MicroProfile - Quarkus pt. 2

Leon Schloemmer

Course

- **User Endpoint** - creating a user and securely storing passwords
- **JWT** - generate a jwt which grants access to secured resources
- **JWT RBAC** - accessing secured resources
- **Native**
- **Docker**

Goal

- Create an application where users can register themselves
- Using JWT

Showcase - finished Application

Livecoding

- Repo to code along: https://github.com/leonschloemmer/quarkus-livecoding

Hashing the password pt. 1

```
26      @POST
27  @   public Response createUser(UserInterface ui) {
28          String hashedPassword;
29          try {
30              hashedPassword = PasswordSecurityUtils.generatePasswordHash(ui.getPassword());
31          } catch (NoSuchAlgorithmException e) {
32              return Response.serverError().build();
33          } catch (InvalidKeySpecException e) {
34              return Response.serverError().build();
35          }
```

Why hash a password?

- “cleartext” passwords are vulnerable to attacks, where attackers gain READ access on the User-database
- Hashing is irreversible - once hashed, you can’t get the original password back
- You can generate a hash from a password, but not vice versa

https://security.blogoverflow.com/2011/11/why-passwords-should-be-hashed/

The generatePasswordHash function

```
13  public class PasswordSecurityUtils {
14
15  @      private PasswordSecurityUtils() {}
16
17  @      public static String generatePasswordHash(String password)
18                 throws NoSuchAlgorithmException, InvalidKeySpecException {
19             int iterations = 1000;
20             char[] chars = password.toCharArray();
21             byte[] salt = getSalt();
22
23             PBEKeySpec spec = new PBEKeySpec(chars, salt, iterations, 64 * 8);
24             SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
25             byte[] hash = skf.generateSecret(spec).getEncoded();
26             return iterations + ":" + toHex(salt) + ":" + toHex(hash);
27         }
```

https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/

Adding “salts” to passwords

- Salts are randomness that are appended to a password before hashing
- This way, two users can have the same password, but different hashes are stored in our database
- The salt is then stored in cleartext in our database, so we can still validate the password

https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

getSalt()

```
47 @    private static byte[] getSalt() throws NoSuchAlgorithmException {
48          SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
49          byte[] salt = new byte[16];
50          sr.nextBytes(salt);
51          return salt;
52      }
```

implementing fromHex and toHex

```
54 @    private static String toHex(byte[] array) throws NoSuchAlgorithmException {
55          BigInteger bi = new BigInteger(1, array);
56          String hex = bi.toString(16);
57          int paddingLength = (array.length * 2) - hex.length();
58          if (paddingLength > 0) {
59              return String.format("%0" + paddingLength + "d", 0) + hex;
60          } else {
61              return hex;
62          }
63      }
64
65 @    private static byte[] fromHex(String hex) throws NoSuchAlgorithmException {
66          byte[] bytes = new byte[hex.length() / 2];
67          for (int i = 0; i < bytes.length; i++) {
68              bytes[i] = (byte)Integer.parseInt(hex.substring(2 * i, 2 * i + 2), 16);
69          }
70          return bytes;
71      }
```

Persistence

- Now we can persist the user

```
36
37          MyUser myUser = new MyUser(ui.getUsername(), hashedPassword);
38          userRepo.persistUser(myUser);
39
```

Generate JWT and return it

```
40          // return jwt to permit access to secure information
41          try {
42              String token = TokenUtils.generateTokenString(myUser.getId().toString());
43              JsonObjectBuilder builder = Json.createObjectBuilder();
44              builder.add("accessToken", token);
45              return Response.ok().entity(builder.build()).build();
46          } catch (Exception e) {
47              return Response.serverError().build();
48          }
49      }
```

Generating a JWT pt. 1

1

```
29      public static String generateTokenString(String uid)
30              throws Exception{
31          PrivateKey pk = readPrivateKey("/private.pem");
32          return generateTokenString(pk, "/private.pem", uid);
33      }
```

2

```
35      public static PrivateKey readPrivateKey(final String pemResName) throws Exception {
36          InputStream contentIS = TokenUtils.class.getResourceAsStream(pemResName);
37          byte[] tmp = new byte[4096];
38          int length = contentIS.read(tmp);
39          return decodePrivateKey(new String(tmp, 0, length, "UTF-8"));
40      }
```

3

```
85      public static int currentTimeInSecs() {
86          long currentTimeMS = System.currentTimeMillis();
87          return (int) (currentTimeMS / 1000);
88      }
```

Generating a JWT pt. 2

```
42      public static String generateTokenString(PrivateKey privateKey, String kid, String uid) throws Exception {
43  //          JwtClaims claims = JwtClaims.parse(readTokenContent(jsonResName));
44          JwtClaims claims = JwtClaims.parse(getJwtClaims(uid));
45          long currentTimeInSecs = currentTimeInSecs();
46  //          long exp = timeClaims != null && timeClaims.containsKey(Claims.exp.name())
47  //                  ? timeClaims.get(Claims.exp.name()) : currentTimeInSecs + 300; // 300 seconds until expires
48          long exp = currentTimeInSecs + 60*15; // 15 minutes until token expires
```

The token will expire in 15 minutes, this means, that holders of the token can access secured resources as long as the token isn’t expired

JWT Claims

```
72      private static String getJwtClaims(String uid) {
73          JsonObjectBuilder builder = Json.createObjectBuilder();
74          builder.add("iss", "https://this-is-totally-our.domain"); // The issuer of the token, us
75          builder.add("sub", uid); // sub = subject = the user that the token was created for
76
77          JsonArrayBuilder groupsBuilder = Json.createArrayBuilder();
78          groupsBuilder.add("defaultUsers");
79
80          builder.add("groups", groupsBuilder.build());
81
82          return builder.build().toString();
83      }
```

- Claims basically contain information about a *principal*
- User *claims* to be user with ID …
- User *claims* to be of group … This is needed so we can do Role-based authorisation

```
50          // setting claims for jwt
51          claims.setIssuedAt(NumericDate.fromSeconds(currentTimeInSecs));
52          claims.setClaim(Claims.auth_time.name(), NumericDate.fromSeconds(currentTimeInSecs));
53          claims.setExpirationTime(NumericDate.fromSeconds(exp));
54
55          JsonWebSignature jws = new JsonWebSignature();
56          jws.setPayload(claims.toJson());
57          jws.setKey(privateKey);
58          jws.setKeyIdHeaderValue(kid);
59          jws.setHeader("typ", "JWT");
60          jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.RSA_USING_SHA256);
61
62          return jws.getCompactSerialization();
63      }
```

- Using a private / public key pattern, we can ensure the legitimacy of the token

- We will now generate the keys

Generating Keys to encrypt JWT

- In the folder src/main/resources execute following 2 commands
- `% openssl genpkey -out private.pem -algorithm RSA -pkeyopt rsa_keygen_bits:2048`
- `% openssl rsa -in private.pem -outform PEM -pubout -out public.pem`
- This generates RSA keys using SHA-256 cryptography
- the private key **has to be kept confidential**

https://www.openssl.org/docs/man1.1.0/man1/genpkey.html

https://rietta.com/blog/openssl-generating-rsa-key-from-command/

- If another encryption algorithm is used, change the algorithm-header accordingly!

```
50          // setting claims for jwt
51          claims.setIssuedAt(NumericDate.fromSeconds(currentTimeInSecs));
52          claims.setClaim(Claims.auth_time.name(), NumericDate.fromSeconds(currentTimeInSecs));
53          claims.setExpirationTime(NumericDate.fromSeconds(exp));
54
55          JsonWebSignature jws = new JsonWebSignature();
56          jws.setPayload(claims.toJson());
57          jws.setKey(privateKey);
58          jws.setKeyIdHeaderValue(kid);
59          jws.setHeader("typ", "JWT");
60          jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.RSA_USING_SHA256);
61
62          return jws.getCompactSerialization();
63      }
```

It works!

```
curl -X POST -H 'Content-Type: application/json' -i http://localhost:8080/users --data '{"username": "leonschloemmer","password": "hello"}'
```

- This should now return a JWT we can use to access a secure endpoint!

The secured resource Project

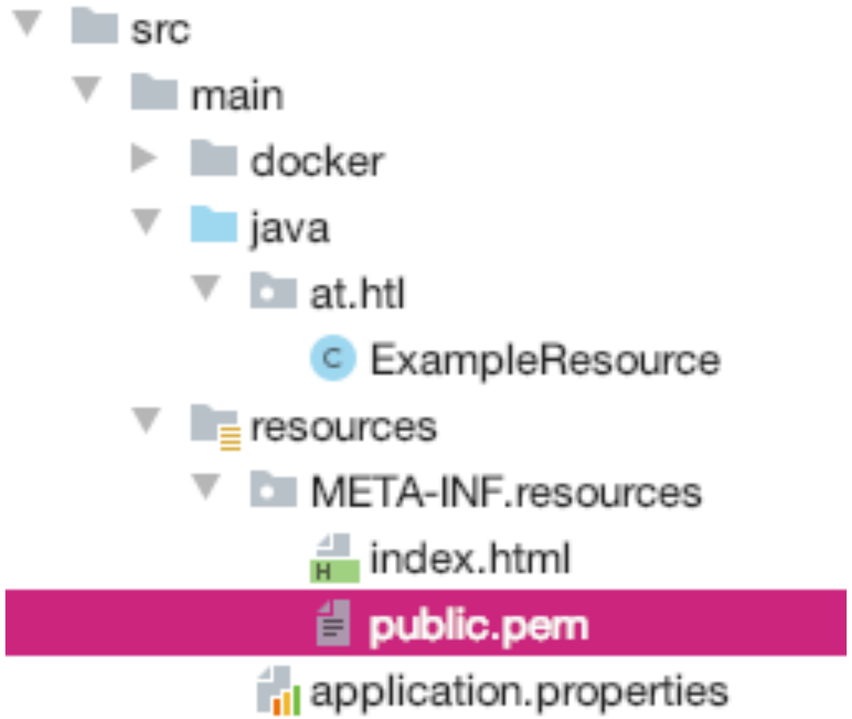
- First we need to add some configuration in application.properties

```
4  mp.jwt.verify.publickey.location=META-INF/resources/public.pem   1
5  mp.jwt.verify.issuer=https://this-is-totally-our.domain          2
6  quarkus.smallrye-jwt.auth-mechanism=MP-JWT
7  quarkus.smallrye-jwt.enabled=true
```

1. The location of the public key for decrypting our JWT

2. To verify that the issuer of the JWT is the one we want

Add public key to second project



Securing our endpoint

```
14      @GET
15      @Produces(MediaType.TEXT_PLAIN)
16      @RolesAllowed({"defaultUsers"})
17      public String hello() { return "hello"; }
```

- Now only people sending a JWT claiming to be of role *defaultUser* can access our very secret hello message.

Testing it out

- In the header of our request send:
- Authorization: Bearer <your_token>

```
curl -X GET -H 'Authorization: Bearer
eyJraWQiOiIvcHJpdmF0ZS5wZW0iLCJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL3RoaXMtaXMtdG90YWxseS1vdXIuZG9tYWluIiwic3ViIjoiMSIsImdyb3VwcyI6WyJkZWZhdWx0VXNlcnMiXSwiaWF0IjoxNTc1OTM0NzQw
LCJhdXRoX3RpbWUiOiJOdW1lcmljRGF0ZXsxNTc1OTM0NzQwIC0-IERlYyAxMCwgMjAxOSwgMTI6Mzk6MDAgQU0gQ0VUfSIsImV4cCI6MTU3NTkzNTY0MH0.SueNOBFEQ89HPCfHTAYXCEuKaw8crkNPLRQgePD7E-Dkhmq-
S9Lkf7NuOcvDfIq2APdHec3EUFqHQhsRIskgNXOjP7pyQCgHQ-liY5jxEEwy6NlRIyqzCDanx5MdYeeiAlVf9RsQ6yZH7KFYzKoLkeAVRtSk5cwRdRpVn_X1nbT2oq301Yu4pEW_Rx8SOOol7697LD4brHSkIX2sM3hnIfYGcNHslK2XtXcVUpFpUzv1segd-
s-4zIRW7cqQfZF-F69t8QaPUhuAgvCgVxIJ23juAeKiYzlQtZgGFB1nWjIUyVeWzYbTRjcJwACvYRSumc1Z8sZxwfvm4mh0GI2YAw' -i http://localhost:8181/hello
```

Status Code : 200 OK

content-length : 5

content-type : text/plain;charset=UTF-8

Running with docker

- This couldn’t be easier!
- just run `./mvnw package -Pnative -Dquarkus.native.container-runtime=docker`
  - This builds a linux native executable to put inside of our image
- to actually build the image from the Dockerfile run `docker build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .`
- `docker container run -i --rm -p 8080:8080 \ quarkus-quickstart/getting-started`

Dockerfile

```
...
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```

- In the *quarkus-livecoding-resource* project change the EXPOSE command to 8081