Veebirakenduse projekt (ITI0302)

Loeng 8

Authentication: passwords, tokens, Spring Security, the frontend side · Siim Rebane

The problem

- The backend of lecture 6 answers anyone. From any computer on the internet:

```
DELETE /api/employees/2
```

- nginx forwards `/api` to the backend; nothing on the way asks who is calling
- The application does not know **who** the caller is, and has no idea **what that caller may do**
- Every endpoint is open to every visitor, the ones that change or delete data included
- An application that keeps data for people needs two decisions on every request:
  - Who is this? → **authentication**
  - May they do this? → **authorization**

Authentication and authorization

- **Authentication**: establishing who the caller is
  - the caller proves an identity, usually with something only they know: a password
  - the application checks the proof
- **Authorization**: deciding whether that identity may do the requested thing
  - an ordinary user may read the list; only an administrator may delete from it
- Two different questions, two different answers from the server:
  - no proof of identity, or a bad one → **401 Unauthorized**
  - identity known, this action not allowed for it → **403 Forbidden**
- 401 is a misnomer that stuck: it means "not authenticated". 403 is the real "not authorized".

Proving an identity once, then on every request

- HTTP has no memory: each request is on its own; the server does not know that the previous one came from the same person
- So authentication has two parts:
    i. **Logging in**: the caller sends username and password once, the server checks them
    ii. **A proof for every later request**: the server gives the caller something to send instead of the password, and checks it on every request
- The password is sent exactly once, at login, and the application never stores it
- Two ways to make the proof: a **session** or a **token**

Hashing, encryption, encoding

- Three different things that are often mixed up. None of them is the other.
- **Hashing**: any input → a fixed-size value. One way: nothing turns the value back into the input. No key. The same input always gives the same value.
  - for: storing passwords, checking that data has not changed
- **Encryption**: input → unreadable output, with a key. Reversible: whoever has the key reads the input again.
  - for: data that somebody must read back later: HTTPS traffic, a disk, a message
  - not for passwords: whoever gets the key gets every password
- **Encoding**: input → the same data written differently. No key, no secret: anyone can reverse it.
  - for: carrying bytes where only certain characters are allowed: base64 in a header, `%20` in a URL
  - protects nothing
- In this lecture: a password is **hashed**; the parts of a JWT are **encoded**; the connection is **encrypted** (HTTPS); a signature is a hash computed with a key

What the application stores of a password

- THE RULE: a password is **never stored**, never logged, never put in a URL, never sent back
- The database holds a **hash** of it: the result of a function that turns any input into a fixed-size value and cannot be run backwards

```
sha256("test")  = 9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
sha256("Test")  = 532eaabd9574880dbf76b9b8cc00832c20a6ec113d682299550d7a6e0f345e25
```

- At login the application hashes what the caller sent and compares the two hashes
- Someone who copies the database gets hashes, not passwords

Why a plain hash is not enough

- THE PROBLEM 1: same password → same hash
  - two users with the password `test` have identical rows
  - **rainbow table**: precomputed hashes of common passwords; a copied database turns into the passwords of everyone who chose a common one
- THE PROBLEM 2: hashing is fast
  - SHA-256 was designed for speed; a graphics card computes billions per second
  - every password of eight characters against a stolen hash: hours to days
- THE FIXES:
  - **salt**: a random value per user, mixed into the hash and stored next to it → the same password gives different hashes
  - **slow hash**: a function with a configurable amount of work → a thousandth of a second per guess for the user, years for the attacker

bcrypt

- **bcrypt** is a password hash that does both: it generates the salt itself and repeats its work 2^{cost} times
- One string holds everything the check needs:

```
$2a$10$ixEk4dMoBuMXqops1WbPgOfMykF9PuDHphIRyGVQ9ZwHyUJ5PLv2G
 │   │  └── 22 characters of salt, then 31 characters of hash
 │   └── cost 10: 2^10 = 1024 rounds, about 60 ms on a laptop
 └── the algorithm version
```

- Two users of the sample, registered with the **same** password `correct horse`:

```
 id | username | role  |                        password_hash
----+----------+-------+-------------------------------------------------------------
  1 | siim     | ADMIN | $2a$10$ixEk4dMoBuMXqops1WbPgOfMykF9PuDHphIRyGVQ9ZwHyUJ5PLv2G
  2 | mari     | USER  | $2a$10$N8SW0XdScz2aoc5O/5m83.o0jJ6lhm70fUCF9KyYWJ73lDuRn9q0e
```

- Different salts, different hashes
- The cost can be raised as computers get faster; old hashes keep working, the cost is inside the string

`PasswordEncoder`

- Spring Security provides bcrypt as a bean; the application never sees a salt or a round count

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

```
String hash = passwordEncoder.encode("correct horse");          // at registration: a new salt, a new hash
boolean ok  = passwordEncoder.matches("correct horse", hash);  // at login: true
```

- `encode` : a new salt, a new hash, every time
- `matches` : reads the salt and the cost out of the stored string, hashes the candidate the same way, compares
- There is no `decode` : nothing turns the hash back into the password

Logging in

- The login is a `POST` with the credentials in the body

```
POST /api/login
Content-Type: application/json

{"username": "siim", "password": "correct horse"}
```

- Never a `GET` with the credentials in the URL: URLs end up in browser history, server logs and proxies
- The server does three things:
  i. finds the user by name
  ii. `matches` the password against the stored hash
  iii. if both succeed, makes the proof the caller sends from now on
- A wrong name and a wrong password get the **same** answer, 401: the response must not reveal which usernames exist

A session

- You log in: username and password, the server checks them
- The server makes up a random string, say `k3f9…`, and writes down: `k3f9… = siim, logged in at 14:02`
  - the record is the **session**, the string is the **session id**
- The server gives the string to the browser
- The next request brings the string back. The server looks it up, finds `siim`, and knows who is calling without a password.
- The string itself means nothing: the meaning is in the server's record
  - delete the record and the string is dead: a logout from the server side
- The server keeps a record for every logged-in user, and a second backend instance has to see the same records: a shared store, the database for example, not the memory of one instance
- This is how server-rendered web applications have worked for decades, and what Spring Security does when nothing else is configured

Cookies

- A **cookie** is a small piece of data that the browser stores for one site and sends back to that site by itself, with every request
- The server sets it in a response header, a name and a value:

```
HTTP/1.1 200
Set-Cookie: JSESSIONID=k3f9a2c1; Path=/; HttpOnly
```

- From then on every request to the site carries it, in a request header:

```
GET /employees
Cookie: JSESSIONID=k3f9a2c1
```

- The application code in the browser does nothing: no JavaScript, no header to add. A server-rendered site logs in with a form and a redirect and nothing else.
- A cookie belongs to the site that set it. The browser never sends it to another site, and a page of another site cannot read it.
- The browser sends it on every request to the site, whichever page started the request

Cookie flags

- The attributes after the value tell the browser how to treat the cookie:

Attribute	Meaning	Without it
`Expires` / `Max-Age`	keep it until then, across browser restarts	a **session cookie**: gone when the browser closes
`Path=/`	send it for every path of the site	only for paths under the one that set it
`Domain`	also send it to subdomains	this host only
`Secure`	send it over HTTPS only	also over plain HTTP, readable on the network
`HttpOnly`	JavaScript on the page cannot read it	`document.cookie` returns it, and so does any injected script
`SameSite=Strict` / `Lax`	not sent on requests another site started; `Lax` allows a plain link click	the browser's default: Chrome treats a missing `SameSite` as `Lax`

- A login cookie is set with `HttpOnly`, `Secure` and `SameSite`: the three that protect it. `JSESSIONID` above has one of the three.
- A cookie is at most about 4 KB, and the browser keeps a few dozen per site. A session id is 32 characters; a JWT fits too.

A token

- The other way: the server writes nothing down. It writes the fact into the string itself, `siim, ADMIN, valid until 15:02`, and **signs** it with a secret only the server has.
- The next request brings the string back. The server checks the signature: if it matches, the content is true, because nobody else could have signed it. No lookup, no record.
- The server keeps nothing per user, so any instance with the secret can check any token: that is what **stateless** means here
- The price: the server cannot take a token back. There is no record to delete; it is valid until the date inside it.
- How it travels is a separate choice: a cookie the browser sends by itself, or a header the frontend's code adds. A token works both ways; the sample uses the header.
- The common format for this: the **JSON Web Token**, JWT

Session and token

	Session	Token
the server keeps per user	a record: **stateful**	nothing: **stateless**
what the client holds	a random id with no meaning of its own	the identity and the expiry, signed
what the server holds	the records	only the signing secret
check on each request	look the record up	verify the signature
logging someone out from the server side	delete the record	not possible until the token expires
two backend instances	both need the same store	both need the same secret
how it travels	a cookie	a cookie or a header; the sample: a header
typical use	server-rendered pages	a separate frontend talking to an API

- The course project is the last row: a Vue frontend and a REST backend
- The sample uses a JWT in the `Authorization` header

The three parts of a JWT

- A JWT is three strings joined with dots: `header.payload.signature`. The token the sample gave to `siim`:

```
eyJhbGciOiJIUzI1NiJ9
.eyJzdWIiOiJzaWltIiwidXNlcklkIjoxLCJyb2xlIjoiQURNSU4iLCJpYXQiOjE3OTIwMzE3NjQsImV4cCI6MTc5MjAzNTM2NH0
.fYtfgcNcSUb1MftbsfRRCeG5p7qgcxm8XSBWnZcKiwY
```

- **header**: the signing algorithm
- **payload**: the **claims**, statements about the caller, as JSON
- **signature**: a value computed from the first two parts and a secret
- The first two parts are **base64url**: bytes written as letters, digits, `-` and `_`, so that they survive in a header
- Encoding is not encryption: anyone can decode them
- https://jwt.io decodes a token in the browser

The payload is readable

- The same token, first two parts decoded:

```
header  : {"alg":"HS256"}
payload : {"sub":"siim","userId":1,"role":"ADMIN","iat":1792031764,"exp":1792035364}

iat = 2026-10-15 02:36:04 UTC    exp = 2026-10-15 03:36:04 UTC
```

- `sub` : subject, whom the token is about
- `iat` , `exp` : issued at, expires; seconds since 1970
- `userId` , `role` : the application's own claims
- Anyone holding the token reads all of this
- So a payload holds what the server needs on each request and **nothing secret**: no password hash, no e-mail the caller should not see

The signature

- The third part is an **HMAC**: a hash of the first two parts computed together with a secret only the server knows
- `HS256` = HMAC with SHA-256

```
signature = HMAC-SHA256( base64url(header) + "." + base64url(payload), secret )
```

- The server recomputes it on every request and compares
- Change one character of the payload, `"role":"USER"` into `"role":"ADMIN"`, and the signature no longer matches:

```
$ curl -i localhost:8088/api/employees -H "Authorization: Bearer $TAMPERED"
HTTP/1.1 401
```

```
WARN  JwtAuthFilter : rejected token: JWT signature does not match locally computed signature.
```

- Anyone can read a token. Only the holder of the secret can make one.

Expiry

- A token cannot be taken back, so it must **end by itself**: `exp` is checked on every request
- A token past its `exp` is rejected like a forged one:

```
$ curl -i localhost:8088/api/me -H "Authorization: Bearer $TOKEN"
HTTP/1.1 401
```

```
WARN  JwtAuthFilter : rejected token: JWT expired 3043 milliseconds ago at 2026-10-15T02:36:33.000Z.
```

- The sample's expiry is 60 minutes
- The trade-off:
  - short expiry → a stolen token is worth little
  - long expiry → the user logs in rarely

Access token and refresh token

- The usual way out of that trade-off is two tokens:

Token	Lifetime	Stored by the server	Used for
access token	minutes	no	every API request
refresh token	days or weeks	yes, in the database	getting a new access token when the old one expires

- The access token is the JWT of this lecture
- The refresh token is a long random string with a row in the database, so it can be revoked: that row is what a "log out everywhere" button deletes
- A `/refresh` endpoint takes the refresh token and returns a fresh access token, with no password
- A refresh token is a session by another name: a random string, a record on the server, revocable by deleting the record. Only the access token is stateless, and only for its few minutes.
- The sample has the access token only, 60 minutes. For the course project that is enough.

Adding Spring Security

- Maven:

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

- Gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```

- With that line and no configuration the application locks itself:

```
$ curl -i localhost:8088/api/employees
HTTP/1.1 401
WWW-Authenticate: Basic realm="Realm", charset="UTF-8"
Set-Cookie: JSESSIONID=662D727912EA0FFF5F653ACB9D2EC33B; Path=/; HttpOnly
```

```
Using generated security password: ae0f702a-22d6-40ea-b46e-68da23c6256b
```

- Every URL asks for a login, `/api/version` included; one user `user` with a password printed to the log; a session cookie; HTTP Basic
- None of that is what an API with a frontend wants. The configuration replaces the defaults.

The JWT library

- **jjwt** (`io.jsonwebtoken`) makes and checks tokens. Three artifacts: the API, the implementation and the JSON mapping.
- Maven, with `spring-boot-starter-security-test` for the tests:

```
<dependency><groupId>io.jsonwebtoken</groupId><artifactId>jjwt-api</artifactId><version>0.13.0</version></dependency>
<dependency><groupId>io.jsonwebtoken</groupId><artifactId>jjwt-impl</artifactId><version>0.13.0</version><scope>runtime</scope></dependency>
<dependency><groupId>io.jsonwebtoken</groupId><artifactId>jjwt-jackson</artifactId><version>0.13.0</version><scope>runtime</scope></dependency>
<dependency><groupId>org.springframework.security</groupId><artifactId>spring-security-test</artifactId><scope>test</scope></dependency>
```

- Gradle:

```
implementation 'io.jsonwebtoken:jjwt-api:0.13.0'
runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.13.0'
runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.13.0'
testImplementation 'org.springframework.security:spring-security-test'
```

Filters

- A **filter** is code that a request passes through before it reaches the controller, and the response passes through on the way back
- Several filters in a row make a **filter chain**; each filter can
  - look at the request and pass it on: `chain.doFilter(...)`
  - change it: add a header, note who the caller is
  - stop it and write the response itself: a 401, a redirect
- Filters are a standard part of the Java web stack; the controller never sees a request a filter stopped
- `OncePerRequestFilter` is Spring's base class for a filter that runs exactly once per request

Spring Security is a filter chain

```
request ⟶ [ CORS ] ⟶ [ CSRF ] ⟶ [ login filters, if configured ] ⟶ [ exception translation ] ⟶ [ authorization ] ⟶ controller
```

- One servlet filter at the front, which runs Spring Security's own list of filters: the **SecurityFilterChain**
- The `SecurityFilterChain` bean we write is that list: every line of the configuration turns a filter on, off, or sets what it does
  - `csrf.disable()` removes the CSRF filter; `httpBasic(...)` or `formLogin(...)` would add a login filter, and a chain of your own has none unless it asks
  - `addFilterBefore` puts our own filter in
- The authorization filter is the last one before the controller: it compares the caller with the rules
- The exception translation filter turns "no caller" into the entry point's 401 and "a caller without the right" into 403

The security context

- The **SecurityContext** is where "who is the caller" lives for the length of one request; `SecurityContextHolder` gives access to it from anywhere in that request
- It holds an **Authentication**: the principal (who), the credentials (the proof, dropped once checked), the authorities (the roles)
- Empty at the start of every request: nobody is logged in until a filter says so
- Our `JwtAuthFilter` fills it from the token; the authorization filter reads it; a controller's `Principal` is the principal from it
- Cleared at the end of the request. A token proves the caller once per request; nothing is remembered between requests.

What the sample adds

- One filter of our own, `JwtAuthFilter`: reads the `Authorization` header, checks the token, and puts the caller into the security context. Nothing else in the chain changes.
- The chain then does its usual work: the authorization filter compares that caller with our rules; the exception translation filter answers 401 or 403

```
request ⟶ [ JwtAuthFilter: token → security context ] ⟶ [ authorization: context against the rules ] ⟶ controller
```

- What we write, package `ee.taltech.staff.auth`:
  - `SecurityConfig`: the `SecurityFilterChain` bean, the rules, our filter added to the chain
  - `JwtAuthFilter`: the filter
  - `JwtService`: makes and checks tokens
  - the users table: the entity `AppUser`, its repository, and a Liquibase changeset that creates the table `app_user`
  - `AuthService`, `AuthController`: register, login, `/api/me`

The users table

```java
@Entity
@Getter @Setter @NoArgsConstructor
public class AppUser {

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

    private String username;

    private String passwordHash;

    private String role;
}
```

```xml
<changeSet id="006-app-user" author="siim">
  <createTable tableName="app_user">
    <column name="id" type="bigint" autoIncrement="true">
      <constraints primaryKey="true" nullable="false"/>
    </column>
    <column name="username" type="text">
      <constraints nullable="false" unique="true"/>
    </column>
    <column name="password_hash" type="text">
      <constraints nullable="false"/>
    </column>
    <column name="role" type="text">
      <constraints nullable="false"/>
    </column>
  </createTable>
</changeSet>
```

- An entity like any other, a changeset like any other
- The table is `app_user`: `user` is a reserved word in PostgreSQL. A table named `user` works, but every SQL statement has to write it as `"user"`, with the quotes; the name `app_user` avoids that.
- The column holds the bcrypt string, and its name says so

The repository

```
public interface AppUserRepository extends JpaRepository<AppUser, Long> {

    Optional<AppUser> findByUsername(String username);
}
```

- One derived query: the login looks a user up by name
- `Optional`, because the name may not exist; the service decides what that means: a 401, worded exactly like a wrong password

`JwtService` makes a token

```
@Service
public class JwtService {

    private final SecretKey key;
    private final Duration expiry;

    public JwtService(@Value("${app.jwt.secret}") String secret,
                      @Value("${app.jwt.expiry-minutes}") long expiryMinutes) {
        this.key = Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
        this.expiry = Duration.ofMinutes(expiryMinutes);
    }

    public String generate(AppUser user) {
        Instant now = Instant.now();
        return Jwts.builder()
                .subject(user.getUsername())
                .claim("userId", user.getId())
                .claim("role", user.getRole())
                .issuedAt(Date.from(now))
                .expiration(Date.from(now.plus(expiry)))
                .signWith(key)
                .compact();
    }
```

- `subject` : the username; `claim` : the application's own claims, here the id and the role
- `signWith(key)` : the HMAC with the secret; `compact()` : the `header.payload.signature` string

`JwtService` checks a token

```
    /** Throws JwtException when the signature is wrong or the token has expired. */
    public Claims parse(String token) {
        return Jwts.parser()
                .verifyWith(key)
                .build()
                .parseSignedClaims(token)
                .getPayload();
    }
}
```

- `parseSignedClaims`: recomputes the signature with the key, compares, checks `exp` against the clock, and only then returns the claims
- A bad signature and an old token are both a `JwtException`; the caller of `parse` treats both the same way: not logged in

The secret

- `HS256` needs a secret of at least 32 bytes, the same on every instance of the backend, known to nobody else
- It is a credential, handled like the database password:

```
app.jwt.secret=${JWT_SECRET:local-development-secret-at-least-32-bytes-long}
app.jwt.expiry-minutes=60
```

- on the laptop: the default after the colon; nobody attacks localhost
- in GitLab: a masked CI/CD variable `JWT_SECRET`, next to `DB_PASSWORD`
- in the deploy job: one more line into the server's `.env`: `echo JWT_SECRET=$JWT_SECRET >> .env`
- in `compose.yaml`: `JWT_SECRET: ${JWT_SECRET}` in the backend's `environment`
- A changed secret logs everyone out: every existing token fails its signature check. A secret generated at startup does the same on every restart.

The filter

```
@Component
@RequiredArgsConstructor
@Slf4j
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtService jwtService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {
        String header = request.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            try {
                Claims claims = jwtService.parse(header.substring(7));
                var authentication = new UsernamePasswordAuthenticationToken(
                        claims.getSubject(), null,
                        List.of(new SimpleGrantedAuthority("ROLE_" + claims.get("role", String.class))));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            } catch (JwtException e) {
                log.warn("rejected token: {}", e.getMessage());
            }
        }
        chain.doFilter(request, response);
    }
}
```

The filter, line by line

Line	What it does
`OncePerRequestFilter`	a filter that Spring runs exactly once per request, even when the request is forwarded inside the server
`Authorization: Bearer <token>`	the header the frontend sends; **Bearer**: whoever holds this is the caller
`header.substring(7)`	the token after `Bearer`
`jwtService.parse(...)`	signature and expiry checked here
`UsernamePasswordAuthenticationToken(subject, null, authorities)`	Spring Security's "the caller is X with these rights"; the password slot is `null`, the token already proved it
`"ROLE_" + role`	Spring's convention: a role is an authority named `ROLE_…`; `hasRole("ADMIN")` looks for `ROLE_ADMIN`
`SecurityContextHolder.getContext().setAuthentication(...)`	from here on, this request has a caller
`catch (JwtException)`	a bad token sets nobody; the request continues as anonymous and the rules decide
`chain.doFilter`	on to the next filter, always

- The filter never answers 401 itself. It only says who the caller is, or says nothing.

The configuration

```
@Configuration
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthFilter jwtAuthFilter;

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

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/login", "/api/register", "/api/version").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/**").authenticated()
                .requestMatchers(HttpMethod.DELETE, "/api/**").hasRole("ADMIN")
                .anyRequest().authenticated())
            .exceptionHandling(e -> e
                .authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED))
                .accessDeniedHandler((req, res, ex) -> res.setStatus(HttpStatus.FORBIDDEN.value())))
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

Configuration of session, login and the 401

Line	Why
`csrf.disable()`	**CSRF**, cross-site request forgery: another site makes the browser send a request that carries the victim's cookie. The protection is a per-session token in forms. This API has no session and no cookie; the proof is a header another site cannot set, so there is nothing to protect
`STATELESS`	no session is ever created; no `JSESSIONID` cookie
`HttpStatusEntryPoint(UNAUTHORIZED)`	the answer when a login is required and there is none: a bare 401. Without it Spring Security answers 403 for everything
`accessDeniedHandler(...)`	the answer when the caller is known but the rule says no: a bare 403. Both answers come from the filter chain, before any controller, so the exception handler of lecture 7 is not involved
`addFilterBefore`	the JWT filter runs before the filter that would handle a form login

Configuration of the rules

Line	Why
`permitAll()`	the three URLs anyone may call: login, register, version
`GET /api/**` → `authenticated()`	any logged-in user may read
`DELETE /api/**` → `hasRole("ADMIN")`	only an administrator may delete
`anyRequest().authenticated()`	everything else needs a login

- The rules are read **top to bottom, the first match wins**
- `anyRequest()` is last because it matches everything; a `permitAll()` after it would never be reached
- A rule names a URL pattern and, optionally, the HTTP method; `/api/**` is every path under `/api/`
- The same path can have different rules per method, as `GET` and `DELETE` here
- A convention that keeps the list short: every endpoint that needs no login under one prefix, `/api/public/**` → `permitAll()`, everything else `authenticated()`. The URL says the rule, and a new endpoint is protected unless it is put under `public` on purpose. The same split works for the frontend's pages.

Registering

```
@Transactional
public UserDto register(RegisterDto dto) {
    if (userRepository.findByUsername(dto.username()).isPresent()) {
        throw new ResponseStatusException(HttpStatus.CONFLICT, "username taken");
    }
    AppUser user = new AppUser();
    user.setUsername(dto.username());
    user.setPasswordHash(passwordEncoder.encode(dto.password()));
    user.setRole(userRepository.count() == 0 ? "ADMIN" : "USER");
    user = userRepository.save(user);
    return new UserDto(user.getId(), user.getUsername(), user.getRole());
}
```

- The password goes through `encode` and nowhere else; the DTO that comes back has no password field
- The first user of an empty table becomes `ADMIN`, so that an administrator exists without a hand-written insert; every later user is `USER`

```
$ curl -X POST localhost:8088/api/register -d '{"username":"siim","password":"correct horse"}'
HTTP/1.1 201    {"id":1,"username":"siim","role":"ADMIN"}
$ curl -X POST localhost:8088/api/register -d '{"username":"mari","password":"correct horse"}'
HTTP/1.1 201    {"id":2,"username":"mari","role":"USER"}
```

Logging in, in the sample

```java
@Transactional(readOnly = true)
public TokenDto login(LoginDto dto) {
    AppUser user = userRepository.findByUsername(dto.username())
            .filter(u -> passwordEncoder.matches(dto.password(), u.getPasswordHash()))
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.UNAUTHORIZED, "wrong username or password"));
    log.info("login: {}", user.getUsername());
    return new TokenDto(jwtService.generate(user));
}
```

- An unknown name and a wrong password fall into the same `orElseThrow`: the same 401, the same message. The body is lecture 7's `ApiError`, from the global exception handler the sample has.
- The log line names the user and never the password

```
$ curl -X POST localhost:8088/api/login -d '{"username":"siim","password":"correct horse"}'
HTTP/1.1 200    {"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzaWltIiwidXNlcklkIjoxLCJyb2xlIjoiQURNSU4i…"}

$ curl -X POST localhost:8088/api/login -d '{"username":"siim","password":"wrong"}'
HTTP/1.1 401    {"status":401,"message":"wrong username or password","path":"/api/login","timestamp":"2026-10-22T09:19:57.497Z"}
```

A protected endpoint

- The employee endpoints did not change. The rules in front of them did.

```
$ curl -i localhost:8088/api/employees
HTTP/1.1 401

$ curl -i localhost:8088/api/employees -H "Authorization: Bearer $TOKEN"
HTTP/1.1 200
[{"id":1,"name":"John Smith","email":"john@example.com","departmentName":"Engineering"},
 {"id":2,"name":"Jane Doe","email":"jane@example.com","departmentName":"Engineering"}]
```

- Same URL, same controller, same service
- The only difference between the two requests is one header, and the filter turned it into a caller before the rules ran

The `Principal` names the caller

- A controller that needs to know the caller asks for a `Principal`, Spring's name for "the authenticated party"
- Its `getName()` is the `sub` claim the filter set:

```
@GetMapping("/me")
public UserDto me(Principal principal) {
    return authService.me(principal.getName());
}
```

```
$ curl localhost:8088/api/me -H "Authorization: Bearer $TOKEN"
{"id":1,"username":"siim","role":"ADMIN"}
```

- This is how "my orders" or "my profile" knows whose data to return: from the token, never from a `userId` in the request
- A `userId` in the request is the caller's claim about themselves; the token is the server's

Roles

- Authorization in the sample is one rule: deleting needs the role `ADMIN`. `mari` is a `USER`, `siim` is an `ADMIN`:

```
$ curl -i -X DELETE localhost:8088/api/employees/2 -H "Authorization: Bearer $MARI"
HTTP/1.1 403

$ curl -i -X DELETE localhost:8088/api/employees/2 -H "Authorization: Bearer $TOKEN"
HTTP/1.1 204
```

- Both callers are authenticated; one is not authorized. That is 401 against 403 in practice. The 403 has no body: it comes from the filter chain, not from a controller.
- The role came from the token's `role` claim, which came from the users table at login
- A role changed in the table reaches the user at their next login, not before: the token is a snapshot

Rules on the URL, or on the method

- The sample puts the rules on URLs in `SecurityConfig`. The other way is an annotation on the service method:

```
@PreAuthorize("hasRole('ADMIN')")
@Transactional
public void delete(Long id) { ... }
```

- enabled with `@EnableMethodSecurity` on the configuration class

Rules on URLs	Rules on methods
all in one place	next to the code they protect
by URL pattern and HTTP method	by any expression, including the arguments
enough when the rule is "this endpoint needs this role"	needed when the rule is "this user may edit **their own** record"

- For the course project the URL rules are usually enough
- "Own record" rules compare the `Principal` with the data inside the service

A test for the rules

- Security rules are code and get tests like any other
- `spring-security-test` adds `user(...)` to MockMvc: a caller without any token

```
@SpringBootTest
@AutoConfigureMockMvc
class AuthTest {

    @Autowired
    private MockMvc mockMvc;

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

    @Test
    void versionIsPublic() throws Exception {
        mockMvc.perform(get("/api/version")).andExpect(status().isOk());
    }

    @Test
    void anAuthenticatedUserSeesEmployees() throws Exception {
        mockMvc.perform(get("/api/employees").with(user("siim").roles("USER"))).andExpect(status().isOk());
    }
}
```

```
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0 -- in ee.taltech.staff.auth.AuthTest
[INFO] BUILD SUCCESS
```

Where the frontend keeps the token

- The browser has to keep the token between requests and page loads. Two places, with opposite weaknesses:

	`localStorage`	cookie with `HttpOnly`
who can read it	any JavaScript on the page	no JavaScript; the browser only
sent with requests	by the frontend code, as a header	automatically, to that site, by the browser
weak against	XSS	CSRF

- XSS, cross-site scripting: an attacker gets their JavaScript to run on your page, for instance through a comment rendered as HTML; that script reads `localStorage` and sends the token away
- CSRF, cross-site request forgery: another site makes the browser send a request to yours, and the browser attaches your cookie; the attacker never sees the cookie, the request still happens as you

Where the sample keeps the token

- The sample: `localStorage`, sent in the `Authorization` header
  - the server needs no cookie handling and no CSRF protection
  - Vue escapes everything it renders with `{{ }}`: the main defence against XSS. The one thing that undoes it is `v-html` on user input.
- The other valid choice: a cookie the server sets at login, with three flags
  - `HttpOnly`: JavaScript cannot read it
  - `Secure`: sent over HTTPS only
  - `SameSite=Strict`: not sent on requests another site started; the CSRF protection
- Either way the token is in the browser, and a token in the browser is only as safe as the page it is on

Sending the token on every request

- An axios **interceptor** runs before every request the frontend makes. One of them adds the header:

```
// src/services/api.js
const api = axios.create({ baseURL: '/api' })

api.interceptors.request.use((config) => {
  const token = localStorage.getItem('token')
  if (token) {
    config.headers.Authorization = `Bearer ${token}`
  }
  return config
})
```

- No view has to know about tokens
- `api.get('/employees')` in `EmployeesView.vue` is unchanged from lecture 6, and now carries the header

When the server says 401

- A second interceptor looks at every response
- A 401 means: no token, a token the server rejected, or an expired one. In all three cases the only fix is a new login:

```
api.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      localStorage.removeItem('token')
      router.push('/login')
    }
    return Promise.reject(error)
  },
)
```

- The expiry is handled here, once, for the whole frontend: an hour after login the next request fails, the user is sent to the login form, and after it the request works again

The login call

```
// src/services/auth.js
export async function login(username, password) {
  const response = await api.post('/login', { username, password })
  localStorage.setItem('token', response.data.token)
}

export function logout()     { localStorage.removeItem('token') }
export function isLoggedIn() { return localStorage.getItem('token') !== null }
```

- `login` is the one place the token is written
- The response of `/api/login` is `{"token": "…"}`; the string goes into `localStorage` under one key that the interceptor and `isLoggedIn` read

The login view

```
<!-- src/views/LoginView.vue -->
<form @submit.prevent="submit">
  <input v-model="username" placeholder="username" autocomplete="username" />
  <input v-model="password" type="password" placeholder="password" autocomplete="current-password" />
  <button type="submit">Log in</button>
</form>
<p v-if="error">{{ error }}</p>
```

```
async function submit() {
  try {
    await login(username.value, password.value)
    router.push('/employees')
  } catch (err) {
    error.value = err.response?.status === 401 ? 'Wrong username or password' : err.message
  }
}
```

- `type="password"` hides the input; the `autocomplete` values let the browser's password manager fill the form

Pages that need a login

- The router can refuse to show a page to a visitor without a token
- A convenience for the user, not a security measure: the data is protected by the backend; the page only avoids showing an empty list and a 401

```
const router = createRouter({
  routes: [
    { path: '/login', name: 'login', component: LoginView },
    { path: '/employees', name: 'employees', component: EmployeesView, meta: { requiresLogin: true } },
    ...
  ],
})

router.beforeEach((to) => {
  if (to.meta.requiresLogin && !isLoggedIn()) {
    return { name: 'login' }
  }
})
```

- `beforeEach` runs before every navigation; returning a route sends the visitor there instead

Logging out

- With a token, logging out is something the **frontend** does: it forgets the token

```
function logOut() {
  logout()              // localStorage.removeItem('token')
  router.push('/login')
}
```

- The server is not involved and cannot be: it keeps no list of valid tokens, so there is nothing to delete
- The token itself stays valid until `exp`; someone who copied it before the logout can use it until then
- That is the price of a stateless token: hence the short expiry, and the refresh token with a database row when "log out everywhere" has to work

What goes over the wire

Item	Where it travels	Rule
the password	the login request body, once	never in a URL, never in a log, never in a response
the token	the `Authorization` header, every request	readable by whoever sees the request
the secret	nowhere	the server only, from an environment variable

- The first two rows are readable on the network unless the connection is **HTTPS**
- The server of lab 2 has the certificate from Certbot; the frontend calls `/api` on the same host, so every request to the backend goes through that nginx and that certificate
- A token sent over plain HTTP from a café network is a token someone else may have

What milestone 2 requires of authentication

- Milestone 2 asks for **working authentication**. In the terms of this lecture:
  - a users table defined in Liquibase, with bcrypt hashes and no plain passwords
  - a login endpoint that answers with a token, and a 401 for wrong credentials
  - the endpoints that read or change the team's data require the token; the frontend sends it
  - the pages of the team's application that show data need a login
- Registration is the team's choice: a `/register` endpoint, or users inserted by an administrator
- Roles are needed only if the application has actions that not every user may do; a single `USER` role with no `hasRole` rule is a working authentication too