Security + JWT

Installation

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
        </dependency>
```

Configuration

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
class SecurityConfig {

    @Autowired
    lateinit var rsaKeyConfigProperties: RsaKeyConfigProperties

    @Autowired
    lateinit var userDetailsService: JpaUserDetailsService

    @Value("\${cors.allowedOrigins}")
    private lateinit var allowedOrigins: String


    @Bean
    fun authManager(): AuthenticationManager {
        val authProvider = DaoAuthenticationProvider()
        authProvider.setUserDetailsService(userDetailsService)
        authProvider.setPasswordEncoder(passwordEncoder())
        return ProviderManager(authProvider)
    }


    @Bean
    @Throws(Exception::class)
    fun filterChain(http: HttpSecurity, introspector: HandlerMappingIntrospector?):
SecurityFilterChain {
        return http
            .csrf { csrf: CsrfConfigurer<HttpSecurity> ->
                csrf.disable()
```

```
            }
            .cors(Customizer.withDefaults())
            .authorizeHttpRequests { auth ->
                auth.requestMatchers("/error/**").permitAll()
                auth.requestMatchers("/api/auth/**").permitAll()
                auth.requestMatchers("/h2-console/**").permitAll()
                auth.requestMatchers("/swagger-ui/**").permitAll()
                auth.requestMatchers("/api-docs/**").permitAll()
                auth.requestMatchers("/uploads/**").permitAll()
                auth.requestMatchers("/images/**").permitAll()


                auth.requestMatchers("/api/**").authenticated()

                auth.anyRequest().authenticated()
            }.headers { headers ->
                headers.frameOptions { it.disable() }
            }
            .sessionManagement { s: SessionManagementConfigurer<HttpSecurity?> ->
                s.sessionCreationPolicy(
                    SessionCreationPolicy.STATELESS
                )
            }
            .oauth2ResourceServer { oauth2:
OAuth2ResourceServerConfigurer<HttpSecurity?> ->
                oauth2.jwt { jwt ->
                    jwt.decoder(
                        jwtDecoder()
                    )
                }
            }
            .userDetailsService(userDetailsService)
            .httpBasic(Customizer.withDefaults())
            .build()
    }

    @Bean
    fun jwtDecoder(): JwtDecoder {
        return
NimbusJwtDecoder.withPublicKey(rsaKeyConfigProperties.publicKey).build()
    }

    @Bean
    fun jwtEncoder(): JwtEncoder {
        val jwk: JWK =
RSAKey.Builder(rsaKeyConfigProperties.publicKey).privateKey(rsaKeyConfigProperties.
privateKey).build()

        val jwks: JWKSource<SecurityContext> = ImmutableJWKSet(JWKSet(jwk))
        return NimbusJwtEncoder(jwks)
    }

    @Bean
    fun passwordEncoder(): PasswordEncoder {
        return BCryptPasswordEncoder()
    }
```

```
    @Bean
    fun corsConfigurationSource(): CorsConfigurationSource {
        val source = UrlBasedCorsConfigurationSource()
        val config = CorsConfiguration()
        config.allowedOrigins = allowedOrigins.split(",")
        config.allowedMethods = listOf("GET", "POST", "PUT", "DELETE", "OPTIONS",
"PATCH", "HEAD")
        config.allowedHeaders = listOf("*")
        config.allowCredentials = true
        source.registerCorsConfiguration("/api/**", config)
        return source
    }


    companion object {

        private val log: Logger =
LoggerFactory.getLogger(SecurityConfig::class.java)
    }
}
```

RSA config

```
@ConfigurationProperties(prefix = "rsa")
@JvmRecord
data class RsaKeyConfigProperties(val publicKey: RSAPublicKey, val privateKey:
RSAPrivateKey)
```

AuthUser

```
class AuthUser(val user: User) : UserDetails {

    override fun getAuthorities(): MutableCollection<out GrantedAuthority> {
        return mutableListOf(SimpleGrantedAuthority("ROLE_${user.role.name}"))
    }

    override fun getPassword(): String? = user.password

    override fun getUsername(): String? = user.username

    override fun isAccountNonExpired(): Boolean = true

    override fun isAccountNonLocked(): Boolean = true

    override fun isCredentialsNonExpired(): Boolean = true

    override fun isEnabled(): Boolean = user.enabled
}
```

Services

```
@Service
class JpaUserDetailsService(
    val userRepository: UserRepository,
    val logEventRepository: LogEventRepository,
) : UserDetailsService {


    @Throws(UsernameNotFoundException::class)
    @Transactional
    override fun loadUserByUsername(usernameOrEmail: String): UserDetails {
        val user: User = userRepository
            .findByUsernameOrEmail(usernameOrEmail, usernameOrEmail)
            .orElseThrow { UsernameNotFoundException("User name or email not found:
$usernameOrEmail") }
        return AuthUser(user)
    }
}
```

```
@Service
class AuthService {

    @Autowired
    lateinit var jwtEncoder: JwtEncoder

    @Autowired
    lateinit var JwtDecoder: JwtDecoder

    @Autowired
    lateinit var passwordEncoder: PasswordEncoder

    @Autowired
    lateinit var userRepository: UserRepository

    fun generateToken(authentication: Authentication): String {
        val now = Instant.now()

        val scope: String = authentication.getAuthorities()
            .stream()
            .map { obj: GrantedAuthority -> obj.authority }
            .collect(Collectors.joining(" "))
        val user = (authentication.principal as AuthUser).user
        val claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plus(10, ChronoUnit.HOURS))
            .subject(authentication.getName())
            .claim("scope", scope)
            .claim("sub", user.id)
            .claim("role", user.role.name)
```

```
            .claim("username", user.username)
            .build()

        return jwtEncoder.encode(JwtEncoderParameters.from(claims)).tokenValue
    }

    fun getActiveUser(token: String): User {
        val claims = JwtDecoder.decode(token).claims
        val userId = claims["sub"] as UUID
        return userRepository.findById(userId).orElseThrow { RuntimeException("User
not found") }
    }

    fun hashPassword(password: String): String {
        if (!isBCryptHash(password)) {
            return passwordEncoder.encode(password)
        }
        return password
    }

    fun isBCryptHash(password: String): Boolean {
        return password.matches(Regex("^\\$2[aby]\\$\\d{2}\\$[./A-Za-z0-9]{53}$"))
    }

}
```