

Spring Boot OAuth2 example

spring boot oauth2 example from scratch

Here's a **complete step-by-step example** of setting up **Spring Boot OAuth2** from scratch using **Authorization Code Flow** with **GitHub** as the OAuth2 provider.

✓ Use Case:

You will build a Spring Boot app where:

- Users can log in using their GitHub accounts.
- OAuth2 login is handled via Spring Security.

🔧 Step 1: Create Spring Boot Project

You can generate from <https://start.spring.io>:

- **Dependencies:**
 - Spring Web
 - Spring Security
 - OAuth2 Client
 - Thymeleaf (optional for UI)

📁 Directory Structure

CSS

src/

```
|— main/
|   |— java/
|   |   |— com.example.oauth2/
|   |       |— Oauth2DemoApplication.java
|   |       |— controller/
|   |           |— HomeController.java
|   |— resources/
|   |— application.yml
```

```
|   └── templates/
|       ├── index.html
|       └── secured.html
```

Step 2: Main Class

```
java

// OAuth2DemoApplication.java
package com.example.oauth2;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class OAuth2DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(OAuth2DemoApplication.class, args);
    }
}
```

Step 3: Controller

```
java

// HomeController.java
package com.example.oauth2.controller;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.oidc.user.OidcUser;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

    @GetMapping("/")
    public String index() {
```

```

        return "index";
    }

    @GetMapping("/secured")
    public String secured(Model model, @AuthenticationPrincipal OidcUser principal) {
        model.addAttribute("name", principal.getFullName());
        model.addAttribute("email", principal.getEmail());
        return "secured";
    }
}

```

Step 4: Security Configuration

If you want a custom configuration (optional):

```

java

// SecurityConfig.java
package com.example.oauth2;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@org.springframework.context.annotation.Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/css/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(); // enables OAuth2 login
        return http.build();
    }
}

```

⚙️ Step 5: application.yml

yaml

```
# application.yml
```

```
server:
```

```
  port: 8080
```

```
spring:
```

```
  security:
```

```
    oauth2:
```

```
      client:
```

```
        registration:
```

```
          github:
```

```
            client-id: YOUR_GITHUB_CLIENT_ID
```

```
            client-secret: YOUR_GITHUB_CLIENT_SECRET
```

```
            scope: read:user, user:email
```

```
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
```

```
        provider:
```

```
          github:
```

```
            authorization-uri: https://github.com/login/oauth/authorize
```

```
            token-uri: https://github.com/login/oauth/access_token
```

```
            user-info-uri: https://api.github.com/user
```

👉 Replace `YOUR_GITHUB_CLIENT_ID` and `YOUR_GITHUB_CLIENT_SECRET` with actual values from your GitHub OAuth app.

📝 Step 6: HTML Templates

templates/index.html

html

```
<!DOCTYPE html>
```

```
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head><title>Welcome</title></head>
```

```
<body>
```

```
<h2>Welcome to Spring Boot OAuth2 Example</h2>
```

```
<a href="/secured">Login with GitHub</a>
```

```
</body>
```

```
</html>
```

html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Secured Page</title></head>
<body>
<h2>Logged in User Details</h2>
<p><strong>Name:</strong> <span th:text="${name}"></span></p>
<p><strong>Email:</strong> <span th:text="${email}"></span></p>
<a href="/logout">Logout</a>
</body>
</html>
```

Step 7: Run the App

bash




```
./mvnw spring-boot:run
```

Open: <http://localhost:8080> → Click login → GitHub OAuth page → You're authenticated!

Optional Enhancements

- Customize user info mapping (PrincipalExtractor)
- Use a database to store tokens
- Add role-based access
- Integrate other providers (Google, Facebook, etc.)

Would you like:

-  JWT-based token validation (Resource Server)?
-  Google or Keycloak OAuth2 integration?
-  OpenID Connect example?

Let me know!

To write your own OAuth2 Authorization Server in Spring Boot, follow this **step-by-step guide** using the official **Spring Authorization Server library** (recommended for Spring Boot 3.x+ and Spring Security 6).

✓ Use Case

You'll build a **custom OAuth2 Authorization Server** that supports:

- `/oauth2/authorize` (Authorization endpoint)
 - `/oauth2/token` (Token endpoint)
 - `/oauth2/jwks` (JWK Set endpoint)
 - `/oauth2/introspect` (optional)
 - Client authentication and user login
-

🔧 Tech Stack

- Spring Boot 3.x+
 - Spring Security 6+
 - [Spring Authorization Server](#)
 - Java 17+
-

📁 Step-by-Step Setup

✓ Step 1: Create a Spring Boot Project

Generate from <https://start.spring.io>:

- **Dependencies:**
 - Spring Web
 - Spring Security
 - Spring Authorization Server
 - Spring Boot DevTools (optional)
 - Thymeleaf (for login UI)

Step 2: Add Dependencies (Maven)

xml

```
<dependencies>
  <!-- Spring Boot Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

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

  <!-- Spring Authorization Server -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
    <version>1.3.0</version> <!-- Check latest version -->
  </dependency>

  <!-- Thymeleaf for login page -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
</dependencies>
```

Step 3: Main Application Class

java

```
@SpringBootApplication
public class AuthServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(AuthServerApplication.class, args);
    }
}
```

```
}  
}
```

🔗 Step 4: Authorization Server Configuration

java

@Configuration

```
public class AuthorizationServerConfig {
```

@Bean

```
public SecurityFilterChain authServerSecurityFilterChain(HttpSecurity http) throws  
Exception {  
    OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);  
    return http.build();  
}
```

@Bean

```
public RegisteredClientRepository registeredClientRepository() {  
    RegisteredClient registeredClient =  
RegisteredClient.withId(UUID.randomUUID().toString())  
        .clientId("client-id")  
        .clientSecret("{noop}client-secret") // {noop} = plain text for demo  
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)  
        .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)  
        .redirectUri("http://localhost:8081/login/oauth2/code/custom-client")  
        .scope(OidcScopes.OPENID)  
        .scope("read")  
        .build();  
  
    return new InMemoryRegisteredClientRepository(registeredClient);  
}
```

@Bean

```
public JWKSource<SecurityContext> jwkSource() {  
    RSAKey rsaKey = generateRsa();  
    JWKSet jwkSet = new JWKSet(rsaKey);  
    return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);  
}
```

```
private RSAKey generateRsa() {
```



```

try {
    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(2048);
    KeyPair keyPair = keyPairGenerator.generateKeyPair();
    return new RSAKey.Builder((RSAPublicKey) keyPair.getPublic())
        .privateKey((RSAPrivateKey) keyPair.getPrivate())
        .keyID(UUID.randomUUID().toString())
        .build();
} catch (Exception e) {
    throw new IllegalStateException(e);
}
}

@Bean
public ProviderSettings providerSettings() {
    return ProviderSettings.builder()
        .issuer("http://localhost:9000") // Authorization Server URL
        .build();
}
}

```

Step 5: Default Login Configuration

```

java

@Configuration
public class DefaultSecurityConfig {

    @Bean
    public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authz -> authz
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults());

        return http.build();
    }

    @Bean
    public UserDetailsService users() {

```

```
    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("password")
        .roles("USER")
        .build();
    return new InMemoryUserDetailsManager(user);
}
}
```

Step 6: application.yml

yml

```
server:
  port: 9000
spring:
  thymeleaf:
    cache: false
```

Step 7: Add Custom Login Page (optional)

resources/templates/login.html

html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Login</title></head>
<body>
<h2>Login</h2>
<form th:action="@{/login}" method="post">
  <label>Username: <input type="text" name="username"/></label><br/>
  <label>Password: <input type="password" name="password"/></label><br/>
  <button type="submit">Login</button>
</form>
</body>
</html>
```

✓ Step 8: Run the Server

```
bash
```

```
./mvnw spring-boot:run
```

Go to: <http://localhost:9000/oauth2/authorize>

🔧 To Test with OAuth2 Client

Use another Spring Boot app with:

```
yaml
```

```
spring:
  security:
    oauth2:
      client:
        registration:
          custom-client:
            client-id: client-id
            client-secret: client-secret
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            scope: openid,read
        provider:
          custom-provider:
            issuer-uri: http://localhost:9000
```

🧰 Optional Add-ons

- ✓ Enable JWT-based access tokens
 - ✓ Use JDBCRegisteredClientRepository
 - ✓ Add PKCE, Introspection, Device Grant
 - ✓ Integrate with LDAP, DB, etc.
-

Would you like:

- A working client app for this server?

- JWT introspection support?
- Persist clients and tokens in the DB?

Let me know and I'll guide you step-by-step.

/oauth2/authorize (Authorization endpoint)

/oauth2/token (Token endpoint)

/oauth2/jwks (JWK Set endpoint)

/oauth2/introspect (optional)

explain details

Great! Let's go **endpoint-by-endpoint** and explain what each does in a **custom OAuth2 Authorization Server** built using **Spring Authorization Server**:

✓ 1. /oauth2/authorize – Authorization Endpoint

♦ Purpose:

Used in **Authorization Code Grant** and **Implicit Grant** flows.

♦ Flow:

A client (like a web app) redirects the user here to **request authorization**.

♦ Example request:

```
bash
```

```
GET /oauth2/authorize?  
  response_type=code  
  &client_id=client-id  
  &redirect_uri=http://localhost:8081/callback  
  &scope=openid read  
  &state=xyz
```

♦ What happens:

1. User is redirected to a login page (if not authenticated).

2. If authenticated, user is asked to **consent** to scopes.
3. Server redirects back to `redirect_uri` with:

```
perl
```

```
?code=AUTH_CODE&state=xyz
```

◆ Response:

The **authorization code** is exchanged later for a token.

✓ 2. `/oauth2/token` – Token Endpoint

◆ Purpose:

To **exchange the authorization code** for access and refresh tokens.

◆ Grant Types supported:

- `authorization_code`
- `refresh_token`
- `client_credentials` (if configured)

◆ Example request:

```
http
```

```
POST /oauth2/token
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Authorization: Basic base64(client-id:client-secret)
```

```
grant_type=authorization_code&
```

```
code=AUTH_CODE&
```

```
redirect_uri=http://localhost:8081/callback
```

◆ Response (JSON):

```
json
```

```
{
```

```
  "access_token": "xyz123...",
```

```
  "refresh_token": "abc456...",
```

```
  "token_type": "Bearer",
```

```
"expires_in": 3600,  
"scope": "openid read",  
"id_token": "... // if using OpenID Connect  
}
```

✓ 3. /oauth2/jwks – JWK Set Endpoint

♦ Purpose:

Exposes the **public keys** used by the Authorization Server to **sign tokens**.

♦ Response:

Returns a **JWKS** (JSON Web Key Set) document. It's used by **resource servers** or clients to **verify JWTs**.

♦ Example:

http

GET /oauth2/jwks

♦ Response:

json

```
{  
  "keys": [  
    {  
      "kty": "RSA",  
      "kid": "some-key-id",  
      "use": "sig",  
      "alg": "RS256",  
      "n": "...",  
      "e": "AQAB"  
    }  
  ]  
}
```

- **n** = modulus
- **e** = public exponent

This key matches the one used by the server to sign JWTs.

✓ 4. `/oauth2/introspect` – Token Introspection Endpoint (Optional)

♦ Purpose:

To **validate opaque tokens** or introspect JWTs (especially if they are encrypted or not directly verifiable).

♦ Example:

```
http

POST /oauth2/introspect
Content-Type: application/x-www-form-urlencoded
Authorization: Basic base64(client-id:client-secret)

token=access_token_here
```

♦ Response:

```
json

{
  "active": true,
  "scope": "read",
  "username": "user",
  "exp": 1629036000,
  "sub": "user",
  "client_id": "client-id"
}
```

- `"active": false` if token is invalid or expired.

This endpoint is especially useful in **Resource Servers** that cannot validate JWTs directly (e.g., opaque tokens or dynamic signing).

Flow Recap (Authorization Code Grant):

```
bash
```

Client → /oauth2/authorize → User Login & Consent → redirect_uri with code
↓
Client → /oauth2/token → gets access_token
↓
Client → protected resource (API), with Bearer token
↓
API → validate token using /oauth2/jwks or /oauth2/introspect

Key Concepts:

Endpoint	Used By	Function
/oauth2/authorize	Client apps (frontend)	Starts user login & consent
/oauth2/token	Backend (confidential) clients	Exchanges code for token
/oauth2/jwks	Resource servers / clients	Validates signed JWTs
/oauth2/introspect	Resource servers (optionally)	Validates tokens (especially opaque)

Would you like:

- JWT vs Opaque token difference?
- How to expose a protected resource server that validates tokens?
- How to write a client to test this server?

Let me know!

types Authorization Code Grant and explain all in detail

The **Authorization Code Grant** is the **most secure and widely used OAuth2 flow**. It is ideal for **web applications** (server-side) that can securely store client secrets and exchange authorization codes for access tokens.

Let's break it down **step-by-step** with **full explanation**, **actors**, **endpoints**, **security mechanisms**, and **real-world analogy**.

What is Authorization Code Grant?

The **Authorization Code Grant** is an OAuth2 flow in which:

- The **user authenticates** via the **Authorization Server**.
 - The client (your app) **receives an authorization code** (not a token).
 - The client **exchanges** this code (via a secure backchannel) for **access and optionally refresh tokens**.
-

Key Actors

Actor	Description
Resource Owner	The user who is granting access
Client	The app (e.g., web app) requesting access
Authorization Server	Issues tokens (your own auth server)
Resource Server	The API that holds protected data (like user profile)

Full Flow Diagram

SCSS

```
[User] ———(1)———> [Client App: /authorize]
<——(2)——— Authorization prompt (Login + Consent)
——(3)———> [Authorization Server] — validates user
<——(4)——— Redirect with Authorization Code
——(5)———> [Client exchanges code for token]
```

<—(6)— Access Token (+ ID Token + Refresh Token)
—(7)—> [Client uses token to access API (Resource Server)]

Step-by-Step Breakdown

✓ Step 1: Client redirects user to authorization server

http

```
GET /oauth2/authorize?
  response_type=code
  &client_id=client123
  &redirect_uri=https://myapp.com/callback
  &scope=openid profile email
  &state=xyz
```

- **response_type=code** → This tells the server to return an authorization **code**, not a token.
- **scope** → Requested permissions
- **state** → Random string to prevent CSRF (must be validated later)

✓ Step 2: User authenticates and consents

- The user is prompted to **login** and **approve scopes**.
- If already authenticated, they may skip login.

✓ Step 3: Authorization server issues an authorization code

Redirects the user to the client's `redirect_uri`:

http

```
GET https://myapp.com/callback?code=abc123&state=xyz
```

- `code=abc123` is the temporary authorization code (usually valid for a few minutes).
- `state=xyz` should match what client sent (CSRF protection).

✓ Step 4: Client sends code to /oauth2/token endpoint

Now on the backend (server-to-server request):

```
http

POST /oauth2/token
Content-Type: application/x-www-form-urlencoded
Authorization: Basic base64(client_id:client_secret)

grant_type=authorization_code&
code=abc123&
redirect_uri=https://myapp.com/callback
```

🔒 This is secure because:

- It uses HTTP Basic auth
- It's server-to-server (not exposed to browser)

✓ Step 5: Authorization server returns access token

```
json

{
  "access_token": "xyz...",
  "refresh_token": "abc...",
  "id_token": "jwt...",    // if using OpenID Connect
  "expires_in": 3600,
  "token_type": "Bearer",
  "scope": "openid profile"
}
```

✓ Step 6: Client uses access token to call resource server

```
http

GET /user/profile
Authorization: Bearer xyz...
```

The **resource server** (API) validates the access token (via:

- /oauth2/jwks (JWT)
- /oauth2/introspect (opaque token)

Optional: Refreshing the Token

Later, the client can use the **refresh_token**:

```
http
```

```
POST /oauth2/token
grant_type=refresh_token&
refresh_token=abc...
```

Security Benefits of Authorization Code Flow

Feature	Why it matters
Backchannel token exchange	Tokens are not exposed in URLs or browser
Refresh tokens	Allows long-lived sessions
PKCE (Proof Key for Code Exchange)	Prevents code interception (required in public clients)
State parameter	Protects against CSRF
Scopes	Granular control over permissions

Real-World Analogy

Imagine you want to access your bank records via a tax filing service:

1. You log into your bank (auth server) and approve access.
2. The bank gives the tax app a temporary "access slip" (authorization code).
3. The app then sends that slip back to the bank (securely), and gets an "access token" that allows it to read your statements.

PKCE (Recommended for Public Clients)

For public clients (like mobile or SPA), you must use **PKCE** (Proof Key for Code Exchange):

```
http
```

```
# Step 1: Client generates code_challenge and code_verifier
```

```
# Sends code_challenge to /authorize
```

```
GET /authorize?...&code_challenge=xyz&code_challenge_method=S256
```

```
# Step 2: On token exchange, client sends code_verifier:
```

```
POST /token
```

```
...&code_verifier=original_string
```

The server will verify it matches what was sent in step 1.

Summary

Step	Endpoint	Description
1	<code>/oauth2/authorize</code>	Redirect user to login and consent
2	Redirect with <code>code</code>	Temporary code returned
3	<code>/oauth2/token</code>	Exchange code for token
4	Use <code>access_token</code>	Access protected APIs
5	<code>/oauth2/jwks</code>	Validate JWT
6	<code>/oauth2/introspect</code> (optional)	Validate opaque token

✓ Best Practices

- Always use **PKCE** (required for public clients)
- Use HTTPS
- Validate `state` parameter
- Use `scope` properly
- Don't store tokens in insecure places

Would you like a **Postman collection**, or a **working frontend + backend client example** using this flow?

Let me know!

i asked types of grants

Ah, got it! You're asking about the **different types of OAuth2 grants**, not just the Authorization Code Grant.

Here's a complete breakdown of all **OAuth2 Grant Types** (a.k.a. Authorization Flows) with full explanations:

✓ OAuth2 Grant Types (Flows) – Explained in Detail

Grant Type	Use Case	Secure?	Recommended for
1. Authorization Code (with PKCE)	Browser/client → backend	✓✓✓	Web apps, SPAs, mobile apps
2. Client Credentials	App-to-app (no user)	✓✓✓	Backend systems, microservices
3. Resource Owner Password Credentials (ROPC)	User gives app their username/password	✗ Deprecated	Legacy apps (NOT recommended)
4. Refresh Token	Refreshing access token	✓✓✓	Long sessions

Grant Type	Use Case	Secure?	Recommended for
5. Implicit Flow	Tokens via browser only (deprecated)	XX	Legacy SPAs (DO NOT USE)
6. Device Code	Devices without browser (TVs)	✓✓	Smart devices, CLI apps

1. Authorization Code Grant (with PKCE)

Most secure and recommended.

◆ Use Case:

- Web apps
- Mobile apps (with PKCE)
- SPAs

◆ Flow:

pgsql

Client → /authorize → User login + consent

↓

Receive Authorization Code

↓

Client exchanges Code → /token → Receives Access Token (+Refresh Token)

◆ Security:

- Backend exchange
- Supports PKCE
- Keeps client secret safe

✓ Recommended flow in modern apps.

2. Client Credentials Grant

No user involved — system-to-system authentication.

◆ Use Case:

- Microservices calling internal APIs
- Machine-to-machine

♦ **Flow:**

arduino

Client → /token

grant_type=client_credentials

client_id + secret

↓

Receive Access Token

♦ **Security:**

- No user login
- Must keep client secret safe

✓ Best for backend systems, daemons, services.

3. ✗ Resource Owner Password Credentials (ROPC)

User provides username/password directly to the app.

♦ **Use Case:**

- Legacy apps
- First-party apps

♦ **Flow:**

pgsql

Client → /token

grant_type=password

username, password

↓

Receive Access Token

♦ **Security Risks:**

- App handles raw credentials
- Not recommended for third-party or modern apps

4. 🔄 Refresh Token Grant

Used to refresh access tokens without re-authentication.

♦ Use Case:

- Long sessions
- Avoid re-login

♦ Flow:

arduino

Client → /token

grant_type=refresh_token

refresh_token=xxx

↓

Receive new access token

♦ Notes:

- Typically issued alongside access token
 - Used with Authorization Code, Device, or Password grants
-

5. ⚠️ Implicit Grant (Deprecated)

Tokens are returned in URL fragment after login.

♦ Use Case:

- Old single-page apps (SPA)
- No backend token exchange

♦ Flow:

nginx

Client → /authorize

response_type=token



Access token returned in [redirect](#) URL (fragment)

◆ **Problems:**

- No refresh tokens
- Tokens exposed in URL
- Vulnerable to interception

✗ Depreciated in OAuth 2.1. **Do not use.**

6. 📺 Device Authorization Grant

Used for devices without browsers (TVs, consoles).

◆ **Use Case:**

- Smart TVs
- CLI apps
- IoT devices

◆ **Flow:**

pgsql

Device → /device_authorization → Gets user_code + verification_uri



User uses another device to visit URI and log in



Device polls /token until authorized



Receives access token

◆ **Secure and User-Friendly:**

- User never types password on the device
- Recommended for input-constrained environments



Summary Table

Grant Type	User Auth?	Client Secret Used?	Supports Refresh?	Modern OAuth2.1?
Authorization Code (PKCE)	✓	✓/No (PKCE)	✓	✓✓✓
Client Credentials	✗	✓	✗	✓✓✓
Password (ROPC)	✓	✓	✓	✗ Deprecated
Refresh Token	✗	✓	🔄 Yes	✓
Implicit	✓	✗	✗	✗ Deprecated
Device Code	✓	✓/No (PKCE)	✓	✓✓✓

✓ What Should You Use?

App Type	Recommended Grant Type
Web App (browser + backend)	Authorization Code + PKCE
SPA / Mobile App	Authorization Code + PKCE
Backend Service	Client Credentials
Smart TV / CLI App	Device Authorization Grant

Would you like:

- Example requests for each grant?
- Implement these using Spring Authorization Server?
- Postman collection for testing?

Let me know!