JWT/JWS Token Authn/Authz

jim stafford

Fall 2022 v2020-07-15: Built: 2022-12-07 06:16 EST

Table of Contents

1. Introduction . . . 1
    1.1. Goals . . . 1
    1.2. Objectives . . . 1
2. Identity and Authorities . . . 2
    2.1. BASIC Authentication/Authorization . . . 2
3. Tokens . . . 4
    3.1. Token Authentication/Login . . . 4
    3.2. Token Authorization/Operation . . . 5
    3.3. Authentication Separate from Authorization . . . 6
    3.4. JWT Terms . . . 6
4. JWT Authentication . . . 8
    4.1. Example JWT Authentication/Login Flow . . . 8
    4.2. Example JWT Authorization/Operation Call Flow . . . 8
5. Maven Dependencies . . . 10
6. JwtConfig . . . 11
    6.1. JwtConfig application.properties . . . 11
7. JwtUtil . . . 13
    7.1. Dependencies on JwtUtil . . . 13
    7.2. JwtUtil: generateToken() . . . 14
    7.3. JwtUtil: generateToken() Helper Methods . . . 14
    7.4. Example Encoded JWS . . . 15
    7.5. Example Decoded JWS Header and Body . . . 15
    7.6. JwtUtil: parseToken() . . . 16
    7.7. JwtUtil: parseToken() Helper Methods . . . 17
8. JwtAuthenticationFilter . . . 18
    8.1. JwtAuthenticationFilter Relationships . . . 18
    8.2. JwtAuthenticationFilter: Constructor . . . 19
    8.3. JwtAuthenticationFilter: attemptAuthentication() . . . 19
    8.4. JwtAuthenticationFilter: attemptAuthentication() DTO . . . 20
    8.5. JwtAuthenticationFilter: attemptAuthentication() Helper Method . . . 20
    8.6. JwtAuthenticationFilter: successfulAuthentication() . . . 21
9. JwtAuthorizationFilter . . . 22
    9.1. JwtAuthorizationFilter Relationships . . . 22
    9.2. JwtAuthorizationFilter: Constructor . . . 23
    9.3. JwtAuthorizationFilter: doFilterInternal() . . . 23
    9.4. JwtAuthenticationToken . . . 24
    9.5. JwtEntryPoint . . . 26
10. API Security Configuration . . . 27

10.1. API Authentication Manager Builder 27
10.2. API HttpSecurity Key JWS Parts 28
10.3. API HttpSecurity Full Details 28
11. Example JWT/JWS Application 30
11.1. Roles and Role Inheritance 30
11.2. CartsService 30
11.3. Login 31
11.4. createCart() 32
11.5. addItem() 33
11.6. getCart() 34
11.7. removeCart() 35
12. Summary 37

Chapter 1. Introduction

In previous lectures we have covered many aspects of the Spring/Spring Boot authentication and authorization frameworks and have mostly demonstrated that with HTTP Basic Authentication. In this lecture we are going to use what we learned about the framework to implement a different authentication strategy — JSON Web Token (JWT) and JSON Web Signature (JWS).

The focus on this lecture will be a brief introduction to JSON Web Tokens (JWT) and how they could be implemented in the Spring/Spring Boot Security Framework. The real meat of this lecture is to provide a concrete example of how to leverage and extend the provided framework.

1.1. Goals

You will learn:

- what is a JSON Web Token (JWT) and JSON Web Secret (JWS)
- what problems does JWT/JWS solve with API authentication and authorization
- how to write and integrate custom authentication and authorization framework classes to implement an alternate security mechanism
- how to leverage Spring Expression Language to evaluate parameters and properties of the `SecurityContext`

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. construct and sign a JWT with claims representing an authenticated user
2. verify a JWS signature and parse the body to obtain claims to re-instantiate an authenticated user details
3. identify the similarities and differences in flows between HTTP Basic and JWS authentication/authorization flows
4. build a custom JWS authentication filter to extract login information, authenticate the user, build a JWS bearer token, and populate the HTTP response header with its value
5. build a custom JWS authorization filter to extract the JWS bearer token from the HTTP request, verify its authenticity, and establish the authenticated identity for the current security context
6. implement custom error reporting with authentication and authorization

Chapter 2. Identity and Authorities

Some key points of security are to identify the caller and determine authorities they have.

- When using BASIC authentication, we presented credentials each time. This was all in one shot, every time on the way to the operation being invoked.
- When using FORM authentication, we presented credentials (using a FORM) up front to establish a session and then referenced that session on subsequent calls.

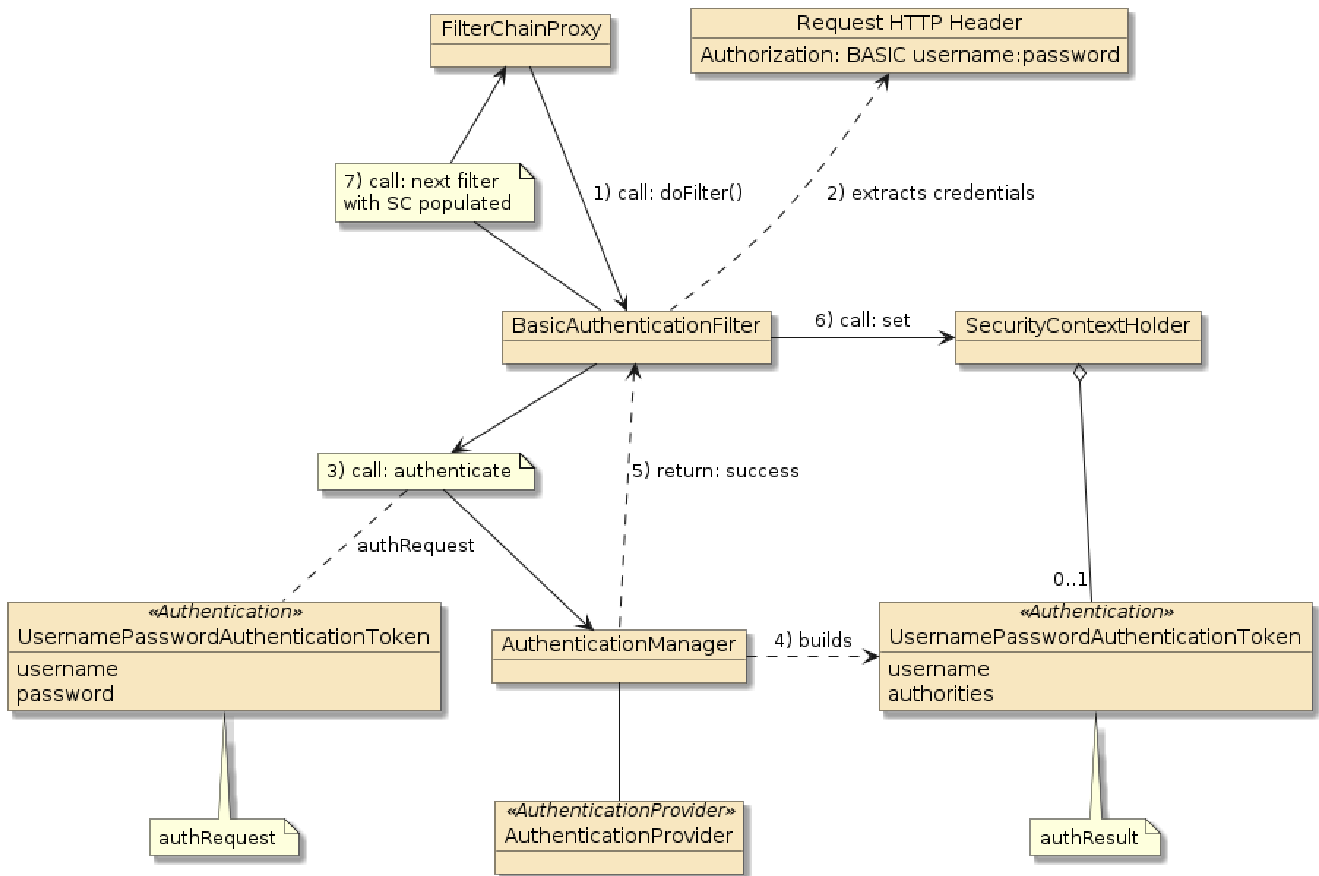
The benefit to BASIC is that is stateless and can work with multiple servers — whether clustered or peer services. The bad part about BASIC is that we must present the credentials each time and the services must have access to our user details (including passwords) to be able to do anything with them.

The benefit to FORM is that we present credentials one time and then reference the work of that authentication through a session ID. The bad part of FORM is that the session is on the server and harder to share with members of a cluster and impossible to share with peer services.

What we intend to do with token-based authentication is to mimic the one-time login of FORM and stateless aspects of BASIC. To do that — we must give the client at login, information they can pass to the services hosting operations that can securely identify them (at a minimum) and potentially identify the authorities they have without having that stored on the server hosting the operation.

2.1. BASIC Authentication/Authorization

To better understand the token flow, I would like to start by reviewing the BASIC Auth flow.



1. the `BasicAuthenticationFilter` ("the filter") is called in its place within the `FilterChainProxy`
2. the filter extracts the username/password credentials from the `Authorization` header and stages them in a `UsernamePasswordAuthenticationToken` ("the authRequest")
3. the filter passes the authRequest to the `AuthenticationManager` to authenticate
4. the `AuthenticationManager`, thru its assigned `AuthenticationProvider`, successfully authenticates the request and builds an authResult
5. the filter receives the successful response with the authResult hosting the user details — including username and granted authorities
6. the filter stores the authResult in the `SecurityContext`
7. the filter invokes the next filter in the chain — which will eventually call the target operation

All this — authentication and user details management — must occur within the same server as the operation for BASIC Auth.

Chapter 3. Tokens

With token authentication, we are going to break the flow into two parts: authentication/login and authorization/operation.

3.1. Token Authentication/Login

The following is a conceptual depiction of the authentication flow. It differs from the BASIC Authentication flow in that nothing is stored in the `SecurityContext` during the login/authentication. Everything needed to authorize the follow-on operation call is encoded into a `Bearer Token` and returned to the caller in an `Authorization` header. Things encoded in the bearer token are referred to as "claims".

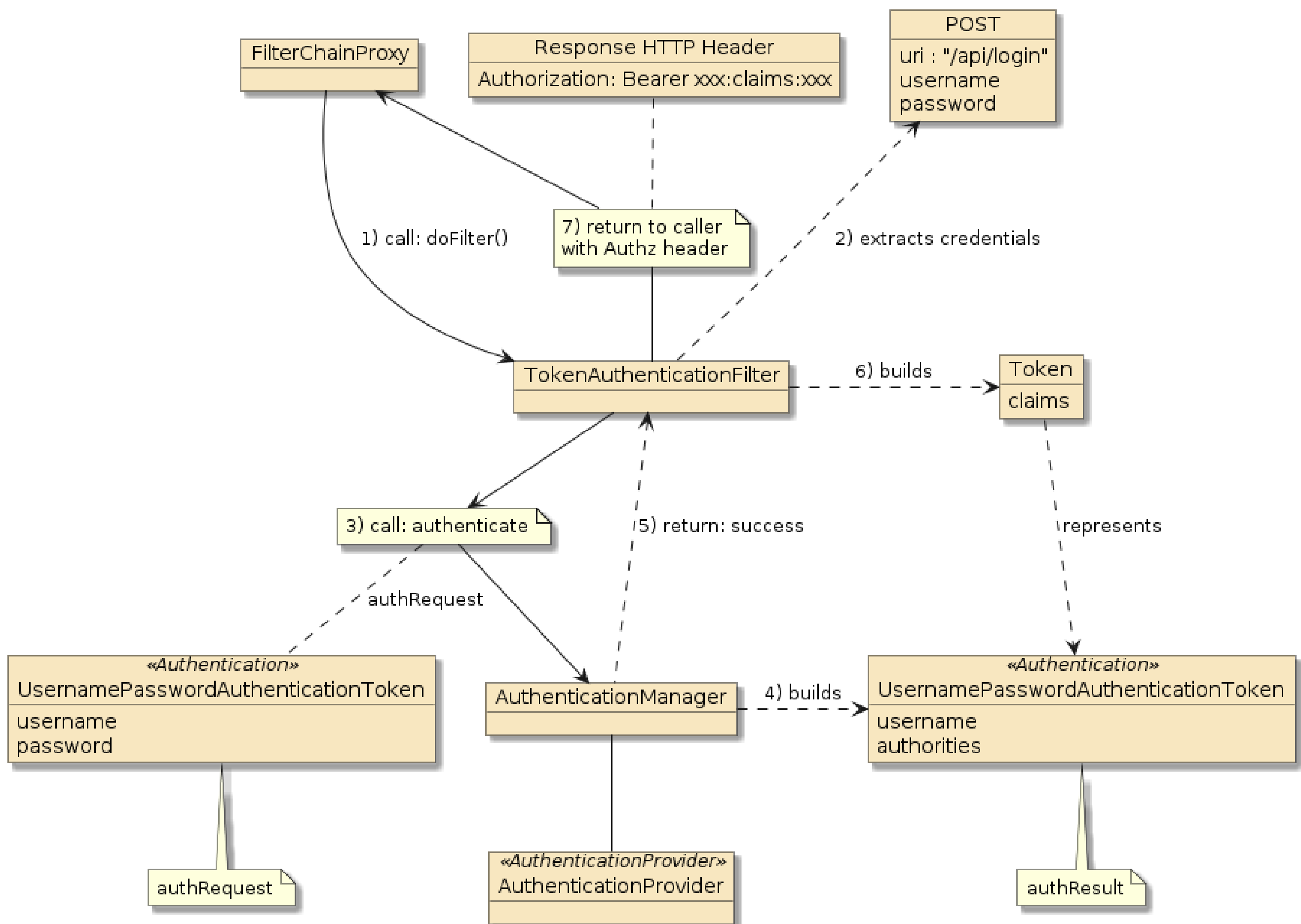


Figure 1. Example Notional Token Authentication/Login

Step 2 extracts the username/password from a POST payload — very similar to FORM Auth. However, we could have just as easily implemented the same extract technique used by BASIC Auth.

Step 7 returns the the token representation of the authResult back to the caller that just successfully authenticated. They will present that information later when they invoke an operation in this or a different server. There is no requirement for the token returned to be used locally. The token can be used on any server that trusts tokens created by this server. The biggest requirement is that we must trust the token is built by something of trust and be able to verify that it never gets modified.

3.2. Token Authorization/Operation

To invoke the intended operation, the caller must include an `Authorization` header with the bearer token returned to them from the login. This will carry their identity (at a minimum) and authorities encoded in the bearer token's claims section.

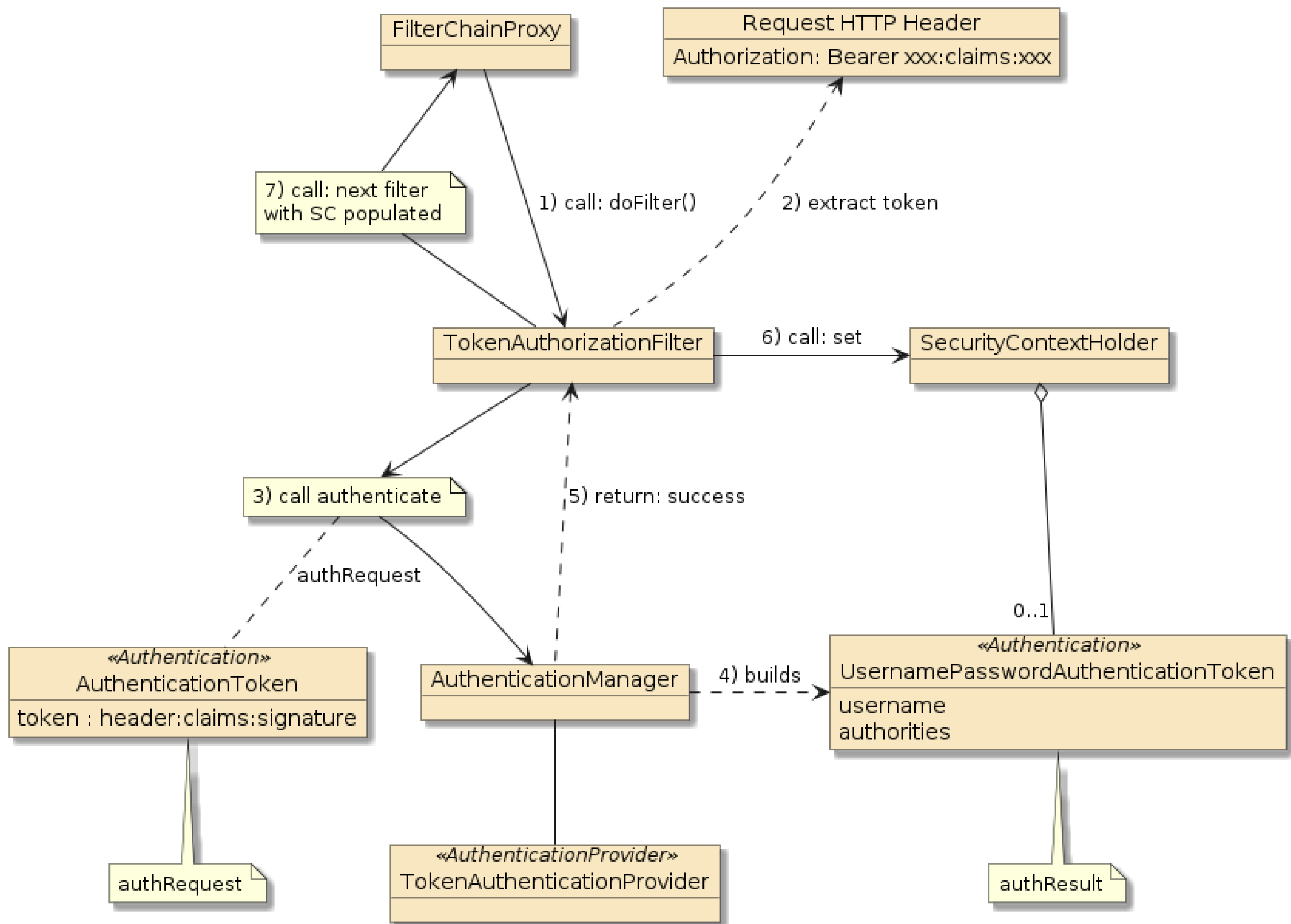


Figure 2. Example Notational Token Authorization/Operation

1. the Token AuthorizationFilter ("the filter") is called by the `FilterChainProxy`
2. the filter extracts the bearer token from the `Authorization` header and wraps that in an authRequest
3. the filter passes the authRequest to the `AuthenticationManager` to authenticate
4. the `AuthenticationManager` with its Token `AuthenticationProvider` are able to verify the contents of the token and re-build the necessary portions of the authResult
5. the authResult is returned to the filter
6. the filter stores the authResult in the `SecurityContext`
7. the filter invokes the next filter in the chain — which will eventually call the target operation

Bearer Token has Already Been Authenticated

Since the filter knows this is a bearer token, it could have bypassed the call to the `AuthenticationManager`. However, by doing so — it makes the responsibilities of the classes consistent with their original purpose and also gives the

`AuthenticationProvider` the option to obtain more user details for the caller.

3.3. Authentication Separate from Authorization

Notice the overall client to operation call was broken into two independent workflows. This enables the client to present their credentials a limited amount of times and for the operations to be spread out through the network. The primary requirement to allow this to occur is **TRUST**.

We need the ability for the authResult to be represented in a token, carried around by the caller, and presented later to the operations with the trust that it was not modified.

JSON Web Tokens (JWT) are a way to express the user details within the body of a token. JSON Web Signature (JWS) is a way to assure that the original token has not been modified. JSON Web Encryption (JWE) is a way to assure the original token stays private. This lecture and example will focus in JWS — but it is common to refer to the overall topic as JWT.

3.4. JWT Terms

The following table contains some key, introductory terms related to JWT.

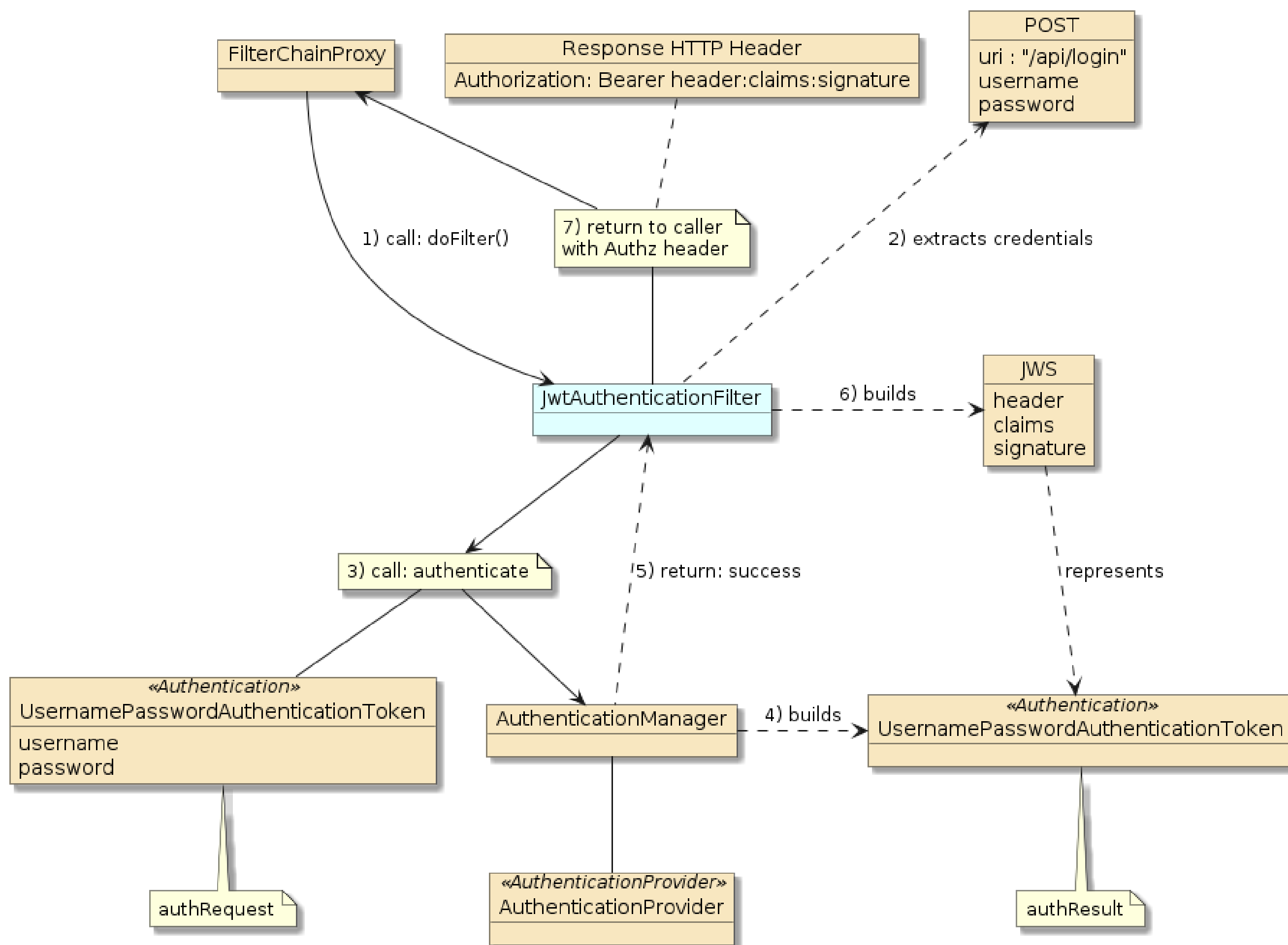
JSON Web Token (JWT)	a compact JSON claims representation that makes up the payload of a JWS or JWE structure e.g., `{"sub":"user1", "auth":["ROLE_ADMIN"]}`. The JSON document is referred to as the JWT Claim Set. Basically — this is where we place what we want to represent. In our case, we will be representing the authenticated principal and their assigned authorities.
JSON Web Signature (JWS)	represents content secured with a digital signature (signed with a private key and verifiable using a sharable public key) or Message Authentication Codes (MACs) (signed and verifiable using a shared, symmetric key) using JSON-based data structures
JSON Web Encryption (JWE)	represents encrypted content using JSON-based data structures
JSON Web Algorithms (JWA)	a registry of required, recommended, and optional algorithms and identifiers to be used with JWS and JWE
JSON Object Signing and Encryption (JOSE) Header	JSON document containing cryptographic operations/parameters used. e.g., `{"typ":"JWT","alg":"HW256"}`
JWS Payload	the message to be secured — an arbitrary sequence of octets
JWS Signature	digital signature or MAC over the header and payload
Unsecured JWS	JWS without a signature (`"alg":"none"`)

JWS Compact Serialization	a representation of the JWS as a compact, URL-safe String meant for use in query parameters and HTTP headers `base64({"typ":"JWT","alg":"HW256"})` `.base64({"sub":"user1", "auth":["ROLE_ADMIN"]})` `.base64(signature(JOSE + Payload))`
JWS JSON Serialization	a JSON representation where individual fields may be signed using one or more keys. There is no emphasis for compact for this use but it makes use of many of the underlying constructs of JWS.

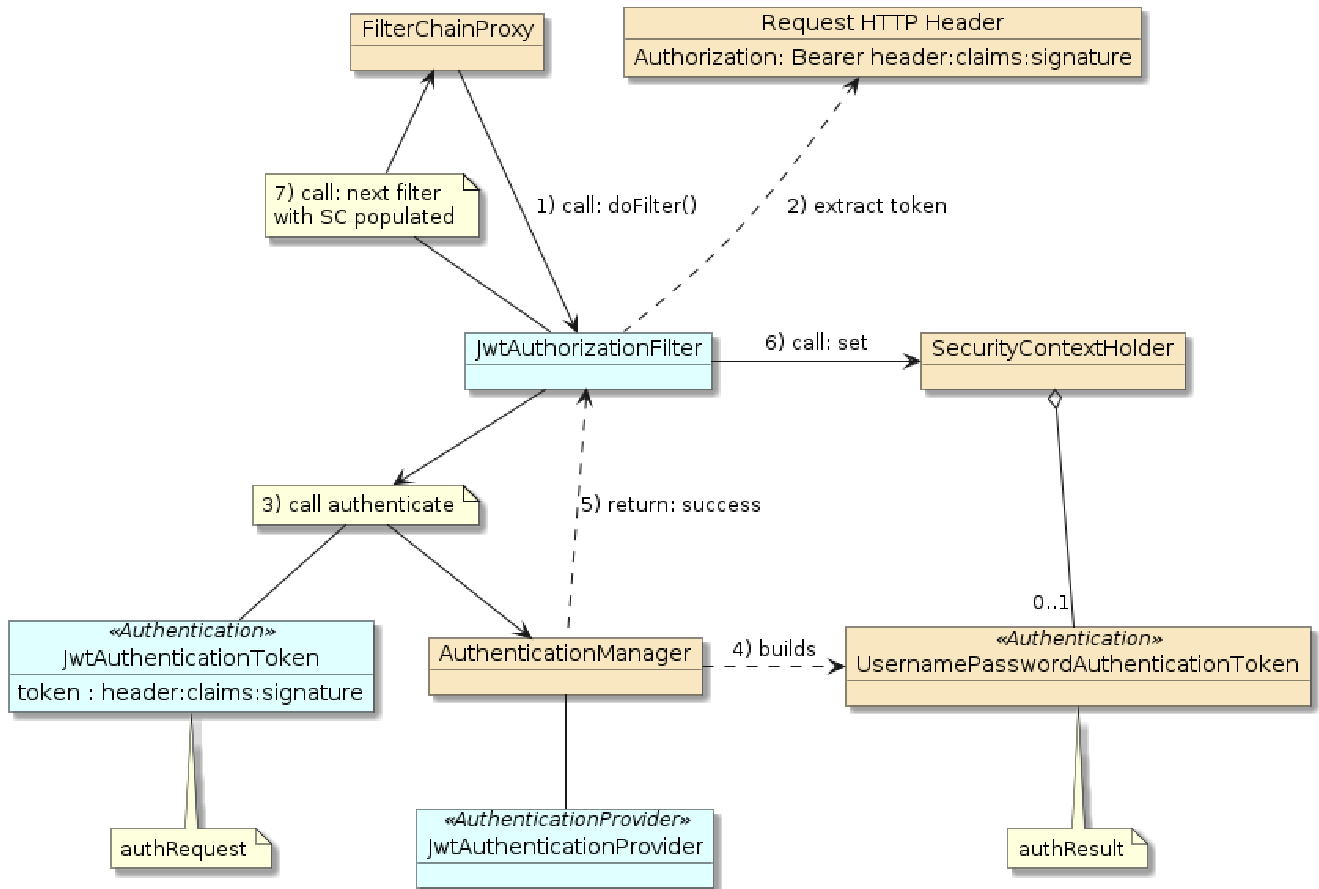
Chapter 4. JWT Authentication

With the general workflows understood and a few concepts of JWT/JWS introduced, I want to update the diagrams slightly with real classnames from the examples and walk through how we can add JWT authentication to Spring/Spring Boot.

4.1. Example JWT Authentication/Login Flow



4.2. Example JWT Authorization/Operation Call Flow



Lets take a look at the implementation to be able to fully understand both JWT/JWS and leveraging the Spring/Spring Boot Security Framework.

Chapter 5. Maven Dependencies

Spring does not provide its own standalone JWT/JWS library or contain a direct reference to any. I happen to be using the jjwt library from jsonwebtoken.

JWT/JWS Maven Dependencies

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <scope>runtime</scope>
</dependency>
```

Chapter 6. JwtConfig

At the bottom of the details of our JWT/JWS authentication and authorization example is a `@ConfigurationProperties` class to represent the configuration.

Example JwtConfig @ConfigurationProperties Class

```
@ConfigurationProperties(prefix = "jwt")
@Data
@Slf4j
public class JwtConfig {
    @NotNull
    private String loginUri; ①
    private String key; ②
    private String authoritiesKey = "auth"; ③
    private String headerPrefix = "Bearer "; ④
    private int expirationSecs=60*60*24; ⑤

    public String getKey() {
        if (key==null) {
            key=UUID.randomUUID().toString();
            log.info("generated JWT signing key={}",key);
        }
        return key;
    }
    public SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(getKey().getBytes(Charset.forName("UTF-8")));
    }
    public SecretKey getVerifyKey() {
        return getSigningKey();
    }
}
```

① `login-uri` defines the URI for the JWT authentication

② `key` defines a value to build a symmetric `SecretKey`

③ `authorities-key` is the JSON key for the user's assigned authorities within the JWT body

④ `header-prefix` defines the prefix in the `Authorization` header. This will likely never change, but it is good to define it in a single, common place

⑤ `expiration-secs` is the number of seconds from generation for when the token will expire. Set this to a low value to test expiration and large value to limit login requirements

6.1. JwtConfig application.properties

The following shows an example set of properties defined for the `@ConfigurationProperties` class.

Example property value

```
①
```

```
jwt.key=123456789012345678901234567890123456789012345678901234567890
jwt.expiration-secs=300000000
jwt.login-uri=/api/login
```

① the `key` must remain protected — but for symmetric keys must be shared between signer and verifiers

Chapter 7. JwtUtil

This class contains all the algorithms that are core to implementing token authentication using JWT/JWS. It is configured by value in `JwtConfig`.

Example JwtUtil Utility Class

```
@RequiredArgsConstructor
public class JwtUtil {
    private final JwtConfig jwtConfig;
```

7.1. Dependencies on JwtUtil

The following diagram shows the dependencies on `JwtUtil` and also on `JwtConfig`.

- `JwtAuthenticationFilter` needs to process requests to the loginUri, generate a JWS token for successfully authenticated users, and set that JWS token on the HTTP response
- `JwtAuthorizationFilter` processes all messages in the chain and gets the JWS token from the `Authorization` header.
- `JwtAuthenticationProvider` parses the String token into an `Authentication` result.

`JwtUtil` handles the meat of that work relative to JWS. The other classes deal with plugging that work into places in the security flow.

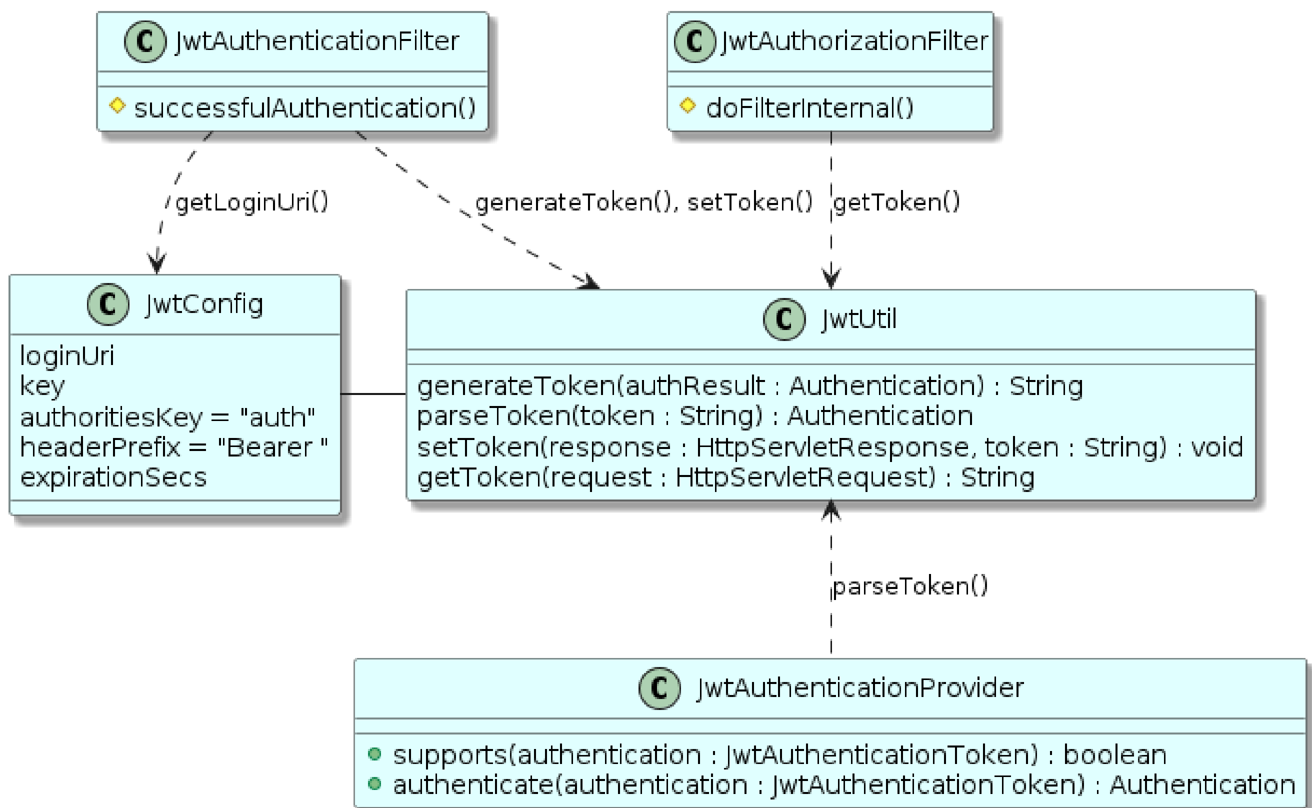


Figure 3. Dependencies on JwtUtil

7.2. JwtUtil: generateToken()

The following code snippet shows creating a JWS builder that will end up signing the header and payload. Individual setters are called for well-known claims. A generic claim(key, value) is used to add the authorities.

JwtUtil generateToken() for Authenticated User

```
import io.jsonwebtoken.Jwts;
...
public String generateToken(Authentication authenticated) {
    String token = Jwts.builder()
            .setSubject(authenticated.getName()) ①
            .setIssuedAt(new Date())
            .setExpiration(getExpires()) ②
            .claim(jwtConfig.getAuthoritiesKey(), getAuthorities(authenticated))
            .signWith(jwtConfig.getSigningKey())
            .compact();
    return token;
}
```

① JWT has some well-known claim values

② `claim(key,value)` used to set custom claim values

7.3. JwtUtil: generateToken() Helper Methods

The following helper methods are used in setting the claim values of the JWT.

JwtUtil generateToken() Helper Methods

```
protected Date getExpires() { ①
    Instant expiresInstant = LocalDateTime.now()
            .plus(jwtConfig.getExpirationSecs(), ChronoUnit.SECONDS)
            .atZone(ZoneOffset.systemDefault())
            .toInstant();
    return Date.from(expiresInstant);
}
protected List<String> getAuthorities(Authentication authenticated) {
    return authenticated.getAuthorities().stream() ②
            .map(a->a.getAuthority())
            .collect(Collectors.toList());
}
```

① calculates an instant in the future — relative to local time — the token will expire

② strip authorities down to String authorities to make marshalled value less verbose

The following helper method in the `JwtConfig` class generates a `SecretKey` suitable for signing the JWS.

JwtConfig getSigningKey() Helper Method

```
...
import io.jsonwebtoken.security.Keys;
import javax.crypto.SecretKey;

public class JwtConfig {
    public SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(getKey() ①
                    .getBytes(Charset.forName("UTF-8")));
    }
```

① the `hmacSha` algorithm and the 40 character key will generate a `HS384 SecretKey` for signing

7.4. Example Encoded JWS

The following is an example of what the token value will look like. There are three base64 values separated by a period "." each. The first represents the header, the second the body, and the third the cryptographic signature of the header and body.

Example Encoded JWS


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk0ODk1Nzk3LCJle
HAiOjE1OTQ4OTk1MTcsImF1dGhvcml0aWVzIjpbIlBSSUNFX0NIRUNLIiwiUk9MRV9DVVNUT01FUiJdLCJqdGk
iOiI5NjQ3MzE1OS03MTNjLTRlN2EtYmE4Zi0zYWMwMzlmODhjZGQifQ.ED-j7mdO2bwNdZdI4I2Hm_88j-
aSeYkrbdlEacmjotU
①
```


① base64(JWS Header).base64(JWS body).base64(sign(header + body))

There is no set limit to the size of HTTP headers. However, it has been pointed out that Apache defaults to an 8KB limit and IIS is 16KB. The default size for Tomcat is 4KB. In case you were counting, the above string is 272 characters long.

7.5. Example Decoded JWS Header and Body

Example Decoded JWS Header and Body

```
{
 "typ": "JWT",
 "alg": "HS384"
}
{
 "sub": "frasier",
 "iat": 1594895797,
 "exp": 1894899397,
 "auth": [
  "PRICE_CHECK",
  "ROLE_CUSTOMER"
 ]
}
```

The following is what is produced if we base64 decode the first two sections. We can use sites like jsonwebtoken.io and jwt.io to inspect JWS tokens. The header identifies the type and signing algorithm. The body carries the claims. Some claims (e.g., subject/`sub`) are well known and standardized. All standard claims are shortened to try to make the token as condensed as possible.

7.6. JwtUtil: parseToken()

The `parseToken()` method verifies the contents of the JWS has not been modified, and re-assembles an authenticated `Authentication` object to be returned by the `AuthenticationProvider` and `AuthenticationManager` and placed into the `SecurityContext` for when the operation is executed.

Example JwtUtil parseToken()

```
...
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;

public Authentication parseToken(String token) throws JwtException {
    Claims body = Jwts.parserBuilder()
            .setSigningKey(jwtConfig.getVerifyKey()) ①
            .build()
            .parseClaimsJws(token)
            .getBody();
    User user = new User(body.getSubject(), "", getGrantedAuthorities(body));
    Authentication authentication=new UsernamePasswordAuthenticationToken(
            user, token, ②
            user.getAuthorities());
    return authentication;
}
```

① verification and signing keys are the same for symmetric algorithms

② there is no real use for the token in the authResult. It was placed in the password position in the event we wanted to locate it.

7.7. JwtUtil: parseToken() Helper Methods

The following helper method extracts the authority strings stored in the (parsed) token and wraps them in `GrantedAuthority` objects to be used by the authorization framework.

JwtUtil parseToken() Helper Methods

```
protected List<GrantedAuthority> getGrantedAuthorities(Claims claims) {
    List<String> authorities = (List) claims.get(jwtConfig.getAuthoritiesKey());
    return authorities==null ? Collections.emptyList() :
        authorities.stream()
                .map(a->new SimpleGrantedAuthority(a)) ①
                .collect(Collectors.toList());
}
```

① converting authority strings from token into `GrantedAuthority` objects used by Spring security framework

The following helper method returns the verify key to be the same as the signing key.

Example JwtConfig parseToken() Helper Methods

```
public class JwtConfig {
    public SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(getKey().getBytes(Charset.forName("UTF-8")));
    }
    public SecretKey getVerifyKey() {
        return getSigningKey();
    }
```

Chapter 8. JwtAuthenticationFilter

The `JwtAuthenticationFilter` is the target filter for generating new bearer tokens. It accepts POSTS to a configured `/api/login` URI with the username and password, authenticates those credentials, generates a bearer token with JWS, and returns that value in the `Authorization` header. The following is an example of making the end-to-end authentication call. Notice the bearer token returned. We will need this value in follow-on calls.

Example End-to-End Authentication Call

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"frasier",
"password":"password"}'
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk0OTgwMTAyLCJleHAiOjE4OTQ5ODM3M
DIsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.u2MmzTxaDoVNFGGCnrAcWBusS_NS2N
ndZXkaT964hLgcDTvCYAW_sXtTxRw8g_13
```

The `JwtAuthenticationFilter` delegates much of the detail work handling the header and JWS token to the `JwtUtil` class shown earlier.

JwtAuthenticationFilter

```
@Slf4j
public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private final JwtUtil jwtUtil;
```

8.1. JwtAuthenticationFilter Relationships

The `JwtAuthenticationFilter` fills out the abstract workflow of the `AbstractAuthenticationProcessingFilter` by implementing two primary methods: `attemptAuthentication()` and `successfulAuthentication()`.

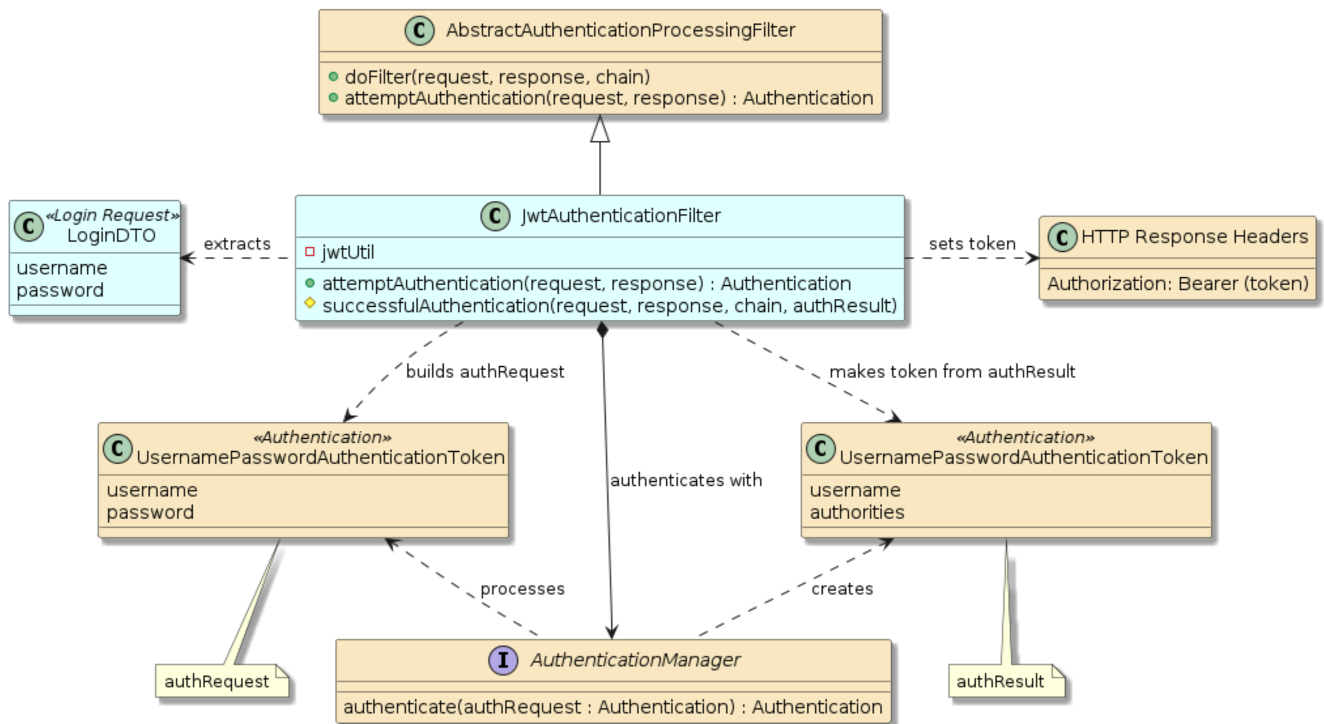


Figure 4. JwtAuthenticationFilter Relationships

The `attemptAuthenticate()` callback is used to perform all the steps necessary to authenticate the caller. Unsuccessful attempts are returned the the caller immediately with a 401/Unauthorized status.

The `successfulAuthentication()` callback is used to generate the JWS token from the authResult and return that in the response header. The call is returned immediately to the caller with a 200/OK status and an Authorization header containing the constructed token.

8.2. JwtAuthenticationFilter: Constructor

The filter constructor sets up the object to only listen to POSTs against the configured loginUri. The base class we are extending holds onto the `AuthenticationManager` used during the `attemptAuthentication()` callback.

JwtAuthenticationFilter Constructor

```
public JwtAuthenticationFilter(JwtConfig jwtConfig, AuthenticationManager authm) {
    super(new AntPathRequestMatcher(jwtConfig.getLoginUri(), "POST"));
    this.jwtUtil = new JwtUtil(jwtConfig);
    setAuthenticationManager(authm);
}
```

8.3. JwtAuthenticationFilter: attemptAuthentication()

The `attemptAuthentication()` method has two core jobs: obtain credentials and authenticate.

- The credentials could have been obtained in a number of different ways. I have simply chosen to create a DTO class with username and password to carry that information.

- The credentials are stored in an Authentication object that acts as the authRequest. The authResult from the AuthenticationManager is returned from the callback.

Any failure (getCredentials() or authenticate()) will result in an AuthenticationException thrown.

JwtAuthenticationFilter attemptAuthentication()

```
@Override
public Authentication attemptAuthentication(
            HttpServletRequest request, HttpServletResponse response)
            throws AuthenticationException { ①

    LoginDTO login = getCredentials(request);
    UsernamePasswordAuthenticationToken authRequest =
        new UsernamePasswordAuthenticationToken(login.getUsername(), login.
getPassword());

    Authentication authResult = getAuthenticationManager().authenticate(authRequest);
    return authResult;
}
```

① any failure to obtain a successful Authentication result will throw an AuthenticationException

8.4. JwtAuthenticationFilter: attemptAuthentication() DTO

The LoginDTO is a simple POJO class that will get marshalled as JSON and placed in the body of the POST.

JwtAuthenticationFilter attemptAuthentication() DTO

```
package info.ejava.examples.svc.auth.cart.security.jwt;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class LoginDTO {
    private String username;
    private String password;
}
```

8.5. JwtAuthenticationFilter: attemptAuthentication() Helper Method

We can use the Jackson Mapper to easily unmarshal the POST payload into DTO form any rethrown any failed parsing as a BadCredentialsException. Unfortunately for debugging, the default

401/Unauthorized response to the caller does not provide details we supply here but I guess that is a good thing when dealing with credentials and login attempts.

JwtAuthenticationFilter attemptAuthentication() Helper Method

```
...
import com.fasterxml.jackson.databind.ObjectMapper;
...
protected LoginDTO getCredentials(HttpServletRequest request) throws
AuthenticationException {
    try {
        return new ObjectMapper().readValue(request.getInputStream(), LoginDTO.class);
    } catch (IOException ex) {
        log.info("error parsing loginDTO", ex);
        throw new BadCredentialsException(ex.getMessage()); ①
    }
}
```

① `BadCredentialsException` extends `AuthenticationException`

8.6. JwtAuthenticationFilter: successfulAuthentication()

The `successfulAuthentication()` is called when authentication was successful. It has two primary jobs: encode the authenticated result in a JWS token and set the value in the response header.

JwtAuthenticationFilter successfulAuthentication()

```
@Override
protected void successfulAuthentication(
        HttpServletRequest request, HttpServletResponse response, FilterChain chain,
        Authentication authResult) throws IOException, ServletException {

    String token = jwtUtil.generateToken(authResult); ①
    log.info("generated token={}", token);
    jwtUtil.setToken(response, token); ②
}
```

① authResult represented within the claims of the JWS

② caller given the JWS token in the response header

This callback fully overrides the parent method to eliminate setting the `SecurityContext` and issuing a redirect. Neither have relevance in this situation. The authenticated caller will not require a `SecurityContext` now — this is the login. The `SecurityContext` will be set as part of the call to the operation.

Chapter 9. JwtAuthorizationFilter

The `JwtAuthorizationFilter` is responsible for realizing any provided JWS bearer tokens as an authResult within the current `SecurityContext` on the way to invoking an operation. The following end-to-end operation call shows the caller supplying the bearer token in order to identity themselves to the server implementing the operation. The example operation uses the username of the current `SecurityContext` as a key to locate information for the caller.

Example Operation Call with JWS Bearer Token

```
$ curl -v -X POST http://localhost:8080/api/carts/items?name=thing \
-H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk0OTgwMTAyLCJleHAiOjE4OTQ5ODM3M
DIsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.u2MmzTxaDoVNFGGCnrAcWBusS_NS2N
ndZXkaT964hLgcDTvCYAW_sXtTxRw8g_13"
> POST /api/carts/items?name=thing HTTP/1.1
...
< HTTP/1.1 200
{"username":"frasier","items":["thing"]} ① ②
```

① username is encoded within the JWS token

② cart with items is found by username

The `JwtAuthorizationFilter` did not seem to match any of the Spring-provided authentication filters — so I directly extended a generic filter support class that assures it will only get called once per request.

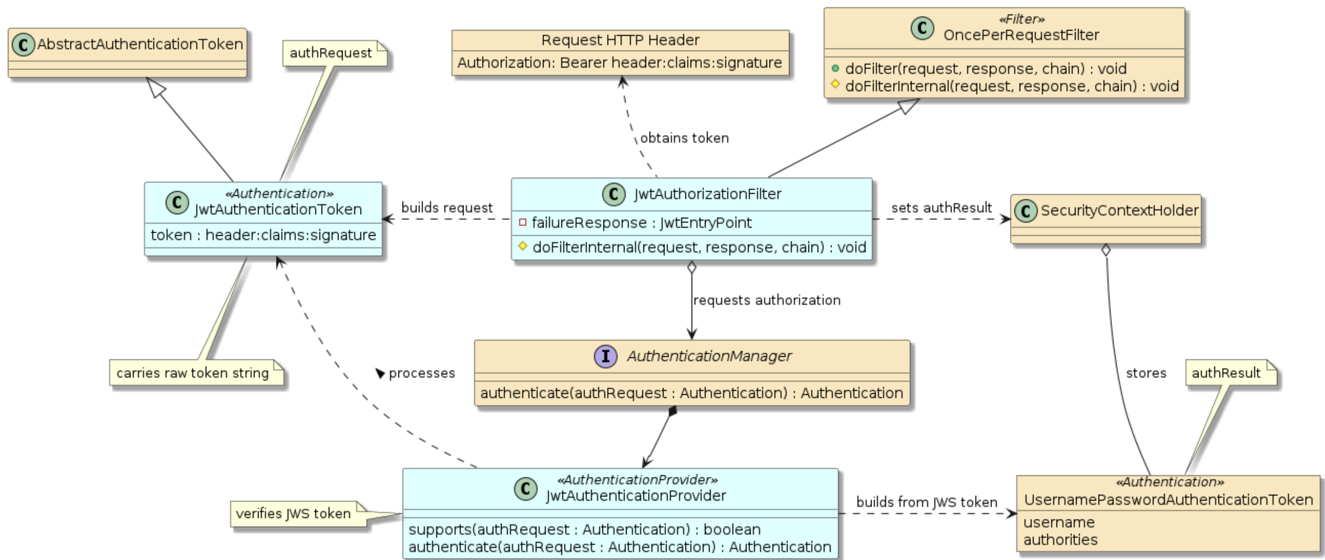
This class also relies on JwtUtil to implement the details of working with the JWS bearer token

JwtAuthorizationFilter

```java
public class JwtAuthorizationFilter extends OncePerRequestFilter {
    private final JwtUtil jwtUtil;
    private final AuthenticationManager authenticationManager;
    private final AuthenticationEntryPoint failureResponse = new JwtEntryPoint();
```

9.1. JwtAuthorizationFilter Relationships

The `JwtAuthorizationFilter` extends the generic framework of `OncePerRequestFilter` and performs all of its work in the `doFilterInternal()` callback.



The `JwtAuthenticationFilter` obtains the raw JWS token from the request header, wraps the token in the `JwsAuthenticationToken` authRequest and requests authentication from the `AuthenticationManager`. Placing this behavior in an `AuthenticationProvider` was optional but seemed to be consistent with the framework. It also provided the opportunity to lookup further user details if ever required.

Supporting the `AuthenticationManager` is the `JwtAuthenticationProvider`, which verifies the JWS token and re-builds the authResult from the JWS token claims.

The filter finishes by setting the authResult in the `SecurityContext` prior to advancing the chain further towards the operation call.

9.2. JwtAuthorizationFilter: Constructor

The `JwtAuthorizationFilter` relies on the `JwtUtil` helper class to implement the meat of the JWS token details. It also accepts an `AuthenticationManager` that is assumed to be populated with the `JwtAuthenticationProvider`.

JwtAuthorizationFilter Constructor

```
public JwtAuthorizationFilter(JwtConfig jwtConfig, AuthenticationManager
authenticationManager) {
    jwtUtil = new JwtUtil(jwtConfig);
    this.authenticationManager = authenticationManager;
}
```

9.3. JwtAuthorizationFilter: doFilterInternal()

Like most filters the `JwtAuthorizationFilter` initially determines if there is anything to do. If there is no `Authorization` header with a "Bearer " token, the filter is quietly bypassed and the filter chain is advanced.

If a token is found, we request authentication — where the JWS token is verified and converted

back into an `Authentication` object to store in the `SecurityContext` as the authResult.

Any failure to complete authentication when the token is present in the header will result in the chain terminating and an error status returned to the caller.

JwtAuthorizationFilter doFilterInternal()

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

    String token = jwtUtil.getToken(request);
    if (token == null) { //continue on without JWS authn/authz
        filterChain.doFilter(request, response); ①
        return;
    }

    try {
        Authentication authentication = new JwtAuthenticationToken(token); ②
        Authentication authenticated = authenticationManager.authenticate
(authentication);
        SecurityContextHolder.getContext().setAuthentication(authenticated); ③
        filterChain.doFilter(request, response); //continue chain to operation ④
    } catch (AuthenticationException fail) {
        failureResponse.commence(request, response, fail); ⑤
        return; //end the chain and return error to caller
    }
}
```

① chain is quietly advanced forward if there is no token found in the request header

② simple authRequest wrapper for the token

③ store the authenticated user in the `SecurityContext`

④ continue the chain with the authenticated user now present in the `SecurityContext`

⑤ issue an error response if token is present but we are unable to complete authentication

9.4. JwtAuthenticationToken

The `JwtAuthenticationToken` has a simple job — carry the raw JWS token string through the authentication process and be able to provide it to the `JwtAuthenticationProvider`. I am not sure whether I gained much by extending the `AbstractAuthenticationToken`. The primary requirement was to implement the `Authentication` interface. As you can see, the implementation simply carries the value and returns it for just about every question asked. It will be the job of `JwtAuthenticationProvider` to turn that token into an `Authentication` instance that represents the authResult, carrying authorities and other properties that have more exposed details.

JwtAuthenticationToken Class

```
public class JwtAuthenticationToken extends AbstractAuthenticationToken {
    private final String token;
    public JwtAuthenticationToken(String token) {
        super(Collections.emptyList());
        this.token = token;
    }
    public String getToken() {
        return token;
    }
    @Override
    public Object getCredentials() {
        return token;
    }
    @Override
    public Object getPrincipal() {
        return token;
    }
}
```

The `JwtAuthenticationProvider` class implements two key methods: `supports()` and `authenticate()`

JwtAuthenticationProvider Class

```
public class JwtAuthenticationProvider implements AuthenticationProvider {
    private final JwtUtil jwtUtil;
    public JwtAuthenticationProvider(JwtConfig jwtConfig) {
        jwtUtil = new JwtUtil(jwtConfig);
    }
    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
    @Override
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        try {
            String token = ((JwtAuthenticationToken)authentication).getToken();
            Authentication authResult = jwtUtil.parseToken(token);
            return authResult;
        } catch (JwtException ex) {
            throw new BadCredentialsException(ex.getMessage());
        }
    }
}
```

The `supports()` method returns true only if the token type is the `JwtAuthenticationToken` type.

The `authenticate()` method obtains the raw token value, confirms its validity, and builds an

`Authentication` authResult from its claims. The result is simply returned to the `AuthenticationManager` and the calling filter.

Any error in `authenticate()` will result in an `AuthenticationException`. The most likely is an expired token — but could also be the result of a munged token string.

9.5. JwtEntryPoint

The `JwtEntryPoint` class implements an `AuthenticationEntryPoint` interface that is used elsewhere in the framework for cases when an error handler is needed because of an `AuthenticationException`. We are using it within the `JwtAuthorizationProvider` to report an error with authentication — but you will also see it show up elsewhere.

JwtEntryPoint

```
package info.ejava.examples.svc.auth.cart.security.jwt;

import org.springframework.http.HttpStatus;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

public class JwtEntryPoint implements AuthenticationEntryPoint {
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
                         AuthenticationException authException) throws IOException {
        response.sendError(HttpStatus.UNAUTHORIZED.value(), authException.getMessage(
));
    }
}
```

Chapter 10. API Security Configuration

With all the supporting framework classes in place, I will now show how we can wire this up. This, of course, takes us back to the `WebSecurityConfigurer` class.

- We inject required beans into the configuration class. The only thing that is new is the `JwtConfig` `@ConfigurationProperties` class. The `UserDetailsService` provides users/passwords and authorities from a database
- `configure(HttpSecurity)` is where we setup our `FilterChainProxy`
- `configure(AuthenticationManagerBuilder)` is where we setup our `AuthenticationManager` used by our filters in the `FilterChainProxy`.

API Security Configuration

```
@Configuration
@Order(0)
@RequiredArgsConstructor
@EnableConfigurationProperties(JwtConfig.class) ①
public class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final JwtConfig jwtConfig; ②
    private final UserDetailsService jdbcUserDetailsService; ③

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // details here ...
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //details here ...
    }
```

① enabling the `JwtConfig` as a `@ConfigurationProperties` bean

② injecting the `JwtConfig` bean into out configuration class

③ injecting a source of user details (i.e., username/password and authorities)

10.1. API Authentication Manager Builder

The `configure(AuthenticationManagerBuilder)` configures the builder with two `AuthenticationProviders`

- one containing real users/passwords and authorities
- a second with the ability to instantiate an `Authentication` from a JWS token

API Authentication Manager Builder

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```
    auth.userDetailsService(jdbcUserDetailsService); ①
    auth.authenticationProvider(new JwtAuthenticationProvider(jwtConfig));
}
```

① configuring an `AuthenticationManager` with both the `UserDetailsService` and our new `JwtAuthenticationProvider`

The `UserDetailsService` was injected because it required setup elsewhere. However, the `JwtAuthenticationProvider` is stateless — getting everything it needs from a startup configuration and the authentication calls.

10.2. API HttpSecurity Key JWS Parts

The following snippet shows the key parts to wire in the JWS handling.

- we register the `JwtAuthenticationFilter` to handle authentication of logins
- we register the `JwtAuthorizationFilter` to handle restoring the `SecurityContext` when the caller presents a valid JWS bearer token
- not required — but we register a custom error handler that leaks some details about why the caller is being rejected when receiving a 403/Forbidden

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    //...
    http.addFilterAt(new JwtAuthenticationFilter(jwtConfig, ①
            authenticationManager()),
            UsernamePasswordAuthenticationFilter.class);
    http.addFilterAfter(new JwtAuthorizationFilter(jwtConfig, ②
            authenticationManager()),
            JwtAuthenticationFilter.class);
    http.exceptionHandling(cfg->cfg.defaultAuthenticationEntryPointFor( ③
                    new JwtEntryPoint(),
                    new AntPathRequestMatcher("/api/**")));

    http.authorizeRequests(cfg->cfg.antMatchers("/api/login").permitAll());
    http.authorizeRequests(cfg->cfg.antMatchers("/api/carts/**").authenticated());
}
```

① `JwtAuthenticationFilter` being registered at location normally used for `UsernamePasswordAuthenticationFilter`

② `JwtAuthorizationFilter` being registered after the authn filter

③ adding an optional error reporter

10.3. API HttpSecurity Full Details

The following shows the full contents of the `configure(HttpSecurity)` method. In this view you can see how FORM and BASIC Auth have been disabled and we are operating in a stateless mode with

various header/CORS options enabled.

API HttpSecurity Full Details

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.requestMatchers(m->m.antMatchers("/api/**"));
    http.httpBasic(cfg->cfg.disable());
    http.formLogin(cfg->cfg.disable());
    http.headers(cfg->{
        cfg.xssProtection().disable();
        cfg.frameOptions().disable();
    });
    http.csrf(cfg->cfg.disable());
    http.cors();
    http.sessionManagement(cfg->cfg
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.addFilterAt(new JwtAuthenticationFilter(jwtConfig,
                    authenticationManager()),
            UsernamePasswordAuthenticationFilter.class);
    http.addFilterAfter(new JwtAuthorizationFilter(jwtConfig,
                    authenticationManager()),
            JwtAuthenticationFilter.class);
    http.exceptionHandling(cfg->cfg.defaultAuthenticationEntryPointFor(
                    new JwtEntryPoint(),
                    new AntPathRequestMatcher("/api/**")));

    http.authorizeRequests(cfg->cfg.antMatchers("/api/login").permitAll());
    http.authorizeRequests(cfg->cfg.antMatchers("/api/whoami").permitAll());
    http.authorizeRequests(cfg->cfg.antMatchers("/api/carts/**").authenticated());
    }
```

Chapter 11. Example JWT/JWS Application

Now that we have thoroughly covered the addition of the JWT/JWS to the security framework of our application, it is time to look at the application and with a focus on authorizations. I have added a few unique aspects since the previous lecture's example use of `@PreAuthorize`.

- we are using JWT/JWS — of course
- access annotations are applied to the service interface versus controller class
- access annotations inspect the values of the input parameters

11.1. Roles and Role Inheritance

I have reused the same users, passwords, and role assignments from the authorities example and will demonstrate with the following users.

- ROLE_ADMIN - `sam`
- ROLE_CLERK - `woody`
- ROLE_CUSTOMER - `norm` and `frasier`

However, role inheritance is only defined for ROLE_ADMIN inheriting all accesses from ROLE_CLERK. None of the roles inherit from ROLE_CUSTOMER.

Role Inheritance

```
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(Arrays.asList(
            "ROLE_ADMIN > ROLE_CLERK"),System.lineSeparator()));
    return roleHierarchy;
}
```

11.2. CartsService

We have a simple CartsService with a Web API and service implementation. The code below shows the interface to the service. It has been annotated with `@PreAuthorize` expressions that use the Spring Expression Language to evaluate the principal from the SecurityContext and parameters of the call.

CartsService

```
package info.ejava.examples.svc.auth.cart.services;

import info.ejava.examples.svc.auth.cart.dto.CartDTO;
import org.springframework.security.access.prepost.PreAuthorize;

```

```
public interface CartsService {

    @PreAuthorize("#username == authentication.name and hasRole('CUSTOMER')") ①
    CartDTO createCart(String username);

    @PreAuthorize("#username == authentication.name or hasRole('CLERK')") ②
    CartDTO getCart(String username);

    @PreAuthorize("#username == authentication.name") ③
    CartDTO addItem(String username, String item);

    @PreAuthorize("#username == authentication.name or hasRole('ADMIN')") ④
    boolean removeCart(String username);
}
```

① anyone with the `CUSTOMER` role can create a cart but it must be for their username

② anyone can get their own cart and anyone with the `CLERK` role can get anyone's cart

③ users can only add item to their own cart

④ users can remove their own cart and anyone with the `ADMIN` role can remove anyone's cart

11.3. Login

The following shows creation of tokens for four example users

Sam

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"sam",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiY
XV0aCI6WyJST0xFX0FETUlOIl19.ICzAn1r2UyrpGJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6U
it-cb4
```

① `sam` has role `ADMIN` and inherits role `CLERK`

Woody

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"woody",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai
```

① woody has role CLERK

Norm and Frasier

```
$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"norm",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJliOtZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw

$ curl -v -X POST http://localhost:8080/api/login -d '{"username":"frasier",
"password":"password"}' ①
> POST /api/login HTTP/1.1
< HTTP/1.1 200
< Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2N
zEsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ik
uir9CJPdOT8Ow2lI5Z1GQY6ZaKvW883txI
```

① norm and frasier have role CUSTOMER

11.4. createCart()

The access rules for createCart() require the caller be a customer and be creating a cart for their username.

createCart() Access Rules

```
@PreAuthorize("#username == authentication.name and hasRole('CUSTOMER')") ①
CartDTO createCart(String username); ①
```

① #username refers to the username method parameter

Woody is unable to create a cart because he lacks the CUSTOMER role.

Woody Unable to Create Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai" #woody
[woody, [ROLE_CLERK]]

$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai" #woody
```

```
{"url":"http://localhost:8080/api/carts","message":"Forbidden","description":"caller[w
oody] is forbidden from making this request","timestamp":"2020-07-
17T20:24:14.159507Z"}
```

Norm is able to create a cart because he has the `CUSTOMER` role.

Norm Can Create Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw" #norm
[norm, [ROLE_CUSTOMER]]

$ curl -X POST http://localhost:8080/api/carts -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJli0tZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw" #norm
{"username":"norm","items":[]}
```

11.5. addItem()

The `addItem()` access rules only allow users to add items to their own cart.

addItem() Access Rules

```
@PreAuthorize("#username == authentication.name")
CartDTO addItem(String username, String item);
```

Frasier is forbidden from adding items to Norm's cart because his identity does not match the username for the cart.

Frasier Cannot Add to Norms Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2N
zEsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ik
uir9CJPdOT8Ow2lI5Z1GQY6ZaKvW883txI" #frasier
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl -X POST "http://localhost:8080/api/carts/items?username=norm&name=chardonnay"
-H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2N
zEsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ik
uir9CJPdOT8Ow2lI5Z1GQY6ZaKvW883txI" #frasier
{"url":"http://localhost:8080/api/carts/items?username=norm&name=chardonnay","message"
:"Forbidden","description":"caller[frasier] is forbidden from making this
request","timestamp":"2020-07-17T20:40:10.451578Z"} ①
```

① frasier received a 403/Forbidden error when attempting to add to someone else's cart

Norm can add items to his own cart because his username matches the username of the cart.

Norm Can Add to His Own Cart

```
$ curl -X POST http://localhost:8080/api/carts/items?name=beer -H "Authorization:
Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJliOtZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw" #norm
{"username":"norm","items":["beer"]}
```

11.6. getCart()

The getCart() access rules only allow users to get their own cart, but also allows users with the CLERK role to get anyone's cart.

getCart() Access Rules

```
@PreAuthorize("#username == authentication.name or hasRole('CLERK')") ②
CartDTO getCart(String username);
```

Frasier cannot get Norm's cart because anyone lacking the CLERK role can only get a cart that matches their authenticated username.

Frasier Cannot Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJmcmFzaWVyIiwiaWF0IjoxNTk1MDE3MDcxLCJleHAiOjE4OTUwMjA2N
zEsImF1dGgiOlsiUFJJQ0VfQ0hFQ0siLCJST0xFX0NVU1RPTUVSIl19.ELAe5foIL_u2QyhpjwDoqQbL4Hl1Ik
uir9CJPdOT8Ow2lI5Z1GQY6ZaKvW883txI" #frasier
{"url":"http://localhost:8080/api/carts?username=norm","message":"Forbidden","descript
ion":"caller[frasier] is forbidden from making this request","timestamp":"2020-07-
17T20:44:05.899192Z"}
```

Norm can get his own cart because the username of the cart matches the authenticated username of his accessing the cart.

Norm Can Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJliOtZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw" #norm
{"username":"norm","items":["beer"]}
```

Woody can get Norm's cart because he has the CLERK role.

Woody Can Get Norms Cart

```
$ curl -X GET http://localhost:8080/api/carts?username=norm -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai" #woody
{"username":"norm","items":["beer"]}
```

11.7. removeCart()

The `removeCart()` access rules only allow carts to be removed by their owner or by someone with the `ADMIN` role.

removeCart() Access Rules

```
@PreAuthorize("#username == authentication.name or hasRole('ADMIN')")
boolean removeCart(String username);
```

Woody cannot remove Norm's cart because his authenticated username does not match the cart and he lacks the `ADMIN` role.

Woody Cannot Remove Norms Cart

```
$ curl -X DELETE http://localhost:8080/api/carts?username=norm -H "Authorization:
Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJ3b29keSIsImlhdCI6MTU5NTAxNzA1MSwiZXhwIjoxODk1MDIwNjUxL
CJhdXRoIjpbIlJPTEVfQ0xFUksiXX0.kreSFPgTIr2heGMLcjHFrglydvhPZKR7Iy4F6b76WNIvAkbZVhfymbQ
xekuPL-Ai" #woody
{"url":"http://localhost:8080/api/carts?username=norm","message":"Forbidden","descript
ion":"caller[woody] is forbidden from making this request","timestamp":"2020-07-
17T20:48:40.866193Z"}
```

Sam can remove Norm's cart because he has the `ADMIN` role. Once Same deletes the cart, Norm receives a 404/Not Found because it is not longer there.

Sam Can Remove Norms Cart

```
$ curl -X GET http://localhost:8080/api/whoAmI -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiY
XV0aCI6WyJST0xFX0FETUlOIl19.ICzAn1r2UyrpGJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6U
it-cb4" #sam
[sam, [ROLE_ADMIN]]

$ curl -X DELETE http://localhost:8080/api/carts?username=norm -H "Authorization:
Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJzYW0iLCJpYXQiOjE1OTUwMTcwNDQsImV4cCI6MTg5NTAyMDY0NCwiY
XV0aCI6WyJST0xFX0FETUlOIl19.ICzAn1r2UyrpGJQSYk9uqxMAAq9QC1Dw7GKe0NiGvCyTasMfWSStrqxV6U
it-cb4" #sam
```

```
$ curl -X GET http://localhost:8080/api/carts -H "Authorization: Bearer
eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJub3JtIiwiaWF0IjoxNTk1MDE3MDY1LCJleHAiOjE4OTUwMjA2NjUsI
mF1dGgiOlsiUk9MRV9DVVNUT01FUiJdfQ.UX4yPDu0LzWdEAObbJliOtZ7ePU1RSIH_o_hayPrlmNxhjU5DL6X
Q42iRCLLuFgw" #norm
{"url":"http://localhost:8080/api/carts","message":"Not Found","description":"no cart
found for norm","timestamp":"2020-07-17T20:50:59.465210Z"}
```

Chapter 12. Summary

I don’t know about you — but I had fun with that!

To summarize — in this module we learned:

- to separate the authentication from the operation call such that the operation call could be in a separate server or even an entirely different service
- what is a JSON Web Token (JWT) and JSON Web Secret (JWS)
- how trust is verified using JWS
- how to write and/or integrate custom authentication and authorization framework classes to implement an alternate security mechanism in Spring/Spring Boot
- how to leverage Spring Expression Language to evaluate parameters and properties of the `SecurityContext`