1. Add Dependencies

Make sure you have the following dependencies in your `pom.xml` for JWT and Spring Security:

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

    <!-- JWT Library -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.11.5</version>
    </dependency>

    <!-- Spring Boot Starter Web (For REST API support) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data JPA (For database integration) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Spring Boot Starter Validation -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
</dependencies>
```

---

2. Create JWT Utility Class

This class will contain methods for generating, validating, and parsing JWT tokens.

```
package com.example.security.config;

import io.jsonwebtoken.*;
import java.util.Date;
import org.springframework.stereotype.Component;

@Component
public class JwtTokenUtil {

    private final String SECRET_KEY = "your-secret-key"; // Use a stronge

    // Generate JWT token
    public String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 8640000
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
            .compact();
    }

    // Validate JWT token
    public boolean validateToken(String token) {
        try {
            Jwts.parser()
                .setSigningKey(SECRET_KEY)
                .parseClaimsJws(token);
            return true;
        } catch (JwtException e) {
            return false;
        }
    }

    // Extract username from JWT token
    public String getUsernameFromToken(String token) {
        return Jwts.parser()
            .setSigningKey(SECRET_KEY)
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
    }
}
```

3. Create JWT Authentication Filter

This filter intercepts every request and extracts the JWT token from the **Authorization** header. If the token is valid, it sets the `Authentication` object in the Spring Security context.

```
package com.example.security.filters;

import com.example.security.config.JwtTokenUtil;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtTokenUtil jwtTokenUtil;

    public JwtAuthenticationFilter(JwtTokenUtil jwtTokenUtil) {
        this.jwtTokenUtil = jwtTokenUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, FilterCha
            throws ServletException, IOException {

        String token = request.getHeader("Authorization");

        if (token != null && token.startsWith("Bearer ")) {
            token = token.substring(7);  // Remove 'Bearer ' prefix

            if (jwtTokenUtil.validateToken(token)) {
                String username = jwtTokenUtil.getUsernameFromToken(toker
                // Set the authentication context if the token is valid
                SecurityContextHolder.getContext().setAuthentication(new
            }
        }
```

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

---

4. Implement `UserDetailsService`

The `UserDetailsService` is responsible for fetching user details from your database for authentication. It's used by Spring Security to authenticate and load user information based on the JWT token.

```
package com.example.security.service;

import com.example.security.model.User;
import com.example.security.repository.UserRepository;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundExce
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws Usernam
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User nc
        return new org.springframework.security.core.userdetails.User(use
    }
}
```

---

5. Configure Spring Security

We will now configure Spring Security to use the `JwtAuthenticationFilter` and `UserDetailsService`.

```
package com.example.security.config;

import com.example.security.filters.JwtAuthenticationFilter;
import com.example.security.service.CustomUserDetailsService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSe
import org.springframework.security.config.annotation.web.configuration.E
import org.springframework.security.config.annotation.web.configuration.W
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAu

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final CustomUserDetailsService userDetailsService;
    private final JwtTokenUtil jwtTokenUtil;

    public SecurityConfig(CustomUserDetailsService userDetailsService, Jw
        this.userDetailsService = userDetailsService;
        this.jwtTokenUtil = jwtTokenUtil;
    }

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/auth/**").permitAll()  // Allow public access
            .anyRequest().authenticated()       // Require authenticatior
            .and()
            .addFilterBefore(new JwtAuthenticationFilter(jwtTokenUtil), U
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Ex
        auth.userDetailsService(userDetailsService).passwordEncoder(passw
    }
}
```

6. Create Authentication Controller

The authentication controller provides endpoints for logging in and generating the JWT token.

```
package com.example.security.controllers;

import com.example.security.config.JwtTokenUtil;
import com.example.security.service.CustomUserDetailsService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

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

    private final JwtTokenUtil jwtTokenUtil;
    private final CustomUserDetailsService userDetailsService;

    @Autowired
    public AuthController(JwtTokenUtil jwtTokenUtil, CustomUserDetailsSer
        this.jwtTokenUtil = jwtTokenUtil;
        this.userDetailsService = userDetailsService;
    }

    // Login and generate JWT token
    @PostMapping("/login")
    public String login(@RequestParam String username, @RequestParam Stri
        // Authenticate user (simplified)
        if ("user".equals(username) && "password".equals(password)) {
            return jwtTokenUtil.generateToken(username);  // Return the g
        }
        return "Invalid Credentials";
    }
}
```

7. Database Setup

Make sure to set up a `User` model and `UserRepository` for fetching user data from the database:

User Model

```
package com.example.security.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class User {

    @Id
    private Long id;
    private String username;
    private String password;

    // Getters and Setters
}
```

User Repository

```
package com.example.security.repository;

import com.example.security.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

---

8. Test the JWT Authentication

- Send a **POST** request to `/auth/login` with `username` and `password`.
- If credentials are correct, a JWT token will be returned.
- Use the returned JWT token in the **Authorization** header (`Bearer <token>`) for subsequent requests to protected endpoints.
