

Authentication

Objectives

- Authentication vs. Authorization
- Describe in general JWT and how they are used to authenticate users
- Use a common tool to decode an encoded JWT to inspect its contents
- HTTP response status codes commonly associated with authentication and authorization failures
- Write client code in Java that can authenticate a user.
- Utilize the auth features of an application framework to:
 - Specify that a particular resource requires authentication to be accessed
 - Specify that a particular resource can be accessed anonymously
 - Apply simple authorization rules for resources
 - Obtain the identity of an authenticated user

Authentication vs. Authorization

- Authentication is validating the user is who they claim to be
 - Front gate to any secure web application
 - Based on authentication, authorization can be granted
- Authorization is giving user permission to access specific resources or functions.
 - Only authorized to access certain pages in web site
 - Only authorized to perform certain actions on a database

Forms of authentication

- Something the user *knows* (Knowledge factor)
 - Password, partial password, pass phrase, PIN, challenge response
- Something the user *has* (Ownership factor)
 - Wrist band, ID card, security token, cell phone with built-in hardware token, software token
- Something the user *is* (Inherence factor)
 - Fingerprint, retinal pattern, DNA sequence, signature, face, voice, unique bio-electric signals
- *Where the user is located (Location)*

2FA or MFA

2 Factor Authentication or Multi-Factor Authentication involves using a combination of these elements to authenticate:

- Codes generated by smartphone apps
- Badges, USB devices, or other physical devices
- Soft tokens, certificates
- Fingerprints
- Codes sent to an email address
- Facial recognition
- Retina or iris scanning
- Behavioral analysis
- Risk score
- Answers to personal security questions

Most web applications require only a username and password



Strong password policy example

- Password must meet at least 3 out of the following 4 complexity rules
 - at least 1 uppercase character (A-Z)
 - at least 1 lowercase character (a-z)
 - at least 1 digit (0-9)
 - at least 1 special character (punctuation) — do not forget to treat space as special characters too
- at least 10 characters
- at most 128 characters
- not more than 2 identical characters in a row (e.g., 111 not allowed)



1.3. Use of authentication protocols that require no password

While authentication through a user/password combination and using multi-factor authentication is considered generally secure, there are use cases where it isn't considered the best option or even safe. An example of this are third party applications that desire connecting to the web application, either from a mobile device, another website, desktop or other situations. When this happens, it is NOT considered safe to allow the third party application to store the user/password combo, since then it extends the attack surface into their hands, where it isn't in your control. For this, and other use cases, there are several authentication protocols that can protect you from exposing your users' data to attackers.

1.3.1. OAuth

Open Authorization (OAuth) is a protocol that allows an application to authenticate against a server as a user, without requiring passwords or any third party server that acts as an identity provider. It uses a token generated by the server, and provides how the authorization flows most occur, so that a client, such as a mobile application, can tell the server what user is using the service.

The recommendation is to use and implement OAuth 1.0a or OAuth 2.0, since the very first version (OAuth1.0) has been found to be vulnerable to session fixation.

- https://owasp.org/www-pdf-archive/OWASP_Cheatsheets_Book.pdf

HTTP is stateless!

- Does not keep track of state between connections
 - Each request is executed independently, without knowledge of the requests executed before
 - Once transaction ends, connection between browser and server is lost
 - Log in, but how to keep track that you are authorized??
- Each request must contain information about user's identity

JWT – JSON Web Token

- String passed in header or url while making a network request to pass data safely
 - Separated by three dots
 - header.payload.signature
 - Header provides information about token
 - Payload is actual data of token (also called claims)
 - Signature is verification that data hasn't been tampered with

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MzkwMjQ0fQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

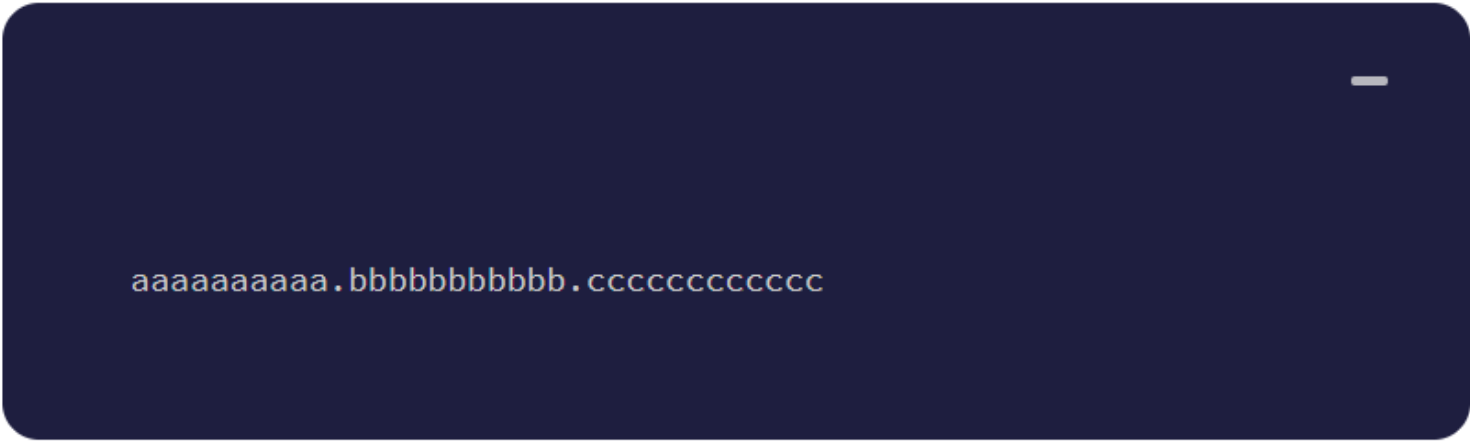
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Authentication with JWT

- Token is created and returned to client in response
 - Token must be supplied for every API request that requires authorization
 - Token goes in http header with word bearer, which indicates type of authentication



aaaaaaaaa.bbbbbbbbbb.cccccccccc

Header

- Carries two parts

```
{  
  "typ": "JWT",  
  "alg": "HS256"  
}
```

Claims

- Statements about an entity
- Registered claims (predefined keys which are not mandatory but recommended)
- Public claims (claims we recreate, like user name, information, or other important data -- required to be collision resistant)
- Private claims (custom key value pairs created to share information between parties – neither registered or public)

```
PAYLOAD:
{
  "sub": "23",
  "name": "dave",
  "role": "admin",
  "iat": 1590780570,
  "exp": 1591385370
}
```

Payload example

- Contains
 - two registered claims (iss and exp)
 - two public claims (name and admin)

```
{  
  "iss": "scotch.io",  
  "exp": 1300819380,  
  "name": "Chris Sevilleja",  
  "admin": true  
}
```

```
eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOiJzY290Y2guaW8iLCJpcyBT
```

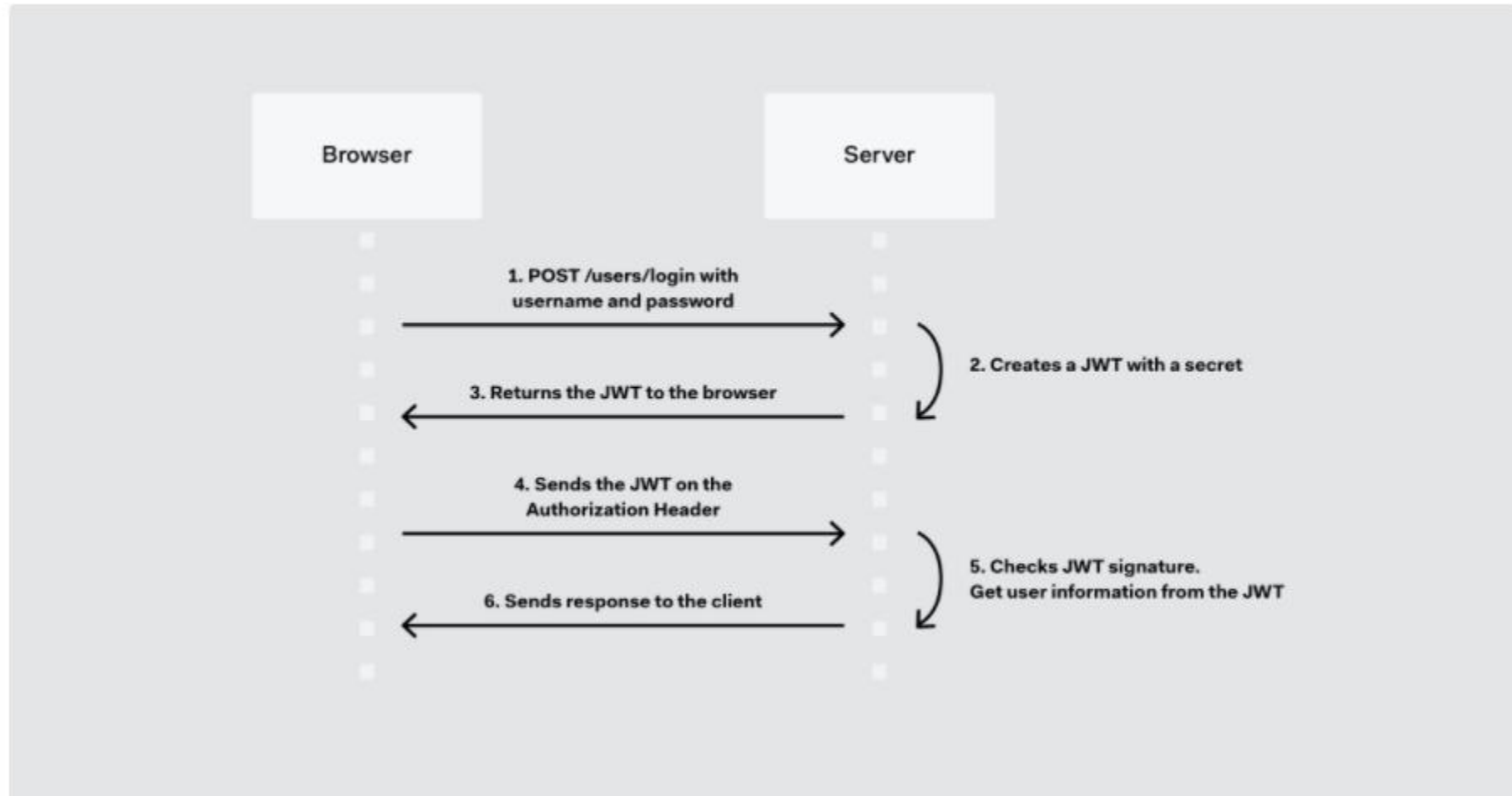
Signature

- Takes and signs
 - Encoded header
 - Encoded payload
 - Secret

```
HMACHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret)
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezDI1AVTmud2fU4
```


How JWT works



Using JWTs in a client app

Instead of `restTemplate.getForObject`, we use

`restTemplate.exchange`

RestTemplate exchange method needs 4 parameters

- String URL (“http://localhost:8080/login”)
- Http Method (HttpMethod.POST)
- HttpEntity object (entity) – contains headers and payload (can be JSON)
- Class response type (Map.class)

Client side code

```
public ResponseEntity<Map> login(String credentials) throws
    AuthenticationServiceException {

    LoginDTO loginDTO = new LoginDTO(credentials);    // object that holds user
                                                         // name and password

    HttpHeaders headers = new HttpHeaders();

    headers.setContentType(MediaType.APPLICATION_JSON);

    HttpEntity<LoginDTO> entity = new HttpEntity<>(loginDTO, headers);

    ResponseEntity<Map> response = null;

    try {
        response = restTemplate.exchange(BASE_URL + "/login",
            HttpMethod.POST, entity, Map.class);
    } catch (RestClientResponseException ex) {
```

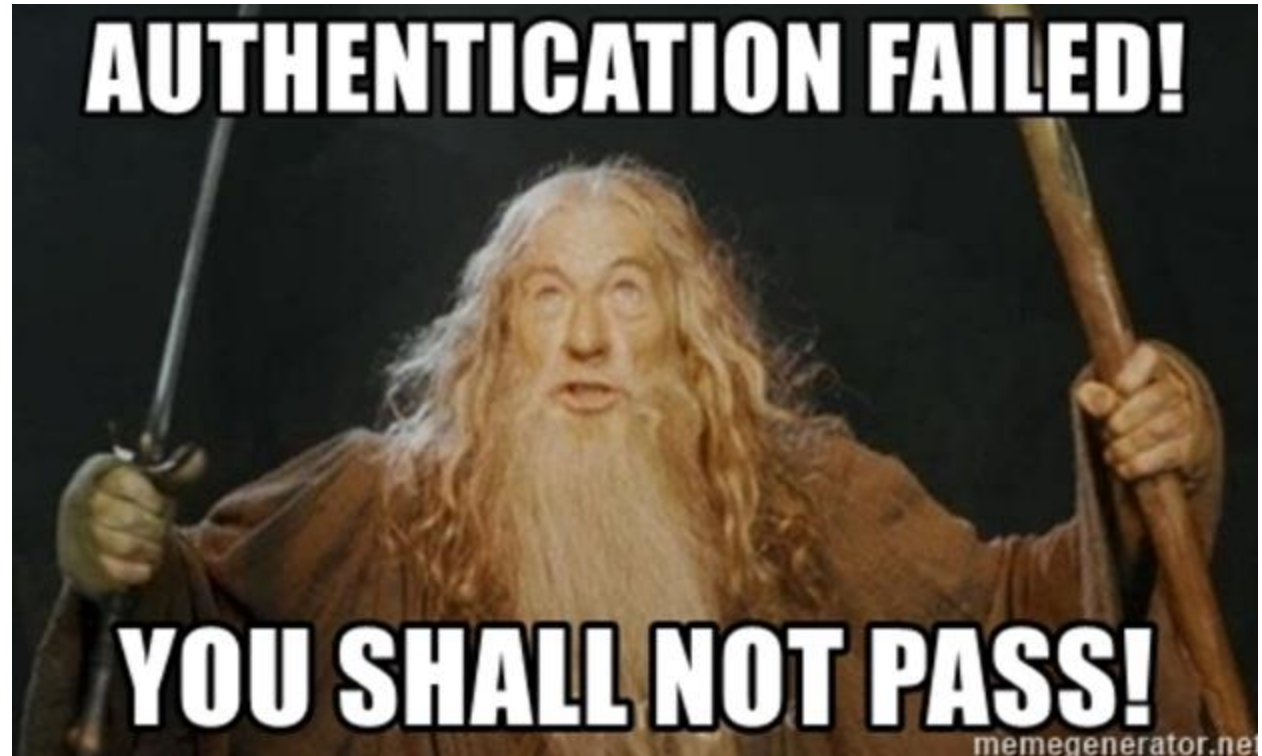
To use Token in Request

Pass the token in the bearer authentication header before making request:

```
Location location = null;
try {
    // Authorization: Bearer {AUTH_TOKEN} Header
    HttpHeaders headers = new HttpHeaders();
    headers.setBearerAuth(AUTH_TOKEN);
    HttpEntity <Location> entity = new HttpEntity<>(headers);
    location = restTemplate.exchange(BASE_URL + "/" + id,
        HttpMethod.GET, entity, Location.class).getBody();
} catch (RestClientResponseException ex) {
    throw new LocationServiceException(ex.getRawStatusCode() +
        " : " + ex.getResponseBodyAsString());
}
return location;
```

HTTP response codes

- 401 – Unauthorized
- 403 - Forbidden



Securing API methods

- @PreAuthorize annotation
 - Class level or method level in controller
 - Spring Expression Language argument
 - `@PreAuthorize("isAuthenticated()")` : The user must be authenticated.
 - `@PreAuthorize("permitAll")` : The user doesn't have to be authenticated.
 - `@PreAuthorize("hasRole('ADMIN')")` : The user must be authenticated and have the role ADMIN.
 - `@PreAuthorize("hasAnyRole('ADMIN', 'USER')")` : The user must be authenticated and have either the ADMIN or USER role.

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@Override
public void addEmployee(EmployeeEntity employee) {
    //System.out.println(((User)SecurityContextHolder.getContext().getAuthentication().getPrincipal()).getAuthorities());
    this.sessionFactory.getCurrentSession().save(employee);
}
```

User Identity

Certain situations require you to know the identity of the user. You can add a new argument `Principal` and Spring will resolve this for you.

```
@ResponseStatus(HttpStatus.CREATED)
@RequestMapping(value = "", method = RequestMethod.POST)
public Location add(@Valid @RequestBody Location location, Principal principal) {
    System.out.println(principal.getName());
    return dao.create(location);
}
```

Let's Code!