**Идентификация, аутентификация и авторизация**

Перед тем как сесть на самолёт или поезд, вам нужно предъявить билет. Эта простая процедура не даёт другим людям занять место, которое вы забронировали. Также она помогает защитить пассажиров от потенциальных преступников на этапе посадки или покупки билета.

То же самое происходит и в интернете. Мы выкладываем фотографии, комментируем их, ставим лайки. А если кто-то оставит обидный комментарий, то решить проблему можно по-взрослому — сообщением «Слабо приехать ко мне на район и лично сказать?» Ведь сервер может отличить одного пользователя от другого, потому что за каждым закреплён аккаунт.

Пользователи обладают разными правами: среди них могут быть модераторы, администраторы, работники службы поддержки. К примеру, у сотрудника интернет-магазина должна быть возможность отслеживать заказ по номеру: так он сможет проконсультировать покупателя. Пользователям, которые не имеют к этому заказу отношения, такая возможность ни к чему.

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

**Идентификация**

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

Например, при посадке в поезд контролёр просит Василия предъявить документы. Василий очень спешит и называет просто своё имя. Произошла идентификация — теперь контролёр знает имя странного пассажира.

**Аутентификация**

Аутентифицировать — значит доказать, что идентифицированный — действительно тот, кем назвался. Идентификация — часть процесса аутентификации.

Контролёру недостаточно имени, и он просит Василия предъявить паспорт. В этот момент контролёр аутентифицирует пассажира. Паспорт — доказательство, что Василий — это именно Василий.

В интернете аутентификация часто сводится к вводу имени пользователя и пароля. Но бывают и другие способы: код из SMS, ответ на секретный вопрос, отпечаток пальца или снимок лица.

**Авторизация**

Авторизация происходит после аутентификации. Это процесс, при котором система решает, какие права можно выдать пользователю.

Контролёру, помимо паспорта Василия, требуется и билет. Билет — своего рода авторизация. В соответствии с ним контролёр определяет, какое место займёт пассажир.

В интернете после авторизации в соцсети пользователи получают права на редактирование собственных публикаций и отправку сообщений от своего имени.

**Итого**

Чтобы эта тема вызвала меньше затруднений, важно чётко понять, чем отличаются понятия идентификации, аутентификации и авторизации. Дальше поговорим о том, из каких процессов состоит авторизация в клиент-серверном взаимодействии и что такое токены.

# Как работает защита на клиенте и сервере. Токены

Вы узнали, что такое авторизация и аутентификация, и чем они отличаются. Теперь перенесёмся в плоскость фронтенд- и бэкенд-части приложения и по шагам разберём, как устроены все эти процессы.

## Регистрация

Чтобы начать взаимодействие с большинством сервисов, нужно в них зарегистрироваться. Для этого пользователь должен предоставить серверу свои данные в клиентской части приложения, к примеру логин и пароль. Их потребуется вводить каждый раз при аутентификации в приложении. Сервер, в свою очередь, будет хранить не только данные, предоставленные пользователем, но и привязывать к ним другие записи: ленту новостей или диалоги в социальной сети.

Разберём самый простой процесс регистрации на шаги:

1. Пользователь пытается попасть на роут приложения. Здесь происходит первое знакомство и попытка опознать пользователя. Если пользователь не опознан, его переводят на страницу регистрации.
2. Пользователь вводит данные, например, логин и пароль. Подтверждает регистрацию и данные отправляются на сервер.
3. Сервер проверяет корректность присланных данных. Если с ними всё хорошо, создаёт для пользователя запись в базе данных с логином, зашифрованным паролем и идентификатором.
4. Сервер возвращает в клиентскую часть приложения сообщение о результате регистрации или данные о зарегистрированном пользователе.

Представьте, что покупаете билет на поезд. Сначала вы вводите паспортные данные, затем — оплачиваете покупку. Вам на почту приходит электронный билет, информация о котором есть только у клиента () и сервера.

## Авторизация

После регистрации пользователь проходит процесс авторизации. Он устроен так:

1. Пользователь предоставляет данные для авторизации — логин и пароль. Нажимает на кнопку «Войти», и данные отправляются на сервер.
2. Сервер проверяет данные. Если с ними всё хорошо, ищет пользователя.
3. Если пользователь найден в базе данных, сервер возвращает его приложению и как бы говорит: «Всё ок, этот пользователь может пройти дальше».
4. Клиентская часть приложения на React сохраняет данные в хранилище.
5. Пользователь получает доступ к тем страницам приложения, которые доступны для текущего уровня прав.

Например, чтобы сесть на поезд, нужно показать билет. По такой логике при каждой посадке на поезд пассажир должен предъявлять билет, даже если он едет уже сутки и вышел на остановке за шоколадкой. Так и с приложением: при каждом повторном открытии сайта пользователю нужно заново проходить процесс авторизации.

Но наверняка вы замечали, что при посещении одной социальной сети вам не приходится снова 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. Токен можно сравнить с временным пропуском на закрытую парковку. Как только вы подъезжаете к шлагбауму — процесс сканирования пропуска уже запускается. И если с пропуском всё хорошо, шлагбаум сразу же открывается. Вам, как клиенту закрытого паркинга, не приходится делать ничего лишнего — все процессы происходят автоматически.

Разберём алгоритм авторизации пользователя с использованием токена.

## Авторизация с применением токена

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 браузер 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.

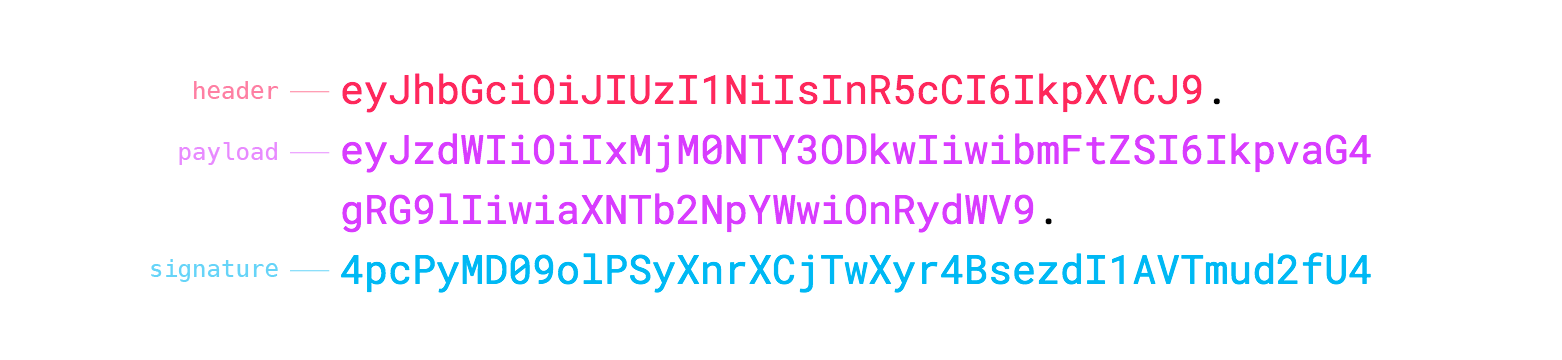
## Структура токена

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. «шапка») содержит 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 кодJSX

{

"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 кодJSX

{

"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 структуре токена и алгоритме работы с ним. Пока этого достаточно. В следующих уроках разберём, как сохранять данные о пользователе в глобальное состояние, взаимодействовать с токеном и сохранять его в браузере.

# Авторизация доступа с помощью сессий

Из предыдущих уроков вы узнали, как работает авторизация с помощью токенов. Это наиболее современный способ, который часто применяется в проектах. Но авторизация с помощью токенов — не единственный механизм: некоторые компании по-прежнему используют немного устаревшую авторизацию через сессии. В этом уроке подробнее расскажем про такой механизм, чтобы вы не растерялись, если столкнётесь с ним на практике.

## Что такое сессия

Во времена, когда клиентские приложения представляли собой статичный HTML и формы, отправка которых перезагружала страницу, пользователи уже хотели, чтобы приложения их узнавали. Для этого использовали сессии. Сессия — файл, или in-memory база данных, которая хранит уникальный идентификатор пользователя и опционально дополнительную информацию (email, последний визит и другое). Файл с сессиями хранится на сервере и очищается по правилам, установленным на сервере, например раз в год или раз в 15 минут. Частота очистки сессий зависит от требований к безопасности. К примеру, в банковских приложениях сессии редко «живут» дольше 15 минут.

## Как происходит авторизация и идентификация

Когда сервер получает от клиента корректные авторизационные данные, например логин и пароль, он создаёт запись в сессии. В ответ на запрос авторизации сервер устанавливает специальный заголовок — Set-Cookie. Он несёт в себе уникальный идентификатор, который обычно состоит из 32 символов, например ac4f4a45bdc853434d95dcaffb1c1824. Благодаря этому заголовку в клиентский браузер помещается кука. Кука, в свою очередь, автоматически прикрепляется к заголовкам запроса при каждом обращении к серверу. Сервер же проверяет наличие сессии с указанным идентификатором и если находит, считает пользователя авторизованным.

Такой тип авторизации подойдёт для приложения, которое работает только в браузере, с минимальной микросервисной архитектурой.

Да. В современном мире микросервисы победили монолитные системы. Это значит, что пользователь видит в браузере единое приложение, которое состоит из чата, ленты новостей, поиска и плейлиста. Со стороны бэкенда это — целый зоопарк технологий и сервисов. Из-за того, что сессии в простом виде хранятся внутри одного из сервисов, другим сервисам, скажем, ленте новостей, не удаётся идентифицировать пользователя. Эти сервисы даже физически расположены на разных континентах. Конечно, бэкенд-разработчики находят способы сделать сессию из главного сервиса доступной для всех остальных, но это накладывает неудобства. Кроме того, из-за использования кук, API сервиса, построенного на сессиях, может обслуживать только браузеры, тогда как для мобильных клиентов требуется дополнительная «прослойка».

В отличие от авторизации с помощью сессий токены не требуют дополнительного хранения на сервере в файле или базе данных и вычисляются мгновенно благодаря криптографии. К тому же API с авторизацией через токены может работать как с браузером, так и с мобильными приложениями.

**Сохранение данных о пользователе**

Скопировать кодJSX

const fakeAuth = {

isAuthenticated: false,

signIn(cb) {

fakeAuth.isAuthenticated = true;

setTimeout(cb, 100); *// fake async*

},

signOut(cb) {

fakeAuth.isAuthenticated = false;

setTimeout(cb, 100);

}

};

Вызовы методов signIn и signOut влияют на значение ключа isAuthenticated.

В этом примере воспользуемся Context API, который входит в пакет библиотеки React. И создадим контекст:

Скопировать кодJSX

const AuthContext = createContext(undefined);

export function ProvideAuth({ children }) {

const auth = useProvideAuth();

return <AuthContext.Provider value={auth}>{children}</AuthContext.Provider>;

}

Затем напишем хук useProvideAuth, внутри которого и будет происходить вся авторизация и деавторизация:

Скопировать кодJSX

export function useProvideAuth() {

const [user, setUser] = useState(null);

const signIn = cb => {

return fakeAuth.signIn(() => {

*// Временные данные, которые будут доступны приложению*

setUser({ id: 1337, name: 'David' });

cb();

});

};

const signOut = cb => {

return fakeAuth.signOut(() => {

setUser(null);

cb();

});

};

return {

user,

signIn,

signOut

};

}

В этом хуке мы определяем одноимённые функции signIn и signOut, а затем возвращаем их вместе с данными о пользователе. Для удобства использования этих данных в приложении добавим хук, который будет использовать контекст AuthContext:

Скопировать кодJSX

export function useAuth() {

return useContext(AuthContext);

}

Готово, осталось обернуть всё приложение провайдером авторизации.

**Использование состояния для авторизации в приложении**

Обернём наше приложение провайдером авторизации ProvideAuth:

Скопировать кодJSX

import './App.css';

import { BrowserRouter as Router, Switch, Route, Redirect } from 'react-router-dom';

import { LoginPage, ListPage, NotFound404 } from './pages';

import { ProvideAuth } from './services/auth';

export default function App() {

return (

<ProvideAuth>

<Router>

<Switch>

<Route path="/login">

<LoginPage />

</Route>

<Route path="/list">

<ListPage />

</Route>

<Route path="/" exact={true}>

<Redirect to="/list" />

</Route>

<Route>

<NotFound404 />

</Route>

</Switch>

</Router>

</ProvideAuth>

);

}

Теперь нам осталось воспользоваться удобным хуком useAuth в каком-нибудь компоненте:

Скопировать кодJSX

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

import { useAuth } from '../services/auth';

import React, { useCallback, useState } from 'react';

import styles from './login.module.css';

import { Button } from '../components/button';

import { Input } from '../components/input';

import { PasswordInput } from '../components/password-input';

export function LoginPage() {

let history = useHistory();

*// Используем хук, в котором доступны все нужные методы и данные о пользователе*

let auth = useAuth();

const [form, setValue] = useState({ email: '', password: '' });

const onChange = e => {

setValue({ ...form, [e.target.name]: e.target.value });

};

let login = useCallback(

e => {

e.preventDefault();

auth.signIn(() => {

history.replace({ pathname: '/list' });

});

},

[auth, history]

);

if (auth.user) {

return (

<Redirect

to={{

pathname: '/list'

}}

/>

);

}

return (

<div className={styles.wrapper}>

<form className={styles.form}>

<h1 className={styles.heading}>Вход</h1>

<Input placeholder="Email" value={form.email} name="Логин" onChange={onChange} />

<PasswordInput

placeholder="Пароль"

value={form.password}

name="password"

onChange={onChange}

/>

<Button onClick={login} primary={true}>

Войти

</Button>

</form>

</div>

);

}

Всё, приложение с авторизацией готово! В следующих уроках мы подключим реальный сервер для авторизации, а пока можете посмотреть, как это работает (авторизацию можно пропустить, не вводя ничего в поля):

В этом уроке мы использовали контекст, но если применить Redux в качестве глобального состояния, алгоритм будет такой же:

1. Отправляем экшен, который выполняет запрос на авторизацию.
2. После успешного выполнения запроса на авторизацию записываем данные в Redux.
3. С помощью useSelector получаем доступ к данным о текущем пользователе.
4. PROFIT!

На этом примере вы могли убедиться, что хранение данных о текущем пользователе ничем не отличается от хранения данных, полученных с сервера, для других частей приложения. Единственный минус — в текущей реализации пользователь вынужден вводить логин и пароль каждый раз при открытии приложения. В следующих уроках разберёмся, как это исправить.

**Хранение токенов. Cookie в веб-приложениях**

Чтобы пользователю не приходилось вводить логин и пароль при каждом посещении сайта, нужно верифицировать его данные. Один из способов — проверить наличие валидного JWT-токена на стороне пользователя. В этом уроке разберёмся, как сохранять токен в браузере.

**Браузерные хранилища**

Существуют несколько способов хранения токенов. Два самых распространённых — cookie и localStorage. В зависимости от особенностей реализации механизма авторизации на сервере, токен хранят в одном из этих хранилищ:

1. **Сookie** (с англ. печенье). Так называют небольшие данные (не более 4 Кбайт), которые хранятся в браузере. Обычно сервер устанавливает куки в браузер с помощью специального заголовка Set-Cookie. Все куки могут автоматически «прикрепляться» к запросам на сервер.

Куки бывают разными — одни работают до закрытия вкладки браузера, другие «живут» по несколько лет. Кроме того, существуют куки, доступ к которым невозможно получить из JavaScript.

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 данные остаются сохранёнными.

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

Для хранения токена в приложении с чатом воспользуемся куками, ведь мы не хотим заставлять пользователей вводить логин и пароль при каждом посещении сайта. Данные в куках хранятся в виде строки и доступны через document.cookie. Выглядят куки так:

Скопировать кодJSX

"user=Alex; someValue=123; lastVisit=1608393401"

Для работы с куками обычно пишут вспомогательные функции или пользуются библиотеками. Так можно установить куку:

Скопировать кодJSX

document.cookie = "user=Alex; path=/; max-age=300"

Такая кука будет доступна в течении 5 минут на всех страницах сайта.

**Сохраняем токен**

Напишем код для получения токена:

Скопировать кодJSX

*// Отправляем данные формы на сервер для авторизации*

export const loginRequest = async form => {

return await fetch('https://cosmic.nomoreparties.space/login', {

method: 'POST',

mode: 'cors',

cache: 'no-cache',

credentials: 'same-origin',

headers: {

'Content-Type': 'application/json'

},

redirect: 'follow',

referrerPolicy: 'no-referrer',

body: JSON.stringify(form)

});

};

При успешной авторизации сервер возвращает данные о пользователе:

Скопировать кодJSX

{

"success": true,

"user": {

"\_id": 1337,

"password": "ToPsEcReT007421337",

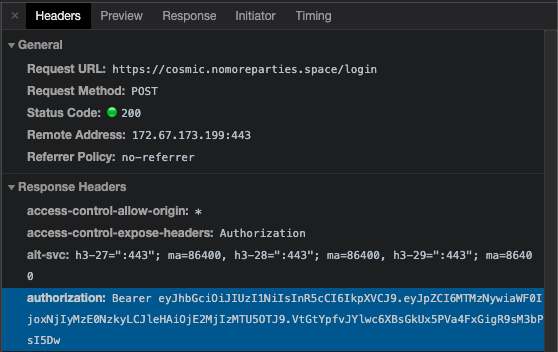
"email": "spacedancer1337@nomoreparties.space",

"name": "Mocktantin Testov"

}

}

И передаёт токен в authorization заголовке:



Теперь сохраним этот токен в куку. Для удобства воспользуемся функцией setCookie:

Скопировать кодJAVASCRIPT

export function setCookie(name, value, props) {

props = props || {};

let exp = props.expires;

if (typeof exp == 'number' && exp) {

const d = new Date();

d.setTime(d.getTime() + exp \* 1000);

exp = props.expires = d;

}

if (exp && exp.toUTCString) {

props.expires = exp.toUTCString();

}

value = encodeURIComponent(value);

let updatedCookie = name + '=' + value;

for (const propName in props) {

updatedCookie += '; ' + propName;

const propValue = props[propName];

if (propValue !== true) {

updatedCookie += '=' + propValue;

}

}

document.cookie = updatedCookie;

}

Эта функция нормализует работу с временем жизни куки и обрабатывает те случаи, когда время жизни куки не было передано. Сохраним токен в куку внутри функции авторизации:

Скопировать кодJSX

const signIn = async form => {

const data = await loginRequest(form)

.then(res => {

let authToken;

*// Ищем интересующий нас заголовок*

res.headers.forEach(header => {

if (header.indexOf('Bearer') === 0) {

*// Отделяем схему авторизации от "полезной нагрузки токена",*

*// Стараемся экономить память в куках (доступно 4кб)*

authToken = header.split('Bearer ')[1];

}

});

if (authToken) {

*// Сохраняем токен в куку token*

setCookie('token', authToken);

}

return res.json();

})

.then(data => data);

if (data.success) {

*// Сохраняем пользователя в состояние приложения и нормализуем поле id (\_id => id)*

setUser({ ...data.user, id: data.user.\_id });

}

};

При сохранении куки мы не указываем дополнительных параметров. Такая кука будет работать до завершения работы браузера. Если закрыть браузер и потом опять открыть — придётся заново вводить логин и пароль. Осталось проверить работоспособность токена. Для этого выполним запрос на защищённый роут сервера.

**Отправляем токен в запросах**

Чтобы авторизовывать запросы, внутри них нужно отправлять токен — по нему сервер понимает, есть ли у клиента права на выполнение действия или получение данных.

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. Вы уже видели это в приходящем заголовке.

Для удобного доступа к куке, нам потребуется ещё одна функция:

Скопировать кодJSX

export function getCookie(name) {

const matches = document.cookie.match(

new RegExp('(?:^|; )' + name.replace(/([\.$?\*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + '=([^;]\*)')

);

return matches ? decodeURIComponent(matches[1]) : undefined;

}

В этой функции мы можем получить интересующую нас куку с помощью [регулярного выражения](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp). Теперь передадим в заголовок authorization схему аутентификации и токен из куки:

Скопировать кодJSX

*// отправляем запрос на роут аутентификации*

export const getChatsRequest = async () =>

await fetch('https://cosmic.nomoreparties.space/api/chat', {

method: 'GET',

mode: 'cors',

cache: 'no-cache',

credentials: 'same-origin',

headers: {

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

*// Отправляем токен и схему авторизации в заголовке при запросе данных*

Authorization: 'Bearer ' + getCookie('token')

},

redirect: 'follow',

referrerPolicy: 'no-referrer'

});

Если просто перейти по https://cosmic.nomoreparties.space/api/chat, эндпоинт не отдаст данные о чатах, так как не сможет идентифицировать пользователя, но с токеном он возвращает нужные данные.

В этом уроке вы узнали как сохранять токен в браузере для того, чтобы пользователям не приходилось вводить логин и пароль каждый раз при открытии сайта или приложения. Мы разобрали один из возможных сценариев авторизации, а их вариативность зависит от реализации на сервере. При использовании токена для авторизации, его следует передавать в заголовках запроса.

В следующем уроке расскажем, как защитить страницы приложения от неавторизованных пользователей. А пока — закрепите полученные знания с помощью задач.

**Защита маршрутов на фронте**

Вы уже умеете реализовывать регистрацию и аутентификацию в проекте: пользователь может создать учётную запись и зайти под ней в приложение. К тому же, вы многое знаете о роутинге. В этом уроке мы расскажем, как совместить роутинг с авторизацией и аутентификацией, и реализуем полноценную авторизацию.

Сейчас любой пользователь приложения может получить доступ к роуту, просто набрав нужный URL. Например, чтобы увидеть список своих диалогов, пользователь должен пройти процесс аутентификации. В противном случае приложение не будет знать, какой именно список диалогов относится к конкретному пользователю. Это не только неудобно, но и опасно — данные пользователя не защищены. В этом уроке разберём, что с этим делать — настроим процесс авторизации. Но сперва рассмотрим понятие защищённых маршрутов в приложении.

**Защищённые маршруты**

Это маршруты, на которые могут попасть только пользователи с нужным набором прав. В случае нашего чата список диалогов может быть доступен только пользователям, которые прошли процесс аутентификации в приложении.

Теперь реализуем эту функциональность и запретим доступ посторонним пользователям. Для этого нужно создать отдельный компонент — ProtectedRoute. Внутри этого компонента мы извлекаем данные о пользователе из хранилища. Если они корректны, пропускаем пользователя на переданный в компонент маршрут. Если данных в хранилище нет, переадресуем на незащищённый маршрут.

Но обо всём по порядку. Сначала разберёмся, какие маршруты в нашем чате надо защищать, а какие — нет:

* Маршрут /login и компонент экрана LoginPage. Этот маршрут не должен быть защищённым, иначе пользователь не сможет даже попытаться пройти аутентификацию.
* Маршрут / и компонент экрана ListPage. А этот маршрут, как и все вложенные в него, надо защитить — там хранятся приватные данные конкретного пользователя.

Сам компонент ProtectedRoute должен оборачивать все компоненты экранов, которые нужно защитить. Кроме этого, он должен насыщать их переданными в ProtectedRoute пропсами (например, пропсом path) и отрисовывать.

Перейдём в компонент App и обернём маршрут ListPage в пока что пустой компонент ProtectedRoute:

Скопировать кодJSX

import './App.css';

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

import { LoginPage, ListPage } from './pages';

import { ProtectedRoute } from './components/protected-route';

import { ProvideAuth } from './services/auth';

export default function App() {

return (

<ProvideAuth>

<Router>

<Switch>

<Route path="/login">

<LoginPage />

</Route>

{/\* Передадим в ProtectedRoute пропс path \*/}

<ProtectedRoute path="/">

{/\* И сам компонент ListPage \*/}

<ListPage />

</ProtectedRoute>

<Route>404</Route>

</Switch>

</Router>

</ProvideAuth>

);

}

Теперь в компонент ProtectedRoute передаётся маршрут / и экран ListPage. Перейдём в сам компонент ProtectedRoute:

Скопировать кодJSX

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

export function ProtectedRoute({ children, ...rest }) {

return (

<Route

{...rest}

render={() => (

children

)

}

/>

);

}

Сейчас компонент ProtectedRoute — просто обёртка, которая насыщает компонент Route пропсами.

Обратите внимание, что мы используем пропс render компонента Route. Этот пропс вызывает функцию при совпадении URL-маршрута, который передаётся как пропс в хук ProtectedRoute. У Route есть похожий пропс component. Но на деле пропсы render и component работают по-разному: component содержит «под капотом» React.createElement, а render используется как функция. Другими словами, мы избегаем лишнего монтирования компонента при использовании пропса render.

При этом внутри пропса render есть доступ к пропсам маршрута, например к текущему маршруту страницы в пропсе location. Но это нам пригодится немного позже. А пока настроим переадресацию.

**Переадресация. Компонент Redirect**

Чтобы компонент ProtectedRoute перестал быть обычной обёрткой, нужно добавить переадресацию пользователя. Сейчас при попытке перехода по маршруту / может быть только два исхода:

* в хранилище есть данные о пользователе;
* в хранилище отсутствуют данные, и пользователя нужно переадресовать на страницу логина.

Для переадресации используется компонент Redirect — он заменяет текущую запись в навигации истории и перенаправляет пользователя на другую страницу. Его часто используют в веб-приложениях после авторизации. Например, чтобы авторизованный пользователь не мог вернуться на страницу входа. Но пока применим его для другой задачи — перенаправим неавторизованного пользователя по маршруту /login.

Добавим Redirect в компонент ProtectedRoute, но перед этим получим данные о пользователе:

Скопировать кодJSX

import { useAuth } from '../services/auth';

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

import { useEffect, useState } from 'react';

export function ProtectedRoute({ children, ...rest }) {

*// Вернём из хранилища запрос на получение данных о пользователе и*

*// текущий объект с пользователем*

let { getUser, ...auth } = useAuth();

const [isUserLoaded, setUserLoaded] = useState(false);

const init = async () => {

*// Вызовем запрос getUser и изменим состояние isUserLoaded*

await getUser();

setUserLoaded(true);

};

useEffect(() => {

*// При монтировании компонента запросим данные о пользователе*

init();

}, []);

if (!isUserLoaded) {

return null;

}

return (

<Route

{...rest}

render={() =>

// Если пользователь есть, используем компонент, который передан в ProtectedRoute

auth.user ? (

children

) : ()

}

/>

);

}

Теперь, если пользователь не прошёл аутентификацию и объекта user в хранилище нет, перенаправим его по маршруту /login:

Скопировать кодJSX

import { useAuth } from '../services/auth';

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

import { useEffect, useState } from 'react';

export function ProtectedRoute({ children, ...rest }) {

let { getUser, ...auth } = useAuth();

const [isUserLoaded, setUserLoaded] = useState(false);

const init = async () => {

await getUser();

setUserLoaded(true);

};

useEffect(() => {

init();

}, []);

if (!isUserLoaded) {

return null;

}

return (

<Route

{...rest}

render={() =>

auth.user ? (

children

) : (

// Если пользователя нет в хранилище, происходит переадресация на роут /login

<Redirect

to='/login'

/>

)

}

/>

);

}

Компонент ProtectedRoute полностью готов к использованию. Осталось разобрать только несколько нюансов.

**О чём ещё стоит подумать при реализации защищённых маршрутов**

При разработке приложения важно продумывать не только повседневные сценарии. Но начнём с повседневных. Например, пользователь переходит по ссылке из одного небезызвестного поисковика и попадает в наш проект. Сервер пытается аутентифицировать пользователя по наличию токена в куках. Если всё заканчивается неудачей, сервер переадресует его на страницу регистрации или авторизации.

Но нужно проработать ещё и краевые случаи взаимодействия пользователя с приложением. Например, когда пользователь прошёл аутентификацию и авторизацию, но пытается попасть на страницу регистрации или авторизации. Неважно, как он это делает: вводит маршрут вручную или нажимает на кнопку «Назад» в браузере. Важно, что мы не закладываем такое поведение в приложение. Забавная ситуация: до этого мы защищали маршруты от неавторизованных пользователей, а теперь нужно сделать наоборот.

Добавим в компонент экрана Login простое условие: если авторизованный пользователь пытается попасть на маршрут /login, переадресуем его на главную страницу:

Скопировать кодJSX

*// pages/login.jsx*

*// Импортируем все необходимые компоненты*

export function LoginPage() {

let auth = useAuth();

const [form, setValue] = useState({ email: '', password: '' });

const onChange = e => {

setValue({ ...form, [e.target.name]: e.target.value });

};

let login = useCallback(

e => {

e.preventDefault();

auth.signIn(form);

},

[auth, form]

);

*// Проверяем, авторизован ли пользователь*

if (auth.user) {

return (

*// Переадресовываем авторизованного пользователя на главную страницу*

<Redirect

to={{

pathname: '/'

}}

/>

);

}

return (

*// ...*

);

}

Таким простым условием мы защитим приложение от возможных коллизий.

Другой важный сценарий: после авторизации вернуть пользователя на предыдущий экран, если он есть в истории. Например, вы нашли пост со смешными мемами где-нибудь в социальной сети и скидываете ссылку вашему другу. Друг давно не заходил в социальную сеть и его переадресует на страницу авторизации хуком ProtectedRoute. Вот было бы здорово, если после успешной аутентификации и авторизации пользователя переадресовало бы обратно на пост с мемами.

Чтобы реализовать этот сценарий, нужно вернуться к компоненту ProtectedRoute и добавить в него новую функциональность: при переадресации пользователя изменить свойство from объекта state пропса to компонента Redirect. Для решения этой задачи мы могли бы использовать хук useLocation, но изначально выбрали путь воина и применяем пропс render у всего компонента Route. Из функции пропса render можно вытащить свойство location с информацией о текущем маршруте, на который зашёл пользователь.

Достанем свойство location из пропса render и при переадресации укажем текущую страницу как предыдущую:

Скопировать кодJSX

import { useAuth } from '../services/auth';

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

import { useEffect, useState } from 'react';

export function ProtectedRoute({ children, ...rest }) {

let { getUser, ...auth } = useAuth();

const [isUserLoaded, setUserLoaded] = useState(false);

const init = async () => {

await getUser();

setUserLoaded(true);

};

useEffect(() => {

init();

}, []);

if (!isUserLoaded) {

return null;

}

return (

<Route

{...rest}

// Получим текущий маршрут, с которого произойдёт переадресация

// для неавторизованного пользователя

render={({ location }) =>

auth.user ? (

children

) : (

<Redirect

// Передадим в пропс to не строку, а объект.

to={{

// Маршрут, на который произойдёт переадресация

pathname: '/login',

// В from сохраним текущий маршрут

state: { from: location }

}}

/>

)

}

/>

);

}

Теперь, если пользователь прошёл авторизацию или уже авторизован и попадает на экран Login, потребуется просто проверить, есть ли объект state в истории. И, если он есть, отправить пользователя на предыдущий маршрут:

Скопировать кодJSX

*// pages/login.jsx*

*// Импортируем все необходимые компоненты*

export function LoginPage() {

let auth = useAuth();

const [form, setValue] = useState({ email: '', password: '' });

const onChange = e => {

setValue({ ...form, [e.target.name]: e.target.value });

};

let login = useCallback(

e => {

e.preventDefault();

auth.signIn(form);

},

[auth, form]

);

if (auth.user) {

return (

<Redirect

// Если объект state не является undefined, вернём пользователя назад.

to={ state?.from || '/' }

/>

);

}

return (

*// ...*

);

}

Для разработчика это несколько строчек кода в компонентах, а для пользователя — удобный и интуитивный интерфейс.

Теперь точно всё — самое время переходить к заданиям. А в следующем уроке изучим функциональность выхода из учётной записи пользователя.

**Выход из системы**

Вы научились реализовывать аутентификацию и авторизацию в приложении, хранить токены и переадресовывать пользователя. Все эти навыки пригодятся вам прямо сейчас.

Иногда происходит так, что пользователю нужно расстаться со своей авторизованной учётной записью, то есть выйти из неё. В нашем чате после очередного сообщения от «опасного мужика» пользователь тоже может захотеть выйти, а такой функциональности в приложении пока нет. Мы могли бы ввести модерацию сообщений, но проще добавить возможность выходить из приложения по клику на соответствующую иконку.

С точки зрения бизнес-логики приложения потребуется создать функцию signOut со следующими действиями:

1. Отправить POST-запрос к бэкенду. Обычно это отдельный маршрут /logout или /signout.
2. Из хранилища приложения удалить данные о текущем пользователе.
3. Удалить куку с токеном, чтобы после повторного попадания на сайт пользователь не авторизовывался автоматически.

С точки зрения представления в приложении следует вызвать в обработчике события signOut хранилища, а после успешного выполнения запроса — переадресовать пользователя на страницу авторизации.

Напишем функцию signOut в файле auth.js:

Скопировать кодJSX

*// services/auth.js*

export function useProvideAuth() {

const [user, setUser] = useState(null);

const getUser = async () => {

*// Получаем пользователя*

};

const signIn = async form => {

*// Авторизуемся*

};

const signOut = async () => {

*// Отправляем запрос на сервер*

await logoutRequest();

*// Удаляем пользователя из хранилища*

setUser(null);

*// Удаляем куку token*

deleteCookie('token');

};

return {

user,

getUser,

signIn,

signOut

};

}

*// services/utils.js*

export function deleteCookie(name) {

*// Находим куку по ключу token, удаляем её значение,*

*// устанавливаем отрицательное время жизни, чтобы удалить сам ключ token*

setCookie(name, null, { expires: -1 });

}

Основная функция логаута из приложения написана. Остаётся только корректно обработать выход пользователя — настроить переадресацию. Добавим обработчик logout слушателю onClick в компоненте страницы List:

Скопировать кодJSX

import React, { useEffect, useState, useMemo, useCallback } from 'react';

import { ChatsList } from '../components/chats-list';

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

import listStyles from './list.module.css';

import { deserializeQuery, useChatsData } from '../services/api';

import { Avatar } from '../components/avatar';

import { Input } from '../components/input';

import logoutImage from '../images/logout.svg';

import { ChatPage } from './chat';

import { useProvideAuth } from '../services/auth';

export const ListPage = () => {

*// Используем хук useHistory и функцию signOut из хранилища*

const history = useHistory();

const { signOut } = useAuth();

*// ...Основная логика компонента*

const content = loading ? (

'loading'

) : preparedData && preparedData.length ? (

<ChatsList chats={preparedData} />

) : null;

const logout = useCallback(

() => {

},

[signOut, history]

);

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

<div className={listStyles.searchbar}>

<Avatar name={'A'} />

<Input placeholder="Поиск" onChange={onChange} value={searchValue} />

{/\* Добавим обработчик событий на иконку выхода \*/}

<img alt="logout" src={logoutImage} onClick={logout} />

</div>

{content}

</div>

<Switch>

<Route path={`/:chatId`}>

<ChatPage />

</Route>

</Switch>

</div>

);

};

В функции logout вызовем функцию signOut. После её успешного выполнения используем метод replace объекта history, чтобы переадресовать пользователя на страницу авторизации:

Скопировать кодJSX

import React, { useEffect, useState, useMemo, useCallback } from 'react';

import { ChatsList } from '../components/chats-list';

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

import listStyles from './list.module.css';

import { deserializeQuery, useChatsData } from '../services/api';

import { Avatar } from '../components/avatar';

import { Input } from '../components/input';

import logoutImage from '../images/logout.svg';

import { ChatPage } from './chat';

import { useProvideAuth } from '../services/auth';

export const ListPage = () => {

const history = useHistory();

const { signOut } = useAuth();

*// ...Основная логика компонента*

const content = loading ? (

'loading'

) : preparedData && preparedData.length ? (

<ChatsList chats={preparedData} />

) : null;

const logout = useCallback(

() => {

*// Вызовем функцию signOut*

signOut().then(() => {

*// После выхода переадресуем пользователя на маршрут /login*

history.replace({ pathname: '/login' });

});

},

[signOut, history]

);

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

<div className={listStyles.searchbar}>

<Avatar name={'A'} />

<Input placeholder="Поиск" onChange={onChange} value={searchValue} />

<img alt="logout" src={logoutImage} onClick={logout} />

</div>

{content}

</div>

<Switch>

<Route path={`/:chatId`}>

<ChatPage />

</Route>

</Switch>

</div>

);

};

Это всё, что требуется для реализации механизма выхода из учётной записи. Ничего сложного — запрос к бэкенду, очистка хранилища и кук. В конце — переадресация пользователя с текущего маршрута. Усвоить этот материал помогут задания — переходите к ним.