Java Technologies

Lecture 6

Securing Applications and Services

Fall, 2025

Agenda

- Basic Security Aspects, Authentication, Authorizarion
- Auditing, Non-Repudiation, Accountability
- Monolith vs. Microservices Authentication
- Securing a Traditional Spring App, RBAC vs. ABAC, CSRF
- Securing the Communication with HTTPS
- HTTP Sessions, The Session Fixation Problem
- Stateless Security, JSON Web Tokens (JWT)
- Authentication Managers and Providers, Subject, Principal
- Securing a REST Controller with JWT
- Single-Sign-On (SSO), OAuth, OIDC, Token Types
- Using OAuth & OIDC in a Spring Web App
- Creating a Resource Server in Spring

Securing Applications

- **Software Security**: Protecting applications against various threats and vulnerabilities.
- **The CIA Triad**
  - Confidentiality
  - Integrity
  - Availability

- **Desktop**
  - What kind of code is executed by the application, on the client machine?
  - Where does the code comes from?
  - Who wrote the code?
- **Web**
  - Who is accessing the application?
  - What operations does the client want to execute?

Basic Security Aspects

- **Authentication:** "Who are you?"
  - Form-based login
  - Token-based authentication
  - External providers
  - Multi-factor authentication (MFA)
- **Authorization:** "Are you allowed to do that?"
  - Role-Based Access Control (RBAC)
  - Attribute-Based Access Control (ABAC)
  - Method-level security
  - URL/endpoint restrictions
- **Secure communication**: "Is our conversation private?"
  - TLS/SSL (HTTPS)
  - Mutual TLS (mTLS)
  - Encrypted protocols for APIs: HTTPS, WSS
  - Certificate pinning

Proving Who Did What, and When

- **Auditing**: The systematic recording and examination of events, actions, and changes in an application or system, to ensure that your security controls are effective and compliant over time.
  - Event logging
  - Database auditing
  - Application-level auditing
  - Security event auditing
  - Centralized audit logging
- **Non-Repudiation**: Ensuring actions cannot be denied later
  - Digital signatures & Public Key Infrastructure (PKI)
  - Secure time-stamping services
- **Accountability**: "Who did that?" 

Monolith vs. Microservices Authentication

- **Traditional Monolith**
  - **Centralized authentication**: single login system.
  - **Session-based**: server keeps user sessions in memory or DB. User logs in once $\rightarrow$ session ID $\rightarrow$ used across all pages.
  - **Easy to implement**: single codebase, one security context.

Monolith vs. Microservices Authentication

- **Traditional Monolith**
  - **Centralized authentication**: single login system.
  - **Session-based**: server keeps user sessions in memory or DB. User logs in once $\rightarrow$ session ID $\rightarrow$ used across all pages.
  - **Easy to implement**: single codebase, one security context.

- **Microservices**
  - **Distributed architecture**: many small services.
  - **Stateless authentication**: JWT (JSON Web Tokens), OAuth2, OpenID Connect.
  - **API Gateway / Identity Provider** for centralized login (SSO).
  - **Tokens** are passed between services.
  - **More complex**: key management, token validation, service-to-service trust.

Securing a Traditional Spring Web App

1. Add the dependency: `spring-boot-starter-security`
   → All your application's endpoints are now protected.
   → Spring generates a default *user* with a random password.
2. Create a security configuration class
3. Configure authentication
   - In-Memory authentication
   - Database authentication
   - Password encoding
4. Configure authorization
   - Permit access to certain endpoints to all users and require authentication for all other requests.
   - Apply fine-grained control to specify different permissions for different URLs based on user roles.
5. Implement CSRF (Cross-Site Request Forgery) protection.

1. spring-boot-starter-security

- Spring generates a default *user* with a random password:

```
Using generated security password:
    7a47d931-ae76-423e-81f4-3658356413b2
Global AuthenticationManager configured
    with UserDetailsService bean
    with name inMemoryUserDetailsManager
```

- Accessing an endpoint from Postman:

```
401 Unauthorized: The request in unauthenticated
```

- Accessing an endpoint from browser:

2. Create a Security Configuration Class

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
          throws Exception { ... }
  // Defines filters that intercept every HTTP request
  // and apply security rules.

  @Bean
  public PasswordEncoder passwordEncoder() { ... }
  // Specifies how passwords are encoded

  @Bean
  public UserDetailsService userDetailsService() { ... }
  // Defines users and roles
}
```

The filter chain is registered with the servlet container through a **DelegatingFilterProxy** web filter. 

3. Configure Authentication

```
@Configuration
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
          throws Exception {
      http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults())  // default login page
        .logout(logout -> logout.permitAll()); // /logout endpoint

    return http.build();
  }
}
// authorizeHttpRequests: rules for different request paths
// Use .formLogin() for /web/** endpoints
// Use .httpBasic() only for /api/**.
```

Encoding Passwords

- Define how passwords are hashed and verified:

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

- **BCrypt** is a password hashing algorithm designed specifically for securely storing passwords.
  - **Slow and adaptive**: makes brute-force attacks harder.
  - **Salted**: adds a random value to each password before hashing, preventing rainbow table attacks.
- **Alternatives**
  - `Pbkdf2, Argon2, SCrypt`: modern, strong.
  - `MessageDigest(MD5), Standard(SHA256)` ← Don't use.
  - `NoOpPasswordEncoder`: just plain text, only for demos.

4. Configure Authorization

- First of all - where ar the users? 😟
  - In a in-memory list?
  - In a database?
  - Do you use an LDAP server?
  - Do you call a remote API to get user details?
- The **UserDetailsService** is the bridge between your unique user storage system and the Spring Security framework.

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

- A **UserDetails** implementation must provide the most basic method regarding user management.

```
getUsername(), getPassword(), getAuthorities()
isAccountNonExpired(), isEnabled(), etc.
```

Getting Users from a In-Memory List

- Define **users and roles**.

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails user = User.withUsername("user")
        .password(passwordEncoder().encode("secret"))
        .roles("USER")
        .build();
    UserDetails admin = User.withUsername("admin")
        .password(passwordEncoder().encode("qwerty"))
        .roles("ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
} // User is a predefined Spring class implementing UserDetails
```

- Configure **security rules**.

```
http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/user/**").hasRole("USER") )
```

Getting Users from a Database

- Assume we have the classes: `AppUser, UserRepository`

```
@Service
public class CustomUserDetailsService
        implements UserDetailsService {

  @Autowired private UserRepository userRepository;

  @Override
  public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
    AppUser user = userRepository.findByUsername(username)
      .orElseThrow(() -> new UsernameNotFoundException("Nope."));
    return new User(user.getUsername(), user.getPassword(),
      List.of(new SimpleGrantedAuthority(user.getRole())));
  }
}
```

- Make sure you encode passwords before storing them in DB.

```
userApp.setPassword(passwordEncoder().encode("secret"));
```

Method Level Security

- It allows using method-level **security annotations**.

```
@Configuration
@EnableMethodSecurity // ← Must be activated.
public class SecurityConfig {
    // SecurityFilterChain bean here.
}
```

- Works in controllers, services, or any managed bean.

```
@Service
public class AccountService {
    @PreAuthorize("hasRole('ADMIN')")      // @PostAuthorize
    public void deleteAccount(Long id) {
        // Only admins can delete accounts.
    }

    @Secured("USER")                       // @RolesAllowed
    public Account getAccount(Long id) {
        // Only users with USER role can access it.
    }
} // HTTP 403 Forbidden → AccessDeniedException
```

RBAC vs. ABAC

- **RBAC** = Role-Based Access Control
  Based only on roles: "Admin can delete all records."

- **ABAC** = Attribute-Based Access Control
  Based on multiple attributes (user, resource, environment, roles).

```
A user can edit a document if:
    user.department == document.department AND
    user.role == "EDITOR"
```

```
@Service
public class DocumentService {

  @PreAuthorize("principal.department == #document.department" +
      " and principal.role == 'EDITOR'")
  public void editDocument(Document document) {
      // edit logic
  }
} // Uses SpEL syntax
```

5. Cross-Site Request Forgery (CSRF)

- **Definition:** CSRF is an attack that tricks a user into executing unwanted actions on a web app where they are authenticated.
- **Mechanism:**
  - Attacker exploits the user's active session.
  - Malicious request (e.g., form submission) is sent without the user's knowledge: changing a password, making a purchase, etc.
- **Example**: User logs in to their bank website → User has an active session cookie in the browser → User visits the attacker's page while logged in → The attackers page contain a hidden form that submits a request to the bank → Bank processes the request. 💀
- **Prevention in Spring Security:**
  - CSRF tokens are automatically added to forms and validated on POST/PUT/DELETE operations.
  - For stateless APIs (e.g., JWT), CSRF can be disabled safely.

Securing the Communication with HTTPS

- HTTPS (Hypertext Transfer Protocol Secure)
- It uses a secure TLS (Transport Layer Security) channel to protect the data transmitted between a client and a server from being intercepted and read by a third party.
- A **TLS certificate** is required for the server, from a trusted Certificate Authority (CA) like Let's Encrypt, DigiCert.
  1. The server generates a (private, public) key pair.
  2. Server owner sends a certificate signing request (CSR).
  3. CA verifies the request and responds with a signed certificate.
- The communication between the client and the server:
  1. **TLS Handshake**: the server sends its certificate to the client.
     $\rightarrow$ asymmetric encryption to set up a secure connection
     $\rightarrow$ **session key**: known by the client & server
  2. **Symmetric Encryption** using the session key.

Using HTTPS in Spring

- Generate or obtain an SSL certificate

```
keytool -genkeypair -alias mydomain \
  -keyalg RSA -keysize 2048 -storetype PKCS12 \
  -keystore mykeystore.p12 -validity 3650
```

- Configure Spring Boot for HTTPS.

```
#application.properties
server.port=8443
server.ssl.key-store=classpath:mykeystore.p12
server.ssl.key-store-password=yourpassword
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=mydomain
```

- Redirect HTTP to HTTPS (optional, but recommended).
- Enforce only secure requests in the Spring config file.

```
http.requiresChannel(
  channel -> channel.anyRequest().requiresSecure())
```

HTTP Sessions

- **HTTP is stateless**.
- A **session** is a server-side mechanism that keeps user-specific data across multiple HTTP requests.
- By default, Spring Boot uses **Servlet container** sessions.
  - Session data is stored in server memory.
  - The session is identified via the JSESSIONID cookie.
  - Session is created lazily (only when needed).
  - Invalidated automatically when expired or explicitly.
  - Not suitable in clustered or distributed environments.
- **Distributed sessions**
  - Spring Boot can use external session stores via **Spring Session**.
  - **JDBC-Based**: Relational DB with dedicated tables.
  - **Redis-Based**: Redis is an in-memory key–value database, used as a distributed cache and message broker.

Using Sessions in Spring

- Access via Java EE standard `HttpSession` class.
  → Generic session storage independent of controllers.

```
@RestController
@RequestMapping("/session")
public class SessionController {

    @GetMapping("/set")
    public String setSession(HttpSession session) {
        session.setAttribute("user", "Alice");
        return "User set in session.";
    }

    @GetMapping("/get")
    public String getSession(HttpSession session) {
        return "User: " + session.getAttribute("user");
    }
}
```

- Access via `@SessionAttributes` in Spring MVC.
  → Model-driven session (multi-step forms).

The Session Fixation Problem

- Security vulnerability that allows an attacker to hijack a valid user's session by using a **session ID chosen by the attacker**.
  1. Attacker visits app → gets session ID ABC123.
  2. Attacker sends victim a link with ;jsessionid=ABC123
  3. Victim logs in with ABC123.
  4. Attacker uses ABC123 → logged in as victim.
- Spring protects against this by default. When you log in:
  - Old session is invalidated.
  - New session is created.
  - Security context (authentication) is migrated.
- **Best practices**: Always regenerate session ID after login (default), disable URL rewriting, use secure cookies, short session timeout (30min), invalidate session on logout.

```
# application.properties
server.servlet.session.cookie.secure=true
```

Stateless Security

- **Stateful security**: Session-based
  $\rightarrow$ Traditional web application model.
- **Stateless security**: Token-based
  $\rightarrow$ RESTful APIs and microservices.
- **Key principles** of stateless security
  - **No server-side sessions**, each request is independent.
  - Each request contains a **token**, either
    - self-contained – JWT (JSON Web Token), or
    - opaque – validated against a trusted authority.
  - **Stateless authorization**: Permissions are encoded in the token.
- **Advantages**: Horizontal scalability, resilience and fault tolerance, mobile and cross-platform friendly.
- **Disdvantages**: Token management (revocation, security), increased request size.

How Stateless Security Works

1. User logs in with credentials.
2. Server validates credentials and generates a signed token (JWT) that contains user identity and roles (called "claims").
3. The token is sent back to the client.
4. The client must send this token with every subsequent request, typically in the HTTP Authorization header.
5. The server does not store the token or any session. It only does three things:
    a. Receives the token from the request.
    b. Verifies the token's signature to ensure it was issued by a trusted source and hasn't been tampered with.
    c. Extracts user information and permissions from the token itself.

What is JWT?

- **JSON Web Token (JWT)** is a compact, URL-safe token format for securely transmitting claims between parties.
- **Structure:** `header.payload.signature`
  - *Header*: token type + signing algorithm (e.g., HS256)
  - *Payload*: claims (key-value pieces of information)
  - *Signature*: verifies integrity & authenticity
- **How it's used:** client sends `Authorization: Bearer <JWT>` to access protected resources (stateless).
- **Types of claims**:
  - Registered – standard, by the JWT specification.
  - Public – meant to be shared or used across different systems.
  - Private (Custom) – application specific claims.
- **Base64Url encoding**: `eyJhbGciOiJIUzI1NiIsInR5cCI...`

Example of a JWT Payload

```
{
  "sub": "alice",
  // Subject: usually the username or user ID

  "iat": 1692979200,
  // Issued At: when the token was created (Unix timestamp)

  "exp": 1692982800,
  // Expiration: when the token expires (Unix timestamp)

  "iss": "my-app",
  // Issuer: who issued the token

  "aud": "my-client-app"
  // Audience: who the token is intended for

  "https://myapp.com/email": "alice@example.com",
  // Public claim, use a URI namespace to avoid conflicts

  "roles": ["ROLE_USER"],
  // Custom claim: user roles/authorities
}
```

How to Generate a JWT?

Create a JWT utility class using **jjwt (Java JWT)** library.

```
@Component
public class JwtService {
  private final String secret = "mysecretkey..."; // at least 32 chars
  private final SecretKey key = Keys.hmacShaKeyFor(secret.getBytes());
  // You should store the secret in application.properties.

  public String generateToken(String username) {
    Map<String, Object> claims = new HashMap<>();
    // Map of public or custom claims

    var expDate = new Date(System.currentTimeMillis() + 3600_000);
    return Jwts.builder()
            .claims(claims)
            .subject(username)
            .issuedAt(new Date())
            .expiration(expDate)
            .signWith(key, Jwts.SIG.HS256)
            .compact();
    }
} // The generateToken method will be invoked at /login.
```

How to Parse & Validate a JWT?

```
@Service
public class JwtService {
  // ...
  public boolean isTokenValid(String token) {
      return parseToken(token).isPresent();
  } // You could check other claims too: nbf, iss, aud

  private Optional<Claims> parseToken(String token) {
      try {
          Claims claims = Jwts.parser()
              .verifyWith(getSigningKey())
              .build()
              .parseSignedClaims(token)
              .getPayload();
          return Optional.of(claims);
      } catch (JwtException e) {
          return Optional.empty();
      }
  }
} // or catch specific JwtException types such as ExpiredJwtException
```

Authenticating Users

- User sends username/password to `/login` endpoint.
- Spring creates an object (token) with credentials.
- You verify if the credentials are valid (`/login`), identify the principal and send as response a signed JWT token.

```
  @PostMapping("/login")
  public ResponseEntity<AuthResponse>
          login(@RequestBody AuthRequest req) {
    var token = new UsernamePasswordAuthenticationToken(
        req.username(), req.password()));
    Authentication auth =
        authenticationManager.authenticate(token);
    UserDetails userDetails = (UserDetails) auth.getPrincipal();
    String token = jwtService.generateToken(userDetails);
    return ResponseEntity.ok(new AuthResponse(token));
  } // Authentication = successfully authenticated principal.
```

```
// DTOs for login.
public record AuthRequest(String username, String password) {}
public record AuthResponse(String token) {}
```

Authentication Provider

- An `AuthenticationProvider` is a "guard at the door". It does the actual authentication logic: validate credentials and return an authenticated Authentication object).

```
@Bean
public DaoAuthenticationProvider daoAuthenticationProvider() {
  DaoAuthenticationProvider provider =
      new DaoAuthenticationProvider();
  provider.setUserDetailsService(userDetailsService);
  provider.setPasswordEncoder(passwordEncoder());
  return provider;
}
```

- There can be multiple providers defined.
  - Username/password login
  - LDAP authentication
  - JWT-based authentication

Authentication Manager

The `AuthenticationManager` loops through all registered `AuthenticationProviders`. The first one that says "I support this authentication type" and "credentials are valid" wins.

```
@Configuration
public class AuthManagerConfig {

    @Autowired
    private DaoAuthProvider daoAuthProvider;

    @Autowired
    private LdapAuthenticationProvider ldapAuthProvider;

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(daoAuthProvider, ldapAuthProvider);
        // order matters
    }
}
```

What Happens After Login?

We need to create an **authentication filter** that checks if an HTTP request **carries a valid JWT**, and if so, authenticates the user into the application's security context. Stateless security, remember?

```
@Component
public class JwtAuthFilter extends OncePerRequestFilter {
  @Autowired private JwtService jwtService;

  @Override
  protected void doFilterInternal(HttpServletRequest request,
                                  HttpServletResponse response,
                                  FilterChain filterChain)
              throws ServletException, IOException {
    // Extract JWT Token from header
    //     request.getHeader("Authorization").substring(7)
    //     The header must start with "Bearer "
    // Validate token (checks signature, expiration, issuer)
    // Set Authentication in the SecurityContext ← this comes next
    filterChain.doFilter(request, response);
  }
} // Why not use an ordinary filter? 💡
```

Subject, Principal, Authentication

- **Subject (sub)** is the identity included in the JWT token. Usually a user ID or username. The "who" of the token.
- **Principal** represents the identity of the currently authenticated user. It is created based on the subject of the token.
- **Credentials** represent a secret used to verify the identity: password at login, the key used for signing the token, etc.
  $\rightarrow$ The password is never included in a JWT.
- **Authorities / Roles** are the permissions granted to the principal (e.g., USER, ADMIN).
- **User / UserDetails** is a concrete representation of a user. Often used as the principal; holds username, password, and roles.
- **Authentication** is an interface representing the current authentication request or context. It contains principal, credentials, authorities, details.

Setting Authentication in the SecurityContext

```
// In the JwtAuthFilter.doFilterInternal method
// ...
String token = authHeader.substring(7);
String username = jwtService.extractUsername(token);
SecurityContext context = SecurityContextHolder.getContext();

if (username != null && context.getAuthentication() == null) {
  // Find the user in our user repository
  var userDetails = userDetailsService.loadUserByUsername(username);

  if (jwtService.isTokenValid(token)) {
    var authToken =
        new UsernamePasswordAuthenticationToken(
            userDetails,
            null, // We don't care for credentials
            userDetails.getAuthorities());
    authToken.setDetails(
        new WebAuthenticationDetailsSource().buildDetails(request));

    context.setAuthentication(authToken);
  }
}
```

Securing a REST Controller with JWT

- Configure stateless security in the config class.

```
http
  .csrf(AbstractHttpConfigurer::disable)
  .sessionManagement(sm ->
    sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

- **Register the authentication filter in the security chain.**

```
http.addFilterBefore(jwtAuthFilter,
     BasicAuthenticationFilter.class);

// BasicAuthenticationFilter: for HTTP Basic auth headers.
// UsernamePasswordAuthenticationFilter: for login form requests.
```

`JwtAuthFilter` is best placed before `BasicAuthenticationFilter` in pure stateless JWT APIs.

Extracting Claims in Controllers

- When your `JwtAuthFilter` validates the token, it usually: extracts the username (sub) → loads UserDetails → stores an `Authentication` token in the `SecurityContext`.

```
@GetMapping("/auth")
public String auth(Authentication authentication) {
    return "Hello, " + authentication.getName();
} // You have access to name, credentials, authorities.
```

- But you might also want access to the other claims? Right?
- Create a custom `Principal` (POJO) and store claims in it.

```
public class MyUserPrincipal {
    private String username, email, orgId; // ...
}
```

```
@GetMapping("/claim")
public String claim(@AuthenticationPrincipal MyUserPrincipal u) {
    return u.getEmail();
}
```

Declarative vs. Programatic Security

- **Declarative Security**: Security rules are defined using **annotations or configuration**, not inside the method logic.

```
@PreAuthorize, @PostAuthorize, @Secured, @AuthenticationPrincipal
@DenyAll, @PermitAll, @RolesAllowed // Java EE standard
```

- **Programatic Security**: Security decisions are made **inside the code**, using the `SecurityContext` or `Authentication` object.

```
public void editDocument(Document doc) {
    Authentication auth =
        SecurityContextHolder.getContext().getAuthentication();
    User user = (User) auth.getPrincipal();

    if (!user.getId().equals(doc.getOwnerId())) {
        throw new AccessDeniedException(
            "You do not own this document");
    }
    // method logic
}
```

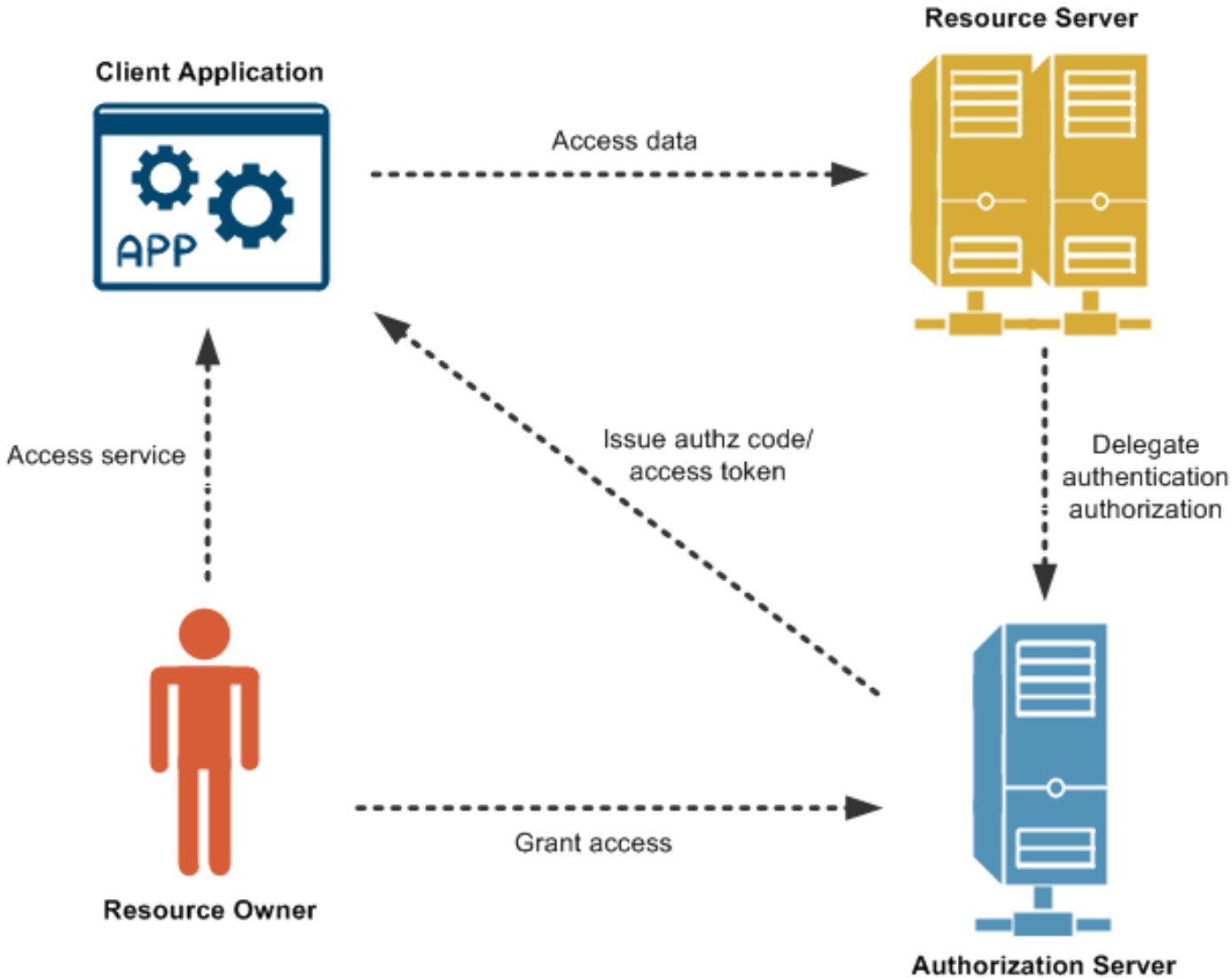
Single-Sign-On (SSO)

- SSO is an authentication process that allows a user to **access multiple applications or services** with just one set of login credentials (username and password).
- **The Identity Provider (IdP)** is the system that authenticates the user and confirms their identity.
- **The Service Provider** is the application or system that relies on the IdP to authenticate users. It trusts the IdP.
- **The Flow**: You try to access a SP $\rightarrow$ SP redirects you to the IdP $\rightarrow$ You log in at the IdP $\rightarrow$ IdP sends a token back to SP $\rightarrow$ SP grants access.
- **Common SSO Protocols**:
  - Open Authorization(OAuth), OpenID Connect (OIDC)
  - SAML (Security Assertion Markup Language)
  - Kerberos

Open Authorization (OAuth)

- OAuth is an **authorization framework**.
  $\rightarrow$ Its purpose is to grant an application or service permission to access protected resources on behalf of a user, without giving away the user's password.
- **Key Roles**
  - Resource Owner: The user who owns the data.
  - Resource Server: The API that holds the user's data.
  - Client: The app that want's to access the user's data.
  - Authorization Server: The server that authenticates the user.
- **The Central Concepts**
  - Authorization Code
  - Access Token
  - Refresh Token (optional)
  - Two-step authorization process

Common OAuth Flow



JWT Tokens vs. Opaque Tokens

- **JWT Tokens**
  - Self-contained, include claims (e.g., user, roles, expiry)
  - Digitally signed for integrity and authenticity
  - APIs can validate locally (no introspection call)
  - Fast and scalable, but harder to revoke early (blacklist)
  - Larger size (Base64-encoded JSON)
- **Opaque Tokens**
  - Random string with no readable claims
  - Validation requires introspection at Authorization Server
  - Smaller and simpler format
  - Easier to revoke centrally
  - More secure since clients cannot inspect content
  - Adds latency due to validation round-trip

Access Tokens vs. Refresh Tokens

- **Access Tokens**
  - Short-lived (minutes)
  - Used to access APIs directly
  - Often JWT (self-contained, verifiable signature)
  - Stored in memory
  - Should contain minimal claims (e.g., user id, roles, expiry)

- **Refresh Tokens**
  - Long-lived (hours or days)
  - Used to obtain new access tokens
  - Usually implemented as opaque tokens (random strings)
  - Must be stored securely at the client side (e.g., HTTP-only cookie, secure storage)
  - Supports rotation and revocation strategies
    Important for logout, password changes, or detected breaches.

OpenID Connect (OIDC)

- OIDC is an **authentication layer** built on top of OAuth 2.0.
  $\rightarrow$ Its purpose is to verify the user's identity.
- OIDC standardizes how to get identity information by using an **ID token**. When the Client exchanges the authorization code, the AS returns not just an access token, but also an ID Token.
- The ID Token is a JWT that contains **claims about the user's authentication**. It is cryptographically signed by the Authorization Server, so the Client can verify it was issued by a trusted party and hasn't been tampered with.
- **OIDC Use Cases:**
  - Customer Identity & Access Management (CIAM)
  - Single Sign-On (SSO)

Using OAuth & OIDC in Spring Boot

- Add the starter: `spring-boot-starter-oauth2-client`
- Define your OAuth credentials in the Google Cloud Console.
- Specify the authorized redirect URIs:

```
http://localhost:8080/login/oauth2/code/google
# add your production URL too
```

- Configure application properties

```
# OIDC with Google
spring.security.oauth2.client.registration.google.client-id
    =YOUR_GOOGLE_CLIENT_ID
spring.security.oauth2.client.registration.google.client-secret
    =YOUR_GOOGLE_CLIENT_SECRET
spring.security.oauth2.client.registration.google.scope
    =openid,profile,email

spring.security.oauth2.client.provider.google.issuer-uri
    =https://accounts.google.com
```

OAuth Security Configuration

By default, Spring Security auto-configures login at `/oauth2/authorization/google` and redirects to `/login/oauth2/code/google`.

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {

        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/index").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login();
            // Browser-based login flow for users
        return http.build();
    }
}
```

Creating a Web Controller Using OIDC

```java
@Controller
public class HomeController {

    @GetMapping("/")
    public String index() {
        return "index";
    }

    @GetMapping("/profile")
    public String profile(Model model,
            @AuthenticationPrincipal OidcUser oidcUser) {
        model.addAttribute("userName", oidcUser.getFullName());
        model.addAttribute("userEmail", oidcUser.getEmail());
        return "profile";
    }
}
```

`OidcUser` is the interface representing an authenticated user from an OpenID Connect (OIDC) provider (like Google). It extends `OAuth2User` and adds OIDC-specific claims: email, fullName, etc.

Creating a "Login with Google" Page

- The `index.html` page.

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <h2>Welcome!</h2>
  <a href="/oauth2/authorization/google">Login with Google</a>
  </body>
</html>
```

- The `profile.html` page.

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <h2>User Profile</h2>
  <p>Name: <span th:text="${userName}"></span></p>
  <p>Email: <span th:text="${userEmail}"></span></p>
  <a href="/logout">Logout</a>
</body>
</html>
```

- Make sure they are in `src/main/resources/templates`.

Resource Server

- **A Resource Server is your API backend.**
- **Protects Resources**: It holds protected data or services, only grants access to clients that provide a valid access token.
- **Accepts Access Tokens**: Clients send requests with an access token, typically in the Authorization header.
- **Validates Tokens**
  - **Signature**: To ensure the token hasn't been tampered with.
  - **Expiration**: To ensure the token is still active.
  - **Issuer**: To confirm the token came from a trusted AS.
  - **Audience/Scope**: To verify the intention and the permissions.
- **Extracts Authorization Information**: Based on the user ID or roles/scopes it makes an access control decision.
  Example: The University IT system. It has many apps like student portal, faculty portal, grading – they act like RS which trust tokens from a central IdP which is the AS.

Creating a Resource Server in Spring

- `spring-boot-starter-oauth2-resource-server`
- Configure application.properties (Google as provider).
  You need to tell your resource server where to find the public key (JWK – JSON Web Key) to validate the JWT signatures.

```
spring.security.oauth2.resourceserver.jwt.issuer-uri
    =https://accounts.google.com
```

- Security Configuration

```
http.oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> {}))
// Uses the JWK Set URI from application.properties
```

- Access the claims in the REST controllers.

```
    @GetMapping("/api/user")
    public Object userInfo(@AuthenticationPrincipal Jwt jwt) {
        return jwt.getClaims(); // returns all claims
    } // Jwt is a class from oauth2-resource-server module
```

oauth2Login vs. oauth2ResourceServer

- **.oauth2Login():**
  - Browser-based login flow (SSO)
  - Uses OAuth2 / OIDC for user authentication
  - Session-based by default
  - Typical clients: web apps (Thymeleaf)
  - Access user via `@AuthenticationPrincipal OidcUser`

oauth2Login vs. oauth2ResourceServer

- **.oauth2Login():**
  - Browser-based login flow (SSO)
  - Uses OAuth2 / OIDC for user authentication
  - Session-based by default
  - Typical clients: web apps (Thymeleaf)
  - Access user via `@AuthenticationPrincipal OidcUser`
- **.oauth2ResourceServer():**
  - Protects REST APIs / microservices
  - Validates OAuth2 tokens (JWT or opaque)
  - Stateless authentication
  - Typical clients: mobile apps, SPAs, other services
  - Access token via `@AuthenticationPrincipal Jwt`