This class configures the security mechanisms, defines endpoint access permissions, and integrates JWT for stateless authentication.

**Code Walkthrough: SecurityConfig**

Here’s the full SecurityConfig class for reference:

@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
 @Autowired  
 private OurUserDetailsService ourUserDetailsService;  
  
 @Autowired  
 private JWTAuthFilter jwtAuthFilter;  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.csrf(AbstractHttpConfigurer::disable)  
 .cors(Customizer.withDefaults())  
 .authorizeHttpRequests(request -> request  
 .requestMatchers("/auth/\*\*", "/public/\*\*").permitAll()  
 .requestMatchers("/admin/\*\*").hasAnyAuthority("ADMIN")  
 .requestMatchers("/user/\*\*").hasAnyAuthority("USER")  
 .requestMatchers("/adminuser/\*\*").hasAnyAuthority("ADMIN", "USER")  
 .anyRequest().authenticated())  
 .sessionManagement(manager -> manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))  
 .authenticationProvider(authenticationProvider())  
 .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);  
  
 return httpSecurity.build();  
 }  
  
 @Bean  
 public AuthenticationProvider authenticationProvider() {  
 DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();  
 daoAuthenticationProvider.setUserDetailsService(ourUserDetailsService);  
 daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());  
 return daoAuthenticationProvider;  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {  
 return authenticationConfiguration.getAuthenticationManager();  
 }  
}

**Step 1: Understanding the Class Components**

**Annotations:**

**@Configuration**: Indicates this class contains Spring configuration beans.

**@EnableWebSecurity**: Activates web security for the application.

**Autowired Fields:**

**OurUserDetailsService**: Custom service for loading user details (e.g., username, roles).

**JWTAuthFilter**: Custom JWT filter to validate tokens for every request.

**Step 2: SecurityFilterChain Method**

This method configures the HTTP security for the application. Here’s how it works:

**1. Disable CSRF**

httpSecurity.csrf(AbstractHttpConfigurer::disable)

**What it does**: Disables Cross-Site Request Forgery (CSRF) protection since APIs often use stateless authentication (like JWT), making CSRF irrelevant.

**2. Enable CORS**

.cors(Customizer.withDefaults())

**What it does**: Allows cross-origin requests from other domains (e.g., Angular frontend running on a different port).

**3. Endpoint Authorization**

.authorizeHttpRequests(request -> request  
 .requestMatchers("/auth/\*\*", "/public/\*\*").permitAll() // Public paths  
 .requestMatchers("/admin/\*\*").hasAnyAuthority("ADMIN") // Admin-only paths  
 .requestMatchers("/user/\*\*").hasAnyAuthority("USER") // User-only paths  
 .requestMatchers("/adminuser/\*\*").hasAnyAuthority("ADMIN", "USER") // Accessible by both roles  
 .anyRequest().authenticated()) // All other paths need login

**What it does**:

🔹Grants unrestricted access to /auth/\*\* and /public/\*\*.

🔹Restricts /admin/\*\* to users with the **ADMIN** role.

🔹Restricts /user/\*\* to users with the **USER** role.

🔹Allows both roles to access /adminuser/\*\*.

**4. Stateless Session Management**

.sessionManagement(manager -> manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

**What it does**: Configures the app to not store user sessions on the server. Each request must include a valid JWT token for authentication.

**5. Add JWT Filter**

.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

**What it does**: Adds the JWTAuthFilter to validate tokens before the UsernamePasswordAuthenticationFilter processes authentication.

**Step 3: AuthenticationProvider Method**

@Bean  
public AuthenticationProvider authenticationProvider() {  
 DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();  
 daoAuthenticationProvider.setUserDetailsService(ourUserDetailsService);  
 daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());  
 return daoAuthenticationProvider;  
}

**Purpose:**Verifies user credentials and loads user details from the database.

**Components**:

1. **DaoAuthenticationProvider**: Uses our custom OurUserDetailsService to fetch user data.
2. **PasswordEncoder**: Encodes passwords using BCryptPasswordEncoder for secure storage.

**How It Works**

When a user attempts to log in:

🔹The DaoAuthenticationProvider fetches user details using the OurUserDetailsService.

🔹It verifies the provided password against the stored (encoded) password.

🔹If the credentials are valid, the user is authenticated.

**Step 4: PasswordEncoder Method**

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

**Purpose**: Encodes passwords securely using the BCrypt hashing algorithm.

**Why BCrypt?**

🔹BCrypt adds a random “salt” to passwords before hashing, making it resistant to rainbow table attacks.

🔹It’s widely used and trusted for secure password storage.

**Step 5: AuthenticationManager Method**

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

**Purpose**: Retrieves the default AuthenticationManager, which integrates all the authentication providers configured in the application.

**How It Works**:

🔹Combines all AuthenticationProvider beans (e.g., DaoAuthenticationProvider) to handle different authentication scenarios.

🔹Returns the default AuthenticationManager, which will authenticate users during login.