Token Based Authentication

Token-based authentication is a popular method for securing APIs, especially in microservices and stateless applications. The most commonly used token-based authentication method in modern applications is **JWT (JSON Web Token)**. Let's dive into how token-based authentication works in Spring Boot.

1. Overview of Token-Based Authentication:

- **Token-based authentication** allows users to authenticate with a system and then use the generated token for subsequent requests.
- Unlike session-based authentication, where the server stores user session data, token-based authentication is stateless. The token itself contains all the necessary information for authentication and authorization.

2. JSON Web Token (JWT):

- **JWT** is a compact, URL-safe token format that encodes claims or information between two parties (e.g., a client and a server).
- A JWT consists of three parts:
    1. **Header**: Specifies the type of token and the signing algorithm (e.g., HMAC SHA256).
    2. **Payload**: Contains the claims, including user information, roles, and expiration.
    3. **Signature**: Ensures the token's integrity and authenticity by verifying the header and payload with a secret key.

3. How JWT Authentication Works:

- **Step 1: User Authentication**:
    - The user sends a login request with credentials (username and password) to the server.
    - The server authenticates the user (e.g., via a database lookup) and, if successful, generates a JWT.
- **Step 2: Token Issuance**:

    - The server sends the JWT back to the client, typically in the response body.
- **Step 3: Subsequent Requests**:
    - For each subsequent request, the client includes the JWT in the `Authorization` header (usually prefixed with `Bearer`).
    - The server validates the token by checking the signature and expiration time.
- **Step 4: Token Validation**:
    - If the token is valid, the server7 processes the request. If invalid, the server responds with an authentication error.

Implementing JWT Authentication

To implement JWT authentication in the latest Spring Boot 3, the steps are slightly different due to some changes in the framework and dependencies. Below is a step-by-step guide:

1. Add Required Dependencies

First, add the necessary dependencies to your `pom.xml` for Spring Security and JWT:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
```

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

2. Create the JWT Utility Class

Create a utility class to handle JWT creation, parsing, and validation:

```java

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;
import java.util.function.Function;

@Component
public class JwtUtil {

    private Key key = Keys.secretKeyFor(SignatureAlgorithm.
HS256);

    @Value("${jwt.expiration}")
    private long jwtExpiration;
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

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

    public <T> T extractClaim(String token, Function<Claim
s, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder().setSigningKey(key).buil
d().parseClaimsJws(token).getBody();
    }

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

    public String generateToken(String username) {
        return createToken(username);
    }

    private String createToken(String subject) {
        return Jwts.builder()
                .setSubject(subject)
                .setIssuedAt(new Date(System.currentTimeMil
lis()))
                .setExpiration(new Date(System.currentTimeM
illis() + jwtExpiration))
                .signWith(key)
                .compact();
    }
    public Boolean validateToken(String token, String usern
ame) {
        final String tokenUsername = extractUsername(toke
n);
        return (tokenUsername.equals(username) && !isTokenE
xpired(token));
    }
}
```

3. Create a Custom `UserDetailsService`

This service will load the user details from the database or any other source.

```

import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.core.userdetails.Userna
meNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetail
sService {

    @Override
    public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {
        // Load user from database, this is just a placehol
der
        return org.springframework.security.core.userdetail
s.User
                .withUsername(username)
                .password("{noop}password") // {noop} is ju
st for demonstration purposes
                .roles("USER")
                .build();
    }
}
```

4. Create the JWT Authentication Filter

This filter will intercept the requests and validate the JWT.

```
javaCopy code
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.security.authentication.Username
PasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityCo
ntextHolder;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.web.authentication.WebA
uthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
public class JwtAuthenticationFilter extends OncePerRequest
Filter {

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private CustomUserDetailsService userDetailsService;
    @Override
    protected void doFilterInternal(HttpServletRequest requ
est, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        final String authorizationHeader = request.getHeade
r("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHea
der.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getCo
ntext().getAuthentication() == null) {

            UserDetails userDetails = this.userDetailsServi
ce.loadUserByUsername(username);

            if (jwtUtil.validateToken(jwt, userDetails.getU
sername())) {

                UsernamePasswordAuthenticationToken usernam
ePasswordAuthenticationToken = new UsernamePasswordAuthenti
cationToken(
                        userDetails, null, userDetails.getA
uthorities());
                usernamePasswordAuthenticationToken
                        .setDetails(new WebAuthenticationDe
tailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthe
ntication(usernamePasswordAuthenticationToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

5. Configure Spring Security

In Spring Boot 3, the security configuration is done differently compared to previous versions. Here’s how you can set it up:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.authentication.Authenti
cationManager;
import org.springframework.security.config.annotation.authe
ntication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.config.annotation.web.c
onfiguration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCrea
tionPolicy;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.crypto.bcrypt.BCryptPas
swordEncoder;
import org.springframework.security.crypto.password.Passwor
dEncoder;
import org.springframework.security.web.SecurityFilterChai
n;
import org.springframework.security.web.authentication.User
namePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilte
r;

    @Bean
    public UserDetailsService userDetailsService() {
        return new CustomUserDetailsService();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(Auth
enticationConfiguration authenticationConfiguration) throws
Exception {
        return authenticationConfiguration.getAuthenticatio
nManager();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http.csrf().disable()
                .authorizeHttpRequests()
                .requestMatchers("/api/auth/**").permitAll
()
                .anyRequest().authenticated()
                .and()
                .sessionManagement().sessionCreationPolicy
(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(jwtAuthenticationFilter, Usern
amePasswordAuthenticationFilter.class);

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

6. Create Authentication Controller

This controller will handle login requests and return a JWT token upon successful authentication.

```
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.security.authentication.Authenti
cationManager;
import org.springframework.security.authentication.Username
PasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationExce
ption;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.web.bind.annotation.*;

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

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private CustomUserDetailsService userDetailsService;

    @PostMapping("/login")
    public String createAuthenticationToken(@RequestBody Au
thRequest authRequest) throws Exception {
        try {
            Authentication authentication = authenticationM
anager.authenticate(
                    new UsernamePasswordAuthenticationToken
(authRequest.getUsername(), authRequest.getPassword())
            );
        } catch (AuthenticationException e) {
            throw new Exception("Incorrect username or pass
word", e);
        }

        final UserDetails userDetails = userDetailsService.
loadUserByUsername(authRequest.getUsername());
        return jwtUtil.generateToken(userDetails.getUsernam
e());
    }
}

class AuthRequest {
    private String username;
    private String password;

    // getters and setters
}
```

7. Testing the Application

- **Login**: Send a POST request to `/api/auth/login` with a JSON body containing the username and password. If the credentials are correct, the server will return a JWT token.
- **Access Secured Endpoints**: Use the JWT token in the `Authorization` header (e.g., `Authorization: Bearer <token>`) to access secured endpoints in your application.

8. Security Best Practices

- **Use Strong Encryption**: Replace `{noop}` with a strong password encoder like `BCryptPasswordEncoder`.

- **Token Expiry**: Ensure that your JWTs have a reasonable expiration time and consider implementing token refresh logic.
- **Secure Key Management**: Store your signing key securely, possibly in an environment variable or a secret management service.

Cors

What is CORS?

- **CORS** is a mechanism that uses HTTP headers to tell browsers to give a web application running at one origin (domain) permission to access selected resources from a different origin.
- This security feature is enforced by browsers to prevent malicious scripts from making unauthorized requests to sensitive endpoints.

2. How CORS Works:

- When a request is made from a different origin, the browser sends a **preflight request** (an `OPTIONS` request) to the server to check whether the actual request is safe to send.
- The server responds with specific CORS headers that indicate whether the actual request can proceed.
- If the server allows the request, the browser sends the actual request; otherwise, the browser blocks the request.

Understanding Key CORS Headers:

- **Access-Control-Allow-Origin**: Specifies the allowed origin(s). Use `"*"` to allow all origins.
- **Access-Control-Allow-Methods**: Specifies the allowed HTTP methods (e.g., `GET`, `POST`, `PUT`).
- **Access-Control-Allow-Headers**: Specifies the allowed request headers (e.g., `Content-Type`, `Authorization`).
- **Access-Control-Allow-Credentials**: If true, allows credentials (like cookies) to be sent with the request.
- **Access-Control-Max-Age**: Specifies how long the results of a preflight request can be cached.

Define a `CorsConfigurationSource` Bean

The `CorsConfigurationSource` bean defines the CORS settings for your application. This includes allowed origins, methods, headers, and other configurations.

```
javaCopy code
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurati
onSource;
import org.springframework.web.cors.CorsConfigurationSourc
e;

@Configuration
public class CorsConfig {

    @Bean
    public CorsConfigurationSource corsConfigurationSource
() {
        CorsConfiguration configuration = new CorsConfigura
tion();
        configuration.addAllowedOrigin("http://localhost:30
00"); // Replace with your frontend URL
        configuration.addAllowedMethod("*"); // Allow all H
TTP methods
        configuration.addAllowedHeader("*"); // Allow all h
eaders
        configuration.setAllowCredentials(true); // Allow c
redentials (e.g., cookies)

        UrlBasedCorsConfigurationSource source = new UrlBas
edCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configurati
on); // Apply to all endpoints
        return source;
    }
}
```

Integrate `CorsConfigurationSource` with Spring Security

Now that you have the `CorsConfigurationSource` bean, you need to integrate it into your Spring Security configuration.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.config.annotation.web.c
onfiguration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChai
n;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final CorsConfigurationSource corsConfiguration
Source;

    public SecurityConfig(CorsConfigurationSource corsConfi
gurationSource) {
        this.corsConfigurationSource = corsConfigurationSou
rce;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http
            .cors(cors -> cors.configurationSource(corsConf
igurationSource)) // Use the CorsConfigurationSource bean
```

```
            .csrf(csrf -> csrf.disable()) // Disable CSRF for simplicity, enable it as needed
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/public/**").permitAll() // Public endpoints
                .anyRequest().authenticated() // All other endpoints require authentication
            )
            .formLogin(withDefaults()) // Default login form
            .logout(withDefaults()); // Default logout

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

Explanation

- **CorsConfigurationSource**: This is a Spring interface that defines the CORS configuration for your application. The implementation provided here uses `UrlBasedCorsConfigurationSource`, which allows mapping specific CORS configurations to different URL patterns.
- **HttpSecurity Configuration**: In the security configuration (`SecurityFilterChain`), you enable CORS by calling the `cors()` method and passing your custom `CorsConfigurationSource` bean.