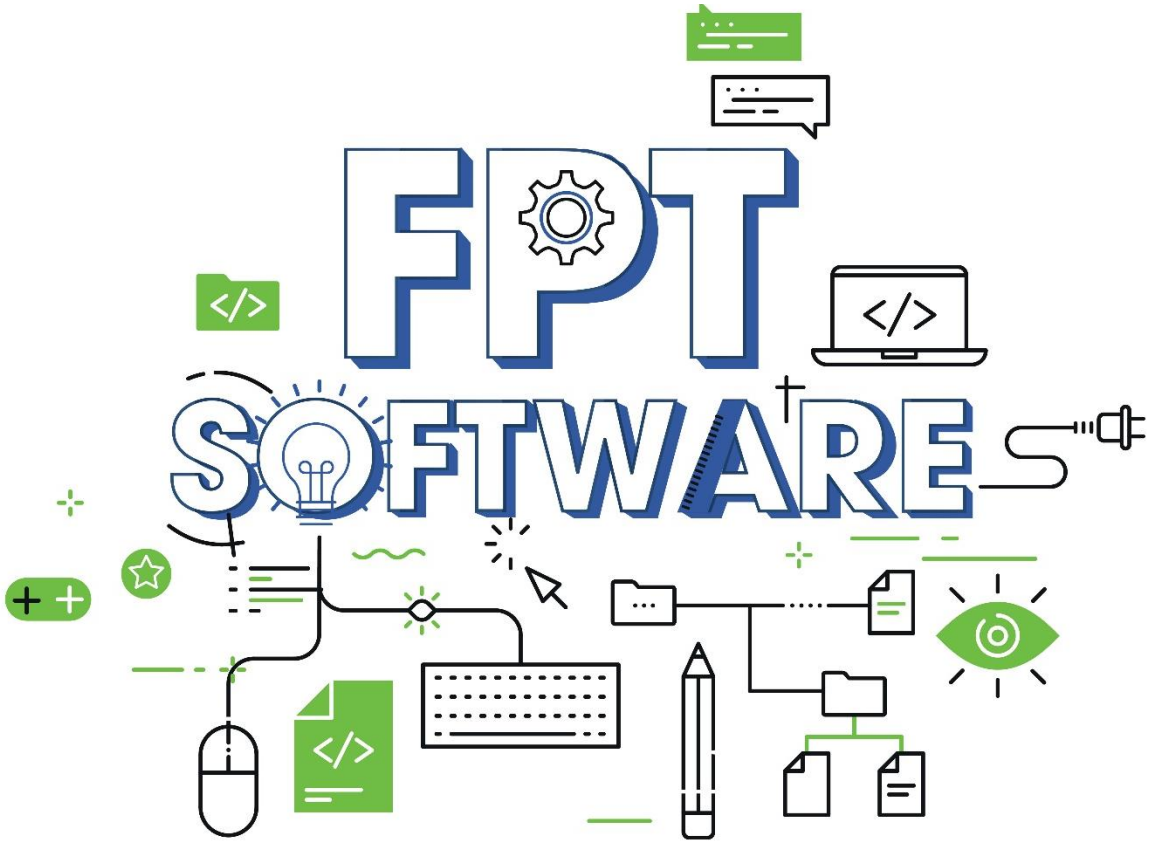
Spring Security for RESTful Webservice



Fsoft Academy

Agenda

1. Token Based Authentication
2. JSON Web Token
3. PasswordEncoders
4. Authentication & Authorization
5. AuditorAware with spring security

Lesson Objectives

1. Understand the fundamentals of Spring Security and its architecture in web applications.
2. Able to configure and utilize username/password authentication.
3. Able to use of password encoders to secure user passwords in Spring Security.
4. Know how to enable and use the Remember-Me feature in Spring Security to maintain user logins for an extended period.
5. Able to handle logouts in Spring Security and configure the logout functionality in a web application.

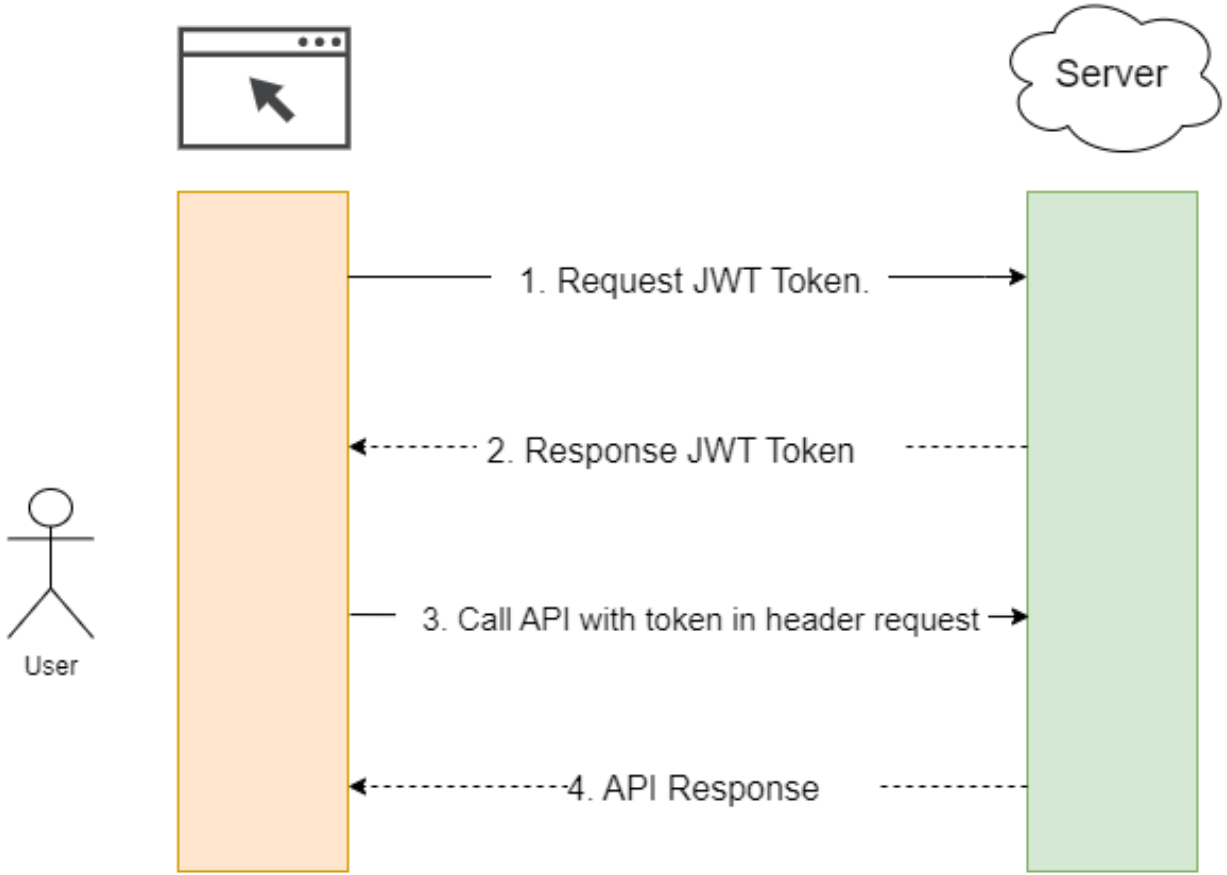
Section 1

Token Based Authentication

Disadvantages of session-based authentication

- ✓ **Server-Side Storage Requirement:** Sessions must be stored on the server, requiring resources and potentially adding strain to the system.
- ✓ **Scalability Challenges:** Scaling session management across different servers can be challenging, especially in a distributed environment, and may require complex synchronization.
- ✓ **Limited Suitability for RESTful APIs:** For applications based on RESTful APIs, maintaining session state may not be appropriate, as users access various resources from multiple devices and applications.

Token based authentication flow



- A Token can be a plain string of format UUID or it can be of type JSON Web Token (JWT)
- On every request to a restricted resource, the client sends the access token in the query string or Authorization header.

Advanced of Token based authentication

- ✓ Tokens can be invalidated during any suspicious activities without invalidating the user credentials.
- ✓ Tokens can be used to store the user related information like roles/authorities etc.
- ✓ **Reusability** – We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.
- ✓ **Stateless, easier to scale**. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to same server we logged in on.

Section 2

JSON Web Token

JWT

- JWT means JSON Web Token. It's a token implementation which will be in the JSON format and designed to use for the web requests.

- JWT is the most common and favorite token type that many systems use these days due to its special features and advantages

- JWT tokens can be used both in the scenarios of Authorization/Authentication

JWT - Structure

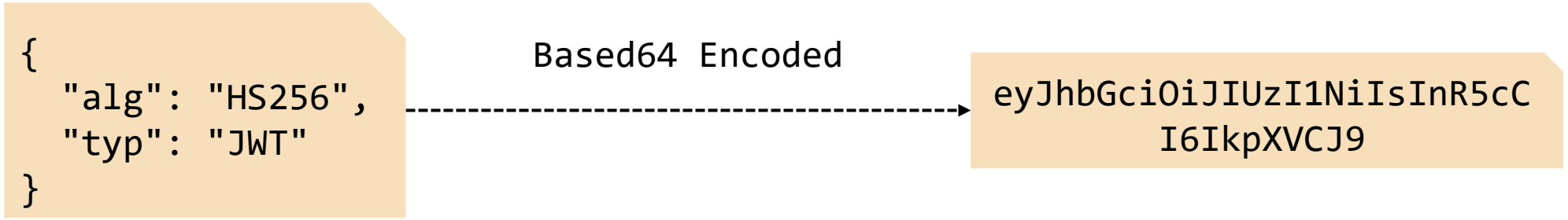
- A JWT token has 3 parts each separated by a period(.). Below is a sample JWT token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIi
wibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF
2QT4fwpMeJf36POk6yJV_adQssw5c
```

1. Header
2. Payload
3. Signature(Optional)

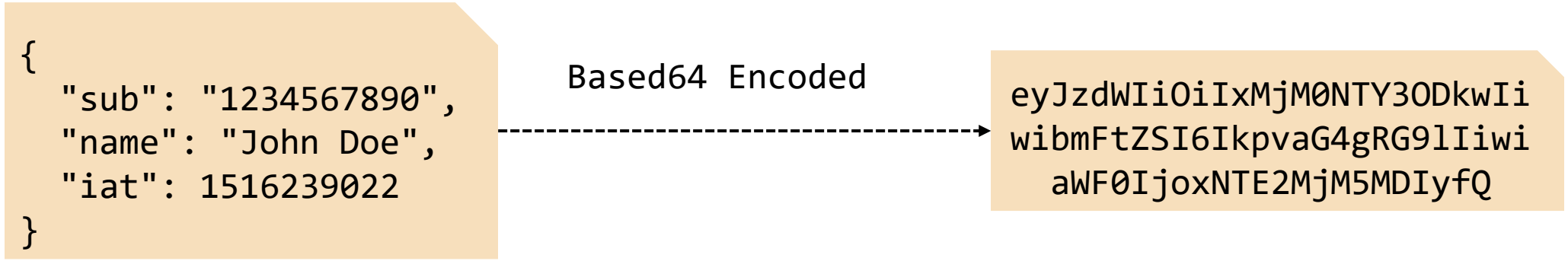
JWT - Header

- Inside the JWT header, metadata and information related to the token are stored. If the token is signed, the header contains the name of the algorithm used for generating the signature.



JWT - Payload

- In the payload, token can store details related to user, role etc. Which can be used later for Authentication and Authorization.
- There is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible



JWT - Signature

- This part can be optional if the party share the JWT token is internal and that someone who you can trust but not open in the web.
- If the token is shared in the open web, we need to ensure that no one has changed the header and body values.

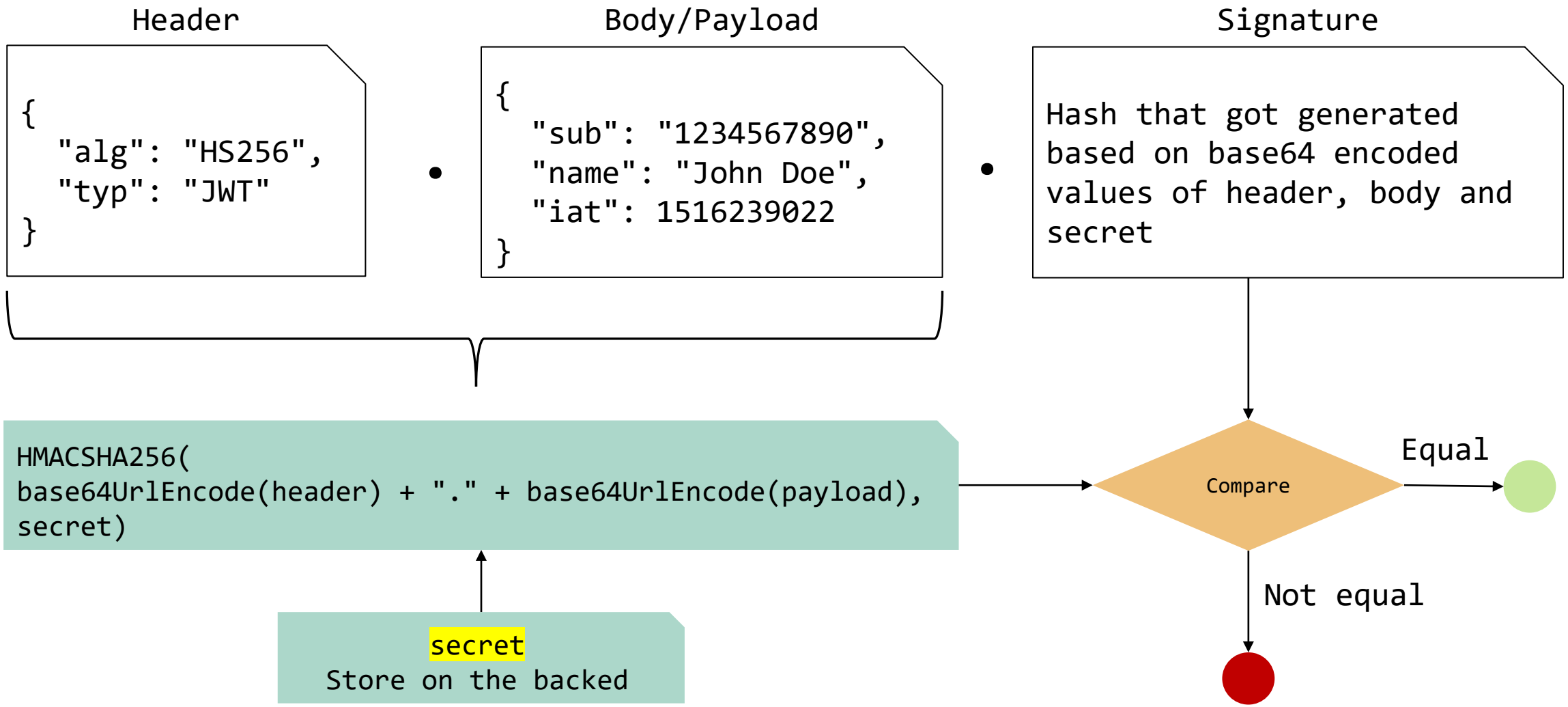
JWT - Signature

- To create the signature, encode the header and payload, use a secret, and apply the algorithm specified in the header for signing.
- The signature is used to verify the message wasn't changed along the way.

For example: use HMAC SHA256 algorithm, the signature will be created:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

JWT - Validation

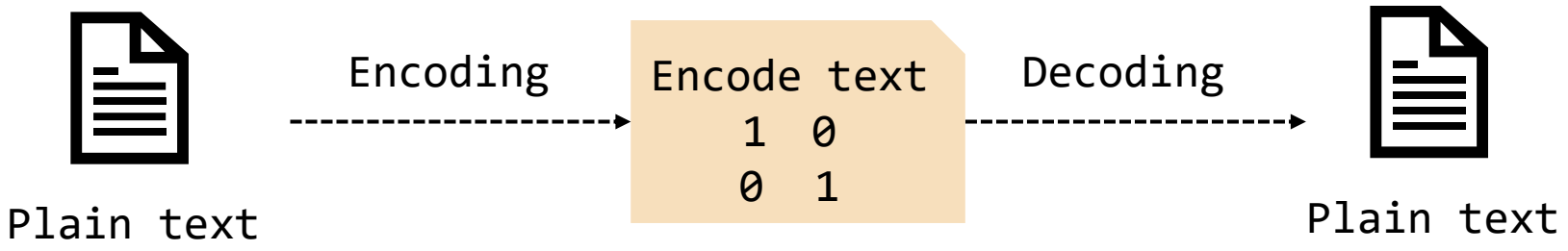


Section 3

PasswordEncoders

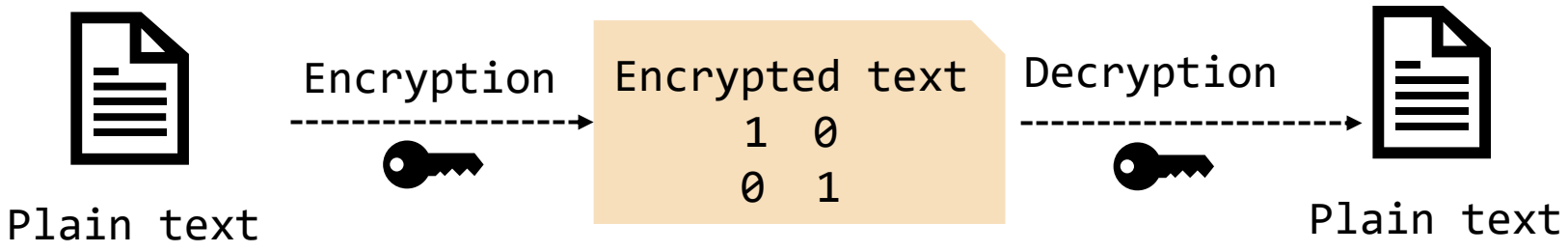
Encoding

- The process of converting data from one form to another and has nothing to do with cryptography.
- It involves no secret and completely reversible.
- Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding. Ex: ASCII, Base64



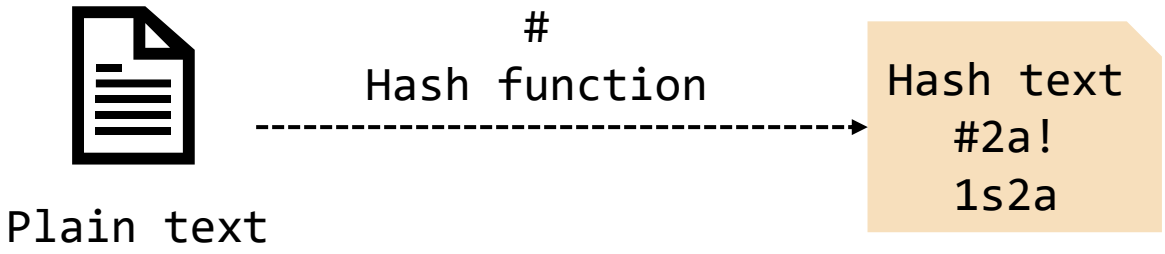
Encryption

- The process of transforming data in such a way that guarantees confidentiality.
- To achieve confidentiality, encryption requires the use of a secret which, in cryptographic terms, we call a "key"
- Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured



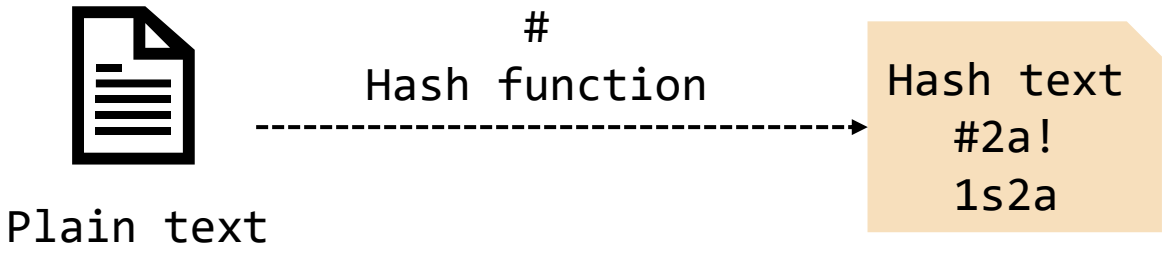
Hashing

- Data is converted to the hash value using some hashing function.
- Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- Can verify whether this data matches the original input data without needing to see the original data.



Hashing

- Data is converted to the hash value using some hashing function.
- Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- Can verify whether this data matches the original input data without needing to see the original data.



Details of PasswordEncoder

- Methods inside PasswordEncoder interface

```
public interface PasswordEncoder {
   String encode(CharSequence rawPassword);

   boolean matches(CharSequence rawPassword, String encodedPassword);

   default boolean upgradeEncoding(String encodedPassword) {
      return false;
   }
}
```

Details of PasswordEncoder

- Implementations of PasswordEncoder inside Spring Security

BCryptPasswordEncoder

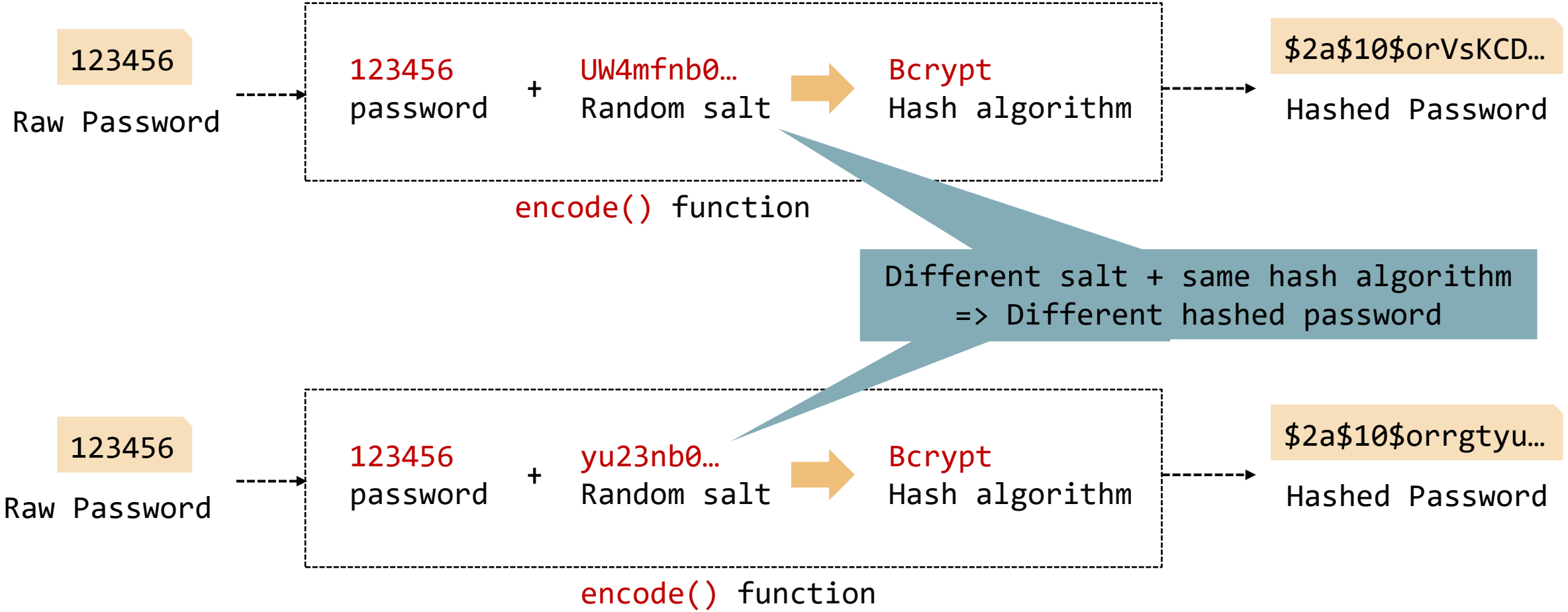
Argon2PasswordEncoder

Pbkdf2PasswordEncoder

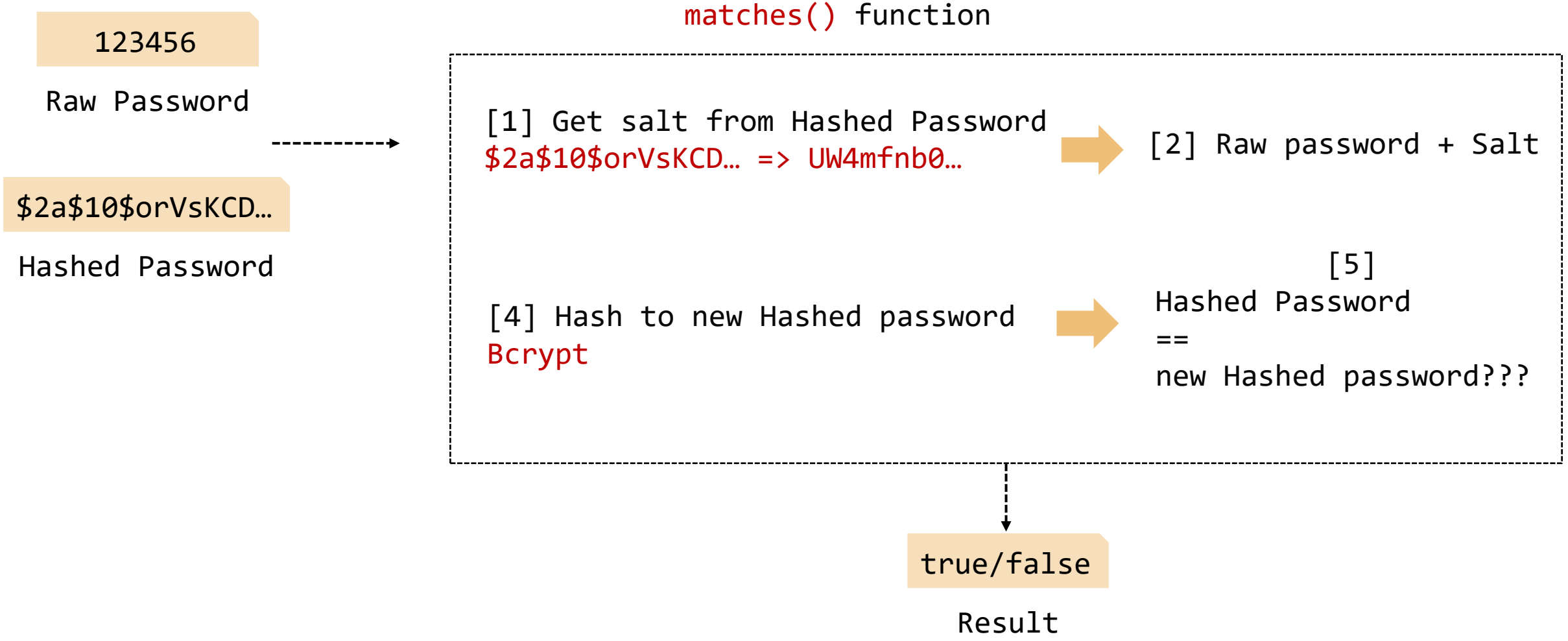
SCryptPasswordEncoder

Other PasswordEncoders

BCryptPasswordEncoder – encode()



BCryptPasswordEncoder – matches()



Section 3

Authentication & Authorization

Installation - Spring Security

- Spring Boot provides a **spring-boot-starter-security** starter that aggregates Spring Security-related dependencies

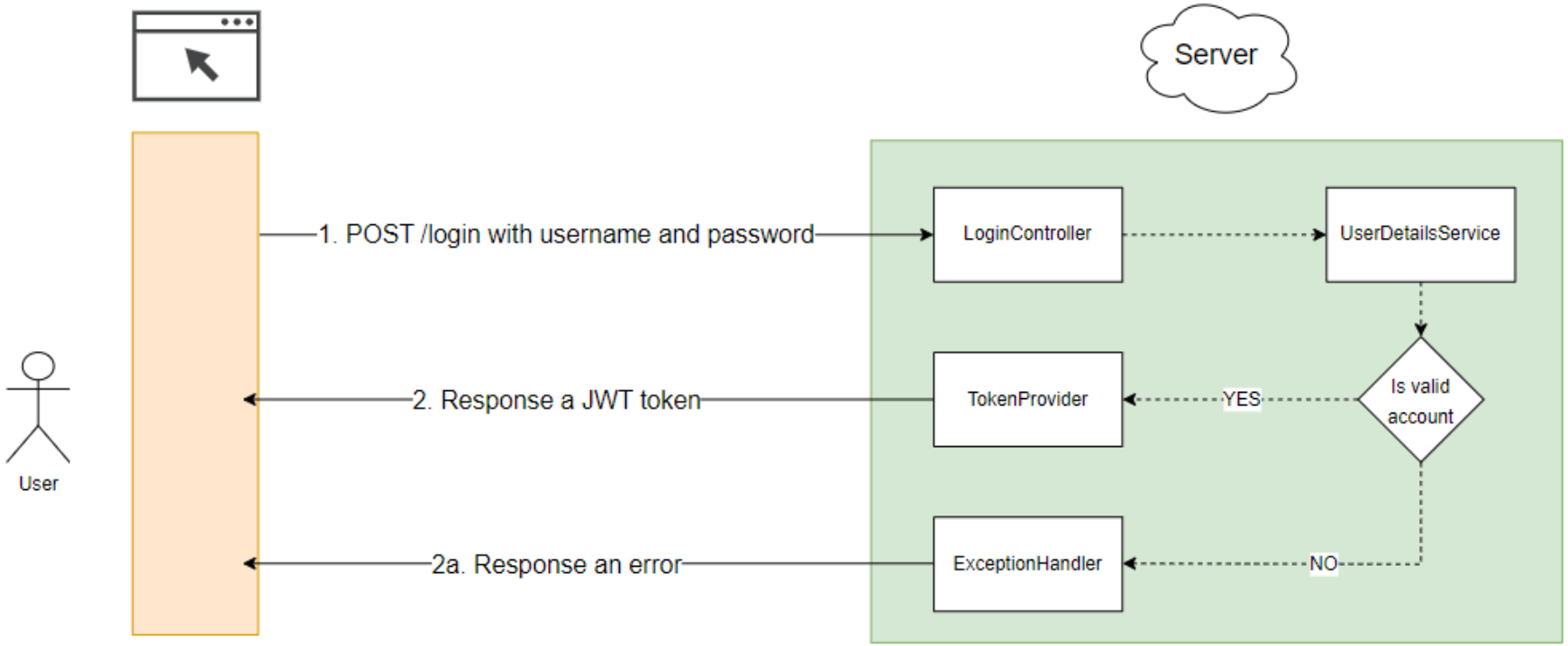
```
<dependencies>
   <!-- ... other dependency elements ... -->
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
   </dependency>
</dependencies>
```

Installation – JWT library

- The "io.jsonwebtoken:jjwt" library serves the purpose of creating, verifying, and managing JSON Web Tokens (JWTs) in Java applications.

```
<dependencies>
   <!-- ... other dependency elements ... -->
  <dependency>
     <groupId>io.jsonwebtoken</groupId>
     <artifactId>jjwt</artifactId>
     <version>0.12.3</version>
   </dependency>
</dependencies>
```

Validate user & generate token flow



LoginController

```
@PostMapping("/api/login")
public ResponseEntity<LoginResponseDto> login(@Valid @RequestBody LoginRequestDto loginRequestDto) {
    UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
            loginRequestDto.getUsername(),
            loginRequestDto.getPassword()
    );

    // Call the implementation of UserDetailsService
    Authentication authentication = authenticationManagerBuilder.getObject().authenticate(authenticationToken);
    SecurityContextHolder.getContext().setAuthentication(authentication);

    String accessToken = tokenProvider.generateAccessToken(authentication);
    LoginResponseDto loginResponseDto = LoginResponseDto.builder()
            .accessToken(accessToken)
            .build();

    return ResponseEntity.ok(loginResponseDto);
}
```

Explain:

- ✓ *Exposing a POST API '/api/login' that takes parameters 'username' and 'password'*
- ✓ *Valid and respond a JWT token when the account is valid.*

UserDetailsService

```
@Override
@Transactional(readOnly = true)
public UserDetails loadUserByUsername(final String username) {

    return accountRepository
            .findByUsernameIgnoreCase(username)
            .map(account -> createSpringSecurityUser(username, account))
            .orElseThrow(() -> new UsernameNotFoundException("Account: " + username + " was not found in the
database"));
}
```

Explain :

- ✓ *It takes an `AccountRepository` to fetch user information from a database*
- ✓ *The `loadUserByUsername` method finds a user by username, maps it to a `UserDetails` object*

TokenProvider

```
public String generateAccessToken(String account, String userRoles) {

    LocalDateTime expiredTime = LocalDateTime.now().plusSeconds(accessTokenExpiredInSecond);
    SecretKey secretKey = Keys.hmacShaKeyFor(Decoders.BASE64URL.decode(secretKeyStr));
    return Jwts.builder()
            .subject(account)
            .claim("authorities", userRoles)
            .expiration(Date.from(expiredTime.atZone(ZoneId.systemDefault()).toInstant()))
            .signWith(secretKey)
            .compact();
}
```

Explain:

- ✓ *Sets the subject (sub) of the token,' identifying the user associated with the token.*
- ✓ *Includes a custom claim named "authorities" to represent the user's roles.*
- ✓ *Specifies the token's expiration time.*
- ✓ *Signs the token with the secret key to protect it from tampering.*

TokenProvider

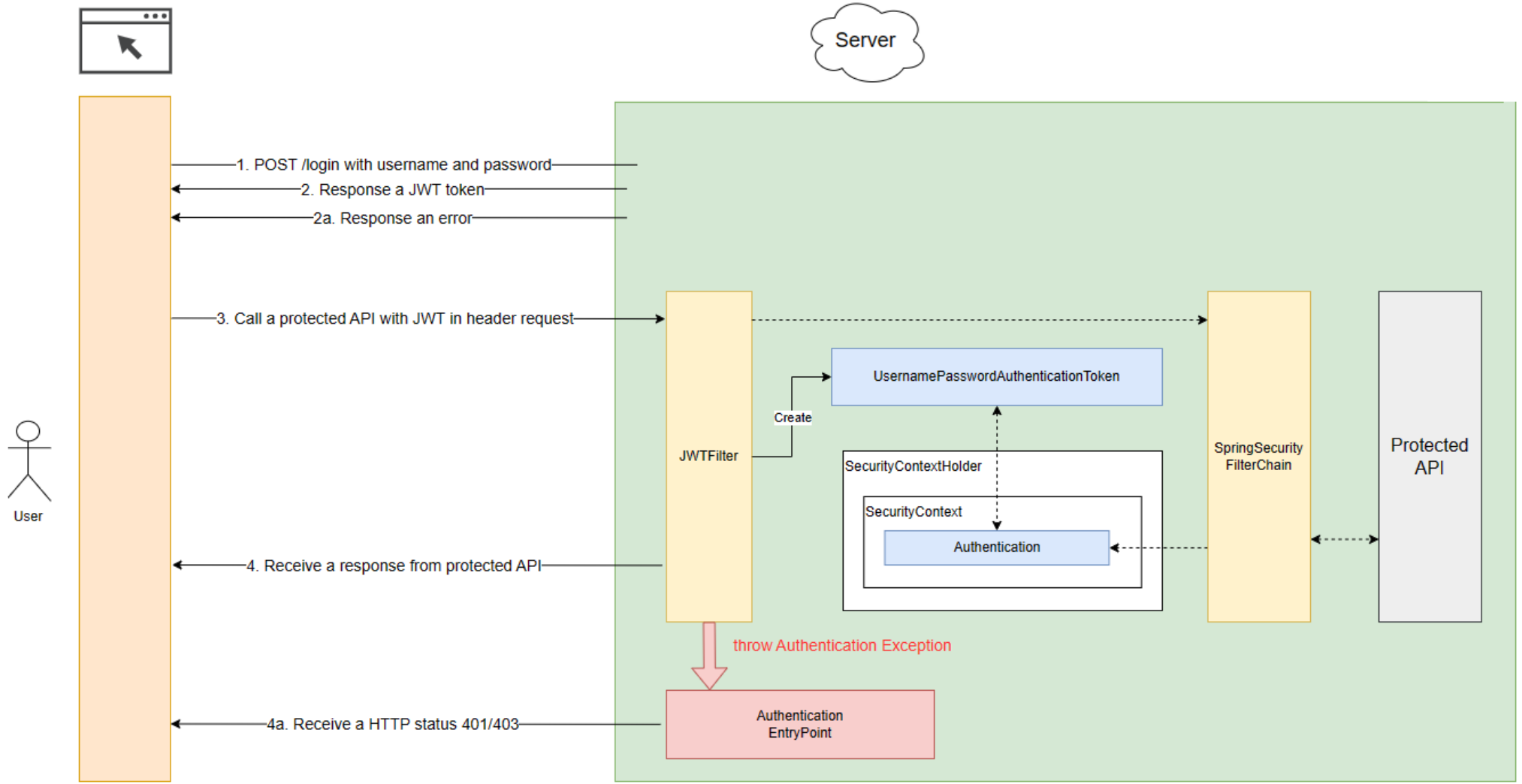
```
public Authentication getAuthentication(String token) {
    // …. Other implementation
     Claims claims = Jwts.parser()
                    .verifyWith(secretKey)
                    .build().parseSignedClaims(token).getPayload();

     String authorities = claims.get("authorities").toString();

     List<GrantedAuthority> authorityList = Arrays.stream(authorities.split(SecurityUtils.ROLE_DELIMITER))
                                    .map(SimpleGrantedAuthority::new)
                                    .collect(Collectors.toList());
     User principal = new User(claims.getSubject(), "", authorityList);
     // …. Other implementation
}
```

Explain:

- Parse JWT token to get information

- Throw exception if JWT token is invalid or expired

Validate token flow



JWTFilter

- The task at hand is to create a `JWTFilter` for token authentication, ensuring that it runs before the Spring Security filter chain.
- When a valid token is encountered, a `UsernamePasswordAuthenticationToken` is initialized and placed into the `SecurityContext`.
- In case of an invalid token, an HTTP Status 401 response is sent.

JWTFilterConfiguration

```
public class JWTFilterConfiguration
      extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {

   private final TokenProvider tokenProvider;
   public JWTFilterConfiguration(TokenProvider tokenProvider) {
      this.tokenProvider = tokenProvider;
   }

   @Override
   public void configure(HttpSecurity builder) {
      JWTFilter filter = new JWTFilter(tokenProvider);
      builder.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
   }
}
```

Explain:

The purpose of `JWTFilterConfiguration` is to configure and ensure that `JWTFilter` runs before the `UsernamePasswordAuthenticationFilter`.

SecurityConfiguration

```
@Bean
public CorsFilter corsFilter() {
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowCredentials(true);
    config.setAllowedOriginPatterns(Collections.singletonList("*"));
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");
    source.registerCorsConfiguration("/**", config);

    return new CorsFilter(source);
}
```

Explain:

- ✓ *`@Bean` method for creating a `CorsFilter`.*
- ✓ *Configures CORS settings for the application.*
- ✓ *Allows all origins, headers, and HTTP methods.*

SecurityConfiguration

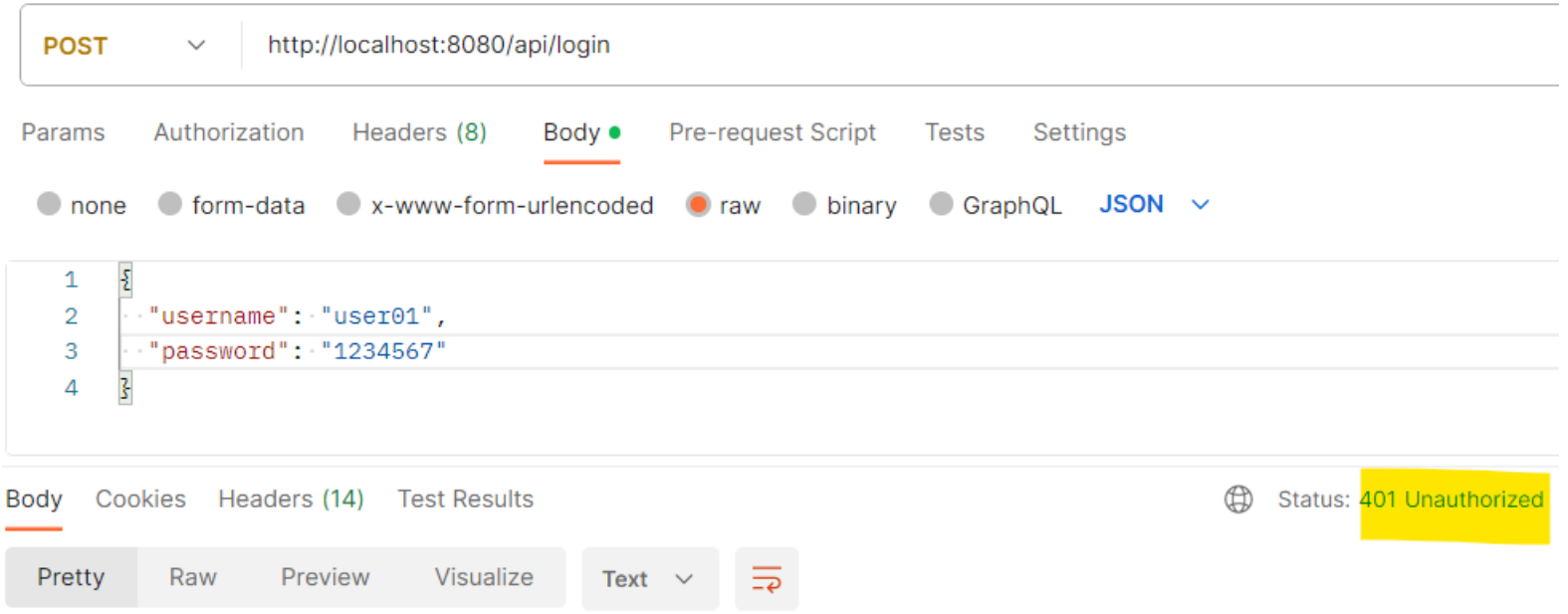
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .addFilterBefore(corsFilter(), UsernamePasswordAuthenticationFilter.class)
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(request -> request
            .requestMatchers("/api/login").permitAll().anyRequest().authenticated()
        ).httpBasic(Customizer.withDefaults())
        .apply(new JWTFilterConfiguration(tokenProvider));
    return http.build();
}
```

Explain:

- ✓ *Disable CSRF (Cross-Site Request Forgery) protection.*
- ✓ *Apply a CORS (Cross-Origin Resource Sharing) filter.*
- ✓ *Unprotected API at the following URL: /api/login.*
- ✓ *Apply JWTConfigurer and JwtAuthenticationEntryPoint.*

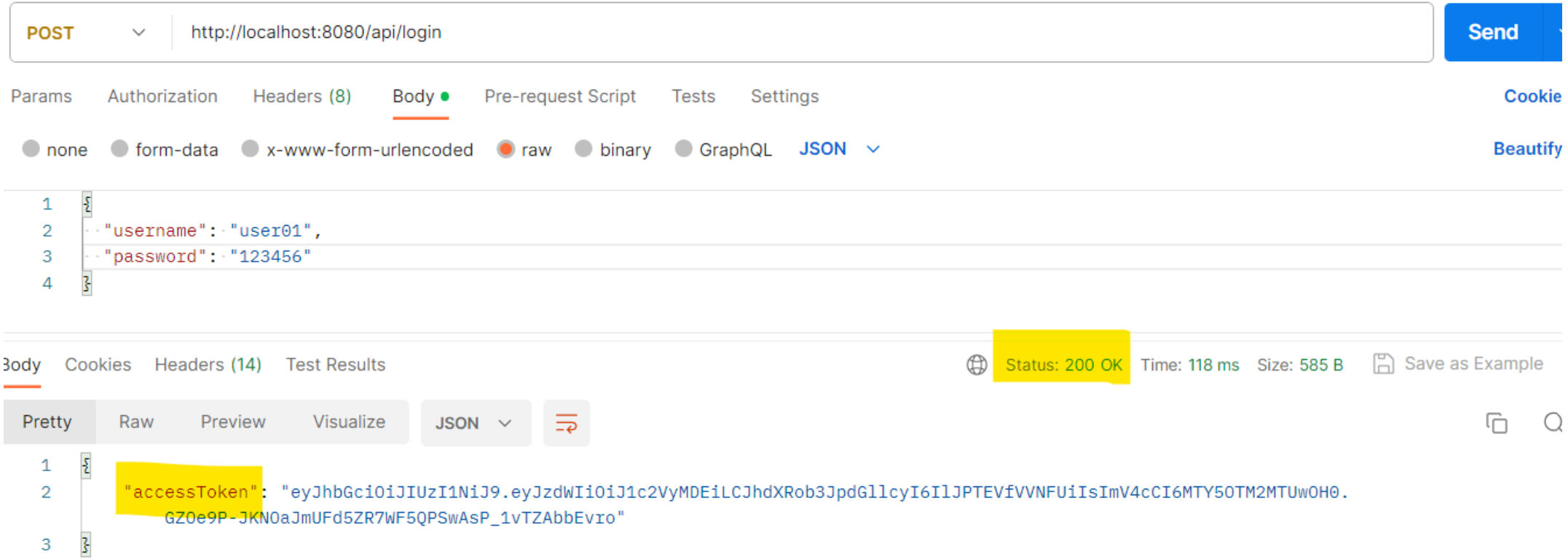
Authentication Demo

- Case of authentication with an invalid account.



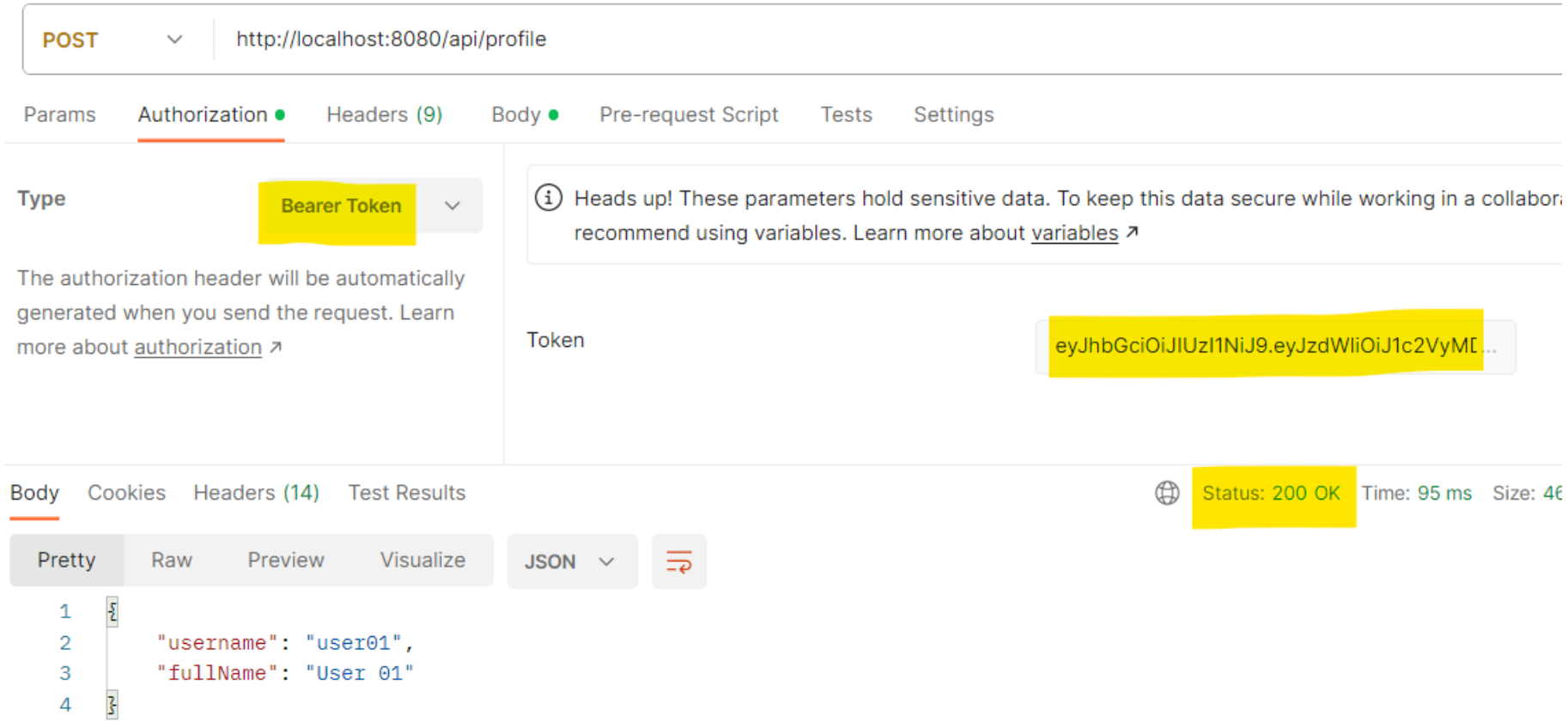
Authentication Demo

- Case of authentication with a valid account.

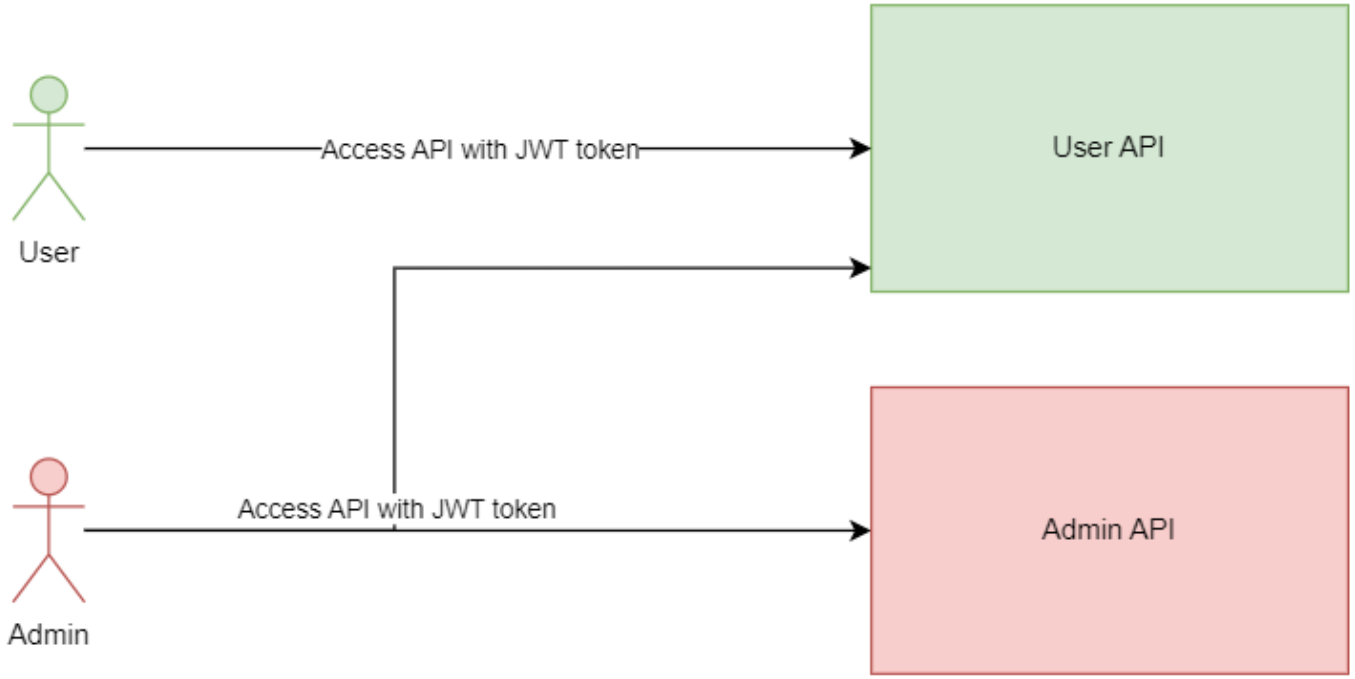


Authentication Demo

- Using the received JWT token, add it to the request header and call the protected API.



Authorization overview



Authorization with hasRole()

- For example, protected APIs with the `"/api/admin"` prefix, only users with the “ADMIN" role can access them.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http
        // …. Others implementation
        .authorizeHttpRequests(request -> request
                .requestMatchers("/api/login").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
         )
        // …. Others implementation
   return http.build();
}
```

Authorization with hasAnyRole()

- For example, protected APIs with the `"/api/user"` prefix, only users with the "ADMIN" or "USER" role can access them.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        // .... Others implementation
        .authorizeHttpRequests(request -> request
                .requestMatchers("/api/login").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/user/**").hasAnyRole("ADMIN", "USER")
                .anyRequest().authenticated()
        )
        // .... Others implementation
    return http.build();
}
```

Authorization Demo

- We will perform testing with the matrix as follows:

Role	API: /api/user/**	API: /api/admin/**
USER	ALLOWED ACCESS	ACCESS DENIED
ADMIN	ALLOWED ACCESS	ALLOWED ACCESS

Section 5

AuditorAware with spring security

Annotation based auditing metadata

- Spring Data provides sophisticated support to transparently keep track of who created or changed an entity.

```
@Column(name = "created_by")
@CreatedBy
private String createdBy;

@Column(name = "last_modified_by")
@LastModifiedBy
private String lastModifiedBy;
```

AuditorAware

- Use either `@CreatedBy` or `@LastModifiedBy`, the auditing infrastructure somehow needs to become aware of the current principal.
- Spring provide an `AuditorAware<T>` interface that you have to implement to tell the infrastructure who the current user.

```
@Component
public class AppAuditorAware implements AuditorAware<String> {

    @Override
    public Optional<String> getCurrentAuditor() {
        return SecurityUtils.getCurrentUserLogin();
    }
}
```

Summary

- Token Based Authentication
- JSON Web Token
- PasswordEncoder
- Authentication
- Authorization
- AuditorAware with spring security

THANK YOU!