

Реализация авторизации через Google OAuth 2.0 в веб-приложении на базе NestJS и Next.js

Андриенко Иван Сергеевич

Приамурский государственный университет имени Шолом-Алейхема
Студент

Пащенко Екатерина Александровна

Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация

В статье рассматривается практическая реализация авторизации пользователей через протокол OAuth 2.0 с использованием провайдера Google в полнофункциональном веб-приложении. Описывается подготовительный этап конфигурации OAuth-клиента в Google Cloud Console, необходимый для получения идентификатора клиента и секрета, а также корректной настройки redirect URI. Основное внимание уделено серверной части на базе NestJS с использованием Passport.js и стратегии passport-google-oauth20, где выполняется обработка OAuth-потока, получение access/refresh токенов и формирование данных пользователя. Показано, что предложенная архитектура обеспечивает устойчивую сессию пользователя без server-side сессий, позволяет использовать токены для вызовов Google API и поддерживает механизм обновления access-токена через refresh-токен.

Ключевые слова: OAuth 2.0, Google OAuth, NestJS, Next.js, Passport.js, access token, refresh token, аутентификация, авторизация, localStorage.

Implementing authorization via Google OAuth 2.0 in a NestJS-based web application and Next.js

Andrienko Ivan Sergeevich

Sholom-Aleichem Priamursky State University
Student

Paschenko Ekaterina Alexandrovna

Sholom-Aleichem Priamursky State University
Student

Abstract

The article discusses the practical implementation of user authorization via the OAuth 2.0 protocol using the Google provider in a fully functional web application. The preparatory stage of the OAuth client configuration in the Google Cloud Console is described, which is necessary to obtain the client ID and secret, as well

as the correct configuration of the redirect URI. The main focus is on the NestJS-based server side using Passport.js and passport-google-oauth20 strategies, which process the OAuth stream, receive access/refresh tokens, and generate user data. It is shown that the proposed architecture provides a stable user session without server-side sessions, allows the use of tokens for Google API calls, and supports an access token update mechanism via a refresh token.

Keywords: OAuth 2.0, Google OAuth, NestJS, Next.js, Passport.js, access token, refresh token, authentication, authorization, localStorage.

1 Введение

1.1 Актуальность

В современных веб-системах авторизация через внешних провайдеров стала стандартом пользовательского опыта, поскольку снижает порог входа и позволяет применять доверенную инфраструктуру безопасности, предоставляемую крупными платформами. Протокол OAuth 2.0 является одной из наиболее распространённых моделей делегированной авторизации, при которой система получает ограниченный доступ к данным пользователя без прямого хранения пароля, что особенно важно при интеграции с API внешних сервисов.

1.2 Обзор исследований

Реализация авторизации пользователей с использованием протокола Google OAuth активно рассматривается в научной и прикладной литературе. В работе Р.И. Круглика анализируются практические аспекты внедрения Google OAuth API в веб-приложениях, включая особенности настройки внешней авторизации и упрощения процесса входа для пользователей без хранения паролей на стороне сервиса [1]. А.С. Мосягин в своём исследовании подробно рассматривает принципы функционирования системы авторизации Google, описывая этапы OAuth-аутентификации, получение токенов доступа и обработку пользовательских данных на примерах реальных веб-сценариев [2]. В статье И.С. Красильникова исследуются вопросы разработки функций авторизации в сервисах компании Google, где подчёркивается универсальность OAuth-подхода и его применимость для интеграции сторонних веб-систем с экосистемой Google [3]. А.А. Дудак в своём исследовании проводит сравнительный анализ инструментов разработки для проектных систем, выделяя преимущества стека технологий React и TypeScript по производительности и удобству сопровождения кода [4]. В статье Д.А.Хайруллина и М.Р. Богданова рассматриваются особенности применения статической типизации в React-приложениях, включая методики типизации компонентов, хуков и состояния, а также влияние строгой типизации на надёжность пользовательских интерфейсов [5].

1.3 Цель исследования

Целью работы является разработка и описание реализации авторизации через Google OAuth 2.0 в веб-приложении с разделением ответственности

между серверной частью на NestJS и клиентской частью на Next.js. В рамках исследования рассматриваются настройки стратегии Passport.js, организация маршрутов входа и callback-обработчика, безопасное хранение данных пользователя на клиенте, применение контекста для управления состоянием и реализация обновления access-токена с использованием refresh-токена.

2 Материалы и методы

В качестве технологической основы использовались серверный фреймворк NestJS на платформе Node.js и клиентский фреймворк Next.js (React) с TypeScript. Для реализации OAuth-потока применялся Passport.js и стратегия passport-google-oauth20, обеспечивающая стандартный механизм перенаправления пользователя к провайдеру и обработки callback-запроса. На сервере использовались переменные окружения для хранения чувствительных параметров OAuth-клиента, включая идентификатор и секрет, а также базовые URL для формирования callback и redirect-ссылок.

3 Результаты и обсуждения

Перед реализацией OAuth-авторизации требуется выполнить базовую конфигурацию в Google Cloud Console: создать проект, включить необходимые API, сформировать OAuth-клиент и задать корректные redirect URI для обработки возврата пользователя после согласия. После получения client ID и client secret система может инициировать OAuth-поток и обменивать данные согласия на токены доступа. На практическом уровне данный этап является подготовительным, однако он критичен для корректной работы всей схемы аутентификации [6].

В результате выполненной работы реализована полнофункциональная схема авторизации пользователей через Google OAuth 2.0, охватывающая как серверную, так и клиентскую части веб-приложения. Архитектура решения основана на классическом OAuth-потоке с редиректом пользователя к провайдеру, получением токенов доступа и последующим управлением состоянием авторизации на стороне клиента.

На серверной стороне ключевым элементом является стратегия Passport для Google OAuth, в которой задаются идентификатор клиента, секрет и URL обратного вызова. В процессе аутентификации Passport автоматически иницирует перенаправление пользователя на страницу входа Google, а после успешного подтверждения прав доступа получает профиль пользователя и набор токенов. Конфигурация стратегии и перечень запрашиваемых scope, включая доступ к профилю и API Google Forms, представлены в файле backend/src/auth/google.strategy.ts (рис. 1). В методе валидации формируется объект пользователя, содержащий электронную почту, имя, фотографию, а также access и refresh токены, которые далее используются в системе.

```
backend > src > auth > T$ google.strategy.ts > ...
1 import { Injectable } from '@nestjs/common';
2 import { PassportStrategy } from '@nestjs/passport';
3 import { Strategy, Profile } from 'passport-google-oauth20';
4
5 @Injectable()
6 export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
7   constructor() {
8     super({
9       clientID: process.env.GOOGLE_CLIENT_ID!,
10      clientSecret: process.env.GOOGLE_CLIENT_SECRET!,
11      callbackURL: `${process.env.BACKEND_URL}/auth/google/redirect`,
12      passReqToCallback: false, // ✅ важное исправление
13      scope: [
14        'email',
15        'profile',
16        'https://www.googleapis.com/auth/forms.body',
17        'https://www.googleapis.com/auth/drive.file',
18      ],
19    });
20   }
21
22   async validate(accessToken: string, refreshToken: string, profile: Profile) {
23     return {
24       email: profile.emails?.[0]?.value,
25       accessToken,
26       refreshToken,
27       name: profile.displayName,
28       picture: profile.photos?.[0]?.value,
29     };
30   }
31 }
32
```

Рис. 1. Конфигурация стратегии Google OAuth и запрашиваемые права доступа

Обработка возврата пользователя после успешной аутентификации реализована в контроллере авторизации. Callback-маршрут принимает данные, предоставленные Passport, и выполняет редирект на клиентскую часть приложения, передавая информацию о пользователе в закодированном виде. Реализация маршрутов и логика редиректа представлены в файле backend/src/auth/auth.controller.ts (рис. 2). Дополнительно в контроллере реализован endpoint обновления access-токена, который выполняет запрос к Google OAuth API с использованием refresh-токена, что позволяет поддерживать длительную пользовательскую сессию без повторного входа.

```
4 Controller,  
5 Get,  
6 Post,  
7 Req,  
8 Res,  
9 UseGuards,  
10 } from '@nestjs/common';  
11 import { AuthGuard } from '@nestjs/passport';  
12 import axios from 'axios';  
13  
14 @Controller('auth')  
15 export class AuthController {  
16   @Get('google')  
17   @UseGuards(AuthGuard('google'))  
18   login() {  
19     // redirect to Google Login  
20   }  
21  
22   @Get('google/redirect')  
23   @UseGuards(AuthGuard('google'))  
24   async googleAuthRedirect(@Req() req, @Res() res) {  
25     const user = req.user;  
26  
27     const encoded = encodeURIComponent(JSON.stringify(user));  
28  
29     return res.redirect(`${process.env.GOOGLE_REDIRECT_URL}?user=${encoded}`);  
30   }  
31  
32   @Post('refresh')  
33   async refreshAccessToken(  
34     @Body('refreshToken') refreshToken: string,  
35   ): Promise<{ accessToken: string; expiresIn: number }> {  
36     if (!refreshToken) {  
37       throw new BadRequestException('refreshToken is required');  
38     }  
39  
40     const params = new URLSearchParams({  
41       client_id: process.env.GOOGLE_CLIENT_ID!,  
42       client_secret: process.env.GOOGLE_CLIENT_SECRET!,  
43       refresh_token: refreshToken,  
44       grant_type: 'refresh_token',  
45     });  
46  
47     const response = await axios.post(  
48       'https://oauth2.googleapis.com/token',  
49       params,  
50       {  
51         headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
```

Рис. 2. Контроллер обработки OAuth-авторизации и редиректа пользователя

На клиентской стороне ключевую роль играет отдельная страница обработки OAuth-редиректа. После возврата от Google данные пользователя извлекаются из параметров URL, декодируются и сохраняются в localStorage браузера. Данный этап обеспечивает сохранение состояния авторизации между перезагрузками страницы и повторными визитами пользователя. Логика обработки редиректа и сохранения данных представлена в файле frontend/app/oauth/page.tsx (рис. 3).

```
frontend > app > oauth > page.tsx > ...
1  "use client";
2
3  import { useSearchParams } from "next/navigation";
4  import { Suspense, useEffect } from "react";
5  import { useI18n } from "@/hooks/use-i18n";
6
7  type GoogleUser = {
8    accessToken: string;
9    refreshToken?: string;
10   email?: string;
11   name?: string;
12 };
13
14 function OAuthContent() {
15   const params = useSearchParams();
16   const { dictionary } = useI18n();
17
18   useEffect(() => {
19     const raw = params.get("user");
20     if (!raw) return;
21
22     const user = JSON.parse(decodeURIComponent(raw)) as GoogleUser;
23     localStorage.setItem("google_user", JSON.stringify(user));
24     const redirect = localStorage.getItem("post_login_redirect");
25     localStorage.removeItem("post_login_redirect");
26     window.location.href = redirect || "/";
27   }, [params]);
28
29   return <div>{dictionary.oauth.loading}</div>;
30 }
31
32 export default function OAuthPage() {
33   return (
34     <Suspense fallback=<div>Loading...</div>>
35     <OAuthContent />
36   </Suspense>
37 );
38 }
39
```

Рис. 3. Обработка возврата пользователя после авторизации на стороне клиента

Централизованное управление состоянием пользователя реализовано с использованием React-контекста. Контекст инициализируется при загрузке приложения, считывает данные пользователя из `localStorage` и синхронизирует их с состоянием интерфейса. В контексте также реализованы функции входа, выхода, обновления профиля, получения списка пользовательских данных и обновления `access`-токена. Данная логика сосредоточена в файле `frontend/context/user-context.tsx` (рис. 4). Такой подход позволяет избежать дублирования логики авторизации в отдельных компонентах и обеспечивает единый источник данных о текущем пользователе.

```

1  "use client";
2
3  /* eslint-disable react-hooks/set-state-in-effect */
4
5  import {
6    ReactNode,
7    createContext,
8    useCallback,
9    useContext,
10   useEffect,
11   useMemo,
12   useState,
13 } from "react";
14 import { api, RemoteProfile, RemoteTest } from "@lib/api";
15
16 export type GoogleUser = {
17   email: string;
18   accessToken: string;
19   refreshToken?: string;
20   name?: string;
21   picture?: string;
22 };
23
24 type UserContextValue = {
25   user: GoogleUser | null;
26   profile: RemoteProfile | null;
27   tests: RemoteTest[];
28   loading: boolean;
29   login: () => void;
30   logout: () => void;
31   refreshProfile: (emailOverride?: string) => Promise<void>;
32   refreshTests: (emailOverride?: string) => Promise<void>;
33   refreshAccessToken: () => Promise<string | null>;
34 };
35
36 const UserContext = createContext<UserContextValue | undefined>(undefined);
37
38 const BACKEND_URL = process.env.NEXT_PUBLIC_BACKEND_URL;
39 const STORAGE_KEY = "google_user";
40 const REDIRECT_KEY = "post_login_redirect";
41
42 export function UserProvider({ children }: { children: ReactNode }) {
43   const [user, setUser] = useState<GoogleUser | null>(null);
44   const [profile, setProfile] = useState<RemoteProfile | null>(null);
45   const [tests, setTests] = useState<RemoteTest[]>([]);
46   const [loading, setLoading] = useState(true);
47
48   const readStoredUser = useCallback(() => {
49     if (typeof window === "undefined") return null;

```

Рис. 4. Контекст управления состоянием пользователя и OAuth-токенами

Интеграция механизма авторизации с пользовательским интерфейсом реализована через компоненты хедера. Кнопки «Войти» и «Выйти» динамически отображаются в зависимости от состояния контекста пользователя, а инициирование авторизации выполняется через перенаправление на серверный маршрут `/auth/google`. Использование хука доступа к контексту обеспечивает прозрачное взаимодействие UI с системой авторизации без прямой работы с токенами (рис. 5).

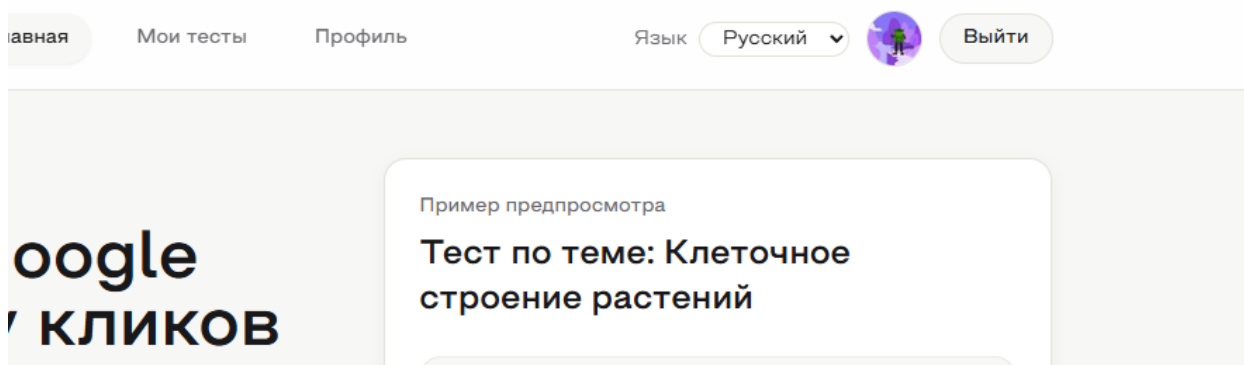


Рис. 5. Использование механизма авторизации в интерфейсе хедера

Выбранная архитектура позволяет реализовать авторизацию без использования серверных сессий, полностью опираясь на OAuth-механизмы и клиентское хранение данных. Access-токен используется для выполнения запросов к Google API, а refresh-токен обеспечивает обновление доступа без повторного участия пользователя.

4 Выводы

Реализация авторизации пользователей с использованием технологии Google OAuth показала, что применение серверной стратегии на базе Passport.js и клиентского контекстного управления состоянием позволяет надёжно и последовательно организовать процесс аутентификации в веб-приложении. Разделение логики между backend-частью, отвечающей за OAuth-взаимодействие и управление токенами, и frontend-частью, обеспечивающей хранение пользовательских данных и обновление интерфейса, обеспечило устойчивую работу системы входа. Проведённая реализация подтвердила корректность обработки access и refresh токенов, сохранения состояния пользователя и автоматического восстановления сессии, что делает предложенный подход пригодным для использования в современных веб-сервисах с интеграцией Google API.

Библиографический список

1. Круглик Р.И. Авторизация на сайте через Google OAuth API // Постулат. 2020. № 1 (51). С. 98.
2. Мосягин А.С. Рассмотрение принципов авторизации в системе Google с примерами // Аллея науки. 2018. Т. 3. № 7 (23). С. 119–124.
3. Красильников И.С. Разработка функций для авторизации пользователя в сервисах компании Google // В сборнике: XX Всероссийская студенческая научно-практическая конференция Нижневартковского государственного университета. Нижневартовск, 2018. С. 188–190.
4. Дудак А.А. Сравнительный анализ инструментов разработки для систем управления проектами: преимущества стек-технологий TypeScript и React // Новая наука: от идеи к результату. 2024. № 9. С. 32–40.
5. Хайруллин Д.А., Богданов М.Р. Изучение использования TypeScript в React-приложениях, преимущества статической типизации, подходы к типизации компонентов, хуков и состояния // Международный журнал гуманитарных и естественных наук. 2024. № 12-3 (99). С. 233–238.
6. Google Cloud console. URL: <https://console.cloud.google.com> (дата обращения: 28.11.2025)