**Document a Detailed Solution Plan for Implementing the Registration Process**

**Overview**

The registration process for the "Beaus Website" involves enabling users to register and authenticate using OAuth2 (e.g., Google, GitHub) and issuing JWT (JSON Web Token) credentials for subsequent API requests. The solution must ensure a seamless user experience, robust security, and maintainable code. This plan outlines the requirements, design, implementation, testing, and deployment steps to achieve this goal.

**Requirements**

The registration process must meet the following requirements:

* **User Registration via OAuth2:** Users should be able to register and log in using an OAuth2 provider (e.g., Google, GitHub).
* **Automatic Account Creation:** If a user logs in via OAuth2 for the first time, the system should automatically create a local user account based on the information provided by the OAuth2 provider (e.g., email, name).
* **JWT Issuance:** After successful registration or login, the system must issue a JWT to the user for use in future authenticated API requests.
* **Security:** Passwords (if any) must be securely hashed, and sensitive data (e.g., access tokens) must be handled securely.
* **Validation:** Ensure that user data (e.g., email) is unique and valid.
* **Scalability:** The solution should be scalable to handle a growing number of users.
* **Standards Compliance:** The implementation must follow clean code principles, clean architecture, Java 8 standards, and best practices.

**Design**

The design of the registration process involves defining the system architecture, database schema, API endpoints, and the overall flow of data.

**System Architecture**

The system will follow a clean architecture approach, dividing the application into distinct layers:

* **Presentation Layer:** Handles HTTP requests and responses (e.g., REST controllers).
* **Application Layer:** Contains business logic (e.g., services for registration and JWT generation).
* **Domain Layer:** Defines the core entities and business rules (e.g., User entity).
* **Infrastructure Layer:** Manages external interactions (e.g., database access, OAuth2 provider communication).

**Database Schema**

The database will include a users table to store user information. The schema is as follows:

* **Table: users**
  + id: Long (Primary Key, Auto-incremented)
  + username: String (Unique, derived from email or OAuth2 provider)
  + email: String (Unique, required)
  + provider: String (e.g., "google", "github")
  + provider\_id: String (Unique identifier from the OAuth2 provider)
  + created\_at: Timestamp (When the user was created)
  + is\_active: Boolean (Whether the user account is active)

**API Endpoints**

The system will expose the following endpoints for registration and authentication:

* **POST /api/auth/register:** Initiates the OAuth2 login flow (though typically handled by redirecting to the OAuth2 provider).
* **GET /login/oauth2/code/{provider}:** Callback URL for the OAuth2 provider to return the authorization code (handled by Spring Security).
* **GET /api/auth/login-success:** Returns the JWT after successful OAuth2 login.

**Data Flow**

The registration flow, as depicted in the sequence diagram, involves:

1. The user initiates registration via OAuth2.
2. The system redirects the user to the OAuth2 provider for authentication.
3. After authentication, the OAuth2 provider returns an authorization code.
4. The system exchanges the code for an access token and retrieves user information.
5. The system checks if the user exists in the database; if not, it creates a new user.
6. The system generates a JWT and returns it to the user.

**Implementation**

The implementation will use Spring Boot with Java 8, leveraging Spring Security for OAuth2 integration and JWT generation. Below are the key components and code snippets.

**Technology Stack**

* **Framework:** Spring Boot (version compatible with Java 8, e.g., 2.x)
* **Security:** Spring Security for OAuth2 and JWT
* **Database:** MySQL/PostgreSQL with JPA (Hibernate)
* **Libraries:**
  + spring-boot-starter-oauth2-client for OAuth2 integration
  + spring-boot-starter-security for security configuration
  + spring-boot-starter-data-jpa for database access
  + jjwt or Spring Security's built-in JWT support for token generation

**Code Structure**

The code will be organized into layers as per clean architecture.

**1. Domain Layer (Entity)**

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(unique = true, nullable = false)

private String username;

@Column(unique = true, nullable = false)

private String email;

@Column(nullable = false)

private String provider;

@Column(name = "provider\_id", nullable = false)

private String providerId;

@Column(name = "created\_at", nullable = false)

private LocalDateTime createdAt = LocalDateTime.now();

@Column(name = "is\_active")

private boolean isActive = true;

*// Getters and setters*

}

**2. Infrastructure Layer (Repository)**

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByProviderAndProviderId(String provider, String providerId);

boolean existsByEmail(String email);

}

**3. Application Layer (Service)**

* **Custom OAuth2 User Service:** Handles user creation after OAuth2 login.

@Service

public class CustomOAuth2UserService extends DefaultOAuth2UserService {

@Autowired

private UserRepository userRepository;

@Override

public OAuth2User loadUser(OAuth2UserRequest userRequest) throws OAuth2AuthenticationException {

OAuth2User oAuth2User = super.loadUser(userRequest);

String provider = userRequest.getClientRegistration().getRegistrationId();

String providerId = oAuth2User.getAttribute("id");

String email = oAuth2User.getAttribute("email");

User user = userRepository.findByProviderAndProviderId(provider, providerId)

.orElseGet(() -> {

User newUser = new User();

newUser.setProvider(provider);

newUser.setProviderId(providerId);

newUser.setEmail(email);

newUser.setUsername(email.split("@")[0]); *// Derive username from email*

return userRepository.save(newUser);

});

return new CustomOAuth2User(oAuth2User, user);

}

}

* **JWT Service:** Generates JWT for the user.

@Service

public class JwtService {

private final JwtEncoder jwtEncoder;

private final long ttl = 3600000; *// 1 hour*

@Autowired

public JwtService(JwtEncoder jwtEncoder) {

this.jwtEncoder = jwtEncoder;

}

public String generateToken(User user) {

List<GrantedAuthority> authorities = AuthorityUtils.createAuthorityList("ROLE\_USER");

Instant now = Instant.now();

JwtClaimsSet claims = JwtClaimsSet.builder()

.issuer("beaus-website")

.subject(user.getId().toString())

.issuedAt(now)

.expiresAt(now.plusMillis(ttl))

.claim("authorities", authorities)

.build();

return jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();

}

}

**4. Presentation Layer (Controller)**

@RestController

@RequestMapping("/api/auth")

public class AuthController {

@Autowired

private JwtService jwtService;

@GetMapping("/login-success")

public ResponseEntity<String> loginSuccess(@AuthenticationPrincipal CustomOAuth2User user) {

String token = jwtService.generateToken(user.getUser());

return ResponseEntity.ok(token);

}

}

**5. Security Configuration**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorize -> authorize

.requestMatchers("/api/auth/\*\*").permitAll()

.anyRequest().authenticated()

)

.oauth2Login(oauth2 -> oauth2

.loginPage("/oauth2/authorization/{registrationId}")

.successHandler((request, response, authentication) -> {

CustomOAuth2User oAuth2User = (CustomOAuth2User) authentication.getPrincipal();

String token = jwtService.generateToken(oAuth2User.getUser());

response.setHeader("Authorization", "Bearer " + token);

}))

.oauth2ResourceServer(oauth2 -> oauth2.jwt());

return http.build();

}

}

**6. Application Configuration (application.yml)**

spring:

security:

oauth2:

client:

registration:

google:

client-id: <your-google-client-id>

client-secret: <your-google-client-secret>

scope: email,profile

**7. Conclusion**

This solution plan provides a comprehensive roadmap for implementing the user registration process in the "Beaus Website" project. By following clean code principles, clean architecture, and Java 8 standards, the implementation will be maintainable, scalable, and secure. The testing strategy ensures reliability, while the design aligns with the OAuth2 and JWT requirements of the task.