# Complete Spring Security Learning Plan with JWT Implementation

## Table of Contents

---

## 1. Learning Roadmap

### Phase 1: Foundation (Week 1-2)

- [ ] Understand servlet filters and filter chain
- [ ] Learn Spring Security filter chain architecture
- [ ] Study authentication vs authorization concepts
- [ ] Explore security contexts and principals

### Phase 2: Core Components (Week 3-4)

- [ ] Security Configuration deep dive
- [ ] UserDetailsService implementation
- [ ] Authentication providers and managers
- [ ] Password encoders and validation

### Phase 3: JWT Implementation (Week 5-6)

- [ ] JWT theory and structure
- [ ] Custom JWT filters
- [ ] Token generation and validation
- [ ] Complete authentication flow

### Phase 4: Advanced Topics (Week 7-8)

- [ ] Method-level security
- [ ] OAuth2 integration

- [ ] Testing security configurations
- [ ] Production considerations

---

## 2. Core Filter Concepts

### What are Servlet Filters?

Servlet filters are Java components that intercept HTTP requests and responses before they reach the servlet or after they leave it. They form a chain where each filter can:

- **Examine requests/responses**
- **Modify requests/responses**
- **Block requests**
- **Log information**
- **Perform security checks**

### Filter Chain Execution Flow

```
HTTP Request → Filter1 → Filter2 → Filter3 → Servlet → Filter3 → Filter2 → Filter1 → HTTP Response
```

### Key Filter Concepts:

#### 1. Filter Interface

java

```java
public interface Filter {
    void init(FilterConfig config);
    void doFilter(ServletRequest request, ServletResponse response, FilterChain chain);
    void destroy();
}
```

**Role**: Base contract for all servlet filters **Responsibilities**:

- Initialize filter resources
- Process requests and responses
- Clean up resources

#### 2. FilterChain

java

```java
public interface FilterChain {
    void doFilter(ServletRequest request, ServletResponse response);
}
```

**Role**: Represents the chain of filters **Responsibilities**:

- Invoke the next filter in chain
- Pass control to servlet if last filter

**3. Filter Registration**

**Role**: Register filters with servlet container **Methods**:

- `@WebFilter` annotation
- `FilterRegistrationBean` in Spring Boot
- XML configuration (legacy)

### Basic Filter Example:

```java
@Component
public class LoggingFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                    FilterChain chain) throws IOException, ServletException {

        System.out.println("Request received: " + ((HttpServletRequest) request).getRequestURI());

        // Continue the chain
        chain.doFilter(request, response);

        System.out.println("Response sent");
    }
}
```

---

## 3. Spring Security Architecture

### Spring Security Filter Chain

Spring Security is built on servlet filters. It creates a `FilterChainProxy` that contains multiple `SecurityFilterChain` instances.

```
HTTP Request → DelegatingFilterProxy → FilterChainProxy → Security Filters → Controller
```

## Key Architectural Components:

### 1. DelegatingFilterProxy

**Role**: Bridge between Servlet container and Spring context **Responsibilities**:

- Delegate filter calls to Spring-managed beans
- Enable Spring dependency injection in filters
- Lazy initialization support

### 2. FilterChainProxy

**Role**: Central filter that manages multiple security filter chains **Responsibilities**:

- Route requests to appropriate SecurityFilterChain
- Manage multiple security configurations
- Handle exceptions from security filters

### 3. SecurityFilterChain

**Role**: Ordered list of security filters for specific request patterns **Responsibilities**:

- Define which filters apply to which URLs
- Maintain filter order
- Enable/disable specific security features

## Default Security Filter Order:

1. **ChannelProcessingFilter** - HTTP/HTTPS redirection
2. **SecurityContextPersistenceFilter** - Security context management
3. **ConcurrentSessionFilter** - Session management
4. **Authentication Filters** (UsernamePasswordAuthenticationFilter, etc.)
5. **SessionManagementFilter** - Session handling
6. **ExceptionTranslationFilter** - Exception handling
7. **FilterSecurityInterceptor** - Authorization decisions

---

# 4. Core Components Deep Dive

## 4.1 SecurityConfig (Security Configuration)

**Purpose and Role:**

- **Central configuration point** for Spring Security
- **Defines security rules** and behavior
- **Configures authentication and authorization**
- **Customizes security filter chain**

**Key Responsibilities:**

1. Configure HTTP security (URLs, methods, access rules)
2. Set up authentication mechanisms
3. Define password encoding
4. Configure session management
5. Handle CORS and CSRF
6. Set up custom filters

**Implementation Example:**

```
java
```

```java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private JwtAuthenticationEntryPoint jwtEntryPoint;

    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    // Password encoder configuration
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // Authentication manager configuration
    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    // Configure authentication provider
    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    // Main security configuration
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/products/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .exceptionHandling(ex -> ex
                .authenticationEntryPoint(jwtEntryPoint)
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .authenticationProvider(authenticationProvider())
            .addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
    }
}
```

**Configuration Breakdown:**

**HTTP Security Configuration**:

- `permitAll()` - Allow access without authentication
- `hasRole()` - Require specific role
- `authenticated()` - Require any authentication
- `sessionCreationPolicy(STATELESS)` - Disable session creation for JWT

**Filter Configuration**:

- `addFilterBefore()` - Add custom filter before existing filter
- `addFilterAfter()` - Add custom filter after existing filter

## 4.2 UserDetailsService

**Purpose and Role:**

- **Core interface for loading user data**
- **Bridge between your user storage and Spring Security**
- **Provides user information for authentication**

**Key Responsibilities:**

1. Load user by username/email
2. Return UserDetails object
3. Handle user not found scenarios
4. Interface with your user repository

**Interface Definition:**

```java
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

**Custom Implementation:**

```java
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // Find user in database
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found: " + username));

        // Convert to UserDetails
        return UserPrincipal.create(user);
    }

    // Additional method for loading by ID (useful for JWT)
    public UserDetails loadUserById(Long id) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new UsernameNotFoundException("User not found with id: " + id));

        return UserPrincipal.create(user);
    }
}
```

## 4.3 UserDetails Interface

**Purpose and Role:**

- **Represents authenticated user information**
- **Provides core user data to Spring Security**
- **Contains authorities and account status**

**Key Methods:**

```java
public interface UserDetails extends Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();
    String getPassword();
    String getUsername();
    boolean isAccountNonExpired();
    boolean isAccountNonLocked();
    boolean isCredentialsNonExpired();
    boolean isEnabled();
}
```

### Custom Implementation:

```java
```

```java
public class UserPrincipal implements UserDetails {

    private Long id;
    private String username;
    private String email;
    private String password;
    private Collection<? extends GrantedAuthority> authorities;
    private boolean enabled;
    private boolean accountNonExpired;
    private boolean accountNonLocked;
    private boolean credentialsNonExpired;

    public UserPrincipal(Long id, String username, String email, String password,
                    Collection<? extends GrantedAuthority> authorities,
                    boolean enabled, boolean accountNonExpired,
                    boolean accountNonLocked, boolean credentialsNonExpired) {
        this.id = id;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
        this.enabled = enabled;
        this.accountNonExpired = accountNonExpired;
        this.accountNonLocked = accountNonLocked;
        this.credentialsNonExpired = credentialsNonExpired;
    }

    // Factory method to create UserPrincipal from User entity
    public static UserPrincipal create(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName().name()))
            .collect(Collectors.toList());

        return new UserPrincipal(
            user.getId(),
            user.getUsername(),
            user.getEmail(),
            user.getPassword(),
            authorities,
            user.isEnabled(),
            true, // accountNonExpired
            true, // accountNonLocked
            true  // credentialsNonExpired
        );
    }
    // Implement all UserDetails methods
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return accountNonExpired;
    }

    @Override
    public boolean isAccountNonLocked() {
        return accountNonLocked;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return credentialsNonExpired;
    }

    @Override
    public boolean isEnabled() {
        return enabled;
    }

    // Additional getters
    public Long getId() {
        return id;
    }

    public String getEmail() {
        return email;
    }
}
```

## 4.4 AuthenticationManager

**Purpose and Role:**

- **Coordinates authentication process**
- **Delegates to authentication providers**
- **Returns authentication result**

**Key Responsibilities:**

1. Receive authentication requests
2. Find appropriate AuthenticationProvider
3. Delegate authentication logic
4. Return Authentication object or throw exception

**Usage Example:**

java

```java
@Service
public class AuthService {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtTokenProvider tokenProvider;

    public JwtAuthResponse authenticateUser(LoginRequest loginRequest) {

        // Create authentication token
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                loginRequest.getUsername(),
                loginRequest.getPassword()
            )
        );

        // Set authentication in security context
        SecurityContextHolder.getContext().setAuthentication(authentication);

        // Generate JWT token
        String jwt = tokenProvider.generateToken(authentication);

        return new JwtAuthResponse(jwt);
    }
}
```

### 4.5 AuthenticationProvider

**Purpose and Role:**

- **Performs actual authentication logic**
- **Validates credentials against data source**
- **Returns authenticated user or throws exception**

**Built-in Providers:**

1. **DaoAuthenticationProvider** - Database authentication
2. **LdapAuthenticationProvider** - LDAP authentication
3. **AnonymousAuthenticationProvider** - Anonymous users

**Custom Provider Example:**

```java
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {

        String username = authentication.getName();
        String password = authentication.getCredentials().toString();

        // Load user details
        UserDetails userDetails = userDetailsService.loadUserByUsername(username);

        // Verify password
        if (!passwordEncoder.matches(password, userDetails.getPassword())) {
            throw new BadCredentialsException("Invalid credentials");
        }

        // Create authenticated token
        return new UsernamePasswordAuthenticationToken(
            userDetails,
            password,
            userDetails.getAuthorities()
        );
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

## 4.6 SecurityContext

**Purpose and Role:**

- **Holds authentication information**
- **Available throughout request processing**

- **Thread-local storage**

**Key Components:**

```java
// Get current authentication
Authentication auth = SecurityContextHolder.getContext().getAuthentication();

// Check if user is authenticated
if (auth != null && auth.isAuthenticated()) {
    UserDetails userDetails = (UserDetails) auth.getPrincipal();
    String username = userDetails.getUsername();
}

// Set authentication programmatically
SecurityContextHolder.getContext().setAuthentication(authentication);
```

---

## 5. JWT Authentication Implementation

### What is JWT?

**JSON Web Token (JWT)** is a compact, URL-safe means of representing claims to be transferred between two parties. It consists of three parts:

1. **Header** - Token type and signing algorithm
2. **Payload** - Claims (user data)
3. **Signature** - Verification signature

```
JWT Structure: header.payload.signature
Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNT
```

### JWT Authentication Flow:

1. User sends credentials to `/auth/login`
2. Server validates credentials
3. Server generates JWT token
4. Client stores token (localStorage/sessionStorage)
5. Client sends token in Authorization header for protected requests
6. Server validates token and processes request

### 5.1 JWT Token Provider

**Purpose and Role:**

- **Generate JWT tokens after successful authentication**
- **Validate JWT tokens from requests**
- **Extract user information from tokens**
- **Handle token expiration**

**Implementation:**

```java
```

```
@Component
public class JwtTokenProvider {

  private static final String JWT_SECRET = "mySecretKey";
  private static final int JWT_EXPIRATION = 86400; // 24 hours in seconds

  // Generate token from authentication
  public String generateToken(Authentication authentication) {
    UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();

    Date expiryDate = new Date(System.currentTimeMillis() + JWT_EXPIRATION * 1000L);

    return Jwts.builder()
        .setSubject(userPrincipal.getId().toString())
        .setIssuedAt(new Date())
        .setExpiration(expiryDate)
        .claim("username", userPrincipal.getUsername())
        .claim("email", userPrincipal.getEmail())
        .signWith(SignatureAlgorithm.HS512, JWT_SECRET)
        .compact();
  }

  // Get user ID from token
  public Long getUserIdFromToken(String token) {
    Claims claims = Jwts.parser()
        .setSigningKey(JWT_SECRET)
        .parseClaimsJws(token)
        .getBody();

    return Long.parseLong(claims.getSubject());
  }

  // Validate token
  public boolean validateToken(String token) {
    try {
      Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(token);
      return true;
    } catch (SignatureException ex) {
      System.err.println("Invalid JWT signature");
    } catch (MalformedJwtException ex) {
      System.err.println("Invalid JWT token");
    } catch (ExpiredJwtException ex) {
      System.err.println("Expired JWT token");
    } catch (UnsupportedJwtException ex) {
      System.err.println("Unsupported JWT token");
    } catch (IllegalArgumentException ex) {
            System.err.println("JWT claims string is empty");
        }
        return false;
    }

    // Extract token from request
    public String getTokenFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7);
        }
        return null;
    }
}
```

## 5.2 JWT Authentication Filter

**Purpose and Role:**

- **Intercept HTTP requests**
- **Extract and validate JWT tokens**
- **Set authentication in SecurityContext**
- **Allow request to proceed if valid**

**Implementation:**

```java
```

```java
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtTokenProvider tokenProvider;

    @Autowired
    private CustomUserDetailsService userDetailsService;

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthenticationFilter.class);

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

        try {
            // Extract JWT token from request
            String jwt = tokenProvider.getTokenFromRequest(request);

            if (StringUtils.hasText(jwt) && tokenProvider.validateToken(jwt)) {
                // Get user ID from token
                Long userId = tokenProvider.getUserIdFromToken(jwt);

                // Load user details
                UserDetails userDetails = userDetailsService.loadUserById(userId);

                // Create authentication token
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities()
                    );

                // Set authentication details
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                // Set authentication in security context
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception ex) {
            logger.error("Could not set user authentication in security context", ex);
        }

        // Continue filter chain
        filterChain.doFilter(request, response);
    }
}
```

## 5.3 JWT Authentication Entry Point

**Purpose and Role:**

- **Handle authentication failures**
- **Send appropriate error responses**
- **Provide consistent error format**

**Implementation:**

```java
@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthenticationEntryPoint.class);

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
                         AuthenticationException authException) throws IOException {

        logger.error("Responding with unauthorized error. Message - {}", authException.getMessage());

        response.setContentType("application/json");
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        // Create error response
        Map<String, Object> errorResponse = new HashMap<>();
        errorResponse.put("error", "Unauthorized");
        errorResponse.put("message", "Full authentication is required to access this resource");
        errorResponse.put("status", 401);
        errorResponse.put("timestamp", System.currentTimeMillis());
        errorResponse.put("path", request.getRequestURI());

        // Write JSON response
        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getWriter(), errorResponse);
    }
}
```

## 6. Complete Spring Boot JWT Project

### 6.1 Project Structure

```
src/main/java/com/example/jwtauth/
├── JwtAuthApplication.java
├── config/
│   ├── SecurityConfig.java
│   └── WebConfig.java
├── controller/
│   ├── AuthController.java
│   ├── UserController.java
│   └── AdminController.java
├── dto/
│   ├── LoginRequest.java
│   ├── SignupRequest.java
│   ├── JwtResponse.java
│   └── MessageResponse.java
├── entity/
│   ├── User.java
│   ├── Role.java
│   └── RoleName.java
├── repository/
│   ├── UserRepository.java
│   └── RoleRepository.java
├── security/
│   ├── CustomUserDetailsService.java
│   ├── JwtAuthenticationEntryPoint.java
│   ├── JwtAuthenticationFilter.java
│   ├── JwtTokenProvider.java
│   └── UserPrincipal.java
└── service/
    ├── AuthService.java
    └── UserService.java
```

### 6.2 Dependencies (pom.xml)

```xml
```

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

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>

  <!-- JWT -->
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
  </dependency>

  <!-- Database -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>

  <!-- Development -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

## 6.3 Entity Classes

### User Entity:

```java
```

```
@Entity
@Table(name = "users", uniqueConstraints = {
    @UniqueConstraint(columnNames = "username"),
    @UniqueConstraint(columnNames = "email")
})
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    @Size(max = 20)
    private String username;

    @NotBlank
    @Size(max = 120)
    private String password;

    @NotBlank
    @Size(max = 50)
    @Email
    private String email;

    @Size(max = 100)
    private String firstName;

    @Size(max = 100)
    private String lastName;

    private boolean enabled = true;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    // Constructors
    public User() {}
    public User(String username, String email, String password) {
        this.username = username;
        this.email = email;
        this.password = password;
    }

    // Getters and Setters
    // ... (standard getters and setters)
}
```

**Role Entity:**

```java
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @Column(length = 20)
    private RoleName name;

    // Constructors
    public Role() {}

    public Role(RoleName name) {
        this.name = name;
    }

    // Getters and Setters
    // ... (standard getters and setters)
}

// RoleName enum
public enum RoleName {
    ROLE_USER,
    ROLE_ADMIN,
    ROLE_MODERATOR
}
```

### 6.4 Repository Interfaces

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByUsername(String username);

    Optional<User> findByEmail(String email);

    Boolean existsByUsername(String username);

    Boolean existsByEmail(String email);

    @Query("SELECT u FROM User u WHERE u.username = ?1 OR u.email = ?1")
    Optional<User> findByUsernameOrEmail(String usernameOrEmail);
}

@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {

    Optional<Role> findByName(RoleName roleName);
}
```

### 6.5 DTOs

**Login Request:**

```java
public class LoginRequest {

    @NotBlank
    @Size(min = 3, max = 60)
    private String usernameOrEmail;

    @NotBlank
    @Size(min = 6, max = 20)
    private String password;

    // Constructors, getters, and setters
}
```

**JWT Response:**

```java
```

```
public class JwtResponse {

    private String accessToken;
    private String tokenType = "Bearer";
    private Long id;
    private String username;
    private String email;
    private List<String> roles;

    public JwtResponse(String accessToken, Long id, String username, String email, List<String> roles) {
        this.accessToken = accessToken;
        this.id = id;
        this.username = username;
        this.email = email;
        this.roles = roles;
    }

    // Getters and setters
}
```

### 6.6 Auth Controller

```
java
```

```java
@RestController
@RequestMapping("/api/auth")
@CrossOrigin(origins = "*", maxAge = 3600)
public class AuthController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private RoleRepository roleRepository;

    @Autowired
    private PasswordEncoder encoder;

    @Autowired
    private JwtTokenProvider jwtProvider;

    @PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {

        // Authenticate user
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                loginRequest.getUsernameOrEmail(),
                loginRequest.getPassword()
            )
        );

        // Set authentication in security context
        SecurityContextHolder.getContext().setAuthentication(authentication);

        // Generate JWT token
        String jwt = jwtProvider.generateToken(authentication);

        // Get user details
        UserPrincipal userDetails = (UserPrincipal) authentication.getPrincipal();
        List<String> roles = userDetails.getAuthorities().stream()
            .map(item -> item.getAuthority())
            .collect(Collectors.toList());

        return ResponseEntity.ok(new JwtResponse(
            jwt,
            userDetails.getId(),
        userDetails.getUsername(),
        userDetails.getEmail(),
        roles
    ));
}

@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {

    // Check if username exists
    if (userRepository.existsByUsername(signUpRequest.getUsername())) {
        return ResponseEntity.badRequest()
            .body(new MessageResponse("Error: Username is already taken!"));
    }

    // Check if email exists
    if (userRepository.existsByEmail(signUpRequest.getEmail())) {
        return ResponseEntity.badRequest()
            .body(new MessageResponse("Error: Email is already in use!"));
    }

    // Create new user
    User user = new User(signUpRequest.getUsername(),
                signUpRequest.getEmail(),
                encoder.encode(signUpRequest.getPassword()));

    // Set roles
    Set<String> strRoles = signUpRequest.getRoles();
    Set<Role> roles = new HashSet<>();

    if (strRoles == null) {
        Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
            .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
        roles.add(userRole);
    } else {
        strRoles.forEach(role -> {
            switch (role) {
                case "admin":
                    Role adminRole = roleRepository.findByName(RoleName.ROLE_ADMIN)
                        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                    roles.add(adminRole);
                    break;
                case "mod":
                    Role modRole = roleRepository.findByName(RoleName.ROLE_MODERATOR)
                        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                    roles.add(modRole);
                    break;
                default:
                    Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
                        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                    roles.add(userRole);
                }
            });
        }

        user.setRoles(roles);
        userRepository.save(user);

        return ResponseEntity.ok(new MessageResponse("User registered successfully!"));
    }

    @PostMapping("/signout")
    public ResponseEntity<?> logoutUser() {
        // Clear security context
        SecurityContextHolder.clearContext();
        return ResponseEntity.ok(new MessageResponse("User logged out successfully!"));
    }
}
```

## 6.7 Protected Controllers

### User Controller:

```
java
```

```java
@RestController
@RequestMapping("/api/user")
@PreAuthorize("hasRole('USER')")
public class UserController {

    @GetMapping("/profile")
    public ResponseEntity<?> getUserProfile() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();

        Map<String, Object> profile = new HashMap<>();
        profile.put("id", userPrincipal.getId());
        profile.put("username", userPrincipal.getUsername());
        profile.put("email", userPrincipal.getEmail());
        profile.put("authorities", userPrincipal.getAuthorities());

        return ResponseEntity.ok(profile);
    }

    @GetMapping("/dashboard")
    public ResponseEntity<?> getUserDashboard() {
        return ResponseEntity.ok(new MessageResponse("User dashboard accessed successfully!"));
    }
}
```

### Admin Controller:

```
java
```

```
@RestController
@RequestMapping("/api/admin")
@PreAuthorize("hasRole('ADMIN')")
public class AdminController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public ResponseEntity<?> getAllUsers() {
        List<User> users = userRepository.findAll();
        return ResponseEntity.ok(users);
    }

    @GetMapping("/dashboard")
    public ResponseEntity<?> getAdminDashboard() {
        return ResponseEntity.ok(new MessageResponse("Admin dashboard accessed successfully!"));
    }

    @DeleteMapping("/users/{id}")
    public ResponseEntity<?> deleteUser(@PathVariable Long id) {
        if (!userRepository.existsById(id)) {
            return ResponseEntity.notFound().build();
        }

        userRepository.deleteById(id);
        return ResponseEntity.ok(new MessageResponse("User deleted successfully!"));
    }
}
```

## 6.8 Application Properties

properties

```
# Application Configuration
spring.application.name=JWT Authentication App
server.port=8080

# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/jwt_auth_db?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true

# JWT Configuration
app.jwtSecret=jwtSecretKey
app.jwtExpirationInMs=86400000

# Logging
logging.level.com.example.jwtauth=DEBUG
logging.level.org.springframework.security=DEBUG

# CORS Configuration
app.cors.allowedOrigins=http://localhost:3000,http://localhost:4200
```

## 6.9 Data Initialization

java

```java
@Component
public class DataInitializer implements ApplicationRunner {

    @Autowired
    private RoleRepository roleRepository;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public void run(ApplicationArguments args) throws Exception {

        // Initialize roles
        initializeRoles();

        // Create admin user if not exists
        createAdminUser();
    }

    private void initializeRoles() {
        if (roleRepository.count() == 0) {
            roleRepository.save(new Role(RoleName.ROLE_USER));
            roleRepository.save(new Role(RoleName.ROLE_ADMIN));
            roleRepository.save(new Role(RoleName.ROLE_MODERATOR));
        }
    }

    private void createAdminUser() {
        if (!userRepository.existsByUsername("admin")) {
            User admin = new User("admin", "admin@example.com",
                          passwordEncoder.encode("admin123"));

            Role adminRole = roleRepository.findByName(RoleName.ROLE_ADMIN)
                .orElseThrow(() -> new RuntimeException("Admin role not found"));

            admin.setRoles(Set.of(adminRole));
            userRepository.save(admin);
        }
    }
}
```

## 7. Testing & Best Practices

### 7.1 Unit Testing Security Configuration

```java
@ExtendWith(MockitoExtension.class)
class SecurityConfigTest {

    @Mock
    private CustomUserDetailsService userDetailsService;

    @Mock
    private JwtAuthenticationEntryPoint jwtEntryPoint;

    @Mock
    private JwtAuthenticationFilter jwtFilter;

    @InjectMocks
    private SecurityConfig securityConfig;

    @Test
    void passwordEncoder_ShouldReturnBCryptPasswordEncoder() {
        PasswordEncoder encoder = securityConfig.passwordEncoder();
        assertThat(encoder).isInstanceOf(BCryptPasswordEncoder.class);
    }

    @Test
    void passwordEncoder_ShouldEncodePassword() {
        PasswordEncoder encoder = securityConfig.passwordEncoder();
        String encoded = encoder.encode("password");

        assertThat(encoded).isNotEqualTo("password");
        assertThat(encoder.matches("password", encoded)).isTrue();
    }
}
```

### 7.2 Integration Testing

```java
```

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@Transactional
class AuthControllerIntegrationTest {

  @Autowired
  private TestRestTemplate restTemplate;

  @Autowired
  private UserRepository userRepository;

  @Autowired
  private PasswordEncoder passwordEncoder;

  @Test
  void signin_WithValidCredentials_ShouldReturnJwtToken() {
    // Setup
    User user = new User("testuser", "test@example.com",
                passwordEncoder.encode("password123"));
    userRepository.save(user);

    LoginRequest request = new LoginRequest("testuser", "password123");

    // Execute
    ResponseEntity<JwtResponse> response = restTemplate.postForEntity(
      "/api/auth/signin", request, JwtResponse.class);

    // Verify
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getBody().getAccessToken()).isNotNull();
    assertThat(response.getBody().getUsername()).isEqualTo("testuser");
  }

  @Test
  void signin_WithInvalidCredentials_ShouldReturnUnauthorized() {
    LoginRequest request = new LoginRequest("invalid", "invalid");

    ResponseEntity<String> response = restTemplate.postForEntity(
      "/api/auth/signin", request, String.class);

    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);
  }
}
```

## 7.3 Security Testing with Mock MVC

```java
@WebMvcTest(UserController.class)
@Import(SecurityConfig.class)
class UserControllerSecurityTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private CustomUserDetailsService userDetailsService;

    @MockBean
    private JwtTokenProvider jwtTokenProvider;

    @Test
    @WithMockUser(roles = "USER")
    void getUserProfile_WithAuthenticatedUser_ShouldReturnProfile() throws Exception {
        mockMvc.perform(get("/api/user/profile"))
               .andExpect(status().isOk())
               .andExpected(jsonPath("$.username").exists());
    }

    @Test
    void getUserProfile_WithoutAuthentication_ShouldReturnUnauthorized() throws Exception {
        mockMvc.perform(get("/api/user/profile"))
               .andExpect(status().isUnauthorized());
    }

    @Test
    @WithMockUser(roles = "ADMIN")
    void getUserProfile_WithWrongRole_ShouldReturnForbidden() throws Exception {
        mockMvc.perform(get("/api/user/profile"))
               .andExpect(status().isForbidden());
    }
}
```

## 7.4 JWT Token Testing

```java
```

```
@ExtendWith(MockitoExtension.class)
class JwtTokenProviderTest {

    private JwtTokenProvider jwtTokenProvider;

    @BeforeEach
    void setUp() {
        jwtTokenProvider = new JwtTokenProvider();
        // Set test secret and expiration
        ReflectionTestUtils.setField(jwtTokenProvider, "jwtSecret", "testSecret");
        ReflectionTestUtils.setField(jwtTokenProvider, "jwtExpirationInMs", 3600000);
    }

    @Test
    void generateToken_ShouldCreateValidToken() {
        // Setup
        UserPrincipal userPrincipal = createTestUserPrincipal();
        Authentication authentication = mock(Authentication.class);
        when(authentication.getPrincipal()).thenReturn(userPrincipal);

        // Execute
        String token = jwtTokenProvider.generateToken(authentication);

        // Verify
        assertThat(token).isNotNull();
        assertThat(jwtTokenProvider.validateToken(token)).isTrue();
        assertThat(jwtTokenProvider.getUserIdFromToken(token)).isEqualTo(1L);
    }

    @Test
    void validateToken_WithExpiredToken_ShouldReturnFalse() {
        // Create token with past expiration
        String expiredToken = Jwts.builder()
            .setSubject("1")
            .setExpiration(new Date(System.currentTimeMillis() - 1000))
            .signWith(SignatureAlgorithm.HS512, "testSecret")
            .compact();

        assertThat(jwtTokenProvider.validateToken(expiredToken)).isFalse();
    }
}
```

## 7.5 Best Practices

**Security Best Practices:**

1. **JWT Secret Management**:

```java
// Use environment variables or external configuration
@Value("${app.jwtSecret}")
private String jwtSecret;

// Use strong, random secrets
private String generateSecureSecret() {
    SecureRandom random = new SecureRandom();
    byte[] bytes = new byte[64];
    random.nextBytes(bytes);
    return Base64.getEncoder().encodeToString(bytes);
}
```

2. **Token Expiration**:

```java
// Short-lived access tokens
private static final int ACCESS_TOKEN_EXPIRATION = 900; // 15 minutes

// Longer-lived refresh tokens
private static final int REFRESH_TOKEN_EXPIRATION = 86400; // 24 hours
```

3. **Password Security**:

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12); // Higher strength
}

// Validate password strength
@Component
public class PasswordValidator {
    private static final String PASSWORD_PATTERN =
        "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[!@#&()–[{}]:;',?/*~$^+=<>]).{8,}$";

    public boolean isValid(String password) {
        return password.matches(PASSWORD_PATTERN);
    }
}
```

4. **CORS Configuration**:

```java
```

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowedOriginPatterns("https://yourdomain.com")
        .allowedMethods("GET", "POST", "PUT", "DELETE")
        .allowedHeaders("*")
        .allowCredentials(true)
        .maxAge(3600);
  }
}
```

5. **Rate Limiting**:

```
java
```

```java
@Component
public class RateLimitingFilter implements Filter {

    private final Map<String, List<Long>> requestCounts = new ConcurrentHashMap<>();
    private final int MAX_REQUESTS_PER_MINUTE = 60;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                       FilterChain chain) throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String clientIp = getClientIp(httpRequest);

        if (isRateLimited(clientIp)) {
            HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.setStatus(429); // Too Many Requests
            return;
        }

        chain.doFilter(request, response);
    }

    private boolean isRateLimited(String clientIp) {
        long currentTime = System.currentTimeMillis();
        long oneMinuteAgo = currentTime - 60000;

        requestCounts.computeIfAbsent(clientIp, k -> new ArrayList<>())
                .removeIf(time -> time < oneMinuteAgo);

        List<Long> requests = requestCounts.get(clientIp);
        requests.add(currentTime);

        return requests.size() > MAX_REQUESTS_PER_MINUTE;
    }
}
```

## 8. Advanced Topics

### 8.1 Refresh Token Implementation

java

```java
@Entity
public class RefreshToken {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    @JoinColumn(name = "user_id", referencedColumnName = "id")
    private User user;

    @Column(nullable = false, unique = true)
    private String token;

    @Column(nullable = false)
    private Instant expiryDate;

    // Constructors, getters, setters
}

@Service
public class RefreshTokenService {

    @Autowired
    private RefreshTokenRepository refreshTokenRepository;

    @Autowired
    private UserRepository userRepository;

    public RefreshToken createRefreshToken(Long userId) {
        RefreshToken refreshToken = new RefreshToken();
        refreshToken.setUser(userRepository.findById(userId).get());
        refreshToken.setExpiryDate(Instant.now().plusMillis(86400000)); // 24 hours
        refreshToken.setToken(UUID.randomUUID().toString());

        return refreshTokenRepository.save(refreshToken);
    }

    public Optional<RefreshToken> findByToken(String token) {
        return refreshTokenRepository.findByToken(token);
    }

    public RefreshToken verifyExpiration(RefreshToken token) {
        if (token.getExpiryDate().compareTo(Instant.now()) < 0) {
            refreshTokenRepository.delete(token);
            throw new RuntimeException("Refresh token was expired. Please make a new signin request");
        }
        return token;
    }
}
```

## 8.2 Method-Level Security

```java
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping
    @PreAuthorize("hasRole('USER')")
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @PostMapping
    @PreAuthorize("hasRole('ADMIN')")
    public Product createProduct(@RequestBody Product product) {
        return productService.save(product);
    }

    @PutMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN') or @productService.isOwner(#id, authentication.name)")
    public Product updateProduct(@PathVariable Long id, @RequestBody Product product) {
        return productService.update(id, product);
    }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public void deleteProduct(@PathVariable Long id) {
        productService.delete(id);
    }
}
```

## 8.3 Custom Security Expressions

```java
```

```
@Component("productSecurity")
public class ProductSecurityExpression {

    @Autowired
    private ProductService productService;

    public boolean isOwner(Long productId, String username) {
        Product product = productService.findById(productId);
        return product != null && product.getOwner().getUsername().equals(username);
    }

    public boolean canAccess(Long productId, String username) {
        Product product = productService.findById(productId);
        return product != null &&
               (product.isPublic() || product.getOwner().getUsername().equals(username));
    }
}

// Usage in controller
@PreAuthorize("@productSecurity.canAccess(#id, authentication.name)")
@GetMapping("/{id}")
public Product getProduct(@PathVariable Long id) {
    return productService.findById(id);
}
```

### 8.4 OAuth2 Integration

```java
```

```java
@Configuration
@EnableWebSecurity
public class OAuth2SecurityConfig {

    @Bean
    public SecurityFilterChain oauth2FilterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/login**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth2 -> oauth2
                .loginPage("/login")
                .successHandler(oauth2AuthenticationSuccessHandler())
                .failureHandler(oauth2AuthenticationFailureHandler())
            )
            .build();
    }

    @Bean
    public OAuth2AuthenticationSuccessHandler oauth2AuthenticationSuccessHandler() {
        return new OAuth2AuthenticationSuccessHandler() {
            @Override
            public void onAuthenticationSuccess(HttpServletRequest request,
                                                HttpServletResponse response,
                                                Authentication authentication) throws IOException {

                // Generate JWT token for OAuth2 user
                String jwt = jwtTokenProvider.generateTokenFromOAuth2User(authentication);

                // Redirect to frontend with token
                response.sendRedirect("http://localhost:3000/oauth2/redirect?token=" + jwt);
            }
        };
    }
}
```

## 8.5 Security Auditing

```
java
```

```java
@Entity
@EntityListeners(AuditingEntityListener.class)
public class SecurityAuditLog {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String action;
    private String resource;
    private String ipAddress;
    private String userAgent;

    @CreatedDate
    private LocalDateTime timestamp;

    private boolean success;
    private String failureReason;

    // Constructors, getters, setters
}

@Component
public class SecurityAuditListener {

    @Autowired
    private SecurityAuditLogRepository auditRepository;

    @EventListener
    public void handleAuthenticationSuccess(AuthenticationSuccessEvent event) {
        UserDetails user = (UserDetails) event.getAuthentication().getPrincipal();

        SecurityAuditLog log = new SecurityAuditLog();
        log.setUsername(user.getUsername());
        log.setAction("LOGIN");
        log.setSuccess(true);

        auditRepository.save(log);
    }

    @EventListener
    public void handleAuthenticationFailure(AbstractAuthenticationFailureEvent event) {
        SecurityAuditLog log = new SecurityAuditLog();
        log.setUsername(event.getAuthentication().getName());
        log.setAction("LOGIN_FAILED");
        log.setSuccess(false);
        log.setFailureReason(event.getException().getMessage());

        auditRepository.save(log);
    }
}
```

---

## Summary

This comprehensive Spring Security learning plan covers:

### ✅ Foundation Concepts

- Servlet filters and filter chain architecture
- Spring Security filter chain and components
- Authentication vs Authorization principles

### ✅ Core Components

- **SecurityConfig**: Central configuration with detailed explanations
- **UserDetailsService**: Custom implementation with role management
- **UserDetails**: Custom user principal implementation
- **AuthenticationManager & Providers**: Authentication flow coordination

### ✅ JWT Implementation

- Complete JWT authentication flow
- Token generation, validation, and extraction
- Custom JWT filters and entry points
- Security context management

### ✅ Complete Spring Boot Project

- Full project structure with all components
- Entity relationships and repository layers
- RESTful API endpoints with proper security
- Database integration and data initialization

### ✅ Testing & Best Practices

- Unit and integration testing strategies
- Security testing with MockMVC

- Production-ready security practices
- Rate limiting and CORS configuration

### ✅ Advanced Topics

- Refresh token implementation
- Method-level security annotations
- Custom security expressions
- OAuth2 integration basics
- Security auditing and logging

### Next Steps for Learning:

1. **Week 1-2**: Study filter concepts and Spring Security architecture
2. **Week 3-4**: Implement core components and understand their roles
3. **Week 5-6**: Build the complete JWT authentication system
4. **Week 7-8**: Add testing, implement advanced features, and optimize for production

This plan provides a solid foundation for mastering Spring Security with hands-on JWT implementation experience.