

OAuth2.0 JWT Spring Boot

Develop a small demo springboot app and please mention Step by step hands on code to implement OAuth2.0 with springboot and 2 common endpoints with example with JWT and bearer token .

Below is a **small, hands-on Spring Boot demo** showing OAuth2.0 using JWT + **Bearer Token**, written **step by step**, with **working code**, **two common endpoints**, and **clear flow**.

I'll keep it **simple, clean, and practical**, suitable for a **Java + Spring Boot developer** like you.

OAuth2.0 with JWT – Spring Boot Demo (Hands-On)

Tech Stack

- Java 17+
 - Spring Boot 3.x
 - Spring Security 6
 - OAuth2 Resource Server
 - JWT (Bearer Token)
 - Maven
-

Architecture (Simple)

pgsql

Client → [Login](#) → JWT Token

Client → API Request ([Authorization](#): Bearer <JWT>)

Spring [Security](#) → Validates JWT → Allows API Access

We will build:

1. Auth Controller → generates JWT
2. Protected API → accessible only with Bearer token
3. Public API → accessible without token

STEP 1: Create Spring Boot Project

Using Spring Initializr

Dependencies

- Spring Web
- Spring Security
- OAuth2 Resource Server

STEP 2: Maven pom.xml

xml

```
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    </dependency>

</dependencies>
```

STEP 3: Application Properties

application.yml

yaml

```
server:  
port: 8080  
  
spring:  
security:  
oauth2:  
resourceServer:  
jwt:  
issuer-uri: http://localhost:8080
```

We'll use **self-issued JWT** for demo purposes.

STEP 4: JWT Utility (Token Creation)

JwtUtil.java

```
java  
  
package com.demo.security;  
  
import java.time.Instant;  
import java.util.Map;  
  
import org.springframework.security.oauth2.jwt.Jwt;  
import org.springframework.security.oauth2.jwt.JwtClaimsSet;  
import org.springframework.security.oauth2.jwt.JwtEncoder;  
import org.springframework.stereotype.Component;  
  
@Component  
public class JwtUtil {  
  
    private final JwtEncoder jwtEncoder;  
  
    public JwtUtil(JwtEncoder jwtEncoder) {  
        this.jwtEncoder = jwtEncoder;  
    }  
  
    public String generateToken(String username) {  
        JwtClaimsSet claims = JwtClaimsSet.builder()
```

```

    .issuer("http://localhost:8080")
    .issuedAt(Instant.now())
    .expiresAt(Instant.now().plusSeconds(3600))
    .subject(username)
    .claim("scope", "USER")
    .build();

    Jwt jwt = jwtEncoder.encode(
        org.springframework.security.oauth2.jwt.JwtEncoderParameters.from(claims)
    );

    return jwt.getTokenValue();
}
}

```

STEP 5: Security Configuration

SecurityConfig.java

java

```

package com.demo.config;

import java.security.KeyPair;
import java.security.KeyPairGenerator;

import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.JWKSet;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.oauth2.jwt.*;

@Configuration
public class SecurityConfig {

    @Bean

```

```

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/auth/**", "/public").permitAll()
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(oauth2 ->
            oauth2.jwt(Customizer.withDefaults())
        );
}

return http.build();
}

// =====JWT Beans =====

@Bean
JwtEncoder jwtEncoder() {
    return new NimbusJwtEncoder(jwkSource());
}

@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(rsaKey().toRSAPublicKey()).build();
}

@Bean
ImmutableJWKSet<com.nimbusds.jose.jwk.JWK> jwkSource() {
    JWKSet jwkSet = new JWKSet(rsaKey());
    return new ImmutableJWKSet<>(jwkSet);
}

private RSAKey rsaKey() {
    KeyPair keyPair = generateRsaKey();
    return new RSAKey.Builder((java.security.interfaces.RSAPublicKey) keyPair.getPublic())
        .privateKey((java.security.interfaces.RSAPrivateKey) keyPair.getPrivate())
        .keyID("demo-key")
        .build();
}

private KeyPair generateRsaKey() {
}

```

```
try {
    KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
    generator.initialize(2048);
    return generator.generateKeyPair();
} catch (Exception e) {
    throw new IllegalStateException(e);
}
}
```

STEP 6: Authentication Controller (Token Generation)

AuthController.java

java

```
package com.demo.controller;

import com.demo.security.JwtUtil;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@RestController
@RequestMapping("/auth")
public class AuthController {

    private final JwtUtil jwtUtil;

    public AuthController(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @PostMapping("/login")
    public Map<String, String> login(@RequestParam String username) {

        String token = jwtUtil.generateToken(username);

        return Map.of(
            "token", token,
            "type", "Bearer"
        );
    }
}
```

```
    );  
}  
}
```

█ STEP 7: API Controller (2 Common Endpoints)

DemoController.java

```
java
```

```
package com.demo.controller;  
  
import org.springframework.security.core.annotation.AuthenticationPrincipal;  
import org.springframework.security.oauth2.jwt.Jwt;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
public class DemoController {  
  
    // PUBLIC ENDPOINT  
    @GetMapping("/public")  
    public String publicApi() {  
        return "This is a PUBLIC API";  
    }  
  
    // PROTECTED ENDPOINT  
    @GetMapping("/secure")  
    public String secureApi(@AuthenticationPrincipal Jwt jwt) {  
        return "Hello " + jwt.getSubject() + ", this is a SECURE API";  
    }  
}
```

█ STEP 8: Test Using CURL / Postman

1 Get Token

```
bash
```

```
POST http://localhost:8080/auth/login?username=rupesh
```

Response

json

```
{  
  "token": "eyJraWQiOijkZW1vLWtleSIsInR5cCI6IkpXVCJ9...",  
  "type": "Bearer"  
}
```

2 Call Public API (No Token)

bash

```
GET http://localhost:8080/public
```

 Works

3 Call Secure API (With Bearer Token)

bash

```
GET http://localhost:8080/secure
```

```
Authorization: Bearer eyJraWQiOijkZW1vLWtleSIsInR5cCI6IkpXVCJ9...
```

 Works

4 Call Secure API Without Token

 401 Unauthorized

OAuth2.0 Flow Summary

Step	Action
1	Client authenticates (/auth/login)
2	Server issues JWT
3	Client sends JWT as Bearer token
4	Spring Security validates JWT
5	Protected endpoint accessed

🎯 What You Learned

- OAuth2 Resource Server setup
- JWT creation & validation
- Bearer token authentication
- Public vs protected endpoints
- Spring Security 6 configuration

➡️ SOON NEXT (If You Want)

I can also show:

- OAuth2 with Keycloak
- OAuth2 Authorization Server
- Role-based access (@PreAuthorize)
- Refresh tokens
- BFF pattern with OAuth2

Just tell me 

Please also explain step by step code for Refresh tokens

Perfect 

I'll explain **both topics step by step with code and flow**, keeping it **simple, practical, and production-oriented**, and aligned with **Spring Boot 3 + OAuth2 + JWT**.

PART 1: Refresh Token – Step by Step (OAuth2 + JWT)

? Why Refresh Token?

- **Access Token (JWT)** → short-lived (e.g. 5–15 min)
 - **Refresh Token** → long-lived (e.g. days)
 - When access token expires → client uses refresh token to get a new one **without re-login**
-

Flow (Simple)

```
pgsql|
```

Login → Access Token + Refresh Token

API Call → Access Token

Token Expired → Refresh Token → New Access Token

STEP 1: Token Types

Token	Purpose	Stored At
Access Token	API access	Client memory
Refresh Token	Get new access token	Secure storage

STEP 2: Extend Token Model

TokenResponse.java

java

```
package com.demo.model;

public class TokenResponse {

    private String accessToken;
    private String refreshToken;
    private String tokenType = "Bearer";
    private long expiresIn;

    public TokenResponse(String accessToken, String refreshToken, long expiresIn) {
        this.accessToken = accessToken;
        this.refreshToken = refreshToken;
        this.expiresIn = expiresIn;
    }

    // getters
}
```

STEP 3: Refresh Token Storage (In-Memory Demo)

In real apps → DB / Redis

RefreshTokenStore.java

java

```
package com.demo.security;

import org.springframework.stereotype.Component;

import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;

@Component
public class RefreshTokenStore {
```

```
private final Map<String, String> refreshTokenMap = new ConcurrentHashMap<>();  
  
public String createRefreshToken(String username) {  
    String refreshToken = UUID.randomUUID().toString();  
    refreshTokenMap.put(refreshToken, username);  
    return refreshToken;  
}  
  
public String validateRefreshToken(String refreshToken) {  
    return refreshTokenMap.get(refreshToken);  
}  
  
public void deleteRefreshToken(String refreshToken) {  
    refreshTokenMap.remove(refreshToken);  
}  
}
```

STEP 4: Update JWT Utility

JwtUtil.java (Updated)

java

```
public String generateAccessToken(String username) {  
  
    JwtClaimsSet claims = JwtClaimsSet.builder()  
        .issuer("http://localhost:8080")  
        .issuedAt(Instant.now())  
        .expiresAt(Instant.now().plusSeconds(300)) // 5 min  
        .subject(username)  
        .claim("scope", "USER")  
        .build();  
  
    return jwtEncoder.encode(  
        JwtEncoderParameters.from(claims)  
        .getTokenValue());  
}
```

STEP 5: Login API → Issue Both Tokens

AuthController.java

```
java

@PostMapping("/login")
public TokenResponse login(@RequestParam String username) {

    String accessToken = jwtUtil.generateAccessToken(username);
    String refreshToken = refreshTokenStore.createRefreshToken(username);

    return new TokenResponse(
        accessToken,
        refreshToken,
        300
    );
}
```

STEP 6: Refresh Token API

AuthController.java

```
java

@PostMapping("/refresh")
public TokenResponse refresh(@RequestParam String refreshToken) {

    String username = refreshTokenStore.validateRefreshToken(refreshToken);

    if (username == null) {
        throw new RuntimeException("Invalid refresh token");
    }

    String newAccessToken = jwtUtil.generateAccessToken(username);

    return new TokenResponse(
        newAccessToken,
        refreshToken,
        300
    );
}
```

```
});  
}
```

🧪 Refresh Token Flow

1 Login

```
http
```

```
POST /auth/login?username=rupesh
```

2 API call fails (401 – token expired)

3 Refresh token

```
http
```

```
POST /auth/refresh?refreshToken=UUID
```

✓ New access token returned

🔒 Security Best Practices

- Refresh token **never sent to APIs**
- One refresh token per device
- Rotate refresh tokens
- Store refresh token securely (DB/Redis)

🧩 PART 2: BFF Pattern with OAuth2 (Backend For Frontend)

❓ What is BFF?

Each frontend has its own backend

```
nginx
```

Mobile App → Mobile BFF → APIs

Web App → Web BFF → APIs

🧠 Why BFF?

Problem	Solution
Token exposed to browser	BFF hides token
Multiple APIs	BFF aggregates
Security complexity	Centralized

🧠 BFF OAuth2 Flow

arduino

Frontend → BFF → OAuth2 Auth **Server**

BFF → Receives Tokens

BFF → Calls Backend APIs

Frontend NEVER sees token

🧱 STEP 1: BFF Responsibilities

- Handles OAuth2 login
- Stores access & refresh tokens
- Calls downstream APIs
- Exposes frontend-friendly APIs

🧱 STEP 2: BFF Security Config

`SecurityConfig.java (BFF)`

java

```
http
    .authorizeHttpRequests(auth -> auth
        .requestMatchers("/login", "/public").permitAll()
        .anyRequest().authenticated()
    )
    .oauth2Login(Customizer.withDefaults())
    .oauth2Client(Customizer.withDefaults());
```

■ STEP 3: OAuth2 Client Config

application.yml

```
yaml

spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: bff-client
            client-secret: secret
            scope: openid,profile
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
        provider:
          keycloak:
            issuer-uri: http://localhost:8081/realms/demo
```

■ STEP 4: Token Storage (Server Side)

java

```
@Component
public class BffTokenStore {

    private String accessToken;
    private String refreshToken;
```

```
public void store(String access, String refresh) {  
    this.accessToken = access;  
    this.refreshToken = refresh;  
}  
  
public String getAccessToken() {  
    return accessToken;  
}  
}
```

█ STEP 5: Call Backend API from BFF

java

```
@Service  
public class Backend ApiService {  
  
    private final RestTemplate restTemplate = new RestTemplate();  
    private final BffTokenStore tokenStore;  
  
    public Backend ApiService(BffTokenStore tokenStore) {  
        this.tokenStore = tokenStore;  
    }  
  
    public String callSecureApi() {  
  
        HttpHeaders headers = new HttpHeaders();  
        headers.setBearerAuth(tokenStore.getAccessToken());  
  
        HttpEntity<Void> entity = new HttpEntity<>(headers);  
  
        ResponseEntity<String> response =  
            restTemplate.exchange(  
                "http://backend-api/secure",  
                HttpMethod.GET,  
                entity,  
                String.class  
            );  
  
        return response.getBody();  
    }  
}
```

```
    }  
}
```

STEP 6: BFF Controller

```
java
```

```
@RestController  
@RequestMapping("/bff")  
public class BffController {  
  
    private final Backend ApiService service;  
  
    public BffController(Backend ApiService service) {  
        this.service = service;  
    }  
  
    @GetMapping("/dashboard")  
    public String dashboard() {  
        return service.callSecureApi();  
    }  
}
```

BFF Security Advantages

- No token in browser
- CSRF protected
- Centralized auth
- Easy API evolution

When to Use What?

Scenario	Use
SPA + mobile apps	BFF
Microservices	OAuth2 + JWT

Scenario	Use
High security	BFF + OAuth2
Simple API	JWT only

Next Level (Optional)

I can next show:

- ✓ Keycloak full setup
- ✓ Token rotation
- ✓ Redis-based refresh tokens
- ✓ Gateway + BFF
- ✓ Diagram + sequence flow

Just tell me 