upGrad ENTERPRISE

Day 7 — Authentication (JWT + RBAC) + Okta

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Previous day:**
  - REST principles & HTTP status codes
  - Designing resource-oriented APIs (URIs, nesting, filtering, pagination)
  - API versioning strategies
  - API documentation basics (Swagger / OpenAPI)
  - API testing with Postman (environments, requests, tests, collections)
- Day 7
  - Explain the difference between **authentication and authorization**
  - Implement **JWT-based authentication** in a NestJS app using Sequelize and Oracle
  - Protect API endpoints with **JWT guards**
  - Add role-based authorization, **Role-based access control** (RBAC)
  - Understand **refresh tokens and expiry strategy**
  - Describe how **Okta** works and create a developer account

---

1. Authentication vs Authorization

Concept	Description	Example
Authentication (AuthN)	Verifies *who* the user is.	Login using username & password.
Authorization (AuthZ)	Determines *what* the user can do.	Admins can delete users; others cannot.

In simple words:

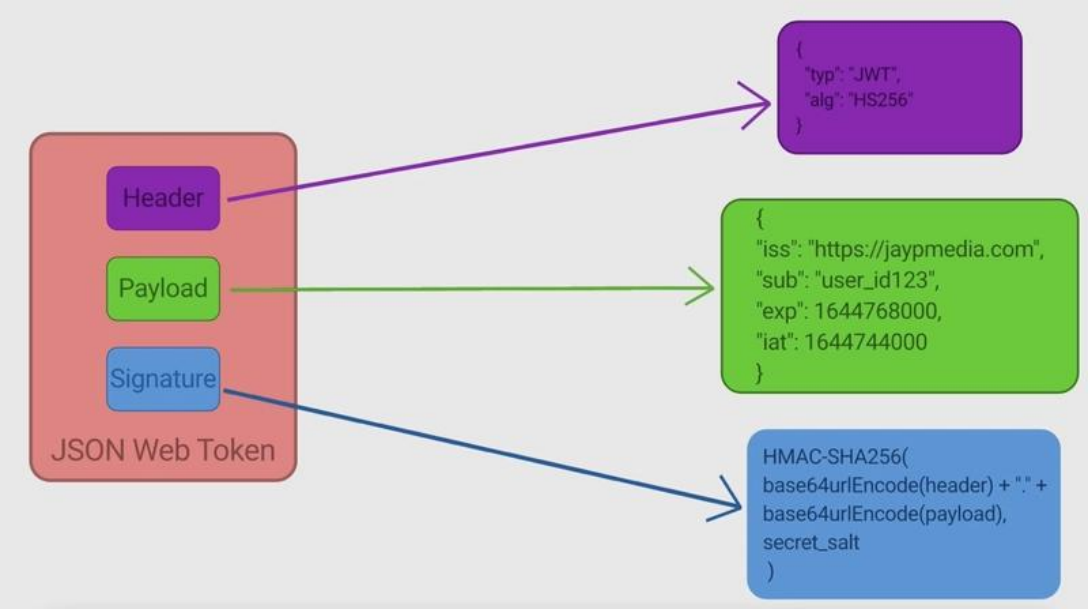
"Authentication proves identity, Authorization defines access/permissions based on identity/roles/claims."

---

2. Introduction to JSON Web Tokens (JWT)

- A **JWT** is a secure, self-contained token for authentication.
- It is base64url encoded.
- Signed (HMAC or RSA/ECDSA). Integrity + authenticity. Do not put secrets in payload.
- Use short expiry for access tokens (e.g. 5–15 min) + refresh token (longer lived, stored securely) to obtain new access tokens.

Structure:



`Header.Payload.Signature`

- **Header:** algorithm & token type
  → `{"alg":"HS256","typ":"JWT"}`
- **Payload:** claims about the user
  → `{"sub":1,"email":"john@example.com","roles":["admin"]}`
- **Signature:** digital proof to verify authenticity. sign(header + "." + payload) with secret or private key

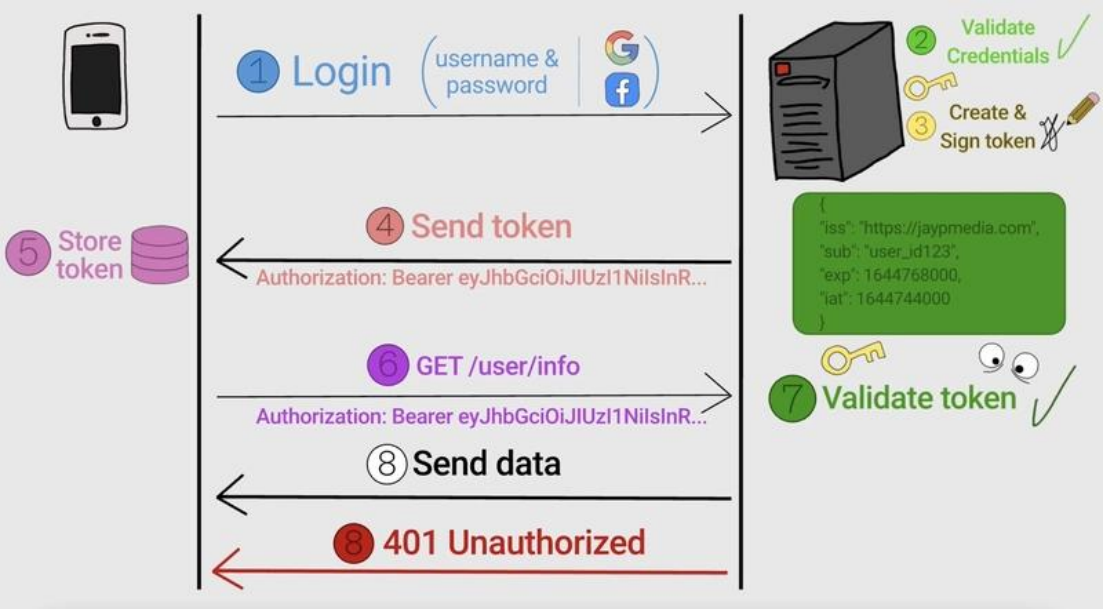
Why JWT?

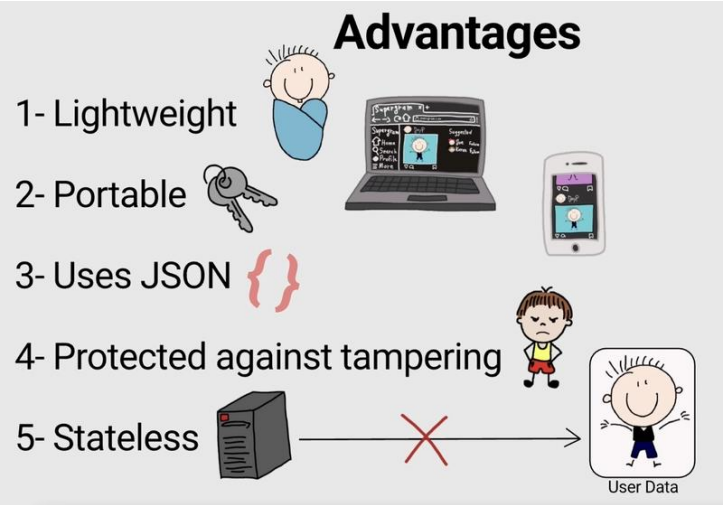
- Compact, stateless, easy to verify
- Works across languages and microservices
- Supports expiration and custom claims

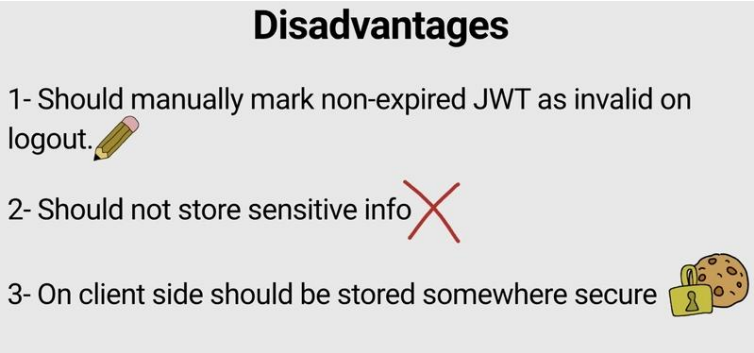
Typical Flow:

1. User logs in → Server verifies credentials
2. Server issues JWT (signed with secret key)

3. Client stores token securely (e.g., LocalStorage / cookie)
4. Client sends token with every request (`Authorization: Bearer <token>`)
5. Server verifies token & processes the request







3. Implementing JWT-based Authentication in NestJS

Our **Employees API** (from previous days) already uses:

- **NestJS**
- **Sequelize ORM**
- **Oracle database**

Now, we'll add a **Users module** to handle login & authentication, and secure the Employees API using JWT.

Folder Structure

```
src/
│
├── main.ts
├── app.module.ts
│
├── config/
│   ├── database.config.ts              # Sequelize (Oracle) DB connection
config
│   └── swagger.config.ts               # Swagger setup (from previous day)
│
├── models/
│   ├── employee.model.ts               # Existing employee entity
│   └── user.model.ts                   # New User entity for
authentication
│
├── modules/
│   │
│   ├── employees/
│   │   ├── employees.module.ts
│   │   ├── employees.service.ts
│   │   ├── employees.controller.ts
│   │   └── dto/
│   │       ├── create-employee.dto.ts
│   │       └── update-employee.dto.ts
│   │
│   ├── users/                          # NEW Users module
│   │   ├── users.module.ts
│   │   ├── users.service.ts
│   │   ├── users.controller.ts         # (optional - for future
registration endpoint)
│   │   └── dto/
│   │       ├── create-user.dto.ts
│   │       └── login-user.dto.ts
│
├── auth/                               # NEW Auth module
│   ├── auth.module.ts
│   ├── auth.service.ts
│   ├── auth.controller.ts
│   ├── jwt.strategy.ts
│   ├── jwt-auth.guard.ts
│   ├── roles.decorator.ts
│   ├── roles.guard.ts
```

```
│   └── interfaces/
│       └── jwt-payload.interface.ts    # optional interface for JWT
payload
│
├── common/
│   ├── filters/                        # future: custom exceptions or
filters
│   ├── interceptors/                   # future: response transform,
logging, etc.
│   └── pipes/
│
└── utils/
    └── bcrypt.util.ts                  # optional helper for hashing &
verifying passwords
```

Explanation of Key Folders

Folder / File	Description
config/	Centralized configuration for database (Sequelize + Oracle), Swagger setup, environment variables.
models/	Sequelize entities — `Employee` (existing) and new `User` (for login/auth).
modules/employees/	Existing CRUD API module for Employee data — now protected by JWT.
modules/users/	Handles creation and lookup of `User` records for login authentication.
auth/	Complete JWT-based authentication layer (service, controller, guards, decorators).
common/	Reserved for reusable interceptors, filters, pipes — ensures clean architecture.
utils/	Optional utility functions, such as password hashing, token utilities, etc.

Typical Import Relationships

- `AppModule` imports:
  - `SequelizeModule.forRoot(database.config)`
  - `EmployeesModule`
  - `UsersModule`
  - `AuthModule`
- `AuthModule` imports:
  - `UsersModule` (to validate users and issue JWTs)
- `EmployeesModule` controllers use:
  - `JwtAuthGuard` & `RolesGuard` from `auth/`

3.1 Database Model — User Entity

We will create a new **users** table to store login credentials (separate from employees).

File: `src/models/user.model.ts`

```
import { Table, Column, Model, DataType, PrimaryKey, AutoIncrement, Unique
} from 'sequelize-typescript';

@Table({ tableName: 'USERS', timestamps: true })
export class User extends Model<User> {
  @PrimaryKey
  @AutoIncrement
  @Column(DataType.INTEGER)
  id!: number;

  @Unique
  @Column({ type: DataType.STRING(100), allowNull: false })
  email!: string;

  @Column({ type: DataType.STRING(100), allowNull: false })
  password!: string;

  @Column({ type: DataType.STRING(50), allowNull: false, defaultValue:
'user' })
  role!: string;  // 'admin' or 'user'
}
```

3.2 DTO for User Login and Registration

File: `src/modules/users/dto/login-user.dto.ts`

```
import { IsEmail, IsString } from 'class-validator';

export class LoginUserDto {
  @IsEmail()
  email!: string;

  @IsString()
  password!: string;
}
```

3.3 Users Service

File: `src/modules/users/users.service.ts`

```
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/sequelize';
import { User } from '../../models/user.model';
import * as bcrypt from 'bcrypt';

@Injectable()
```

```
export class UsersService {
  constructor(@InjectModel(User) private userModel: typeof User) {}

  async create(email: string, password: string, role: string = 'user'):
Promise<User> {
    const hash = await bcrypt.hash(password, 10);
    return this.userModel.create({ email, password: hash, role });
  }

  async findByEmail(email: string): Promise<User | null> {
    return this.userModel.findOne({ where: { email } });
  }
}
```

3.4 Auth Module Setup

Install dependencies:

```
npm install @nestjs/jwt @nestjs/passport passport passport-jwt bcrypt
npm install -D @types/passport-jwt
```

3.5 Auth Module Files

auth.module.ts

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { UsersModule } from '../modules/users/users.module';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { JwtStrategy } from './jwt.strategy';

@Module({
  imports: [
    UsersModule,
    PassportModule,
    JwtModule.register({
      secret: process.env.JWT_SECRET || 'super_secret_key',
      signOptions: { expiresIn: '15m' },
    }),
  ],
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  exports: [AuthService],
})
export class AuthModule {}
```

---

auth.service.ts

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
```

```
import { UsersService } from '../modules/users/users.service';
import * as bcrypt from 'bcrypt';

@Injectable()
export class AuthService {
  constructor(private usersService: UsersService, private jwtService:
JwtService) {}

  async validateUser(email: string, password: string): Promise<any> {
    const user = await this.usersService.findByEmail(email);
    if (!user) return null;
    const valid = await bcrypt.compare(password, user.password);
    if (!valid) return null;
    const { password: _p, ...result } = user.toJSON();
    return result;
  }

  async login(email: string, password: string) {
    const user = await this.validateUser(email, password);
    if (!user) throw new UnauthorizedException('Invalid credentials');
    const payload = { sub: user.id, email: user.email, role: user.role };
    return {
      accessToken: this.jwtService.sign(payload),
      expiresIn: '15m',
      user,
    };
  }
}
```

---

auth.controller.ts

```
import { Controller, Post, Body } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LoginUserDto } from '../modules/users/dto/login-user.dto';

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @Post('login')
  login(@Body() loginDto: LoginUserDto) {
    return this.authService.login(loginDto.email, loginDto.password);
  }
}
```

---

jwt.strategy.ts

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

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
```

```
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: process.env.JWT_SECRET || 'super_secret_key',
    });
  }

  async validate(payload: any) {
    return { userId: payload.sub, email: payload.email, role: payload.role
};
  }
}
```

3.6 Guards and Role Decorator

jwt-auth.guard.ts

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

roles.decorator.ts

```
import { SetMetadata } from '@nestjs/common';
export const ROLES_KEY = 'roles';
export const Roles = (...roles: string[]) => SetMetadata(ROLES_KEY, roles);
```

roles.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { ROLES_KEY } from './roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles =
this.reflector.getAllAndOverride<string[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) return true;
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.includes(user.role);
  }
}
```

3.7 Securing Employees API

Modify the `EmployeesController` to use JWT + RBAC.

File: `src/modules/employees/employees.controller.ts`

```
import { Controller, Get, Post, Delete, UseGuards, Param, Body } from
'@nestjs/common';
import { EmployeesService } from './employees.service';
import { JwtAuthGuard } from '../../auth/jwt-auth.guard';
import { RolesGuard } from '../../auth/roles.guard';
import { Roles } from '../../auth/roles.decorator';

@Controller('employees')
@UseGuards(JwtAuthGuard, RolesGuard)
export class EmployeesController {
  constructor(private employeesService: EmployeesService) {}

  @Get()
  @Roles('user', 'admin')
  findAll() {
    return this.employeesService.findAll();
  }

  @Post()
  @Roles('admin')
  create(@Body() dto) {
    return this.employeesService.create(dto);
  }

  @Delete(':id')
  @Roles('admin')
  remove(@Param('id') id: string) {
    return this.employeesService.remove(Number(id));
  }
}
```

---

4. Role-Based Access Control (RBAC)

- Assign each user a role (`user`, `admin`, etc.)
- Store role in JWT payload
- Use `@Roles()` decorator & `RolesGuard` to restrict routes

- RBAC maps users → roles → permissions. Roles are simple (e.g., `admin`, `manager`, `user`).

- Implementation patterns:

  - Add `roles` claim to JWT at issuance time. On protected routes, check for required role(s).
  - In NestJS use a `@Roles()` decorator + `RolesGuard` to centralize checks.
  - For fine-grained permissions, use attribute-based (ABAC) or permission lists.

- Keep RBAC logic centralized in a guard/middleware rather than scattered in controllers.

Example:

```
@Get('admin-only')
@Roles('admin')
getSensitiveData() { ... }
```

---

5. Refresh Tokens & Token Expiry

- Access token: short-lived (5–15 minutes). Stored in memory or secure cookie. Used to call APIs.

- Refresh token: long-lived (days/weeks), stored in secure `HttpOnly` cookie (browser) or secure storage (mobile). Used to request new access tokens.

- Strategy:

  - On login: issue access token + refresh token.
  - When access token expires: client calls `POST /auth/refresh` with refresh token (cookie) → server verifies refresh token (check DB/blacklist) → issue new access token (and optionally new refresh token).
  - Logout/revocation: delete refresh token server-side or mark revoked.

- Security: protect refresh endpoints, store refresh tokens hashed server-side if stored in DB.

In short -

- **Access Token:** short-lived (e.g., 15 min)
- **Refresh Token:** long-lived (e.g., 7 days)
- When access token expires, client calls `/auth/refresh` using the refresh token.
- Always store refresh tokens securely (DB + HttpOnly cookie).

Recommended flow for enterprise:

```
Login → Access Token (15 min)
       ↳ Refresh Token (7 days)
       ↳ /auth/refresh → new Access Token
```

---

6. Okta — Theory and Backend Integration

Okta is a cloud-based Identity Provider (IdP) offering:

- User authentication & management
- OAuth2 / OpenID Connect flows
- SSO (Single Sign-On)
- MFA (Multi-Factor Authentication)

How Okta works with backends

- Use Okta as Authorization Server (OAuth2 / OIDC). Your backend accepts IdP-issued tokens (ID tokens for login, access tokens for APIs).

- Typical flows:

  - **Authorization Code (with PKCE)** for SPAs / mobile — recommended.
  - **Authorization Code** for server-side web apps.
  - **Client Credentials** for machine-to-machine.

- Backend validates tokens (via JWKS or introspection) and trusts claims (e.g., `sub`, `email`, `groups`, `roles`). Many SDKs handle validation.

In short -

1. Backend trusts Okta as an **Authorization Server**.
2. Clients authenticate with Okta.
3. Okta issues **ID token** and **Access token**.
4. Backend verifies tokens (using Okta JWKS endpoint).
5. Authenticated user data (email, roles, etc.) is used in your API.

High-level steps to integrate later

1. Register app in Okta developer console → get Client ID, Client Secret (if applicable), redirect URIs, scopes.
2. Configure authorization server and scopes in Okta.
3. In backend, validate tokens using Okta's JWKS endpoint (fetch public keys) or using Okta SDKs.
4. Use claims to map to your app roles or create users on first login (just-in-time provisioning).

How to create an Okta developer account

1. Visit https://developer.okta.com
2. Click **Get Started for Free**
3. Verify email and log in
4. In Okta Admin Console → **Applications** → **Create App Integration**
5. Choose type (e.g. Web or SPA), configure redirect URIs
6. Note down **Client ID**, **Client Secret**, and **Issuer URL**
7. You'll use these values when integrating in a later session

---

7. Tips & Best Practices

- Never store passwords in plain text — always hash using bcrypt.
- Never include sensitive data inside JWT payloads.
- Protect routes using `JwtAuthGuard` and `RolesGuard`.
- Store JWT secrets and DB credentials in `.env`.
- Rotate refresh tokens regularly.
- Use HTTPS in all environments.

---

8. Summary

Concept	Key Takeaways
JWT Auth	Stateless authentication; token stored client-side
RBAC	Control access based on roles
NestJS Integration	`JwtModule`, `AuthGuard`, `Passport`, `RolesGuard`
Okta	Enterprise Identity Provider for OAuth2/OIDC
Security Basics	Always hash passwords and protect routes

Coverage

- Explain the difference between **authentication and authorization**
- Implement **JWT-based authentication** in a NestJS app using Sequelize and Oracle
- Protect API endpoints with **JWT guards**
- Add role-based authorization, **Role-based access control** (RBAC)
- Understand **refresh tokens and expiry strategy**
- Describe how **Okta** works and create a developer account