirect to="/login" />}

</Route>

Мы проверяем тернарным оператором значение loggedIn. Если пользователь посетит / или любой другой маршрут, который не определён в приложении, неавторизованные пользователи будут перенаправлены на /login.

Одна проблема решена. Но пользователи по-прежнему могут посещать адреса /ducks и /my-profile, если будут вручную вводить их в адресную строку браузера — даже если значение loggedIn равно false. Исправим это в следующем уроке.

Пример

Сделайте так, чтобы пользователи не видели панель навигации NavBar, если значение loggedIn равно false. Внесите необходимые изменения в App.js. Маленькая подсказка: вам хватит одной строчки кода.

render(){

return (

<BrowserRouter>

<Header />

<main className="content">

{this.state.loggedIn && <NavBar />}

<Switch>

<Route exact path="/diary">

<Diary />

</Route>

# Защита маршрутов на фронте

В предыдущем уроке мы создали в компоненте App переменную состояния loggedIn. Её значение должно меняться с false на true, если пользователь авторизован. Эту логику опишем позже, а пока оставим значение false. Но есть проблема: любой пользователь может получить доступ к компонентам Ducks и MyProfile, просто набрав их URL в адресной строке браузера. То есть на стороне фронтенда пути остаются публичными — исправим это.

# Немного о компонентах более высокого порядка

Мы уже говорили [о компонентах более высокого порядка в «Реакте»](https://praktikum.yandex.ru/trainer/web/lesson/a5b21c62-7beb-43c6-a858-d2240cb6ae71). Создадим такой компонент, чтобы защитить роуты.

Компоненты Ducks и MyProfile отличаются друг от друга. Ducks отображает одну и ту же информацию для авторизованных и неавторизованных пользователей. А компонент MyProfile — личную информацию для конкретного авторизованного пользователя.

Но есть и схожая функциональность: оба компонента проверяют, авторизован ли пользователь. В зависимости от этого, они либо отрисовывают соответствующий компонент, либо переадресовывают пользователя на страницу входа /login.

Было бы избыточно добавлять логику проверки и переадресации в каждый компонент, поэтому расположим её в одном месте. Для этого понадобится компонент более высокого порядка.

НОС-компоненты позволяют добавить функциональность одному или нескольким компонентам. Если какие-то компоненты нуждаются в общей функциональности, можно обернуть их в HOC-компонент и описать в нём всё, что нужно. В больших проектах компоненты более высокого порядка позволяют лучше структурировать код и сделать его понятнее.

Для начала создадим HOC-компонент ProtectedRoute в директории /components:

Скопировать кодJSX

*// ProtectedRoute.js*

import React from "react";

import { Route, Redirect } from "react-router-dom";

*// этот компонент принимает другой компонент в качестве пропса*

*// он также может взять неограниченное число пропсов и передать их новому компоненту*

const ProtectedRoute = ({ component: Component, ...props }) => {

return (

<Route>

{() =>

props.loggedIn ? <Component {...props} /> : <Redirect to="./login" />

}

</Route>

);

};

export default ProtectedRoute;

Чтобы понять, что происходит внутри нового HOC-компонента, посмотрим на строку ({ component: Component, ...props }). Для получения компонентов, которые переданы через пропс component, HOC-компонент ProtectedRoute использует деструктуризацию. Пропсы пробрасываются в него также при помощи деструктуризации. Это значит, что данные могут поступать от родителя к HOC-компоненту, который потом используется повторно для создания любых новых компонентов.

HOC-компонент ProtectedRoute возвращает компонент Route. Если значение loggedIn — true, Route отрисует компонент, который передан HOC-компоненту как пропс, включая переданные пропсы. Если значение false — вернёт компонент Redirect и переадресует пользователя на страницу авторизации.

Реализуем компонент более высокого порядка в файле App.js:

Скопировать кодJSX

*// App.js*

import React from "react";

import { Route, Switch, Redirect } from "react-router-dom";

import Login from "./Login.js";

import Register from "./Register.js";

import Ducks from "./Ducks.js";

import MyProfile from "./MyProfile.js";

import ProtectedRoute from "./ProtectedRoute"; *// импортируем HOC*

import "./styles/App.css";

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

loggedIn: false,

};

}

render() {

return (

<Switch>

{/\* ниже разместим защищённые маршруты \*/}

{/\* и передадим несколько пропсов: loggedIn, path, component \*/}

<ProtectedRoute

path="/ducks"

loggedIn={this.state.loggedIn}

component={Ducks}

/>

<ProtectedRoute

path="/my-profile"

loggedIn={this.state.loggedIn}

component={MyProfile}

/>

<Route path="/login">

<div className="loginContainer">

<Login handleLogin={this.handleLogin} />

</div>

</Route>

<Route path="/register">

<div className="registerContainer">

<Register />

</div>

</Route>

<Route>

{this.state.loggedIn ? (

<Redirect to="/ducks" />

) : (

<Redirect to="/login" />

)}

</Route>

</Switch>

);

}

}

export default App;

Мы убрали компоненты Route для роутов /ducks и /my-profile, поскольку теперь они находятся в HOC-компоненте ProtectedRoute.

Новый компонент, который создаётся при помощи компонента ProtectedRoutes, имеет доступ ко всем пропсам, переданным на вход при его создании. В нашем случае это пропсы path и loggedIn. Мы можем передать их новому компоненту с помощью оператора spread: ...props. Иными словами, и у Ducks, и у MyProfile есть доступ ко всем пропсам родительского компонента.

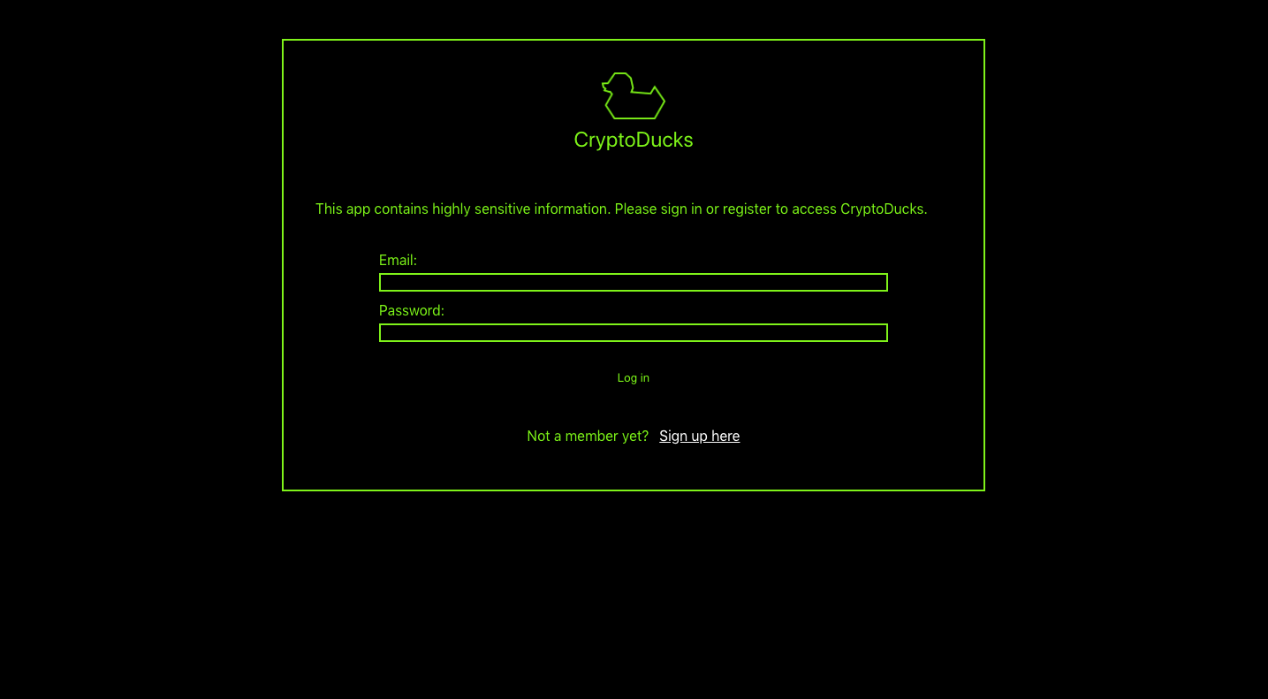
Теперь роуты /ducks/ и /my-profile защищены. Если пользователь не авторизуется в приложении и просто наберёт их адреса в браузере, то будет автоматически переадресован на страницу компонента Login.

# Посторонним вход воспрещён

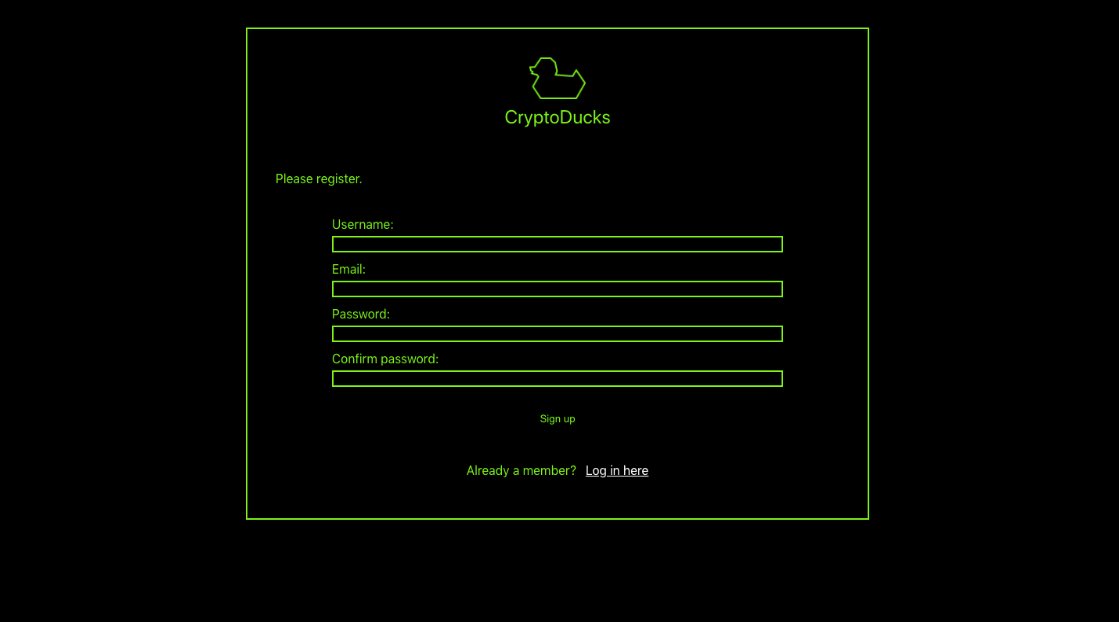
Роуты-то мы защитили, но образовалась новая проблема — пользователи не могут зарегистрироваться в приложении и, как следствие, войти в систему. Поэтому значение loggedIn всегда будет false. В следующем уроке наладим регистрацию.

# Регистрация

Если пользователь не авторизован, приложение переадресует его на страницу входа. Чтобы войти и посмотреть на криптоуток, нужно зарегистрироваться. На странице авторизации есть ссылка, которая ведёт на форму регистрации:



Когда пользователь переходит по ссылке, он попадёт с экрана компонента Login на экран Register:



Этот маршрут доступен для всех пользователей.

Разберём код компонента Register:

Скопировать кодJSX

*// Register.js*

import React, { Button } from 'react';

import { Link } from 'react-router-dom';

import Logo from './Logo.js';

import './styles/Register.css';

class Register extends React.Component {

constructor(props){

super(props);

this.state = {

username: '',

email: '',

password: '',

confirmPassword: ''

}

this.handleChange = this.handleChange.bind(this);

this.handleSubmit = this.handleSubmit.bind(this);

}

handleChange(e) {

const {name, value} = e.target;

this.setState({

[name]: value

});

}

handleSubmit(e){

e.preventDefault()

if (this.state.password === this.state.confirmPassword){

*// сюда добавим логику обработки формы регистрации*

}

}

render(){

return(

<div className="register">

<Logo title={'CryptoDucks'}/>

<p className="register\_\_welcome">

Please register.

</p>

<form onSubmit={this.handleSubmit} className="register\_\_form">

<label htmlFor="username">

Username:

</label>

<input id="username" name="username" type="text" value={this.state.username} onChange={this.handleChange} />

<label htmlFor="email">

Email:

</label>

<input id="email" name="email" type="email" value={this.state.email} onChange={this.handleChange} />

<label htmlFor="password">

Password:

</label>

<input id="password" name="password" type="password" value={this.state.password} onChange={this.handleChange} />

<label htmlFor="confirmPassword">

Confirm password:

</label>

<input id="confirmPassword" name="confirmPassword" type="password" value={this.state.confirmPassword} onChange={this.handleChange} />

<div className="register\_\_button-container">

<button type="submit" className="register\_\_link">Sign up</button>

</div>

</form>

<div className="register\_\_signin">

<p>Already a member?</p>

<Link to="login" className="register\_\_login-link">Log in here</Link>

</div>

</div>

)

}

}

export default Register;

Код может показаться монструозным, но на самом деле в нём нет ничего нового. Это уже знакомый вам [управляемый компонент](https://praktikum.yandex.ru/trainer/web/lesson/9f27d1c4-33fe-4e7a-b464-42104e8fe53e/task/93e982f6-b6a6-4e5b-93bb-db0c0c6a61ac/) для обработки инпутов формы, а также обработчики изменений и отправки формы.

Обратите внимание, что если оба пароля в форме совпадают, то внутри метода handleSumbit нужно что-то сделать с отправленными данными, например, отправить их на сервер.

# Создаём файл Auth

Придётся написать много кода для авторизации пользователей, поэтому поместим JavaScript-код в отдельный файл. Так мы избежим повторяющегося кода, если проект разрастётся. В директории /src создадим файл duckAuth.js:

Скопировать кодJSX

*// duckAuth.js*

export const BASE\_URL = 'https://api.nomoreparties.co';

export const register = (username, password, email) => {

return fetch(`${BASE\_URL}/auth/local/register`, {

method: 'POST',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify({username, password, email})

})

.then((response) => {

try {

if (response.status === 200){

return response.json();

}

} catch(e){

return (e)

}

})

.then((res) => {

return res;

})

.catch((err) => console.log(err));

};

На бэкенде приложений создают методы, которые добавляют пользователей в базу данных. В следующем спринте вы создадите их сами. Сейчас научимся вызывать их с фронтенда. Чтобы создать пользователя, вызовем функцию register в файле duckAuth.js.

Эта функция принимает три аргумента: username, password и email. С помощью fetch она создаёт POST-запрос.

После этого в первом обработчике then мы получим объект res, который содержит информацию об ответе и статус ответа. При успешном ответе вернётся статус 200, а второй обработчик then вернёт ещё один res с данными пользователя (которые он ввёл в форму регистрации), адресом запроса и уникальным подписанным JWT-токеном.

Теперь нужно импортировать файл duckAuth.js в Register.js: import \* as duckAuth from '../duckAuth.js';.

Следующий шаг — добавить новую функцию в обработчик handleSubmit:

Скопировать кодJSX

*// Register.js*

handleSubmit(e){

e.preventDefault()

if (this.state.password === this.state.confirmPassword){

const { username, password, email } = this.state;

duckAuth.register(username, password, email);

}

}

Для этого применим синтаксис деструктуризации к this.state. Затем передадим серверу username, password и email.

# HOC-компонент withRouter

Теперь пользователи смогут зарегистрироваться в приложении. Что дальше?

При успешной регистрации второй обработчик then вернёт токен JWT. Мы могли бы запустить этот код, сохранить токен в браузере и сразу залогинить пользователей. Но это слишком просто, поэтому переадресуем пользователя на страницу входа, чтобы он авторизовался самостоятельно. Кстати, теперь вы знаете, почему на некоторых сайтах приходится повторно вводить логин-пароль, хотя вы уже заполняли форму регистрации буквально мгновением ранее, — не написана логика.

Register — классовый компонент, поэтому использовать в нём хуки не получится. Чтобы получить доступ к объекту history и перенаправить пользователя на страницу авторизации, воспользуемся сторонним HOC-компонентом из библиотеки React Router — WithRouter.

Сперва импортируем его в Register.js. Для этого добавим в список импортов такую строку:

Скопировать кодJSX

*// Register.js*

import { Link, withRouter } from 'react-router-dom';

Всё, что остаётся сделать, — поправить директиву export в конце файла:

Скопировать кодJSX

// изменим последнюю строку в файле Register.js

export default withRouter(Register);

Теперь компонент Register точно обёрнут в HOC-компонент. Он сможет получить доступ к объекту history в props. Объект history позволяет использовать метод push. В качестве аргумента он принимает путь, на который мы перенаправляем пользователя. В нашем случае — '/login'.

Добавим ещё один then() в метод handleSubmit(). Пользователь должен быть переадресован, только если форма регистрации правильно заполнена и отправлена:

Скопировать кодJSX

*// Register.js*

handleSubmit(e){

e.preventDefault();

if (this.state.password === this.state.confirmPassword){

let { username, password, email } = this.state;

duckAuth.register(username, password, email).then((res) => {

if(res){

this.setState({

message: ''

}, () => {

this.props.history.push('/login');

})

} else {

this.setState({

message: 'Что-то пошло не так!'

})

}

});

}

}

Мы прописали в коде условие, что если statusCode равен 200, значит, форма отправлена успешно. Тогда при помощи обработчика then перенаправим пользователя на страницу авторизации.

У нас две новости: плохая и хорошая. Плохая — компонент Login всё ещё не работает. Хорошая — у зарегистрированных пользователей теперь есть JWT-токен, который привязан к их данным.

Вы научились пользоваться токенами, чтобы авторизовывать пользователей и сохранять вход в систему между сеансами. Но где хранить токен? В следующем уроке изучим методы хранения данных в браузере.

# Локальное хранилище

Чтобы пользователю не приходилось при каждом посещении сайта вводить пароль, нужно верифицировать его данные. Один из способов — проверить наличие валидного JWT-токена на стороне пользователя. В этом уроке напишем фронтенд для сохранения токена в браузере.

# Два браузерных хранилища

В зависимости от цели токен хранят в одном из двух браузерных хранилищ:

1. **Session storage** (с англ. сессионное хранилище). Данные отсюда удаляются в конце сессии, то есть когда пользователь закрывает окно браузера.

Сессионное хранилище можно сравнить с оперативной памятью: при выключении компьютера все несохранённые документы пропадут, потому что удалятся из оперативной памяти.

1. **Local storage** (с англ. локальное хранилище). Данные хранятся в нём до тех пор, пока их не удалят вручную.

Локальное хранилище работает как постоянная память: после перезагрузки компьютера все документы на жёстком диске остаются в сохранности.

Оба хранилища принадлежат домену. То есть к данным невозможно обратиться из хранилища yandex.ru, если вы находитесь на google.com.

В приложении Криптоутки будем пользоваться локальным хранилищем, ведь мы не хотим заставлять пользователей вводить логин и пароль при каждом посещении сайта.

Данные в local storage хранятся как JS-объекты, которые доступны глобально. У каждого объекта есть встроенные методы. Вот самые важные:

* setItem для сохранения данных,
* getItem для получения сохранённых данных,
* removeItem для удаления данных из памяти.

Скопировать кодJSX

*// сохраняем username*

localStorage.setItem('username', 'Стас Басов');

*// получаем username*

localStorage.getItem('username'); *// "Стас Басов"*

*// удаляем username*

localStorage.removeItem('username');

*// если ключа нет, вернётся null*

localStorage.getItem('username'); *// null*

setItem принимает два аргумента: ключ и его значение. Значением может быть только строка. Потому метод setItem приведёт любой аргумент к строке.

Для перевода объекта в строку пользуйтесь методом JSON.stringify. Если просто передать методу setItem объект, он будет приведён к строке встроенным методом toString, результатом работы которого будет [Object object] для любого объекта.

Метод JSON.stringify работает иначе: он создаст строку из объекта ровно в том виде, в каком он оформлен.

Чтобы превратить JSON-строку обратно в объект, вызовите метод JSON.parse:

Скопировать кодJSX

*// сохраняем username*

localStorage.setItem('user', JSON.stringify({

firstName: 'Стас',

lastName: 'Басов'

}));

*// достаём username*

JSON.parse(localStorage.getItem('user'));

*// {*

*// firstName: 'Стас',*

*// lastName: 'Басов'*

*// }*

# Сохраняем токен

Напишем код для сохранения токена в локальном хранилище. Напомним, что мы выбрали localStorage, чтобы пользователю не нужно было повторно логиниться после закрытия браузера:

Скопировать кодJSX

*// отправляем запрос на роут аутентификации*

fetсh('https://api.mywebsite.com/signin', {

method: 'POST',

body: JSON.stringify({

email: 'stasbasov@yandex.ru',

password: 'StasBasov1989'

})

})

.then(res => res.json())

.then((data) => {

*// сохраняем токен*

localStorage.setItem('token', data.token);

});

Теперь токен сохранён в локальном хранилище. Осталось настроить его отправку на сервер.

# Отправляем токен в запросах

Сделаем запросы авторизованными. Для этого внутри запроса нужно отправлять токен — по нему сервер понимает, есть ли у клиента права на выполнение действия.

Токен отправляют в заголовке authorization. Кроме самого токена заголовок должен содержать схему аутентификации — она сообщает серверу, что проверять наличие прав у пользователя нужно по токену. Имя такой схемы — Bearer. Есть и другие, но не будем в них углубляться.

В заголовок authorization нужно записать через пробел схему аутентификации и токен:

Скопировать кодJSX

*// отправляем запрос на роут аутентификации*

fecth('https://api.mywebsite.com/posts', {

method: 'GET',

headers: {

authorization: `Bearer ${localStorage.getItem('token')}`

}

});

Готово! В следующем уроке научимся реализовывать вход в систему.

# Вход в систему

Теперь вы знаете, как хранить токены внутри браузера, и можете проверить подлинность токена на сервере. В этом уроке рассмотрим, что происходит «под капотом», когда пользователь авторизуется в приложении. Кроме этого, научимся работать с localStorage, чтобы гарантировать, что браузер «запоминает» пользователя, вошедшего в систему.

Когда пользователь впервые входит в систему, нужно проверить введённые им учётные данные. Если они верны, в ответ с сервера отправляется JWT. Затем нужно сохранить этот JWT-токен в localStorage браузера.

При последующих входах пользователя в систему нужно получить токен из localStorage, а затем сделать запрос к серверу с этим токеном, чтобы проверить его достоверность. Если всё хорошо, пользователь получит доступ к своим данным, которые без токена были бы скрыты.

Продолжим совершенствовать приложение Криптоутки. Мы закончили на том, что после регистрации пользователи перенаправляются на экран входа в систему. Обратимся к компоненту Login:

Скопировать кодJSX

*// Login.js*

import React from 'react';

import { Link, withRouter } from 'react-router-dom';

import Logo from './Logo.js';

import \* as duckAuth from '../duckAuth.js';

import './styles/Login.css';

class Login extends React.Component {

constructor(props){

super(props);

this.state = {

username: '',

password: ''

}

this.handleChange = this.handleChange.bind(this);

this.handleSubmit = this.handleSubmit.bind(this);

}

handleChange(e) {

const {name, value} = e.target;

this.setState({

[name]: value

});

}

handleSubmit(e){

e.preventDefault()

if (!this.state.username || !this.state.password){

return;

}

*// здесь авторизуем пользователя*

*// далее проверяем токен*

*// наконец, перенаправляем пользователя на страницу `/ducks`*

}

render(){

return(

<div className="login">

<Logo title={'CryptoDucks'}/>

<p className="login\_\_welcome">

Это приложение содержит конфиденциальную информацию.

Пожалуйста, войдите или зарегистрируйтесь, чтобы получить доступ к CryptoDucks.

</p>

<form onSubmit={this.handleSubmit} className="login\_\_form">

<label htmlFor="username">

Логин:

</label>

<input id="username" required name="username" type="text" value={this.state.username} onChange={this.handleChange} />

<label htmlFor="password">

Пароль:

</label>

<input id="password" required name="password" type="password" value={this.state.password} onChange={this.handleChange} />

<div className="login\_\_button-container">

<button type="submit" className="login\_\_link">Войти</button>

</div>

</form>

<div className="login\_\_signup">

<p>Ещё не зарегистрированы?</p>

<Link to="/register" className="signup\_\_link">Зарегистрироваться</Link>

</div>

</div>

)

}

}

export default withRouter(Login);

Этот компонент аналогичен компоненту Register. Мы уже обернули компонент Login в HOC withRouter. Теперь применим history.push для отправки пользователей на страницу /ducks после авторизации в приложении.

Нам потребуется функция, которая будет проверять логин и пароль пользователя на соответствие какому-либо профилю, хранящемуся в базе данных. Вызывать её будем внутри handleSubmit.

Создадим такую функцию в duckAuth.js и назовём её authorize:

Скопировать кодJSX

*// duckAuth.js*

export const authorize = (identifier, password) => {

return fetch(`${BASE\_URL}/auth/local`, {

method: 'POST',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify({identifier, password})

})

.then((response => response.json()))

.then((data) => {

if (data.jwt){

localStorage.setItem('jwt', data.jwt);

return data;

}

})

.catch(err => console.log(err))

};

Сейчас код выглядит довольно массивно. Разобьём его на части и разберёмся, что там происходит. Первая часть должна показаться вам знакомой:

Скопировать кодJSX

*// duckAuth.js*

return fetch(`${BASE\_URL}/auth/local`, {

method: 'POST',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify({identifier, password})

})

.then((response => response.json()))

Мы делаем API запрос с помощью fetch. В теле запроса отправляем учётные данные пользователя: identifier — логин и password — пароль пользователя. Этот запрос возвращает промис с response. Затем внутри первого обработчика then вызываем response.json. Этот then также возвращает промис. Вы можете получить доступ к data внутри второго обработчика then.

Изучите этот код — именно тут происходит магия:

Скопировать кодJSX

*// duckAuth.js*

.then((data) => {

if (data.jwt){

localStorage.setItem('jwt', data.jwt);

return data;

} else {

return;

}

})

Проверяем, есть ли свойство jwt в объекте data, который вернул сервер. Если пользователь нашёлся и его учётные данные действительны, у нас есть доступ к токену. Сохраним токен в localStorage пользователя, чтобы он был там всякий раз, когда пользователь вернётся в приложение. Наконец, вернём объект с данными пользователя. В противном случае, если свойство jwt отсутствует, ничего не вернётся и токена не будет.

Теперь нужно настроить это внутри Login.js:

Скопировать кодJSX

*// Login.js*

handleSubmit(e){

e.preventDefault();

if (!this.state.username || !this.state.password){

return;

}

duckAuth.authorize(this.state.username, this.state.password)

.then((data) => {

if (data.jwt){

this.setState({username: '', password: ''} ,() => {

this.props.handleLogin();

this.props.history.push('/ducks');

})

}

})

.catch(err => console.log(err)); *// запускается, если пользователь не найден*

}

После запуска authorize, если всё получилось, возвращается объект с JWT-токеном. После этого нужно очистить стейт и перенаправить пользователя на страницу /ducks с помощью метода history.push.

Обратите внимание, что мы не изменили значение loggedIn на true внутри App.js, поэтому пользователи будут немедленно перенаправляться обратно на страницу авторизации. Ведь ProtectedRoute будет отображать маршрут /ducks только в том случае, если loggedIn равно true.

Чтобы поменять значение loggedIn, создадим метод внутри App.js, поскольку loggedIn находится в state внутри компонента App:

Скопировать кодJSX

*// App.js*

constructor(props){

super(props);

this.state = {

loggedIn: false

}

this.handleLogin = this.handleLogin.bind(this);

}

componentDidMount() {

*// позже здесь тоже нужно будет проверить токен пользователя!*

};

handleLogin(e){

e.preventDefault();

this.setState({

loggedIn: true

})

}

Передадим Login этот новый метод как пропс:

Скопировать кодJSX

<Route path="/login">

<div className="loginContainer">

<Login handleLogin={this.handleLogin} />

</div>

</Route>

Затем вызовем его в Login.js:

Скопировать кодJSX

handleSubmit(e) {

e.preventDefault();

if (!this.state.username || !this.state.password) {

return;

}

duckAuth.authorize(this.state.username, this.state.password)

.then((data) => {

if (data.jwt) {

this.setState({

username: '',

password: ''

}, () => {

this.props.handleLogin(); *// обновляем стейт внутри App.js*

this.props.history.push('/ducks'); *// и переадресуем пользователя!*

})

}

})

.catch(err => console.log(err)); *// запускается, если пользователь не найден*

}

Фух! Когда пользователь входит в систему, из Login.js вызывается метод handleLogin, который изменяет состояние loggedIn внутри App.js на true. Теперь, когда пользователь входит в систему, он сразу попадает на страницу /ducks.

Но не всё так гладко. Если пользователь покидает приложение, а потом возвращается, ему нужно снова войти в систему. Мы знаем, как это исправить. Для этого мы сохранили JWT-токен пользователя внутри localStorage. В следующем уроке вы научитесь делать так, чтобы браузер мог его использовать и «запоминать», что пользователь уже авторизован.

# Проверка токенов

В этом уроке научимся проверять токены авторизованных пользователей, которые вернулись в приложение. Это нужно, чтобы им не пришлось снова вводить логин и пароль.

Сначала напишем код, который проверит, есть ли токен в localStorage. Если он там, то можно сделать запрос API с токеном, чтобы убедиться, что он действительно принадлежит пользователю. Затем изменим значение loggedIn внутри App на true и перенаправим пользователя в /ducks. Для этого импортируем HOC-компонент withRouter() из react-router-dom.

Теперь нужно решить, в каком файле написать код проверки токена. В нашем случае подойдёт метод жизненного цикла компонента componentDidMount().

Создадим внутри App.js метод, который будет вызываться в componentDidMount() при монтировании компонента. Назовём этот метод tokenCheck(). Не забывайте, что строку this.tokenCheck = this.tokenCheck.bind(this); нужно поместить в функцию constructor():

Скопировать кодJSX

*// В App.js*

constructor(props){

super(props);

this.state = {

loggedIn: false

}

this.tokenCheck = this.tokenCheck.bind(this);

this.handleLogin = this.handleLogin.bind(this);

}

componentDidMount() {

*// настало время проверить токен*

this.tokenCheck();

};

handleLogin(){

this.setState({

loggedIn: true

})

}

tokenCheck () {

*// если у пользователя есть токен в localStorage,*

*// эта функция проверит, действующий он или нет*

if (localStorage.getItem('jwt')){

const jwt = localStorage.getItem('jwt');

*// здесь будем проверять токен*

}

}

Этот код проверяет наличие у пользователя токена. Если он есть в localStorage — берём токен оттуда.

Теперь, чтобы проверить токен и получить данные пользователя, создадим ещё один метод авторизации внутри duckAuth.js:

Скопировать кодJSX

export const getContent = (token) => {

return fetch(`${BASE\_URL}/users/me`, {

method: 'GET',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

}

})

.then(res => res.json())

.then(data => data)

}

Функция getContent() принимает в качестве параметра один аргумент — JWT. Он будет отправлен на сервер (API) по маршруту /users/me, и, если токен действителен, вернёт ответ с информацией о пользователе.

Импортируем новую функцию в App.js, а затем вызовем её внутри метода tokenCheck():

Скопировать кодJSX

tokenCheck () {

*// если у пользователя есть токен в localStorage,*

*// эта функция проверит валидность токена*

const jwt = localStorage.getItem('jwt');

if (jwt){

*// проверим токен*

duckAuth.getContent(jwt).then((res) => {

if (res){

*// авторизуем пользователя*

this.setState({

loggedIn: true,

}, () => {

*// обернём App.js в withRouter*

*// так, что теперь есть доступ к этому методу*

this.props.history.push("/ducks");

});

}

});

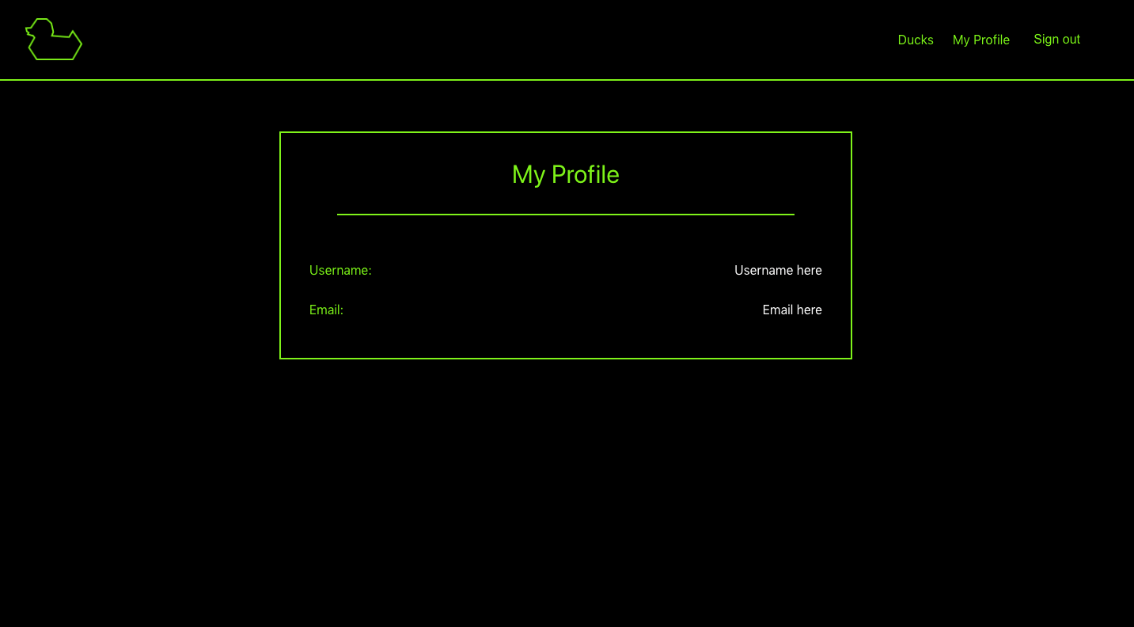
}

}

Теперь, когда пользователь вернётся в приложение, всё будет отображаться так, будто он и не уходил.

# Работа с данными

Когда пользователь будет заходить на эту страницу, он по-прежнему будет видеть стандартные плейсхолдеры:



Это можно исправить. Вызов duckAuth.getContent() внутри метода tokenCheck() возвращает некоторые полезные данные. Мы можем использовать их для заполнения этих полей:

Скопировать кодJSX

*// В App.js*

tokenCheck () {

*// если у пользователя есть токен в localStorage,*

*// эта функция проверит валидность токена*

const jwt = localStorage.getItem('jwt');

if (jwt){

*// проверим токен*

duckAuth.getContent(jwt).then((res) => {

if (res){

*// здесь можем получить данные пользователя!*

const userData = {

username: res.username,

email: res.email

}

*// поместим их в стейт внутри App.js*

this.setState({

loggedIn: true,

userData

}, () => {

this.props.history.push("/ducks");

});

}

});

}

}

Мы нашли пользовательские данные и обновили state с помощью userData.

Теперь внутри App.js нужно передать userData компоненту ProtectedComponent, который отвечает за отрисовку MyProfile внутри Switch:

Скопировать кодJSX

*// защищённые роуты должны выглядеть так*

*// обратите внимание, что мы передаём userData второму роуту*

<ProtectedRoute path="/ducks" loggedIn={this.state.loggedIn} component={Ducks} />

<ProtectedRoute path="/my-profile" loggedIn={this.state.loggedIn} userData={this.state.userData} component={MyProfile} />

Данные будут переданы через ProtectedRoute в MyProfile, и там у нас будет доступ к ним:

Скопировать кодJSX

*// MyProfile.js*

import React from 'react';

import NavBar from './NavBar.js';

import './styles/MyProfile.css';

function MyProfile (props) {

let { username, email} = props.userData;

return (

<>

<NavBar />

<div className="my-profile">

<div className="my-profile\_\_container">

<div className="my-profile\_\_header">

<p>Мой профиль</p>

<hr className="my-profile\_\_rule"/>

</div>

<div className="my-profile\_\_info">

<div className="my-profile\_\_user">

<p className="my-profile\_\_key">Логин:</p>

<p className="my-profile\_\_value">{username}</p>

</div>

<div className="my-profile\_\_user">

<p className="my-profile\_\_key">Email:</p>

<p className="my-profile\_\_value">{email}</p>

</div>

</div>

</div>

</div>

</>

)

}

export default MyProfile;

Авторизованные пользователи теперь могут часами любоваться своими данными.

Но что если по какой-то странной причине пользователь захочет выйти из приложения Криптоутки? Не будем его удерживать силой и в следующем уроке настроим функцию выхода из системы.

**Выход из системы**

В NavBar расположена кнопка «Выйти», которая ещё не работает. Разберёмся, как это исправить.

Сам компонент NavBar используется внутри компонентов MyProfile и Ducks:

Скопировать кодJSX

import React from 'react';

import { Link } from 'react-router-dom';

import Logo from './Logo.js';

import './styles/NavBar.css';

function NavBar () {

return (

<div className="navbar">

<div className="navbar\_\_logo">

<Logo/>

</div>

<ul className="navbar\_\_nav">

<li><Link to="ducks" className="navbar\_\_link">Утки</Link></li>

<li><Link to="my-profile" className="navbar\_\_link">Мой профиль</Link></li>

<li><button className="navbar\_\_link navbar\_\_button">Выйти</button></li>

</ul>

</div>

)

}

export default NavBar;

Для реализации выхода из системы нужно удалить JWT-токен из localStorage и переадресовать пользователя на страницу /login. Поскольку NavBar — функциональный компонент, воспользуемся «Реакт-хуком» useHistory:

Скопировать кодJSX

import React from 'react';

import { Link, useHistory } from 'react-router-dom';

import Logo from './Logo.js';

import './styles/NavBar.css';

function NavBar () {

const history = useHistory();

function signOut(){

localStorage.removeItem('jwt');

history.push('/login');

}

return (

<div className="navbar">

<div className="navbar\_\_logo">

<Logo/>

</div>

<ul className="navbar\_\_nav">

<li><Link to="ducks" className="navbar\_\_link">Утки</Link></li>

<li><Link to="my-profile" className="navbar\_\_link">Мой профиль</Link></li>

<li><button onClick={signOut} className="navbar\_\_link navbar\_\_button">Выйти</button></li>

</ul>

</div>

)

}

export default NavBar;

Мы добавили метод signOut. Он удаляет JWT из localStorage и затем использует метод history.push, чтобы направить пользователя обратно к роуту /login.

Мы также добавили обработчик onClick, который будет вызывать этот метод всякий раз, когда пользователь нажимает на элемент <button> внутри компонента NavBar.

Теперь всё работает как надо. Вы получили доступ к тайным знаниям и методам работы, которые известны немногим. И речь не об аутентификации... А теперь главный вопрос: готовы ли вы узнать всю правду о криптоутках?

**Заключение**

В этой теме вы узнали, что означают три этих длинных слова: аутентификация, авторизация и идентификация. Теперь вы можете сказать, в чём различия между процессами, которые скрываются за этими сложными словами. Вы сделали первые шаги на пути к облегчению жизни пользователя — узнали что такое JSON веб-токены.

Вы научились реализовывать регистрацию и авторизацию в приложении, переадресовывать пользователя на нужную страницу и создавать приватные маршруты. Также вы узнали, как написать код входа и выхода из системы и проверить токен в браузере пользователя.

Ответственный разработчик следит за тем, чтобы данные пользователя были под надёжной защитой. Недостаточно сделать фронтенд «просто чтобы хоть как-то работало», — нужно проектировать веб-приложение защищённым.

Теперь пора переходить к заключительной работе над проектом в курсе по «Реакту» — созданию авторизации и регистрации в вашем приложении. Удачи!