Step 1: Set Up the Spring Boot Project

Use Spring Initializr or your preferred IDE to create a new Spring Boot project.

<dependencies>

<!-- Spring Boot Starter Dependencies -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Add other necessary dependencies, such as Spring Security, Spring Data JPA, etc. -->

</dependencies>

Add the necessary dependencies: Spring Web, Spring Security, Spring Data JPA, OAuth2 Security libraries, and any other required dependencies.

Configure the database connection (Postgres or H2) in your application.properties or application.yml file.

Step 2: Implement Multi-Tenancy

Configure multi-tenancy at the data source and Hibernate levels. Use separate schemas, databases, or any other approach you prefer to isolate tenant data.

Implement the CurrentTenantIdentifierResolver and MultiTenantConnectionProvider based on your chosen multi-tenancy strategy.

Step 3: Implement Authorization Server

@Entity

public class TenantMaster {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// Other properties and getters/setters

}

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String password;

@ManyToOne

@JoinColumn(name = "tenant\_id")

private TenantMaster tenant;

// Other properties and getters/setters

}

Configure the OAuth2 authorization server using Spring Security's @EnableAuthorizationServer and related annotations.

Define OAuth2 client details and scopes for the clients that will interact with the Authorization Server.

public interface TenantMasterRepository extends JpaRepository<TenantMaster, Long> {

}

public interface UserRepository extends JpaRepository<User, Long> {

}

Step 4: Implement Resource Server

@Configuration

@EnableAuthorizationServer

public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private DataSource dataSource;

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

clients.jdbc(dataSource);

}

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) {

endpoints

.authenticationManager(authenticationManager)

// Other endpoint configurations

}

}

Configure the OAuth2 resource server using Spring Security's @EnableResourceServer annotation.

Implement access control mechanisms based on the tenant associated with the token.

Implement resource endpoints and ensure they enforce proper tenant segregation and access controls.

Step 5: Create Client Application

@SpringBootApplication

public class ClientApplication {

public static void main(String[] args) {

SpringApplication.run(ClientApplication.class, args);

}

// Implement client application logic

}

Develop a client application that interacts with the Authorization Server and Resource Server.

Configure the client application with the necessary OAuth2 client details.

Handle multi-tenancy considerations when making requests and handling responses.

Step 6: Resource Server Configuration

Configure the Resource Server to enforce access controls based on the tenant:

@Configuration

@EnableResourceServer

public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

@Override

public void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/resources/\*\*").access("#oauth2.hasScope('read') and hasAuthority('ROLE\_USER')")

.antMatchers("/admin/\*\*").access("#oauth2.hasScope('read') and hasAuthority('ROLE\_ADMIN') and hasAuthority('TENANT\_1')")

// Other endpoint configurations

}

}

Step 7: Set Up the Database

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

Set up the database configuration in your project. If you're using Postgres, configure the connection details and credentials.

Define JPA entities for the TenantMaster and User tables.

Implement repositories for data access and manipulation.

Define the database entities (TenantMaster and User) with appropriate relationships and constraints.

Implement repositories and services to manage tenant and user data.

Step 8: Test the System

Write integration tests to verify the functionality, security, and multi-tenancy support of your system.

Ensure that each tenant's data is properly segregated, and users can access only their respective tenant's resources.

Please note that this is a high-level outline, and the actual implementation details can vary based on your specific requirements, the latest Spring Security 6 features, and any changes that have occurred since my last update in September 2021. Be sure to consult the latest Spring Security documentation and relevant resources for detailed guidance on implementing the described components.

Due to the complexity of the task and potential updates since my last knowledge update, I recommend referring to the official Spring Security documentation, tutorials, and community resources for step-by-step guidance on setting up your project.