CHAPTER 5

SPRING SECURITY – JWT
(JSON Web Token)

Chapter 5: SPRING SECURITY - JWT

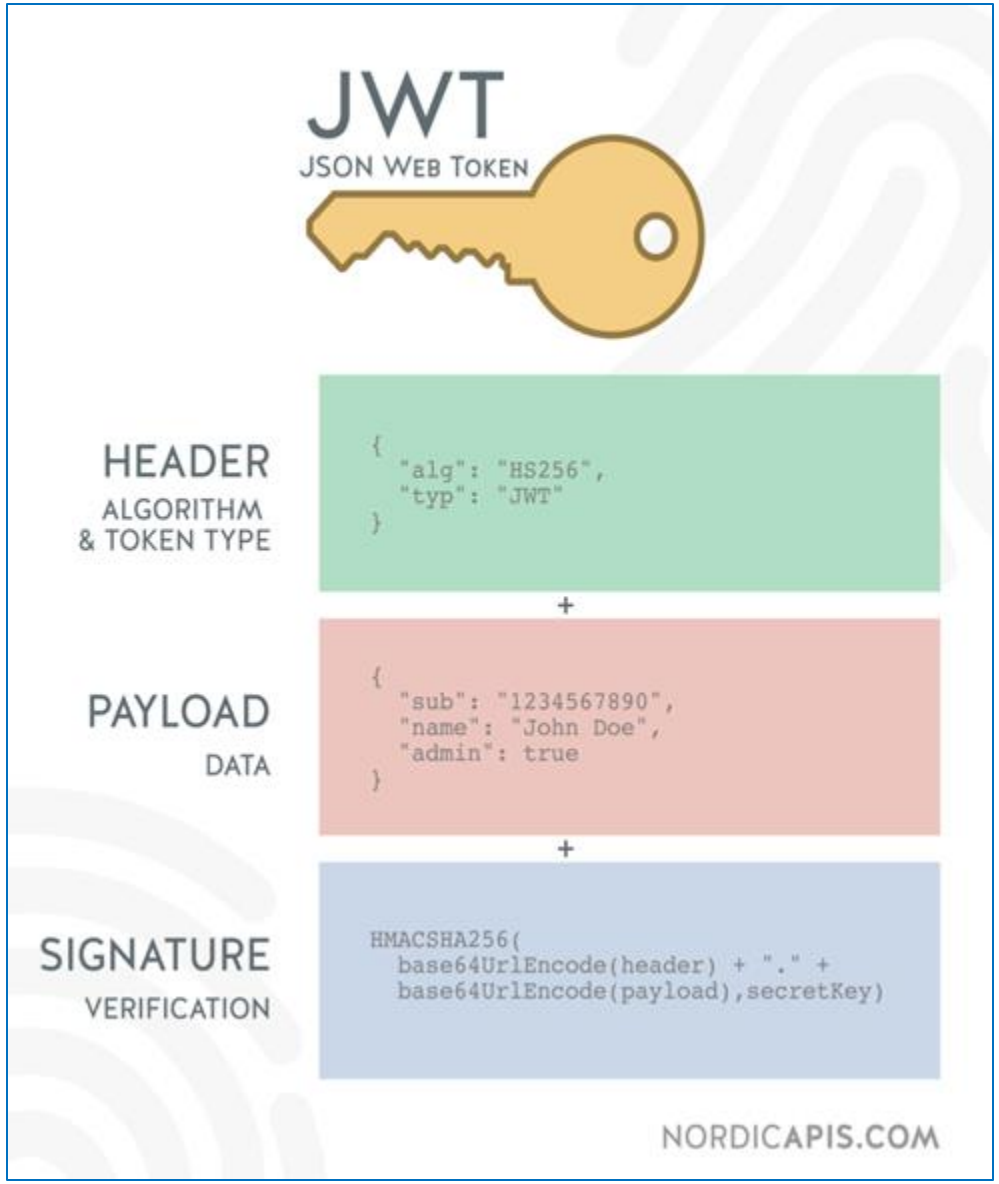
OUTLINE

1. What is JWT?
2. Practices

What is JWT?

- JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties: client and server. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

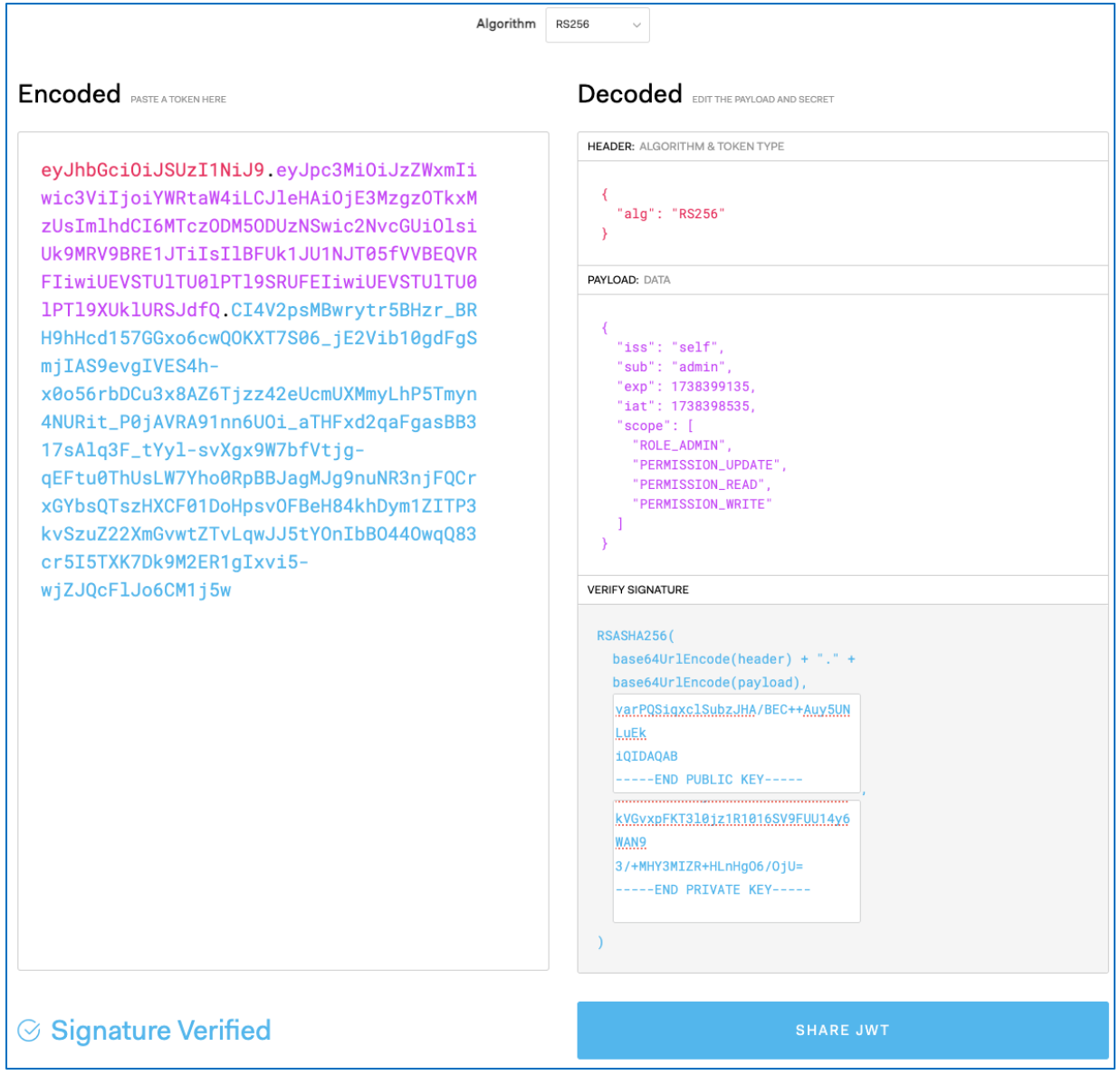
Structure of JWT



header.payload.signature

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJzZWxmIi
wic3ViIjoiYWRtaW4iLCJleHAiOjE3MzgzOTkxM
zUsImlhdCI6MTczODM5ODUzNSwic2NvcGUiOlsi
Uk9MRV9BRE1JTiIsIlBFUk1JU1NJT05fVVBEQVR
FIiwiUEVSTUlTU0lPTl9SRUFEIiwiUEVSTUlTU0
lPTl9XUklURSJdfQ.CI4V2psMBwrytr5BHzr_BR
H9hHcd157GGxo6cwQOKXT7S06_jE2Vib10gdFgS
mjIAS9evgIVES4h-
x0o56rbDCu3x8AZ6Tjzz42eUcmUXMmyLhP5Tmyn
4NURit_P0jAVRA91nn6UOi_aTHFxd2qaFgasBB3
17sAlq3F_tYyl-svXgx9W7bfVtjg-
qEFtu0ThUsLW7Yho0RpBBJagMJg9nuNR3njFQCr
xGYbsQTszHXCF01DoHpsvOFBeH84khDym1ZITP3
kvSzuZ22XmGvwtZTvLqwJJ5tYOnIbBO44OwqQ83
cr5I5TXK7Dk9M2ER1gIxvi5-
wjZJQcFlJo6CM1j5w
```

Structure of JWT (cont.) - https://jwt.io/



JWT - header

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

- The header typically tells us the type of the token, which is JWT, and the algorithm that is used for signing the token, like **HMAC SHA256 or RSA** (**R**onald Rivest, Adi **S**hamir and Leonard **A**dleman).
- This is a simple JSON object that states the JWT is using the ‘RS256' algorithm for encryption. It's encoded in Base64Url format to make it URL-safe.

JWT - payload

PAYLOAD: DATA

```
{
  "iss": "self",
  "sub": "admin",
  "exp": 1738399135,
  "iat": 1738398535,
  "scope": [
    "ROLE_ADMIN",
    "PERMISSION_UPDATE",
    "PERMISSION_READ",
    "PERMISSION_WRITE"
  ]
}
```

- The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data.
- This JSON object contains information about the user, other metadata and has administrative rights.
- Claims type:
  - Registered claims
  - Public claims
  - Private claims

JWT – Registered Claims
https://www.iana.org/assignments/jwt/jwt.xhtml#claims

Registered claims are a set of predetermined claims that aren't required but are recommended to be used to deliver useful and interoperable claims.

- iss (issuer): Issuer of the JWT.
- sub (subject): Subject of the JWT (the user).
- aud (audience): The JWT intended recipient or audience.
- exp (expiration time): The time the JWT expires.
- nbf (not before policy): Identifies the time before which JWT can not be accepted into processing.
- iat (issued at time): Identifies the time at which the JWT was issued. This can be used to establish the age of the JWT or the exact time the token was generated.
- jti (JWT ID): Unique identifier; this can be used to prevent the JWT from being used more than once.

JWT – Public Claims

https://www.iana.org/assignments/jwt/jwt.xhtml#claims

Public claims are not required, they are recommended to provide useful and interoperable claims and can be used by various parties if they agree on their meaning. Public claims can be found in the IANA JSON Web Token Claims Registry.

Common public names used by developers include:

- name, given_name, family_name, middle_name: the name of the user
- email: the email address of the particular user
- locale: the user's preferred language

JWT – Private Claims

- Private claims are additional bits of information relating to your specific application or organization and much like public claims, they are not standardized by the JWT specification.
- For example, a public claim may contain a user's name and email, but private claims can provide more information about a user such as their department name, role in an organization, and permissions.

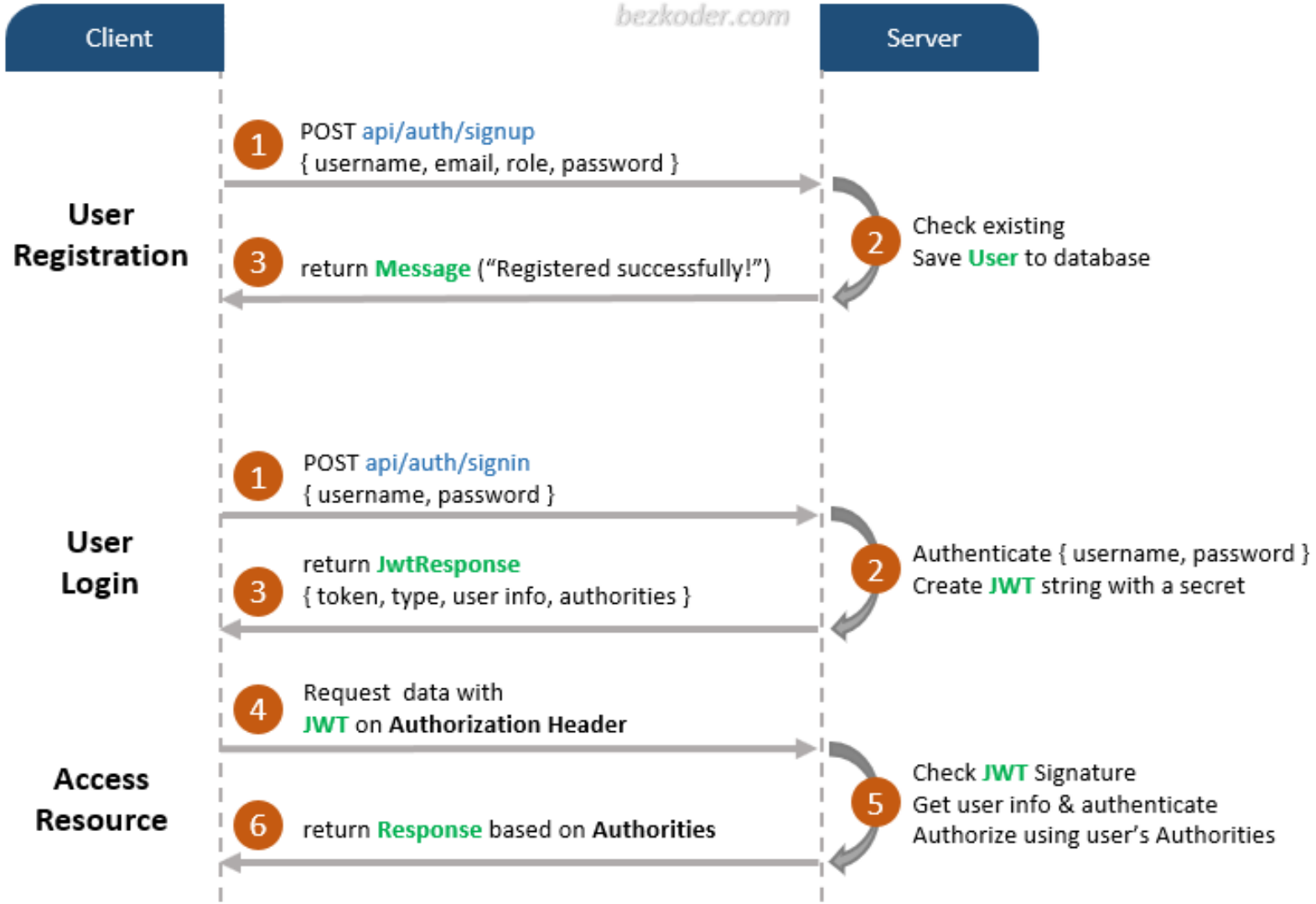
JWT – Signature

- Used to validate that the token is trustworthy and has not been tampered with. When you use a JWT, you **must** check its signature before storing and using it.

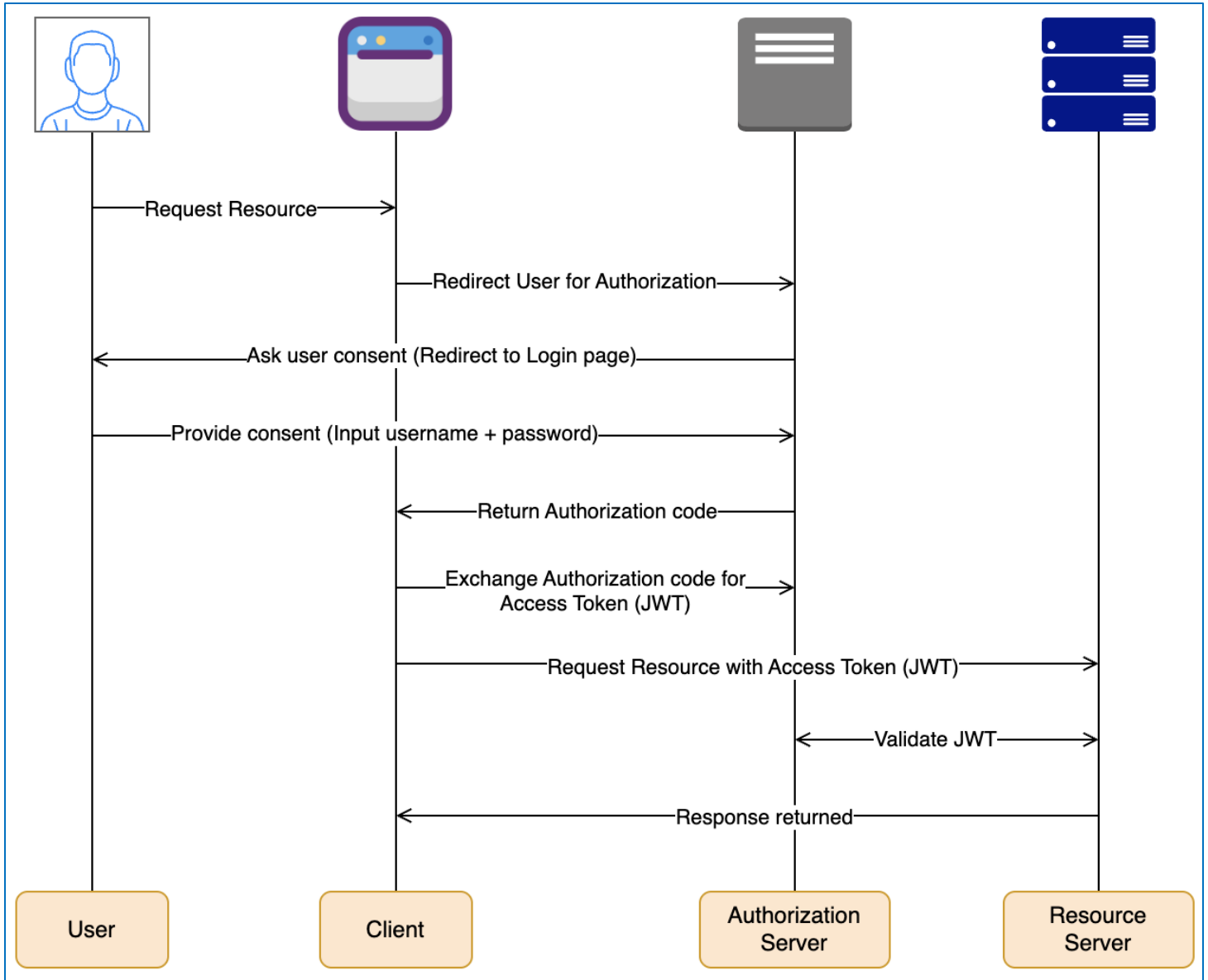
Authentication JWT vs Session

- JWT: stateless
- Session: state

How JWT work?


bezkoder.com
Client
Server
User Registration
1
POST api/auth/signup
{ username, email, role, password }
2
Check existing
Save User to database
3
return Message ("Registered successfully!")
User Login
1
POST api/auth/signin
{ username, password }
2
Authenticate { username, password }
Create JWT string with a secret
3
return JwtResponse
{ token, type, user info, authorities }
Access Resource
4
Request data with
JWT on Authorization Header
5
Check JWT Signature
Get user info & authenticate
Authorize using user's Authorities
6
return Response based on Authorities

JWT with OAuth (Spring Security)



OAuth2.0 Roles

- Ref: https://auth0.com/docs/authenticate/protocols/oauth
- An OAuth 2.0 flow has the following roles:
  - **Resource Owner**: Entity that can grant access to a protected resource. Typically, this is the end-user.
  - **Resource Server**: Server hosting the protected resources. This is the API you want to access.
  - **Client**: Application requesting access to a protected resource on behalf of the Resource Owner.
  - **Authorization Server**: Server that authenticates the Resource Owner and issues access tokens after getting proper authorization.

Configure Persistent RSA Keys for Spring Authorization & Resource Server

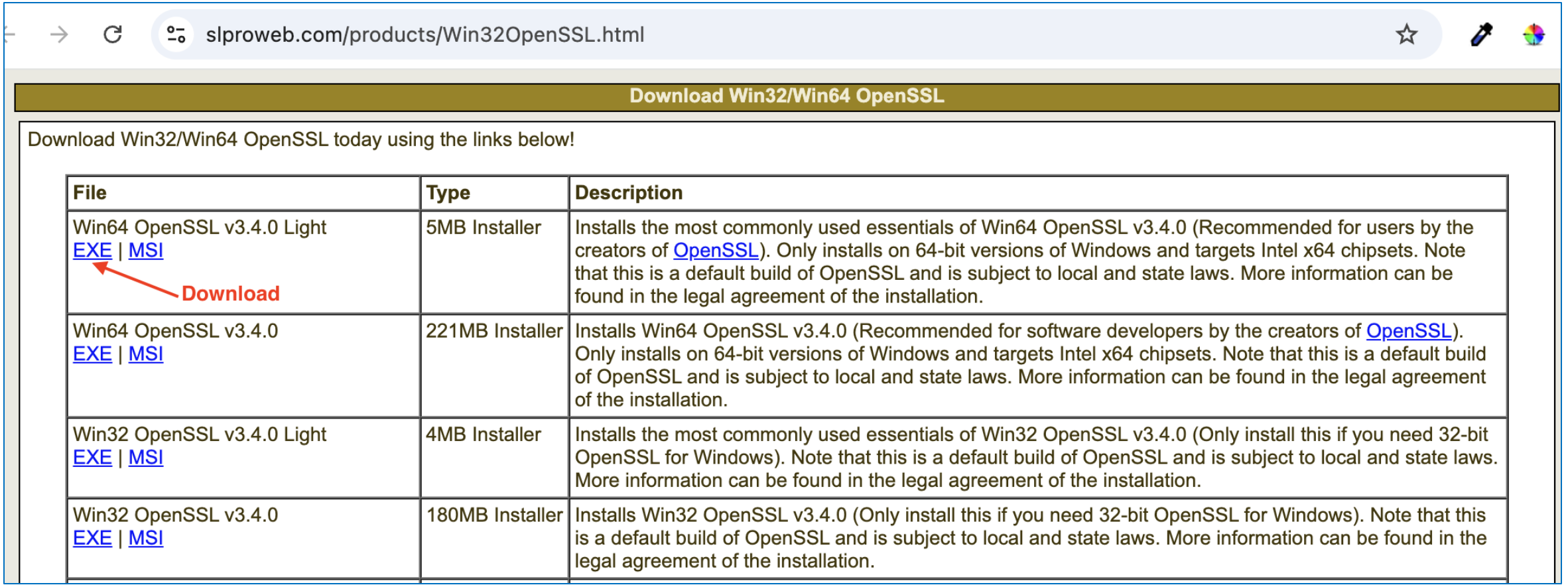
- **Spring Authorization** server uses a **JwtEncoder** to sign access tokens issued to clients
  - The **encoder** uses the **private key** in the RSA keypair.
- **Spring Resource** server uses a **JwtDecoder** to verify access tokens sent from the clients
  - The **decoder** uses the **public key** in the RSA keypair.
- By default, Spring Security generates an in-memory RSA keypair on each server restart
  - Clients won't be able to use issued access tokens if the server restarted, even the tokens have not expired.
- It's recommended to use persistent RSA keys instead of in-memory ones.

Configure Persistent RSA Keys for Spring Authorization & Resource Server

- Steps:
  - User OpenSSL tool to generate an RSA keypair which is stored under **src/resources** directory
  - Configure path to the generated key files in the **application.properties** file
  - Create a Java record class that represents the RSA keys
  - Use Spring's **ConfigurationProperties** annotation to load content of private and public key files to Java objects of type **RSAPrivateKey** and **RSAPublicKey**
  - Declare two beans of the type **JwtEncoder** and **JwtDecoder** that use the configured public and private keys in the Security configuration class

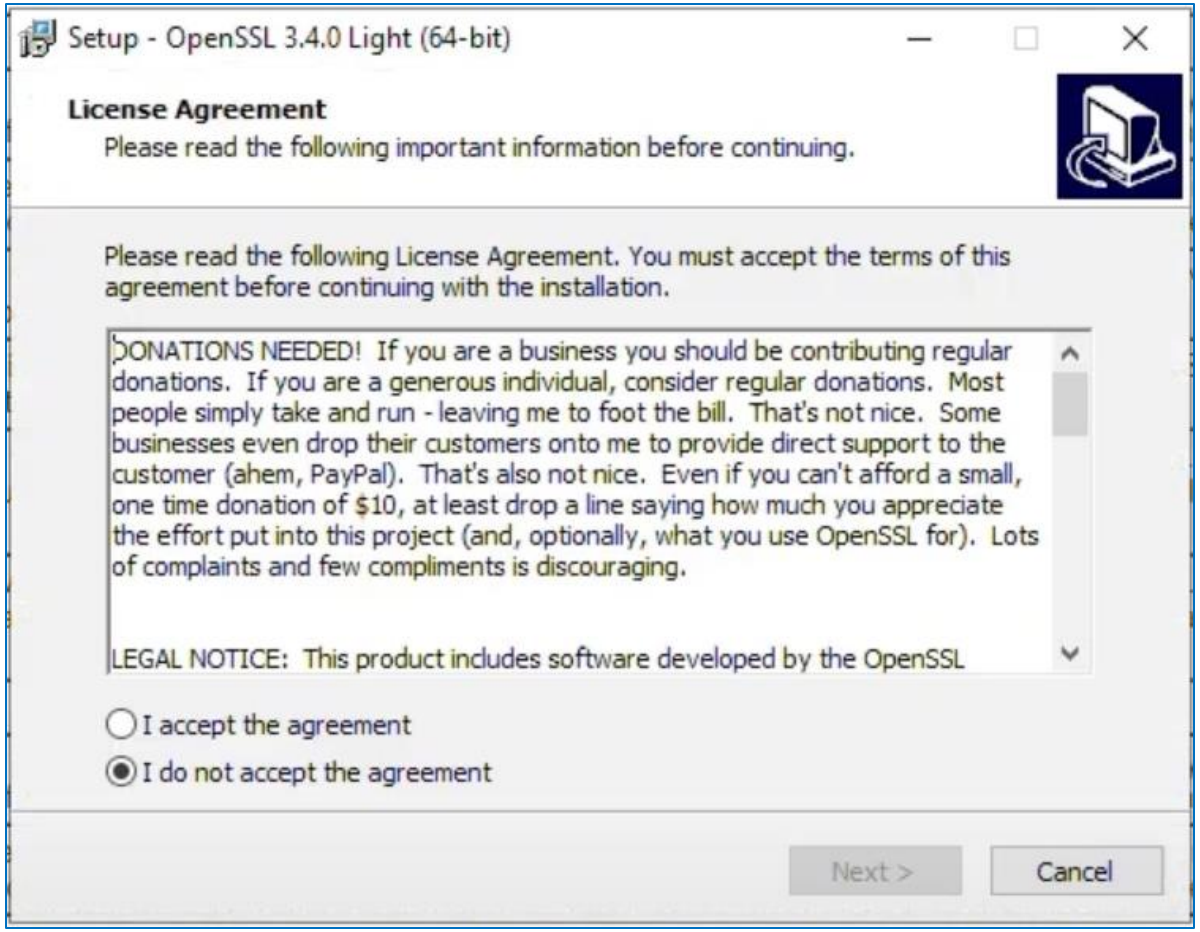
Install OpenSSL on the Windows

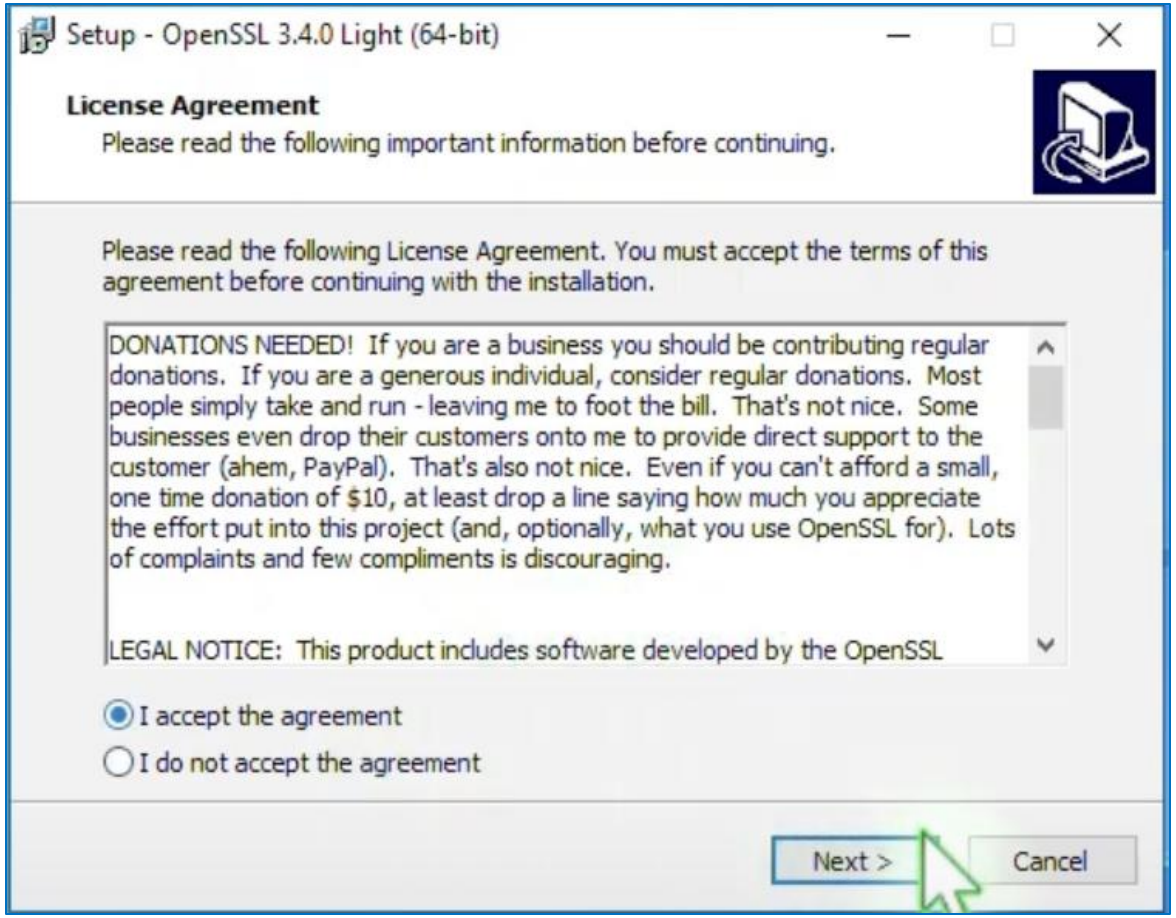
- ***Step 1:*** download OpenSSL Installer
  - Link: https://slproweb.com/products/Win32OpenSSL.html



Install OpenSSL on the Windows (cont.)

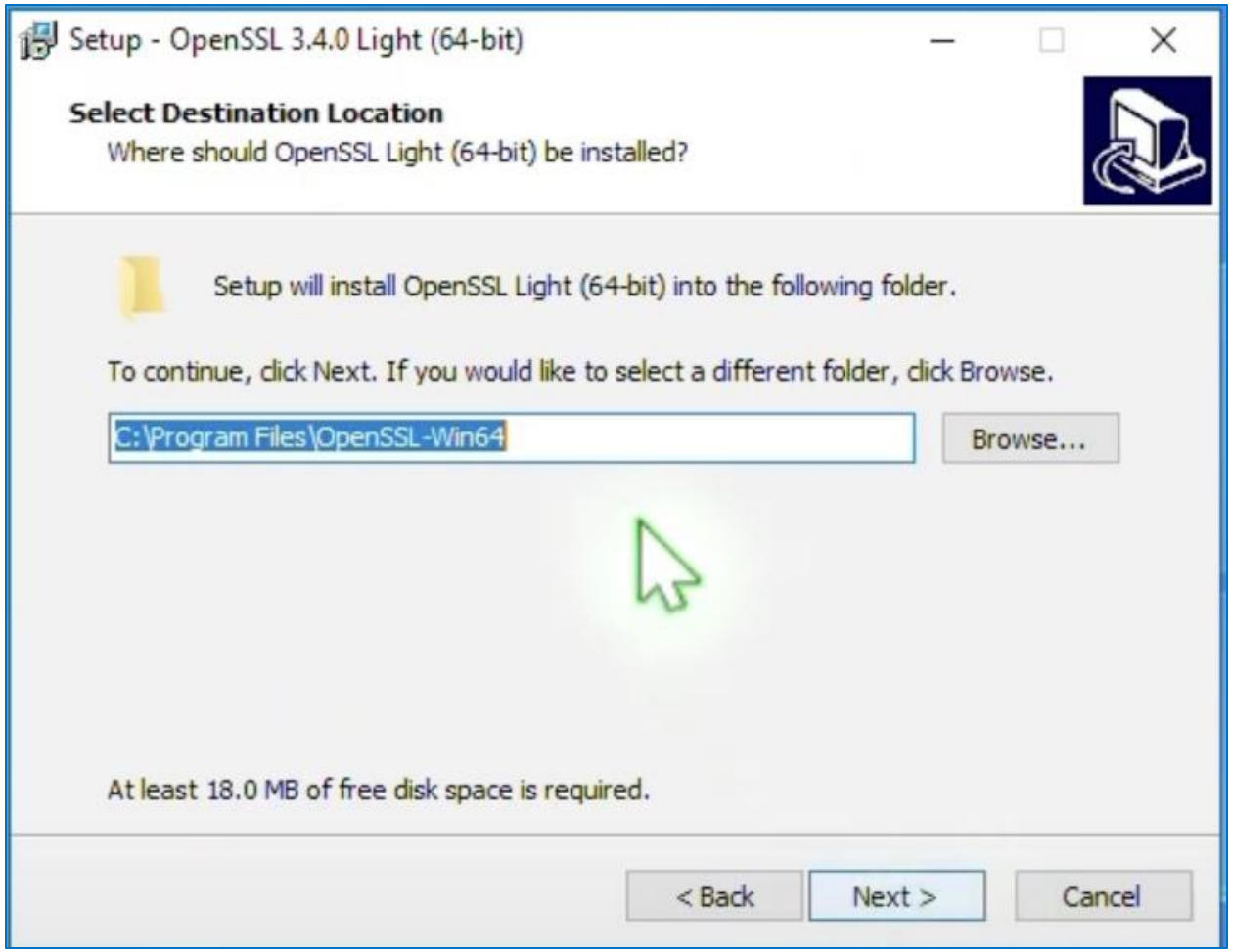
- ***Step 2:*** run the OpenSSL Installer
  - Run the OpenSSL installer (.exe file) by double-clicking on it

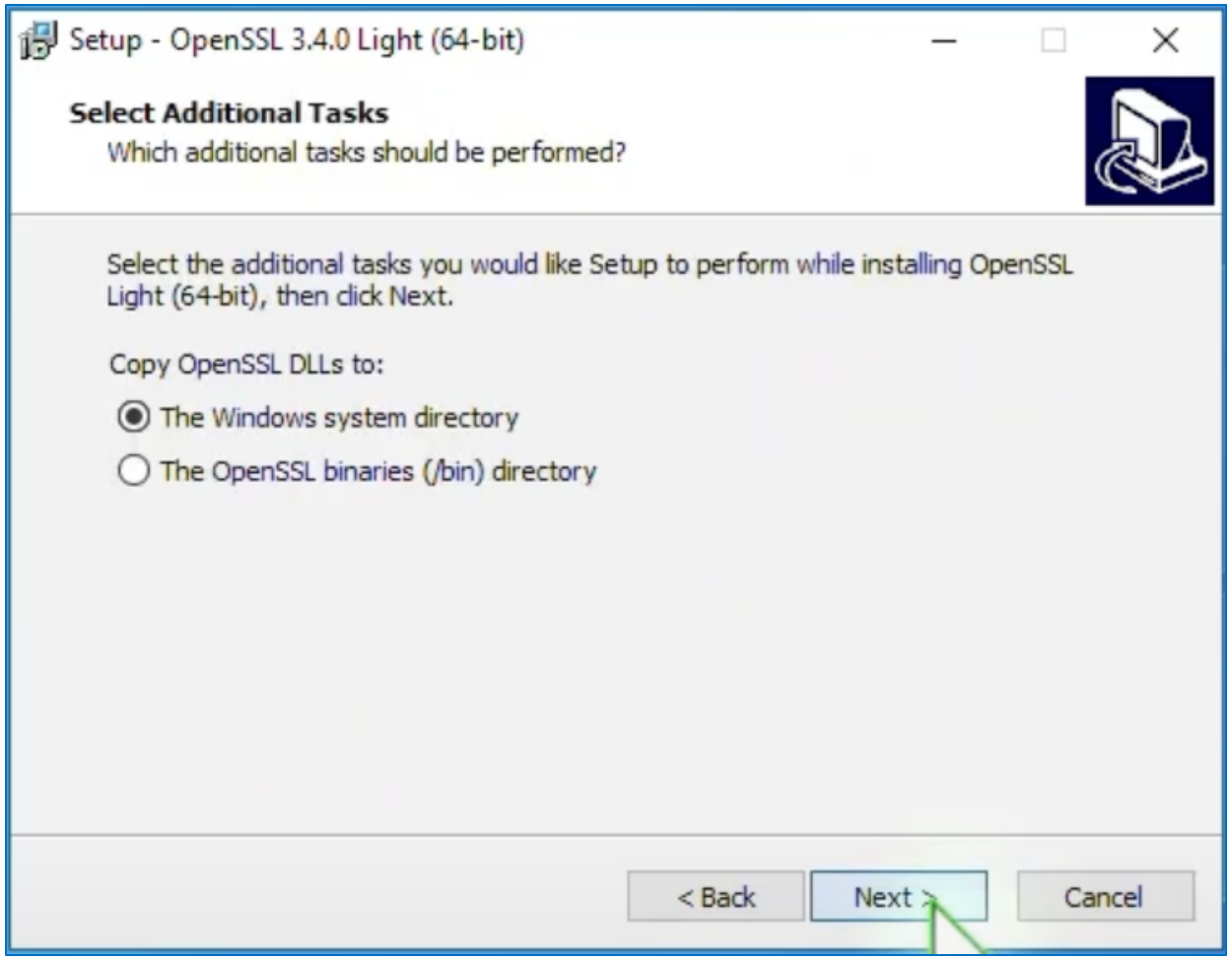




Install OpenSSL on the Windows (cont.)

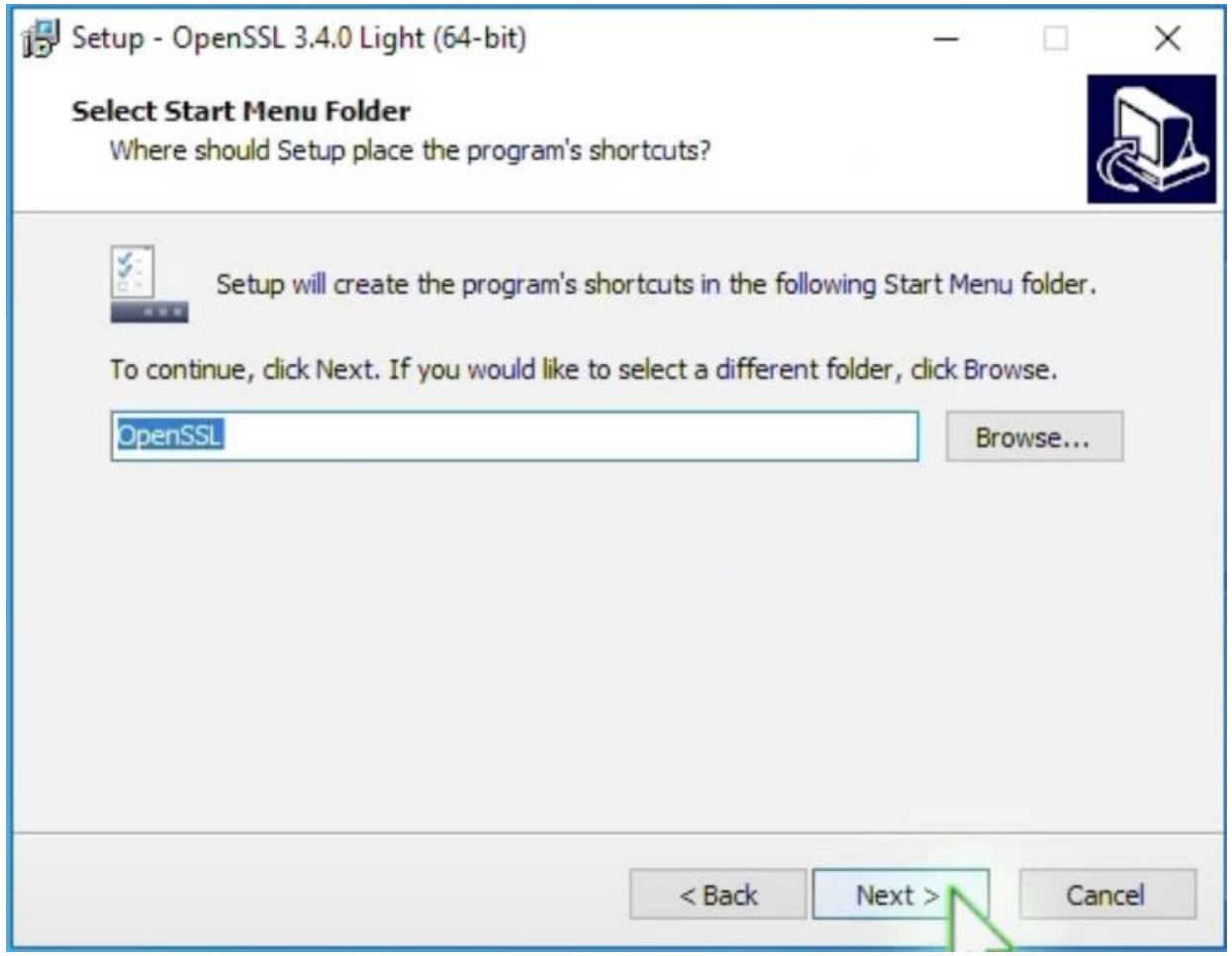
- ***Step 2:*** run the OpenSSL Installer (cont.)

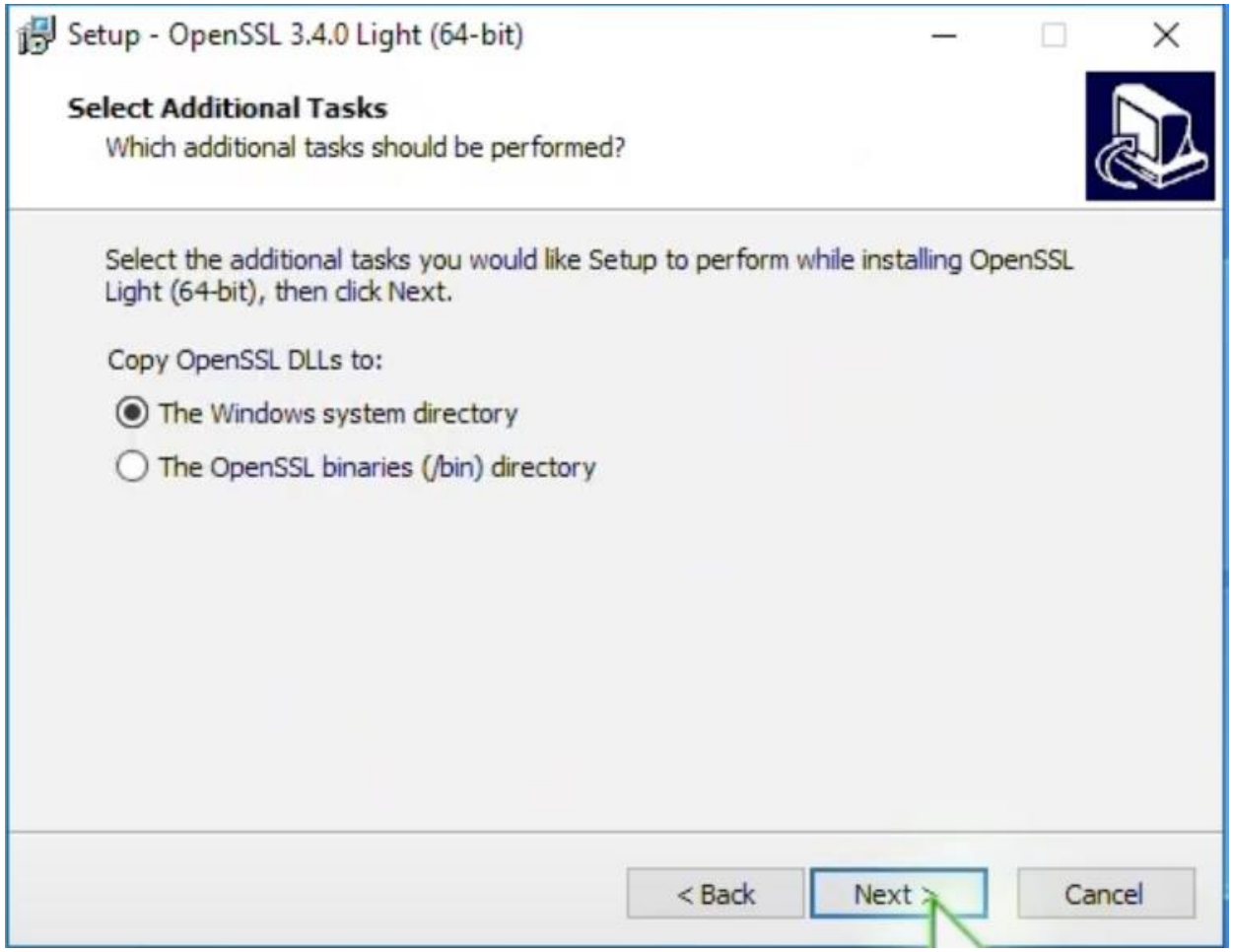




Install OpenSSL on the Windows (cont.)

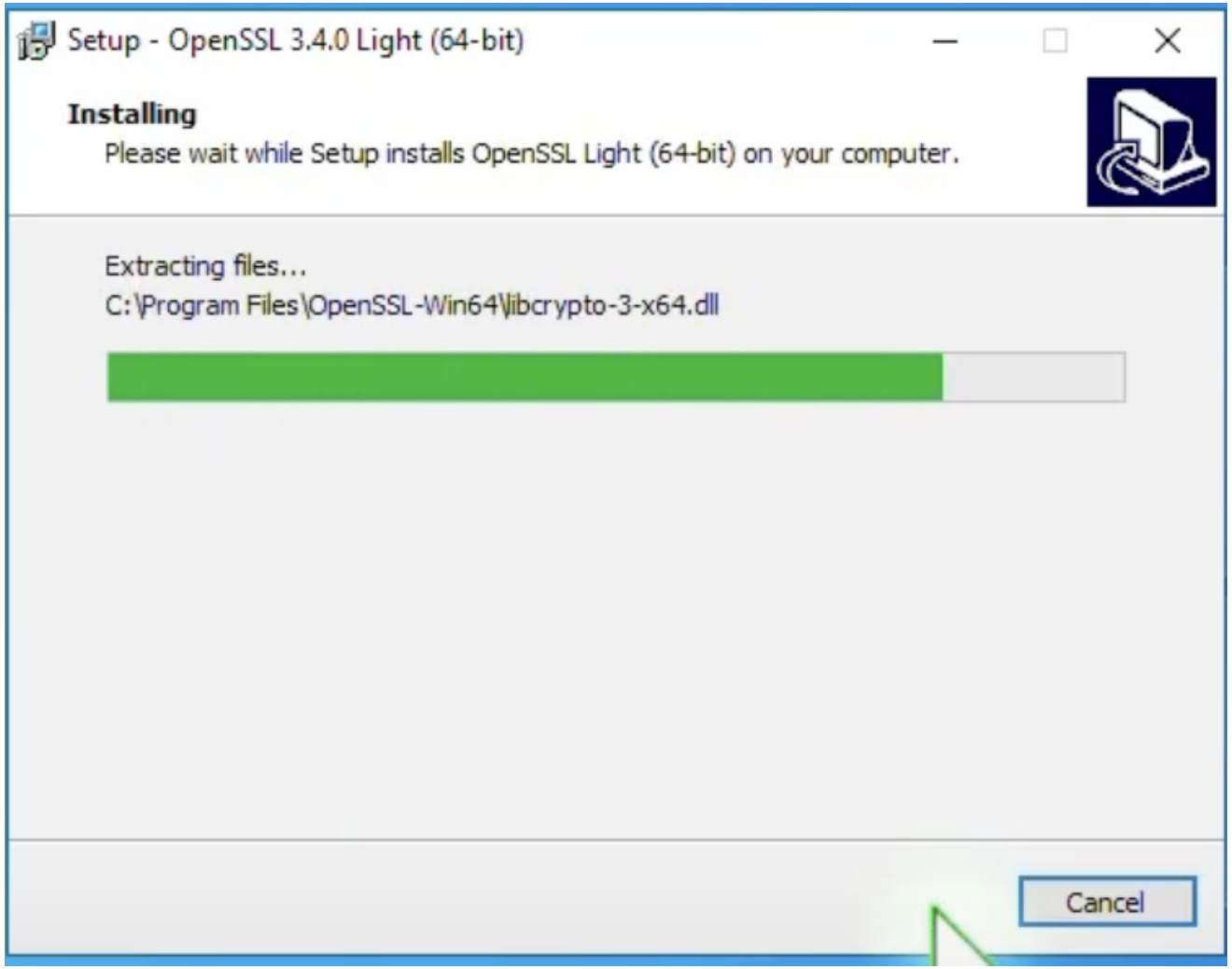
- ***Step 2:*** run the OpenSSL Installer (cont.)





Install OpenSSL on the Windows (cont.)

- ***Step 3:*** Installation in progress…….



Install OpenSSL on the Windows (cont.)

- ***Step 4:*** Finish OpenSSL Installation

Install OpenSSL on the Windows (cont.)

- ***Step 5:*** Set Environment Variables
  1. Open System Properties:
     - **Right-click** on **`This PC`** or **`Computer`** on your desktop or in File Explorer.
     - Select **`Properties`**.
     - Click on **`Advanced system settings`**.
  2. Set Path Variable:
     - System Properties window, click on the **`Environment Variables`** button.
     - Environment Variables window, under the **`System variables`** section, find and select the **`Path`** variable, then click **`Edit`**.
     - Click **`New`** and add the path to the OpenSSL `bin` directory: **C:\Program Files\OpenSSL-Win64\bin**.

Install OpenSSL on the Windows (cont.)

- ***Step 5:*** Set Environment Variable (cont.)
  3. Add OPENSSL_CONF variable:
     - Still in the Environment Variables window, click `New` under the `System variables` section.
     - Set the `Variable name` to `**OPENSSL_CONF**`.
     - Set the `Variable value` to the path of the `openssl.cfg` file. This is typically **C:\Program Files\OpenSSL-Win64\bin\openssl.cfg**

Install OpenSSL on the Windows (cont.)

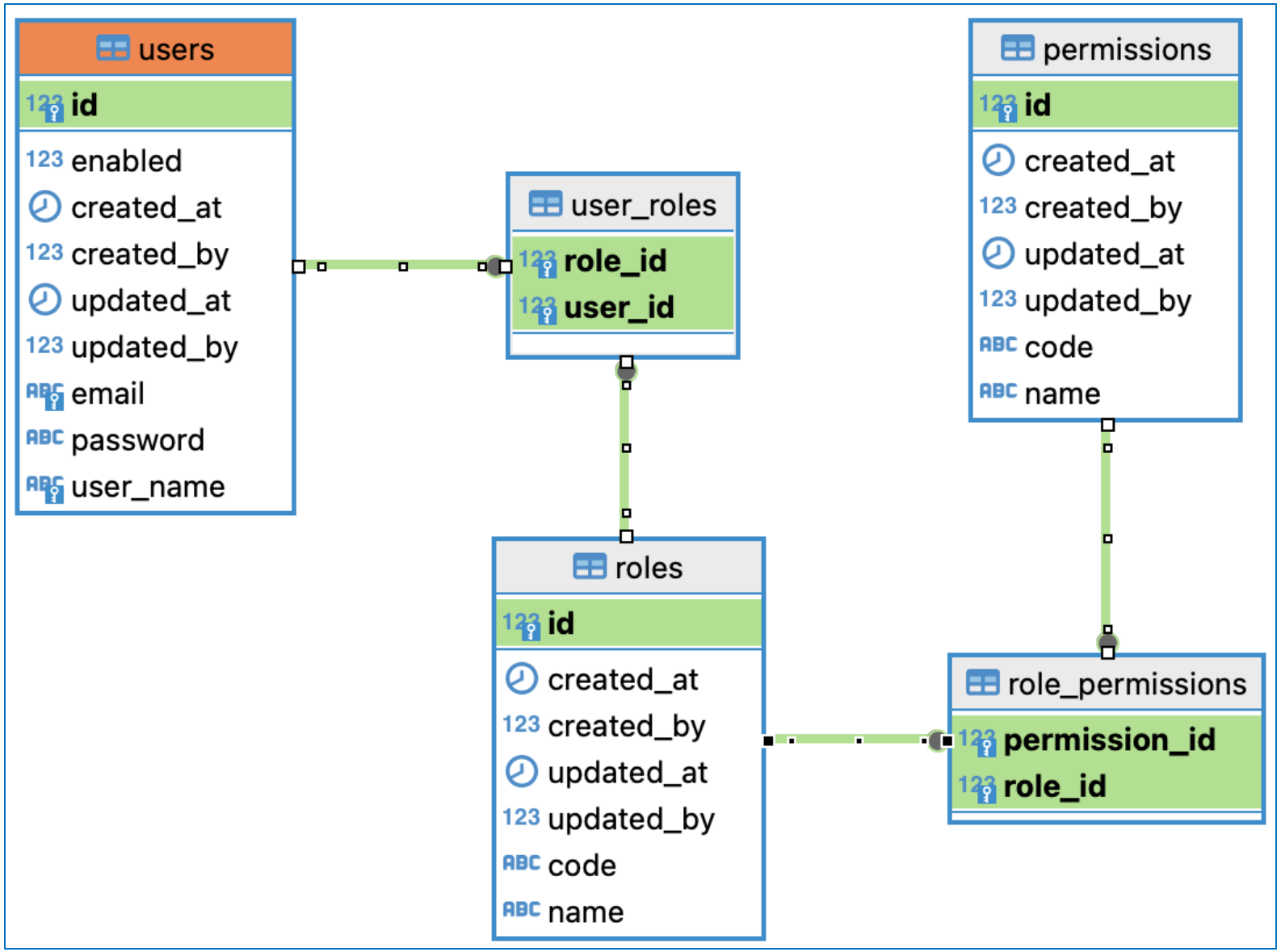
- ***Step 6:*** Verify Installation

```
thoaha@Has-MacBook-Pro rsa % openssl version
OpenSSL 3.4.0 22 Oct 2024 (Library: OpenSSL 3.4.0 22 Oct 2024)
```

- ***Step 7:*** Basic Usage
  - Document: https://docs.openssl.org/master/man1/#openssl-commands
  - Example: Generate a Private Key

    openssl genpkey -algorithm RSA -out private_key.pem

Authentication – Roles - Permissions

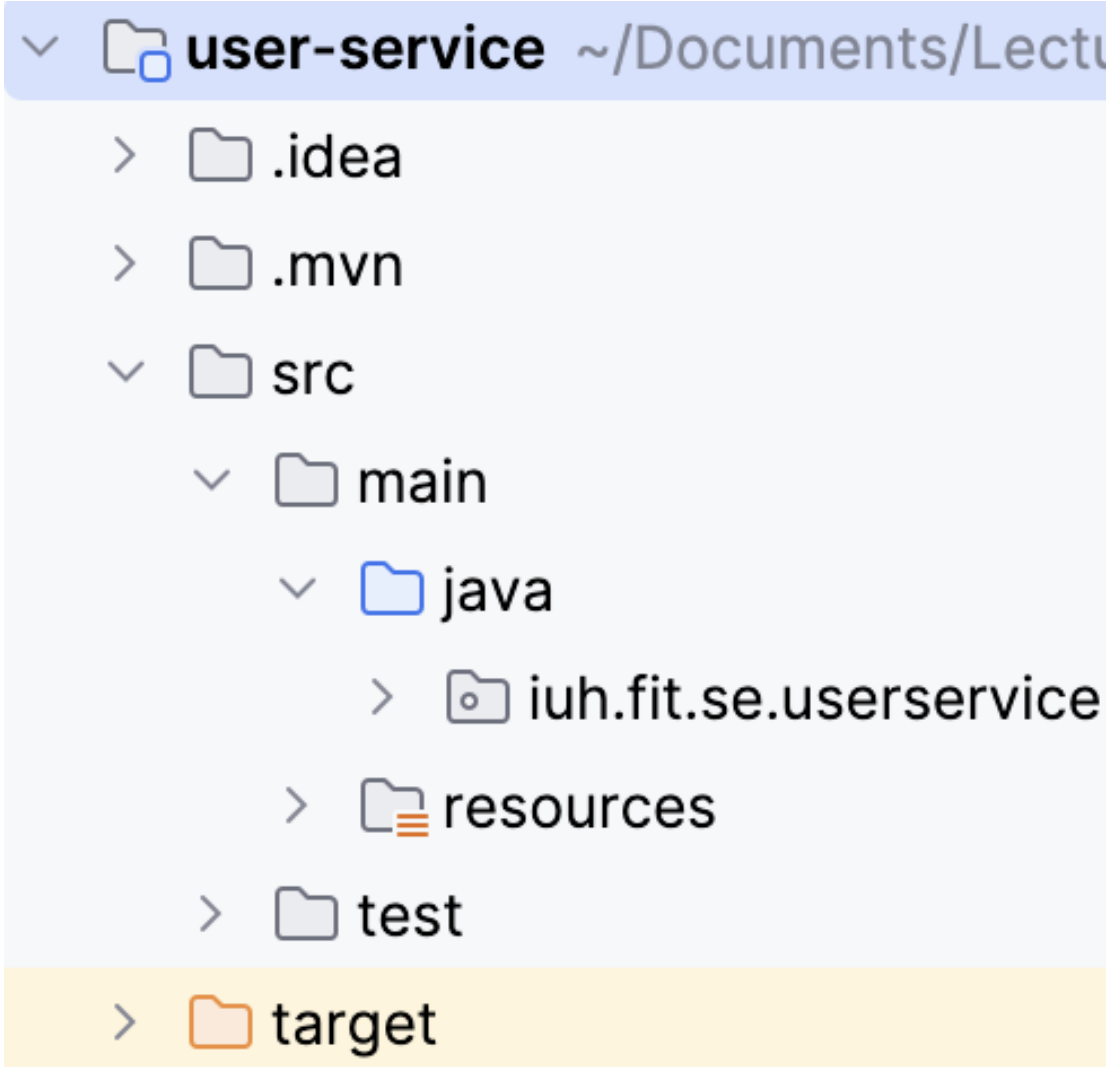


- ROLES:
  - *USER*
  - *ADMIN*
  - *SUPER_ADMIN*
- PERMISSIONS:
  - *READ*
  - *WRITE*
  - *DELETE*
  - *UPDATE*

IMPLEMENTATION GUIDE

Step 1: Create Project

1. ***Dependencies***
   - Spring Web
   - MariaDB Driver
   - Spring Data JPA
   - Lombok
   - **OAuth2 Resource Server**
   - Spring Configuration Processor
   - Spring Boot DevTools
2. ***Start a new Spring Boot project***



Step 2: Setting database

src/main/resources/application.properties

```
# Setting database
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.datasource.url=jdbc:mariadb://localhost:3306/spring_security_jwt\
  ?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=123456


spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
```

Step 3: Creating Entities and Generating Database Tables

1. ***Entities***
   - BaseEntity
   - User
   - Role
   - Permission
   - Token

=> Package:
entities

```
@Data  4 usages  4 inheritors
@Getter
@Setter
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public class BaseEntity implements Serializable {
    @CreatedDate
    private Date createdAt;
    @LastModifiedDate
    private Date updatedAt;
    @CreatedBy
    private Long createdBy;
    @LastModifiedBy
    private Long updatedBy;
}
```

Step 3: Creating Entities and Generating Database Tables

```java
@Entity
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = "user_name"),
        @UniqueConstraint(columnNames = "email")
})
public class User extends BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String userName;
    private String password;
    private String email;
    private boolean enabled;

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

Step 3: Creating Entities and Generating Database Tables

```java
@Entity
@Table(name = "roles")
public class Role extends BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String code;
    private String name;

    @ManyToMany(cascade = CascadeType.REFRESH, fetch = FetchType.EAGER)
    @JoinTable(name = "role_permissions",
            joinColumns = {@JoinColumn(name = "role_id")},
            inverseJoinColumns = {@JoinColumn(name = "permission_id")})
    private Set<Permission> permissions = new HashSet<>();
}
```

```java
@Entity
@Table(name = "permissions")
public class Permission
        extends BaseEntity {
    @Id
    @GeneratedValue(
            strategy = GenerationType.IDENTITY)
    private Long id;
    private String code;
    private String name;
}
```

Step 3: Creating Entities and Generating Database Tables

```java
@Entity
@Table(name = "tokens")
public class Token extends BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

    @Column(nullable = false, unique = true, length = 10000)
    private String token;

    @Column(nullable = false)
    private Instant expiryDate;
    public boolean revoked;
}
```

Step 4: Creating UserPrincipal class

- Create a **UserPrincipal** class which implements **UserDetails** interface, which provides core user information which is later encapsulated into Authentication objects

=> Package: ***auths***

```
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;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;}
    @Override
    public boolean isAccountNonExpired() { return true; }
    @Override  9 usages
    public boolean isAccountNonLocked() { return true; }
    @Override  9 usages
    public boolean isCredentialsNonExpired() { return true; }
}
```

Step 4: Creating Repositories

1. *UserRepository*
2. *RoleRepository*
3. *PermissionRepository*
4. *TokenRepository*

Step 5: Creating Services Interface

1. *UserService*
2. *RoleService*
3. *PermissionService*
4. *TokenService*
5. *AuthService*

Step 5: Creating Services Interface

```
public interface UserService {  8 usages  1 implem
    User findByUserName(String userName);  no us
    void saveUser(User user);  4 usages  1 impleme
    boolean existsByEmail(String email);  1 usag
    boolean existsByUserName(String userName);
}
```

```
public interface RoleService {  8 usages  1
    void saveRole(Role role);  3 usages  1
    List<Role> getAllRoles();  no usages  1
    Role getRoleByCode(String roleCode);
}
```

```
public interface PermissionService {  6 usages  1 ir
    void savePermission(Permission permission);
}
```

```
public interface TokenService {  13 us
    void saveToken(Token token);  2 us
    Token findByToken(String token);
}
```

Step 5: Creating Services Interface

```
public interface AuthService {  5 usages  1 implementation
    ResponseEntity<ApiResponse<?>> signUp(SignUpRequest signUpRequest)
            throws UserAlreadyExistsException;
    ResponseEntity<ApiResponse<?>> signIn(SignInRequest signInRequest);
}
```

Step 6: Implement Service

1. Create a ***UserDetailsServiceImpl*** class that implements the ***UserDetailsService*** interface, used to **retrieve user-related data,** using **loadUserByUsername(), and returns *UserPrincipal***

```java
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    private UserRepository userRepository;  2 usages
    @Autowired
    public UserDetailsServiceImpl(UserRepository userRepository) { this.userRepository = userR
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByUserName(username).orElseThrow(() ->
                new UsernameNotFoundException("User Not Found with username: " + username));
        Collection<GrantedAuthority> authorities = new ArrayList();
        if(null != user.getRoles()) {
            user.getRoles().forEach(role -> {
                authorities.add(new SimpleGrantedAuthority(role.getCode()));
                role.getPermissions().forEach(permission -> {
                    authorities.add(new SimpleGrantedAuthority(permission.getCode()));
                });
            });
        }
        return new UserPrincipal(user.getId(), user.getUserName(),
                user.getEmail(), user.getPassword(), authorities, user.isEnabled()
        );
    }
}
```

Step 6: Implement Service

2. AuthServiceImpl

```
public class AuthServiceImpl implements AuthService {
    @Override  1 usage
    public ResponseEntity<ApiResponse<?>> signUp(SignUpRequest signUpRequest)
            throws UserAlreadyExistsException {
        if (userService.existsByUserName(signUpRequest.getUserName())) {
            throw new UserAlreadyExistsException("Username already exist");
        }

        if (userService.existsByEmail(signUpRequest.getEmail())) {
            throw new UserAlreadyExistsException("Email already exist");
        }

        User user = createUser(signUpRequest);
        userService.saveUser(user);
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(ApiResponse.builder()
                        .status(String.valueOf( obj: "SUCCESS"))
                        .message("User account has been successfully created!")
                        .build()
        );
    }
```

Step 6: Implement Service

2. AuthServiceImpl

```
public ResponseEntity<ApiResponse<?>> signIn(SignInRequest signInRequest) {
    Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(signInRequest.getUserName(),
                    signInRequest.getPassword()));

    SecurityContextHolder.getContext().setAuthentication(authentication);
    UserPrincipal userDetails = (UserPrincipal) authentication.getPrincipal();
    User user = new User();
    user.setId(userDetails.getId());
    user.setUserName(userDetails.getUsername());

    SignInResponse signInResponse = SignInResponse.builder()
            .username(userDetails.getUsername()).email(userDetails.getEmail())
            .id(userDetails.getId()).roles(userDetails.getAuthorities()).build();
    return ResponseEntity.ok(
            ApiResponse.builder().status("SUCCESS").message("Sign in successfully!")
                    .response(signInResponse).build()
    );
}
```

Step 7: Create SecurityConfig

@Configuration
@EnableWebSecurity
@EnableMethodSecurity

```
public class SecurityConfig {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoder());
    }
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig)
            throws Exception {
        return authConfig.getAuthenticationManager();
    }
    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
```

Step 7: Create SecurityConfig

```
@Bean
public SecurityFilterChain signInSecurityFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            .securityMatcher((new AntPathRequestMatcher("/sign-in")))
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .userDetailsService(userDetailsService)
            .exceptionHandling(ex -> {
                ex.authenticationEntryPoint((request, response, authException) ->
                        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMessage()));
            })
            .httpBasic(Customizer.withDefaults());

    return httpSecurity.build();
}
```

Step 8: Create Controllers

```java
@RestController
public class AuthController {
    private static final Logger logger = LoggerFactory.getLogger(AuthController.class);  no usages
    private final AuthService authService;  3 usages
    @Autowired
    public AuthController(AuthService authService) { this.authService = authService; }

    @PostMapping("/sign-up")
    public ResponseEntity<ApiResponse<?>> registerUser(@RequestBody @Valid SignUpRequest signUpRequest)
            throws UserAlreadyExistsException {
        return authService.signUp(signUpRequest);
    }

    @PostMapping("/sign-in")
    public ResponseEntity<ApiResponse<?>> signInUser(@RequestBody @Valid SignInRequest signInRequest){
        return authService.signIn(signInRequest);
    }
}
```

Step 8: Create Controllers

```java
@RestController
@RequestMapping("/api")
public class DashboardController {
    @PreAuthorize("hasAnyRole('SCOPE_ROLE_USER','SCOPE_ROLE_ADMIN','SCOPE_ROLE_SUPER_ADMIN')")
    @GetMapping("/welcome-message")
    public ResponseEntity<String> getFirstWelcomeMessage(Authentication authentication) {
        return ResponseEntity.ok(
                body: "Welcome to user service: " + authentication.getName() +
                        " with scope: " + authentication.getAuthorities());
    }

    @PreAuthorize("hasAnyRole('SCOPE_ROLE_ADMIN','SCOPE_ROLE_SUPER_ADMIN') " +
            "or hasAuthority('SCOPE_PERMISSION_UPDATE')")
    @GetMapping("/admin-message")
    public ResponseEntity<String> getAdminData(Authentication authentication, Principal principal) {
        return ResponseEntity.ok( body: "Welcome to Admin Role: " + authentication.getAuthorities());
    }
}
```

Step 9: Create Controllers

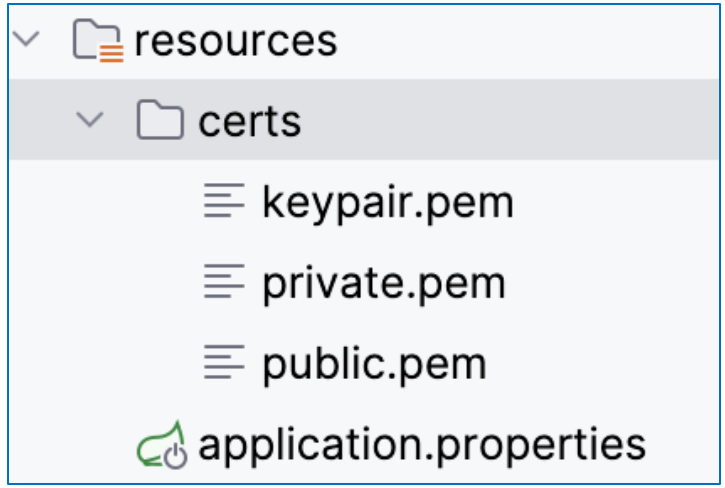
```java
@RestController
@RequestMapping("/api")
public class DashboardController {
    @PreAuthorize("hasAnyRole('SCOPE_ROLE_USER','SCOPE_ROLE_ADMIN','SCOPE_ROLE_SUPER_ADMIN')")
    @GetMapping("/welcome-message")
    public ResponseEntity<String> getFirstWelcomeMessage(Authentication authentication) {
        return ResponseEntity.ok(
                body: "Welcome to user service: " + authentication.getName() +
                        " with scope: " + authentication.getAuthorities());
    }

    @PreAuthorize("hasAnyRole('SCOPE_ROLE_ADMIN','SCOPE_ROLE_SUPER_ADMIN') " +
            "or hasAuthority('SCOPE_PERMISSION_UPDATE')")
    @GetMapping("/admin-message")
    public ResponseEntity<String> getAdminData(Authentication authentication, Principal principal) {
        return ResponseEntity.ok( body: "Welcome to Admin Role: " + authentication.getAuthorities());
    }
}
```

Step 10: JWT Access Token

https://docs.openssl.org/master/man1/

1. ***Generating Asymmetric Keys with OpenSSL:***



- Create certs folder and navigate to it:

  cd /src/main/resources/certs

- Generate a KeyPair: generates an RSA private key

  openssl genrsa -out keypair.pem 2048

- Generate a Public key from the private key

  openssl rsa -in keypair.pem -pubout -out public.pem

- Format the Private key:

  openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem -out private.pem

***Note*: pem -** Privacy Enhanced Mail chỉ là các tệp DER (Distinguished Encoding Rules - định dạng mã hóa nhị phân) được mã hóa Base64

pkcs8: private key information syntax standard

Step 10: JWT Access Token

2. *Setting jwt in the application.properties*

```
rsa.rsa-private-key=classpath:certs/private.pem
rsa.rsa-public-key=classpath:certs/public.pem
```

3. *Create RSAKeyRecord.class at configs package*

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
@ConfigurationProperties(prefix = "rsa")  4 usages
public record RSAKeyRecord(RSAPublicKey rsaPublicKey,
                           RSAPrivateKey rsaPrivateKey) { }  1 usage
```

Step 10: JWT Access Token

4. ***Using EnableConfigurationProperties to enable it to be found in properties file.***

```
@EnableConfigurationProperties(RSAKeyRecord.class)
@SpringBootApplication
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
```

Step 10: JWT Access Token

5. *Create 2 Beans jwtDecoder and jwtEncoder at configs/SecurityConfig.java*

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(rsaKeyRecord.rsaPublicKey()).build();
}

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(rsaKeyRecord.rsaPublicKey()).
            privateKey(rsaKeyRecord.rsaPrivateKey()).build();
    JWKSource<SecurityContext> jwkSource = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwkSource);
}
```

Step 10: JWT Access Token

6. *Create JwtTokenUtil at utils package*

```
public class JwtTokenUtil {
    private static final Logger logger = LoggerFactory.getLogger(JwtTokenUtil.class);  1 usag
    @SuppressWarnings("ReassignedVariable")  no usages
    public String generateToken(Authentication authentication, JwtEncoder jwtEncoder) {...}
    public String getUsernameFromToken(Jwt jwtToken) { return jwtToken.getSubject(); }
    private boolean isTokenExpired(Jwt jwtToken) {...}
    public boolean isTokenValid(Jwt jwtToken, UserPrincipal userPrincipal) {...}
    public Instant generateExpirationDate() {  1 usage
        return Instant.now().plus( amountToAdd: 10, ChronoUnit.MINUTES);
    }
}
```

Step 10: JWT Access Token

```
public String generateToken(Authentication authentication, JwtEncoder jwtEncoder) {
    String token = "";
    UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();
    try {
        Instant now = Instant.now();
        JwtClaimsSet claims = JwtClaimsSet.builder()
                .issuer("iuh.fit.se")
                .issuedAt(now)
                .expiresAt(generateExpirationDate())
                .subject(userPrincipal.getUsername())
                .claim( name: "scope", userPrincipal.getAuthorities()
                        .stream().map(r -> r.getAuthority()).collect(Collectors.toList()))
                .build();

        token = jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
    } catch (Exception e) {
        logger.error(e.getMessage());
    }
    return token;
}
```

Step 10: JWT Access Token

```
public String generateToken(Authentication authentication, JwtEncoder jwtEncoder) {
    String token = "";
    UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();
    try {
        Instant now = Instant.now();
        JwtClaimsSet claims = JwtClaimsSet.builder()
                .issuer("iuh.fit.se")
                .issuedAt(now)
                .expiresAt(generateExpirationDate())
                .subject(userPrincipal.getUsername())
                .claim( name: "scope", userPrincipal.getAuthorities()
                        .stream().map(r -> r.getAuthority()).collect(Collectors.toList()))
                .build();

        token = jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
    } catch (Exception e) {
        logger.error(e.getMessage());
    }
    return token;
}
```

Step 11: Call generateToken() method

```
public ResponseEntity<ApiResponse<?>> signIn(SignInRequest signInRequest) {
    Authentication authentication = …
    SecurityContextHolder.getContext().setAuthentication(authentication);
    String jwt = jwtTokenUtil.generateToken(authentication, jwtEncoder);
    UserPrincipal userDetails = (UserPrincipal) authentication.getPrincipal();
    User user = new User(); ….
    Token token = Token.builder().token(jwt).user(user)
                .expiryDate(jwtTokenUtil.generateExpirationDate())
                .revoked(false).build();
    tokenService.saveToken(token);
    SignInResponse signInResponse = SignInResponse.builder()….
            .token(jwt)
            .type("Bearer").build();
    ….
}
```

Step 11: *SecurityConfig.java*

```java
@Order(1)
@Bean
public SecurityFilterChain signUpSecurityFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            .securityMatcher((new AntPathRequestMatcher("/sign-up")))
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(authorize -> authorize.anyRequest().permitAll())
            .httpBasic(Customizer.withDefaults());

    return httpSecurity.build();
}
```

Step 11: *SecurityConfig.java*

```java
@Order(2)
@Bean
public SecurityFilterChain signInSecurityFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            .securityMatcher((new AntPathRequestMatcher("/sign-in")))
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(authorize -> authorize.anyRequest().permitAll())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .userDetailsService(userDetailsService)
            .exceptionHandling(ex -> {
                ex.authenticationEntryPoint((request, response, authException) ->
                        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMessage()));
            })
            .httpBasic(Customizer.withDefaults());

    return httpSecurity.build();
}
```

Step 12: *Add JwtAccessTokenFilter class*

```
@Component
public class JwtAccessTokenFilter extends OncePerRequestFilter {

    private JwtDecoder jwtDecoder;  2 usages
    private JwtTokenUtil jwtTokenUtil;  2 usages
    private UserDetailsServiceImpl userDetailsService;  2 usages
    private TokenService tokenService;  2 usages

    public JwtAccessTokenFilter(JwtDecoder jwtDecoder, JwtTokenUtil jwtTokenUtil,
                                UserDetailsServiceImpl userDetailsService, TokenService tokenService) {...}

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

Step 12: *SecurityConfig.java*

```
@Order(3)
@Bean
SecurityFilterChain apiFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity.securityMatcher((new AntPathRequestMatcher("/api/**")))
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(
                    authorize -> authorize.anyRequest().authenticated()
            )
            .oauth2ResourceServer((oauth2) -> oauth2.jwt(Customizer.withDefaults()))
            .sessionManagement(
                    session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .addFilterBefore(new JwtAccessTokenFilter(this.jwtDecoder(),
                    this.jwtTokenUil,this.userDetailsService,this.tokenService),
                    UsernamePasswordAuthenticationFilter.class)
            .exceptionHandling(
                    (ex) -> ex.authenticationEntryPoint(new BearerTokenAuthenticationEntryPoint())
                            .accessDeniedHandler(this.customAccessDeniedHandler))
            .httpBasic(Customizer.withDefaults());

    return httpSecurity.build();
}
```

Demo

Q&A