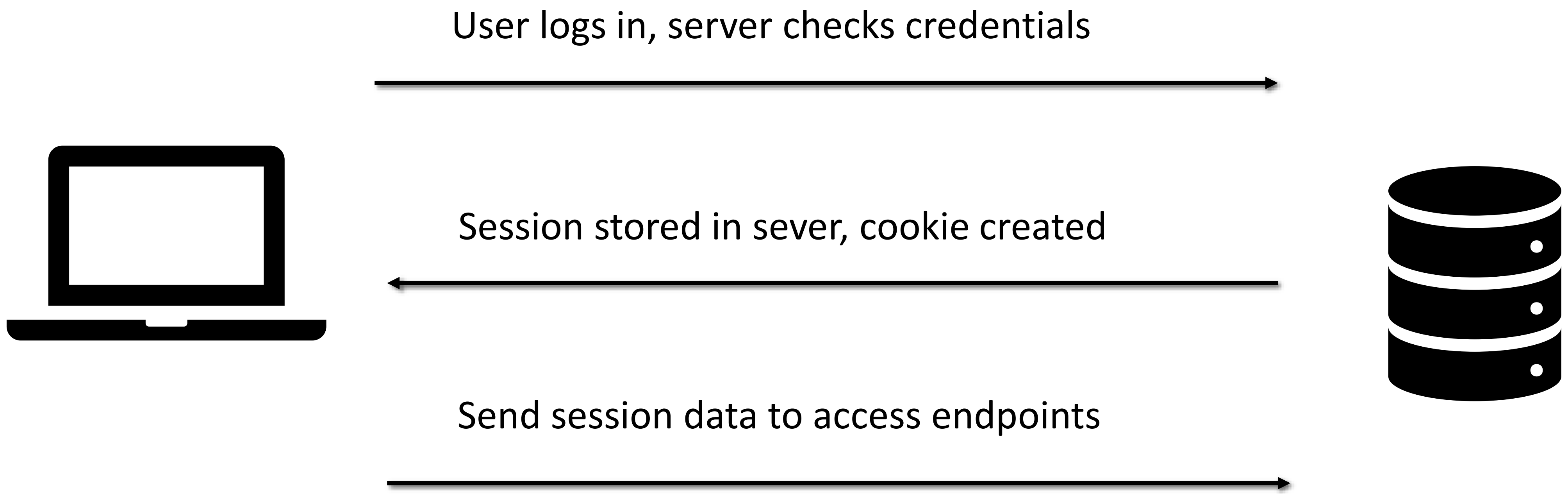
Spring Security 6 - JWT

MISS XING

Traditional Authentication System


User logs in, server checks credentials
Session stored in sever, cookie created
Send session data to access endpoints

Issues with Traditional Systems

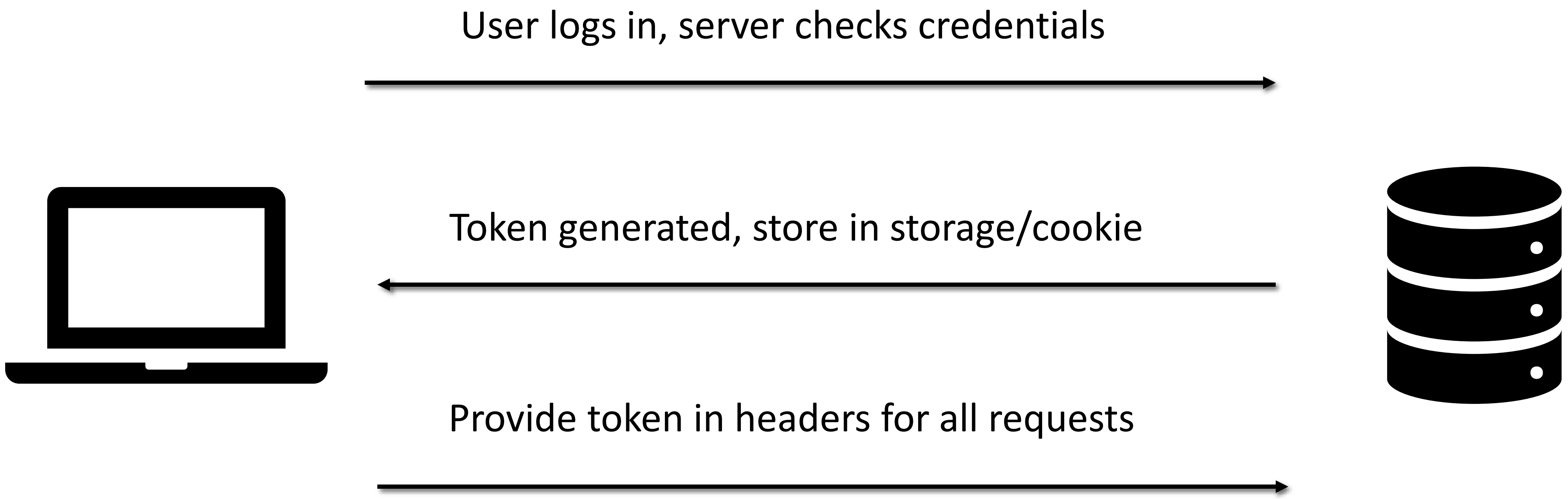
Sessions: Record needs to be stored on server.

Scalability: With sessions in memory, load increases drastically in a distributed system.

CORS: When using multiple devices grabbing data via AJAX requests, may run into forbidden requests.

CSRF: Riding session data to send requests to server from a browser that is trusted via session.

Token-Based Authentication Systems


User logs in, server checks credentials
Token generated, store in storage/cookie
Provide token in headers for all requests

Token-Based Authentication System

Stateless: self contained

Scalability: no need to store session in memory

CSRF: no session being used

Digitally-signed

Mobile-ready

Decoupled

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

This information can be verified and trusted because it is digitally signed.

Compact: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.

- Simply a string in the format of **header.payload.signature**

Self-contained: The payload contains all the required information about the user, avoiding the need to query the database more than once.

JSON Web Token Structure

JSON Web Tokens consist of three parts separated by dots (.), which are:

- header
- payload
- signature

Therefore, a JWT typically looks like the following:

- xxxxx.yyyyy.zzzzz
- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.

For example:

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

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT Payload

The second part of the token is the payload, which contains the claims.

Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims:

- *reserved*
  - The JWT specification defines seven reserved claims that are not required, but are recommended to allow interoperability with third-party applications.
- *Public*
  - These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
- *Private*
  - These are the custom claims created to share information between parties that agree on using them.

JWT Payload

For example:

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

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

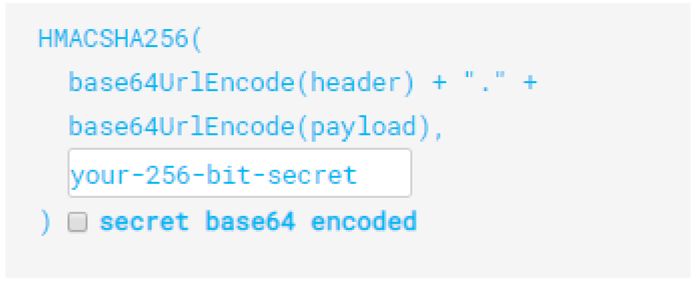
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaW
F0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

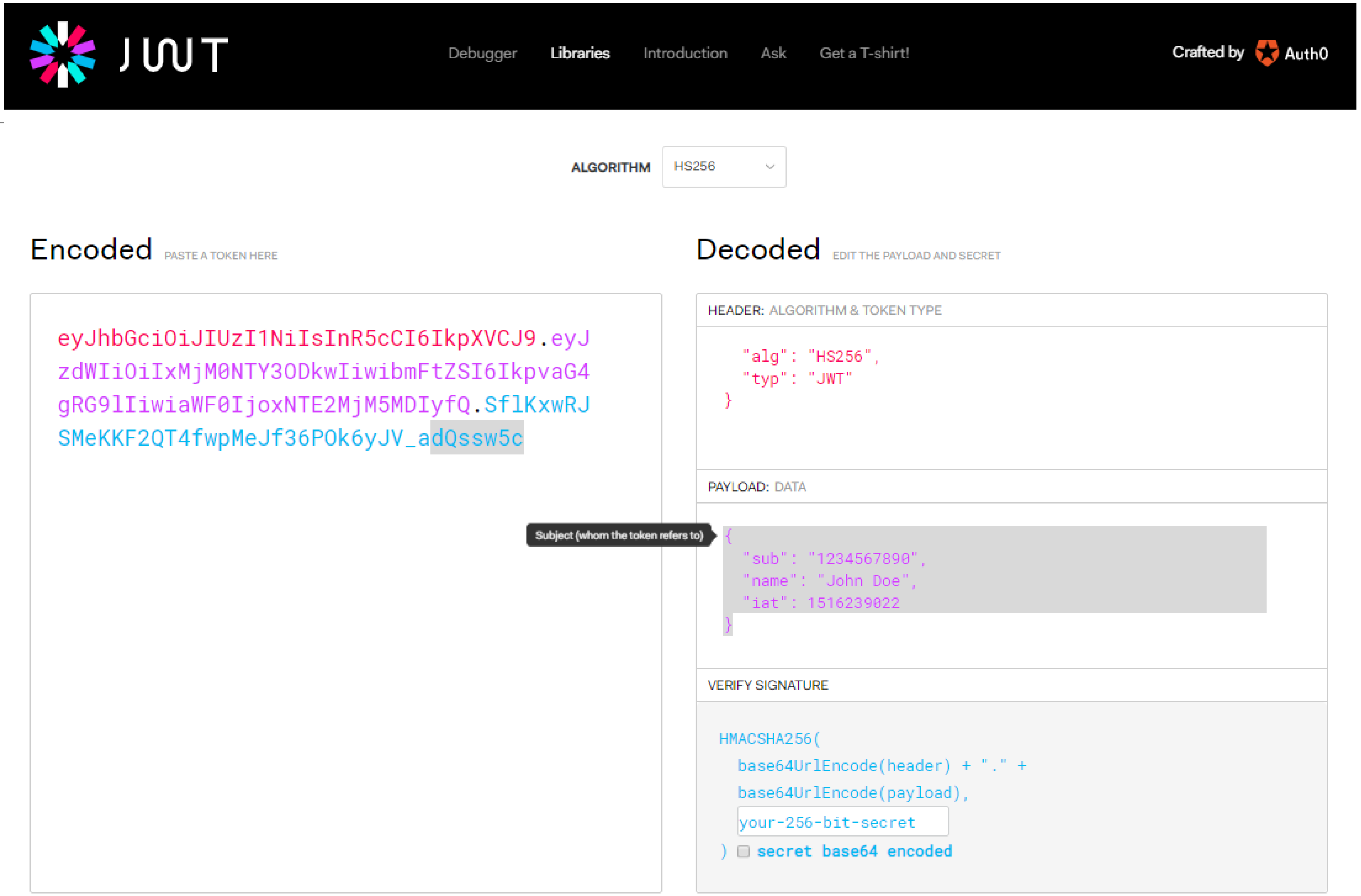


```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

jwt.io

JWT.IO allows you to decode, verify and generate JWT.



ALGORITHM HS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

Subject (whom the token refers to)

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

VERIFY SIGNATURE

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

How does JWT work?

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used).

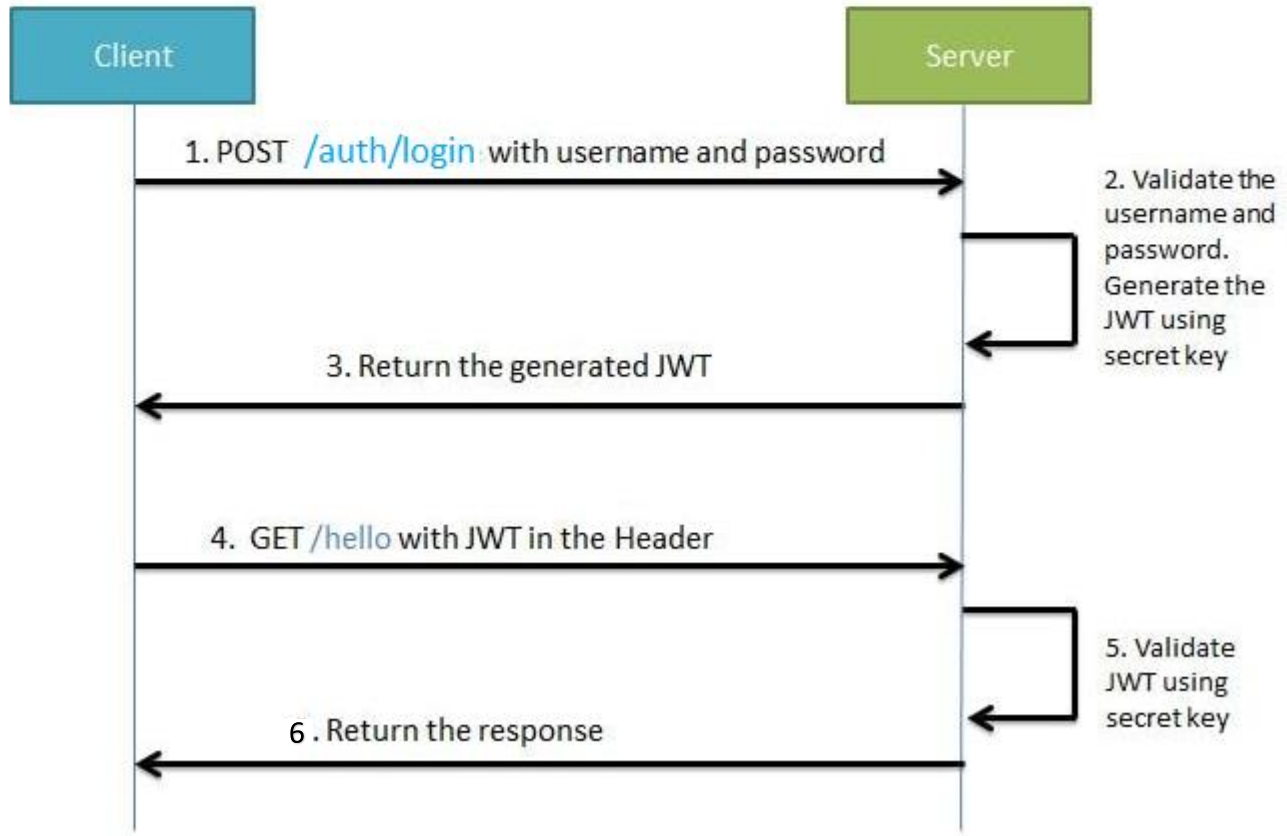
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

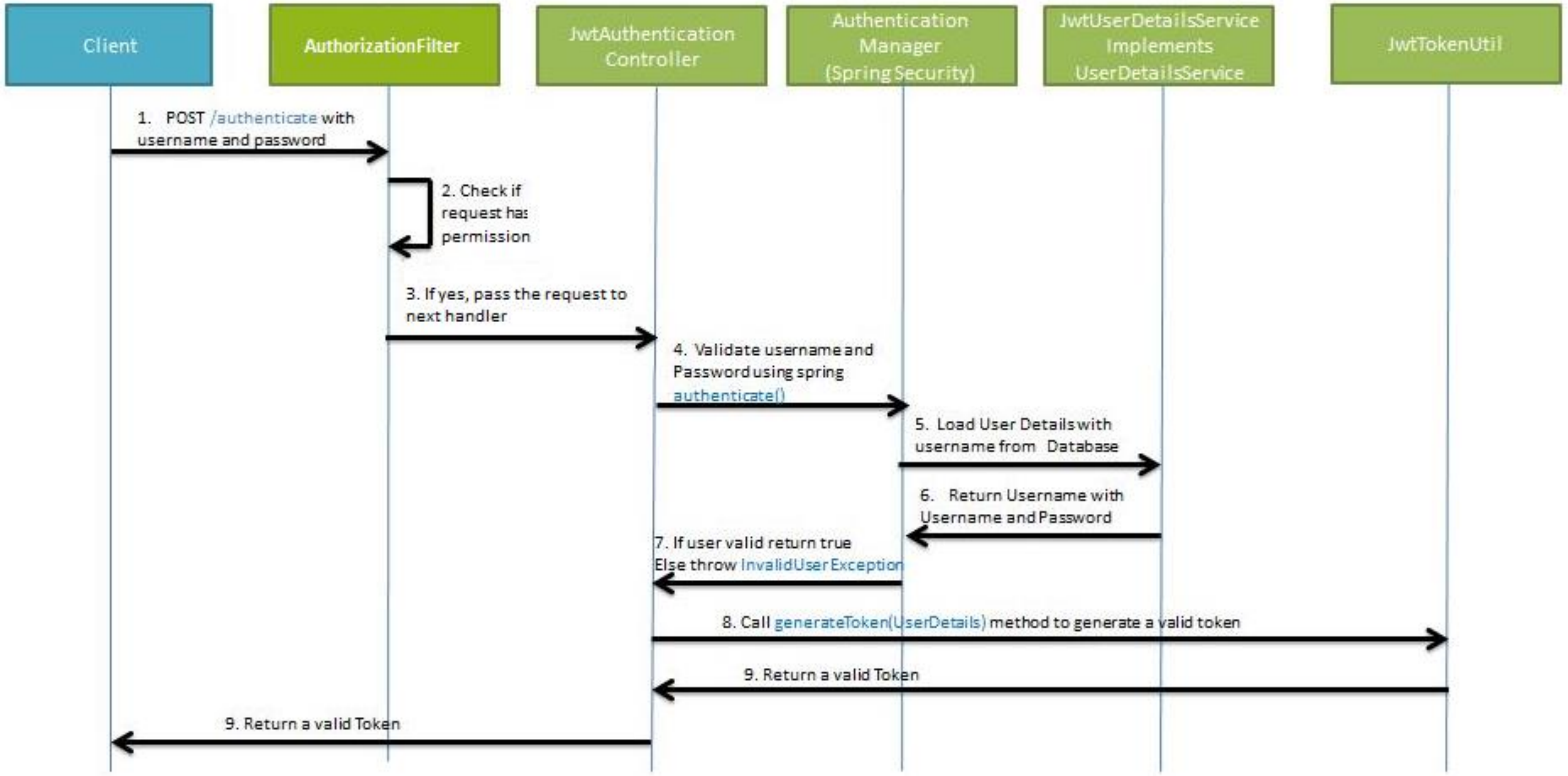
Authentication: Generating JWT

MISS XING

JWT Authentication Process


Client
Server
1. POST /auth/login with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /hello with JWT in the Header
5. Validate JWT using secret key
6 . Return the response

Generating JWT


Client
AuthorizationFilter
JwtAuthentication Controller
Authentication Manager (Spring Security)
JwtUserDetailsService Implements UserDetailsService
JwtTokenUtil
1. POST /authenticate with username and password
2. Check if request has permission
3. If yes, pass the request to next handler
4. Validate username and Password using spring authenticate()
5. Load User Details with username from Database
6. Return Username with Username and Password
7. If user valid return true Else throw InvalidUserException
8. Call generateToken(UserDetails) method to generate a valid token
9. Return a valid Token
9. Return a valid Token

JWT Demo – Project Setup

1. Create a Spring Security Project with CRUD operations for Product Domain Model.

2. Enable HttpBasic Authentication

3. Implement DaoAuthenticationProvider (see previous example)

JWT Demo – User Setup

3.1 User Domain Model

```
@Data
@Entity
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String email;
    private String password;
    private String role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role));
    }

    @Override
    public String getUsername() {
        return email;
    }

}
```

JWT Demo – JWT Authentication API UserDetailsService

3.2 Update UserRepository to have findByEmail method

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {
    Optional<User> findByEmail(String email);
}
```

3.3 Implement UserDetailsService as used in AuthenticationProvider

```
@Service
@RequiredArgsConstructor
public class JwtUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userRepository.findByEmail(username).orElseThrow(() -> new UsernameNotFoundException(username + " Not Found"));
    }
}
```

3.4 Spring Security Config

```
@RequiredArgsConstructor
@Configuration
public class SpringSecurityConfig {

    private final CustomUserDetailsService customUserDetailsService;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
                .httpBasic(Customizer.withDefaults())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers(HttpMethod.DELETE, "/products/**").hasAuthority("admin")
                        .requestMatchers("/products/**").authenticated()
                        .anyRequest().permitAll()
                )
                .csrf(csrf -> csrf.disable())
                .sessionManagement(manager -> manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .build();
    }

    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(customUserDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

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

}
```

JWT Demo - Testing

1. Test if HttpBasic Authentication works.

2. Test if Authorization works

JWT Demo – JWT Dependency

- JJWT (https://github.com/jwtk/jjwt) is a Java library providing end-to-end JSON Web Token creation and verification.
- Free, open source
- The primary operations in using JJWT involve building and parsing JWTs.

```xml
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
</dependency>
```

JWT Demo – JWT Token Utility Class

4. Implement generate token method

```java
@Component
public class JwtTokenUtil {
    private final static Logger logger = LoggerFactory.getLogger(JwtTokenUtil.class);

    @Value("${app.jwt.secret}")
    private String SECRET;

    @Value("${app.jwt.duration}")
    private long EXPIRATION;

    public String generateToken(User u) {
        return Jwts.builder()
                .setSubject(u.getId() + "," + u.getEmail() + "," + u.getRole())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION))
                .signWith(getSignKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    private Key getSignKey() {
        byte[] keyBytes = Decoders.BASE64.decode(SECRET);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

application.properties

```
app.jwt.secret=1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
# 1 hour = 60*60*1000
app.jwt.expire.duration=3600000
```

JWT Demo – JWT Authentication API
POJO classes for Auth Request and Response

5. A POJO class that represents an authentication request or response

```
@Data
public class AuthRequest {

    private String email;
    private String password;

}
```

```
@Data
public class AuthResponse {

    private String email;
    private String accessToken;

}
```

JWT Demo – JWT Authentication API Login

6. Add a Login method in Controller. It'll utilize AuthenticationManager authenticate method to authenticate. So we need to add a bean for AuthenticationManager in Cofiguration class.

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/auth")
public class AuthController {
    private final AuthenticationManager authenticationManager;
    private final JwtTokenUtil jwtTokenUtil;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthRequest request) {
        try {
            Authentication authentication = authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(request.getEmail(),
request.getPassword()));
            User user = (User) authentication.getPrincipal();
            String accessToken = jwtTokenUtil.generateAccessToken(user);
            AuthResponse response = new AuthResponse(user.getEmail(), accessToken);
            return ResponseEntity.ok(response);
        } catch (BadCredentialsException e) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
}
```

JWT Demo – JWT Authentication API AuthenticationManager

7. Config AuthenticationManager Bean in Spring Security Configuration file

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration) throws Exception {
    return configuration.getAuthenticationManager();
}
```

8. Update the authorization configuration to secure REST APIs and test.

```
@Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      return http
              ~~.httpBasic(Customizer.withDefaults())~~
              .authorizeHttpRequests(auth -> auth
                      .requestMatchers(HttpMethod.DELETE, "/products/**").hasAuthority("admin")
                      .requestMatchers("/products/**").authenticated()
                      .anyRequest().permitAll()
              )
              .csrf(csrf -> csrf.disable())
              .sessionManagement(manager -> manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
              .build();
  }
```

Spring Security 6: Validating JWT

MISS XING

JWT Demo – JWT Token Filter Class JwtTokenUtil – validateAccessToken

To access the secure REST APIs, the client must include an access token in the Authorization header of the request. So we need to insert our own filter in the middle of Spring Security filters chain, before the UsernameAndPasswordAuthenticationFilter, in order to check the Authorization header of each request.

9. Add validateAccessToken

```
public boolean validateAccessToken(String token){
    try {
        Jwts.parserBuilder().setSigningKey(SECRET).build().parseClaimsJws(token);
        return true;
    } catch (ExpiredJwtException e) {
        log.error("Expired JWT " +e.getMessage());
    } catch (MalformedJwtException e) {
        log.error("invalid JWT " + e.getMessage());
    } catch (Exception e){
        log.error(e.getMessage());
    }
    return false;
}
```

JWT Demo – JWT Token Filter Class JWTTokenFilter

- Step 1: Check if it has Authorization Bearer Header
- Step 2: Check if the token is valid
- Step 3: Set SecurityContext

- 10. Implement Step 1 and Step 2

```
@Component
@RequiredArgsConstructor
public class JwtRequestFilter extends OncePerRequestFilter {

    private final JwtTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
//        Step 1: Check if it has Authorization Bearer Header
        if (!hasAuthorizationBearer(request)) {
            filterChain.doFilter(request, response);
            return;
        }
//        Step 2: Check if the token is valid
        String token = getAccessToken(request);
        if(!jwtTokenUtil.validateAccessToken(token)){
            filterChain.doFilter(request, response);
            return;
        }

    }

}
```

Implement Step 1 and Step 2

```
private boolean hasAuthorizationToken(HttpServletRequest request) {
    String authorization = request.getHeader("Authorization");
    return authorization != null && authorization.startsWith("Bearer ");
}
```

```
private String getAuthorizationToken(HttpServletRequest request) {
    String authorization = request.getHeader("Authorization");
    return authorization.substring(7);
}
```

JWT Demo – JWT Token Filter Class JWTTokenFilter

Implement Step 3: Set SecurityContext

- In order to set SecurityContext, we need to set values to Authentication, we need to get the claims from the token to extract information there.

11. Implement parseClaims

```
public class JwtTokenUtil {

    public Claims parseClaims(String token){
        return Jwts.parserBuilder().setSigningKey(SECRET_KEY).build().parseClaimsJws(token).getBody();
    }

}
```

JWT Demo – JWT Token Filter Class JWTTokenFilter

12. Set AuthenticationContext

```
@RequiredArgsConstructor
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    private final JwtTokenUtil jwtTokenUtil;

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

        //…
        setAuthenticationContext(token, request);
        filterChain.doFilter(request, response);
    }

    private void setAuthenticationContext(String token, HttpServletRequest request) {
        UserDetails u = getUserDetails(token);
        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(u, null, u.getAuthorities());
        SecurityContext context = SecurityContextHolder.createEmptyContext();
        context.setAuthentication(auth);
        SecurityContextHolder.setContext(context);
    }

    private UserDetails getUserDetails(String token) {
        User u = new User();
        Claims claims = jwtTokenUtil.getClaimsFromToken(token);
        String[] subject = claims.getSubject().split(",");
        u.setId(Integer.parseInt(subject[0]));
        u.setEmail(subject[1]);
        u.setRole(subject[2]);
        return u;
    }
}
```

JWT Demo – Config JWTTokenFilter

Add the `JwtRequestFilter` before the Spring Security internal `UsernamePasswordAuthenticationFilter`. We're doing this because we need access to the user identity at this point to perform authentication/authorization, and its extraction happens inside the JWT token filter based on the provided JWT token.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
            .addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class)
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers(HttpMethod.DELETE, "/products/**").hasAuthority("admin")
                    .requestMatchers("/products/**").authenticated()
                    .anyRequest().permitAll()
            )
            .csrf(csrf -> csrf.disable())
            .sessionManagement(manager -> manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
}
```

Reference

https://docs.spring.io/spring-security/reference/servlet/architecture.html

https://www.codejava.net/frameworks/spring-boot/spring-security-jwt-authentication-tutorial

https://www.javainuse.com/spring/boot-jwt