ReactJS-> Okta -> NestJS

The redirect happens on your FRONTEND (e.g., your React, Angular, or Vue app).

Your NestJS backend **does not** handle the redirect. Its only job is to **validate the token** *after* your frontend successfully gets it from Okta.

Think of it this way:

- **Frontend:** The "receptionist" that sends the user to the "passport office" (Okta).
- **Okta:** The "passport office" that checks the user's ID and gives them a "passport" (a JWT token).
- **NestJS Backend:** The "border agent" who checks the passport (JWT token) on every API request before letting the user access data (like `/employees`).

The Visual Flow (Step-by-Step)

Here is the entire process from start to finish.

1. **Frontend:** The user clicks a "Login" button in your React/Angular/Vue app.
2. **Frontend -> Okta:** Your frontend app *redirects* the user's browser to your special Okta login page (e.g., `https://dev-123.okta.com/authorize?...`).
3. **Okta:** The user sees the Okta login form. They enter their username and password *directly into Okta*. Your app **never** sees their password.
4. **Okta -> Frontend:** After a successful login, Okta redirects the user *back* to your frontend, to a special URL you configured (e.g., `http://localhost:3000/callback`). This redirect includes a temporary, one-time-use **`code`**.
5. **Frontend -> Okta (Again):** Your frontend *in the background* sends this `code` to Okta's `/token` endpoint. Okta verifies the code and sends back the real **JWT Access Token**.
6. **Frontend -> NestJS: FINALLY!** Your frontend now has the token. It makes a request to your NestJS backend to `GET /employees` and puts the token in the header: `Authorization: Bearer [THE_JWT_TOKEN]`
7. **NestJS -> Okta:** Your NestJS app receives the token. It uses the `passport-jwt` strategy to check the token's signature. It contacts your Okta `Issuer URL` in the background to get the public keys and verify the token is 100% authentic and wasn't tampered with.
8. **NestJS:** The token is valid! The `@UseGuards(AuthGuard('jwt'))` decorator allows the request to proceed.
9. **NestJS -> Frontend:** Your `EmployeesController` runs, gets the employee data, and sends it back to the frontend as a normal JSON response.

Step 1: Okta Setup (The "Subscription/Key")

You don't need a paid subscription to start.

1. **Sign Up:** Go to `developer.okta.com` and sign up for a **free Okta Developer Account**.
2. **Create an App:** In your Okta admin dashboard, go to "Applications" -> "Create App Integration".
3. **Choose OIDC:** Select **OIDC - OpenID Connect** as the sign-in method.
4. **Choose App Type:** Select **Single-Page Application (SPA)**. This is crucial. It tells Okta you have a frontend app that will be handling the login flow.
5. **Configure:**
    - **Sign-in redirect URIs:** Tell Okta where it's allowed to send the user back to after they log in. For local development, add `http://localhost:3000/callback`.
6. **Get Your "Keys":** After you save, Okta will give you two critical pieces of information:
    - **Client ID:** You will use this in your **Frontend** (React/Vue/Angular) app.
    - **Issuer URL:** You will use this in your **NestJS Backend**. It looks like `https://dev-12345.okta.com/oauth2/default`.

Step 2: NestJS Backend Code (Validating the Token)

Your NestJS code is surprisingly simple. Its only job is to guard the `/employees` route.

You'll need to install a new package to help: `passport-jwt`. The `jwt.strategy.ts` is the most important file. It's different from the previous example because it doesn't use a local secret. Instead, it fetches the *public keys* from your Okta **Issuer URL** to verify the token.

To implement the Okta JWT authentication for your `EmployeesController`.

the complete structure:

1. **AuthModule**: Contains the `JwtStrategy` to validate Okta's tokens.
2. **JwtStrategy**: The core logic. It fetches Okta's public keys to verify the token signature.
3. **EmployeesModule**: Imports the `AuthModule` so its controller can be protected.
4. **EmployeesController**: The protected route you already have.
5. **AppModule**: The root module to import everything.

Controller code

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Controller('employees')
export class EmployeesController {

  // This is the magic!
  // 'AuthGuard('jwt')' automatically runs your JwtStrategy.
  // If the token is missing or invalid, NestJS sends a 401.
```

```
  // If it's valid, the method runs.
  @UseGuards(AuthGuard('jwt'))
  @Get()
  getAllEmployees(@Request() req) {
    // Because the guard passed, we can trust req.user
    console.log('User is:', req.user);

    // Go to the database and fetch employees...
    return [
      { id: 1, name: 'Alice', department: 'Engineering' },
      { id: 2, name: 'Bob', department: 'Sales' },
    ];
  }
}
```

npm install @nestjs/passport passport passport-jwt jwks-rsa

npm install -D @types/passport-jwt

The JWT Strategy (Okta Verifier)

```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { passportJwtSecret } from 'jwks-rsa';

// --- IMPORTANT ---
// Replace these with your actual Okta details.
// Find them in your Okta application dashboard.
const OKTA_ISSUER_URL = 'https://YOUR_OKTA_DOMAIN/oauth2/default';
const OKTA_AUDIENCE = 'api://default'; // This is the default. Check your Okta
app.

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      // 1. Tell Passport how to find the token (from the Authorization
header)
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

      // 2. We need to fetch the public key from Okta to verify the token.
      // This helper (passportJwtSecret) does it for us.
      secretOrKeyProvider: passportJwtSecret({
        cache: true,
        rateLimit: true,
```

```
        jwksRequestsPerMinute: 5,
        // 3. This is the URL where Okta publishes its public keys
        jwksUri: `${OKTA_ISSUER_URL}/v1/keys`,
      }),

      // 4. Validate the issuer and audience (extra security)
      issuer: OKTA_ISSUER_URL,
      audience: OKTA_AUDIENCE,
      algorithms: ['RS256'], // Okta uses RS256
    });
  }

  /**
   * This method runs *after* the token signature is successfully verified
   * @param payload The decoded JSON from the JWT
   * @returns The user object to be attached to req.user
   */
  validate(payload: any) {
    // The payload is the decoded JSON. We can trust it now.
    // Whatever we return here is attached to the request as `req.user`
    return {
      userId: payload.sub, // 'sub' is the standard JWT claim for the user's
ID
      username: payload.uid, // Okta often puts the username in 'uid'
      roles: payload.groups || [], // Add roles if they are in your token
    };
  }
}
```