**Оглавление**

[**1 JWT – общая информация** 2](#_Toc74484389)

[**2 Виды токенов** 3](#_Toc74484390)

[**3 Клиент-серверное взаимодействие** 4](#_Toc74484391)

[**4 Реализация** 5](#_Toc74484392)

[**4.1 Сервер на node js** 5](#_Toc74484393)

[**4.2 Клиент на React JS** 7](#_Toc74484394)

[**ПОЛНЫЙ ИСХОДНЫЙ КОД** 10](#_Toc74484395)

# **1 JWT – общая информация**

JWT – JSON Web Token – представляет из себя строку, которая состоит из 3 частей, разделенных точкой.



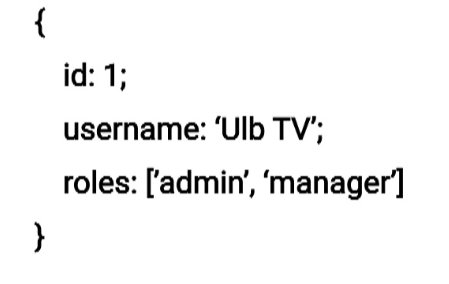
Пример JWT

Первая часть – заголовок (header) – в заголовке указывается информация, которая описывает сам токен. Например, алгоритм шифрования\подписи и тип токена.



Пример заголовка (header)

Вторая часть – данные (payload) – в данные мы можем поместить абсолютно любой объект, но чаще всего туда добавляют информацию о пользователе, например, его ID, email\username и роли [‘ADMIN’, ‘USER’, ‘MANAGER’].



Пример данных (payload)

Данные и заголовок чаще всего кодируются с помощью Base64, что говорит нам о том, что данные из токена может извлечь любой. Именно по этой причине не стоит вшивать в токен пароли или любую другую конфиденциальную информацию.

Безопасность токена осуществляет третья его часть, а именно сигнатура. Для того, чтобы получить сигнатуру необходим специальный секретный ключ, который знает только сервер.

Например: «КАКАЯ\_ТО\_СЕКРЕТНАЯ\_СТРОКА\_12345»

Сигнатура получается в результате шифрования по секретному ключу заголовка и данных.

Шифрование\_по\_ключу (Header + Payload) = сигнатура.

При попытке подделать токен, не зная секретный ключ, практически невозможно получить валидную сигнатуру.

Каждый токен обладает временем жизни, например, 15 минут, 2 часа, 30 дней. По истечению срока годности токен становится не валидным. Это необходимо для того, чтобы если злоумышленник захватил токен, то он не смог использовать его вечно. Через 15 минут токен погибает и злоумышленник теряет доступ к сервису.

# **2 Виды токенов**

Из прошлого раздела мы узнали, что токен обладает сроком жизни. Соответственно, чем меньше срок – тем безопаснее. Если токен обладает сроком жизни в 15 минут, то пользователю придется каждые 15 минут вводить username и пароль?

На этот случай придумали два вида токенов:

* Access token – токен, с помощью которого осуществляется доступ к сервису, обычно срок жизни минимален, 15-60 минут. Хранится на клиенте в localstorage или же глобальном состоянии приложения.
* Refresh token – токен, с помощью которого обновляется просроченный access токен. Срок жизни в разы больше, обычно 15-60 дней. Хранится в cookies и устанавливается сервером. Обязателен флаг httpOnly, чтобы невозможно было обратиться к токену из браузера.

По истечению срока жизни access токена, мы отправляем запрос на сервер с refresh токеном и сервер возвращает нам новую пару access + refresh токенов.

|  |  |  |
| --- | --- | --- |
|  | **ACCESS** | **REFRESH** |
| Срок жизни | 15-60 минут | 15-60 дней |
| Предназначение | Доступ к сервису | Обновление access токена |
| Хранение | LocalStorage браузера или глобальное состояние приложения | База данных + httpOnly cookies |

Сравнительная таблица

# **3 Клиент-серверное взаимодействие**

Пользователь вводит username и пароль, нажимает кнопку «войти», после чего запрос уходит на сервер.

Сервер генерирует пару токенов (access, refresh), refresh токен сохраняет в базу данных/redis (также можно хранить ip адрес, с которого зашел пользователь, fingerprint браузера, и любые другие вспомогательные данные, своего рода сессия). После генерации токенов сервер отправляет их на клиент, при этом refresh токен устанавливается в httpOnly куку сервером.

После ответа от сервера, клиент сохраняет access токен в localStorage и использует его при каждом обращении к серверу в заголовках http запроса.

Headers: {

Authorization: «Bearer ${ACCESS\_TOKEN}»

}

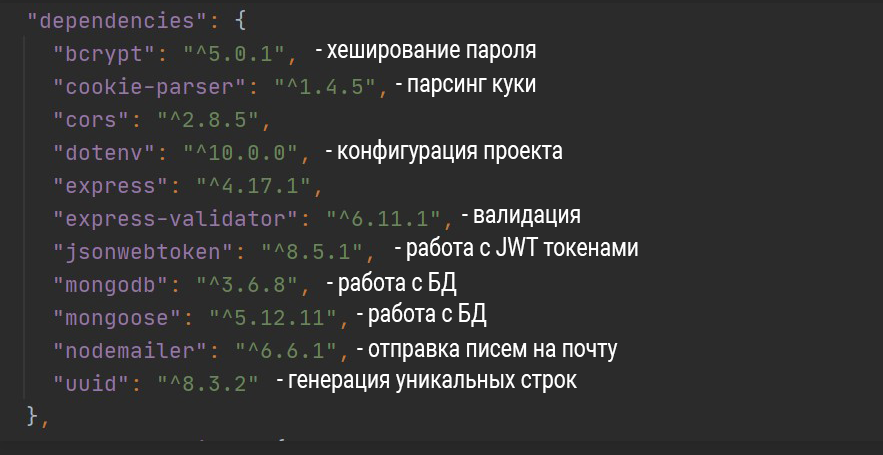
Сервер, при каждом обращении клиента проверяет токен и идентифицирует пользователя, если токен не просрочен и не подделан, то доступ к сервису у пользователя разрешен, иначе возвращает http status 401.

# **4 Реализация**

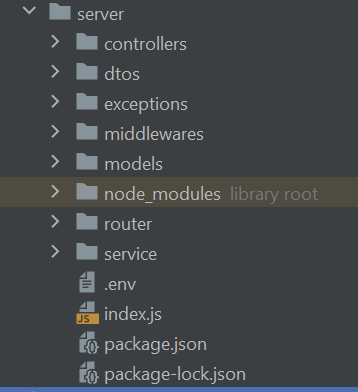
## **4.1 Сервер на node js**

1. Проинициализируем проект командой npm init –y.

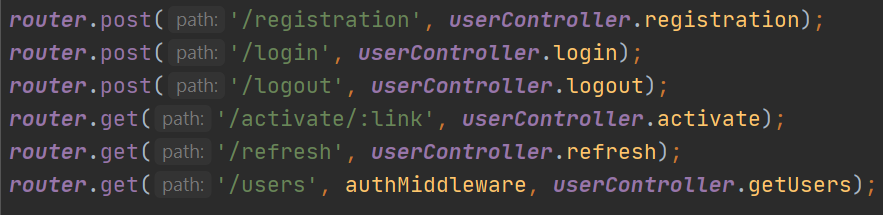
2. Установим необходимые зависимости



3. Структура проекта



4. Ендпоинты



Функция с логикой логина

async login(email, password) {  
 const user = await ***UserModel***.findOne({email})  
 if (!user) {  
 throw ApiError.*BadRequest*('Пользователь с таким email не найден')  
 }  
 const isPassEquals = await bcrypt.compare(password, user.password);  
 if (!isPassEquals) {  
 throw ApiError.*BadRequest*('Неверный пароль');  
 }  
 const userDto = new UserDto(user);  
 const tokens = ***tokenService***.generateTokens({...userDto});  
  
 await ***tokenService***.saveToken(userDto.id, tokens.refreshToken);  
 return {...tokens, user: userDto}  
}

Генерация токенов

generateTokens(payload) {  
 const accessToken = ***jwt***.sign(payload, ***process***.env.JWT\_ACCESS\_SECRET, {expiresIn: '15s'})  
 const refreshToken = ***jwt***.sign(payload, ***process***.env.JWT\_REFRESH\_SECRET, {expiresIn: '30s'})  
 return {  
 accessToken,  
 refreshToken  
 }  
}

Middleware для проверки access токена

module.exports = function (req, res, next) {  
 try {  
 const authorizationHeader = req.headers.authorization;  
 if (!authorizationHeader) {  
 return next(ApiError.*UnauthorizedError*());  
 }  
 const accessToken = authorizationHeader.split(' ')[1];  
 if (!accessToken) {  
 return next(ApiError.*UnauthorizedError*());  
 }  
 const userData = ***tokenService***.validateAccessToken(accessToken);  
 if (!userData) {  
 return next(ApiError.*UnauthorizedError*());  
 }  
 req.user = userData;  
 next();  
 } catch (e) {  
 return next(ApiError.*UnauthorizedError*());  
 }  
};

Валидация Access и Refresh токенов

validateAccessToken(token) {  
 try {  
 const userData = ***jwt***.verify(token, ***process***.env.JWT\_ACCESS\_SECRET);  
 return userData;  
 } catch (e) {  
 return null;  
 }  
}  
  
validateRefreshToken(token) {  
 try {  
 const userData = ***jwt***.verify(token, ***process***.env.JWT\_REFRESH\_SECRET);  
 return userData;  
 } catch (e) {  
 return null;  
 }  
}

Функция для обновления Access токена по Refresh токену

async refresh(refreshToken) {  
 if (!refreshToken) {  
 throw ApiError.*UnauthorizedError*();  
 }  
 const userData = ***tokenService***.validateRefreshToken(refreshToken);  
 const tokenFromDb = await ***tokenService***.findToken(refreshToken);  
 if (!userData || !tokenFromDb) {  
 throw ApiError.*UnauthorizedError*();  
 }  
 const user = await ***UserModel***.findById(userData.id);  
 const userDto = new UserDto(user);  
 const tokens = ***tokenService***.generateTokens({...userDto});  
  
 await ***tokenService***.saveToken(userDto.id, tokens.refreshToken);  
 return {...tokens, user: userDto}  
}

## **4.2 Клиент на React JS**

Создаем инстанс axios.

export const ***API\_URL*** = `http://localhost:5000/api`  
  
const ***$api*** = ***axios***.create({  
 withCredentials: true,  
 baseURL: ***API\_URL***})

Добавляем перехватчик, который к каждому запросу добавляет заголовок Authorization с access токеном.

***$api***.interceptors.request.use((config) => {  
 config.headers.Authorization = `Bearer ${***localStorage***.getItem('token')}`  
 return config;  
})

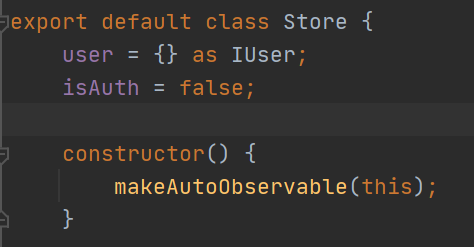
Добавляем перехватчик, который на 401 статус код пробует обновить access токен.

***$api***.interceptors.response.use((config) => {  
 return config;  
},async (error) => {  
 const originalRequest = error.config;  
 if (error.response.status == 401 && error.config && !error.config.\_isRetry) {  
 originalRequest.\_isRetry = true;  
 try {  
 const response = await ***axios***.get<AuthResponse>(`${***API\_URL***}/refresh`, {withCredentials: true})  
 ***localStorage***.setItem('token', response.data.accessToken);  
 return ***$api***.request(originalRequest);  
 } catch (e) {  
 ***console***.log('НЕ АВТОРИЗОВАН')  
 }  
 }  
 throw error;  
})

Реализуем класс с функциями, в которых отправляем запросы на сервер.



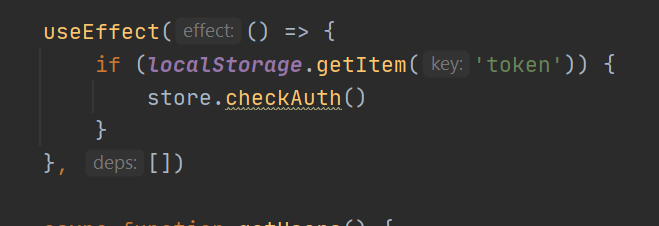
Создаем глобальное хранилище, в котором храним информацию о пользователе и Boolean флаг, который показывает авторизован пользователь или нет.



Создаем компонент с 2 инпутами, для почты и для пароля.

const LoginForm: FC = () => {  
 const [email, setEmail] = useState<string>('')  
 const [password, setPassword] = useState<string>('')  
 const {store} = useContext(***Context***);  
  
 return (  
 <div>  
 <input  
 onChange={e => setEmail(e.target.value)}  
 value={email}  
 type="text"  
 placeholder='Email'  
 />  
 <input  
 onChange={e => setPassword(e.target.value)}  
 value={password}  
 type="password"  
 placeholder='Пароль'  
 />  
 <button onClick={() => store.login(email, password)}>  
 Логин  
 </button>  
 <button onClick={() => store.registration(email, password)}>  
 Регистрация  
 </button>  
 </div>  
 );  
};

В компоненте App при первом запуске приложения отправляем запрос на проверку токена.



В зависимости от того, авторизован пользователь или нет рисуем разный шаблон.

if (!store.isAuth) {  
 return (  
 <div>  
 <LoginForm/>  
 <button onClick={getUsers}>Получить пользователей</button>  
 </div>  
 );  
}  
  
return (  
 <div>  
 <h1>{store.isAuth ? `Пользователь авторизован ${store.user.email}` : 'АВТОРИЗУЙТЕСЬ'}</h1>  
 <h1>{store.user.isActivated ? 'Аккаунт подтвержден по почте' : 'ПОДТВЕРДИТЕ АККАУНТ!!!!'}</h1>  
 <button onClick={() => store.logout()}>Выйти</button>  
 <div>  
 <button onClick={getUsers}>Получить пользователей</button>  
 </div>  
 {users.map(user =>  
 <div key={user.email}>{user.email}</div>  
 )}  
 </div>  
);

# **ПОЛНЫЙ ИСХОДНЫЙ КОД**

https://github.com/utimur/fullstack-jwt-auth