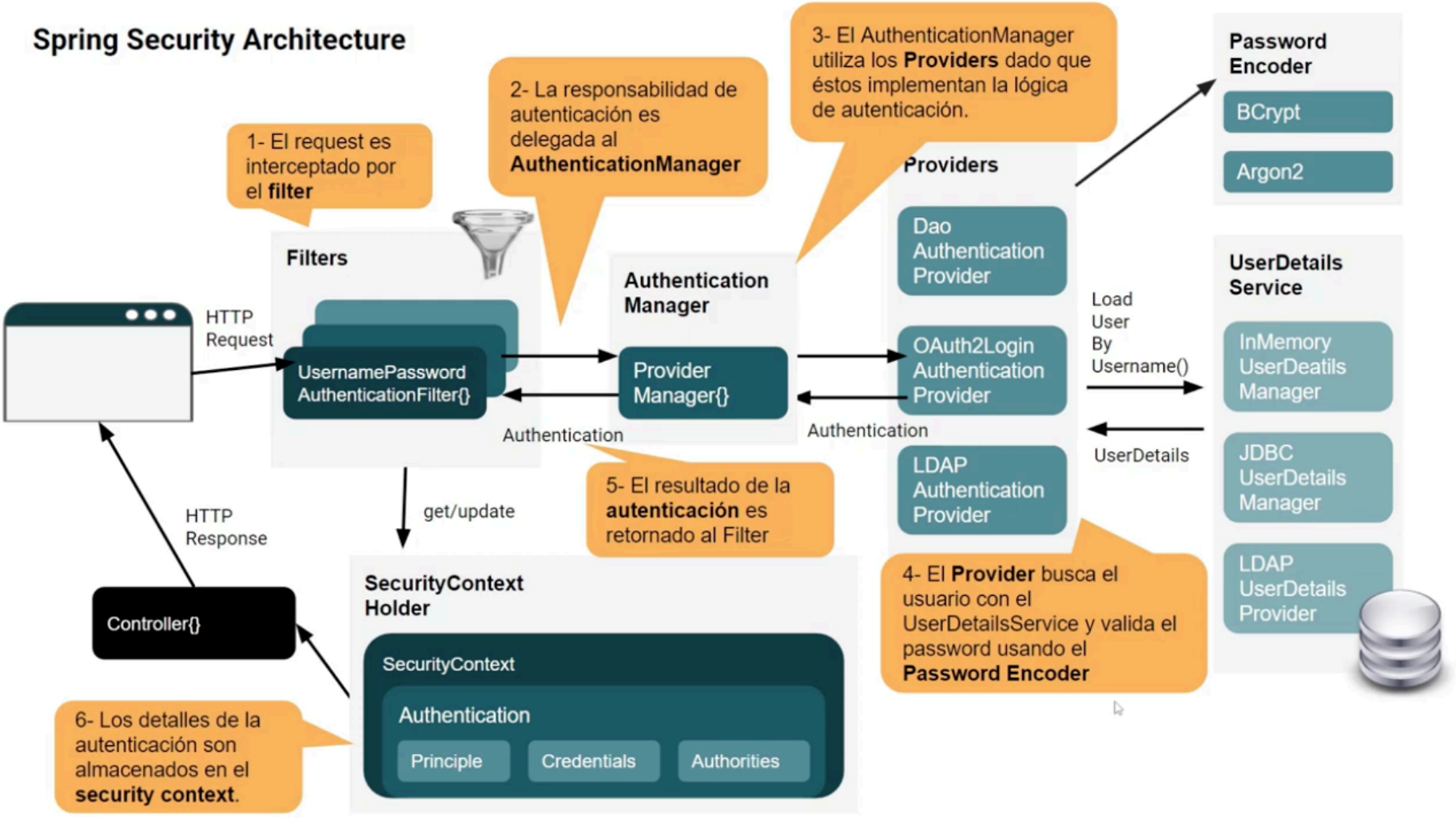
Spring Security

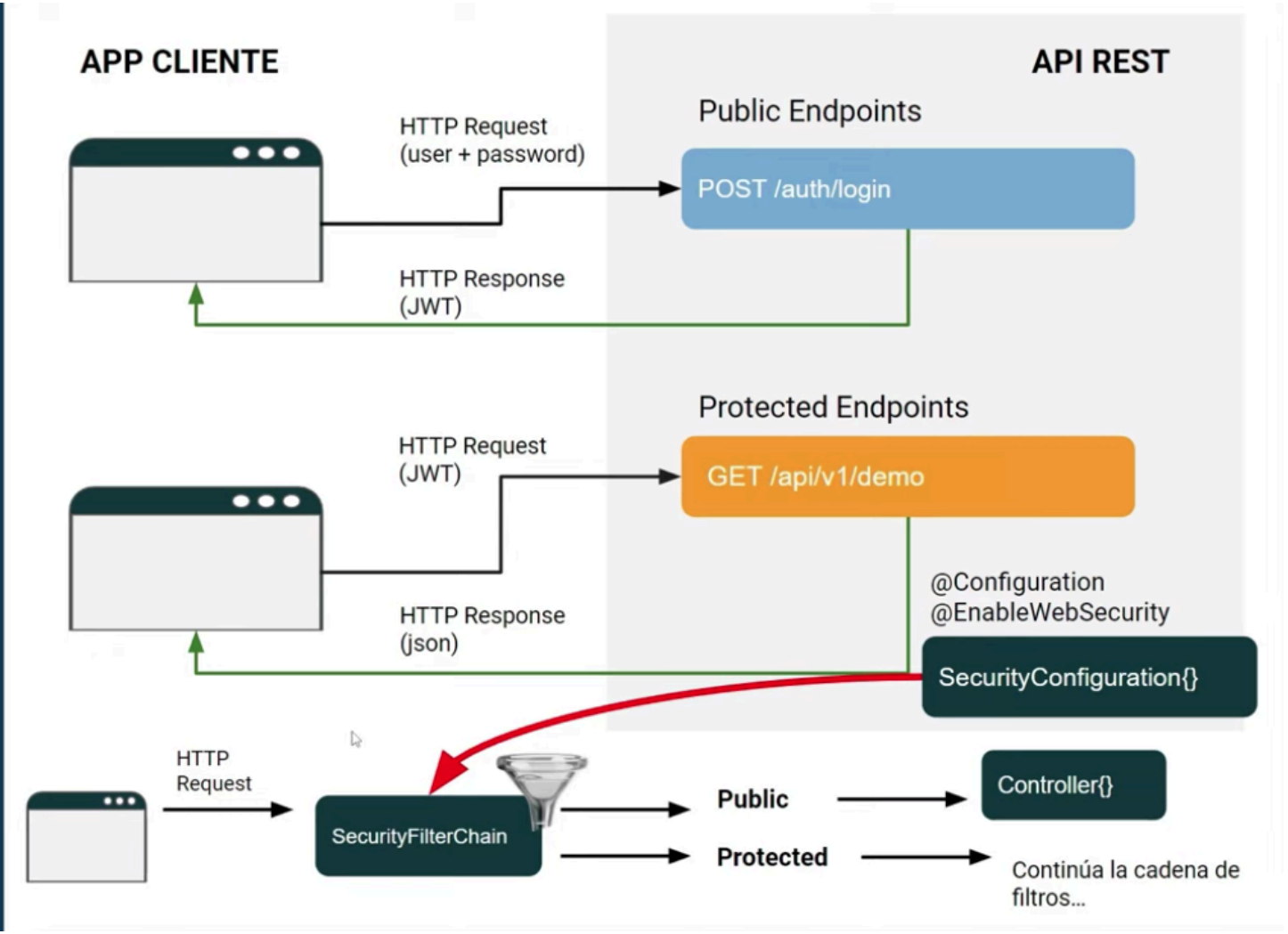
Cuadros sinópticos del flujo de trabajo de Spring Security, con sus respectivos filtros y herramientas varias utilizadas para la autenticación y autorización de los usuarios.

Imágenes extraídas de los video tutoriales en YouTube de Ivana Soledad Rojas Córsico:

- Video 1: https://www.youtube.com/watch?v=nwqQYCM4YT8&list=PLx89vzy-Ta0qIf_swgzVY4z2PcqeGDduO&index=5
- Video 2: https://www.youtube.com/watch?v=Yrj10XGGiF4&list=PLx89vzy-Ta0qIf_swgzVY4z2PcqeGDduO&index=5

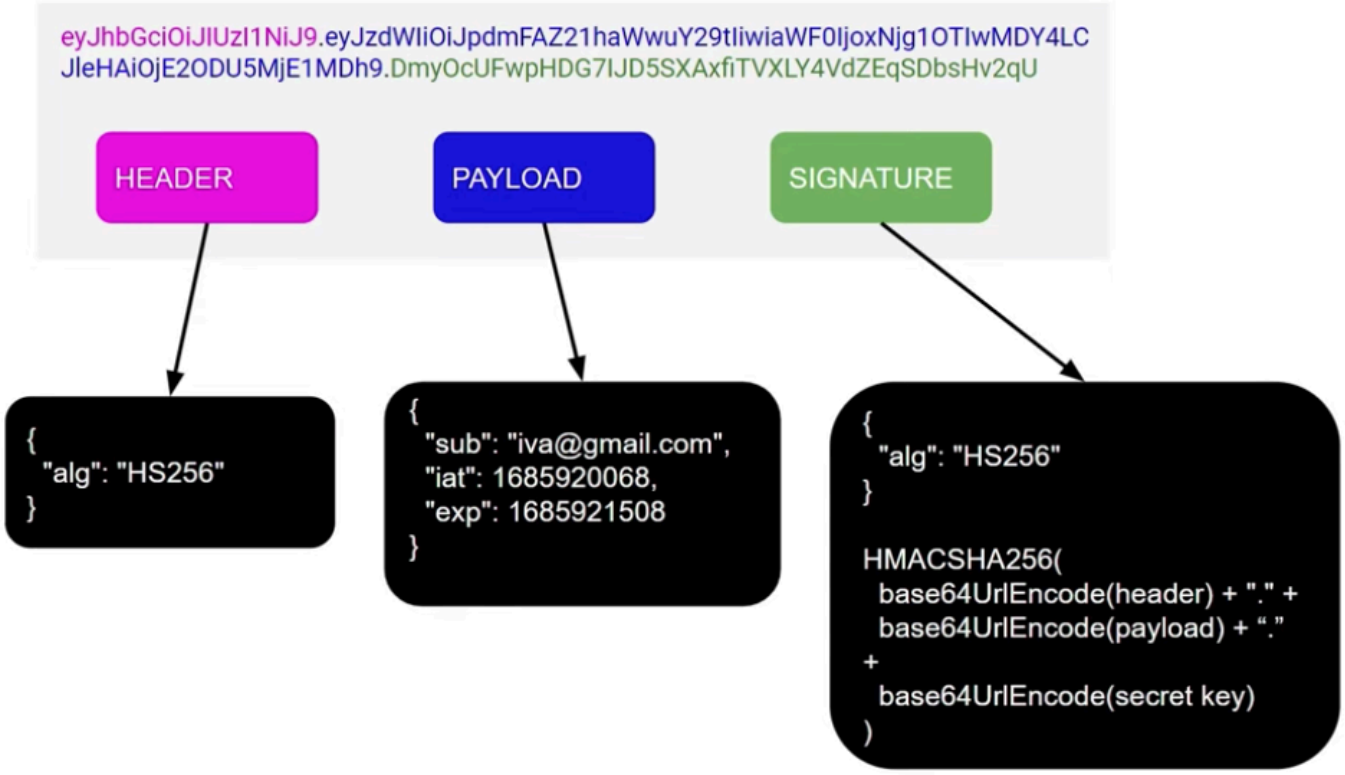

Spring Security Architecture
1- El request es interceptado por el filter
2- La responsabilidad de autenticación es delegada al AuthenticationManager
3- El AuthenticationManager utiliza los Providers dado que éstos implementan la lógica de autenticación.
Password Encoder
BCrypt
Argon2
Filters
Authentication Manager
Providers
Dao Authentication Provider
UserDetails Service
HTTP Request
UsernamePassword AuthenticationFilter{}
Provider Manager{}
OAuth2Login Authentication Provider
Load User By Username()
InMemory UserDeatils Manager
Authentication
Authentication
UserDetails
LDAP Authentication Provider
JDBC UserDetails Manager
LDAP UserDetails Provider
5- El resultado de la autenticación es retornado al Filter
HTTP Response
get/update
4- El Provider busca el usuario con el UserDetailsService y valida el password usando el Password Encoder
SecurityContext Holder
Controller{}
SecurityContext
Authentication
Principle
Credentials
Authorities
6- Los detalles de la autenticación son almacenados en el security context.

Proceso de Autenticación basado en JWT

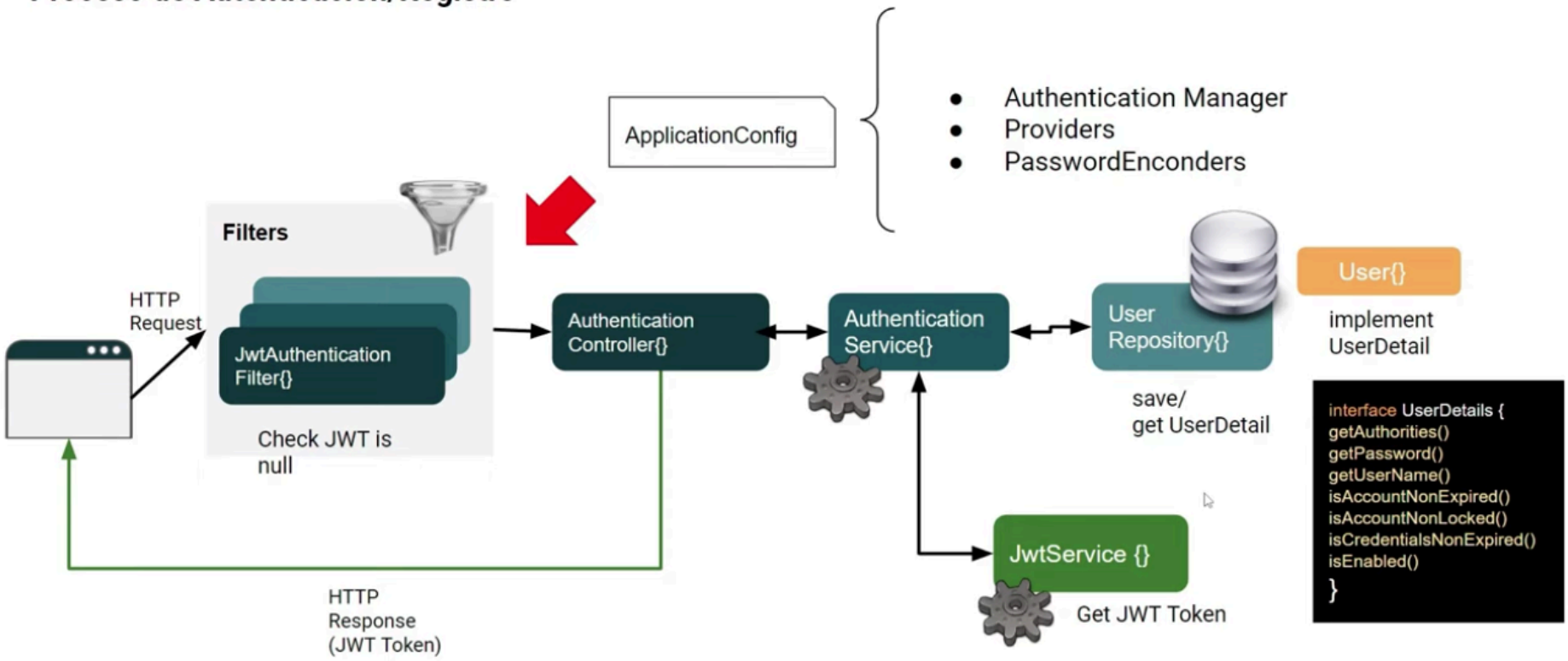


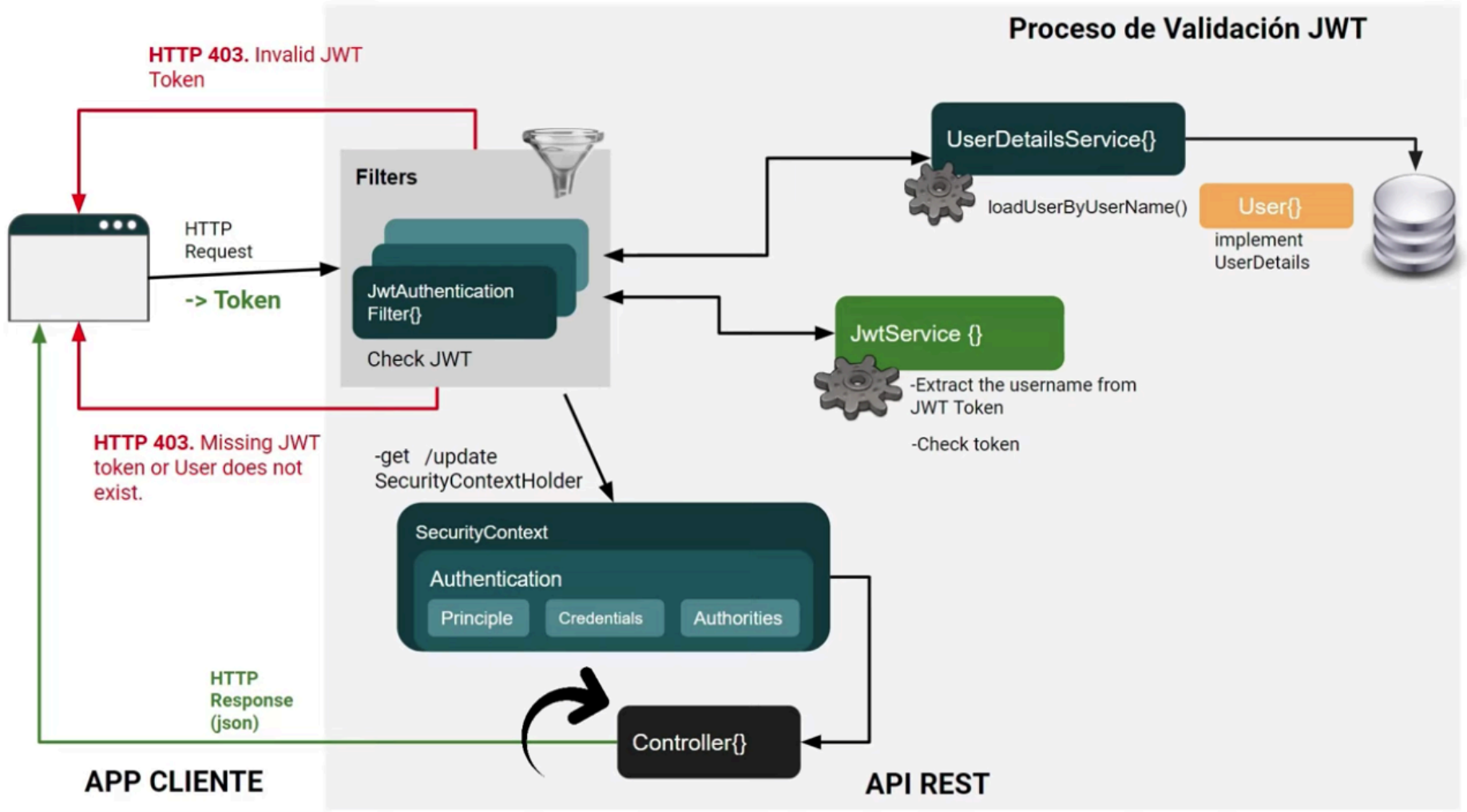
Estructura de un JWT

Estructura de un JWT (JSON Web Token)



Proceso de Autenticación/Registro




Proceso de Validación JWT
HTTP 403. Invalid JWT Token
Filters
UserDetailsService{}
loadUserByUserName()
User{}
implement UserDetails
HTTP Request
-> Token
JwtAuthentication Filter{}
Check JWT
JwtService {}
-Extract the username from JWT Token
-Check token
HTTP 403. Missing JWT token or User does not exist.
-get /update SecurityContextHolder
SecurityContext
Authentication
Principle
Credentials
Authorities
HTTP Response (json)
Controller{}
APP CLIENTE
API REST

io.jsonwebtoken

Crear un **JWT**

1. Utilizar el método `Jwts.builder()` para crear una instancia `JwtBuilder`.
2. Configurar el header y payload llamando a los métodos que nos provee `JwtBuilder`.
3. Firmar el JWT. El forma formato de la key debe ser `SecretKey` o, bien el asimétrico `PrivateKey`.
4. Finalmente, llamar al método `compact()` para compactar/comprimir y firmar, generando los jws finales.

Ejemplo:

```
return Jwts
        .builder()
        .claims(extraClaims)
        .subject(user.getUsername())
        .issuedAt(new Date(System.currentTimeMillis()))
        .expiration(new Date(System.currentTimeMillis()+1000*60*24*60))
        .signWith(getKey())
        .compact();
```

io.jsonwebtoken

JWT **Header**

El `JWT Header` es un objeto que proporciona metadata sobre el contenido, el formato y el algoritmo de la firma.

El mismo se puede, configurar utilizando el `JwtBuilder` de dos maneras: mediante múltiples parámetros o bien de manera individual mediante pares nombre/valor.

```
String jwt = Jwts.builder()
    .header()
        .add("someProperty", value)
        // ...
     .and() // return to the JwtBuilder
       // ...
```

Custom Header Parameters

```
String jwt = Jwts.builder()
    .header()
        .add(multipleHeaderParamsMap)
        //
     .and() // return to the JwtBuilder
       //
```

Header Parameter Map

Nota: No es necesario configurar los encabezados, la librería siempre los configura automáticamente según sea necesario.

io.jsonwebtoken

JWT **Payload**

El JWT Payload puede ser cualquier cosa que pueda representarse como un array de bytes o como un json.

El mismo admite dos opciones:

- **Content**. Si el contenido es un array de bytes. Ej. una imagen, un documento.
- **Claims** . Si el contenido es un JSON Claim. (pares nombre/valor)

io.jsonwebtoken

JWT **Standar** Claims

1. **iss (issuer):** entidad que emite el token.
2. **sub (subject):** usuario.
3. **aud (audience):** aplicación o servicio al cual está destinado el token.
4. **exp (expiration):** fecha y hora de expiración del token.
5. **iat(issuer at):** fecha y hora de creación.
6. **nbf (not before):** fecha y hora en que el token deben comenzar a ser aceptado para su procesamiento.
7. **jti (jwt id):** identificador único del token.

```
String jwt = Jwts.builder()
          .subject(user.getUsername())
          .issuedAt(new Date(System.currentTimeMillis()))
          .expiration(new Date(System.currentTimeMillis()+1000*60*24*60))
          …
```

io.jsonwebtoken

JWT **Custom** Claims

Es posible personalizar claims si no se adaptan a los estándar que nos proporciona la librería.

```
String jwt = Jwts.builder()
    .claim("userId", user.getId())
    .claim("user", user)
```

```
Map<String, Object> extraClaims= new HashMap<>();
        extraClaims.put("userId", user.getId());
        extraClaims.put("user", user);

String jwt = Jwts.builder()
       .claims(extraClaims)
```

io.jsonwebtoken

Signed JWT

- Garantiza la autenticidad.
- Garantiza que no fue manipulado luego de su creación (integridad).

Todos estos están representados como constantes en la `io.jsonwebtoken.Jwts.SIG` clase de registro.

Identificador	Algoritmo de firma
HS256	HMAC usando SHA-256
HS384	HMAC usando SHA-384
HS512	HMAC usando SHA-512
ES256	ECDSA usando P-256 y SHA-256
ES384	ECDSA usando P-384 y SHA-384
ES512	ECDSA usando P-521 y SHA-512
RS256	RSASSA-PKCS-v1_5 usando SHA-256
RS384	RSASSA-PKCS-v1_5 usando SHA-384
RS512	RSASSA-PKCS-v1_5 usando SHA-512
PS256	RSASSA-PSS usando SHA-256 y MGF1 con SHA-256 1
PS384	RSASSA-PSS usando SHA-384 y MGF1 con SHA-384 1
PS512	RSASSA-PSS usando SHA-512 y MGF1 con SHA-512 1
EdDSA	Algoritmo de firma digital Edwards-Curve (EdDSA) 2