Purwadhika
Digital Technology School

AI Fullstack Software Development

User Authentication and Role-Based Authorization

Job Connector Program

Outline

- Authentication
- Role-based Access Control
- Refresh Token Strategy
- Social Authentication (Google)

What Are Authentication and Authorization?

- **Authentication** → Verifies *who* the user is.
- **Authorization** → Defines *what* the user can do.
- Both are essential for securing modern web applications.
- Usually implemented together but serve different purposes.

Example:

When you log into a website, authentication checks your identity, and authorization checks your permissions.

Authentication: Identity Verification

Confirms the user’s identity (e.g., **email**, **username**, **password**). Usually handled with login forms, JWT tokens, or OAuth providers. Common methods:

- Basic Authentication
- Token-based Authentication (JWT)
- Session-based Authentication
- OAuth / Social Login

Authorization: Access Control

- Determines *which actions* a user is allowed to perform.
- Depends on roles or permissions:
    - **Admin**, **User**, **Guest**, etc.

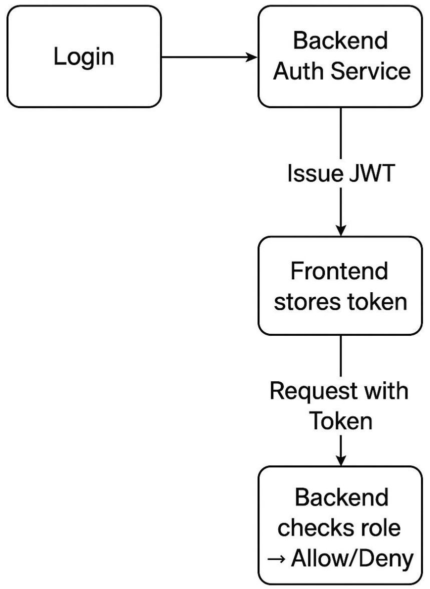
- Implemented after **authentication succeeds**.

Example Scenario:

Only Admins can delete posts, while regular Users can only read or comment.

Authentication & Authorization Flow

1. **Login Request:** User sends credentials to the backend.
2. **Authentication:** Server validates identity (hash check).
3. **Token Issuance:** JWT or session created upon success.
4. **Authorization:** Token verified, roles checked before allowing access.
5. **Access Granted:** Server responds with protected resource.

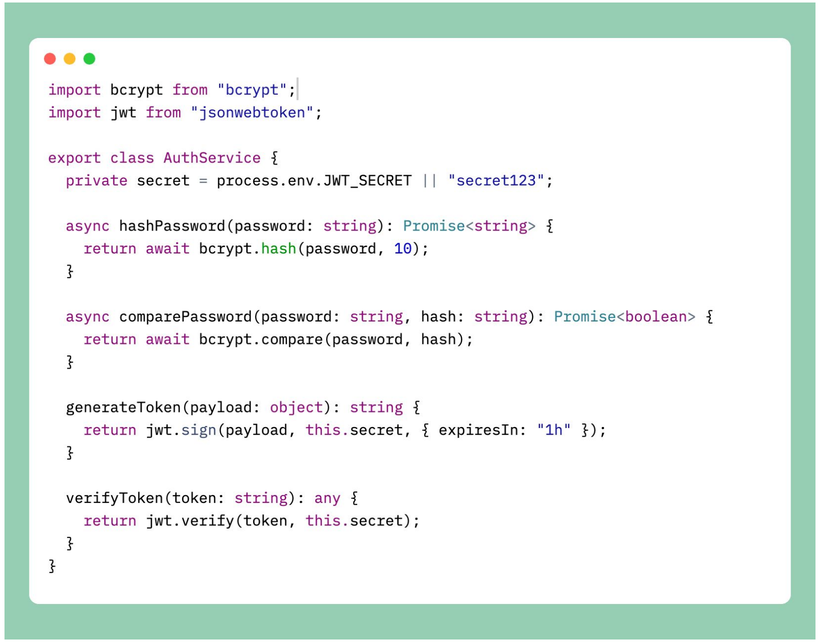


How It Works in the Backend?

Initialize express project, **bcrypt** (for hashing), and **jsonwebtoken** (for jwt generator)

```
npm install express bcrypt jsonwebtoken dotenv
npm install -D typescript ts-node-dev @types/express @types/jsonwebtoken
```

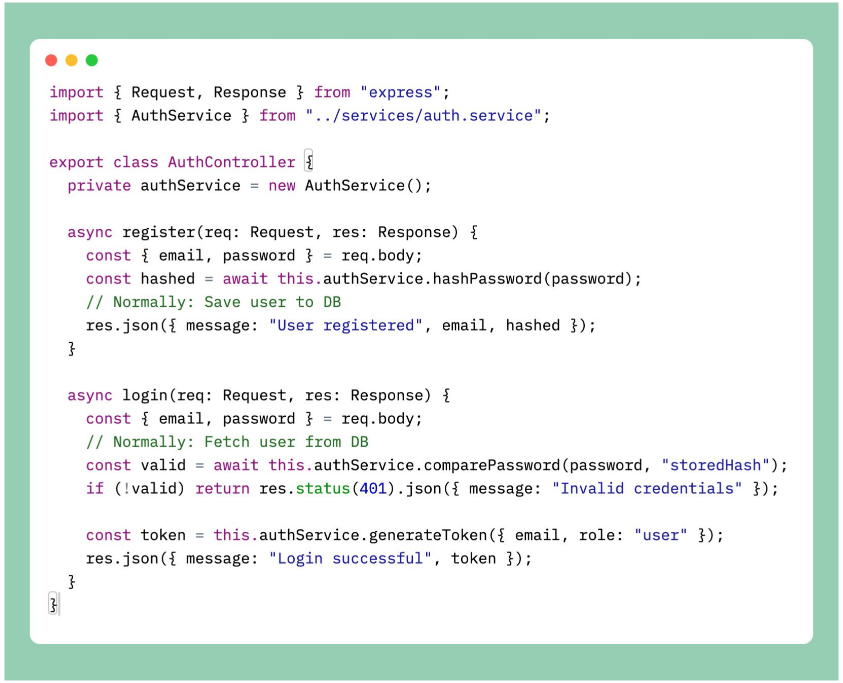
Implementing Authentication Service



```
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";

export class AuthService {
  private secret = process.env.JWT_SECRET || "secret123";

  async hashPassword(password: string): Promise<string> {
    return await bcrypt.hash(password, 10);
  }

  async comparePassword(password: string, hash: string): Promise<boolean> {
    return await bcrypt.compare(password, hash);
  }

  generateToken(payload: object): string {
    return jwt.sign(payload, this.secret, { expiresIn: "1h" });
  }

  verifyToken(token: string): any {
    return jwt.verify(token, this.secret);
  }
}
```

- Handles password hashing, token creation, and verification.
- Keeps logic reusable for controller and middleware.
- To generate **JWT_SECRET**, use the following command in your terminal:
  - `openssl rand -base64 32`
- This command will produce a secure random string that you can set as your **JWT_SECRET** in the .env file, for example:
  - `JWT_SECRET=your_generated_secret_here`

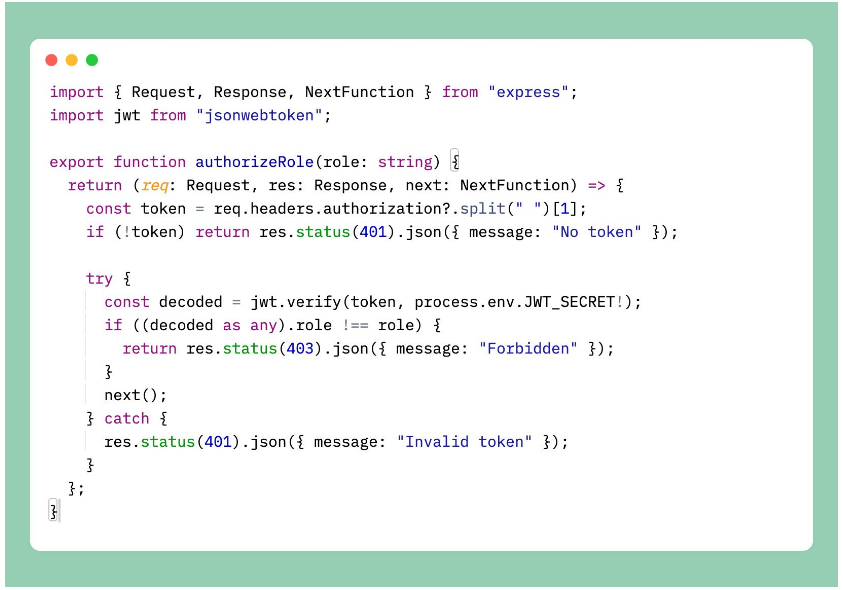
Handling Login and Register



```typescript
import { Request, Response } from "express";
import { AuthService } from "../services/auth.service";

export class AuthController {
  private authService = new AuthService();

  async register(req: Request, res: Response) {
    const { email, password } = req.body;
    const hashed = await this.authService.hashPassword(password);
    // Normally: Save user to DB
    res.json({ message: "User registered", email, hashed });
  }

  async login(req: Request, res: Response) {
    const { email, password } = req.body;
    // Normally: Fetch user from DB
    const valid = await this.authService.comparePassword(password, "storedHash");
    if (!valid) return res.status(401).json({ message: "Invalid credentials" });

    const token = this.authService.generateToken({ email, role: "user" });
    res.json({ message: "Login successful", token });
  }
}
```

- Register ➔ hash and save password.
- Login ➔ verify password and issue JWT token.

Implementing Role-Based Authorization



```
import { Request, Response, NextFunction } from "express";
import jwt from "jsonwebtoken";

export function authorizeRole(role: string) {
  return (req: Request, res: Response, next: NextFunction) => {
    const token = req.headers.authorization?.split(" ")[1];
    if (!token) return res.status(401).json({ message: "No token" });

    try {
      const decoded = jwt.verify(token, process.env.JWT_SECRET!);
      if ((decoded as any).role !== role) {
        return res.status(403).json({ message: "Forbidden" });
      }
      next();
    } catch {
      res.status(401).json({ message: "Invalid token" });
    }
  };
}
```

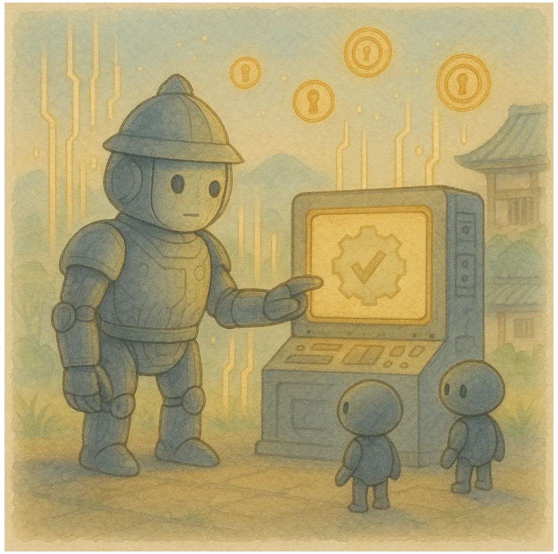
- Checks JWT validity.
- Verifies user’s role (e.g., Admin, User).
- Restricts access based on permissions.

The Purpose of Refresh Tokens

- Access Token (JWT) usually expires quickly — e.g., 15 min or 1 hour.
- Constantly logging in again reduces user experience.
- Refresh Token solves this by issuing a *new access token* without re-login.

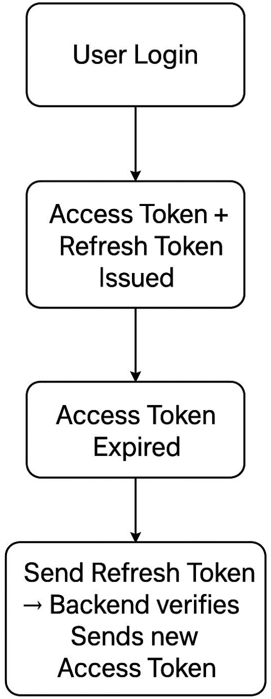
Analogy:

> *Think of an access token as a short-term ticket, and the refresh token as a backstage pass that lets you renew your ticket when it expires.*

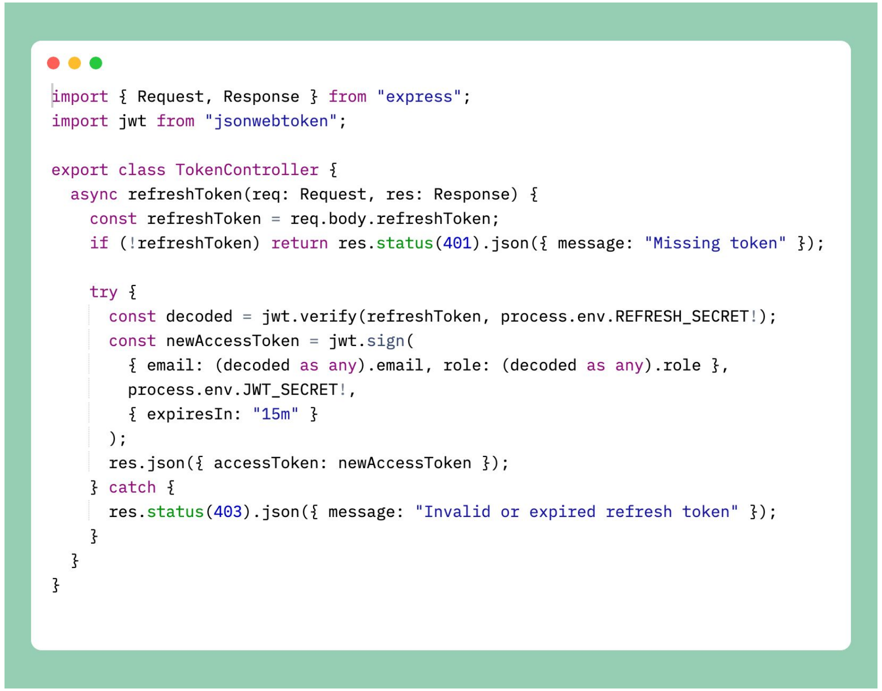


How Refresh Tokens Work

1. **Login**: User logs in ➔ server issues *access token* and *refresh token*.
2. **Access Token Expiry**: When the access token expires, the frontend requests a new one.
3. **Refresh Request**: Frontend sends the refresh token to /refresh-token endpoint.
4. **Validation**: Backend verifies the refresh token (signature + existence).
5. **New Tokens**: Backend issues a new access token (and optionally a new refresh token).


User Login
Access Token + Refresh Token Issued
Access Token Expired
Send Refresh Token → Backend verifies Sends new Access Token

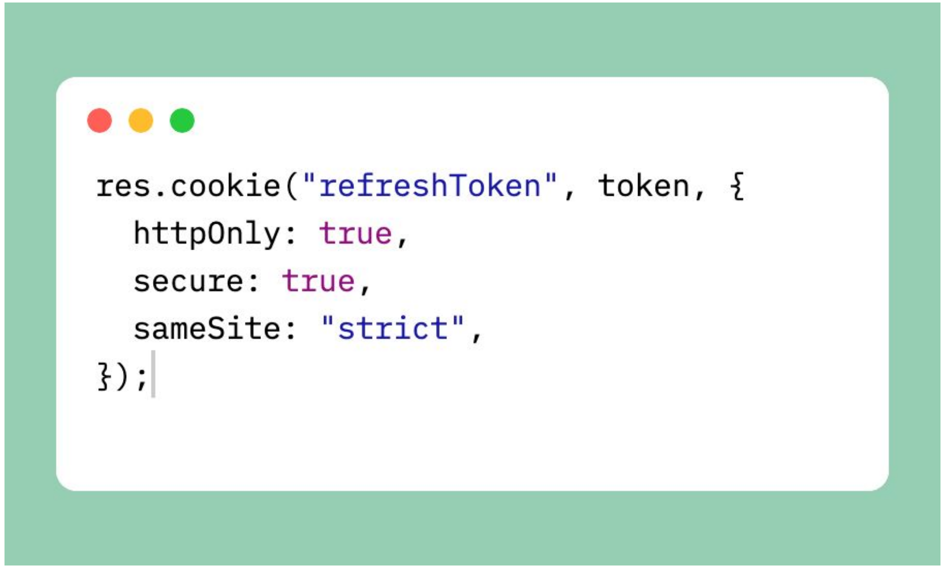
Implementing Refresh Token Logic



```
import { Request, Response } from "express";
import jwt from "jsonwebtoken";

export class TokenController {
  async refreshToken(req: Request, res: Response) {
    const refreshToken = req.body.refreshToken;
    if (!refreshToken) return res.status(401).json({ message: "Missing token" });

    try {
      const decoded = jwt.verify(refreshToken, process.env.REFRESH_SECRET!);
      const newAccessToken = jwt.sign(
        { email: (decoded as any).email, role: (decoded as any).role },
        process.env.JWT_SECRET!,
        { expiresIn: "15m" }
      );
      res.json({ accessToken: newAccessToken });
    } catch {
      res.status(403).json({ message: "Invalid or expired refresh token" });
    }
  }
}
```

- `refreshToken` is verified using a separate secret key (`REFRESH_SECRET`).
- If valid, the system issues a **new access token**.
- Keeps user logged in without re-entering credentials.

How to Store Refresh Tokens Safely



```
res.cookie("refreshToken", token, {
  httpOnly: true,
  secure: true,
  sameSite: "strict",
});
```

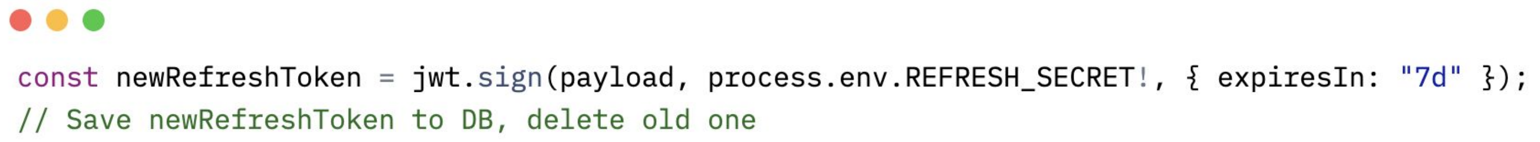
- Store refresh tokens in an HTTP-only cookie (not in localStorage).
- Encrypt or hash refresh tokens before saving them to the database.
- Allow only one active refresh token per user.
- Revoke or delete token on logout or password change.

Why it matters:

Prevents XSS and token theft attacks by avoiding client-side exposure.

Token Rotation Strategy

- **Rotation** = generating a *new **refresh token*** each time a new access token is issued.
- Old refresh tokens are invalidated immediately.
- Helps prevent token replay attacks if a token is leaked.



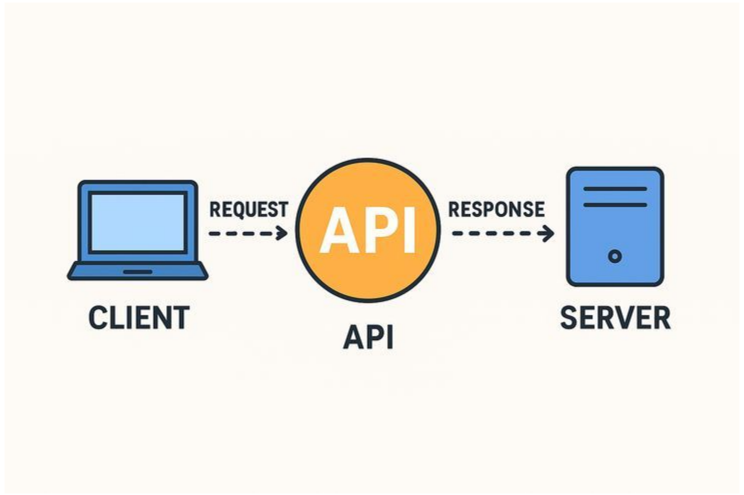
```
const newRefreshToken = jwt.sign(payload, process.env.REFRESH_SECRET!, { expiresIn: "7d" });
// Save newRefreshToken to DB, delete old one
```

React + Vite Integration

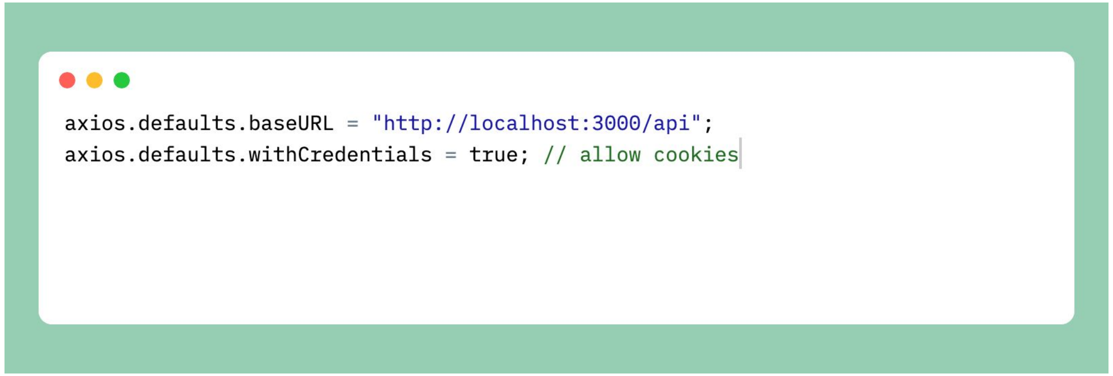
The frontend communicates with the backend using **Axios** or **Fetch API**. Tokens (access & refresh) are used to authenticate API requests.

Common flow:

1. User logs in ➜ backend sends **accessToken** + **refreshToken**.
2. Access token stored (memory/localStorage).
3. Refresh token stored securely (HTTP-only cookie).
4. Frontend attaches token in headers for every request.



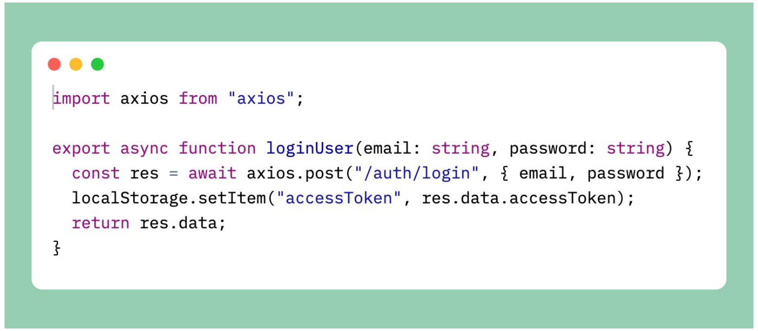
Base URL Setup



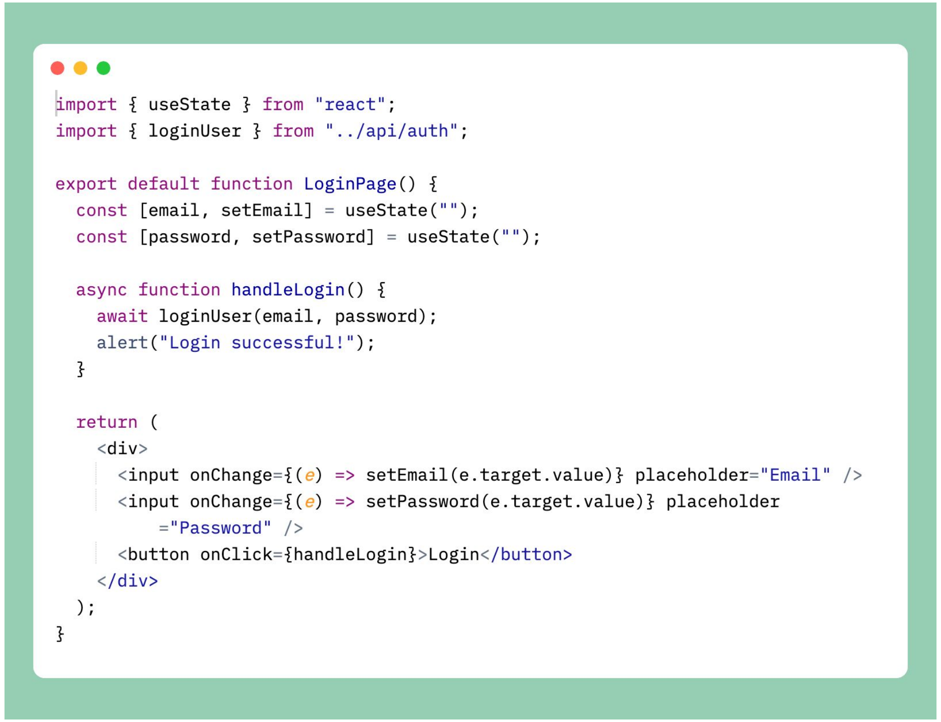
```
axios.defaults.baseURL = "http://localhost:3000/api";
axios.defaults.withCredentials = true; // allow cookies
```

The **withCredentials** option ensures cookies (refresh tokens) are sent automatically.

Implementing Login Function in React



```
import axios from "axios";

export async function loginUser(email: string, password: string) {
  const res = await axios.post("/auth/login", { email, password });
  localStorage.setItem("accessToken", res.data.accessToken);
  return res.data;
}
```

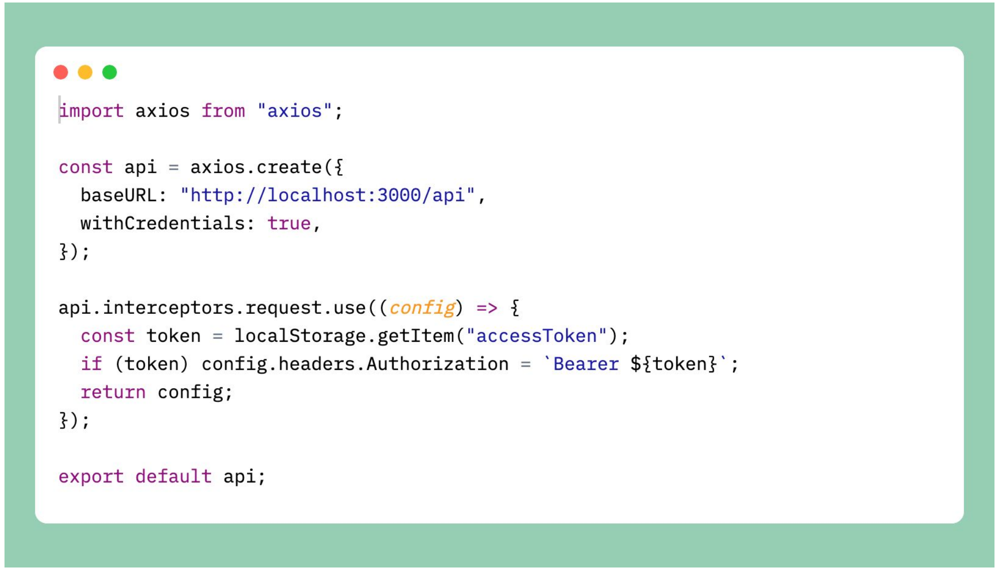


```
import { useState } from "react";
import { loginUser } from "../api/auth";

export default function LoginPage() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  async function handleLogin() {
    await loginUser(email, password);
    alert("Login successful!");
  }

  return (
    <div>
      <input onChange={(e) => setEmail(e.target.value)} placeholder="Email" />
      <input onChange={(e) => setPassword(e.target.value)} placeholder
          ="Password" />
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
```

- Stores **accessToken** locally after login.
- Refresh token is managed by the backend through HTTP-only cookie.

Attaching Access Token to Requests

- Automatically adds **Authorization** header for each request.
- Centralized approach for protected routes.
- Reduces repetitive code across components.



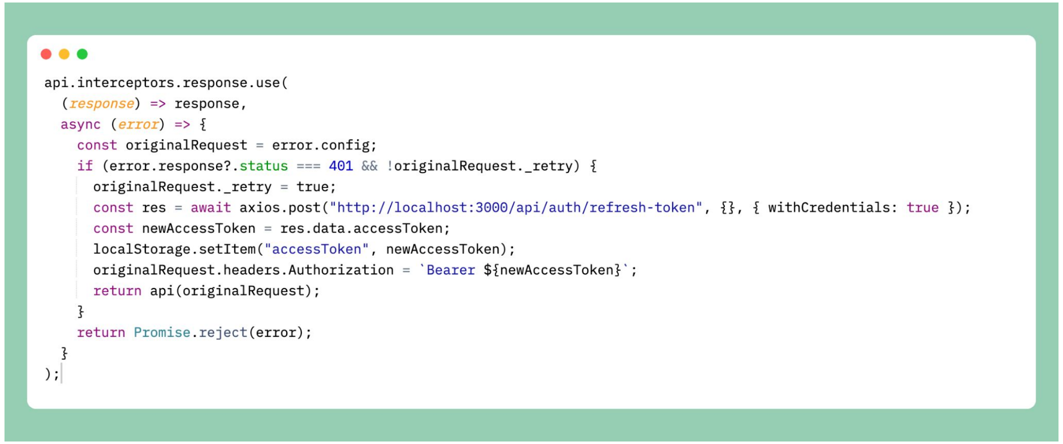
```javascript
import axios from "axios";

const api = axios.create({
  baseURL: "http://localhost:3000/api",
  withCredentials: true,
});

api.interceptors.request.use((config) => {
  const token = localStorage.getItem("accessToken");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

export default api;
```

Handling Token Expiration Automatically

- Intercepts failed requests (**401 Unauthorized**).
- Automatically requests a new access token using refresh token cookie.
- Retries the original API call without logging the user out.

Result:

Smooth, persistent session for users without re-login.



```
api.interceptors.response.use(
  (response) => response,
  async (error) => {
    const originalRequest = error.config;
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;
      const res = await axios.post("http://localhost:3000/api/auth/refresh-token", {}, { withCredentials: true });
      const newAccessToken = res.data.accessToken;
      localStorage.setItem("accessToken", newAccessToken);
      originalRequest.headers.Authorization = `Bearer ${newAccessToken}`;
      return api(originalRequest);
    }
    return Promise.reject(error);
  }
);
```

What is Google Social Authentication?

Google Social Authentication allows users to log in using their Google account instead of creating a new username and password.

In a backend system, we do not trust the frontend directly. The frontend sends a **Google ID Token**, and the backend is responsible for:

- Verifying the token with Google
- Reading user information from the token
- Creating or finding the user in our database
- Returning our own application token (JWT)

This approach is secure, simple for users, and widely used in modern applications.

OAuth Client ID

OAuth Client ID is used to identify your app when users log in with Google.

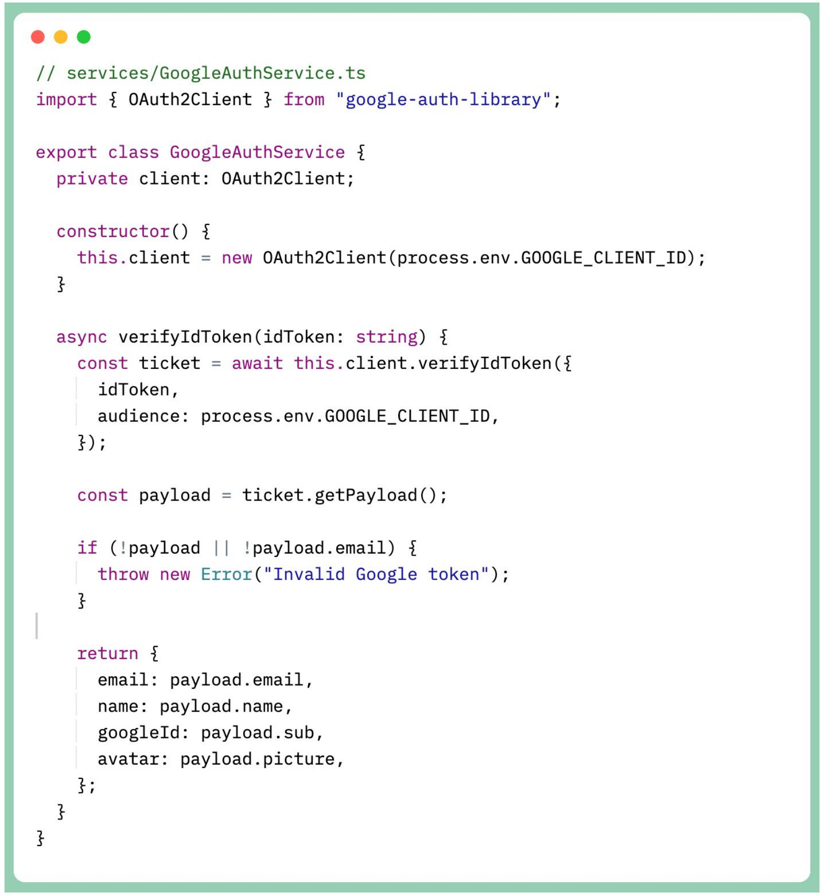
Steps:

1. Open **Google Cloud Console**
2. Create or select a project
3. Go to **OAuth consent screen** → choose **External** → fill basic info
4. Go to **Credentials** → Create **OAuth Client ID**
5. Select **Web Application**
6. Add redirect **URI** (example: **http://localhost:3000**)
7. Create and copy the **Client ID**

The Client ID is used by the frontend and backend to validate Google login requests. The Client Secret must stay on the backend and should never be exposed.

Google Auth Service

The service layer handles all Google-related authentication logic. This keeps controllers clean and focused only on HTTP handling.

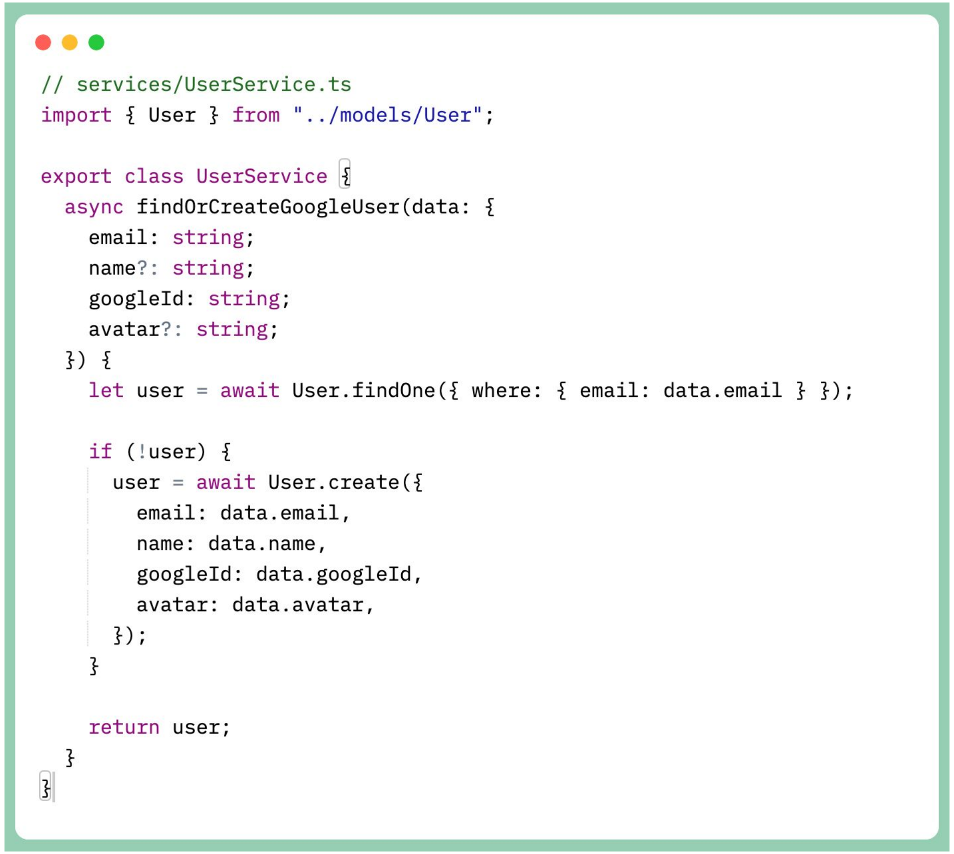


```ts
// services/GoogleAuthService.ts
import { OAuth2Client } from "google-auth-library";

export class GoogleAuthService {
  private client: OAuth2Client;

  constructor() {
    this.client = new OAuth2Client(process.env.GOOGLE_CLIENT_ID);
  }

  async verifyIdToken(idToken: string) {
    const ticket = await this.client.verifyIdToken({
      idToken,
      audience: process.env.GOOGLE_CLIENT_ID,
    });

    const payload = ticket.getPayload();

    if (!payload || !payload.email) {
      throw new Error("Invalid Google token");
    }

    return {
      email: payload.email,
      name: payload.name,
      googleId: payload.sub,
      avatar: payload.picture,
    };
  }
}
```

User Service

After the Google token is verified, we need to check our database.

- If the user already exists ➔ log them in
- If not ➔ create a new user automatically



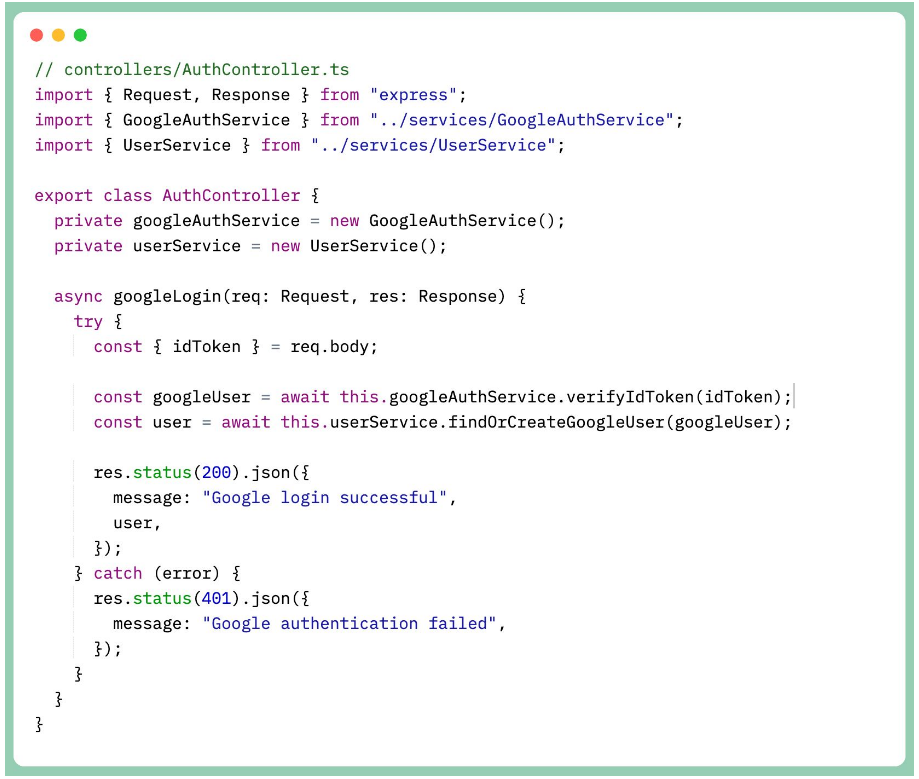
```
// services/UserService.ts
import { User } from "../models/User";

export class UserService {
  async findOrCreateGoogleUser(data: {
    email: string;
    name?: string;
    googleId: string;
    avatar?: string;
  }) {
    let user = await User.findOne({ where: { email: data.email } });

    if (!user) {
      user = await User.create({
        email: data.email,
        name: data.name,
        googleId: data.googleId,
        avatar: data.avatar,
      });
    }

    return user;
  }
}
```

Controller

The controller connects HTTP requests with our services.

Its responsibilities:

- Read request body
- Call services
- Return response

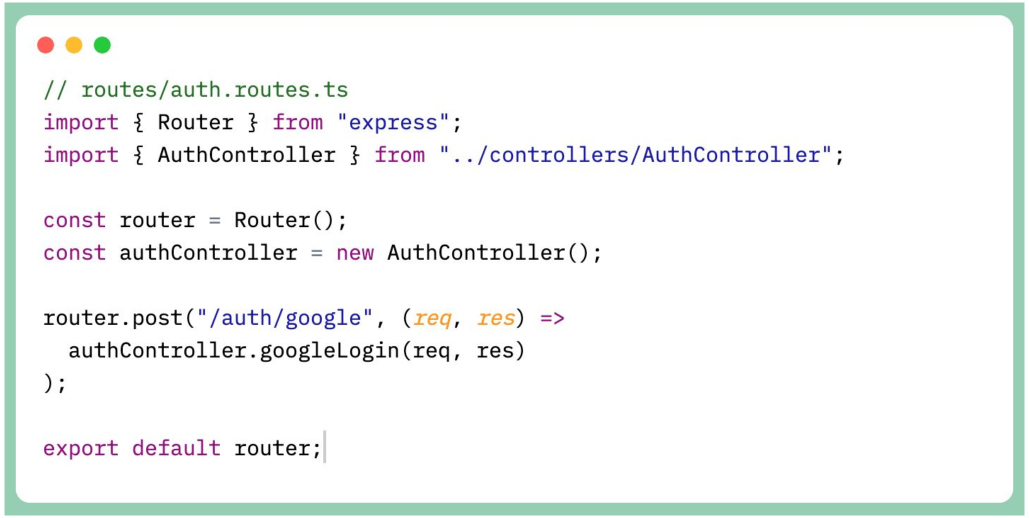


```ts
// controllers/AuthController.ts
import { Request, Response } from "express";
import { GoogleAuthService } from "../services/GoogleAuthService";
import { UserService } from "../services/UserService";

export class AuthController {
  private googleAuthService = new GoogleAuthService();
  private userService = new UserService();

  async googleLogin(req: Request, res: Response) {
    try {
      const { idToken } = req.body;

      const googleUser = await this.googleAuthService.verifyIdToken(idToken);
      const user = await this.userService.findOrCreateGoogleUser(googleUser);

      res.status(200).json({
        message: "Google login successful",
        user,
      });
    } catch (error) {
      res.status(401).json({
        message: "Google authentication failed",
      });
    }
  }
}
```

Route and Final Flow

Final Request Flow:

1. Frontend sends **Google ID Token**
2. Backend verifies token with Google
3. User is found or created
4. Backend returns a successful login response



```
// routes/auth.routes.ts
import { Router } from "express";
import { AuthController } from "../controllers/AuthController";

const router = Router();
const authController = new AuthController();

router.post("/auth/google", (req, res) =>
  authController.googleLogin(req, res)
);

export default router;
```

Exercise

- Enhance your existing Blog App by implementing secure user **authentication** and **role-based authorization**.
- Exercise Objectives:
  - Implement user registration and login system using hashed passwords (e.g., bcrypt).
  - Use **JWT tokens** to manage user sessions and persist authentication between SSR requests.
  - Create role-based access control with at least two roles:
    - **User →** can read and comment on posts.
    - **Admin →** can create, edit, or delete posts.

Thank you