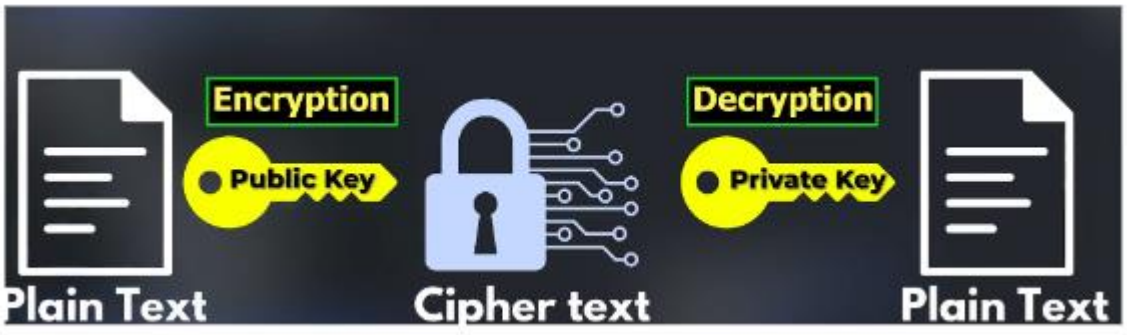
1.Encryption and Decryption

Encryption is the process of converting plain text into ciphertext to protect data from unauthorized access. Decryption is the reverse process of converting ciphertext back into plain text.

How to Secure Our Data?

- **Symmetric Key Encryption**:
  - The same key is used for both encryption and decryption.
  - Examples: AES (Advanced Encryption Standard), DES (Data Encryption Standard).
  - *Note*: The key must be shared securely before communication.
- **Asymmetric Key Encryption**:
  - A pair of keys is used: Public Key and Private Key.
  - Public Key is used for encryption, and Private Key is used for decryption (or vice versa).
  - Examples: RSA (Rivest-Shamir-Adleman), ECC (Elliptic Curve Cryptography).



2.Digital Signature

A digital signature ensures the authenticity and integrity of a message or document in digital communication.

How It Works:

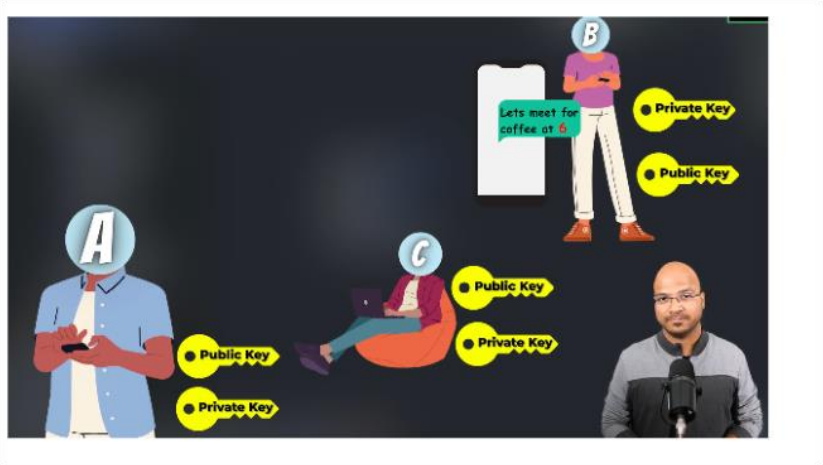
- The sender encrypts the message using their private key to generate a unique signature.
- The receiver uses the sender's public key to decrypt the signature and verify its authenticity.
- This process ensures the message is sent by the claimed sender and has not been tampered with.

Applications:

- Used for secure transactions.
- Proves identity in online communications.

Additional Concept:

- **Double Encryption**:
  - Combines encryption and signing for enhanced security.
  - Example: Encrypt with a public key, sign with a private key.



3.Why Use JWT (JSON Web Token)?

JWT is a lightweight, self-contained token commonly used for secure information exchange. It provides a stateless authentication mechanism.

Key Advantages:

- Scalability: Servers do not need to store session state.
- Portability: The token can be passed between systems.
- Secure Data Sharing: Contains encrypted claims for validation.

4.What is JWT?

JWT is a compact, URL-safe token used to represent claims between two parties. It adheres to the RFC 7519 standard.

Structure of JWT:

- **Header**: Contains metadata, such as the type of token and the signing algorithm.
- **Payload**: Contains claims (user data or other information).
- **Signature**: Verifies the integrity of the token.

Example:

- **Encoded JWT**: Base64-encoded string with header, payload, and signature.
- **Decoded JWT**: A human-readable JSON object with user information.

Validation:

- Verify the signature to confirm the token's authenticity and integrity.
- Check the claims for permissions and expiration.

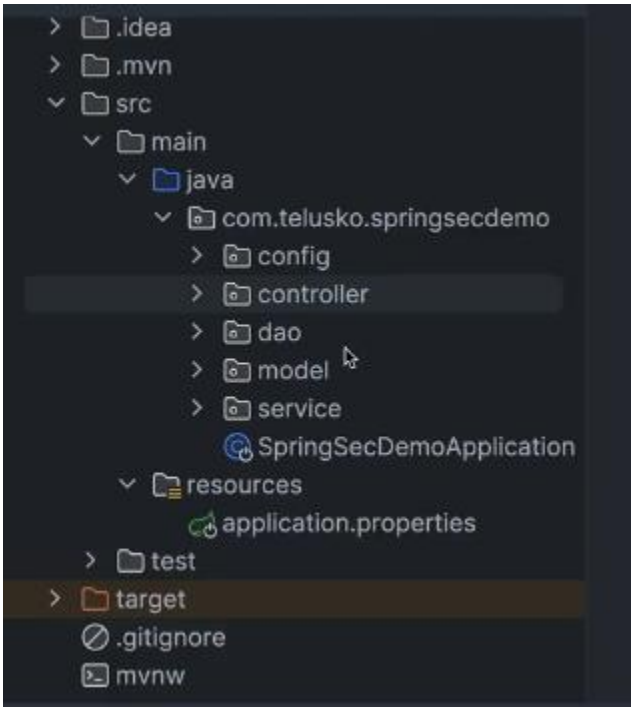
Additional Enhancements

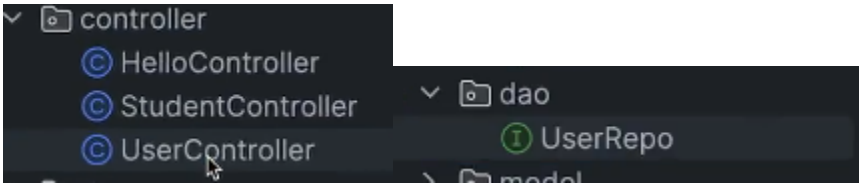
- Include diagrams for each section for better understanding:
  - Symmetric and asymmetric encryption flow.

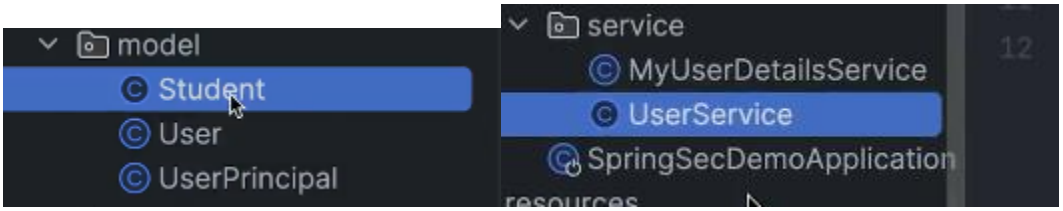
    - Process of signing and verifying digital signatures.
    - Structure of a JWT with Header, Payload, and Signature labeled.
- Highlight practical examples, such as how tokens are used in APIs for secure communication.

5.Project Setup For JWT

Project Structure







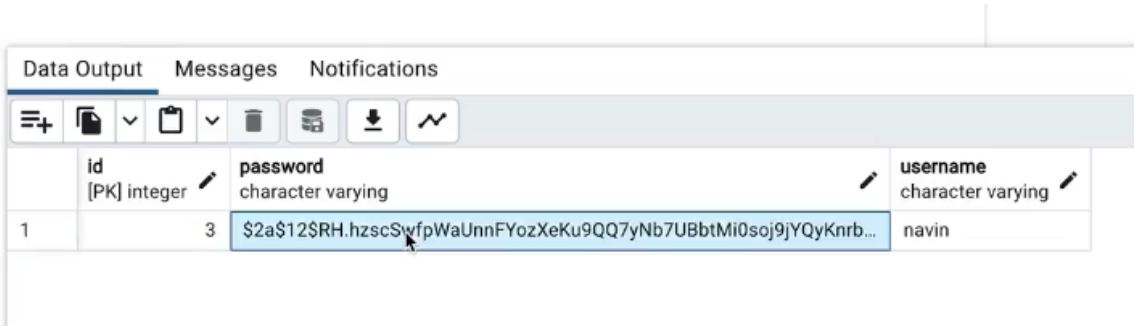
Make changes in that method that are in SecurityConfig:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

		http.csrf(customizer -> customizer.disable())
				.authorizeHttpRequests(request -> request
						.requestMatchers("register")
						.permitAll()
						.anyRequest().authenticated())
				.sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
	return http.build();
}
```

Add @GeneratedValue(strategy=GenerationType.IDENTITY) in this model

```
package com.telusko.springsecdemo.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;


@Data
@Table(name = "users")
@Entity
public class User {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String username;
	private String password;

}
```

Output:



6.Custom Login

Introduction to Custom Login

In Spring Security, custom login allows developers to define their logic for user authentication, bypassing the default Spring Security login page. This involves implementing an API that authenticates users using their credentials against the application's security configuration.

UserController Implementation

The UserController defines an endpoint for login (/login) that accepts user credentials and authenticates them using the AuthenticationManager.

Example:

```
@PostMapping("/login")
public String login(@RequestBody User user) {
    Authentication authentication = authenticationManager
            .authenticate(new UsernamePasswordAuthenticationToken(
                user.getUsername(), user.getPassword()));

    if (authentication.isAuthenticated()) {
        return "Success";
    } else {
        return "Login Failed";
    }
}
```

Explanation:

1. **@PostMapping("/login"):**
    - Defines a POST endpoint at /login for handling login requests.
2. **@RequestBody User user:**

- Maps the incoming JSON request body to a User object containing username and password.

3. **AuthenticationManager:**
   - A central interface in Spring Security used to handle authentication requests.
   - It is responsible for delegating authentication logic to configured AuthenticationProviders.
4. **UsernamePasswordAuthenticationToken:**
   - A token implementation used to represent a user's authentication request containing a username and password.
   - Passed to the authenticate() method of AuthenticationManager.
5. **authentication.isAuthenticated():**
   - A method to check if the user has been successfully authenticated.
   - If true, the API returns "Success"; otherwise, "Login Failed".

SecurityConfig: Defining the AuthenticationManager Bean

To use the AuthenticationManager in the controller, it must be explicitly defined as a bean in the SecurityConfig class.

Code:

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
        throws Exception {
    return config.getAuthenticationManager();
}
```

Explanation:

1. **@Bean:**
    - Marks the authenticationManager method as a Spring-managed bean.
2. **AuthenticationConfiguration:**
    - Provides access to the AuthenticationManager configured by Spring Security.
    - The getAuthenticationManager() method retrieves the default AuthenticationManager instance.
3. **Purpose:**
    - This configuration ensures the AuthenticationManager is available for injection into other components, such as the UserController.

Postman Test

The /login endpoint can be tested using Postman with the following JSON payload:

Request:

- **Endpoint:** POST http://localhost:8081/login
- **Headers:** Content-Type: application/json
- **Body:**

```
{
  "username": "navin",
  "password": "n@123"
}
```

Response:

- **Status:** 200 OK
- **Body:**

```
"Success"
```

Conclusion

This implementation demonstrates how to create a custom login API using AuthenticationManager. Key points include:

- Defining an authentication bean in the SecurityConfig.
- Using UsernamePasswordAuthenticationToken for handling authentication credentials.
- Testing the API using Postman to ensure it works as expected.

7.Generating JWT Tokens

JWT (JSON Web Token) is a compact and self-contained way for securely transmitting information between parties as a JSON object. In a Spring Boot application, JWT is commonly used to secure REST APIs. This guide explains how to generate a JWT token after a successful login.

1. Add Dependencies in pom.xml

To work with JWT, include the required dependencies.

Dependencies:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>compile</scope>
</dependency>
```

Explanation:

1. jjwt-api:
    - Provides the main API for working with JWTs (e.g., building and parsing tokens).
2. jjwt-impl:
    - Contains the implementation of the JWT library.
3. jjwt-jackson:
    - Integrates with the Jackson library to serialize/deserialize claims in the JWT.

2. Create JwtService Class

The JwtService class is responsible for generating JWT tokens.

Example:

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;

public class JwtService {
  public String generateToken(String username) {
    Map<String, Object> claims = new HashMap<>(); // Claims can include custom data (e.g., roles, permissions)
    claims.put("username", username); // Adding custom claim

    return Jwts.builder()
      .setClaims(claims) // Add claims to the token
      .setSubject(username) // Set the subject (e.g., the username)
      .setIssuedAt(new Date(System.currentTimeMillis())) // Current time as issue time
      .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 3)) // Token expiration time (3 hours)
      .signWith( getKey(),SignatureAlgorithm.HS256) // Sign the token with the secret key
      .compact(); // Generate the token
  }
}
```

Explanation:

1. Map<String, Object> claims:
    - A map to store additional information (claims) such as username, roles, or custom data.
2. setClaims(Map<String, Object>):
    - Adds custom claims (key-value pairs) to the token.
3. setSubject(String username):
    - Sets the username as the subject of the token, representing the authenticated user.
4. setIssuedAt(Date date):
    - Specifies when the token was issued.
5. setExpiration(Date date):
    - Sets the token's expiration time, calculated as the current time plus a duration (e.g., 3 hours in this case).
6. signWith(Key key, SignatureAlgorithm algorithm):
    - Signs the token using the specified algorithm (HS256 in this case) and a secret key.
7. compact():
    - Generates and returns the JWT token as a compact string.

3. Use JwtService in UserController

After a successful login, call the JwtService to generate the token.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @Autowired
    private JwtService jwtService;

    @PostMapping("/login")
    public String login(@RequestBody User user) {
        // Authenticate user (logic not shown)
        boolean isAuthenticated = true; // Replace with actual authentication logic

        if (isAuthenticated) {
            return jwtService.generateToken(user.getUsername());
        } else {
            return "Login Failed";
        }
    }
}
```

Explanation:

1. @Autowired JwtService jwtService:
    - Injects the JwtService to use its generateToken() method.
2. Logic in login():
    - Upon successful authentication, calls jwtService.generateToken() with the username and returns the generated JWT.

4. Token Claims and Signing

Claims:

- Claims represent the payload data within the JWT.
- Common claims:
    - sub (subject): Represents the user.
    - iat (issued at): The timestamp of when the token was created.
    - exp (expiration): When the token will expire.

Signing:

- JWT tokens are signed with a secret key to ensure their integrity.
- In this example, the **HS256** algorithm is used, requiring a secure secret key.

8.Token Generated

1. Hardcoded Key

Example:

```
private static final String SECRET = "TmV3U2VjcmV0S2V5Rm9ySldUU29uZ2luZ21hc2sgZmF2b3I=";
```

Explanation:

- **SECRET**: This is a **Base64-encoded static secret key**. It's used for encoding and decoding the token (JWT or similar).
- **Hardcoding**: Hardcoding secrets is generally discouraged because:
    - It can lead to security vulnerabilities if the codebase is exposed.
    - It lacks flexibility for environment-specific configurations.

Recommendation: Use dynamically generated or environment-managed secrets instead of hardcoding them.

2. Dynamic Key Generation Using Method

Example:

```
public String generateSecretKey() {
    try {
        KeyGenerator keyGen = KeyGenerator.getInstance("HmacSHA256");
        SecretKey secretKey = keyGen.generateKey();
        System.out.println("Secret Key : " + secretKey.toString());
        return Base64.getEncoder().encodeToString(secretKey.getEncoded());
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException("Error generating secret key", e);
    }
}
```

Explanation:

- **Dynamic Key Generation**:
    - Uses the KeyGenerator class to dynamically create a cryptographic key for the **HMAC-SHA256** algorithm.
    - This ensures better security compared to hardcoding the key.

- **Encoding**:
    - Converts the generated key into a **Base64-encoded string** for easier storage and transport.
- **Error Handling**:
    - The NoSuchAlgorithmException is caught to handle cases where the cryptographic algorithm is not available in the environment.

Advantages:

- Keys are generated at runtime, making it secure and unique for each session or environment.
- It avoids the risks of hardcoding keys.

3. Assigning and Using the Key in a Service

Example:

```
private String secretKey;

public JwtService() {
    secretKey = generateSecretKey();
}
```

Explanation:

- **secretKey**:
    - An instance variable to store the dynamically generated secret key.
- **JwtService Constructor**:
    - Calls generateSecretKey() during the initialization of the service, ensuring the service always uses a newly generated key.

Best Practices:

- The secretKey can be shared across the service securely for signing or verifying tokens.

4. Decoding and Using the Key for Cryptographic Operations

Example:

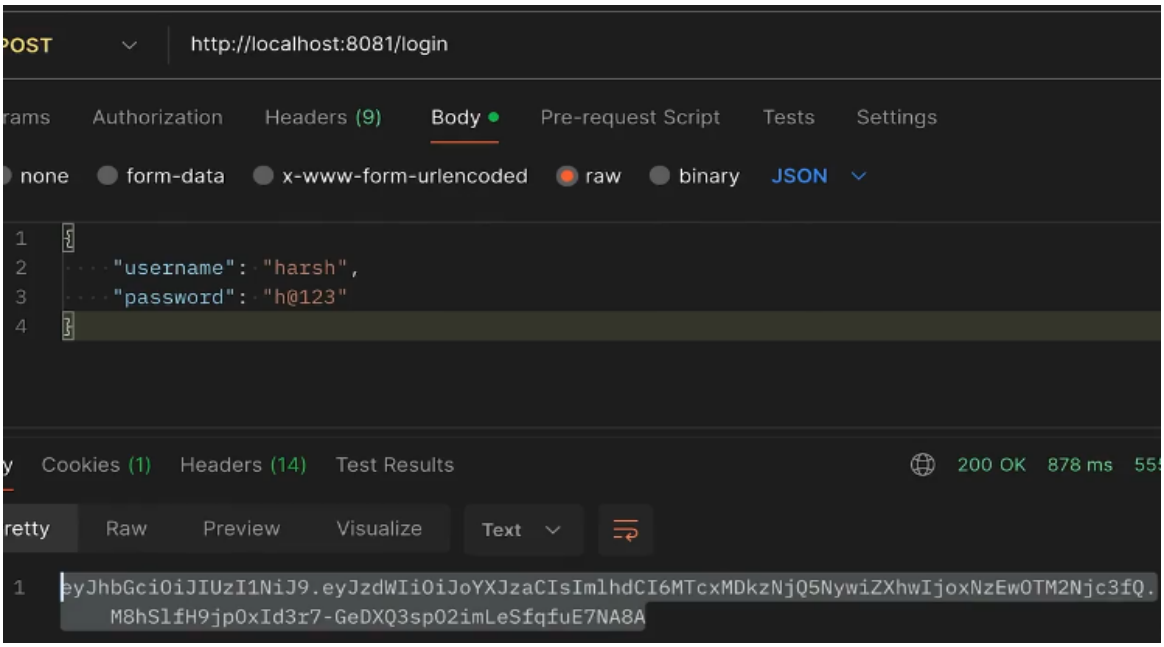
```
private Key getKey() {
    byte[] keyBytes = Decoders.BASE64.decode(secretKey);
    return Keys.hmacShaKeyFor(keyBytes);
}
```

Explanation:

- **Base64 Decoding**:
    - Converts the Base64-encoded secret key back to its raw byte format.
- **Key Conversion**:
    - Uses Keys.hmacShaKeyFor(byte[]) to convert the raw byte array into a Key object, suitable for cryptographic operations (e.g., signing and verifying JWTs).

Utility:

- This method ensures that the key can be directly used for secure cryptographic operations.



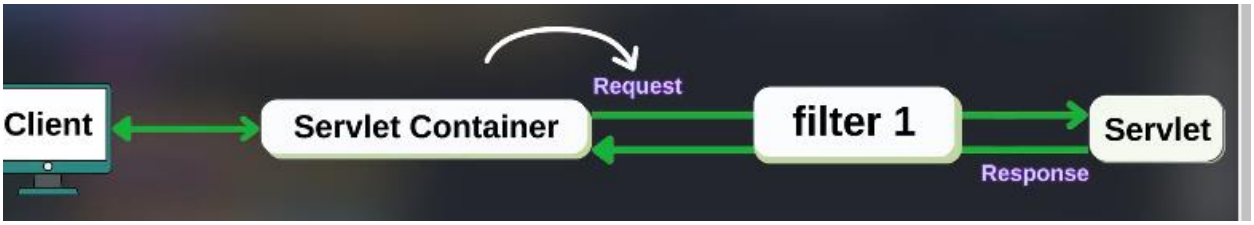
Summary Notes:

1. **Hardcoded Key**:
    - Use sparingly and only in simple test or prototype scenarios.
    - Avoid in production to reduce security risks.
2. **Dynamic Key Generation**:
    - Always preferred for production environments.
    - Enhances security by generating unique keys.
3. **Key Management**:
    - Store dynamically generated keys securely.
    - Use Base64 encoding for easier storage and transfer but decode it back for cryptographic operations.
4. **JWT Workflow**:
    - Generate a key (dynamically or using a secure environment variable).
    - Use the key to sign tokens during authentication.
    - Decode the key when verifying tokens.

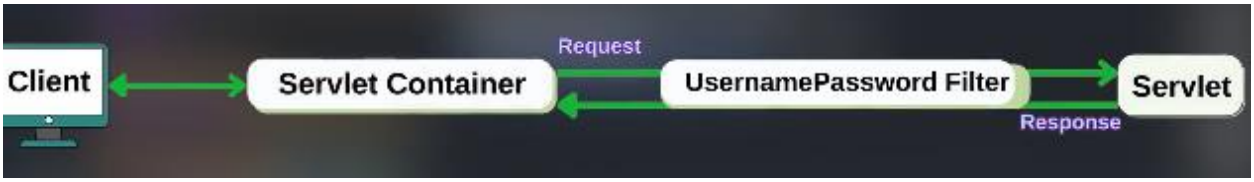
9.Creating A JWT Filter

We created the token, but on the server side we need to verify during sending the request.

By default, your Spring Security verifies using UsernamePasswordAuthentication. Add filter between that



By default spring security uses some filter



Now we add one more filter.

1. Adding a JWT Filter to the Security Configuration

Example:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(request -> request
            .requestMatchers("/register", "/login").permitAll()
            .anyRequest().authenticated()
        )
        .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

Explanation:

- **SecurityFilterChain Bean**:
  - Configures the HTTP security settings for the application.
  - Allows specific routes (e.g., /register and /login) to be accessed without authentication while securing all other endpoints.
- **Session Management**:
  - Configures the application to be **stateless** (using SessionCreationPolicy.STATELESS), as JWT-based authentication doesn't rely on server-side session storage.
- **Adding JwtFilter**:
  - **addFilterBefore()** is used to add the custom JwtFilter to the security filter chain.
  - The filter is executed **before** the UsernamePasswordAuthenticationFilter, which handles basic username-password authentication.

2. Creating the JwtFilter Class

Example:

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.filter.OncePerRequestFilter;
import java.io.IOException;

public class JwtFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws IOException, ServletException {

    }
}
```

Explanation:

- **JwtFilter**:
    - Extends OncePerRequestFilter, ensuring that the filter is executed once per request.

3. Wiring the JwtFilter in SecurityConfig

Example:

```
@Autowired
private JwtFilter jwtFilter;
```

Explanation:

- **Dependency Injection**:
    - The JwtFilter is declared as a Spring-managed bean and injected into the SecurityConfig class using the @Autowired annotation.
    - This ensures that the JwtFilter is available and properly integrated into the Spring Security filter chain.

Key Concepts:

1. CSRF Disable:

- Disables Cross-Site Request Forgery (CSRF) protection. This is common in stateless APIs secured by tokens.

2. Session Management (Stateless):

- Configures the application to not use HTTP sessions, as JWT tokens are self-contained and hold all necessary authentication information.

3. Filter Chain:

- Filters are a core part of Spring Security, enabling the application to intercept HTTP requests and apply security logic (e.g., token validation).

4. JWT Validation:

- Ensures only authenticated and authorized users can access protected endpoints. Validation can include:
    - Checking the token's signature.
    - Decoding the token to extract claims.
    - Validating the token's expiration.

10.Setting AuthToken In SecurityContext

1. Getting the Authorization Header

Example:

```
String authHeader = request.getHeader( "Authorization" );
```

Explanation:

- The Authorization header in HTTP requests is used to send the JWT.
- The JWT typically begins with the Bearer prefix, followed by the actual token.

2. Autowiring JwtService in JwtFilter

Example:

```
@Autowired
private JwtService jwtService;
```

Explanation:

- The JwtService is autowired into the JwtFilter to handle token-related operations such as extracting the username and validating the token.

3. Adding Logic in doFilterInternal

Example:

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
filterChain)
        throws ServletException, IOException {

    String authHeader = request.getHeader("Authorization");
    String token = null;
```

```
    String username = null;

    if (authHeader != null && authHeader.startsWith("Bearer ")) {
        token = authHeader.substring(7);
        username = jwtService.extractUserName(token);
    }

    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        UserDetails userDetails =
            context.getBean(UserDetailsService.class).loadUserByUsername(username);

       if (jwtService.validateToken(token, userDetails)) {
            UsernamePasswordAuthenticationToken authentication = new
                    UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
    }

    filterChain.doFilter(request, response);
}
```

Explanation:

1. **Extracting the Token**:
    - Checks if the Authorization header exists and starts with Bearer .
    - Removes the Bearer prefix to isolate the token.
2. **Extracting the Username**:
    - Calls the extractUserName method in JwtService to decode and retrieve the username from the token.
3. **Validating the Token**:
    - Ensures that:
        - The username from the token matches the UserDetails object.
        - The token is valid (e.g., it hasn't expired).
4. **Setting the Authentication**:
    - If the token is valid, creates a UsernamePasswordAuthenticationToken and sets it in the SecurityContext.

5. **Continuing the Filter Chain**:
    - Calls filterChain.doFilter() to allow further processing of the request.

4. extractUserName Method in JwtService

Example:

```
public String extractUserName(String token) {
    // Extract the username from the JWT token
}
```

Explanation:

- This method parses the JWT token to extract the username claim.
- Typically implemented using libraries like **JJWT** or **Nimbus JOSE**.

5. validateToken Method in JwtService

Example:

```
public boolean validateToken(String token, UserDetails userDetails) {
    // Logic to validate the token (e.g., checking claims, expiration, and signature)
    return true;
}
```

Explanation:

- Validates the token by:
    1. Ensuring the token's signature matches.
    2. Checking the expiration time.
    3. Verifying that the token's username matches the UserDetails object's username.

- Returns true if the token is valid, false otherwise.

Summary of the Workflow:

1. The client sends a request with a JWT in the Authorization header.
2. The JwtFilter extracts and validates the token.
3. If the token is valid, the SecurityContext is updated with authentication.
4. The request proceeds to the next stage in the filter chain.

11.Validating Token

How to validate a JWT (JSON Web Token) in a Spring-based application using a JwtService class. Below is a detailed breakdown of the code and its components, with theoretical explanations for each object and method.

Purpose of JWT Validation

JWT validation is crucial in securing APIs and ensuring that only authenticated users with valid tokens can access protected resources. It involves:

1. Decoding the token to extract claims (e.g., username).
2. Verifying the token's signature to ensure its authenticity.
3. Checking the token's expiration and other validity criteria.

Code Breakdown and Explanation

1. Extracting the Username

Example:

```
public String extractUserName(String token) {
    return extractClaim(token, Claims::getSubject);
}
```

Explanation:

- The extractUserName method retrieves the subject field from the JWT. This typically contains the username or the unique identifier of the user.
- The Claims::getSubject is a functional reference to extract the subject claim from the token.

2. Generic Claim Extraction

Example:

```
private <T> T extractClaim(String token, Function<Claims, T> claimResolver) {
    final Claims claims = extractAllClaims(token);
    return claimResolver.apply(claims);
}
```

Explanation:

- This generic method extracts a specific claim from the token using a Function to resolve the desired claim.
- It takes:
    - token: The JWT string.
    - claimResolver: A functional interface used to extract a specific claim, such as the subject or expiration time.

3. Extracting All Claims

Example:

```
private Claims extractAllClaims(String token) {
    return Jwts.parserBuilder()
            .setSigningKey(getKey())
            .build()
            .parseClaimsJws(token)
            .getBody();
}
```

Explanation:

- **Jwts.parserBuilder()**:
  - Used to create a JWT parser.
- **setSigningKey(getKey())**:
  - Sets the signing key used to validate the token's signature. The getKey() method provides the secret or public key for validation.
- **parseClaimsJws(token)**:
  - Parses the JWT and verifies its signature.
- **getBody()**:
  - Extracts the claims (payload) from the token.

4. Validating the Token

Example:

```
public boolean validateToken(String token, UserDetails userDetails) {
   final String userName = extractUserName(token);
   return (userName.equals(userDetails.getUsername()) &&
!isTokenExpired(token));
}
```

Explanation:

- **Steps in Token Validation**:
  - Extracts the username from the token using the extractUserName method.
  - Compares the extracted username with the UserDetails object's username to ensure the token belongs to the correct user.
  - Calls isTokenExpired to ensure the token has not expired.
- **Returns**:
  - true if:
    - The token username matches the UserDetails username.

        - The token is not expired.
    - false otherwise.

5. Checking Token Expiration

Example:

```
private boolean isTokenExpired(String token) {
    return extractExpiration(token).before(new Date());
}
```

Explanation:

- **extractExpiration(token)**:
    - Retrieves the token's expiration date from its claims.
- **before(new Date())**:
    - Checks if the expiration date is earlier than the current date, indicating that the token has expired.

6. Extracting Expiration Date

Example:

```
private Date extractExpiration(String token) {
    return extractClaim(token, Claims::getExpiration);
}
```

Explanation:

- Uses the extractClaim method to retrieve the expiration claim from the JWT.
- The Claims::getExpiration functional reference is used to access the expiration field in the claims.

Supporting Method: getKey

While not explicitly shown in the provided code, the getKey method is assumed to provide the key (either a secret key or public key) used for signing the JWT. For example:

Example:

```
private Key getKey() {
    byte[] keyBytes = Decoders.BASE64.decode(secretKey);
    return Keys.hmacShaKeyFor(keyBytes);
}
```

Explanation:

- Decodes a Base64-encoded secret key into bytes.
- Converts the byte array into an HMAC-SHA key using Keys.hmacShaKeyFor.

Workflow Summary

1. **Extract Username**:
    - The extractUserName method decodes the token and retrieves the subject field, which represents the username.
2. **Validate Token**:
    - Checks that the token belongs to the correct user and that it has not expired.
3. **Verify Signature**:
    - The extractAllClaims method ensures the token's integrity by verifying its signature with the signing key.
4. **Expiration Check**:
    - Ensures the token has not exceeded its validity period.

Key Objects and Concepts

- **JWT (JSON Web Token)**:
  - A compact, URL-safe token format used for securely transmitting information between parties.
- **Claims**:
  - The payload of the JWT, which contains user-specific data (e.g., username) and metadata (e.g., expiration time).
- **Signature**:
  - Ensures the token has not been tampered with by verifying it using the signing key.
- **Jwts Class**:
  - Part of the **JJWT** library, it provides utilities for parsing and generating JWTs.

Practical Use Case

- The validateToken method ensures that only requests with valid, unexpired JWTs are processed further.
- If validation fails (e.g., due to token tampering or expiration), the user is denied access.

This design ensures robust, stateless security for APIs using JWT authentication.

12. JWT Summary

1.User Authentication (Login):

- **Endpoint**: /login
- **Process**:
  - User submits credentials (username/password) via POST request.
  - Credentials are verified using AuthenticationManager with a UsernamePasswordAuthenticationToken.
  - On successful authentication, a **JWT token** is generated.

2.JWT Token Generation:

- **Class**: JwtService
- **Token Construction**:
  - Claims (e.g., username, roles) are added using setClaims(Map<String, Object>).
  - The token includes metadata such as subject, issuedAt, and expiration.
  - The token is signed using a **HMAC-SHA256** algorithm with a secret key.
  - Output: A compact, self-contained token string.

3.Token Issuance:

- The token is returned to the client in the login response.
- **Client Responsibility**: Store the token securely (e.g., in localStorage or as an HTTP-only cookie).

4.Request with Token:

- For protected endpoints, the client includes the JWT in the Authorization header using the format: Authorization: Bearer <JWT>.

5.JWT Validation on API Requests:

- **Filter**: JwtFilter (extends OncePerRequestFilter)
- **Flow**:
  - Extract the Authorization header.

        - Validate the token signature using the secret key (Key object via Keys.hmacShaKeyFor()).
        - Decode claims using Jwts.parserBuilder().parseClaimsJws(token).getBody().
        - Check token validity:
            - **Signature**: Ensures the token hasn't been tampered with.
            - **Expiration**: Confirms the token is not expired using Claims.getExpiration().

6.Authentication Context Update:

- If the token is valid:
    - Extract the username via extractUserName().
    - Load UserDetails from the user store (via UserDetailsService).
    - Create a UsernamePasswordAuthenticationToken and set it in SecurityContextHolder.
- If invalid:
    - Deny access or return an unauthorized response.

7.Security Filter Chain Configuration:

- **Session Policy**: SessionCreationPolicy.STATELESS (No server-side sessions).
- **CSRF**: Disabled for token-based security.
- **Filters**: Custom JwtFilter added before the UsernamePasswordAuthenticationFilter.

8.Token Claims and Validation:

- Claims extracted (e.g., username, roles) using functional interfaces like Claims::getSubject.
- Token is validated to ensure:
    - Subject matches authenticated user.
    - Token is not expired (using extractExpiration()).

9.JWT Libraries and Key Management:

- **Library**: io.jsonwebtoken (JJWT).
- **Key Management**:
    - Secret key dynamically generated or securely configured using environment variables.

- Base64 encoding for portability, decoded for cryptographic operations.

10.Post-Validation Request Flow:

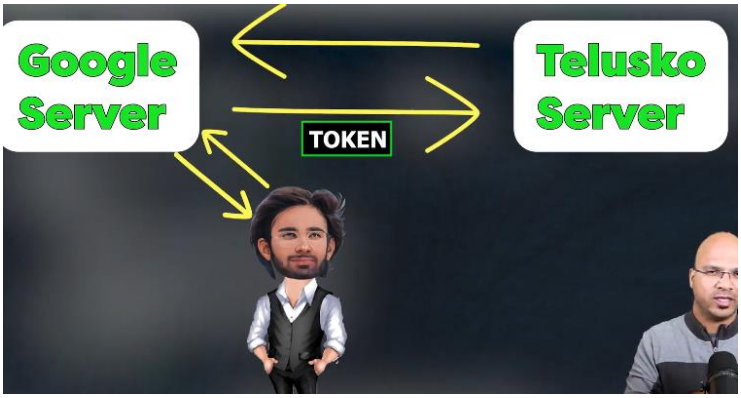
- Once authenticated, the request proceeds to the controller.
- Authorization checks are performed based on roles or permissions included in the token.

End-to-End Lifecycle

1. **Login** → User authenticated → JWT issued.
2. **Request** → JWT provided → Validated → User authenticated.
3. **Protected Resource Access** → Authorization ensured via claims.

This flow enables stateless, secure, and scalable API authentication using JWT.

13.Implementing OAuth2



Implementing Google and GitHub OAuth2 Login in Spring Boot

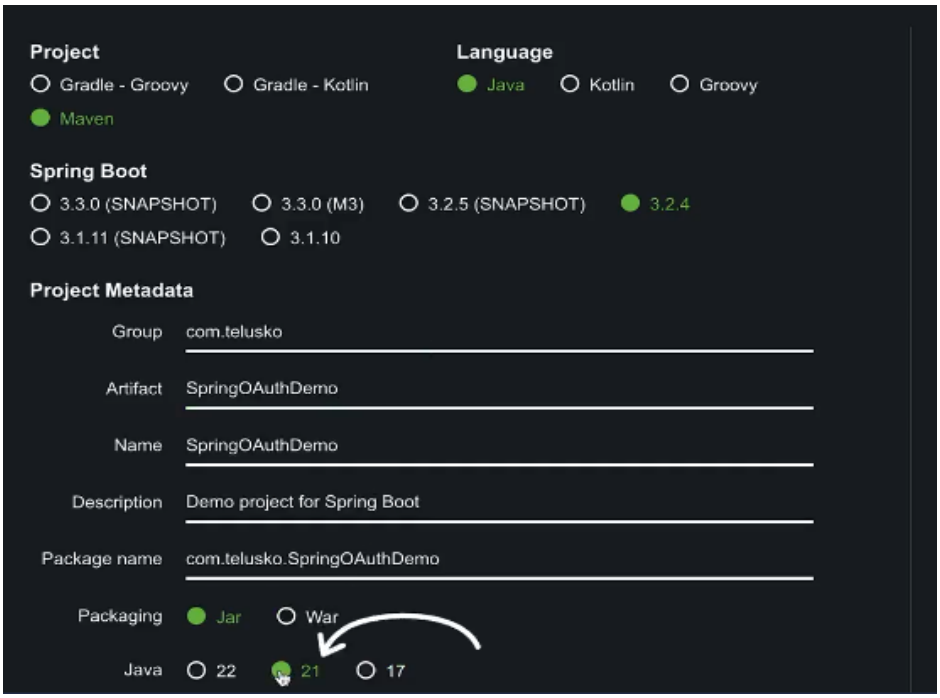
This guide will help you set up **Google OAuth2 Login** and **GitHub OAuth2 Login** for your Spring Boot application. It includes the configurations and steps required to integrate both providers successfully.

Project Setup

1. Prerequisites

- **Java 17+**
- **Spring Boot 3.0+**
- Maven
- IDE (e.g., IntelliJ, Eclipse)

- Google and GitHub developer accounts to register your application.



2. Maven Dependencies

Add the following dependencies in your pom.xml for Spring Security OAuth2 support:

```
<dependencies>
    <!-- Spring Security OAuth2 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>

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

14.Google OAuth2 Login

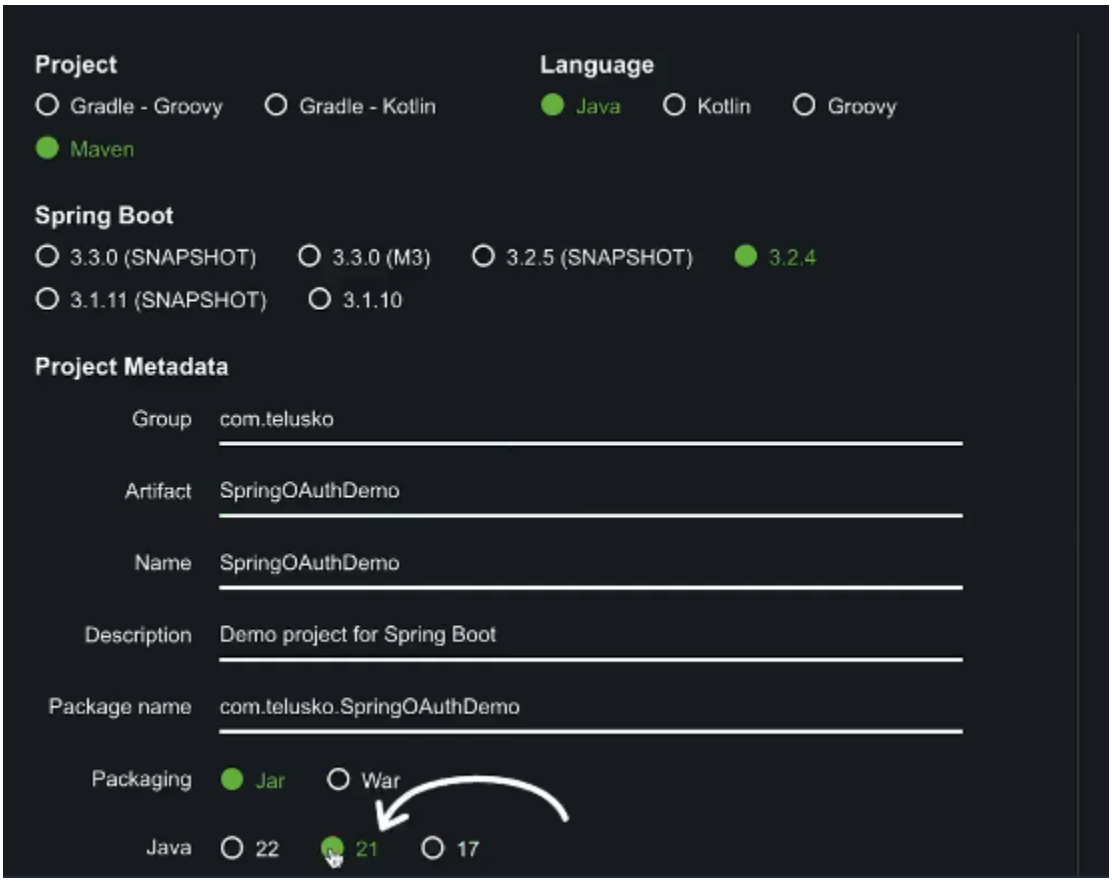
Implementing Google OAuth2 Login in Spring Boot

This guide will help you set up **Google OAuth2 Login** for your Spring Boot application. It includes the configurations and steps required to integrate both providers successfully.

Project Setup

1. Prerequisites

- **Java 17+**
- **Spring Boot 3.0+**
- Maven
- IDE (e.g., IntelliJ, Eclipse)
- Google developer accounts to register your application.



2. Maven Dependencies

Add the following dependencies in your pom.xml for Spring Security OAuth2 support:

```
<dependencies>
    <!-- Spring Security OAuth2 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Spring Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

3. Create the Security Configuration

The **SecurityConfig** class configures the security settings and enables OAuth2 login.

Example:

```
package com.telusko.springoauthdemo;

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

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated() // All requests require authentication
            )
            .oauth2Login(Customizer.withDefaults()); // Enable OAuth2 login
```

```
        return http.build();
    }
}
```

4. Create a REST Controller

The **HelloController** class defines a simple endpoint for testing OAuth2 authentication.

Example:

```
package com.telusko.springoauthdemo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String greet() {
        return "Welcome to Telusko";
    }
}
```

5. Configure application.properties

Add your Google OAuth2 credentials in the application.properties file. Replace the placeholders with your credentials.

Example:

```
# Application name
spring.application.name=SpringOAuthDemo

# Google OAuth2 credentials
spring.security.oauth2.client.registration.google.client-id=<your-google-client-id>
spring.security.oauth2.client.registration.google.client-secret=<your-google-client-secret>
```

6. Obtain OAuth2 Credentials

Google OAuth2

1. Go to Google Cloud Console.
2. Create a new project or select an existing one.
3. Navigate to **APIs & Services > Credentials**.
4. Create an **OAuth 2.0 Client ID**:
    - Application type: **Web application**
    - Authorized redirect URI: http://localhost:8080/login/oauth2/code/google
5. Copy the **Client ID** and **Client Secret** and add them to your application.properties.

7. Run the Application

1. Start your Spring Boot application by running the main class.
2. Visit http://localhost:8080/hello.
3. You will be redirected to a login page where you can select **Google** for authentication.
4. Once authenticated, you will see the Welcome to Telusko message.

---

8. Additional Configuration (Optional)

Custom Redirect After Login

To redirect users to a specific page after login, configure the DefaultOAuth2UserService:

```
http
    .oauth2Login(oauth2 -> oauth2
        .defaultSuccessUrl("/hello", true) // Redirect to /hello after login
    );
```

Customizing Login Page

To use a custom login page, add:

```
http
    .oauth2Login(oauth2 -> oauth2
        .loginPage("/custom-login") // Replace with your custom login page endpoint
    );
```

---

9. Testing

Google Authentication:

- Visit http://localhost:8080/login/oauth2/code/google.
- Authenticate using your Google account.

---

10. Key Components in OAuth2

1. **SecurityFilterChain**: Configures the Spring Security filter chain to enable OAuth2 login.
2. **application.properties**: Stores the OAuth2 client details for Google.
3. **OAuth2 Client**: Spring Security uses spring-boot-starter-oauth2-client to handle authentication flows.
4. **Authorized Redirect URIs**: Ensures the authentication server

15.Github Login

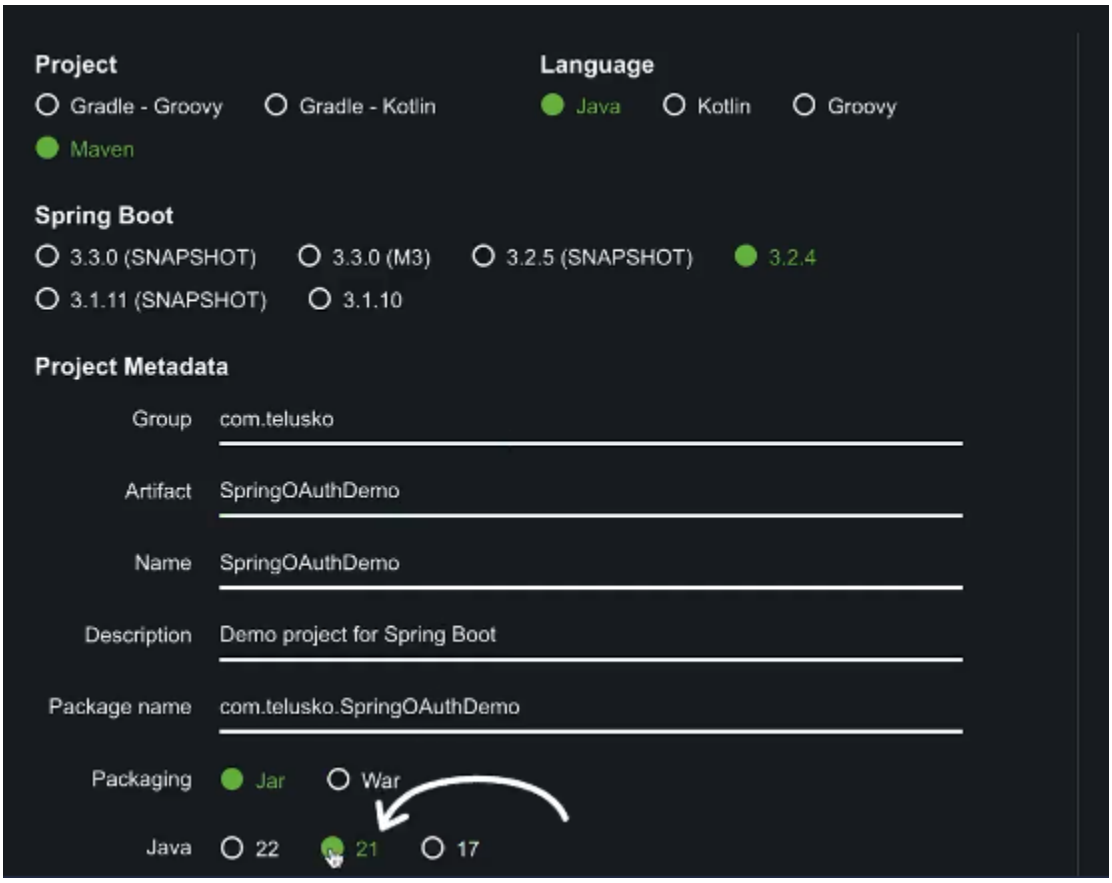
Implementing GitHub OAuth2 Login in Spring Boot

This guide will help you set up **GitHub OAuth2 Login** for your Spring Boot application. It includes the configurations and steps required to integrate both providers successfully.

Project Setup

1. Prerequisites

- **Java 17+**
- **Spring Boot 3.0+**
- Maven
- IDE (e.g., IntelliJ, Eclipse)
- GitHub developer accounts to register your application.



TELUSKO

2. Maven Dependencies

Add the following dependencies in your pom.xml for Spring Security OAuth2 support:

```
<dependencies>
    <!-- Spring Security OAuth2 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Spring Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

3. Create the Security Configuration

The SecurityConfig class configures the security settings and enables OAuth2 login.

Example:

```
package com.telusko.springoauthdemo;

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

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
```

```
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated() // All requests require authentication
            )
            .oauth2Login(Customizer.withDefaults()); // Enable OAuth2 login

        return http.build();
    }
}
```

4. Create a REST Controller

The HelloController class defines a simple endpoint for testing OAuth2 authentication.

Code:

```
package com.telusko.springoauthdemo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String greet() {
        return "Welcome to Telusko";
    }
}
```

5. Configure application.properties

Add your GitHub OAuth2 credentials in the application.properties file. Replace the placeholders with your credentials.

Code:

```
# Application name
spring.application.name=SpringOAuthDemo

# GitHub OAuth2 credentials
spring.security.oauth2.client.registration.github.client-id=<your-github-client-id>
spring.security.oauth2.client.registration.github.client-secret=<your-github-client-secret>
```

6. Obtain OAuth2 Credentials

GitHub OAuth2

1. Go to GitHub Developer Settings.
2. Click **New OAuth App** and fill out the details:
    - Homepage URL: http://localhost:8080
    - Authorization callback URL: http://localhost:8080/login/oauth2/code/github
3. Register the application and copy the **Client ID** and **Client Secret** into your application.properties.

7. Run the Application

1. Start your Spring Boot application by running the main class.
2. Visit http://localhost:8080/hello.
3. You will be redirected to a login page where you can select **GitHub** for authentication.
4. Once authenticated, you will see the Welcome to Telusko message.

8. Additional Configuration (Optional)

Custom Redirect After Login

To redirect users to a specific page after login, configure the DefaultOAuth2UserService:

```
http
    .oauth2Login(oauth2 -> oauth2
        .defaultSuccessUrl("/hello", true) // Redirect to /hello after login
    );
```

Customizing Login Page

To use a custom login page, add:

```
http
    .oauth2Login(oauth2 -> oauth2
        .loginPage("/custom-login") // Replace with your custom login page endpoint
    );
```

9. Testing

1. **GitHub Authentication:**
    - Visit http://localhost:8080/login/oauth2/code/github.
    - Authenticate using your GitHub account.

10. Key Components in OAuth2

1. **SecurityFilterChain**: Configures the Spring Security filter chain to enable OAuth2 login.
2. **application.properties**: Stores the OAuth2 client details for GitHub.
3. **OAuth2 Client**: Spring Security uses spring-boot-starter-oauth2-client to handle authentication flows.
4. **Authorized Redirect URIs**: Ensures the authentication server